A Secure and Fast Dispersal Storage Scheme Based on the Learning with Errors Problem [⋆]

Ling Yang[1,2,3], Fuyang Fang[1,2,3], Xianhui Lu[1,2⋆⋆], Wen-Tao Zhu[1,2], Qiongxiao Wang[1,2], Shen Yan[1,2,3], and Shiran Pan[1,2,3]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China

[2] Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China

[3] University of Chinese Academy of Sciences, Beijing, China

{yangling,fangfuyang,luxianhui}@iie.ac.cn wtzhu@ieee.org

{wangqiongxiao,yanshen,panshiran}@iie.ac.cn

Abstract. Data confidentiality and availability are of primary concern in data storage. Dispersal storage schemes achieve these two security properties by transforming the data into multiple codewords and dispersing them across multiple storage servers. Existing schemes achieve confidentiality and availability by various cryptographic and coding algorithms, but only under the assumption that an adversary cannot obtain more than a certain number of codewords. Meanwhile existing schemes are designed for storing archives. In this paper, we propose a novel dispersal storage scheme based on the learning with errors problem, known as storage with errors (SWE). SWE can resist even more powerful adversaries. Besides, SWE favorably supports dynamic data operations that are both efficient and secure, which is more practical for cloud storage. Furthermore, SWE achieves security at relatively low computational overhead, but the same storage cost compared with the state of the art. We also develop a prototype to validate and evaluate SWE. Analysis and experiments show that with proper configurations, SWE outperforms existing schemes in encoding/decoding speed.

Key words: dispersal storage, data confidentiality, data availability, dynamic data operations, the learning with errors problem

1 Introduction

Data is wealth for both individuals and companies. Guaranteeing data confidentiality and availability are of primary concern in data storage [1, 2, 3]. Because of

⋆ This work was supported by the National 973 Program of China under Grant 2014CB340603, and the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDA06010702. This work was also supported by the National Natural Science Foundation of China under Grants 61272479, 61272040, and 61572495.

⋆⋆ This author is the corresponding author.

vulnerabilities in storage software, unreliability of disk drives and so on, storing data in users' personal computers cannot absolutely guarantee data confidentiality and availability [4]. Thus more and more users resort to cloud storage. However, cloud storage still cannot absolutely guarantee confidentiality and availability [5]. Amazon S3 suffered seven-hour downtime in 2008 [6]. Facebook leaked users' contact information in 2013 [7]. iCloud leaked users' private information in 2014 [8]. All similar incidents lead users to focus on designing storage systems which can provide data confidentiality and availability [3]. Dispersal storage schemes are techniques to guarantee these two security properties.

Existing dispersal storage schemes are (k, n) threshold schemes, where k is the threshold [3]. In these schemes, data is transformed into n related codewords. Then the codewords are stored in separate storage servers, which belong to different administrative and physical domains. Meanwhile, storage servers can belong to either the same service provider or (more favorably) different providers, respectively. Even if $(n - k)$ out of the n codewords are corrupted or completely unavailable, the data can still be recovered. With fewer than k out of the n codewords, no information of the data can be obtained.

In POTSHARDS [9], Shamir's secret sharing algorithm [10] is used as the dispersal scheme. Shamir's algorithm [10] achieves *information-theoretical security*. However, the storage overhead of Shamir's algorithm is n times of the data, and the encoding time linearly grows with $n \times k$ [3]. Compared with Shamir's algorithm, Rabin's IDA [11] improves encoding/decoding speed, and saves storage overhead. However, data confidentiality of Rabin's IDA is far less. SSMS [12] provides a *computationally secure* dispersal scheme. As far as we know, AONT-RS [3] is the best scheme that achieves balance between confidentiality and data processing performance. Meanwhile, data integrity of AONT-RS is protected.

Existing schemes [10, 11, 12, 3] achieves a different level of data confidentiality with different performance and storage overhead. However, existing schemes still suffer several problems. First, existing schemes achieve data confidentiality under the assumption that an adversary cannot obtain more than $(k - 1)$ out of the n codewords. However, this assumption is unsuitable for some dispersal storage scenarios, such as a user setting one login password for all storage servers, public cloud storage, etc. In those scenarios, an adversary can easily obtain k out of the n codewords and then recover users' data. Second, existing schemes [10, 12, 3] are designed for storing archives or static data. In these schemes, while executing dynamic operations, users are required to download and decode corresponding codewords, which is inefficient. However, data is frequently updated by users, especially in cloud storage [13, 14]. Thus, existing schemes may be unsuitable for many storage scenarios.

In this paper, we propose a novel computationally secure dispersal storage scheme based on the learning with errors (LWE) problem [15], called storage with errors (SWE). To the best of our knowledge, SWE is the first work that applies LWE into dispersal storage schemes. The key idea of SWE is reforming LWE to meet the requirements of dispersal storage. Then we set the secret information in LWE to be the stored data. After the data is processed, we can utilize k

out of the n codewords to recover the data, which guarantees data availability. The hardness of LWE guarantees data confidentiality. Meanwhile, the number theoretic transform is applied to optimize arithmetic operations in SWE.

The merits of SWE are three-fold: **(i)** Analysis shows that under our assumptions, SWE with the same storage as the state of the art, achieves higher confidentiality than existing schemes. In SWE, even though an adversary obtains all the codewords, it still cannot recover the data. **(ii)** As SWE has the *additive homomorphic* property and each column of the codeword is independent of others, SWE favorably supports efficient and secure dynamic data operations (i.e., modifying, deleting, and appending). **(iii)** With proper configurations, SWE outperforms the state of the art in encoding/decoding speed.

The rest of this paper is organized as follows. Technical background is introduced in Section 2. We then present preliminaries in Section 3. We detail design and analysis of SWE in Sections 4 and 5, respectively. Experiments and evaluations are shown in Section 6. Section 7 presents the conclusion.

2 Technical Background

2.1 Dispersal Algorithms

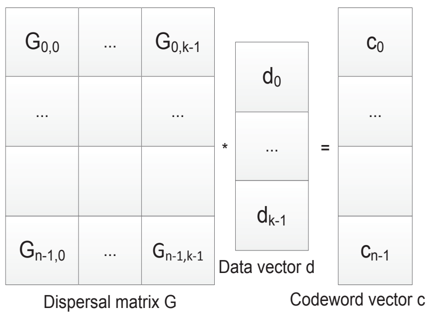


Fig. 1. The key part of dispersal schemes.

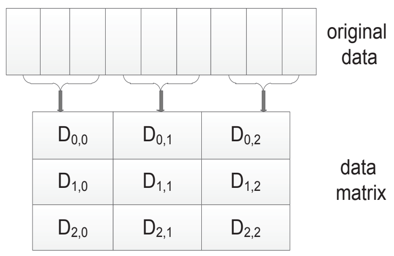


Fig. 2. An example of generating data matrix from original data.

In (k, n) threshold dispersal storage schemes, $\frac{n-k}{n}$ represents the *fault-tolerance ability* (i.e., even though any $(n - k)$ out of the n codewords are corrupted or completely unavailable, the data can still be recovered). The key part of dispersal schemes shown in Fig. 1 is also called the *dispersal algorithm* (DA). Here, G is the dispersal or generator matrix, which is an $n \times k$ integer matrix. G is public. $\mathbf{d}$ is the data vector with k elements. $\mathbf{c}$ is the codeword vector with n elements, and $\mathbf{c} = \mathbf{G} \times \mathbf{d}$. Elements of $\mathbf{d}$ or $\mathbf{c}$ are integers. Each element of $\mathbf{c}$ is stored in a different storage server. Thus even if $(n - k)$ storage servers are out of service, $\mathbf{d}$ can still be reconstructed. Because of the distributed property of storage servers, dispersal storage schemes can survive from non-security-related events (e.g., power failure, water damage).

The dispersal matrix $\mathbf{G}$ should satisfy that any k rows of $\mathbf{G}$ can constitute an invertible matrix. Thus we can reconstruct $\mathbf{d}$ from any k intact codewords. When reconstructing $\mathbf{d}$, if the i-th codeword is chosen, the i-th row of $\mathbf{G}$ should be chosen. Let $\mathbf{c'}$ be the vector which has k codewords, $\mathbf{G'}$ be the corresponding $k \times k$ sub-matrix of $\mathbf{G}$. Then the invertible matrix of $\mathbf{G'}$ multiplies $\mathbf{c'}$ yield $\mathbf{d}$.

2.2 The Learning with Errors (LWE) Problem

The learning with errors (LWE) problem is a generalization of the famous "learning parity with noise" problem to larger modulus. The most attractive feature of LWE is the connection of worst-to-average-case [15, 16, 17, 18]. LWE has two forms [15]: *search-LWE* and *decision-LWE*. We apply *search-LWE* in SWE.

Definition 1. (search-LWE [15].) *Let $n, k \geq 1$ be integers, q be an integer and $q = q(k) \geq 2$, χ be a distribution which can be Gaussian-like distribution or uniform distribution on $\mathcal{Z}_q$, and $\mathbf{x} \in \mathcal{Z}_q^k$ be the secret information. We donate by $\mathcal{L}_{x,\chi}^k$ the probability distribution on $\mathcal{Z}_q^{n\times k} \times \mathcal{Z}_q^n$ obtained by choosing $\mathbf{A} \in \mathcal{Z}_q^{n\times k}$ uniformly at random, choosing the "noise" or errors vector $\mathbf{e} \in \mathcal{Z}_q^n$ according to χ^n, and returning $(\mathbf{A}, \mathbf{Ax} + \mathbf{e}) = (\mathbf{A}, \mathbf{c}) \in \mathcal{Z}_q^{n\times k} \times \mathcal{Z}_q^n$. search-LWE is the problem of recovering $\mathbf{x}$ from $(\mathbf{A}, \mathbf{c}) \in \mathcal{Z}_q^{n\times k} \times \mathcal{Z}_q^n$ sampled according to $\mathcal{L}_{x,\chi}^k$.*

Döttling et al. [16] give the first work that applies uniform error-distribution (i.e., error-distribution is $\mathcal{U}[-r, r]$) to LWE. This work proves that instances of LWE with uniform errors are as hard as lattice problems. Furthermore, the result of [16] shows that the *matrix-version of LWE* (i.e., $A \in \mathcal{Z}_q^{n\times k}$, $X \in \mathcal{Z}_q^{k\times l}$, and $E \in \mathcal{U}[-r, r]^{n\times l}$) is also hard.

2.3 Related Work

Next, we introduce some dispersal storage schemes that are most related to our work, which are summarized in Table 1.

Shamir's algorithm [10] can guarantee information-theoretic security of data. However, the storage overhead of Shamir's algorithm is n times of the data. Besides, encoding/decoding speed of Shamir's algorithm is slow. Compared with Shamir's algorithm, SWE improves data processing performance, and reduces storage overhead.

In Rabin's IDA [11], the *non-systematic erasure code* (e.g., the dispersal matrix is the Vandermonde matrix [19]) is applied to disperse data to achieve availability. Rabin's IDA saves time and reduces storage overhead to n/k times of the data. However, data confidentiality of Rabin's IDA is far less and would be unacceptable in many storage scenarios [3]. Compared with Rabin's IDA, SWE achieves higher confidentiality.

In SSMS [12], a symmetric cryptographic algorithm is applied to encrypt data. Then the ciphertext is dispersed using a erasure code. The key used in the cryptographic algorithm is dispersed using Shamir's algorithm. Bessani et al. [20] apply the idea of SSMS to propose the first cloud-of-clouds storage application.

AONT-RS [3] is the backbone dispersal algorithm of a well-known storage company, Cleversafe. In AONT-RS, a variant of All-Or-Nothing Transform (AONT) is applied to achieve confidentiality. Meanwhile, the *systematic Reed-Solomon code* (RS) [19] (i.e., the first k rows of the dispersal matrix compose a $k \times k$ identity matrix) is applied to achieve availability. Besides, an extra word, "canary", which has a known and fixed value, is applied to check integrity of the data when it is decoded. SSMS, Rabin's IDA, and AONT-RS have approximate the same storage overhead. Compared with AONT-RS, SWE with proper configurations is faster than AONT-RS in encoding/decoding speed.

Furthermore existing schemes [10, 11, 12, 3] guarantee data confidentiality under the assumption that an adversary cannot obtain more than $(k-1)$ out of n codewords. However, SWE can guarantee data confidentiality even though an adversary obtains all the codewords. Besides, SWE favorably supports efficient and secure dynamic data operations, which is more practical in cloud storage.

Table 1. Comparison with existing schemes. "IS" represents information-theoretical security. "CS" represents computational security. "b" represents data size.

Schemes	Security	Techniques	Storage
Shamir's algorithm [10]	IS	Shamir's secret sharing	nb
Rabin's IDA [11]	IS	non-systematic erasure code	$\frac{nb}{k}$
SSMS [12]	CS	encryption algorithms & non-systematic erasure codes	$\frac{nb}{k}$
AONT-RS [3]	CS	AONT & systematic Reed-Solomon code	$\frac{nb}{k}$
SWE	CS	LWE	$\frac{nb}{k}$

3 Preliminaries

3.1 Notation

- $\mathcal{Z}_q$: The mathematic structure of group. q is a prime.
- $x \leftarrow \mathcal{Y}$: x is independently, randomly, and uniformly chosen from the distribution or set $\mathcal{Y}$.
- $\mathbf{A}$: The dispersal matrix of SWE which is an $n \times k$ integer matrix. $\mathbf{A}_{i,j} \leftarrow \mathcal{Z}_q$.
- $\mathbf{D}$: The data matrix which represents the data to be stored. D is a $k \times l$ integer matrix. $\mathbf{D} \in \mathcal{Z}_q^{k \times l}$.
- $f(s,r)$:The pseudorandom function whose inputs are s and r. Each outputs of $f(s,r)$ is independently, randomly, and uniformly chosen from the uniform distribution $\mathcal{U}[-r,r]$.
- $\mathbf{E}$: The error matrix of SWE. which is just like the matrix $\mathbf{E}$ in LWE. $\mathbf{E}$ is generated by using $f(s,r)$ and thus $\mathbf{E}_{i,j} \leftarrow \mathcal{U}[-r,r]$.
- $\mathbf{C}$: The codewords matrix which is the result of $(\mathbf{A} \times \mathbf{D} + \mathbf{E}) \bmod q$.

– q_x: The prime which is the smallest prime bigger than 2^x. For examples, $q_{2048} = 2^{2048} + 981$, $q_{1024} = 2^{1024} + 643$, $q_{768} = 2^{768} + 183$, and so on.

3.2 System Model and Design Goals

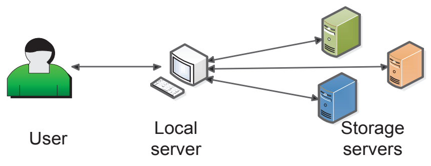


Fig. 3. The system model of SWE.

Fig. 3 shows the system model of SWE. There are three entities in SWE: user, local server, and storage server.

– User: An entity that applies SWE to store data. A user can be a person, a proxy, a storage gateway and so on.
– Local server: Each user has its own local server and stores s, r, q, n, k, l, $\mathbf{A}$, and so on in it. Among these, only s should be securely stored. Users calculate $\mathbf{C}$, recover $\mathbf{D}$, execute dynamic operations and so on on local servers.
– Storage server: An entity which provides storage and computing service. All codewords $\mathbf{C}$ are stored in storage servers. Different storage servers belong to different administrative and physical domains. Storage servers can belong to either the same service provider or different providers, respectively.

We assume that storage servers are honest but curious. Thus, storage servers should honestly execute the operations which are authorized by users. However, storage servers may pry into users' data. Meanwhile, an adversary can obtain all the codewords, but it cannot obtain s which is kept secret by users.

Existing schemes guarantee data confidentiality under the assumptions that an adversary cannot obtain more than $(k-1)$ out of the n codewords, and storage servers will not pry into users' data. However, in practical storage scenarios, data may be stored in public clouds. A user may set one login password for all storage servers. Moreover, storage vendors may collude to pry into users' data. In those scenarios, obtaining more than $(k-1)$ out of the n codewords is feasible for an adversary. As long as an adversary obtains k out of the n codewords, it can recover users' data. Hence, our assumptions are more practical than those of existing schemes.

Based on the above assumptions, we design SWE with the following goals.

– Data confidentiality: SWE can guarantee that even if an adversary obtains all the codewords, it still cannot recover users' data.
– Data availability: SWE can guarantee that even if any $(n-k)$ out of the n codewords are corrupted or unavailable, the data can still be reconstructed.

– Efficient and secure dynamic data operations: While executing dynamic data operations, users are not required to download or decode codewords. An adversary cannot recover users' data from dynamic operations.
– Efficient data processing: With proper configurations, SWE outperforms the state of the art in encoding/decoding speed.

4 A New Dispersal Scheme: SWE

Existing schemes [10, 11, 12, 3] achieve confidentiality under the assumption that an adversary cannot obtain more than $(k-1)$ out of the n codewords. Some schemes achieve high confidentiality but require too many storage and computational costs. Although some schemes reduce storage and improve encoding/decoding speed, these schemes achieve relatively weak confidentiality. Meanwhile, existing schemes are designed for archives storage, which is inefficient in dynamic operations. Meanwhile, trival applying those schemes in dynamic data operations also leak users' data information. To deal with these limitations, we propose a novel dispersal storage scheme, SWE. SWE can achieve higher confidentiality, availability, and fast encoding/decoding speed. Meanwhile, SWE favorably supports efficient and secure dynamic data operations.

4.1 The Basic Scheme

Technical highlights: In order to build a secure and efficient dispersal storage scheme, we reform LWE and make it meet the requirements of dispersal storage. We use the *matrix-version of LWE* (i.e., $\mathbf{A} \in \mathcal{Z}_q^{n\times k}$, $\mathbf{X} \in \mathcal{Z}_q^{k\times l}$, and $\mathbf{E} \in \mathcal{U}[-r,r]^{n\times l}$)[16] to construct SWE. We consider $\mathbf{X}$ in LWE to be the data matrix $\mathbf{D}$, $\mathbf{A}$ to be the dispersal matrix, and the result of $(\mathbf{A}\times\mathbf{D}+\mathbf{E}) \bmod q$ to be the codewords $\mathbf{C}$. Because of the hardness of LWE, it is difficult for an adversary to recover $\mathbf{D}$ by obtaining $\mathbf{A}$, $\mathbf{C}$, r, and q. As $\mathbf{A}$ is an $n\times k$ matrix and $n > k$, we can use any k rows of $\mathbf{C}$ to recover $\mathbf{D}$ and thus SWE achieves availability. We also use uniform errors (i.e., $\mathbf{E}_{i,j} \leftarrow \mathcal{U}[-r,r]$), small dimension $\mathbf{A}$, exponential r and q, and the number theoretic transform and its inverse to build the secure and efficient scheme.

We do not simply use erasure codes and a stream cypher to generate $\mathbf{E}$ to build a computationally secure dispersal scheme. Because such trivial method requires that $\mathbf{E}_{i,j}$ is no shorter than $(\mathbf{A}\times\mathbf{D})_{i,j}$. Obviously, the bit size $\log r$ of $\mathbf{E}_{i,j}$ in LWE is shorter than the bit size $\log q$ of $(\mathbf{A}\times\mathbf{D})_{i,j}$, which saves the time of generating $\mathbf{E}$.

SWE consists of three phases: **(i)** setup, **(ii)** computing the codewords, called *encoding*, and **(iii)** recovering the data, called *decoding.*

4.1.1 Setup

During setup, proper parameters of SWE should be chosen. Then, the dispersal matrix $\mathbf{A}$, the data matrix $\mathbf{D}$, and the error matrix $\mathbf{E}$ are generated.

Table 2. Examples of parameters in LWE and SWE.

Schemes	k	q	r
LWE	200	2^{35}	2^{15}
SWE	10	q_{768}	$\lceil\sqrt{q_{768}}\rceil$

Choose parameters: As $\mathbf{A} \in \mathcal{Z}_q^{n\times k}$ and $\mathbf{E} \in \mathcal{U}[-r,r]^{n\times l}$ is generated by using $f(s,r)$, parameters n, k, q, r, and s should be chosen. Arbitrarily setting parameters of SWE will lead to an insecure dispersal scheme. Directly applying common parameters of LWE into SWE may not be able to achieve data availability or efficient encoding/decoding. Thus, when users choose parameters of SWE, both security and performance should be taken into consideration.

$\frac{n-k}{n}$ represents the fault-tolerance ability. Larger $\frac{n-k}{n}$ means higher availability and more storage overhead. For SWE, the storage overhead of $\mathbf{C}$ is $\frac{n}{k}$ times of that of $\mathbf{D}$. However, higher fault-tolerance also means that an adversary needs to compromise fewer storage servers to recover users' data, which leads to relatively weaker security. Therefore users should set $\frac{n-k}{n}$ according to the demand of availability and security.

Common instances of LWE such as LWE shown in Table 2 cannot satisfy that any k rows of $\mathbf{A}$ can constitute an invertible matrix. Therefore, we cannot use any k out of the n codewords to reconstruct the data and thus SWE cannot guarantee data availability. However, if q is a large integer, randomly generating $\mathbf{A}$ can meet the demand of the dispersal matrix. Besides, in order to compute multiplicative inverse numbers in decoding, users should set q to be a prime too. Thus, q should be a large prime in SWE.

Meanwhile, r and s also affect the security of SWE. If r and s are too small, an adversary can easily recover $\mathbf{E}$ by brute force attacks, and then recover the data. Hence the probability of successful guessing r and s should be negligible. In SWE, $r = \lceil\sqrt{q}\rceil$. s is a random string, whose length is longer than 256 bits.

Directly increasing q and r of common instances of LWE can develop secure dispersal schemes, but these schemes are inefficient and thus impractical. Such inefficient schemes cannot be applied to performance-sensitive large data storage scenarios. For the sake of security and high-performance, we should set n and k to be small integers such as the example shown in Table 2.

After proper parameters are chosen, $\mathbf{A}$, $\mathbf{D}$, and $\mathbf{E}$ are generated as follows.

Generate A: $\mathbf{A}_{i,j} \leftarrow \mathcal{Z}_q$. In SWE, as the smallest q is q_{768}, any k rows of $\mathbf{A}$ can constitute an invertible matrix.

Theorem 1. *If q is a large prime, $\mathbf{A} \in \mathcal{Z}_q^{n\times k}$ and $n > k$, then any k rows of $\mathbf{A}$ constitute an invertible matrix.*

Proof. We calculate the invertibility probability of a $k \times k$ sub-matrix of $A \in \mathcal{Z}_q^{n\times k}$. Specifically, the first vector of A can be any nonzero vector, of which there are $q^k - 1$ (i.e., $q^k - q^0$) choices. The second vector can be chosen from $q^k - q^1$, etc. Hence, the number of ways to choose vectors of A is

$$(q^k - q^0)(q^k - q^1)...(q^k - q^{n-1}) = \prod_{i=1}^{n}(q^k - q^{i-1}).$$

So, the probability that any $k \times k$ sub-matrix of A is invertible is

$$\frac{\prod_{i=1}^{n}(q^k - q^{i-1})}{q^{nk}} = \prod_{i=1}^{n}(1 - q^{i-1-k}) < 1 - \frac{1}{q}.$$

As q is a large prime, the invertibility probability is extremely close to 1. Therefore, we always suppose that any k rows of A constitute an invertible matrix.
$\square$

Generate D: $\mathbf{D} \in \mathcal{Z}_q^{k\times l}$ is generated from the original data. Because of $q = q_x$, each element of $\mathbf{D}$ is transformed from corresponding x bits of the data. For example, if $q = q_{1024}$, $D_{i,j}$ is transformed from corresponding 1024 bits of the original data. Users generate the first column of $\mathbf{D}$, then the second column and so on. Fig. 2 illustrates an example of generating $\mathbf{D}$. If the last column of $\mathbf{D}$ has fewer than k elements, the remaining elements can be filled with 0. For the sake of confidentiality, if the entropy of $\mathbf{D}_{i,j}$ is too low, users can add some random bits in $\mathbf{D}_{i,j}$.

Generate E: In order to build an efficient scheme, we apply uniform errors rather than Gaussian errors in SWE. $\mathbf{E}$ is generated by using $f(s, r)$ and thus $\mathbf{E}_{i,j} \leftarrow \mathcal{U}[-r, r]$. For the sake of confidentiality, when encoding different data matrices, users should apply $f(\cdot)$ to generate new errors matrices. The same inputs of $f(\cdot)$ can generate the same $\mathbf{E}$.

In practical, $\mathbf{A}$, n, k, q, and r can be public. $\mathbf{E}$ should be kept secret. However, users do not need to securely store $\mathbf{E}$ in their local servers. As $\mathbf{E}$ is generated by using $f(s, r)$, only s should be kept secret. In practical, if a user does not want to securely store s, the user can assign a unique identifier to each of its data files, called *file-id*. Then the user chooses a strong password. Subsequently the user combines the password and the *file-id* of corresponding $\mathbf{D}$ as inputs of a pseudorandom function. The output of the pseudorandom function is s. Thus even if the user stores many files, the user only needs to remember the password.

In many storage scenarios (e.g., cloud storage), securing storage of encryption keys (e.g., s in SWE) is not a notoriously difficult problem. In these scenarios, we do not want to keep secret of data for a very long lifetimes or periods of decades (actually, we intend to design a storage system for dynamic data). Thus, an adversary cannot recover users' data by waiting for cryptanalysis techniques to catch up the encryption algorithm. Meanwhile, storing s or encryption keys in a user's local server also saves the storage overhead compared with AONT-RS and SSMS.

4.1.2 Encoding

After $\mathbf{A}$, $\mathbf{D}$, and $\mathbf{E}$ are generated, encoding is calculating $(\mathbf{A} \times \mathbf{D} + \mathbf{E}) \bmod q$ to get the codewords $\mathbf{C}$. Fig. 4 shows an example of encoding, where $n = 5$, $k = 3$. Specifically, $\mathbf{C}_{i,j} = (\sum_{x=0}^{k-1} \mathbf{A}_{i,x} \times \mathbf{D}_{x,j} + \mathbf{E}_{i,j}) \bmod q$.

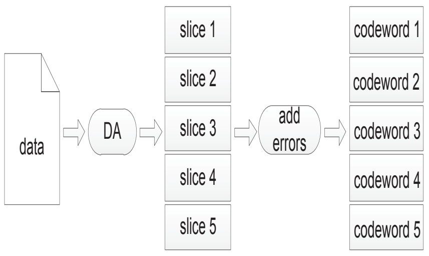


Fig. 4. An example of encoding in SWE.

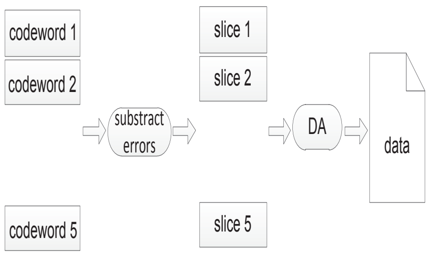


Fig. 5. An example of decoding in SWE.

As $\mathbf{A}_{i,j}$ and $\mathbf{D}_{i,j}$ are all large integers, the complexity of conventional multiplication is $O(m^2)$, where $m = max\{|\mathbf{A}_{i,j}|, |\mathbf{D}_{i,j}|\}/w$ ($|\cdot|$ represents the length of $\cdot$ and w is the data bus width). The number theoretic transform can reduce the complexity to $O(m \log m)$. So, when implementing SWE, we use the number theoretic transform and its inverse to optimize encoding/decoding speed.

After generating $\mathbf{C}$, users should store $\mathbf{C}$ in distributed storage servers based on their storage strategies. For example, if $n = 4$, $k = 3$, and $l = 5$, a user stores the four rows of $\mathbf{C}$ in Amazon S3, Windows Azure, Cleversafe, and Oceanstore, respectively. Even if one of the cloud storage is out of service, the user can recover $\mathbf{D}$ using the remaining codewords. SWE can also be used to protect data avoiding suffering vendor lock-in [21]. Furthermore, users can take full advantage of physically distributed storage servers to achieve better service (e.g., achieving codewords from nearer storage servers reduces downloading time).

4.1.3 Decoding

As any k rows of $\mathbf{A}$ can constitute an invertible matrix, we can recover the data $\mathbf{D}$ using any k intact out of the n codewords. Decoding is recovering $\mathbf{D}$ using k intact out of the n codewords of $\mathbf{C}$, the corresponding sub-matrix of $\mathbf{A}$, and the corresponding sub-matrix of $\mathbf{E}$. Fig. 5 shows an example of decoding, where $n = 5$ and $k = 3$. A user downloads $\mathbf{C}_{0,0}$, $\mathbf{C}_{1,0}$, and $\mathbf{C}_{4,0}$ from storage servers. The user generates $\mathbf{E}_{0,0}$, $\mathbf{E}_{1,0}$, and $\mathbf{E}_{4,0}$ by using $f(s,r)$. Then, the user solves the following liner congruential equations with single modulus q to recover $\mathbf{D}$.

$$\begin{cases} (\mathbf{A}_{0,0}\mathbf{D}_{0,0} + \mathbf{A}_{0,1}\mathbf{D}_{1,0} + \mathbf{A}_{0,2}\mathbf{D}_{2,0} + \mathbf{E}_{0,0}) \bmod q = \mathbf{C}_{0,0} \\ (\mathbf{A}_{1,0}\mathbf{D}_{0,0} + \mathbf{A}_{1,1}\mathbf{D}_{1,0} + \mathbf{A}_{1,2}\mathbf{D}_{2,0} + \mathbf{E}_{1,0}) \bmod q = \mathbf{C}_{1,0} \\ (\mathbf{A}_{4,0}\mathbf{D}_{0,0} + \mathbf{A}_{4,1}\mathbf{D}_{1,0} + \mathbf{A}_{4,2}\mathbf{D}_{2,0} + \mathbf{E}_{4,0}) \bmod q = \mathbf{C}_{4,0} \end{cases} .$$

If the user applies the Gaussian elimination to solve the above equations, it can recover $\mathbf{D}$, but this method is inefficient. In SWE, we store the invertible sub-matrix of $\mathbf{A}$ corresponding to commonly used codewords in local servers, $\hat{\mathbf{A}}$. Let $\tilde{\mathbf{C}}$ be the codewords, $\tilde{\mathbf{E}}$ be the corresponding sub-matrix of $\mathbf{E}$. Thus, decoding is calculating $\mathbf{D}_{i,j} = (\sum_{x=0}^{k-1} \hat{\mathbf{A}}_{i,x} \times (\tilde{\mathbf{C}}_{x,j} - \tilde{\mathbf{E}}_{i,j})) \bmod q$.

Integrity checking: In AONT-RS, an extra word of data, "canary" , which has a fixed value, allows users check the integrity of the data when it is decoded.

In SWE, users do not need to add such a fixed value word. In SWE, users only need to retrieve $(k+1)$ codewords and execute two times of decoding using different k codewords. If the two decoding results are the same, the data is intact and thus we check the integrity of the data.

4.2 Support for Dynamic Operations

Table 3. The procedure of modifying data in SWE.

User		Storage Servers
1. Generate $\mathbf{d'_j}$.		
2. Generate $\mathbf{e'_j}$ using $f(\cdot)$.		
3. Compute $\mathbf{c'_j} =$ $(\mathbf{A}\mathbf{d'_j} + (\mathbf{e'_j} - \mathbf{e_j}))$ mod q.		
	$\xrightarrow[modify\ \mathbf{d_j}]{\mathbf{c'_j}}$	
		4. Compute $\tilde{\mathbf{c}}_\mathbf{j} =$ $(\mathbf{c'_j} + \mathbf{c_j})$ mod q.

So far, we assume that users' data is static. However, data is frequently updated by users especially in cloud storage scenarios [13, 14]. Therefore, the investigations on dynamic data operations are also of paramount importance.

When a user changes a file, we suppose the user changes the *file-id* to a new one. Then the user combines the new *file-id* and its password as s'. Then the user utilizes $f(s', r)$ generating $\mathbf{e'}$. If the entropy of the data in dynamic operations is low, for the sake of confidentiality, the user can add some random bits in it. Then the user execute corresponding dynamic operations.

Modifying: Suppose that a user wants to modify $\mathbf{D}_{i,j}$ to $\mathbf{D}_{i,j} + m$. The user only needs to modify the $(j+1)$-th column of $\mathbf{C}$. Let $\mathbf{d_j}$ be the $(j+1)$-th column of $\mathbf{D}$, $\mathbf{c_j}$ be the $(j+1)$-th column of $\mathbf{C}$, and $\mathbf{e_j}$ be the $(j+1)$-th column of $\mathbf{E}$. Table 3 illustrates the operations. Specifically, **(i)** The user generates the new vector $\mathbf{d'_j}$ in which the $(i+1)$-th element is m and other elements are 0. **(ii)** The user utilizes $f(s', r)$ to generate the error vector $\mathbf{e'_j}$. **(iii)** The user calculates $\mathbf{c'_j} = (\mathbf{A} \times \mathbf{d'_j} + (\mathbf{e'_j} - \mathbf{e_j}))$ mod q. **(iv)** The user stores elements of $\mathbf{c'_j}$ to corresponding storage servers and authorizes storage vendors to do the addition operation. **(v)** Storage vendors compute $\tilde{\mathbf{c}}_\mathbf{j} = (\mathbf{c'_j} + \mathbf{c_j})$ mod q.

As an adversary cannot reconstruct $\mathbf{e'_j} - \mathbf{e_j}$, it cannot recover $\mathbf{D}$ or $\mathbf{d'_j}$ from $\mathbf{c'_j}$ and $\tilde{\mathbf{c}}_\mathbf{j}$. When decoding, the user can recover $\mathbf{D}_{i,j} + m$ with $\mathbf{A}$, $\mathbf{e'_j}$, and $\tilde{\mathbf{c}}_\mathbf{j}$. Subtraction and multiplication can use the same way mentioned above.

Deleting: This operation can be divided into two groups: **(i)** Deleting a whole column of $\mathbf{D}$. As each column of the codewords $\mathbf{C}$ is independent of others, users only need to delete the column of $\mathbf{C}$ corresponding to the column of the deleted data. **(ii)** Deleting some elements of $\mathbf{D}$ in a column. Users apply the modifying operation by setting deleted numbers to corresponding positions.

Appending: When users want to append data to $\mathbf{D}$, they do not need to change the codewords of $\mathbf{D}$. They should generate the new data matrix $\mathbf{D'}$

which is transformed from the appending data. Then they generate the new error matrix $\mathbf{E'}$ using $f(\cdot)$, and then they encode $\mathbf{D'}$ to get the corresponding codewords $\mathbf{C'}$. At last, users store $\mathbf{C'}$ to corresponding storage servers.

In a word, SWE favorably supports efficient and secure dynamic data operations. The reasons are as follows. **(i)** As SWE has the *additive homomorphic* property, the addition of $\mathbf{C}$ equals the addition of $\mathbf{D}$. Therefore, users are not required to download or decode $\mathbf{C}$ while executing dynamic data operations. **(ii)** Each column of $\mathbf{C}$ is independent of others. Therefore, users only need to change the columns of $\mathbf{C}$ corresponding to the changed columns of $\mathbf{D}$. **(iii)** If $\mathbf{C'}$ is the codeword generated in dynamic operations and $\mathbf{C}$ is the original codewords, an adversary cannot recover users' data from $\mathbf{C'}$ or $\mathbf{C}$ due to the hardness of LWE.

5 Scheme Analysis

In this section, we analyze SWE and comparable schemes form confidentiality, availability, and performance. As SSMS [12] does not specify a dispersal or encryption algorithm, we do not include it for the comparison.

5.1 Confidentiality

In SWE, we assume that different storage servers belong to different administrative and physical domains. Storage servers are honest but curious. An adversary can obtain all n codewords, except for s. We consider the attacking scenario that an adversary possesses some codewords and wants to verify whether the data that it encodes matches some predetermined value. Furthermore, if an adversary can verify that one element of $\mathbf{D}$ matches, then the adversary can be assured that the rest matches. Although this scenario is generous, many realistic attacking scenarios can be reduced to this one [3].

Shamir's algorithm [10]: If an adversary obtains k codewords, it can recover the data by solving the k liner equations. If an adversary obtains fewer than k codewords, it cannot discover any information of the data. Suppose that $\mathbf{d}$ is the data whose length is w bits. Each codewords is no shorter than w bits. If an adversary obtains $(k-1)$ codewords, it has to enumerate 2^w times to generate the k-th codewords. Every possible value of $\mathbf{d}$ is equal. Thus, Shamir's algorithm achieves information theoretic security.

Rabin's IDA [11]: In this dispersal scheme, the data is dispersed by the *non-systematic Reed-Solomon code*. Specifically, $\mathbf{c} = \mathbf{G} \times \mathbf{d}$, where $\mathbf{G}$ is the Vandermonde matrix and is public. Hence, there is no randomness in the codewords. Even if an adversary obtains one element of $\mathbf{c}$, it can find some information of $\mathbf{d}$ by verifying that a codeword has a predetermined value.

AONT-RS [3]: The variant of AONT (i.e., the AONT utilizes AES-256 as "generator" and SHA-256 as "hash function") is applied in AONT-RS to guarantee data confidentiality. If an adversary obtains k codewords, it can compute the hash of encrypted data, h. Since the last element of the AONT package is

$key \oplus h$, the adversary can recover the encryption key by computing $key \oplus h \oplus h$. Then the data can be recovered. If an adversary obtains fewer than k codewords, in order to find some information of the data, it has to enumerate 2^{256} times.

SWE: We analyze the confidentiality of SWE by using the brute force attack and the Arora-Ge algorithm[22].

(i) *Scenario 1: An adversary knows $f(\cdot)$ (i.e., the function of generating errors $\mathbf{e}$).* We utilize the brute force attack to analyze the confidentiality of SWE. If an adversary obtains k codewords, it has to figure out s to recover the data $\mathbf{d}$. Let z be the length of s. Hence, the adversary has to enumerate 2^z times to recover $\mathbf{d}$. In SWE, s is longer than 256 bits. If an adversary obtains fewer than k codewords, it cannot find any information of $\mathbf{d}$. For example, an adversary obtains $(k-1)$ codewords. In order to recover $\mathbf{d}$, it has to enumerate at least $2^{z+q_{768}}$ times.

(ii) *Scenario 2: An adversary does not know $f(\cdot)$.* As different users can utilize different pseudorandom generators to implement $f(\cdot)$ as long as $\mathbf{e}_{i,j} \leftarrow \mathcal{U}[-r,r]$, $f(\cdot)$ can be kept secret. Therefore, this storage scenario is common in practical. In this scenario, we utilize both the brute force attack and the Arora-Ge algorithm [22] to analyze the confidentiality of SWE.

(ii-A) The brute force attack: We consider this attack in two cases. The first case is to enumerate $\mathbf{e}$ to recover $\mathbf{d}$ for an adversary obtaining k codewords. In SWE, an adversary has to enumerate $(2r+1)^n$ times. If an adversary obtains fewer than k codewords, it cannot discover any information of $\mathbf{d}$. There is another exhaustive search method [23] for solving $\mathbf{d}$, however, the method needs more than $2k$ codewords to transform the LWE instances to norm form of LWE. In the second case, the entropy of errors $\mathbf{e}$ is determined by s and the total complexity of brute force attack is 2^z times, where z is the length of s. Compared with the above two cases, we set the r is enough large to make that the complexity of brute force attack is at least 2^z. In our scheme, the smallest r is $\lceil \sqrt{q_{768}} \rceil$, which satisfies the requirement.

(ii-B) The Arora-Ge algorithm: The idea of Arora-Ge [22] is generating a non-linear errors-free system of equations from LWE samples. We adapt the results of Arora-Ge from LWE with Gaussian errors to LWE with uniform errors. If an adversary has the computational power as the Arora-Ge algorithm, the time of recovering the data is $k^{O(r)}$. However this method needs $\mathcal{O}(k^{2r+1})$ codewords.

Theorem 2. *Assume that $r > 0$, $(\mathbf{A}, \mathbf{Ad}+\mathbf{e}) = (\mathbf{A}, \mathbf{c}) \in \mathcal{Z}_q^{n\times k} \times \mathcal{Z}_q^n$, and the uniform distribution of errors is $\mathcal{U}[-r,r]$, then the time complexity of the Arora-Ge algorithm is $k^{O(r)}$.*

Proof. The polynomial generated in the Arora-Ge can be $P(x) = X\prod_{i=1}^{r}(X-i)$. Assume that $q > (2r+1)n$ and $1 \le n \le \binom{k+1}{2}$, and generate $f_1, ..., f_n$, where $f_l = P(c_l - \sum_{j=1}^{k} x_j \mathbf{A}_{j,l})$. $f_1^H, ..., f_n^H$ are linearly independent with probability larger than $1 - \frac{(2r+1)n}{q}$ according to Schwartz-Zippel-Demillo-Lipton Lemma [22]. $P(\cdot)$ is monomials polynomial and has degree $D_{AG} \le 2r+1$. Then the time complexity of Arora-Ge algorithm is $k^{O(D_{AG})} = k^{O(r)}$.

□

There are other algorithms which can assess *concrete hardness* of LWE such as BKW and other algorithms developed from BKW [23]. However, these algorithms are designed for standard instances of LWE with large k and relatively small q such as the example shown in Table 2. Contrary to LWE, SWE has relatively small k and large q. Meanwhile, BKW work in the assumption that an adversary can query oracle polynomial times. However, in dispersal storage scenarios, an adversary can obtain at most n codewords. Thus BKW is unsuitable for assessing the confidentiality of SWE.

From the above analysis, we can see that SWE achieves higher confidentiality than existing schemes under our assumptions. Although SWE has small k and exponential q, it still can guarantee data confidentiality. Furthermore, q and r are important to the confidentiality of SWE. When selecting parameters of SWE, we should set q to be large integers.

5.2 Availability

SWE and many existing schemes, such as Shamir's algorithm [10], Rabin's IDA [11], AONT-RS [3], etc., can guarantee data availability even though $(n-k)$ out of the n codewords are corrupted or completely unavailable. Therefore, larger $\frac{n-k}{n}$ means higher availability. However, larger $\frac{n-k}{n}$ also means more storage overhead and relatively lower security (i.e., an adversary needs to compromise fewer storage servers to reconstruct the data).

With the same data availability, the storage overhead of SWE is the same as the state of the art. For example, the data is 10 MB, $k = 3$, and $n = 5$. For Shamir's algorithm, as each codewords has the same length as the original data, the storage overhead is $5 \times 10 = 50$ MB. For SWE, Rabin's IDA, and AONT-RS, $\mathbf{C}_{i,j}$ has the same length as $\mathbf{D}_{i,j}$. Thus SWE, Rabin's IDA, and AONT-RS require approximately the same storage overhead, $\frac{5}{3} \times 10 \approx 16.7$ MB.

5.3 Performance

We suppose that the data $\mathbf{D}$ is 4 KB, $k = 6$, and $n = 12$. As multiplication is the most time consuming operation, we use the number of multiplications as the metric for comparing various algorithms.

Shamir's algorithm: To apply Shamir's algorithm, we divide $\mathbf{D}$ into 4096 bytes, $d_0, ..., d_{4095}$. Each of the 12 slices, $S_0, ..., S_{11}$, is composed of 4096 bytes, $s_{i,0}, ..., s_{i,4095}$. $s_{i,j}$ is a function of d_j and $r_{j,x}$. Specifically,

$$s_{i,j} = d_j \otimes \sum_{x=1}^{5} (i+1)^x \times r_{j,x},$$

where $r_{j,x}$ is a random byte. The arithmetic is over *Galois Field*, $GF(2^8)$. The number of multiplications is $12 \times 5 \times 4096 = 245760$.

Rabin's IDA: With Rabin's IDA, we append 2 bytes to $\mathbf{D}$, and divide it into 6×683 bytes, $d_{0,0}, ..., d_{5,682}$. Subsequently, we calculate the 12×683 codewords using the *non-systematic Reed-Solomon coding.* Specifically,

$$c_{i,j} = \sum_{x=0}^{5} (i+1)^x \times d_{x,j}.$$

The arithmetic is over $GF(2^8)$. Therefore, the number of multiplications is $12 \times 6 \times 683 = 49176$.

AONT-RS: With AONT-RS, we utilize AES-256 as "generator" and SHA-256 as "hash function" in its AONT phase. We use the *systematic (6, 12) Reed-Solomon code* over $GF(2^8)$ in its RS phase. We divide $\mathbf{D}$ into 256 128-bit elements, $d_0, ..., d_{255}$. Then we add 128-bit "canary", d_{256}. We choose *key* to be 256 random bits and compute $c_i = d_i \oplus E(key, i+1)$. Next, we calculate $h = H(c_0, ..., c_{256})$. Subsequently, we set $c_{257} = h \oplus key$. As with the RS phase, we add 2 bytes and then divide the 4146 bytes into 6×691 bytes slices, which are also the first 6×691 codewords. Subsequently we calculate the last 6×691 codewords over $GF(2^8)$. AES-256 calls 14 times of the rounds function. Each rounds function of AES-256 equals six times of exclusive-or. We consider each rounds function of AES-256 to be six times of multiplication. The rounds function of SHA-256 has 64 iterations, and each input calls one time of the rounds function [24]. We consider each iteration in the rounds function of SHA-256 to be one time of multiplication. Thus the number of multiplications in AONT phase is $14 \times 257 \times 6 + 64 \times 257 = 38036$. The number of multiplications in RS phase is $6 \times 6 \times 691 = 24876$. The number of multiplications in AONT-RS is $38036 + 24876 = 62913$.

SWE: With SWE, when $k = 6$ and $n = 12$, the corresponding $q = q_{1024}$. Thus the arithmetic is over $GF(q_{1024})$. We append 4096 bits to $\mathbf{D}$, and divide it into 6×6 1024-bit elements, $d_{0,0}, ..., d_{5,5}$. Next, we calculate 12×6 codewords

$$c_{i,j} = \sum_{x=0}^{5} a_{i,x} \times d_{x,j} + e_{i,j},$$

where $a_{i,j} \leftarrow \mathcal{Z}_q$. Suppose that SWE is implemented on a 64-bit machine. We use the number theoretic transform (NTT) and its inverse (INTT) to optimize the multiplication. Specifically,

$$a_{i,x} \times d_{x,j} = \mathrm{INTT}(\mathrm{NTT}(\hat{a}_{i,x}) \odot \mathrm{NTT}(\hat{d}_{x,j})),$$

where $\odot$ donates point-wise multiplication, $\hat{a}_{i,x}$ is the polynomial corresponding to $a_{i,x}$, and $\hat{d}_{i,x}$ is the polynomial corresponding to $d_{i,x}$. The length of NTT or INTT is $m = 1025 \div 64 \approx 17$. We consider NTT and INTT to be $m \log_{10} m$ times of multiplication. Thus the number of multiplications is $12 \times 6 \times 6 \times (3 \times 17 \log_{10} 17 + 17) \approx 34454$.

From the above analysis, we can see that with those parameters, SWE outperforms Rabin's IDA, AONT-RS, and Shamir's algorithm. Shamir's algorithm is the slowest scheme among the four schemes. Although AONT-RS employs the *systematic Reed-Solomon code* [3] to eliminate the need to encode the first 6×691 codewords, the AONT phase is time consuming. The experiment results in the following section also demonstrate the above analysis.

6 Experiments and Evaluations

In order to validate SWE and obtain optimal parameters for SWE, we build the prototype of SWE. The experiments are conducted on a 64-bit machine with an Intel (R) Core (TM) i7-4790 processor (4 cores) at 3.6 GHZ with 4 GB RAM. We use Ubuntu 14.04 LTS as the operating system. We use NTL-9.3.0 and GMP-5.1.3 [25] as the tools for the number theory. We build the prototype of AONT-RS using OpenSSL-1.0.2f and Jerasure-1.2 [26]. SWE, AONT-RS, and Shamir's algorithm are implemented using C language and a single thread. The following experiment results are averaged from 50 experiments.

6.1 Performance Tuning

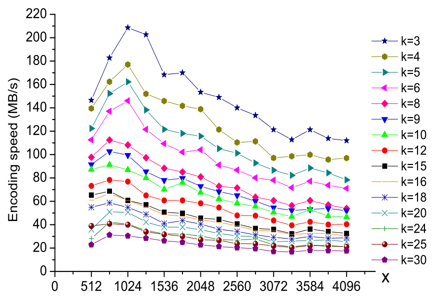


Fig. 6. Encoding speed of SWE varying with k and q_x.

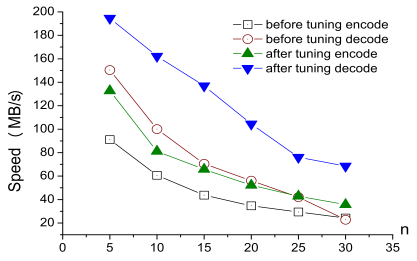


Fig. 7. Comparison data processing performance of SWE before and after optimizing parameters, where $k/n = 3/5$.

If we implement SWE with parameters only satisfying the hardness of LWE, data confidentiality can be guaranteed, but encoding/decoding speed may be slow. Therefore, in order to validate SWE and choose optimal parameters for SWE, we build the prototype of SWE and do the following experiments.

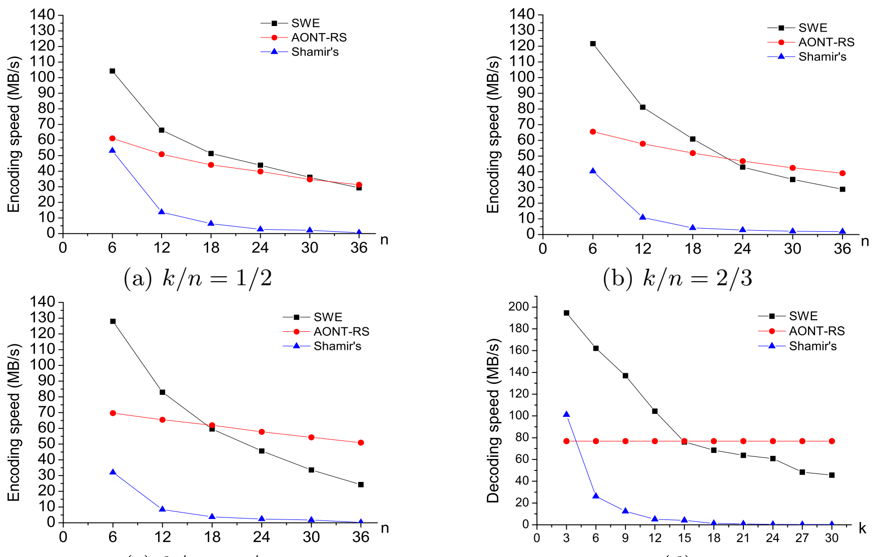
In this experiment, $\mathbf{A} \in \mathcal{Z}_q^{k \times k}$, $q = q_x$, q changes from q_{512} to q_{4096}, and $r = \lceil \sqrt{q} \rceil$. Fig. 6 illustrates encoding speed of SWE varying with k and q_x. The results show that: **(i)** When $3 \leq k \leq 6$, the optimal q is q_{1024} (i.e., achieves fastest speed). **(ii)** When $6 < k \leq 30$, the optimal q is q_{768}.

After performance tuning, we do the experiment of comparison data processing performance of SWE, where $k/n = 3/5$ and $r = \lceil \sqrt{q} \rceil$. Before tuning SWE, $q = q_{2048}$. After tuning SWE, when $3 \leq k \leq 6$, $q = q_{1024}$, and when $6 < k \leq 18$,

$q = q_{768}$. Fig. 7 illustrates the results. The results show that: **(i)** k and n affect performance heavily. In practical, we should use smaller k and n, while SWE with such parameters should guarantee availability and confidentiality. **(ii)** q also affects performance heavily. With proper k and corresponding q, we can build a more efficient dispersal scheme. **(iii)** When encoding speed is high, the corresponding decoding speed is relatively high.

In a word, when implementing SWE, we should tune parameters to achieve better data processing performance. Meanwhile, SWE with such parameters should satisfy data confidentiality and availability.

6.2 Comparison on Encoding and Decoding



(a) $k/n = 1/2$ (b) $k/n = 2/3$ (c) $k/n = 5/6$ (d)

Fig. 8. Comparison of encoding/decoding speed, where (a)-(c) are for encoding and (d) is for decoding.

In this section, we evaluate performance of SWE and comparable schemes. Rabin's IDA [11] cannot guarantee data confidentiality. SSMS [12] does not specify a dispersal or encryption algorithm. Therefore, we do not compare SWE with Rabin's IDA or SSMS. Shamir's algorithm [10] achieves availability and high confidentiality. As far as we know, AONT-RS [3] is the best scheme that achieves balance between confidentiality and data processing performance. Thus, we compare SWE with Shamir's algorithm and AONT-RS in encoding/decoding speed. In AONT-RS, AES-256 and SHA-256 are applied in its AONT phase.

Encoding: We measure the encoding speed of SWE, Shamir's algorithm and AONT-RS. For SWE, $r = \lceil \sqrt{q} \rceil$, when $3 \leq k \leq 6$, $q = q_{1024}$, and when

$6 < k \leq 30$, $q = q_{768}$. Figures 8 (a), (b), and (c) exhibit the encoding comparison results, where $k/n = 1/2$, $k/n = 2/3$, and $k/n = 5/6$, respectively. The results show that: **(i)** When $6 \leq n \leq 30$ and $k/n = 1/2$, SWE outperforms AONT-RS in encoding speed. **(ii)** When $6 \leq n \leq 18$ and $k/n = 2/3$, the encoding speed of SWE is faster than that of AONT-RS. **(iii)** When $6 \leq n \leq 12$ and $k/n = 5/6$, SWE performs better than AONT-RS in encoding speed. **(iv)** Among all the experiments, SWE outperforms Shamir's algorithm in encoding speed.

Decoding: We also measure the decoding speed of SWE, Shamir's algorithm, and AONT-RS. In this experiment, the parameters of SWE are the same as the parameters used in the encoding speed comparison. As the decoding speed of SWE or Shamir's algorithm depends on k (i.e., solve k equations), we only mention k in Fig. 8 (d). The results in Fig. 8 (d) show that: **(i)** As k and n grow, the decoding speed of SWE reduces heavily. The smaller is k, the higher is the decoding speed of SWE. **(ii)** When $k \leq 15$, the decoding speed of SWE is faster than that of AONT-RS. **(iii)** Among all the experiments, SWE outperforms Shamir's algorithm in decoding speed. **(iv)** As the *systematic Reed-Solomon code* is applied to disperse data in AONT-RS, if the codewords are obtained from the first k rows, decoding only involves the AONT phase. Thus, decoding speed of AONT-RS mentioned in Fig. 8 (d) is the best result. For AONT-RS, if there are some codewords which are not obtained from the first k rows, decoding involves both the AONT and Reed-Solomon decoding operations. Hence decoding speed of AONT-RS will be slower than the results mentioned in Fig. 8 (d), and as k and n grow, decoding speed of AONT-RS will reduce heavily.

Resch and Plank [3] suggest that in practical storage scenarios, n should always be smaller than 16. From the experiment results, we can see that with common configuration (i.e., $n \leq 16$), SWE outperforms AONT-RS and Shamir's algorithm in encoding/decoding speed.

Large k and n (e.g., $k = 200$, $n = 240$) lead to slow encoding/decoding speed. However, SWE with such a configuration is resistant to attacks from a quantum computer, which is capable of solving the *generalized discrete Fourier transform problems* [27]. When k and n are large, we can also use many well-established algorithms [28] to optimize matrices operations and thus improve encoding/decoding speed of SWE. However, such algorithms [28] are not efficient for AONT-RS, as the time consuming phase of AONT-RS is the AONT phase. In a word, when setting configurations for SWE, we should make trade-offs between data processing performance, confidentiality, and availability.

7 Conclusion

In this paper, we have proposed a secure and fast dispersal storage scheme, known as SWE. By applying the reformed LWE to SWE, even if an adversary obtains all the codewords, SWE can still guarantee data confidentiality. Theoretical analysis shows that under our assumptions, SWE achieves higher confidentiality than existing schemes, but still at the same storage cost with the state of the art. Furthermore, SWE favorably supports secure and efficient dynamic data operations, where users are not required to download or decode corresponding codewords, and no data information is leaked. Analysis and experiment results also show that with proper configurations, SWE outperforms the state of the art in encoding/decoding speed.

Based on our work, the hardness of LWE with small k, exponential q, and uniform errors can be further investigated. We hope an efficient reduction from the standard LWE to such variant of LWE can be given. Furthermore, efficient and secure dynamic data integrity auditing can be investigated based on SWE. These further investigations will allow us to build a securer and more efficient dispersal storage scheme.

References

1. Valentina Ciriani, Sabrina De Capitani Di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In *European Symposium on Research in Computer Security*, pages 440–455. Springer, 2009.
2. Pierangela Samarati and Sabrina De Capitani Di Vimercati. Data protection in outsourcing scenarios: Issues and directions. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 1–14. ACM, 2010.
3. Jason K. Resch and James S. Plank. AONT-RS: Blending security and performance in dispersed storage systems. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, pages 229–240, 2011.
4. Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):8, 2008.
5. Nurul Hidayah Ab Rahman and Kim-Kwang Raymond Choo. A survey of information security incident handling in the cloud. *Computers & Security*, 49:45–69, 2015.
6. Amazon S3 availability event: July 20, 2008. In *http://status.aws.amazon.com/s3-20080720.html*, 2008.
7. Experts say Facebook leak of 6 million users data might be bigger than we thought. In *http://www.huffingtonpost.com/2013/06/27/facebook-leak-data_n_3510100.html*, 2013.
8. iCloud leaks of celebrity photos. In *https://en.wikipedia.org/wiki/ICloud_leaks_of_celebrity_photos*, 2014.
9. Mark W Storer, Kevin M Greenan, Ethan L Miller, and Kaladhar Voruganti. Potshardsa secure, recoverable, long-term archival storage system. *ACM Transactions on Storage (TOS)*, 5(2):5, 2009.
10. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
11. Michael O Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348, 1989.
12. Hugo Krawczyk. Secret sharing made short. In *Annual International Cryptology Conference*, pages 136–146. Springer, 1993.
13. C Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 17(4):15, 2015.

14. Xin Dong, Jiadi Yu, Yuan Luo, Yingying Chen, Guangtao Xue, and Minglu Li. Achieving an effective, scalable and privacy-preserving data sharing service in cloud computing. *Computers & Security*, 42:151–164, 2014.
15. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.
16. Nico Döttling and Jörn Müller-Quade. Lossy codes and a new variant of the learning-with-errors problem. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 18–34. Springer, 2013.
17. Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing*, pages 575–584. ACM, 2013.
18. Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 700–718. Springer, 2012.
19. Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.
20. Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.
21. Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 229–240. ACM, 2010.
22. Sanjeev Arora and Rong Ge. Learning parities with structured noise. *Electronic Colloquium on Computational Complexity*, 17, 2010.
23. Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.
24. New instructions supporting the secure hash algorithm on Intel architecture processors. In *https://software.intel.com/en-us/articles/intel-sha-extensions*, 2013.
25. NTL: A library for doing number theory. In *http://www.shoup.net/ntl/*, 2015.
26. James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. Technical report, Citeseer, 2008.
27. Marcos Curty. Quantum cryptography: Know your enemy. *Nature Physics*, 10(7):479–480, 2014.
28. Charles-Éric Drevet, Md Nazrul Islam, and Éric Schost. Optimization techniques for small matrix multiplication. *Theoretical Computer Science*, 412(22):2219–2236, 2011.