X-Cipher: Achieving Data Resiliency in Homomorphic Ciphertexts

Adam Caulfield, Nabiha Raza, and Pezhao Hu

Rochester Institute of Technology, Rochester, NY, USA
{ac7717,nr6024,hxpvcs}@rit.edu

Abstract. Homomorphic encryption (HE) allows for computations over ciphertexts while they are encrypted. Because of this, HE supports the outsourcing of computation on private data. Due to the additional risks caused by data outsourcing, the ability to recover from losses is essential, but doing so on data encrypted under an HE scheme introduces additional challenges for recovery and usability. This work introduces X-Cipher, which aims to make HE ciphertexts resilient by ensuring they are private and recoverable simultaneously at all stages during data outsourcing. X-Cipher allows data recovery without requiring the decryption of HE ciphertexts and maintains its ability to recover and keep data private when a cluster server has been compromised. X-Cipher allows for reduced ciphertext storage overhead by introducing novel encoding and leveraging previously introduced ciphertext packing. X-Cipher's capabilities were evaluated on a synthetic dataset to demonstrate that X-Cipher enables secure availability capabilities while enabling privacy-preserving outsourced computations.

Keywords: Homomorphic Encryption · Data Recovery · Applied Cryptography.

1 Introduction

Outsourcing data or computations to the Cloud has become a trend for the private sector, governments, and non-profits. In 2020, corporations spent 32% of their IT budget on Cloud services [15]. Despite this, privacy concerns have limited use of Cloud resources to operate on sensitive (e.g., health and financial data). To address this concern, Homomorphic Encryption (HE) can produce ciphertexts that can undergo computations without decryption. After Gentry's initial work, several schemes were introduced to support partially homomorphic schemes (limited additions or multiplications) to fully homomorphic schemes (addition and multiplication).

In addition to privacy concerns when outsourcing data, there is also a concern about data corruption and loss. Data replication was widely exercised for fault tolerance in distributed data storage systems [12], but the cost of maintaining exact replicas or data chunks dramatically increased. Today's Cloud systems have moved towards adapting erasure codes to reduce the storage overhead [8,14]. These systems typically apply the erasure codes over user-supplied

data. When outsourcing homomorphically encrypted ciphertexts, the erasure codes are produced at the ciphertext level. This introduces two new problems. First, HE ciphertexts are malleable by design [3]; thus, codewords generated through conventional methods must be updated after any computation – even if the underlying plaintext data is unchanged. Secondly, the overhead induced by erasure codes is proportional to the size of the input data. Due to the ciphertext expansion in fully homomorphic schemes, applying erasure codes over these ciphertexts will significantly increase the size of the generated codewords, increasing both the storage and operational overhead.

To combat this problem, one approach is to make use of *Encrypt-with-Redundancy (EwR)* [1], which combines an encryption scheme with erasure codes. However, existing solutions have drawbacks when applying EwR with HE ciphertexts. For instance, some prior work either assumes the ciphertexts are stored and not operated on [26], or the system requires only partial homomorphic operations [28,20,6]. Although partial homomorphic encryption schemes (such as Paillier [21]) are effective for systems tasked with data aggregation or systems equipped with modular multiplication ALUs [28], they are not applicable for a variety of cloud computing operations. In addition, they do not support optimizations available in fully homomorphic schemes (discussed further in Sec. 2.1).

To address these challenges, we propose X-Cipher: a system to enable recovery of homomorphic ciphertexts without requiring decryption. X-Cipher leverages an erasure code called X-Code [17] to generate codewords, which can be subsequently used to recover from data losses. Furthermore, X-Cipher employs encoding and packing techniques to reduce the storage overhead of storing erasure codes. To our knowledge, no previous works have leveraged erasure codes and HE in the same manner as X-Cipher to provide these security guarantees together while minimizing storage overhead and maintaining recoverability and privacy guarantees throughout desired operations on a cloud server.

2 Background

2.1 Homomorphic Encryption

Homomorphic Encryption (HE) describes a class of encryption schemes that enables computations over ciphertexts. Computation under HE produces a ciphertext that decrypts to the same result obtained when operating on the plaintext inputs. Modern HE schemes are classified as partially homomorphic (supports one of addition or multiplication), somewhat homomorphic (supporting addition and a limited number of multiplications), and fully homomorphic (supporting addition and multiplications). Our proposed solutions are designed on the BGV HE scheme [5], a fully homomorphic encryption scheme bases its security on the Ring-LWE (*Learning-with-Error*) problem [19]:

Ring-LWE: *Given a modulus q, assume elements a and s are obtained by sampling polynomial ring $R_q = \mathcal{Z}_q[x]/(\Phi(X))$. Assume noise e is sampled from a*

Gaussian distribution (χ). Given (a, s, e) and b obtained by $b = as + e$, the value b is computationally indistinguishable from elements randomly sampled from R_q.

Given the underlying hardness from Ring-LWE, the following set of probabilistic-polynomial-time BGV algorithms are significant to X-Cipher:

- $\texttt{BGV.KeyGen}() \rightarrow (\mathsf{pk}, \mathsf{sk})$: Assume plaintext modulus t and ciphertext modulus q establish ppolynomial rings $R_t = \mathbb{Z}_t[x]/(\Phi(x))$ and $R_q = \mathbb{Z}_q[x]/(\Phi(x))$, respectively. Uniformly sample $a \leftarrow R_q$, $s \leftarrow R_q$, and $e \leftarrow \chi$. Given these elements, set the secret key as $\mathsf{sk} = s$ and the public key as $\mathsf{pk} = (b, a)$ with $b = -as + te \pmod q$ according to the Ring-LWE assumption.
- $\texttt{BGV.Enc}(\mathsf{pk}, m) \rightarrow c$: Given a plaintext message $m \in R_t$, a public key $\mathsf{pk} = (b, a)$, uniformly sample a random $r \leftarrow \chi$ and errors $e_0, e_1 \leftarrow \chi$. Encrypt the message m as $c = (c_0, c_1) \in R_q^2$ where $c_0 = rb + m + te_0$ and $c_1 = ra + te_1$.
- $\texttt{BGV.Dec}(\mathsf{sk}, c) \rightarrow m$: Given a ciphertext $c = (c_0, c_1) \in R_q^2$ sk, set $s' = (1, sk) \in R_q^2$ and decrypt by computing $m = (\langle c, s' \rangle \mod q) \mod t$.
- $\texttt{BGV.Add}(c, c') \rightarrow c_{add}$: Adding two ciphertexts $c = (c_0, c_1)$, $c' = (c'_0, c'_1)$ results in $c_{add} = (c_0 + c'_0, c_1 + c'_1) \in R_q^2$.
- $\texttt{BGV.Mult}(c, c') \rightarrow \tilde{c}_{mult}$: Given two ciphertexts c and $c' \in R_q^2$, their homomorphic multiplication yields an extended ciphertext $\tilde{c}_{mult} = (\tilde{c_0}, \tilde{c_1}, \tilde{c_2})$ that is encrypted under the element s^2.
- $\texttt{BGV.Relinearize}(\tilde{c}_{mult}) \rightarrow c_{mult}$: Given a long ciphertext $\tilde{c}_{mult}$ obtained from multiplication, perform a relinearization step to change ciphertext from an encryption of s^2 to s. Due to the lack of space, we refer readers to [5] for further details of this algorithm.

In general, homomorphically encrypted ciphertexts are large due to complex lattice elements. Due to this significant ciphertext expansion, HE has been considered impractical for many applications. To address this problem, the ciphertext packing technique [4] was introduced. This allows for encoding multiple plaintext messages into one polynomial and encrypting a plaintext polynomial into a single ciphertext. Compared to individually encrypting each plaintext message for the same amount of plaintext values, ciphertext packing reduces the number of required ciphertexts. It also speeds up the homomorphic computations because operations are performed on plaintext values simultaneously in an element-wise and SIMD (Single-Instruction-Multiple-Data) manner [27]. X-Cipher utilizes ciphertext packing for these efficiency gains.

Given a vector of plaintext values $M = \{m_0, .., m_{k-1}\}; k \leq d$ and a polynomial ring R_t of degree d, ciphertext packing encodes all plaintext values into a single polynomial that is expanded from $\Phi(x)$ to its roots [27] using the Chinese Remainder Theorem (CRT). This method assumes $\Phi(x)$ of degree d can be factorized into exactly r polynomials of degree k; such that, $\Phi(x) \coloneqq \prod_{i=1}^{r} \Phi_i(x)$. Because $\Phi(x)$ and $\Phi_i(x)$ are *isomorphic*, operations performed on them achieve the same effect [27]. Given this property, each plaintext message $m_i \in M$ can be encoded into an arbitrary polynomial $f(x) \pmod{\Phi_i(x)}$. The plaintext capacity of these polynomials is referred to as the slot count (ϱ). With packing, a ϱ-length plaintext vector can be encrypted into a single ciphertext.

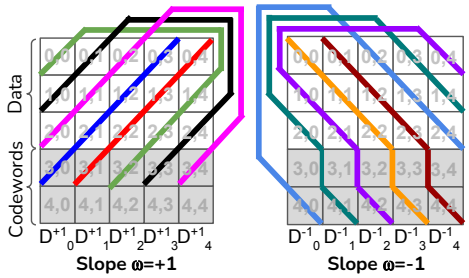

Fig. 1: An example of X-Code setup ($n = 5$).

The CRT-based ciphertext packing technique allows useful homomorphic operations on ciphertexts, such as element-wise addition/multiplication, shifting, and rotation. Element-wise addition and multiplication are somewhat straightforward, but shifting and rotation are not due to changing the order of data in the plaintext vector. Rotation and shifting is possible by modifying the polynomial [11]. Each slot in a packed ciphertext holds a polynomial $f(x) \pmod{\Phi_i(x)}$. Rotation and shifting can take place by modifying x with x^α for some exponent α. A new polynomial $f^{(\alpha)}(x) = f(x^\alpha) \pmod{\Phi_i(x)}$ will have all the same coefficients as $f(x)$ but at different slot locations; this technique is called *automorphism*. X-Cipher makes use of rotation and shifting for its recovery algorithm.

2.2 Erasure Codes

Erasure codes are used to achieve fault tolerance ability in systems [22]. As an alternate solution to data replication, erasure codes require less storage overhead. The codewords generated from an erasure code are less than the original data symbols in size and can be used with partial data to regenerate any lost data. Reed-Solomon codes [24] is a widely used code for detecting and correcting erasures in data storage [26]. The Reed-Solomon codes use Galois Field $GF(2^w)$ multiplication and maintain a Vandermonde matrix. This solution would drastically increase the computational overhead for homomorphically encrypted data that expands beyond the size of unencrypted plaintext. Erasure codes based on RAID-6 algorithms [23] provide dual-parity and require only the XOR operations. This style of code can be computationally efficient and easily implemented homomorphically. As demonstrated in Sec. 2.1, XOR for binary integers is simply addition, which is an efficient homomorphic operation.

Our work is based on X-Code [17], which can recover up to two full columns erasures in an $n \times n$ structure where n is a prime. Figure 1 shows an example of an X-Code setup with n=5. As illustrated, data symbols $d_{i,j}$ are organized into the first $n-2$ rows of the structure, and codewords $p_{i,j}$ make up the final two rows; where i is row, j is column. In X-Code, both data symbols and codewords are in $\mathbb{Z}_2$. Codewords in the last two rows are generated by XOR-ing (or homomorphically adding) data symbols along the diagonal that connects them, as illustrated in Fig. 1. These diagonals have slopes ± 1 and cross each other when overlayed,

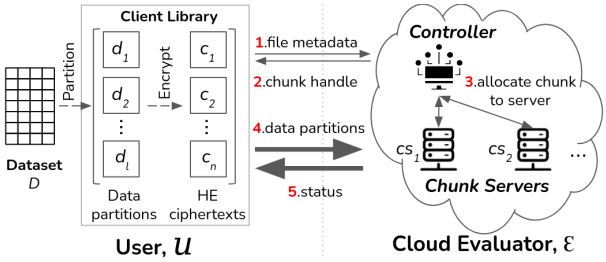

Fig. 2: System Model: User ($\mathcal{U}$) outsources its encrypted dataset to a Cloud Evaluator ($\mathcal{E}$) for computations.

hence the name X-Code. For diagonals with slope=+1, each codeword $p_{n-2,j}$ in the $(n-2)$-th row is computed by $p_{n-2,j} = \sum_{k=0}^{n-3} d_{k,(j-k-2\%n)}$. For the example in Fig. 1, we calculate the first codeword in the first row as $p_{3,0} = d_{2,1} \oplus d_{1,2} \oplus d_{0,3}$. Likewise, each codeword along the diagonals with slope=-1, $p_{n-1,j}$ in the $(n-1)$-th row is computed by $p_{n-1,j} = \sum_{k=0}^{n-3} d_{k,(j+k+2\%n)}$. In X-Code, each data symbol is cross-checked by *exactly* two codewords. Hence, it can recover up to two full-column losses.

2.3 System and Threat Model

System Model: X-Cipher assumes a data and computation outsourcing setup, as illustrated in Fig. 2 in which a user $\mathcal{U}$ wants to delegate a private dataset D to a cloud evaluator $\mathcal{E}$ for outsourced computations (e.g, document analysis, machine learning). Modern distributed filesystems [12,8] typically allow users to divide the dataset into multiple partitions, $D = \{\boldsymbol{d_1}, .., \boldsymbol{d_\ell}\}$ and store these data partitions into a set of available chunk servers assigned by the master, as shown in Fig. 2. The user homomorphically encrypts these data partitions: given a public key pk, $c_j = Enc(\mathsf{pk}, d_j); d_j \in \boldsymbol{d_i}$, before sending them to the cloud evaluator. Clients interact with the Controller for metadata operations but with the chunk servers directly for data operations [12]. Leveraging the capability of homomorphic encryption, the cloud evaluator will perform the requested computations while data stays encrypted. X-Cipher proposes methods to recover from losses of homomorphically encrypted ciphertext. In addition, we design example homomorphic algorithms to demonstrate how computing tasks can be securely evaluated while maintaining recoverability.

Threat Model: In this work, we consider an adversary ($\mathcal{A}$) who is primarily motivated to affect the operational capacity of a chunk server. Therefore, they take actions to shut down the server or make it fail, leading to data loss. If $\mathcal{A}$ cannot cause the server to fail, they act in a semi-honest manner until they can cause the server to fail. Therefore, while the server is operating, $\mathcal{A}$ aims to leak information about the data while following the protocol specification. They can only learn about the data by observing stages of the computation (as input,

Table 1: Notation Table

Symbol	Description	Symbol	Description
$\mathcal{U}, \mathcal{E}$	User and Cloud Evaluator	σ	Codewords of erasure codes
a, $\boldsymbol{a}$, A, A_i, $\boldsymbol{a_i}$	Element, vector, matrix, submatrix, i-th column	n	Dimension of each X-Code block; prime, $n \times n$
$c_i \in \boldsymbol{c}$	The i-th ciphertext in a collection $\boldsymbol{c}$	m	Multiples of X-Code blocks
$(\mathsf{pk}, \mathsf{sk})$	Public and secret keys for HE	ω	Slope in X-Code, where $\omega = \pm 1$
$[\cdot]$	HE encrypted data; or as c interchangeably	θ	Number of ciphertext rotation
ϱ	Plaintext slot count	$\boldsymbol{\Xi}$	A vector acting as mask; $\boldsymbol{\Xi} \in \{0,1\}^{\ell}$

intermediate results, and outputs). With the information it observes, $\mathcal{A}$ aims to leak data or learn the secret key (sk).

3 X-Cipher

Figure 3 shows an overview of the X-Cipher phases, and common notation used for the remaining sections are shown in Table 1. At the base of our X-Cipher design is X-Code, as introduced in Sec. 2.2. The original X-Code scheme [17] was designed for storing bits; thus, the codewords were calculated with simple XOR-ing. X-Cipher design can inherit this setting (arithmetic modulo prime 2), and it generalizes the scheme to support integer arithmetic modulo a prime p. This change from $\mathbb{Z}_2$ to $\mathbb{Z}_p$ allows our X-Cipher to store more information per data cell and allows for a more general solution that can be applied to various applications. Quite simply, X-Cipher uses of addition and subtraction to replace XOR-ing when calculating the codewords and recovering from loss, respectively. These arithmetic operations map to homomorphic addition and subtraction over ciphertexts for efficient evaluation and allow codeword generation and data recovery to stay the same. For all codewords along the slope $\omega = +1$ as example, they can still be computed by $p_{n-2,j} = \sum_{k=0}^{n-3} d_{k,(j-k-2 \pmod n)}$. Recovering any lost data in the general setting is achieved by subtraction. For instance, recovering $d_{r,c}$ connected to $p_{n-2,j}$ is completed by $d_{r,c} = p_{n-2,j} - \sum_{k=0,k\neq r}^{n-2} d_{k,(j-k-2 \pmod n)}$. Another important characteristic to be maintained is the maximum distance separable (MDS) property of X-Code. Xu et al. [17] presents an extensive proof of how the original X-Code achieves the MDS property. The MDS property is dependent upon the dimension of the array n being a prime number and is independent of the data within the array. Because of this, the integer space changing from $\mathbb{Z}_2$ to $\mathbb{Z}_p$ does not affect the MDS property.

Once the X-Cipher structure is instantiated, the first $n-2$ rows are filled with plaintext data and the last two rows are filled with codewords. Then, the $n \times n$ structure is partitioned into columns $\boldsymbol{d_i}; i = (0, .., n-1)$, encrypt each column into a ciphertext $c_i = Enc(pk, \boldsymbol{d_i})$ using the packing techniques described in Sec. 2.1. Then, each packed and encrypted column c_i is distributed to chunk servers so that each chunk server cs_i stores ciphertext c_i, as illustrated in Fig. 2.

Packing n elements along the column into one ciphertext improves space efficiency and allows element-wise operations to be carried out in a SIMD manner. However, there are two observations regarding spatial efficiency. First, the ratio of codewords to data is $\frac{2}{n-2}$ which decreases as n increases. Because the X-Code

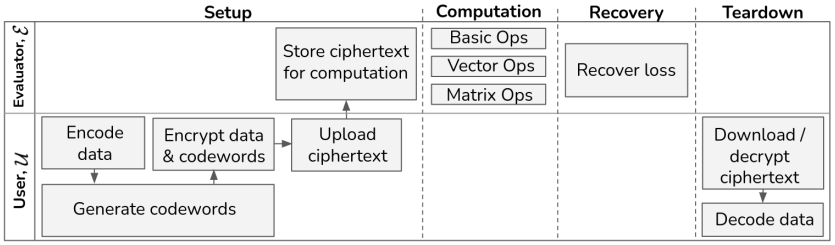

Fig. 3: Tasks by $\mathcal{U}$ and $\mathcal{E}$ during each phase with X-Cipher

structure is a square, increasing the value of n increases the data rows (packed ciphertext slots) and the number of columns (ciphertexts). Consequently, representing the data in an unmodified X-Code structure will require more ciphertexts; thus, more servers will be needed to store the distributed ciphertexts as the total data increases. This can cause a problem because more data pieces distributed over a network not only increases the storage cost but also incurs high data movement overheads when recovery and integrity checking occurs. Secondly, if n is small, additional chunk servers and extra ciphertexts no longer support the X-Code structure; consequently, only n of the plaintext slots within the packed ciphertext are utilized. Ideally, the number of plaintext slots should be maximized for efficiency. In this scenario, the number of plaintext slots will likely be significantly underutilized since there are typically thousands of slots for properly selected set of security parameters.

Given these observations, X-Cipher's internal structure comprises one larger vertical rectangle structure abstracted as m copies of the $n \times n$ X-Code structures stacking on each other, as illustrated in Fig. 4. Each of these $n \times n$ X-Code structures can store different data and codewords. This design allows us to reduce the codewords-to-data ratio and to increase the utilization of the plaintext slots in the packed ciphertext without increasing the number of chunk servers. In X-Cipher, the value of n is a fixed value and depends on the number of chunk servers to be used for storing the packed ciphertext c_i. The value of m is determined by the total number of plaintext slots dividing n.

3.1 Setup Phase:

To initiate X-Cipher, the user instantiates the BGV HE scheme by selecting the initial parameters. Firstly, the plaintext modulus t is selected, and it should be greater than the maximum individual value in the dataset. Next, the plaintext slot count ϱ is calculated, which determines the number of plaintext integers packed into a single ciphertext. These parameters are then used to generate the keys.

Next, the value of n must be selected. In X-Cipher, the upper limit of n is based on the total chunk servers that are available/required. There are some additional constraints on n based on the data structure. First, X-Code requires that n is both a prime number and $n > 3$ [17]. Next, the slot count ϱ and

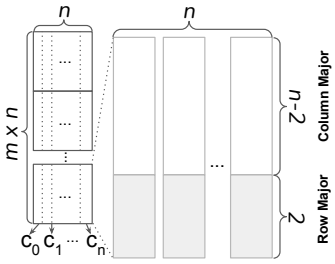

Fig. 4: X-Cipher structure.

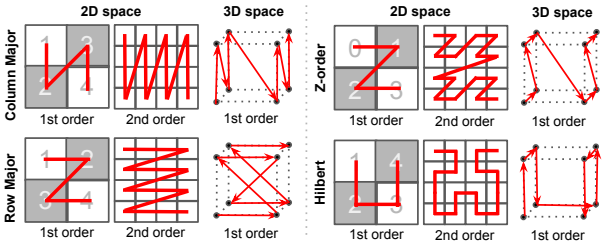

Fig. 5: Dimension reduction techniques.

n determine the number of copies m of the $n \times n$ X-Code structures that are required; that is $m = \frac{\varrho}{n}$. A matrix D is split it into a set of column vectors $\boldsymbol{d_i}$ for distributed storage and parallel tasks. Hence, the user must select the smallest n such that $||\boldsymbol{d_i}|| \geq m \cdot n$. The dimensions of X-Cipher are $(m \times n, n)$, as illustrated in Fig. 4.

The $\mathcal{U}$ then encodes the data into the structure. For multidimensional data, the data is flattened by applying a dimension reduction technique that is most suitable for the intended computations. Figure 5 illustrates four different dimension reduction techniques (also referred to as space-filling curves [10]). Users can use any dimension-reduction technique to convert their data into vectors. After reducing the dimension of the input data into a set of one-dimensional vectors, X-Cipher treats the data as a stream, and the data is encoded into the X-Cipher structure in a column-major fashion. Data is filled into every $n - 2$ row within each of the $n \times n$ X-Code blocks, skipping the last two rows of each block since they are reserved for codewords computed based on the filled data.

Finally, the n columns of partitioned data vectors are encrypted, and the ciphertexts $\{c_i\}$ are uploaded to $\mathcal{E}$, which stores them among its n chunk servers. After this, each chunk server is ready to perform homomorphic computations.

3.2 Recovery Phase:

The X-Cipher structure is reassembled before the recovery starts by collecting the distributed ciphertexts from the remaining online chunk servers. Recovery in X-Cipher can be performed similarly to X-Code since the data in the underlying column structure is preserved in the packed ciphertexts. As illustrated in Fig. 1, recovering lost data requires computation on data along the same slope line. Since homomorphic ciphertexts operate on their underlying vectors using component-wise operations, transformation is required before the recovery operations to align data associated with the same codewords within each n ciphertexts. To complete recovery, data aligned diagonally in the plaintext structure must be aligned horizontally in ciphertext (i.e., aligned in the same plaintext slot). This alignment is achieved by performing homomorphic rotation on ciphertexts.

Associated data and codewords are illustrated in Fig. 6 (a)(b) with the same colored diagonal line. In this Figure, the last two rows are for codewords; each

Algorithm 1: Column rotation, `RotCols`($\boldsymbol{c}$, ω).

Input : $\boldsymbol{c} \coloneqq \{c_0, \ldots c_n\}$, $\omega \coloneqq \{+1, -1\}$
Output: $\boldsymbol{c} \coloneqq \{c_0^{\omega}, \ldots c_n^{\omega}\}$
for $i \in [0, n)$ **do**
  if $\omega = +1$ **then**
    $\theta_{up} = n - 1 - i$ ▷ Calculate rotation count;
  else if $\omega = -1$ **then**
    $\theta_{up} = i$
  $\theta_{down} = n - \theta_{up}$ ▷ Calculate rotation count for the other half;
  $\boldsymbol{\Xi}_{top} = \{0,1\}^{m \times n}$, $\boldsymbol{\Xi}_{top}[i] = 1; i \pmod n < \theta_{up}$,
  $\boldsymbol{\Xi}_{bottom} = \{0,1\}^{m \times n}$, $\boldsymbol{\Xi}_{bottom}[i] = 1; i \pmod n \geq \theta_{up}$,
  $c_{bottom}^{\omega} = \texttt{rotate}(c_i \cdot \boldsymbol{\Xi}_{bottom}, -1 \cdot \theta_{up})$
  $c_{top}^{\omega} = \texttt{rotate}(c_i \cdot \boldsymbol{\Xi}_{top}, \theta_{down})$
  $c_i = c_{top}^{\omega} + c_{bottom}^{\omega}$ ▷ Reassemble the ciphertext;
end

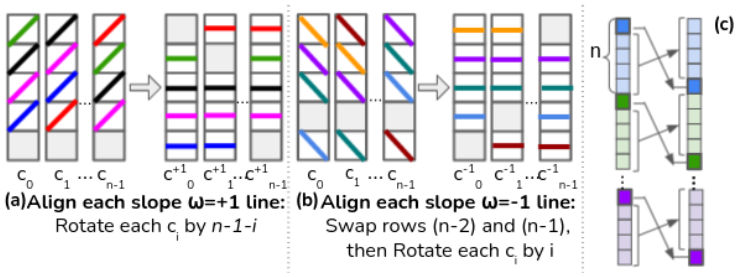

Fig. 6: Column rotation aligns sloped (colored) lines.

subfigure corresponds to one of the two slopes ((a) $\omega = +1$, (b) $\omega = -1$). The effect of rotation is demonstrated in both in Fig. 6(a)(b). The BGV rotation action (`rotate`) moves data within the ciphertext along its plaintext slots. One left rotation moves the first data element to the last slot. Hence, directly rotating to ciphertexts will be incorrect: the X-Code block boundaries will be violated.

To address this problem, the X-Cipher algorithm `RotCols` rotates columns for data recovery. Algorithm 1 shows steps of this function, which complete extraction by masking and swapping. The effect of `RotCols` on one column is demonstrated in Fig. 6 (c). The algorithm extracts the values to be rotated (highlighted with solid color) by applying a mask $\boldsymbol{\Xi}_{top}$ and the rest of the values by another mask $\boldsymbol{\Xi}_{bottom}$, and then rotate them into position by calculating c_{top}^{ω} =`rotate`$(c_i \cdot \boldsymbol{\Xi}_{top}, -1 \cdot \theta_{up})$ where $\theta_{up} = 1$ and c_{bottom}^{ω} =`rotate`$(c_i \cdot \boldsymbol{\Xi}_{bottom}, \theta_{down})$ where $\theta_{down} = n - 1$. Both $\boldsymbol{\Xi}_{top}$ and $\boldsymbol{\Xi}_{bottom}$ are plaintext vectors that have the dimension ρ and have values set to 1 at the appropriate indexes. Finally, the two ciphertexts are combined $c_i = c_{top}^{\omega} + c_{bottom}^{\omega}$ to produce the rotated column.

X-Cipher can recover up to two failed chunk servers (two columns). For illustration, Alg. 2 shows steps to recover from losing one column. To recover the lost column, the column rotation (Alg. 1) transforms the input according to two slope configurations, as shown in Fig. 6. Next, recovery computations take place

Algorithm 2: One-Column Recovery.

Input : Lost column index idx, remaining columns $\boldsymbol{c} = \{c_i\}^{n-1}$ for $i \in [0, n)$ s.t. $i \neq idx$
Output: Recovered Column c_{idx}
for $i \in [0, n), i \neq idx$ **do**
  for $j \in [0, m \times n)$ **do**
    $\boldsymbol{u}[j] \leftarrow 1$ if $c_{id}[j]$ is σ OR if $c_{id,j}$ is data & $c_{i,j}$ is $\sigma_{\omega=+1}$
    $\boldsymbol{u}[j] \leftarrow 0$ if $c_{i,j}$ is $\sigma_{\omega=-1}$
    $\boldsymbol{u}[j] \leftarrow -1$ if $c_{id,j}$ is data & $c_{i,j}$ is $\sigma_{\omega=+1}$
  end
  c_{id} += $\boldsymbol{u} \cdot c_i$
end
`RotCols`$(\boldsymbol{c}, \omega = 1, d = 1)$ `RotCols`$(\boldsymbol{c}, \omega = -1, d = -1)$
`RotCols`$(c_{id}, \omega = 1, d = 1)$ `RotCols`$(c_{id}, \omega = -1)$
for $i \in [0, n), i \neq id$ **do**
  for $j \in [0, m \times n)$ **do**
    $\boldsymbol{u}[j] \leftarrow 1$ if $c_{id,j}$ is $\sigma_{\omega=-1}$
    $\boldsymbol{u}[j] \leftarrow 0$ if else
  end
  c_{id} += $\boldsymbol{u} \cdot c_i$
end

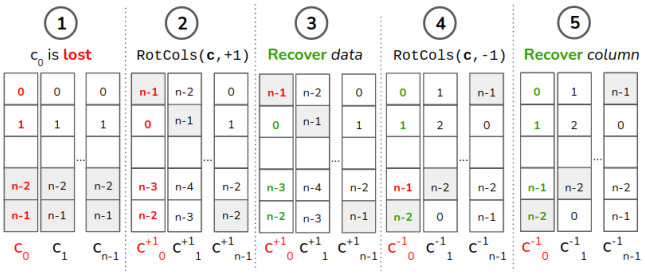

Fig. 7: Illustration of one column recovery.

on the two resulting structures. The shaded cells contain codewords calculated by the other slope line configuration and are not in use. For example, when slope $\omega = +1$, the shaded cells refer to the $\omega = -1$ configuration codewords; thus, they are ignored since they are unused by the $\omega = +1$ configuration.

To elaborate on the recovery process, assume column (ciphertext) c_0 is lost, as illustrated by Fig. 7 (Step 1). Column rotation for slope $\omega = +1$ occurs and results in Step 2. Then $n - 1$ cells are recovered. A lost data cell is recovered by subtracting the remaining data from the codeword. This can be made efficient by constructing a mask u for each column and homomorphically accumulating the masked columns. The correct sign in the bit mask is based on what is required for the cell: 1 to add a data cell, -1 to subtract a codeword, or 0 to ignore. All rows can be updated simultaneously with a vector-wise addition, as shown in (Step 3).

To recover the last cell $c_{0,n-1}$, a column rotation si performed for slope $\omega = -1$ to get a transformed structure shown in (Step 4). The same masking and accumulation process is performed to recover the final cell. After computing the final sum in Step 5, the lost row has been successfully recovered. Note this illustration shows the recovery algorithm using one of the m X-Code blocks.

Since the X-Cipher structure is organized as m X-Code blocks stacked on top of each other , the recovery algorithm acts on all m X-Code blocks.

Two-column recovery is executed in a similar manner to the one-column recovery. Due to space limitations, the algorithm specification is omitted from this paper. The first five steps are identically similar to the single-column recovery algorithm. As a result, the two-column recovery is implemented as multiple iterations while masking the expected rows. As long as the column numbers are known the order of recovery is known. Then, the order of masks is determined at run-time based on the lost columns' IDs and the iteration.

3.3 Computation Phase

This section presents how commonly used operations over one or two datasets as building blocks can occur over ciphertexts in X-Cipher.

Over One Dataset. The total sum of all elements in the structure can be computed by naively adding all elements sequentially. But in X-Cipher, there is a more efficient way. As discussed in Sec. 2.2, each codeword contains the sum of all data elements along its respective slope line. As a result, the overall sum in the structure can be computed by adding all codewords along one of the slopes, followed by summing the resulting codewords from each X-Code block. Alg. 3 presents this idea, which is efficient for using component-wise homomorphic addition over the packed ciphertexts. It achieves $O(log_2 m)$ complexity by simultaneously adding values across multiple X-Code blocks. The final result c_{sum} is a ciphertext that encrypts a vector containing the sum in both the $(n-2)th$ and $(n-1)th$ slots. The result is recoverable by redistributing the resulting ciphertext c_{sum}. The codewords are additively homomorphic, so a total sum avoids the need to regenerate the codewords. In the final result, the first $n-2$ rows contain partial sums of the dataset. In other words, the first $n-2$ rows add up to the codewords in the final two rows. In order to construct a recoverable structure, each slope requires elements in the first $n-2$ rows that add up to the codeword. Therefore, this requirement is met by simply creating n duplicates of c_{sum}, and a valid X-Cipher structure is created.

Over Two Datasets. Component-wise operations over two datasets can be supported by operating on the corresponding ciphertexts. In addition, dot-product is supported with these basic component-wise operations. Given two vectors encoded and encrypted into their corresponding ciphertexts c^A and c^B, the dot-product is $c_i^{prod} = c_i^A \times c_i^B$ for $i = (0, .., n-1)$. Then,the sum over all c_i^{prod} is determined using Alg. 3.

A good example of demonstrating operations over vectors is the private set intersection (PSI) problem. Given two sets $\boldsymbol{x} \in \mathbb{R}^{\iota}, \boldsymbol{y} \in \mathbb{R}^{\zeta}$ which have lengths ι and ζ respectively and typically $\iota >> \zeta$, PSI determines the common elements in these two sets, $\boldsymbol{x} \bigcap \boldsymbol{y}$, for all set elements $x_j \in \boldsymbol{x}$ and $y_i \in \boldsymbol{y}$. This protocol occurs between a sender $\mathcal{U}_{\mathcal{S}}$ who owns $\boldsymbol{x}$ and a receiver $\mathcal{U}_{\mathcal{R}}$ who owns $\boldsymbol{y}$. Using X-Cipher, the users can generate an encrypted, packed, and recoverable PSI result.

Algorithm 3: Summation with $O(log_2 m)$ complexity.

Input : Encrypted columns, $\boldsymbol{c} \coloneqq \{c_0, ...c_{n-1}\}$
Output: Sum of all data elements, c_{sum}
$mask = \{0,1\}^{m \times n}; mask_i = 1; i < n; mask_i = 0; i \geq n$
$c_{sum} = \sum_{i=0}^{n} c_i;$ ▷ Component-wise addition of encrypted columns.
if $m\%2 \neq 0$ **then**
  $m = m --$
end
for $\alpha = 2^j; \alpha <= m/2; j = \{0, .., log_2(m) - 1\}$ **do**
  $c'_{sum} =$`shift`$(c_{sum}, -\alpha \cdot n)$ ▷ Move by block, padding by 0.
  $c_{sum} \mathrel{+}= c'_{sum}$
end
if $m\%2 \neq 0$ **then**
  $c'_{sum} =$ `shift`$(c_{sum}, -(m+1) \cdot n)$
  $c_{sum} \mathrel{+}= c'_{sum}$ ▷ Add the last element.
end
$c_{sum} = c_{sum}$.`multByConst`$(mask)$ ▷ Mask out unwanted data.

A fast PSI protocol using HE [7] defines a basic protocol in which $\mathcal{U}_\mathcal{R}$ sends $[y_i] = Enc(pk, y_i)$ to $\mathcal{U}_\mathcal{S}$ and $\mathcal{U}_\mathcal{S}$ computes and returns $\hat{c}_i = r_i \cdot \prod_j([x_j] - [y_i])$, expecting an encryption of zero when there exists $x_j = y_i$ for any j or a random value masked by the uniformly sampled random number r_i. Since homomorphic subtraction is required for all x_j with y_i, these values are packed X-Cipher ciphertexts. Once packed, these operations are performed simultaneously in a SIMD manner.

PSI can be further optimized as proposed in [7]. Expanding the basic PSI definition results in the following: $\hat{c}_i = r_i \cdot \prod_j([x_j] - [y_i]) = r_i \cdot (x_1 - y_i) \cdot ... \cdot (x_\iota - y_i) = r_i y_i^\iota + r a_{\iota-1} y_i^{\iota-1} + ... + r a_0$ for some combinations of x_j storing in a_i. This shows the PSI is the sum of elements which are the product of 1) the $\mathcal{U}_\mathcal{R}$'s data raised to some degree $(y_i^\iota, y_i^{\iota-1}, ..., 1)$ and 2) coefficients $(r_i, r_i a_{\iota-1}, ..., a_0)$ which depend only on the $\mathcal{U}_\mathcal{S}$'s data. Assembling these elements into two vectors $\boldsymbol{y'_i} = \{y_i^\iota, y_i^{\iota-1}, ..., 1\}$ and $\boldsymbol{a} = \{r_i, r_i a_{\iota-1}, ..., a_0\}$, the PSI can be evaluated by simply computing dot-product: $\boldsymbol{y'_i} \cdot \boldsymbol{a}$. When encoding and packing data into the X-Cipher, the optimized PSI protocol can be completed efficiently using the dot-product algorithm discussed earlier. Of course, data packed into our X-Cipher is recoverable.

Additionally, matrix addition and multiplication are building blocks of many cloud-computing tasks for which X-Cipher may be desirable. By enabling matrix addition and multiplication on data in the structure, X-Cipher allows for data to remain encrypted and recoverable. A matrix can be encoded into the X-Cipher structure in many ways, as discussed in Sec. 3.1. For addition and multiplication, it may be most useful to traverse the matrix using a row-major curve to fill a single column. By using this encoding and fine-tuning the parameters n, m, it

can be ensured that a single ciphertext contains entire matrices, which can be most efficient for distributed tasks.

Once a matrix is encoded, matrix addition is completed through component-wise addition. As described in Sec. 2.1, operations on packed ciphertexts act in a component-wise manner; thus matrix addition is completed by simply executing a homomorphic addition of two ciphertexts which encode the matrices. For instance, given matrices A, B are encoded and encrypted into ciphertexts c_A, c_B respectively, the matrix addition $A + B$ is completed by $c_A \oplus c_B$.

Matrix multiplication on the other hand is not component-wise, and thus homomorphic multiplication of the ciphertexts encoding matrices cannot be directly used. Jiang et al. [16] proposed four different matrix permutations that lower the complexity of homomorphic matrix multiplication to just a depth of one. Given a matrix A and its data element at the i-th row and j-th column indicated as $a_{i,j} \in A$, the four permutations are *diagonal column rotation* $S(A)_{i,j} = a_{i,i+j}$, *diagonal row rotation* $T(A)_{i,j} = a_{i+j,j}$, *column rotation by k spaces* $\phi^k(A)_{i,j} = a_{i,j+k}$, and *row rotation by k spaces* $\gamma^k(A)_{i,j} = a_{i+k,j}$. Given two square matrices $A, B \in \mathbb{Z}^{d \times d}$, the matrix product can be expressed by the following sum: $\sum_{k=0}^{d-1} \phi^k(S(A)) \cdot (\gamma^k T(B))$. In their design transformations occur on the ciphertext in order to reduce the number of ciphertexts. Because our design assumes multiple ciphertexts to compose the internal structure, X-Cipher has the advantage to pre-computed the transformations on the plaintext and can store each transformed version required in the structure. Because of this approach, matrix multiplication is simply a matter of fetching the $2 \cdot d$ transformed matrices from the extended structure, and only performing homomorphic multiplication and addition as described in the previously expressed equation.

4 Evaluation

Space Complexity One of the main advantages of X-Cipher is its capability to significantly lower the overheads incurred when using erasure codes for recoverability. When using X-Cipher, user data is encoded into a $(m \times n, n)$ structure and each of the n columns is packed into a ciphertext c_i that has $\varrho \geq m \times n$ plaintext slots determining by the security parameter selection. As discussed in Sec. 3.1, this is to abstract the m multiples of the X-Code $n \times n$ block stacked vertically, so that each ciphertext is a column of the extended structure. Since each X-Code block can contain $n \times (n-2)$ plaintext data and $2n$ codewords, a single X-Cipher structure can store $m \times n \times (n-2)$ plaintext integers from the user as input. Using the ciphertext packing technique, the entire $m \times n$ data and codewords within a column fit into a ciphertext. Table 2 provides statistics to demonstrate the space efficiency of X-Cipher. For a given plaintext slot count $\varrho = 64$, we vary the dimension of each X-Code block $n = \{5, 7, 11, 13\}$ and the multiples of X-Code blocks $m = \{12, 9, 5, 4\}$ to evaluate the space complexity, with or without using X-Cipher. As demonstrated, an X-Cipher ciphertext can store the provided plaintext data and codewords without incurring a significant overhead. It is almost the same as the plaintext data size, but it is significantly

Table 2: Space complexity for X-Cipher structure

Parameters			**Size (KB)**		
Dimension (n)	Multiples (m)	Data Cells	Plaintext	Ciphertext (X-Cipher)	Ciphertext (without)
5	7	180	0.72	0.93	55.8
7	9	315	1.26	1.30	82.1
11	5	495	1.98	2.05	112.5
13	4	572	2.28	2.42	125.7

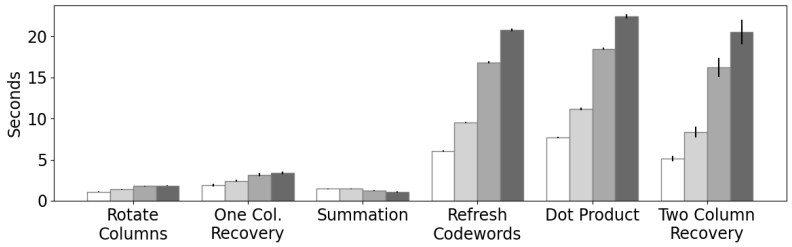

Fig. 8: Running time of primitive functions.

smaller than individually encrypting the provided plaintext data. Finally, our design is based decentralized erasure codes [9]. Storing ciphertexts on independent chunk servers enhances recoverability, load-balancing, and accessibility.

Computational Complexity The computational complexity of X-Cipher algorithms is based on the number of consecutive homomorphic multiplications; this is also called the *multiplicative depth*. After every homomorphic multiplication, noise reduction and linearization are required. The depth of `RotCols` is determined by: n to rotate one column, times n columns, times 2 rotations per recovery - $2 \times n \times n = 2n^2$. One-column recovery adds one additional rotation for the $\omega = +1$ and $\omega = -1$ states of the recovered structure to update each other as described in Sec. 3.2. This results in a total depth of $2n^2 + 1$. Similarly, two-column recovery has the same limitation to a factor of n, resulting in a total depth of $2n^2 + n$. However, because of the approach to use two separate states of the structure during recovery, the depth for both recovery algorithms is one factor of n lower than what they would be in an approach without the separate structure.

5 Experimental Results

We prototype the proposed X-Cipher structure using the HElib library [13], which implements the BGV HE scheme [5] (see Sec. 2.1). Based on this prototype, we conduct experiments multiple times on a CloudLab machine that has Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz. For parameter selection of the BGV scheme, we set the security parameter λ to 128 bits, which corresponds to a 3072-bit asymmetric key [2]. We choose the plaintext modulus $t = 131$ as a value large enough for our experiments. The rest of the BGV scheme parameters are set to the defaults [13]. The multiplicative depth L of is configured according to

Table 3: Application Computation Running Time Statistics

PSI (s)				**Matrix Operations (ms)**		
ι	$\zeta = \iota/8$	$\zeta = \iota/4$	$\zeta = \iota/2$	**Dim**	$+$	$\times$
25	28.1	54.4	109	**3**	24.05	278.5
50	54.5	109	225	**4**	52.25	364.6
75	81.3	162	332	**5**	50.50	450.8
100	111	224	444	**6**	54.20	541.5

what is required by the evaluated circuits in our protocol, for which we derived according to the complexity analysis discussed in Sec. 5.

Figure 8 shows the run-time for X-Cipher core primitive functions with $\varrho = 64$, $n = (5, 7, 11, 13)$, and $m = (12, 9, 5, 4)$. Results demonstrate the efficient execution of many of the primitives. For instance, all primitive functions whose times are in the top graph in Fig. 8 execute with tolerable run times for homomorphic operations. Functions such as codewords refreshing, dot product, and two-column recovery are the slower functions, and their running times grow as the dimension n grows. These functions rely most heavily on ciphertext rotations and multiplications. This current run-time minimizes data duplication and does not implement parallel execution. This is the trade-off that must be considered on implementation since performance improvements could be observed by duplicating data in order to leverage parallel computing.

An empirical study was conducted by executing the PSI of synthetic datasets encoded into an X-cipher structure. Datasets of varying sizes were used to test PSI. The encoding is subject to the slot count (ρ), the size of the Receiver's set (ζ), and the size of the Sender's set (ι). Execution times of the PSI are shown in Table 3 . The times reported represent the time for the Sender and Receiver to preprocess their data, encode into the X-Cipher structure, homomorphically compute the PSI, and return and decrypt the result. Given a constant slot count for experiments $\rho = 64$, the Sender's set was tested at values $\iota = (25, 50, 75, 100)$ and the values $\zeta = (\iota/8, \iota/4, \iota/2)$ were tested. The results demonstrate the execution time increases linearly as the Receiver or Sender set increases. This demonstrates that with additional parallelization, the execution time can be decreased further.

6 Security Analysis

Considering the adversary described in Sec. 2.3, we analyze the security of X-Cipher in two scenarios: before and after $\mathcal{A}$ compromises a chunk server (i.e., causes shutdown/failure). Before $\mathcal{A}$ causes the chunk server (cs_i) to fail, the setup and computation phases take place. During the setup phase, $\mathcal{U}$ is responsible for encoding the data, generating the codewords, and producing the ciphertexts. Because of this, $\mathcal{A}$ only sees ciphertexts after they are sent by $\mathcal{U}$. $\mathcal{A}$ controls cs_i and obtains a ciphertext c_i from $\mathcal{U}$. Given the use of BGV scheme in X-Cipher and a public key $pk = (a, b = -as + te)$, c_i is of the following form (see Sec. 2.1):

$$c_i = (rb + \boldsymbol{d_i} + te_0, ra + te_1)$$

$$c_i = (r(-as + te) + \boldsymbol{d_i} + te_0, ra + te_1)$$
$$c_i = (-ras + rte + \boldsymbol{d_i} + te_0, ra + te_1)$$

$\mathcal{A}$ learns a through knowing pk, but $r, e_0, e1, e$ are randomly sampled Gaussian values and thus unknown. Therefore, $\mathcal{A}$ can only learn $\boldsymbol{d_i}$ if it learns the polynomial modulus t or secret key $sk = s$. Since $\mathcal{A}$ cannot determine these values as provided by Ring-LWE indistinguishability, $\mathcal{A}$ cannot learn $\boldsymbol{d_i}$ from c_i in X-Cipher. During the computation phase, a homomorphic function f might operate over c_i and produce a known ciphertext $f(c_i)$. However, the same reasoning makes determining $f(d_i)$ impossible. Therefore, $\mathcal{A}$ cannot learn d_i by observing any homomorphic function f during the computation phase. As a result, $\mathcal{U}$ will always successfully recover c_i when lost.

Given $\mathcal{A}$ has already caused cs_i to go offline, c_i will be lost. However, $\mathcal{A}$ cannot prevent $\mathcal{E}$ from completing the X-Cipher recovery phase using the remaining chunk servers. If $\mathcal{A}$ causes the server to fail in the middle of some operations, it can resume from the point the codewords were last regenerated. In addition, assuming another cluster server cs_j has gone offline due to non-adversarial actions and c_j becomes lost, $\mathcal{A}$ cannot prevent the recovery of d_j even if it causes cs_i to lose c_i.

For this initial work, we do not consider the colluding adversaries or a malicious adversary that mutates c_i without shutting down cs_i. Thus, these opportunities open avenues for potential future work. For colluding adversaries, future work can include distributing multiple ciphertexts to each cluster server and new encoding algorithms to reduce the additional overhead of doing so. For a malicious adversary, future work is a method to track the operations that occurred as a part of the ciphertext. Then, a subsequent verification algorithm can be developed to detect malicious actions for future work. This verification could detect malicious actions by matching the component in the ciphertext that describes the list of operations to a polynomial representation of the expected operations.

7 Related Work

Whole (or partial) data replication was deployed in earlier distributed file systems, such as HDFS and GFS [12]. However, replication substantially increases the storage overheads, translating to higher operational costs for service providers. Modern distributed file systems, such as Google Colossus [8] and Windows Azure Storage [14], use erasure codes such as Reed-Solomon to reduce the storage overheads. This might work well for data in the clear, but applying erasure codes directly over HE ciphertext will increase the size of the generated codewords.

There are recent works that followed the EwR design to protect data confidentiality while achieving recoverability and integrity. One closely related work is by Lin and Tzeng [18], who proposed a framework to support secure data storage, forwarding, and retrieval on the Cloud. Given a message m, their framework splits m into κ blocks such that $m = \{m_1, m_2, .., m_\kappa\}$ and encrypts these κ blocks individually into ciphertexts $c_i = \mathsf{Enc}(m_i)$ using a bilinear map with a prime or-

der p. For two cyclic multiplicative groups $\mathbb{G}_1$, $\mathbb{G}_2$ and a generator $g \in \mathbb{G}_1$, there is a bilinear map $\varepsilon : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$. For any $x, y \in \mathbb{Z}_p^*$, the following *multiplicative* homomorphism property holds: $\varepsilon(g^x, g^y) = \varepsilon(g, g)^{xy}$. Users will upload these κ ciphertexts to the matching number of storage servers. Once completed, the server will sample a generator matrix $G = [g_{i,j}]$ for $1 \leq i \leq \kappa, 1 \leq j \leq n$ and compute codewords as $\sigma_j = c_1^{g_{1,j}}, c_2^{g_{2,j}}, .., c_\kappa^{g_{\kappa,j}}$ for $1 \leq j \leq n$. In this way, they distribute the codewords to n servers, also called decentralized erasure codes [9] in the literature. Ciphertexts are stored on $\kappa < n$ servers, but the codewords are spared across all n servers. The recovery process is similar to Reed-Solomon, which requires finding a multiplicative inverse of a $\kappa \times \kappa$ submatrix K of G. They also proposed the use of (t, n)-Shamir-secret-sharing [25] to protect the secret key and modified their encryption to a threshold version. As discussed, data is broken into κ blocks before it is individually encrypted and stored on κ storage servers. Retrieving this data requires a distributed decryption and reconstruction protocol. Each of the κ blocks is individually encrypted, which means there will be a large number of ciphertexts. Also, this framework was designed as a Cloud storage solution; how these encrypted and encoded data can be used in various applications is unknown. Complex multi-party computation (MPC) protocols may be needed to support the *limited* homomorphic multiplication due to the chosen algebraic setting.

Building on [18], Shen et al. [26] extended the scheme to support data integrity checking. Based on the same algebraic setting, the authors proposed a tag generation algorithm that produces tags for each of the κ blocks of ciphertexts. These tags can be used for recovery due to loss of data and for checking data integrity. More importantly, these tags are homomorphic in the data forwarding and recovery processes. Although this extension achieves recoverability and integrity simultaneously, it inherits all the drawbacks we discussed earlier. In addition, the generated tags are multiplicatively homomorphic to the data forwarding and recovery, but they are not fully homomorphic in computations due to the multiplicative bilinear map foundation. This means we need to update these tags after homomorphic computations. This process is expensive because it requires reshuffling many shares of the ciphertext blocks.

For integrity, Tsoutsos et al. [28] proposed a protocol that extends the Paillier HE scheme [21] to ensure the correctness of homomorphic ciphertexts after computations. The authors used a Mersenne prime $p = 2^d - 1$ for some integer d to compute the codewords from the ciphertexts to perform efficient residue-based checks. Given a ciphertext c, the corresponding codeword is generated as $\sigma_c = c \pmod{p}$. The Mersenne prime is multiplied by the other two primes to generate the modulo n in the Paillier scheme. The idea for having p is that we can make it public for codeword generation and verification, and we can mix it within the modulo n so that the generated codeword is associated with the ciphertexts. This is because of the following theorem: $(x \pmod{n}) \pmod{p} = x \pmod{p}$ if p dividing n and x is a non-negative integer. Another important advantage of this protocol is that the codeword is additively homomorphic through computations.

More specifically, given two cipertexts a, b we compute $c = a \cdot b \pmod{n^2}$ and $c \pmod{p} = \sigma_a \cdot \sigma_b \pmod{p}$.

Note that the homomorphic multiplication of two ciphertexts in Paillier maps to adding two plaintexts. This protocol was designed based on Paillier for efficient evaluation of homomorphic computations, but it inherits the limitation of only supporting additively homomorphic computations. This protocol also only supports individually encrypted ciphertexts; hence, the codewords are generated for each of these ciphertexts. Our proposed protocol differs fundamentally from the use of Ring-LWE HE schemes (see Sec. 2), which support ciphertext packing to reduce the codeword overheads significantly. Of course, our ciphertexts support both additively and multiplicatively homomorphic computations.

Compared to closely related works by Tsoutsos et al. [28] and Shen et al. [26], our work has a number of advantages. Firstly, when considering the plaintext to ciphertext ratio (number of integers that can be put inside a ciphertext), our work utilizes ciphertext packing (as mentioned in Sec. 2.1) to store many plaintext values into a single ciphertext. This leads to lower space complexity compared to individually encrypting one plaintext value into a ciphertext. Similarly, the computational complexity is reduced because operations applied on the packed ciphertexts are carried out on all underlying plaintext simultaneously. Secondly, previous works were based on partial HE, supporting either addition or multiplication but not both. Our work is based on somewhat or fully HE, which means both addition and multiplication are supported when designing functions to operate on data encoded into the X-Cipher structure. Thirdly, like Shen et al. [26], ciphertexts are distributed but each ciphertext in our work contains a list of plaintext values. If we encode input data using column-major, many applications can operate on the plaintext values on a column-by-column (or ciphertext-by-ciphertext) basis without interaction between chunk server. Fourthly, the codewords in our work are partially homomorphic through addition since only addition is required for recovery and verification. Because they are not fully homomorphic, the codewords can be regenerated after multiplication operations.

8 Conclusion

We introduce X-Cipher, which makes data recoverable and private in all phases of its use. We introduce an approach of encrypting encoded plaintext alongside codewords for homomorphic recovery capabilities. X-Cipher's use of encoding and ciphertext packing reduces the cost of storing data and codewords. Experiments are conducted to demonstrate that standard building blocks of cloud operations, such as matrix operations and PSI, are possible in X-Cipher while maintaining the ability to recover the data at any point during its use. In future work, we aim to add a detection capability for interference by a stronger adversary and aim to add support for multiple colluding corrupted chunk servers.

References

1. An, J.H., Bellare, M.: Does encryption with redundancy provide authenticity? In: Pfitzmann, B. (ed.) Advances in Cryptology — EUROCRYPT 2001. pp. 512–528. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
2. Backes, M., Berrang, P., Bieg, M., Eils, R., Herrmann, C., Humbert, M., Lehmann, I.: Identifying personal DNA methylation profiles by genotype inference. In: IEEE Symposium on Security and Privacy. pp. 957–976 (2017)
3. Boneh, D., Segev, G., Waters, B.: Targeted malleability: Homomorphic encryption for restricted computations. Cryptology ePrint Archive, Report 2011/311 (2011), https://eprint.iacr.org/2011/311
4. Brakerski, Z., Gentry, C., Halevi, S.: Packed ciphertexts in lwe-based homomorphic encryption. In: International Workshop on Public Key Cryptography. pp. 1–13. Springer (2013)
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 309–325. ACM (2012)
6. Chang, Y., Li, J., Lu, N., Shi, W., Su, Z., Meng, W.: Practical privacy-preserving scheme with fault tolerance for smart grids. IEEE Internet of Things Journal (2023)
7. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1243–1255 (2017)
8. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Google's globally distributed database. ACM Transactions on Computer Systems (TOCS) **31**(3), 1–22 (2013)
9. Dimakis, A., Prabhakaran, V., Ramchandran, K.: Decentralized erasure codes for distributed networked storage. IEEE Transactions on Information Theory **52**(6), 2809–2816 (2006). https://doi.org/10.1109/TIT.2006.874535
10. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. **30**(2), 170–231 (Jun 1998)
11. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. Cryptology ePrint Archive, Report 2011/566 (2011), https://eprint.iacr.org/2011/566
12. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles. pp. 20–43. Bolton Landing, NY (2003)
13. Halevi, S., Shoup, V.: Algorithms in helib. In: Annual Cryptology Conference. pp. 554–571. Springer (2014)
14. Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., Yekhanin, S.: Erasure coding in windows azure storage. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12). pp. 15–26. USENIX Association, Boston, MA (Jun 2012), https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang
15. IDG: Idg's 2020 cloud computing study. https://www.idg.com/tools-for-marketers/2020-cloud-computing-study/ (2020), accessed: 2021-05-04
16. Jiang, X., Kim, M., Lauter, K., Song, Y.: Secure outsourced matrix computation and application to neural networks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1209–1222 (2018)

17. Lihao Xu, Bruck, J.: X-code: Mds array codes with optimal encoding. IEEE Transactions on Information Theory **45**(1), 272–276 (1999)
18. Lin, H.Y., Tzeng, W.G.: A secure erasure code-based cloud storage system with secure data forwarding. IEEE Transactions on Parallel and Distributed Systems **23**(6), 995–1003 (2012). https://doi.org/10.1109/TPDS.2011.252
19. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 1–23. Springer (2010)
20. Mohammadali, A., Haghighi, M.S.: A privacy-preserving homomorphic scheme with multiple dimensions and fault tolerance for metering data aggregation in smart grid. IEEE Transactions on Smart Grid **12**(6), 5212–5220 (2021)
21. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) Advances in Cryptology — EUROCRYPT '99. pp. 223–238. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
22. Plank, J.: Erasure codes for storage systems: A brief primer. login Usenix Mag. **38** (2013)
23. Plank, J.S.: The raid-6 liber8tion code. The International Journal of High Performance Computing Applications **23**(3), 242–251 (2009). https://doi.org/10.1177/1094342009106191
24. Reed, I., Solomon, G.: Polynomial codes over certain finite fields. Journal of The Society for Industrial and Applied Mathematics **8**, 300–304 (1960)
25. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979)
26. Shen, S.T., Lin, H.Y., Tzeng, W.G.: An effective integrity check scheme for secure erasure code-based storage systems. IEEE Transactions on reliability **64**(3), 840–851 (2015)
27. Smart, N.P., Vercauteren, F.: Fully homomorphic simd operations. Designs, codes and cryptography **71**(1), 57–81 (2014)
28. Tsoutsos, N.G., Maniatakos, M.: Efficient detection for malicious and random errors in additive encrypted computation. IEEE Transactions on Computers **67**(1), 16–31 (2017)