X-Cipher: Achieving Data Resiliency in Homomorphic Ciphertexts

Adam Caulfield, Nabiha Raza, Peizhao Hu
Rochester Institute of Technology

Abstract

Homomorphic encryption (HE) allows for computations on encrypted data without requiring decryption. HE is commonly applied to outsource 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 data resilient by ensuring it is private and fault-tolerant simultaneously at all stages during data-outsourcing. X-Cipher allows for data recovery without decryption, 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 synthetic dataset, and compared to prior work to demonstrate X-Cipher enables additional security capabilities while enabling privacy-preserving outsourced computations.

1 Introduction

Outsourcing data or computations to the Cloud has become an ever growing trend for the private sector, corporations, governments, and non-profits. In 2020, corporations spent 32% of their IT budget on Cloud services [17]. Despite this, privacy concerns have limited many applications which handle sensitive data from making use of Cloud resources, such as applications which handle health and financial data. To support outsourced computations over sensitive data while preserving its privacy, Homomorphic Encryption (HE) has commonly been leveraged. Data encrypted under an HE can undergo computations without decryption.

In addition to privacy concerns when outsourcing data, there is also a concern of data corruption and loss. To address fault-tolerance, data replication was widely exercised in distributed data storage systems [13], but the cost of maintaining exact replicas or data chunks dramatically increases. Commonly used systems, such as Google Colossus [9], Windows Azure Storage [16], adapt erasure codes to reduce the storage overhead. These systems typically apply the erasure codes over user supplied data, which previously is assumed to be plaintext 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 [5] and thus codewords generated through conventional methods will need to 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 probabilistic HE schemes, applying erasure codes over these ciphertexts will significantly increase the size of the generated codewords and thus increases both the storage and operational overhead.

To combat this problem, one approach is to make use of *Encrypt-with-Redundancy (EwR)* [3], which combines an encryption scheme with erasure codes. There are a few recent works following the EwR paradigm. Shen et. al. [27] proposed a protocol based on (t, n) homomorphic secret-sharing to address homomorphic data recovery. An erasure code and integrity tags are distributed among n storage servers. Recovery and integrity checking can take place by retrieving $t < n$ shares and performing a recombination. A drawback to this approach is that data cannot be recombined without t participants being online, and it requires additional steps of recombination in order to verify the data. Tsoutsos and Maniatakos [29] proposed methods to generate codewords that are homomorphic: the codewords are updated when homomorphic computations occur. These codewords are used for detecting and correcting errors at the hardware-level. This approach is designed in a way that requires each ciphertext to be individually encrypted, incurring high computation and space complexity. Furthermore, these recent works are designed based on partial homomorphic encryption for efficiency, but this choice limits the scheme to use in the applications in which the ciphertext does not require complex arithmetic operations.

To address these challenges, we propose X-Cipher: a framework to simultaneously make data private and recoverable in a way that reduces overhead and enables complex homomorphic operations. X-Cipher leverages an erasure-code called X-Code [19] to generate codewords which can be subsequently used to recover from data losses. Furthermore, X-Cipher employs encoding and packing techniques to significantly reduce the space overheads and speedup the homomorphic computations. Because of this, codewords are stored along with the data to efficiently validate the data authenticity while incurring minimal to no additional overhead. X-Cipher introduces algorithms to verify the state of the data and enables recovery capability without revealing the data. To our knowledge, no previous works have leveraged erasure codes along with HE in the same manner as X-Cipher in order to provide these security guarantees together, while minimizing storage overhead, and maintaining both fault-tolerance and privacy guarantees throughout some homomorphic operations. The main contributions are summarized by the following:

- The use of an erasure code to generate codewords, encrypted alongside plaintext data in order to efficiently recover homomorphically encrypted data without requiring decryption.
- A design and encoding algorithm which leverages ciphertext packing to enable efficiency in terms of time, space, and recovery volume.
- Example application specific and primitive building block algorithms which minimizes or removes the need for codewords regeneration after consecutive homomorphic operations.

Table 1: Common notations

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

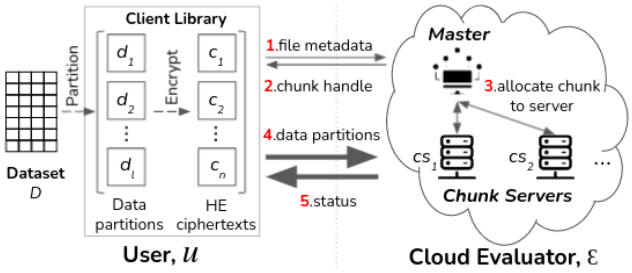


Figure 1: System Model.

The rest of the paper is organized as follows. Section 2 discusses the system and threat models. Section 3 overviews background techniques. Section 4 discussed related works. Section 5 details the proposed solutions. Section 6 analyzes the security and complexity of the proposed solutions. Section 7 presents experiments and empirical study using synthetic datasets. Section 8 concludes the paper.

2 System and Threat Models

2.1 System Model

In this paper, we assume a data and computation outsourcing setup, as illustrated in Fig. 1. More specifically, a user $\mathcal{U}$ wants to delegate a private dataset D to a cloud evaluator $\mathcal{E}$ for outsourced computations, such as document analysis, machine learning. Modern distributed filesystems, from GFS [13] to Colossus [9], 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. 1. Concerned over privacy, the user may want to homomorphically encrypt these data partitions: given a public key pk, $c_j = Enc(\text{pk}, d_j); d_j \in \boldsymbol{d_i}$ (or $c_i = Enc(\text{pk}, \boldsymbol{d_i})$ using ciphertext packing technique discussed in Sec. 3.1), before sending them to the cloud evaluator. To ensure high efficiency, clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunk servers [13].

Once data is uploaded to the distributed file system, cloud data processing frameworks, such as Spark [31], allocate computing tasks to worker nodes that are close to chunk servers where task-specific input data is stored according to some data locality policies. During the execution of the computing tasks, these data partitions are modeled as a resilient distributed dataset (RDD) [30] in Spark for efficient parallel data processing over a cluster. In this cloud setting, fault-tolerance is addressed by data replication on multiple chunk servers, dramatically increasing the storage overhead. Colossus adopts Reed-Solomon codes [25] to reduce the cost of data replication.

Under this system model, disk or system failures render user supplied ciphertexts unavailable. These failures can be from power outages, system malfunctions, or attacks. Data replication is a conventional method to address fault-tolerance but dramatically increases the storage overhead [20]. We address this problem with erasure codes. Different from existing systems, such as Colossus, which adopts Reed-Solomon codes, we propose the use of a different erasure code that has lower computational complexity. Using the generated codewords σ from the erasure codes and the remaining data on other chunk servers, we can recover the lost data partitions.

Leveraging the capability of homomorphic encryption, the cloud evaluator will perform the requested computations while data stays encrypted. We propose new methods to ensure the fault-tolerance and integrity checking of the homomorphically encrypted ciphertext. In addition, we design example homomorphic algorithms to demonstrate how computing tasks can be securely evaluated while maintaining the recoverability.

2.2 Threat Model

For this work we assume a semi-honest cloud evaluator. The evaluator follows the protocol specification and can only learn about the data by observation during any stage of the computational process (as input, intermediate results, outputs). In addition, these ciphertexts might be subjected to corruption or complete loss due to adversarial behaviors that the cloud evaluator cannot detect, such as causing system failures.

In addition, we envision a scenario in which a malicious adversary $\mathcal{A}dv$ has the capability of fully compromising a chunk server, as illustrated in Fig. 1. However, although we assume $\mathcal{A}dv$ can compromise a chunk server cs_i, $\mathcal{A}dv$ does not have access to the secret key sk. Because of this, $\mathcal{A}dv$ modifies the ciphertext(s) stored on cs_i in order to either corrupt the subsequent computation or attempt to leak the data.

3 Preliminaries

3.1 Homomorphic Encryption

Our proposed solutions are designed on the BGV HE scheme [7], which bases its security on the Ring-LWE (*Learning-with-Error*) problem [21]. Detailed discussions on the BGV scheme and the security of Ring-LWE problem are out of the scope of this paper. We refer interested readers to our recent survey paper [2] for these topics. Briefly speaking, the BGV HE scheme defines a tuple of *probabilistic-polynomial-time* (PPT) algorithms (`Setup`, `KeyGen`, `EvalKeyGen`, `Enc`, `Dec`, `Add`, `Mult`, `Relinearize`):

- `BGV.Setup`$(1^\lambda, 1^L) \rightarrow pp$: Given the security parameter λ and maximum multiplicative depth L, choose a cyclotomic polynomial $\Phi(x) = x^d + 1$ with d being a power of 2. Define $R = \mathbb{Z}[x]/(\Phi(x))$ as a polynomial ring of degree d with integer coefficients. Generate the error and key distributions χ and ψ over R respectively. Choose the ciphertext modulus q

and the plaintext modulus t, and define two additional polynomial rings $R_t = \mathbb{Z}_t[x]/(\Phi(x))$ and $R_q = \mathbb{Z}_q[x]/(\Phi(x))$ for plaintext and ciphertext respectively. Finally, output public parameter $pp = (d, q, t, \chi, \psi)$. All following algorithms are assumed to implicitly take pp as an input.

- $\texttt{BGV.KeyGen}() \rightarrow (\mathsf{pk}, \mathsf{sk})$: Sample element $s \leftarrow \psi$ and noise $e \leftarrow \chi$, which are typically Gaussian distributions for small key and noise. Uniformly sample $a \leftarrow R_q$. Given these elements, set the secret key as $\mathsf{sk} = s$ and public key as $\mathsf{pk} = (b, a) \in R_q^2$ with $b = -as + te \pmod q$ according to the Ring-LWE assumption [21].
- $\texttt{BGV.EvalKeyGen}(s) \rightarrow \mathsf{ek}$: Given the secret key s, generate an evaluation key ek by sampling ζ many elements such that $\tilde{\mathbf{a}} \leftarrow R_q^{\zeta}$, $\tilde{\mathbf{e}} \leftarrow \chi^{\zeta}$ and setting $\tilde{\mathbf{b}} = \tilde{\mathbf{a}} \cdot s + \tilde{\mathbf{e}} + \boldsymbol{g} + s^2 \pmod q$. Output $\mathsf{ek} = (\tilde{\mathbf{b}}, \tilde{\mathbf{a}})$. We define a *gadget toolkit* [2] needed for *key-switching*.
  - Gadget vector: $\boldsymbol{g} = (g_0, \ldots, g_{\zeta-1}) \in R^{\zeta}$ for some integer $\zeta \geq 1$.
  - $\text{Decomp}(x)$: Given an element $x \in R_q$, decompose it into a *short* vector $\boldsymbol{u} = (u_0, \ldots, u_{\zeta-1}) \in R^{\zeta}$ such that $\langle \boldsymbol{u}, \boldsymbol{g} \rangle = x \pmod q$.
- $\texttt{BGV.Enc}(\mathsf{pk}, m) \rightarrow c$: Given a plaintext message $m \in R_t$, a public key $\mathsf{pk} = (b, a)$, uniformly sample a random $r \leftarrow \psi$ and errors $e_0, e_1 \leftarrow \chi$, and encrypt the message m as $c = (c_0, c_1) \in R_q^2$ where $c_0 = rb + m + te_0$ and $c_1 = ra + te_1$. Note, c_0, c_1 indicate two ciphertext components, different from the meaning in the rest of this paper.
- $\texttt{BGV.Dec}(\mathsf{sk}, c) \rightarrow m$: Given a ciphertext $c = (c_0, c_1) \in R_q^2$ and the secret key $sk = s$, set $s = (1, s) \in R_q^2$ and decrypt by computing $m = (\langle c, s \rangle \pmod q) \pmod t$.
- $\texttt{BGV.Add}(c, c') \rightarrow c_{add}$: 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 $c, c' \in R_q^2$, 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.
- $\texttt{BGV.Relinearize}(\mathsf{ek}, \tilde{c}_{mult}) \rightarrow c_{mult}$: Given the evaluation key $\mathsf{ek} = (\tilde{\mathbf{b}}, \tilde{\mathbf{a}})$ and a long ciphertext $\tilde{c}_{mult} = (\tilde{c_0}, \tilde{c_1}, \tilde{c_2})$, perform the following relinearization step to change ciphertext from an encryption of s^2 to s:
  - Apply gadget decomposition $\mathbf{g}(\tilde{c_2})^{-1} = (u_0, \ldots, u_{\zeta-1})$
  - Output the relinearized ciphertext as $c_{mult} = (\tilde{c_0}, \tilde{c_1}) + \sum_i u_i \cdot (\tilde{\mathbf{b}}[i], \tilde{\mathbf{a}}[i]) \in R_q^2$

Given the capability to homomorphically evaluate addition and multiplication, any function built on these two arithmetic primitives can be evaluated homomorphically. For instance, given two integers x, y represented in $\{0, 1\}$, many logic gates can be translated into the arithmetic forms: $\text{AND}(x, y) = xy$, $\text{OR}(x, y) = x + y - xy$, $\text{XOR}(x, y) = x + y - 2xy$, and $\text{NOT}(x) = 1 - x$. These logic gates can be homomorphically evaluated if $x, y \in \mathbb{Z}_2$. As an example, we can evaluate XOR using homomorphic addition, as $\text{XOR}(x, y) = x + y$.

In general, homomorphically encrypted ciphertexts are large due to the use of complex lattice elements; e.g., the uniformly sampled $a \leftarrow R_q$ in the KeyGen() algorithm. Due to this significant ciphertext expansion, HE has been considered impractical for many applications. To address this problem,the ciphertext packing technique [6] was introduced. This allows for the encoding multiple plaintext messages into one polynomial and encrypt 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 and speeds up the homomorphic computations because operations are performed on plaintext values simultaneously in a element-wise and SIMD (Single-Instruction-Multiple-Data) manner [28]. We make use of ciphertext packing in this work for computational and storage efficiency.

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 [28] 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 [28]. Given this property, we can encode each plaintext message $m_i \in M$ into an arbitrary polynomial $f(x) \pmod{\Phi_i(x)}$. The total number of plaintext messages we can fit into these polynomials is referred to as the plaintext slot count ϱ. Therefore, ciphertext packing means that we can encrypt a plaintext vector of length ϱ into a single ciphertext.

The CRT-based ciphertext packing technique allows a set of 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 [12]. 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*.

Another useful trick we will use in our proposed solutions is masking. The idea is similar to bit-masking. Given a plaintext vector as a mask $\Xi = \{0, 1\}^k$, we can set one or more positions to one, leaving others default to zero, to select values within a packed ciphertext. After constructing the mask Ξ, performing an element-wise multiplication with the mask Ξ and the ciphertext can extract the desired values.

3.2 Erasure Codes

Erasure codes are used to achieve fault tolerance ability in systems [23]. 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 [25] is a widely used code for detecting and correcting erasures in data storage [27]. The Reed-Solomon codes use Galois Field $GF(2^w)$ multiplication and maintain a Vandermonde matrix. For homomorphically encrypted data which expands beyond the size of unencrypted plaintext, this solution would increase the computational overhead drastically. Erasure codes based on RAID-6

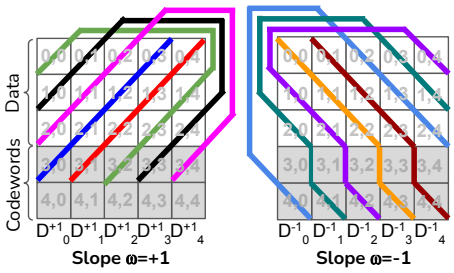


Figure 2: An example of X-Code setup ($n = 5$).

algorithms [24] 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. 3.1, XOR for binary integers is simply addition which is an efficient homomorphic operation.

Our work is based on X-Code [19] which can recover up to two full columns erasures in an $n \times n$ structure where n is a prime. Figure 2 shows an example of 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 XORing (or homomorphically adding) data symbols along the diagonal that connects them, as illustrated in Fig. 2. These diagonals have slopes ± 1 and cross each other when overlayed, 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. 2, 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.

4 Related Work

Protecting confidentiality is essential towards data resiliency and is typically achieved using cryptography primitives, such as encryption. However, encryption alone does not address all the needs for data resiliency. For instance, adversaries can corrupt the ciphertexts causing decryption errors or simply delete them. This section reviews the most related works that propose solutions to tackle the other aspects of data resiliency. More specifically, we focus on fault-tolerance and data integrity.

Loss of data can disrupt service availability. Whole (or partial) data replication was deployed in earlier distributed file systems, such as HDFS, GFS [13]. However, replication substantially increases the storage overheads, which translates to higher operational cost for service providers. Modern distributed file systems, such as Google Colossus [9], Windows Azure Storage [16], use erasure codes such as Reed-Solomon to reduce the storage overheads. This might work well for data in the clear. Applying erasure codes directly over ciphertext will increase both the size of the generated codewords if ciphertext is larger than its plaintext and increase the computation time needed for generating the codewords.

There are recent works that followed EwR design to protect data confidentiality while achieving fault-tolerance and integrity. One of the closely related works is by Lin and Tzeng [20], 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 $c_i = \mathsf{E}(m_i)$ using a bilinear map with a prime order 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; this is also called decentralized erasure codes [10] in the literature. Note that ciphertexts are store 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 [26] to protect the secret key and modified their encryption to a threshold version. As discussed, data is broken into κ blocks before individually encryption 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 mean 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; this is due to the chosen algebraic setting.

Building on [20], Shen et al. [27] 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 fault-tolerance 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 it is 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 of many shares of the ciphertext blocks.

For integrity, Tsoutsos et al. [29] proposed a protocol, which extends the Paillier HE scheme [22], to ensure 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 with 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

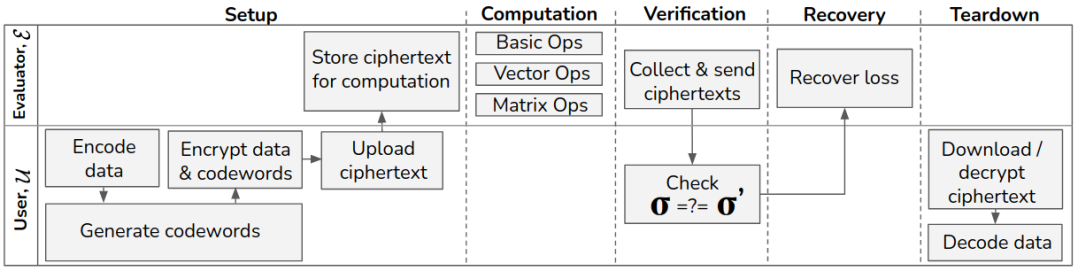


Figure 3: System overview.

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 $c \pmod p = \sigma_a \cdot \sigma_b \pmod p$.

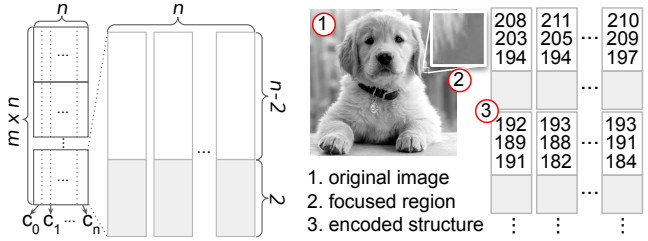
Note that homomorphic multiplication of two ciphertexts in Paillier maps to the addition of two plaintexts. This protocol was designed based on Paillier for efficient evaluation of homomorphic computations, but it inherits limitation of only supporting additively homomorphic computations. This protocol also only support 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. 3) which support ciphertext packing to significantly reduce the codeword overheads. Of course, our ciphertexts support both additively and multiplicatively homomorphic computations.

5 X-Cipher Design

We propose X-Cipher to simultaneously support confidentiality and fault-tolerance of homomorphically encrypted data that is partitioned and stored on a distributed system. An overview of the X-Cipher features and system are demonstrated in Figure 3.

At the base of our X-Cipher design is X-Code, as introduced in Sec. 3.2. The original X-Code scheme [19] was designed for storing bits and thus the codewords were calculated with simple XOR-ing. Our design can inherit this setting (arithmetic modulo prime 2), and we generalize 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 apply to a variety of applications. Quite simply, we propose the use 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. [19] presents an extensive proof of how the original X-Code achieves the MDS property. The MDS property is dependent upon the dimension



(a) Extended structure. **(b) Encoding example.**

Figure 4: Extended structure and its encoding example.

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. Interested readers can refer to [19] for the detailed proof.

Once we instantiate the X-Cipher structure with the appropriate parameters, we construct the first $n-2$ rows with plaintext data and generate codewords in the last 2 rows. We then partition the $n \times n$ structure 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. 3.1. 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. 1.

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 important observations regarding the spatial efficiency. First, the ratio of codewords to data is $\frac{2}{n-2}$ which decreases as n increases. Because the X-Code structure is a square, increasing the value of n increases both the data rows (packed ciphertext slots) and also increases the number of columns (ciphertexts). As a consequence, representing the data in an unmodified X-Code structure will require more ciphertexts and thus more servers 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, are no longer to support the X-Code structure; however, as a consequence, only n of the plaintext slots within the packed ciphertext are utilized. Ideally, the number of plaintext slots should be maximized for efficiency, and in this scenario it is likely that the number of plaintext slots will be significantly underutilized since for a properly selected set of security parameters there are thousands of these slots.

Table 2: Details of each phase.

	User, $\mathcal{U}$	**Cloud Evaluator,** $\mathcal{E}$
Setup	1) Starts cryptosystem with security parameters $\lambda = (\varrho, t)$ 2) Generates keys through $(\mathsf{pk}, \mathsf{sk}, \mathsf{ek}) \leftarrow \mathtt{KeyGen}(\lambda)$ 3) Chooses a prime n and determine multiples $m = \frac{\varrho}{n}$ 4) Encode matrix D to columns $\boldsymbol{d_i}$ 5) Compute codewords σ and update $\boldsymbol{d_i}$ 6) Encrypts columns $c_i = \mathtt{Enc}(\mathsf{pk}, \boldsymbol{d_i}); c_i \in \boldsymbol{c}$ 7) Sends $\boldsymbol{c}$ to $\mathcal{E}$ for storage and computation	1) Receives $\boldsymbol{c}$ from $\mathcal{U}$, stores c_i on cluster server cs_i (Fig. 1)
Computation	1) Calls for homomorphic operations	1) Executes homomorphic operations 2) Regenerates codewords if necessary
Verification	1) Calls for set of ciphertexts $\boldsymbol{c}$ from $\mathcal{E}$ 2) Decrypts ciphertexts $\boldsymbol{d_i} = \mathtt{Dec}(\mathsf{sk}, \boldsymbol{c_i})$ 3) Decodes X-Cipher to get data and codewords (D, σ') 4) Generate expected codewords σ for D 5) If $\sigma \neq \sigma'$, call for recovery.	1) Collects & reassembles ciphertexts $\boldsymbol{c}$ from cluster servers 2) Sends $\boldsymbol{c}$ back to user $\mathcal{U}$
Recovery	1) Sends $\mathcal{E}$ lost column index(es) idx, when verification fails 2) Waits for acknowledgement from $\mathcal{E}$	1) Receives index(es) idx from $\mathcal{U}$ 2) Collects partial set of ciphertexts $\boldsymbol{c}$ from servers 3) Performs `OneColumnRecovery` or `TwoColumnRecovery` 4) Notifies $\mathcal{U}$ of success
Teardown	1) Waits for set of ciphertexts $\boldsymbol{c}$ from $\mathcal{E}$ 2) Decrypts ciphertexts $\boldsymbol{d_i} = \mathtt{Dec}(\mathsf{sk}, \boldsymbol{c_i})$ 3) Decodes X-Cipher structure back to original form	1) Collects & reassembles ciphertexts $\boldsymbol{c}$ from cluster servers 2) Sends $\boldsymbol{c}$ back to user $\mathcal{U}$

Inspired by these observations, we build X-Cipher's internal structure as one larger vertical rectangle structure abstracted as m copies of the $n \times n$ X-Code structures stacking on top of each other, as illustrated in Fig. 4a. 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.

Table 2 details each phase between a Cloud evaluator $\mathcal{E}$ and a user $\mathcal{U}$. We refer readers to Table 1 for common notations used in this paper.

5.1 Setup Phase

To initiate X-Cipher, we begin with selecting some security parameter (λ) which instantiates the BGV HE scheme, as discussed in Sec. 3.1. These parameters should be selected according to the characteristics of the application datasets. In Sec. 5.2, we provide two examples showing how to map application datasets into the X-Cipher structure and to support homomorphic computations. Three of these parameters are of particular importance. Firstly, the plaintext modulus t should be greater than the maximum individual value in the given datasets. Next, the modulus chain should be large enough to support the intended number of homomorphic multiplications. With other parameters, we can calculate the plaintext slot count ϱ, which determines the total number of plaintext integers which can be packed into a single ciphertext. Finally, we generate necessary keys for encryption, decryption, and evaluation.

Then, users determine the value of n, which determines the total columns of the X-Cipher structure and the number of ciphertexts generated from a given dataset. As previously described, n also represents the number of chunk servers that are required to store these ciphertexts. The selection of n should adhere to several constraints based upon the X-Code requirements, cryptographic parameters, and dataset size. First, X-Code requires that n is both a prime number and $n > 3$. Next, the slot count ϱ and n determine the number of copies m of the $n \times n$ X-Code structures that are required; that is $m = \frac{\varrho}{n}$. Given a matrix D, we split it into a set of column vectors $\boldsymbol{d_i}$ for distributed storage and parallel tasks. Note that $\boldsymbol{d_i}$ includes the generated codewords from applying erasure coding. 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. 4a.

Once we have configured the X-Cipher structure, $\mathcal{U}$ encodes the data into the structure. For multidimensional data, the data is flatten by applying a dimension reduction technique which is most suitable for the intended computations. Figure 5 illustrates four different dimension reduction techniques (also referred to as space-filling curves [11]). For example, data representing a two-dimensional matrix is typically flattened by traversing the matrix in either a row-major or column-major manner. Location data is commonly encoded by Z-order or Hilbert curves [1]. 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. This encoding allows the partitioned data to be packed and encrypted into the corresponding ciphertexts and computed independently and simultaneously. Data is filled into every $n - 2$ rows 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.

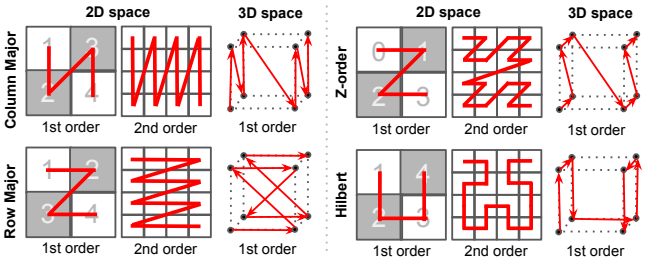


Figure 5: Dimension reduction techniques.

For one-dimensional inputs such as vectors, we can encode the entire vector continuously in a column-major fashion, overflowing to the subsequent column when all plaintext slots are filled. This encoding aims to store data in fewer ciphertexts hence we can avoid moving around the distributed ciphertexts. Alternatively, We can divide the vector into n sub-vectors and fill each of them into the corresponding ciphertext. This encoding encourages parallel processing and is preferred for large vectors. For two or more dimensions, we first perform dimension reduction using one of the discussed techniques. We encode the resulting one-dimension output as aforementioned. Fig. 4b illustrates an example of encoding an image into the proposed X-Cipher structure. For this example, we perform a column-major dimension reduction to transform the focused region into the illustrated X-Cipher structure.

Finally, the n columns of partitioned data vectors are encrypted as $c_i = Enc(\text{pk}, \boldsymbol{d_i})$, and the ciphertexts are uploaded to the Cloud evaluator $\mathcal{E}$. Then, $\mathcal{E}$ stores the ciphertexts among its n chunk servers and is ready to perform homomorphic computations. We distribute the resulting ciphertexts based on decentralized erasure codes [10], which enhances fault-tolerance, load-balancing, and accessibility. During the computation, verification, and recovery phases, we reassemble the X-Cipher structure by collecting these distributed ciphertexts from the respective chunk servers. In the rest of the discussions, we assume a reassembled X-Cipher structure for simplicity.

5.2 Computation Phase

The X-Cipher data structure provides fault-tolerance and recoverability for a wide set of applications that make use of basic linear algebra operations. Here, we present some of the commonly used operations over one or two datasets as building blocks and demonstrate their usefulness in two example applications.

Over One Dataset. Performing arithmetic operations over one dataset, such as a single matrix or an array of integers, is common in many applications. The total sum of all elements in the structure can be computed by naively adding all elements sequentially. For this and other operations, there are more efficient ways. As discussed in Sec. 3.2 and illustrated in Fig. 2, we observe that each codeword contains the sum of all data elements along its respective slope line; as illustrated for calculating $p_{3,0}$ in Sec. 3.2. As a result, the overal sum in the structure can be computed efficiently by adding all codewords for one of the slope line configurations followed by summing the resulting codewords from each X-Code block. We present this idea in Alg. 1, which is efficient for using component-wise homomorphic addition over the packed ciphertexts. It achieves $O(log_2 m)$ complexity by adding values across multiple X-Code

Algorithm 1: 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**
  $m--$
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.
  $c_{sum} += c'_{sum}$
end
if $m\%2 \neq 0$ **then**
  c'_{sum} =`shift`$(c_{sum}, -(m+1) \cdot n)$
  $c_{sum} += c'_{sum}$ ▷ Add the last element.
end
$c_{sum} = c_{sum}$.`multByConst`$(mask)$ ▷ Mask out unwanted data.

blocks simultaneously. To implement this algorithm, we make use of the `shift` and `multByConst` functions that are supported in HElib library [14]. 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 made recoverable by redistributing the resulting ciphertext c_{sum}. The codewords are additively homomorphic, so a total sum avoids the need for regenerating the codewords. In the final result, it is observed that the first $n-2$ rows contain partial sums of the dataset which add up to the total sum. 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 must have elements in the first $n-2$ rows which add to the codeword. Therefore by simply creating n duplicates of c_{sum}, this characteristic is met and a valid X-Cipher structure is created.

Over Two Datasets. Component-wise operations over vectors or matrices can be supported by simply 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, we can calculate dot-product as $c_i^{prod} = c_i^A \times c_i^B$ for $i = (0, .., n-1)$. Then, we calculate the sum over all c_i^{prod} using Alg. 1.

A good example to demonstrate 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 $\iota >> \zeta$, we perform PSI to determine and reveal only the common elements in these two sets, $\boldsymbol{x} \bigcap \boldsymbol{y}$, for all set elements $x_j \in \boldsymbol{x}$ and $y_i \in \boldsymbol{y}$. We explain this protocol between a sender $\mathcal{U}_{\mathcal{S}}$ who owns $\boldsymbol{x}$ and a receiver $\mathcal{U}_{\mathcal{R}}$ who owns $\boldsymbol{y}$. By using X-Cipher, the users can generate a PSI result which is encrypted, packed, and fault-tolerant.

A fast PSI protocol using HE [8] 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 we have to perform homomorphic subtraction for all x_j with y_i, we can pack these values into one or more of the X-Cipher ciphertexts. Once packed, these operations are performed simultaneously in a SIMD-manner.

We can further optimize the PSI as proposed in [8]. 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, we can realize the optimized PSI protocol 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. 5.1. 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. 3.1, 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. [18] 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 X-Cipher structure, we have 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.

5.3 Verification Phase

At the end of the setup phase, the user can call for verification of the data's integrity at any point before, during, or after the computation. A call to verify and execute an integrity check first involves pulling each ciphertext (c_i) from the chunk servers (cs_i) and returning the ciphertexts to the user. Upon receiving the ciphertexts, the user runs the local verification algorithm. This algorithm first locally decrypts the data and separates the current codewords σ' from the user data. Then, the user regenerates the expected codewords σ given the set of plaintext data. After this, a simple check occurs by comparing the two sets of codewords. If they are equal, no recovery is required. If there is any mismatch, the location of the mismatch is used to identify the corrupted chunk server and recovery phase is executed.

5.4 Recovery Phase

Using the data and codewords in conjunction, recovery in X-Cipher can be performed in a similar way as X-Code. Once data is packed into a column and encrypted as a ciphertext, the underlying column structure is preserved in the ciphertext. Hence, homomorphic operations on ciphertexts will have the effect of component-wise operations, and they can be executed to complete the recovery. As illustrated in Fig. 2, 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 of the n ciphertexts: data that is aligned diagonally in the plaintext structure must be aligned horizontally in ciphertext (or in the same plaintext slot) in order to complete recovery correctly. The required alignment can be 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 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). This rotation is implemented by leveraging the available function `rotate` in the HElib library [14]. This function takes the ciphertext and right-rotation amount ($\theta < 0$ means a left rotation) as input parameters. Since we abstract the ciphertexts as vertical columns X-Cipher, $\theta > 0$ corresponds to a downward rotation and $\theta < 0$ corresponds to an upward rotation.

The function `rotate` moves data within the ciphertext along its plaintext slots. One left rotation moves the first data element to the last slot. Hence, directly applying `rotate` to ciphertexts will be incorrect: the X-Code block boundaries will be violated and the X-Cipher structure will no longer contain m stacked $n \times n$ X-Code blocks.

To address this problem, we propose `RotCols` to rotate columns for data recovery. Algorithm 2 show steps of this function. The idea is based on extraction by masking and swapping, and the effect on one column is demonstrated in Fig. 6 (c). Given a ciphertext c_i that has ρ plaintext slots in which $m \times n$ is the dimension of the X-Cipher structure and n is the size of each X-Code block, the algorithm extracts the values to be rotated (highlighted with solid color) by applying a mask Ξ_{top} and the rest of the values by another mask Ξ_{bottom}, and then rotate them into position by

Algorithm 2: 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;
  $\Xi_{top} = \{0,1\}^{m \times n}, \Xi_{top}[i] = 1; i \pmod n < \theta_{up}$,
  $\Xi_{bottom} = \{0,1\}^{m \times n}, \Xi_{bottom}[i] = 1; i \pmod n \geq \theta_{up}$,
  $c_{bottom}^{\omega} = \texttt{rotate}(c_i \cdot \Xi_{bottom}, -1 \cdot \theta_{up})$
  $c_{top}^{\omega} = \texttt{rotate}(c_i \cdot \Xi_{top}, \theta_{down})$
  $c_i = c_{top}^{\omega} + c_{bottom}^{\omega}$ ▷ Reassemble the ciphertext;
end

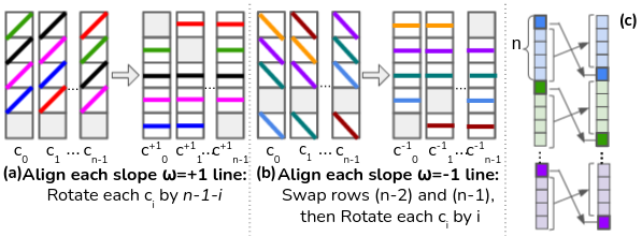


Figure 6: Column rotation aligns sloped (colored) lines.

calculating c_{top}^{ω} =`rotate`$(c_i \cdot \Xi_{top}, -1 \cdot \theta_{up})$ where $\theta_{up} = 1$ and c_{bottom}^{ω} =`rotate`$(c_i \cdot \Xi_{bottom}, \theta_{down})$ where $\theta_{down} = n - 1$. Both Ξ_{top} and Ξ_{bottom} are plaintext vectors which 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 column which is rotated.

X-Cipher inherits the two-column erasure property from X-Code. Hence, X-Cipher can recover up to two columns (ciphertexts) loss. For illustration, Alg. 3 shows steps to recover from losing one of the n columns (ciphertexts). To recover a lost column, we first perform the column rotation algorithm (Alg. 2) to transform the input according to two slope configurations, as shown in Fig. 6. Once rotation is done, we perform the recovery computations on the two resulting structures. Note that the shaded cells within each ciphertext contain codewords calculated by the other slope line configuration. They are not in use when performing computations for a specific slope line configuration. For example, when slope $\omega = +1$ the shaded cells, which refer to the $\omega = -1$ configuration codewords, are ignored since they are unused by the $\omega = +1$ configuration.

To elaborate the recovery process, we assume the first column (ciphertext) c_0 is lost, as illustrated by Fig. 7 (Step 1). We perform column rotation for slope $\omega = +1$ resulting in Step 2, then we recover $n-1$ cells. In a basic understanding of the algorithm, this is done in one direction first. However in our implementation, Step 2 occurs on two versions of the data, one for each rotation direction. The general rule for recovery is that a lost codeword is recovered by adding the remaining data, whereas a lost data is recovered by subtracting the remaining data from the codeword. Therefore, we can restore this cell value by using the codeword minus the

Algorithm 3: 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

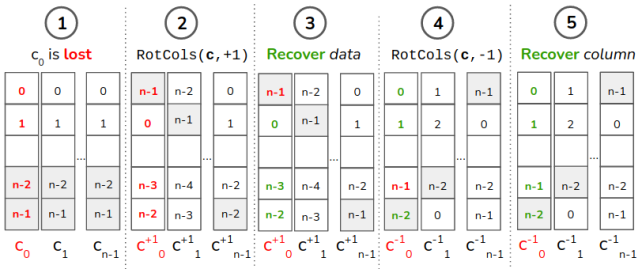


Figure 7: Illustration of one column recovery.

remaining data. This can be made efficient by constructing a mask u for each column and homomorphically accumulate the masked columns. The location of the codewords and data in each column are known, so it is simple to assign the correct sign in the bit mask based on what is in the row of the next column: 1 to add, -1 to subtract, or 0 to ignore. Because operations are performed in component-wise manner, all rows can be updated at once with an addition, as shown in (Step 3).

To recover the remaining cell $c_{0,n-1}$, we perform column rotation for slope $\omega = -1$ and get a transformed structure as shown in Fig. 6 (b) and Fig. 7 (Step 4). We observe that the codeword has been moved to the $n-2$ slot in c_0^{-1}. According to the recovery rule, this cell is a codeword and its value is calculated as the sum of all horizontal data cells within the transformed structure. We also have to apply the masking where $\boldsymbol{u}[n-2] = 1$. To help with this reassembling process, we create a coordinate map to keep track of the transformations applied by the different slope configurations. After computing the final sum in Step 5, the lost row has been successfully recovered.

Note that we illustrate the recovery algorithm using one of the m X-Code blocks. Since the X-Cipher structure is organized as m X-Code blocks stacking on top of each other and homomorphic operations are applied in a component-wise manner, the recovery algorithm acts on all m X-Code blocks.

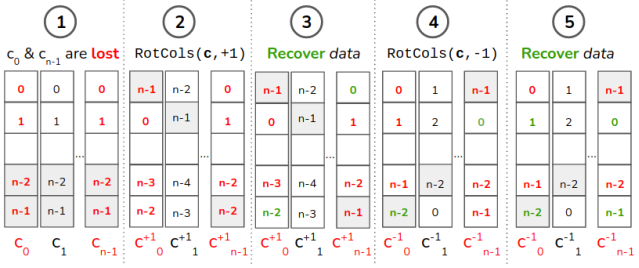


Figure 8: Illustration of two-column recovery.

Two-column recovery is executed in a similar manner to the one-column recovery. Due to space limitation, we omit the algorithm in this paper, but an illustration of one iteration of the two-column recovery algorithm is shown in Fig. 8. In this illustration, columns c_0 and c_{n-1} have been lost. 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, in this case $(0, n-1)$, the order of recovery is known. This is because the recovery sequence is agnostic to the data itself. Thus the order of masks are determined at run-time and based on the ids of the lost columns and the iteration of the recovery process.

6 Evaluation

6.1 Space Complexity

One of the main advantages of X-Cipher is its capability to significantly lower the overheads incurred when using erasure codes for fault-tolerance. 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. 5.1, 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, as described in Sec. 3.2, 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, we compress the entire $m \times n$ data and codewords within a column into a ciphertext. Table 3 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. We can see that the X-Cipher ciphertext can store the provided plaintext data, together with the computed codewords, without incurring a significant overhead. It is almost the same as the plaintext data size, but it is significantly smaller than individually encrypting the provided plaintext data. Finally, our design is based decentralized erasure codes [10]. Storing ciphertexts on independent chunk servers enhances fault-tolerance, load-balancing, and accessibility.

6.2 Computational Complexity

We analyze the computational complexity based on the number of consecutive homomorphic multiplications in the X-Cipher functions; this is also called the *multiplicative depth*. After every homomorphic multiplication, we need to perform the costly noise reduction and relinearization, as described in Sec. 3.1. Hence, it is wise to minimize the multiplicative depth in our algorithm designs. The verification does not require any homomorphic multiplications and therefore has a depth of 0. 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. 5.4. 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. Note, we recommend setting the value of n to 5, which means we will generate 5 ciphertexts stored on 5 separate servers. Also, we can pre-compute the column rotation to speedup the recovery process online.

Table 3: Space complexity for X-Cipher structure

Parameters				
Plaintext slots (ϱ)	64			
Dimension (n)	5	7	11	13
Multiples ($m = \frac{\varrho}{n}$)	12	9	5	4
Total data cells ($mn \times (n-2)$)	180	315	495	572
Size (KB)				
Plaintext	0.72	1.26	1.98	2.28
Ciphertext (X-Cipher)	0.93	1.30	2.05	2.42
Ciphertext (without)	55.8	82.1	112.5	125.7

6.3 Security Analysis

In order to evaluate the security of X-Cipher, we consider the adversary's ($\mathcal{A}dv$) motivations and capabilities, then justify how X-Cipher features counter actions taken by $\mathcal{A}dv$ to achieve their goals. Since $\mathcal{A}dv$ can compromise a chunk server, they may be motivated to attempt to either leak the data/keys or corrupt the data in order to compromise the homomorphic computations.

Firstly, the $\mathcal{A}dv$ may steal the ciphertexts in an effort to leak the secret data which they encrypt. However, the $\mathcal{A}dv$ will not be able to do so due to the secret key (sk) never being transmitted by the user. With access to only the public key (pk) and evaluation key (ek), $\mathcal{A}dv$ can only correctly encrypt plaintext data and compute on encrypted data. As an alternative, $\mathcal{A}dv$ may try to continuously sample the encryption function or perform operations until they output an equal ciphertext. However, this will not be effective since the BGV scheme is a probabilistic encryption algorithm that introduces additional error to produce the ciphertext. As a result, ciphertexts encrypting the same plaintext data are not equal. Thus, $\mathcal{A}dv$ last remaining method to leak the data is to brute force the skvalue, which is computationally infeasible [7].

Secondly, given that $\mathcal{A}dv$ can learn of the X-Cipher structure (values of n and m) by inspecting the algorithms they must execute during the computational phase, the adversary may try to learn of the underlying data by exposing or tampering with the codewords. However, since the codewords are encrypted alongside the data, they are protected in the same way.

Since the $\mathcal{A}dv$ will be unable to leak any of the private data or codewords, they may try to tamper with the data in order to

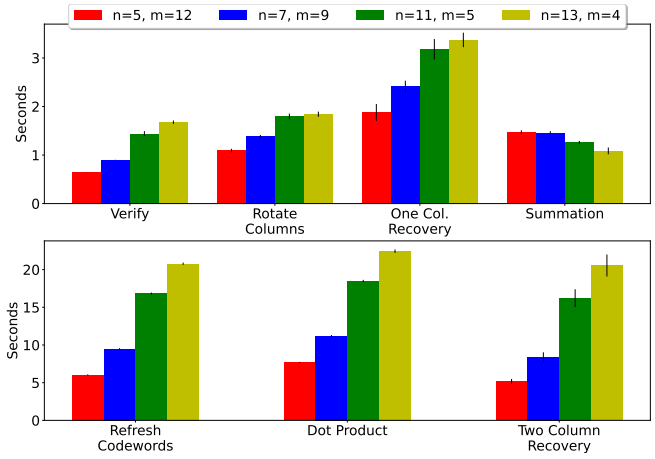


Figure 9: Running time of primitive functions.

prevent homomorphic computations from being correct. Given $\mathcal{A}dv$ can corrupt one cluster server, they can tamper with one column per X-Cipher structure. However no matter the manner in which $\mathcal{A}dv$ tampers with each column, it will always be detected by the user. For instance, if the $\mathcal{A}dv$ tampers with only the codewords, this will with a strong likelihood be detected during the integrity check. This is because $\mathcal{A}dv$ cannot modify the data used to compute the codewords which are accessible on their cluster server. This scenario is also the case for manipulating only the data or both the data and the codewords.

Finally, an $\mathcal{A}dv$ may try to cause a total system failure in order to prevent the user from receiving any of their previously allocated private data. However, because of the homomorphic recovery algorithm in X-Cipher, the user can leverage the remaining untrusted cluster servers to recover the lost column, and continue execution after allocating the recovered column to a new server.

7 Experimental Results

We prototype the proposed X-Cipher structure using the HElib library [15], which implements the BGV HE scheme [7] (see Sec. 3.1). 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 [4]. 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 [15]. The multiplicative depth L of is configured according to what is required by the evaluated circuits in our protocol, for which we derived according to the complexity analysis discussed in Sec. 6.2.

7.1 Primitive Functions

Figure 9 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 efficient execution of many of the primitives. For instance, all primitive functions whose times are in the top graph in Fig. 9 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

Table 4: Application Computation Running Time Statistics

PSI (s)			
ι	$\zeta = \iota/8$	$\zeta = \iota/4$	$\zeta = \iota/2$
25	28.1	54.4	109
50	54.5	109	225
75	81.3	162	332
100	111	224	444

Matrix Operations (ms)		
Dim	+	×
3	24.05	278.5
4	52.25	364.6
5	50.50	450.8
6	54.20	541.5

times grow as the dimension n grows. These functions rely most heavily on the ciphertext rotations and multiplications. This current run-time minimizes data duplication and does not implement parallel execution. We believe 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.

7.2 Real-world applications

To evaluate X-Cipher's effectiveness on real-world applications, we evaluate the running time of two applications: matrix operations and private set intersection (PSI).

Basic matrix operations were tested for square matrices with the dimensions varying from $(3 - 6)$. Table 4 demonstrates the running time required for both matrix operations. This time does not include the time required for code-word regeneration, which is required only after matrix multiplication. These results demonstrate an efficiency and a limitation in-line with typical HE applications: addition can be implemented efficiently whereas multiplication causes the bottleneck.

In addition to evaluating basic matrix arithmetic operations, 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), 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 4 . The times reported represent the time for the sender and receiver to preprocess their data, encode into the X-Cipher structure, and 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 increasing linearly as either the receiver or sender set increase. This demonstrates that with additional parallelization, the execution time can be decreased further.

7.3 Comparison to Related Works

Table 5 shows a comparison to the related works by Tsoutsos et al. [29] and Shen et al. [27] who also proposed solutions for recovering and verifying homomorphic ciphertexts. Our work has a number of advantages over these existing works. Firstly, when considering the plaintext to ciphertext ratio (number of integers which can be put inside a ciphertext), our work utilizes ciphertext packing (as mentioned in Sec. 3.1) to store many plaintext values into a single ciphertext. This not only leads to lower space complexity in comparison with individually encrypting one plaintext value into a ciphertext. Similarly, the computational complexity is reduced because operations applied on the packed ciphertexts

Table 5: Comparison to related works

	Shen [27]	Tsoutsos [29]	This Work
Data per cipher	1	1	ϱ
HE Paradigm	Partial	Partial	Fully
Interaction	Full	No	Partial
Ciphertext storage	Distributed	Local	Distributed
Codeword homomorphic	Multiplicative	Additive	Additive

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. [27], 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

Our approach demonstrates a method to simultaneously achieve data *privacy*, and *fault-tolerance*. We introduce the X-Cipher design which makes ciphertexts recoverable and verifiable in all phases of its use. We introduce a novel approach of encrypting encoded-plaintext alongside codewords for efficient and additively homomorphic recovery capabilities. By evaluating X-Cipher's storage cost, it is determined that through the use of encoding and ciphertext packing, X-Cipher reduces storage overhead. After empirical study shows, it is evident that the timing results show that the primitive functions for verification are efficient. Experiments are conducted which show that useful operations, such as matrix operations and PSI, are possible while maintaining the ability to recover and verify the data. We also demonstrate that even given a cluster server is fully compromised, the data will never be leaked and will always be recoverable with X-Cipher. In future work, we aim to expand upon X-Cipher capabilities by enabling more building block operations and improving the efficiency of primitive operations that rely on multiplication and rotation functions.

References

[1] Asma Aloufi, Peizhao Hu, Hang Liu, Sherman S.M. Chow, and Kim-Kwang Raymond Choo. Universal location referencing and homomorphic evaluation of geospatial query. *Computers Security*, 102:102137, 2021.

[2] Asma Aloufi, Peizhao Hu, Yongsoo Song, and Kristin Lauter. Computing blindfolded on data homomorphically encrypted under multiple keys: An extended survey, 2020.

[3] Jee Hea An and Mihir Bellare. Does encryption with redundancy provide authenticity? In Birgit Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, pages 512–528, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[4] Michael Backes, Pascal Berrang, Matthias Bieg, Roland Eils, Carl Herrmann, Mathias Humbert, and Irina Lehmann. Identifying personal DNA methylation profiles by genotype inference. In *IEEE Symposium on Security and Privacy*, pages 957–976, 2017.

[5] Dan Boneh, Gil Segev, and Brent Waters. Targeted malleability: Homomorphic encryption for restricted computations. Cryptology ePrint Archive, Report 2011/311, 2011. https://eprint.iacr.org/2011/311.

[6] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–13. Springer, 2013.

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[8] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255, 2017.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[10] A.G. Dimakis, V. Prabhakaran, and K. Ramchandran. Decentralized erasure codes for distributed networked storage. *IEEE Transactions on Information Theory*, 52(6):2809–2816, 2006.

[11] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, June 1998.

[12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. Cryptology ePrint Archive, Report 2011/566, 2011. https://eprint.iacr.org/2011/566.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[14] Shai Halevi and Victor Shoup. Algorithms in helib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 554–571, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[15] Shai Halevi and Victor Shoup. Algorithms in helib. In *Annual Cryptology Conference*, pages 554–571. Springer, 2014.

[16] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, Boston, MA, June 2012. USENIX Association.

[17] IDG. Idg's 2020 cloud computing study. https://www.idg.com/tools-for-marketers/2020-cloud-computing-study/, 2020. Accessed: 2021-05-04.

[18] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1209–1222, 2018.

[19] Lihao Xu and J. Bruck. X-code: Mds array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.

[20] Hsiao-Ying Lin and Wen-Guey Tzeng. A secure erasure code-based cloud storage system with secure data forwarding. *IEEE Transactions on Parallel and Distributed Systems*, 23(6):995–1003, 2012.

[21] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[22] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[23] J. Plank. Erasure codes for storage systems: A brief primer. *login Usenix Mag.*, 38, 2013.

[24] James S. Plank. The raid-6 liber8tion code. *The International Journal of High Performance Computing Applications*, 23(3):242–251, 2009.

[25] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of The Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[26] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[27] Shiuan-Tzuo Shen, Hsiao-Ying Lin, and Wen-Guey Tzeng. An effective integrity check scheme for secure erasure code-based storage systems. *IEEE Transactions on reliability*, 64(3):840–851, 2015.

[28] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[29] Nektarios Georgios Tsoutsos and Michail Maniatakos. Efficient detection for malicious and random errors in additive encrypted computation. *IEEE Transactions on Computers*, 67(1):16–31, 2017.

[30] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.

[31] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59:56–65, 2016.