Secure and Reliable Distributed Storage Without Secret Keys

Abstract

We propose a class of algorithms for secure and reliable distributed storage without relying on secret keys. In particular, our keyless algorithms make novel use of binary error control codes to protect and store files. They feature low computation overhead while achieving a joint defense in reliability, confidentiality, and integrity, which are collectively quantified using a new unifying metric, resilience vector. We prove perfect secrecy of the proposed algorithms and derive a calibration method by trading-off the three functionalities under different resilience requirements. Implementation of the algorithm shows that it achieves both security and reliability with lower complexity than systems relying on secret keys and erasure coding.

I. Introduction

Distributed storage of critical data over large networks has become an essential component of distributed computing environment. Since storage servers and client machines are considered as insecure and unreliable, we need to provide *security* (i.e., confidentiality, integrity, and availability) as well as *reliability* (i.e., availability against malicious attacks and tolerance against server/network failures). Traditionally, security and reliability are treated separately, with cryptographic operations and private keys on the client side to keep data secure and redundancy the added on the server side to enhance reliability.

This approach for security and reliability raises several important design issues. First, the use of private keys may be insecure due to possible client-side security breaches. Data confidentiality that is protected by private keys will be violated if the keys are revealed. Further, when a distributively-stored file is shared by multiple clients, its private key has to be synchronized among all the client devices. Key synchronization could introduce security vulnerabilities and are highly complex for large-scale distributed systems. Next, a tradeoff between security and reliability remains to be formulated, since improving security and reliability both require the use of more system resource, e.g., higher redundancy and computation capability. Is it possible to propose a design which captures security, reliability, and their tradeoff within one framework? In this paper, we make novel use of binary error control codes and derive two different algorithms for jointly achieving reliability, confidentiality, and integrity, without relying on secret keys at client side. In order to obtain low complexity, we employ a decoding procedure using *table lookup and guessing* at data retrieving phase.

In Section V and VI, we develop two simple and effective algorithms that do not rely on secret keys. For security, previous solutions for distributed storage normally use encryptions for data confidentiality and digital signatures for data integrity [2], [3], [4], [5]. These solutions uses secret keys and are computationally expensive, while the cryptographic hash algorithms in these solution only detect integrity violations, but cannot correct modified data. A non-cryptographic and keyless approach for security has been taken in [6], [7], known as secret sharing. It has recently been generalized in [19], [21], [22], where theoretical bounds on security and reliability are derived in different forms and for different applications, e.g., multi-path key establishment and secure network coding. In these algorithms, randomness is added to original data to create a number of secret shares, such that a small portion of the shares provide no information about the data. Our two algorithms in this paper can be viewed as extensions on secret sharing for low-complexity applications: Instead of using complicated algebraic operations, our algorithms operate in GF_2. Under certain threshold conditions, our first algorithm serves as a one-time pad scheme and achieves *perfect secrecy* in that no information about the data is revealed to a cryptanalyst, even if he has unbounded computing power. In addition to this strong notion of confidentiality, the algorithm also guarantees that integrity violations at storage servers can be detected and corrected during the retrieving phase. Our second algorithm extends the basic scheme by further reducing its computation overhand, while partially provides perfect secrecy.

For reliability, it normally requires the introduction of redundancy on the server side. Most existing distributed storage solutions separate reliability from security and employ erasure/network coding to tolerate message retrieval

failures. The simplest form of redundancy is replication, which stores multiple copies of the same data on different servers and is adopted in many practical storage systems. To offer better storage efficiency, erasure-coding schemes [8], [9], [10], [11] divide a critical data into smaller pieces, encode them using an erasure code, and allow the original data to be recovered from any subset of a sufficient number of coded pieces. For the same redundancy factor, network coding has also been applied in [12], [13], [15], [16], [17] to improve server repair efficiency after reliability violations. Due to their underlying binary error control coding structure, our algorithms intrinsically allow reliability to be achieved simultaneously with security. We develop a decoding method using table lookup and guessing in our algorithms, for efficiently reconstructing the original data under both erasure and modification.

A main feature of our algorithms is *keyless*. In Section VII, we provide a security analysis and compare our keyless storage algorithms with previous algorithms that rely on secret keys. Our keyless algorithms are shown to outperform previous ones when possibilities of client-side security breaches are not negligible. The approach in this paper also allows us to formulate the security-reliability tradeoff in a closed form using a new unifying metric, *resilience vector*, defined in Section III. For a given distributed storage algorithm, its confidentiality, integrity, and reliability, are each measured by one element in its resilience vector. Therefore, the set of all resilience vectors achieved by a class of distributed storage algorithms form a 3-dimensional region. It quantifies the security and reliability of this various classes of algorithms, and provides a unifying framework for measuring security and reliability. We show that the resilience vector can be used to provide general design guidelines for distributed storage systems.

The rest of this paper is organized as follows: In Section II, we discusses the assumptions and our threat model for distributed storage. Section III defines the new metric of resilience vector. Section IV briefly desribes previous storage algorithms. Section V and VI present the mechanisms and the complete protocols of our keyless distributed storage algorithms. Section VII gives a security and reliability analysis and outlines how to calibrate the algorithm by trading-off the three functionalities under different resilience requirements. We have implemented the proposed distributed storage algorithm in C and evaluated its performance in Section VIII, in terms of CPU cycles, memory requirement, and execution time. Proofs of theorems are collected in Appendix.

II. Threat Model

To store a data $\mathbb{F}$ on n servers, a distributed storage algorithm with input $\mathbb{F}$ generates n messages $\mathbb{M}_1, \ldots, \mathbb{M}_n$, each of which is sent to and stored at a different server. We do not trust servers and networks to properly store data, deliver data or keep data confidential. Therefore, retrieved messages are subject to confidentiality, integrity, and availability attacks. In addition to malicious message-dropping and denial of service attacks on availability (a Security issue), we also consider non-malicious message-dropping failures in the network or failure of storage server nodes (a Reliability issue). In this paper, we denote as *Attack on Reliability* any malicious attacks on availability in the security sense, as well as any non-malicious attacks due to server or network failures. In our threat model, attackers can perform any combination of the following three classes of security and reliability violations, each targeting at one functionality.

- **Data Confidentiality violation**: Attackers try to derive information of the original data $\mathbb{F}$ by obtaining messages and analyzing their content, e.g., a cryptanalysis attack on an encrypted message is viewed as an attack on confidentiality.
- **Data Integrity violation**: Attackers modify messages or make them misrepresent information, while messages are stored on servers or in transit through a network. This class of violations prevent a client from retrieving the original data $\mathbb{F}$ by violating message integrity. We also include non-malicious violations which cause data to be modified, e.g., due to a *noisy* channel. These are ‘signal reliability’ issues and are also included in this class of data Integrity violations.
- **Reliability violations**: Attackers make messages unavailable to their intended clients by destroying them on storage servers, or intercepting and dropping them in the network. Server failures caused by hardware or software errors (i.e., Reliability issues), as well as denial of service attacks (i.e., Availability issues), are considered together under the class of Reliability violations.

The threat model assumes that collusion between servers is unlikely, i.e., servers do not share stored data with other servers. Furthermore, all servers are assumed to have independently and identically distributed failure rates and susceptibility to vulnerabilities. In other words, all servers run the same operating system, implement equivalent

authentication systems, etc. However, client-side security and reliability is beyond the scope of the threat model. The machine's hardware and software are assumed to be protected from security and reliability violations while the distributed storage algorithm executes on critical data on the client's machine. For instance, an adversary cannot read the client machine's memory during the storage or retrieval phase. This is a fair assumption because if the client's machine were compromised with a malicious memory snooper, any storage or cryptographic algorithm would be vulnerable.

III. New Security-Reliability Metric

For a distributed storage algorithm with n servers, we introduce a new metric $(c, d, e)_n$ denoted as a *resilience vector*. The three elements c, d, e are non-negative integers taking values from $\{0, 1, \ldots, n\}$, and measure resilience against malicious or non-malicious violations of confidentiality, integrity, and reliability, respectively.

- For confidentiality, c is a threshold value such that if no more than c messages are revealed to attackers, they can derive no information about the original data $\mathbb{F}$, even if they have unbounded computing power.

  Definition 1: Data $\mathbb{F}$ is *completely unknown* given c revealed messages if

  $$\text{Prob}\left\{\mathbb{F}\middle|\mathbb{M}_{i_1}, \ldots, \mathbb{M}_{i_c}\right\} = \text{Prob}\left\{\mathbb{F}\right\}. \tag{1}$$

  for arbitrary subsets of indices $i_1, \ldots, i_c \in \{1, 2, \ldots, n\}$.

  This is a strong notion of security, sometimes called *perfect secrecy* [6], [18], which does not rely on computational hardness assumptions, unlike conventional encryption algorithms.
- For integrity, d is a threshold value such that the original information $\mathbb{F}$ can be successfully reconstructed through the distributed storage algorithm from retrieved messages, if there are no more than d faulty messages.
- For reliability, e is a threshold value such that the original information $\mathbb{F}$ can be successfully reconstructed through the distributed storage algorithm from any subset of $n - e$ messages.

By stacking the three elements c, d, e into a resilience vector $(c, d, e)_n$, we say that a resilience vector is achievable, if the three confidentiality, integrity, and reliability properties specified above simultaneously hold, i.e.,

Definition 2: A resilience vector $(c, d, e)_n$ is achievable by a distributed storage algorithm, if the original data $\mathbb{F}$ remains completely unknown with no more than c revealed messages, and at the same time, it can be successfully reconstructed from any subset of $n - e$ messages containing no more than d messages with errors.

The resilience vector $(c, d, e)_n$ gives a single, unifying metric for comparing security and reliability of different distributed storage algorithms under our threat model. The entire set of achievable resilience vectors forms a 3-dimensional region, which illustrates an optimal tradeoff among confidentiality, integrity, and reliability. Since cryptographic operations are based on computational hardness assumptions, distributed storage algorithms using secret keys achieve no perfect secrecy.

IV. Previous Algorithms Relying on Secret Keys

In previous distributed storage algorithms, security is normally protected by cryptographic operations and secret keys on the client side, while reliability is achieved by adding redundancy on the server side. To reliably store a file $\mathbb{F}$ on n distributed servers, the erasure coding approach in [8], [9], [10], [11] utilizes linear erasure codes to generate messages $\mathbb{M}_1, \ldots, \mathbb{M}_n$. It is a special class of binary linear error control code and works only for message erasures. An (n, k, s) erasure code has a property that any $n - s + 1$ out of the n messages suffice to recover the original critical data. In a similar approach, network coding has also been applied in [12], [13], [15], [16], [17] to improve data repair efficiency.

Messages $\mathbb{M}_1, \ldots, \mathbb{M}_n$ have to be further encrypted and hashed to provide confidentiality and integrity in distributed storage. Let $h(\cdot)$ be hash function and a $\mathbb{K}$ be a secret key. We attach a hash vale is attached to each message, i.e., $< \mathbb{M}_i | h(\mathbb{M}_i) >$ and then store its encrypted value $E_{\mathbb{K}} < \mathbb{M}_i | h(\mathbb{M}_i) >$ at server $i = 1, \ldots, n$. During file retrieving phase, the secret key $\mathbb{K}$ needs to be used to decrypt $< \mathbb{M}_i | h(\mathbb{M}_i) >$, while hash value $h(\mathbb{M}_i)$ guarantees integrity of message $\mathbb{M}_i$. A standard erasure/network decoding procedure is then employed to recover original file $\mathbb{F}$ from those messages that are successfully retrieved.

This algorithm requires that the encryption key $\mathbb{K}$ is kept secret at client side for the entire data storing period. If the key is revealed, then this algorithm provides no defense against confidentiality breaches. In fact, when the probability of the key being compromised by an adversary is high, this algorithm that relies on encryption, coding, and hash will provide poor protection against confidentiality breaches because any one server being attacked will reveal the stored data. Similarly, the reliability of the data is tied to the availability and integrity of the encryption key - the adversary just attacks the integrity and availability of the encrypting key to prevent the client from accessing his data, independent of servers being attacked. Moreover, perfect secrecy cannot be achieved according to Definition 1, since encryption relies on computation hardness assumptions. Therefore, this algorithm relies on secret keys and result in $c = 0$. In the next two sections, we will propose two distributed storage algorithms which are keyless and achieve strictly positive $c, d, e > 0$.

V. Our Keyless Algorithm for Perfect Secrecy

We present a complete protocol for storing and retrieving over n distributed servers, starting with an illustrative example to describe the underlying mechanism of our algorithm.

A. *An Illustrative Example*

Consider the storage of a single data object $\mathbb{S}_1 = \mathbb{F}$ (represented by a binary vector) on $n = 7$ servers. In the storing phase, we first generate $t = 3$ random vectors $\mathbb{X}_1, \mathbb{X}_2, \mathbb{X}_3$, which all have the same length as $\mathbb{S}_1$ and are used to ensure randomness in our algorithm. To construct $n = 7$ messages, we encode $[\mathbb{S}_1, \mathbb{X}_1, \mathbb{X}_2, \mathbb{X}_3]$ using a $(n + k = 8, t + k = 4, s = 4)$ binary linear error control code, which has a 4×8 generator matrix

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}. \tag{2}$$

We have $k = 1$ in this example, since the file $\mathbb{F}$ is represented by a a single data object $\mathbb{S}_1 = \mathbb{F}$. Parameter $s = 4$ is the Hamming distance of the code. It is equal to the minimal weight (i.e., the number of non-zero components) among all non-zero linear combinations of rows of G, and measures the error correcting capability of the code. Encoding is a bit matrix multiplication with $[\mathbb{S}_1, \mathbb{X}_1, \mathbb{X}_2, \mathbb{X}_3]$ and the generator matrix G:

$$[\mathbb{S}_1, \mathbb{X}_1, \mathbb{X}_2, \mathbb{X}_3] \cdot G = [\mathbb{S}_1, \mathbb{X}_1, \mathbb{X}_2, \mathbb{S}_1 \oplus \mathbb{X}_1 \oplus \mathbb{X}_2, \mathbb{X}_3, \mathbb{S}_1 \oplus \mathbb{X}_1 \oplus \mathbb{X}_3, \mathbb{S}_1 \oplus \mathbb{X}_2 \oplus \mathbb{X}_3, \mathbb{X}_1 \oplus \mathbb{X}_2 \oplus \mathbb{X}_3]. \tag{3}$$

where $\oplus$ is a bit-wise XOR operator. The first column on the right hand side of (3) is $\mathbb{S}_1$ and is discarded. Messages are chosen to be the last $n = 7$ columns on the right hand side of (3): $\mathbb{M}_1 = \mathbb{X}_1$, $\mathbb{M}_2 = \mathbb{X}_2$, . . ., $\mathbb{M}_7 = \mathbb{X}_1 \oplus \mathbb{X}_2 \oplus \mathbb{X}_3$. Matrix G can be public. Random vectors $\mathbb{X}_1, \mathbb{X}_2, \mathbb{X}_3$ are destroyed after the messages are computed.

For confidentiality, it is easy to verify that taking any $c = 2$ messages, $\mathbb{S}_1$ cannot be derived, since it is XORed with some linear combination of $\mathbb{X}_1, \mathbb{X}_2, \mathbb{X}_3$ –this serves as a onetime pad in our distributed storage algorithm. Furthermore, there exist three messages whose XOR generates $\mathbb{S}_1 = \mathbb{M}_1 \oplus \mathbb{M}_2 \oplus \mathbb{M}_3$. This means that the algorithm achieves perfect secrecy for up to $c = 2$ violations of the confidentiality of the messages.

To describe the error decoding procedure, we use the concept of a parity check matrix and dual code. A parity check matrix H for the $(8, 4, 4)$ code is a 4×8 full rank matrix whose rows are orthogonal to every row of G,

$$H = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Since H is orthogonal to the row space of G, i.e., $H \cdot G^T = \mathbf{0}$, we have $r = H[\mathbb{S}_1, \mathbb{M}_1, \ldots, \mathbb{M}_7]^T = \mathbf{0}$, where r is called a syndrome for $[\mathbb{S}_1, \mathbb{M}_1, \ldots, \mathbb{M}_7]$. When there are no errors or erasures, we always have $r = 0$. Otherwise, $r \neq 0$ and it can be used to locate errors.

Next we show that integrity and reliability can be achieved in the retrieving phase. For simplicity, let $\mathbb{S}_1, \mathbb{X}_i, \mathbb{M}_i$ be single bits. (When they are vectors, decoding is performed row-by-row.) We assume that the last two messages are erased and retrieve a subset of $n - e = 5$ messages $[\hat{\mathbb{M}}_1, \ldots, \hat{\mathbb{M}}_5]$. For decoding, we replace the unknown data

$\mathbb{S}_1$ and the two erased messages by a single variable $\mathbb{U}$. Since three bits are unknown, one of the two guesses of $\mathbb{U}$ (i.e., $\mathbb{U} = 0$ or 1) gives at most 1 error. According to linear error control coding theory [14], this single error in $[\mathbb{U}, \hat{\mathbb{M}}_1, \ldots, \hat{\mathbb{M}}_5, \mathbb{U}, \mathbb{U}]$ can be corrected by looking up its syndrome vector in a table, which maps syndrome vectors to error patterns, which are used to refine our guesses. The syndrome vector is computed by bit matrix multiplication with a 4×8 parity check matrix H:

$$R = H \cdot [\mathbb{U}, \hat{\mathbb{M}}_1, \ldots, \hat{\mathbb{M}}_5, \mathbb{U}, \mathbb{U}]. \tag{4}$$

B. Our Keyless Algorithm for Perfect Secrecy

In the storing phase, we first divide a binary file $\mathbb{F}$ of length f into k equal-length segments, $\mathbb{S}_1, \ldots, \mathbb{S}_k$, such that each segment has $m = f/k$ bits and is stored at a different server. Next, t pseudo-random length-m vectors $\mathbb{X}_1, \ldots, \mathbb{X}_t$ are constructed at the client side. We stack the k data segments and the t random vectors as columns in an input matrix $[\mathbb{S}_1, \ldots, \mathbb{S}_k, \mathbb{X}_1, \ldots, \mathbb{X}_t]$ of size $m \times (k + t)$, and apply a $(n + k, t + k, s)$ binary linear error control encoding to each row of this matrix, using the following generator matrix

$$G = \begin{bmatrix} 1 & 0 & \ldots & g_{1,1} & \cdots & g_{1,n} \\ 0 & 1 & \ldots & g_{2,1} & \cdots & g_{2,n} \\ \vdots & \vdots & \ldots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & g_{k+t,1} & \cdots & g_{k+t,n} \end{bmatrix}_{(k+t)\times(k+n)} \tag{5}$$

The first k columns of the generator matrix G are systematic, i.e., the upper left $k \times k$ submatrix of G is an identity matrix. Such generator matrices can always be derived by performing simple row eliminations. Applying G to encode the input matrix row by row, we obtain a coded message matrix $[\mathbb{S}_1, \ldots, \mathbb{S}_k, \mathbb{X}_1, \ldots, \mathbb{X}_t] \cdot G$ of size $m \times (n + k)$. Due to the systematic property of the generator matrix G, it is easy to see that in the message matrix, the first k columns are just the original data segments $\mathbb{S}_1, \ldots, \mathbb{S}_k$, and do not need to be computed. For $i = 1, \ldots, n$, we choose message $\mathbb{M}_i$ as the $(k + i)$'th column of the the coded message matrix and send $\mathbb{M}_i$ to server i to be stored there. Let $G_{k+1:k+n}$ be a submatrix of G by removing the first k columns. We have

$$[\mathbb{M}_1, \ldots, \mathbb{M}_n] = [\mathbb{S}_1, \ldots, \mathbb{S}_k, \mathbb{X}_1, \ldots, \mathbb{X}_t] \cdot G_{k+1:k+n}, \tag{6}$$

or, in a more explicit form,

$$\mathbb{M}_i = (g_{1,i}\mathbb{S}_1) \oplus \ldots \oplus (g_{k,i}\mathbb{S}_k) \oplus (g_{1+k,i}\mathbb{X}_1) \oplus \ldots \oplus (g_{t+k,i}\mathbb{X}_t). \tag{7}$$

The distributed storage protocol for the storing phase is summarized in Algorithm (1a).

To state our protocol for the retrieving phase, without loss of generality, we assume that the last e messages can not be retrieved due to violations of reliability and the remaining $n - e$ retrieved messages $\left[\hat{\mathbb{M}}_1, \ldots, \hat{\mathbb{M}}_{n-e}\right]$ contain d errors due to violations of integrity. Let H be a parity check matrix of size $(n - t) \times (k + n)$ for the code given by (5). For decoding, we make two guesses on erased coordinates (i.e. all-zero and all-one) and calculate two syndrome by bit matrix multiplication with H: in Step 5 below by replacing the erased messages by all-zero vectors, and in Step 6 below by replacing the erased messages by all-one vectors. Then, looking up the syndrome table and comparing the error patterns corresponding to the two syndrome, we can recover the original codeword matrix $[\mathbb{S}_1, \ldots, \mathbb{S}_k, \mathbb{M}_1, \ldots, \mathbb{M}_n]$ from $\left[\hat{\mathbb{M}}_1, \ldots, \hat{\mathbb{M}}_{n-e}\right]$ row by row.

Recall that the original data $\mathbb{F}$ is the first k columns of the codeword matrix $[\mathbb{S}_1, \ldots, \mathbb{S}_k, \mathbb{M}_1, \ldots, \mathbb{M}_n]$. In our protocol, decoding only the first k columns of the codeword matrix are sufficient for reconstruction of the original data. Let $\vec{t^0}$ and $\vec{t^1}$ be the two error vectors for syndromes R_i^0 and R_i^1 respectively, for $i = 1, \ldots, m$. The error vectors are used to refine our guesses. If $\vec{t^0}$ contains fewer errors on the non-erased coordinates, we conclude that the assumption of all zero vectors is correct and $\vec{t^0_{1:k}}$ gives i'th row of $[\mathbb{S}_1, \ldots, \mathbb{S}_k]$. Otherwise, if $\vec{t^1}$ contains fewer number of errors on the non-erased coordinates, we choose $\mathbf{1} \oplus \vec{t^1_{1:k}}$. We define a length-$(n + k)$ mask vector A and use a *popcnt* instruction to compute the number of errors on unerased coordinates. Finally, by concatenating $\mathbb{S}_1, \ldots, \mathbb{S}_k$, we obtain the critical data $\mathbb{F}$.

(1a) Storing Phase:

Public parameters: k, t, n, G.

Private inputs: $\mathbb{F}$.

1) Divide file $\mathbb{F}$ into k segments, each of f/k bits.
$$[\mathbb{S}_1, \dots, \mathbb{S}_k] \leftarrow \mathbb{F}$$
2) Generate t pseudo-random vectors, each of m bits.
$$[\mathbb{X}_1, \dots, \mathbb{X}_t] \leftarrow random()$$
3) Apply bit matrix multiplication of $m \times (t+k)$ and $(t+k) \times n$ matrices to compute n messages by
$$[\mathbb{M}_1, \dots, \mathbb{M}_n] \leftarrow [\mathbb{S}_1, \dots, \mathbb{S}_k, \mathbb{X}_1, \dots, \mathbb{X}_t] \cdot G_{k+1:k+n}$$
4) Destroy $\mathbb{F}, \mathbb{S}_1, \dots, \mathbb{S}_k, \mathbb{X}_1, \dots, \mathbb{X}_t$ on the client side by overwrite operations.
5) Send message $\mathbb{M}_i$ to server i for storage, $\forall i$.
6) Destroy $\mathbb{M}_1, \dots, \mathbb{M}_n$ on the client side by overwrite operations.

(1b) Retrieving Phase:

Public parameters: T_{out}, f, k, t, n, H, $Table$.

Private inputs: None.

1) Initialize auxiliary variables $A \leftarrow 0$, $\tilde{H} \leftarrow 0$, $e \leftarrow 0$, $i \leftarrow 1$, $r \leftarrow 0$.
2) Broadcast message retrieving requests to the n servers.
3) If message j is received, add $(j+k)$'th row of H to $\tilde{H}$, set the $(j+k)$'th bit in vector A, and let $i = i+1$. Proceed if n messages are retrieved or time-out T_{out} is reached.

   while $time() < T_{out}$ and $i < n$ **do**

       if $\hat{\mathbb{M}}_i \leftarrow retrieve(j)$ is TRUE **do**

           $\tilde{H}_i \leftarrow H_{j+k}$

           $A_{j+k} \leftarrow 1$

           $i \leftarrow i+1$

       end if

   end while
4) Get number of erased messages $e = n - i$.
5) Apply bit matrix multiplication of $(n-t) \times (n-e)$ and $m \times (n-e)$ matrices to compute syndrome R^0 for an all-zero guess.
$$R^0 \leftarrow H \cdot \left[\mathbf{0}, \dots, \mathbf{0}, \hat{\mathbb{M}}_1, \dots, \hat{\mathbb{M}}_{n-e}, \mathbf{0}, \dots, \mathbf{0}\right]^T$$
6) Apply bit matrix multiplication of $(n-t) \times (n-e)$ and $m \times (n-e)$ matrices to compute syndrome R^1 for an all-one guess..
$$R^1 \leftarrow H \cdot \left[\mathbf{1}, \dots, \mathbf{1}, \hat{\mathbb{M}}_1, \dots, \hat{\mathbb{M}}_{n-e}, \mathbf{1}, \dots, \mathbf{1}\right]^T$$
7) Lookup the syndrome table with the syndrome vectors to decode $[\mathbb{S}_1, \dots, \mathbb{S}_k]$ row by row.

   for $i = 1, \dots, m$ **do**

       $\vec{t^0} \leftarrow Table[R^0_i]$

       $\vec{t^1} \leftarrow Table[R^1_i]$

       if $popcnt(\vec{t^0} \& A) < popcnt(\vec{t^1} \& A)$ **do**

           $[\mathbb{S}_1, \dots, \mathbb{S}_k]_i \leftarrow 0 \oplus \vec{t^0}_{1:k}$

       else

           $[\mathbb{S}_1, \dots, \mathbb{S}_k]_i \leftarrow 1 \oplus \vec{t^1}_{1:k}$

       end if

   end for
8) Restore and output $\mathbb{F} \leftarrow [\mathbb{S}_1, \dots, \mathbb{S}_k]$.

(1c) Server Repairing Phase:

Public parameters: G.

Private inputs: None.

1) Server i_0 obtains $RepairList$, which is a subset of columns that are linearly dependent with G_{i_0+k}.
$$Repairlist\,[i_1, \dots, i_l] \leftarrow GaussianElimination(G)$$
2) Server i_0 contacts servers $i_1, \dots, i_l$ in $RepairList$ to download l messages.

   for $j = 1, \dots, l$ **do**

       $\mathbb{M}_{i_j} \leftarrow retrieve(i_j)$

   end for
3) Re-create message $\mathbb{M}_{i_0}$ on server i_0.
$$\mathbb{M}_{i_0} \leftarrow \mathbb{M}_{i_1} \oplus \mathbb{M}_{i_2} \oplus \dots \oplus \mathbb{M}_{i_l}$$
4) Destroy $\mathbb{M}_{i_1}, \dots, \mathbb{M}_{i_l}$ on server i_0 by overwrite operations.

Fig. 1. Our Algorithm 1 for keyless distributed storage, including storing phase, retrieving phase, and server repairing phase.

C. Repairing Server Data Loss

To provide long-term security and reliability, messages stored on distributed servers must be continually refreshed as servers fail or leave the system. We propose a message repair algorithm that uses a linear combination of messages downloaded from surviving servers, such that server data loss can be repaired without decoding data $\mathbb{F}$.

According to error control coding theory, any column of generator matrix G can be derived by a linear combination of a subset of other columns. Therefore, any single message can be repaired by a proper linear combination of a subset of remaining messages. To repair message $\mathbb{M}_{i_0}$ that is generated by the $i_0 + k$'th column of G (denoted by G_{i_0+k}), we perform a standard Gaussian Elimination on G to find a subset of columns that generates $G_{i_0+k} = G_{i_1+k} \oplus \ldots \oplus G_{i_l+k}$ for some $i_1, \ldots, i_l$ and $l < n$. The indices $\{i_1, \ldots, i_l\}$ are refereed to as a $RepairList$. All possible Repair Lists can be computed off-line and stored on the client side to avoid Gaussian Elimination in run time. Then, message $\mathbb{M}_{i_0}$ can be recovered by

$$\begin{aligned} \mathbb{M}_{i_0} &= [\mathbb{S}_1, \ldots, \mathbb{S}_k, \mathbb{X}_1, \ldots, \mathbb{X}_t] \cdot G_{i_0+k} \\ &= [\mathbb{S}_1, \ldots, \mathbb{S}_k, \mathbb{X}_1, \ldots, \mathbb{X}_t] \cdot (G_{i_1+k} \oplus \ldots \oplus G_{i_l+k}) \\ &= \mathbb{M}_{i_1} \oplus \ldots \oplus \mathbb{M}_{i_l} \end{aligned} \tag{8}$$

This repair algorithm requires a series of XOR operations on messages and can be carried out sequentially at servers $i_1, \ldots, i_l$ in the $RepairList$, each of which XORs the current version of $\mathbb{M}_{i_0}$ and its own message, and then forwards the result to the next server. Repairing server data loss in our distributed storage protocol can be made distributive and bandwidth efficient. The algorithm is summarized Algorithm (1c).

VI. Our Extended Algorithm for Reducing Complexity

When file size f is large, the algorithm described in Section IV may not be efficient, primarily due to its use of random vectors for providing confidentiality. First, the algorithm requires XOR of file $\mathbb{F}$ with random vectors, whose total size can be several times larger than the file. For example, storing $f = 10MB$ on $n = 7$ servers, with $n = 7, k = 1, t = 3$ in the $(8, 4, 4)$ binary code example in Section V, requires generating random vectors of size $tf/k = 30MB$. It also results in storing messages of a total size $nf/k = 70MB$ on distributed servers, which expands the amount of data stored and retrieved by a factor of $n/k = 7$, all contributed by random vectors. One could choose a larger $k > 1$ to split that data into smaller pieces, and thus, reduce the overhead. However, as we will show in the security analysis in Section VII, choosing $k = 1$ maximizes the security and reliability of the algorithm. Thus, this algorithm suffers from a tradeoff between security and reliability of data and storage overhead, when file size f gets large.

To solve this problem, in this section we propose an extend keyless algorithm which combines the basic algorithm in Section V and data encryption. The main idea is that instead of XORing file $\mathbb{F}$ with random vectors, we encrypt file $\mathbb{F}$ with an auxiliary key $\mathbb{K}$ of size b, and then XOR the key with random vectors to provide confidentiality. Therefore, the size of required random vectors is now determined by auxiliary key size b and is independent of file size. In order to store the auxiliary key along with the file on distributed servers, each message $\mathbb{M}_i$ now contains a data segment as well as a key segment. Integrity and availability are still be protected using the same binary error control coding in Section V. To keep our notations consistent, we describe the extended algorithm with respect to the same $(n + k, t + k, s)$ binary linear error control in Section V.

In the storing phase, we encrypt file $\mathbb{F}$ of size f by an auxiliary key $\mathbb{K}$ of size b. The auxiliary key is divided into k equal-length segments, $\mathbb{K}_1, \ldots, \mathbb{K}_k$, such that each segment has b/k bits. t pseudo-random length-(b/k) vectors $\mathbb{X}_1, \ldots, \mathbb{X}_t$ are constructed at the client side. We stack the k key segments and the t random vectors as columns in an input matrix $[\mathbb{K}_1, \ldots, \mathbb{K}_k, \mathbb{X}_1, \ldots, \mathbb{X}_t]$. Since data confidentiality is now protected by encryption, no random vectors are required for the file. We divide the encrypted file $\bar{\mathbb{F}}$ into $k + t$ equal-length segments, $[\bar{\mathbb{S}}_1, \ldots, \bar{\mathbb{S}}_{t+k}] = \bar{\mathbb{F}}$, to create another input matrix of $t + k$ columns. Let $G_{k+1:k+n}$ be the submatrix of G by removing the first k columns. We derive the messages by applying the same error control coding to the two input matrices, i.e.,

$$[\mathbb{M}_1, \ldots, \mathbb{M}_n] = \begin{bmatrix} \bar{\mathbb{S}}_1 & \ldots & \bar{\mathbb{S}}_k & \bar{\mathbb{S}}_{k+1} & \ldots & \bar{\mathbb{S}}_{t+k} \\ \mathbb{K}_1 & \ldots & \mathbb{K}_k & \mathbb{X}_1 & \ldots & \mathbb{X}_t \end{bmatrix} \cdot G_{k+1:k+n}, \tag{9}$$

In a more explicit form, this is equivalent to

$$\mathbb{M}_i = \begin{bmatrix} (g_{1,i}\bar{\mathbb{S}}_1) \oplus \ldots \oplus (g_{k,i}\bar{\mathbb{S}}_k) \oplus (g_{1+k,i}\bar{\mathbb{S}}_{k+1}) \oplus \ldots \oplus (g_{t+k,i}\bar{\mathbb{S}}_{t+k}) \\ (g_{1,i}\mathbb{K}_1) \oplus \ldots \oplus (g_{k,i}\mathbb{K}_k) \oplus (g_{1+k,i}\mathbb{X}_1) \oplus \ldots \oplus (g_{t+k,i}\mathbb{X}_t) \end{bmatrix}. \tag{10}$$

(2a) Storing Phase:

Public parameters: k, t, n, G.

Private inputs: $\mathbb{F}$.

1) Encrypt file $\mathbb{F}$ into $\bar{\mathbb{F}}$ using a random key $\mathbb{K}$.
2) Divide key $\mathbb{K}$ into k segments, each of b/k bits.
$$[\mathbb{K}_1, \ldots, \mathbb{K}_k] \leftarrow \mathbb{K}$$
3) Generate t pseudo-random vectors, each of b/k bits.
$$[\mathbb{X}_1, \ldots, \mathbb{X}_t] \leftarrow random()$$
4) Divide encrypted file $\bar{\mathbb{F}}$ into $t+k$ segments, each of $f/(t+k)$ bits.
$$[\mathbb{S}_1, \ldots, \mathbb{S}_{t+k}] \leftarrow \bar{\mathbb{F}}$$
5) Apply bit matrix multiplication of $[b/k + f/(t+k)] \times (t+k)$ and $(t+k) \times n$ matrices to compute n messages by $[\mathbb{M}_1, \ldots, \mathbb{M}_n] \leftarrow$
$$\begin{bmatrix} \bar{\mathbb{S}}_1 & \ldots & \bar{\mathbb{S}}_k & \bar{\mathbb{S}}_{k+1} & \ldots & \bar{\mathbb{S}}_{t+k} \\ \mathbb{K}_1 & \ldots & \mathbb{K}_k & \mathbb{X}_1 & \ldots & \mathbb{X}_t \end{bmatrix} \cdot G_{k+1:k+n}$$
6) Destroy $\mathbb{F}, \bar{\mathbb{F}}, \mathbb{K}_1, \ldots, \mathbb{K}_k, \mathbb{S}_1, \ldots, \mathbb{S}_{t+k}, \mathbb{X}_1, \ldots, \mathbb{X}_t$ on the client side by overwrite operations.
7) Send message $\mathbb{M}_i$ to server i for storage, $\forall i$.
8) Destroy $\mathbb{M}_1, \ldots, \mathbb{M}_n$ on the client side by overwrite operations.

(2b) Retrieving Phase:

Public parameters: T_{out}, f, b, k, t, n, H, $Table$.

Private inputs: None.

1) Initialize auxiliary variables $A \leftarrow 0$, $\tilde{H} \leftarrow 0$, $e \leftarrow 0$, $i \leftarrow 1$, $r \leftarrow 0$.
2) Broadcast message retrieving requests to the n servers.
3) Apply Steps (3)-(8) in Algorithm (1b) to recover the encrypted file and the auxiliary key:
$$\mathbb{K} \leftarrow [\mathbb{K}_1, \ldots, \mathbb{K}_k]$$
$$\bar{\mathbb{F}} \leftarrow [\mathbb{S}_1, \ldots, \mathbb{S}_{t+k}]$$
4) Decrypt using the auxiliary key $\mathbb{K}$.
5) Output file $\mathbb{F}$.

Fig. 2. Our Algorithm 2 for keyless distributed storage, including storing phase and retrieving phase.

where the top part of each message $\mathbb{M}_i$ contains an XOR of data segments, while the lower part contains an XOR of key segments and random vectors. It is easy to see that the extended algorithm provides two levels of confidentiality: perfect secrecy for the auxiliary key $\mathbb{K}$ and cryptographic secrecy for the file $\mathbb{F}$.

The retrieving phase is almost identical to Algorithm (1b) in Section V, since the same error control coding is used. By looking up syndrome table and comparing different error patterns, we can recover the encrypted file $\bar{\mathbb{F}}$ and the auxiliary key $\mathbb{K}$, which give the original file $\mathbb{F}$ after a decryption. The algorithm is summarized Algorithm (2a) and Algorithm (2b).

VII. Security and Reliability Analysis

A. Analyze Security Using Resilience Vectors

As discussed in Section III, previous algorithm relying on secret keys cannot achieve perfect secrecy and results in $c = 0$, since cryptographic operations are based on computational hardness assumptions. Our distributed storage algorithms are able to achieve a variety of resilience vectors $(c, d, e)_n$ by choosing different error control code parameters. In particular, Algorithm 1 achieves perfect secrecy. Its security and reliability is characterized in Theorem 1 below. The proof is summarized in Appendix.

Theorem 1: With a linear binary error control code $(n+k, t+k, s)$ and its dual code $(n+k, n-t, s')$, Algorithm 1 achieves resilience vectors $(c, d, e)_n$ for any $c = s' - k - 1$ and $2d + e = s - k - 1$. In particular, when both codes are maximum distance separable (MDS), the proposed algorithm achieves an optimal resilience vector bounded by $c + 2d + e = n - k - 2$.

Remark 1: Choice of parameters for Algorithm 1. Theorem 1 provides a security and reliability analysis by proving the resilience against the three classes of attacks. If a target resilience vector is given, how should we choose parameters for our algorithm to achieve it? Besides the existence of $(n + k, t + k, s)$ and its dual code

$(n+k, n-t, s')$, the following inequalities must be satisfied when choosing algorithm parameters $\{n, t, k, s, s'\}$:

$$\begin{aligned} &2d + e + c \le n - k - 2, \\ &2d + e \le s - k - 1, \\ &c \le s' - k - 1 \le t. \end{aligned} \tag{11}$$

The first three inequality are a direct results from Theorem 1. The last inequality $s' - k - 1 \le t$ is intuitive: Because in the storing phase of our algorithm, t random vectors are used as one-time pads to generate messages, it can not provide confidentiality for more than t message revealing. These inequality constraints can be used to give an estimation on the minimum number of servers required to achieve a certain resilience target.

Remark 2: From Theorem 1, it is easy to see that reducing k improvements on both reliability and security, while it makes the storage system less efficient, because for fixed file size f, a larger k results in smaller sizes of messages and random vectors, i.e., $m = f/k$. When file size f is small, we can choose $k = 1$. This leads to a special case of Theorem 1, where the distributed algorithm achieves $2d + e \le s - 2$ and $c \le s' - 2$, with $2d + e + c = n - 3$ at optimum. This is the optimal reliability and security that can be achieved by the proposed distributed storage algorithm. In this paper, we only focus on binary error control codes in GF_2, although all ideas can be extended to codes with higher dimensions. For example, if a linear ternary error control code is used, the achievable reliability and security resilience can be improved, since the code achieves a larger Hamming distance.

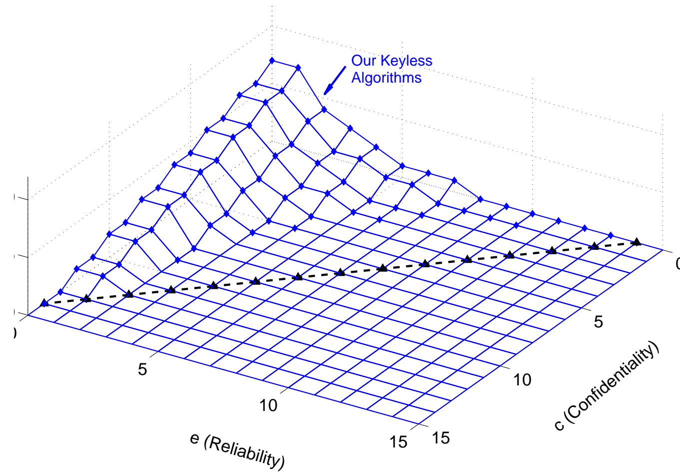

Fig. 3. Plot the set of all possible resilience vectors achieved by our keyless Algorithm 1 for $n = 14$ servers. Our algorithm is able to achieve strictly positive $(c, d, e)_n$ vectors.

Depending on the existence of error codes with different parameters, the set of all possible resilience vectors in Theorem 1 forms a 3-dimensional region, which quantifies a tradeoff among confidentiality, integrity, and reliability of our Algorithm 1, as shown in Figure 3 for $n = 14$ servers. It can be seen that since our algorithm can defend against all three classes of violations, its region of achievable resilience vectors absorbs many prior work as special cases in lower dimensions. With the same $(n+k, t+k, s)$ linear code and $(n+k, n-t, s')$ dual code, in Table I our proposed Algorithms 1,2 are compare to previous algorithm using secret keys, as discussed in Section III. It can be seen that although not achieving perfect secrecy, Algorithm 2 improves previous algorithms due to its keyless feature.

B. Analyze Probability of Secure and Reliable Storage

Previous algorithms using secret keys are vulnerable to client-side security breaches, since an adversary just need to attack the integrity and availability of the encrypting key to prevent the client from accessing his data, independent of servers being attacked. For confidentiality, if the secret key is revealed, revealing breach on any one server gives out its stored data. Intuitively, when the probability of client-side security breaches is high, our proposed should outperform previous algorithm due to their keyless features.

Approaches	Resilience vector $(c,d,e)_n$		Storage on each server	Require secret keys
	c	d,e		
Previous Algorithm in Section III	$c=0$	$d+e=s-1$	$f/(t+k)$	Yes
Our Algorithm 1	$c=s'-2$	$2d+e=s-2$	f/k	No
Our Algorithm 2	$c=0$	$2d+e=s-2$	$f/(t+k)$	No

TABLE I
COMPARING OUR KEYLESS DISTRIBUTED STORAGE ALGORITHMS WITH PREVIOUS ALGORITHM.

To derive a rigorous analysis of probability of secure and reliable storage, we assume that on the server side, the three classes of attacks have probabilities P_c, P_d, P_e to achieve independently a successful violation, such that a server remains unharmed with probability $P_0 = 1 - P_c - P_d - P_e$. Similarly, on the client side, the three classes of attacks have probabilities Q_c, Q_d, Q_e to achieve independently a successful violation. For simplicity, we use the same probability P_c for perfect secrecy violations and cryptographic secrecy violations. Therefore, security performance of our Algorithm 1 is underestimated.

From the analysis in Section VII.A, it is straightforward to derive the probability of secure and reliable storage for our Algorithms 1,2 as follows

$$P_n = \sum_{i\le s'-2, j+2l\le s-2} \binom{n}{i,j,l} P_c^i P_e^j P_d^l P_0^{n-i-j-l}, \tag{12}$$

which only depends on server-side security breaches. For previous algorithm using secret keys, its probability of secure and reliable storage also depends on probability of client-side security breaches. It is given by

$$P_n = (1 - Q_c - Q_d - Q_e) \cdot \sum_{i\le n, j+l\le s-1} \binom{n}{i,j,l} P_c^i P_e^j P_d^l P_0^{n-i-j-l} + Q_c \cdot \sum_{j+l\le s-1} \binom{n}{j,l} P_e^j P_d^l P_0^{n-j-l}, \tag{13}$$

where the first term on the right hand-side correspond the event that the secret key is secure, and the second term corresponds to the event of a successful confidentiality attack on the secret key at client-side.

We plot in Figure 4 the probability P_n of secure and reliable data storage in Equations () and (), for fixed client-side attack probability $Q_c = Q_d = Q_e = 2\%$ and different server-side violation probability $P_c = P_d = P_e$. All algorithms use a $(15, 5, 7)$ linear code with $k = 1$. It is observed that for reasonable server-side attack probability, e.g., $P_c = P_d = P_e < 5\%$, our keyless distributed storage algorithm exhibits a significant improvement over previous algorithm using secret keys. This observation substantiates our conjecture that the use of secret key in previous storage algorithm introduces vulnerability if client-side attack probability is non-negligible.

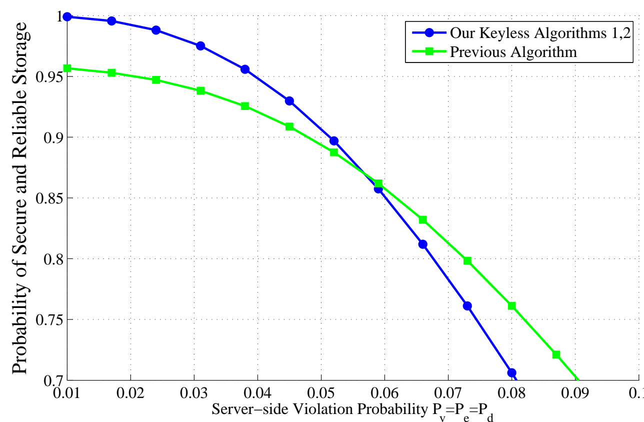

Fig. 4. Compare probability of secure and reliable data storage for different algorithms with $n = 14$ servers. Our algorithms achieve a significant improvement for reasonable server-side attack probability, e.g., $P_c = P_d = P_e < 5\%$.

C. Calibrate Our Algorithms Using Number of Nines

When the number of storage servers in a system is fixed, to maximize the the probability P_n by choosing parameters c, d, e, we consider an optimization problem over $\mathcal{R}_n$:

$$\begin{aligned} P_n^* = \quad & \max_{c,d,e} \sum_{i \le c, j \le e, l \le d} \binom{n}{i, j, l} P_c^i P_e^j P_d^l P_0^{n-i-j-l} \\ & \text{s.t.} \quad (c, d, e) \in \mathcal{R}_n \end{aligned} \tag{14}$$

where the optimal solution P_n^* is a function of the server number n and the three attack probabilities (P_c, P_d, P_e) on the server side. In general, problem (14) can be solved off-line by an exhaustive search over Pareto optimal resilience vectors in $\mathcal{B}_n$ (i.e. vectors lies on the boundary of $\mathcal{R}_n$), before a distributed storage system is deployed.

Theorem 2: For a resilience vector $(c, d, e)_n$ and given attack probabilities $P_c, P_d, P_e < \frac{1}{n}$ on the server side, the probability of secure and reliable data storage on the client side is lower bounded by

$$P_n(c, d, e) \ge 1 - \sum_{x \in \{c,d,e\}} \binom{n}{x+1} \frac{P_x^{x+1}(1 - P_x)^{n-x}}{1 - nP_x/v}. \tag{15}$$

Further, when $P_c = P_d = P_e = P$ and $nP << 1$, we have

$$P_n(c, d, e) \approx 1 - \binom{n}{\min(c, d, e) + 1} P^{\min(c,d,e)+1}. \tag{16}$$

Approximately, $P_n(c, d, e)$ is determined by the smallest resilience element $\min(c, d, e)$ in a vector $(c, d, e)_n$.

According to this theorem, resilience vectors can provide general guidelines for designing a distributed storage system. When $P_c = P_d = P_e = P$ are equal, the probability of secure and reliable storage is primarily determined by the minimum individual resilience elements $\min(c, d, e)$. Therefore, the ability to defend against all three classes of attacks at the same time is critical.

The formula for 'number of nines' as a metric for Availability is the number of '9' digits after the decimal point for the probability of retrieving the desired data when requested by the client-side. For example, '5 nines' is defined as:

$$\mathrm{Prob}\{\mathrm{data\,is\,securely\,retrieved\,when\,requested}\} = .99999.$$

This translates to a down time (with Probability = 0.00001) of approximately 5 minutes per year, when the system is not available.

We provide a much stronger definition of 'nines' in our distributed storage system. We require the data stored to remain confidential, be free from modification (by malicious attacker or by noisy channel or environment) and be free from malicious attacks on availability or from packet dropping due to network operation. We translate the 'nines' requirement to the following equation

$$\mathrm{Nines} = \lfloor \log_{10} \tfrac{1}{1 - P_n(c,d,e)} \rfloor. \tag{17}$$

The 'number of nines' can be regarded as a function of the server number n and the three server violation probabilities (P_c, P_d, P_e). For $P_c = P_d = P_e = P$ and $nP << 1$, using equation (16), we derive the number of 'nines' achieved by our distributed storage algorithm

$$\mathrm{Nines} \quad \approx [\min(c, d, e) + 1] \log_{10} \frac{1}{P} + C_n, \tag{18}$$

where $\min(c, d, e)$ is the minimum individual resilience and C_n is a proper constant independent of server violation probability P. Therefore, the number of 'number of nines' is a linear function of the the minimum individual resilience.

Suppose that a $(15, 5, 7)$ code with a $(15, 10, 4)$ dual code is used in our algorithm for storing data on $n = 14$ servers. Using $\max_{\mathcal{R}_n} \min(c, d, e) + 1 = 3$ in this case, equation (18) implies that in order to improve data storage security and reliability on the client side and get one more 'nine', the server violation probability P on the server side have to be reduced by a factor of $10^{\frac{1}{3}} = 2.2$. This is verified by our simulation in Figure.5, where the number of 'nines' is plotted for different server attack probability $P = P_c = P_d = P_e$. It can be seen that to go from 5 to 6 'nines', the violation probability on the server side has to be reduced from 0.8% to 0.3%.

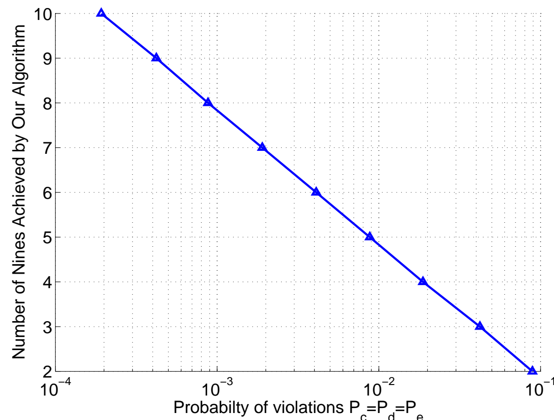

Fig. 5. The number of 'nines' achieved by our algorithm for different server violation probability $P_c = P_d = P_e$. Reducing server attack probability by a factor of 2.2 improves security and reliability by one more 'nine'.

VIII. Performance and Cost Evaluation

We evaluate the implementation of our distributed storage protocol, with respect to computation overhead and memory overhead as seen from the client-device side. In this evaluation, we are not concerned with the time and storage required by a storage server. Rather, the performance and implementation cost to the user of the distributed storage system is the main concern and what we evaluate. We show that our keyless storage algorithms has a low complexity and compares favorably with a previous algorithm that implements AES-256 encryption, SHA-2 hash, and erasure coding, for achieving confidentiality, integrity, and reliability, respectively.

We start with quantifying the computation and memory overhead of the Algorithms 1,2. Since the Algorithm 1 operates in Gf_2, all required operations are binary. A breakdown of the storage protocol in Section IV shows that it consists of four basic operations: bit matrix multiplication (BMM), table lookup, construction of pseudo-random vectors (RV), and population count ($popcnt$) and comparison ($cmpare$), while its storage overhead consists of three parts: a syndrome table, generator and parity check matrices, and auxiliary vectors for storing the random vector and syndrome vectors. Algorithm 2 employs the encoding and decoding procedures in Algorithm 1 for key segments and uses an extra AES-256 encryption for data segments. We derive the overhead of Algorithm 1,2 using these metrics.

Algorithms		**Computation in** $million\ CPU\ cycles$					**Memory in** KB			
		BMM	RV	Lookup	$popcnt$ $cmpare$	AES	Table	Coding	Auxiliary	AES
Algorithm 1 $100KB$	Storing Retrieving	38.2 35.3	83.6 -	- 554	- 1089	- -	- 1.92	0.01 0.02	1900 2400	- -
Algorithm 2 $100KB$	Storing Retrieving	6.1 5.6	0.03 -	- 107	- 178	597 656	- 1.92	0.01 0.01	360 480	320 320

TABLE II
This table evaluates the breakdown of overhead for our distributed storage algorithm, implemented for $n = 14$ servers and data size $m = 100KB$. Its computation overhead is measured with respect to the five basic operations, and its memory overhead is measured with respect to four types of variables. Both Algorithm 1,2 exhibit low-overhead with a maximum of linear growth in data size m.

We start with analyzing the complexity of Algorithm 1. Let $m = f/k$ be the size of messages. In the storing phase, t pseudo-random vectors need to be generated in Step 2. Step 3 for computing n messages needs $nm(t+k-1)$ binary XOR operations. The memory overhead is $m(n+t+k)$ bits for auxiliary vectors $\mathbb{M}_1, \dots, \mathbb{M}_n$, $\mathbb{X}_1, \dots, \mathbb{X}_t$,

$\mathbb{S}_1, \ldots, \mathbb{S}_k$ and $(k+t) \times (k+n)$ bits for the generator matrix. In the retrieving phase, the bit matrix multiplication in Step 6 for computing the syndrome matrix needs $m(n-t)(n-e-1)$ binary XOR operations, while Steps 5 and 7 use $(n+t)(n-k)$ and $m(n+k-t)$ binary XOR operations, respectively. In step 7, two syndrome table lookups, two *popcnt* instructions, and one *cmpare* instruction are performed for each loop, resulting in a total overhead of $2m$ table lookups and $3m$ instructions. In the retrieving phase, the memory overhead for auxiliary vectors is $m(k+n-1)$ bits for messages $\mathbb{S}_1, \ldots, \mathbb{S}_k$, $\hat{\mathbb{M}}_1, \ldots, \hat{\mathbb{M}}_{n-1}$, $m(n-t)$ bits for syndrome matrix R^0, $(2n+k-1)(n-t)$ bits for H and $\tilde{H}$, and $n-t+3(n+k)$ bits for vectors $\tilde{r}$, A, $\vec{t0}$, and $\vec{t1}$. Finally, the syndrome table requires $(n+k)2^{n-t}$ bits, because there can be no more than 2^{n-t} different syndromes of $n-t$ bits and each entry in the syndrome table is an error vector of length $n+k$ bits.

Since Algorithm 2 employs the same steps of Algorithm 1 except for a different message size, we can reuse the previous complexity analysis and replace the message size by $m_2 = f/(t+k) + b/k$ in all the results above. Because the auxiliary key size b is in general much smaller than the file f, ruffly speaking, Algorithm 2 can reduce the complexity of Algorithm 1 by a factor of $1 + t/k$, while an extra overhead for AES encryption of file size b is be added in Algorithm 2.

We first evaluate the breakdown of overhead of our distributed storage protocol with respect to the five basic operations and the four basic types of variables. These evaluations were done using C code on a 1.60 GHz Pentium 4 with 1.0 GB memory, typical of future hand-held devices that make up most of future client devices and have much less computation power than today's top-line desktops. As in Algorithms 1,2, we considered data size of $m = 100KB$ and used a $(15, 5, 7)$ binary code with $t = 4$ and $k = 1$, for storing the data on $n = 14$ distributed servers.

Table II presents the results of this evaluation. First, the table shows that our distributed storage algorithms has low overhead. For instance, Algorithm 1 takes 120 m-cycles to store a $100KB$ data on $n = 14$ servers and 1600 m-cycles to retrieve it, while the total required memory is abo2ut $4MB$. Algorithm 2 further reduces this complexity to about 800 m-cycles and $0.7MB$ for both storing and retrieving. In Algorithm 1, the computation overhead of the protocol is mainly contributed by the cost of the table lookups and the *popcnt* operations, which are about 21% and 68% of the retrieval phase execution time, respectively. These are all scaled down by a factor of $1 + t/k = 5$, since a smaller message size is used in Algorithm 2. For memory overhead, since our algorithms make uses of error control decoding rather than erasure/network decoding, the extra memory cost is introduced by the storage of the syndrome table on the client side. It is a constant $1.9KB$ for the $(15, 5, 7)$ binary code and only takes 1% of the total memory overhead in Algorithms. This evaluation verifies the low-complexity property of our distributed storage protocol. As the data size m increases, the total overhead exhibits a linear growth-rate in data size m.

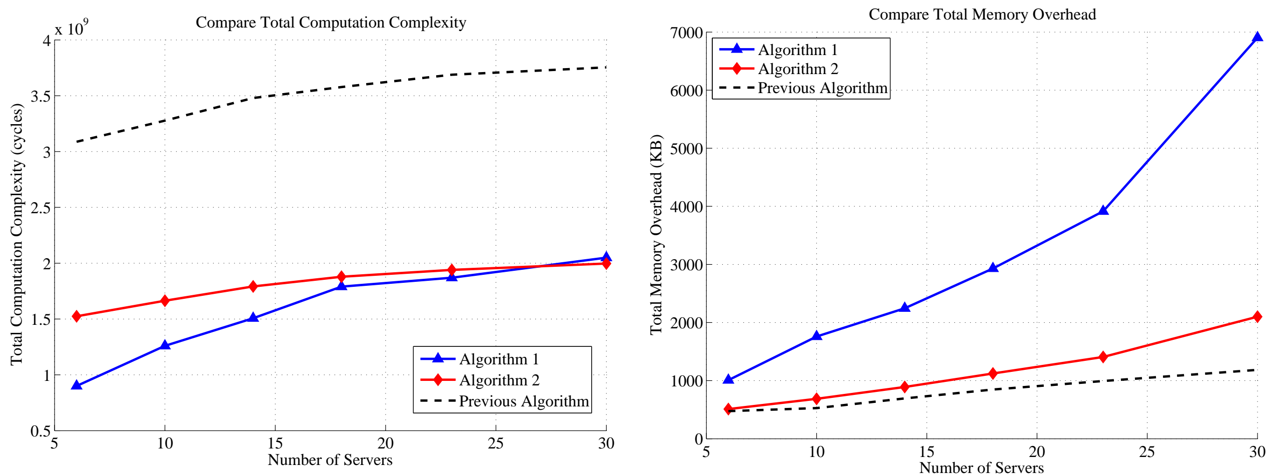

Fig. 6. For $n = 6, 10, 14, 18, 22, 26, 30$ servers and data size $m = 100KB$, these three figures compare overhead and throughput of our distributed storage protocol with $k = 1$ to a simple cryptographic scheme, which uses 256-bit AES encryption for confidentiality, 256-bit SHA-2 hash for integrity, and erasure codes for reliability. The computation overhead and throughput of our non-cryptographic protocol compares favorably to the cryptographic scheme, while extra memory is required for storing the random vectors and the syndrome vectors. This extra cost can be reduced by choosing a larger k in our protocol, thus dividing the original data into smaller segments.

For $n = 6, 10, 14, 18, 22, 26, 30$ servers, we compare the total overhead of our two keyless algorithms to the previous algorithm described in Section III, which employs erasure codes to construct messages for reliability, and 256-bit AES encryption and 256-bit SHA-2 hash to encrypt and sign the messages on the client side for confidentiality and integrity, respectively. Figure 6 presents the results of this evaluation. It is shown that the computation overhead of our keyless protocols is much lower than the computation overhead of the cryptographic scheme. For instance, it takes about 1600 m-cycles (i.e., about $1s$ in our implementation) for both of our algorithms to store and retrieve a file of size $100KB$ on $n = 14$ servers, resulting in an improvement of a factor of 2.3 in contrast to 3600 m-cycles (i.e., about $2.2s$) required by the previous algorithm using secret keys. For $n = 14$ servers, an extra memory overhead of $1.4MB$ is observed in Algorithm 1, while Algorithm 2 has similar memory overhead as the previous algorithm. If the perfect secrecy that is provided by Algorithm 1 is not necessary, Algorithm 2 becomes a more suitable candidate for practical implementation of keyless distributed storage.

IX. Conclusion

We develop two algorithms for secure and reliable distributed storage without relying on secret keys. Both algorithms make novel use of binary error control codes to protect and store files, and provide security and reliability at low complexity. By trading-off reliability, confidentiality, and integrity in a unified 3D tradeoff space, our algorithm achieves a significant security and reliability improvement, as observed in simulation and quantified by analysis. By sacrificing the perfect secrecy property, the extended algorithm further reduce the required memory overhead and is more desirable for practical implementations.

References

[1] V. Kher and Y. Kim, “Securing Distributed Storage: Challenges, Techniques, and Systems”, *in Procedings of StorageSS*, 2005.

[2] M. Blaze, “A cryptographic file system for UNIX”, *In Proceedings of ACM CCS*, 1993.

[3] E. Goh, H. Shacham, N. Modadugu, and D. Boneh, “SiRiUS: Securing Remote Untrusted Storage”, *In Proceedings of NDSS Symposium*, 2003.

[4] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Plutus, “Scalable Secure File Sharing On Untrusted Storage”. *In Proceedings of USENIX File and Storage Technologies*, 2003.

[5] J. Li, M. Krohn, D. Mazires, and D. Shasha, “Secure untrusted data repository (SUNDR)”, *in Procedings of OSDI*, 2004.

[6] A. Shamir, “How to Share a Secret”, *Communications of the ACM*, 1979.

[7] M. Fitzi, J. Garay, S. Gollakota, C. Rangan, and K. Srinathan, ”Round-Optimal and Efficient Verifiable Secret Sharing”. *in Procedings of Third Theory of Cryptography Conference*, 2006,

[8] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, “Pond: the OceanStore Prototype”, *in Procedings of USENIX File and Storage Technologies*, 2003.

[9] R. Bhagwan, K. Tati, Y. C. Cheng, S. Savage, and G. M. Voelker, “Total Recall: System Support for Automated Availability Management”, *in Procedings of NSDI*, 2004.

[10] C. Huang and L. Xu, “STAR: An efficient coding scheme for correcting triple storage node failures”, *in Procedings of USENIX File and Storage Technologies*, 2005.

[11] H. Xia and A. A. Chien, “RobuSTore: A Distributed Storage Architecture With Robust And High Performance”, *in Proceedings of ACM/IEEE Conference on Supercomputing*, 2007.

[12] C. Gkantsidis and P. Rodriguez, “Network coding for large scale content distribution”, *in Proceedings of IEEE Infocom*, 2005.

[13] T. Ho, M. M’edard, R. Koetter, D. Karger, M. Effros, J.Shi, and B. Leong, “A random linear network coding approach to multicast”, *IEEE Transaction on Infomation Theory*, no. 10, pp.4413-4430, 2006.

[14] S. Lin and D.J. Costello, “Error Control Coding”, *Prentice Hall Publisher*, 2nd eddtion, June, 2004.

[15] X. Zhang, G. Neglia, J. Kurose, and D. Towsley, “On the benefits of random linear coding for unicast applications in disruption tolerant networks”, *in Procedings of Second Workshop on Network Coding, Theory, and Applications*, 2006.

[16] D. Wang, Q. Zhang, and J. Liu, “Partial network coding: Theory and application for continuous sensor data collection”, *in Procedings of Fourteenth IEEE International Workshop on Quality of Service*, 2006.

[17] A.G. Dimakis, P.B. Godfrey, M.J. Wainwright, and K. Ramchandran, “Network Coding for Distributed Storage Systems”, *in Proceedings of IEEE INFOCOM*, 2007.

[18] C. Shannon, “Communication Theory of Secrecy Systems”. *Bell System Technical Journal*, 1949.

[19] T. Lan, R. Lee, and Mung Chiang, “Multi-path Key Establishment Against REM Attacks in Wireless Ad Hoc Networks”, *In proceedings of IEEE GLOBECOM*, December 2009.

[20] Y. Hilewitz, Y.L. Yin, R.B. Lee, “Accelerating the Whirlpool Hash Function Using Parallel Table Lookup and Fast Cyclical Permutation”, *in Proceedings of the 15th Fast Software Encryption Workshop (FSE)*, 2008.

[21] C.K. Ngai and R. W. Yeung, “Secure error-correcting (sec) network codes”, *in 2009 Workshop on Network Coding, Theory and Applications (NetCod 2009)*, June 2009.

[22] A. Subramanian and S.W. McLaughlin, “MDS codes on the erasure-erasure wiretap channel”, *Avaialble on-line at http://arxiv.org/abs/0902.3286*.

APPENDIX

A. Proof of Theorem 1

Proof: To show that the critical data $\mathbb{F}$ can be successfully retrieved, we notice that each row of the message matrix $[\mathbb{S}_1, \ldots, \mathbb{S}_k, \mathbb{M}_1, \ldots, \mathbb{M}_n]$ is a valid codeword for the $(n+k, k+t, s)$ error control code. According to coding theory, a linear $(n+k, k+t, s)$ error control code can correct up to $\lfloor \frac{s-1}{2} \rfloor$ errors with a syndrome decoding. In the retrieving algorithm, we choose the $k+e$ erased messages to be all zeros and all ones respectively. Because the error control code is binary, one of the two choices introduces less than $\lfloor \frac{k+e}{2} \rfloor$ new errors, and thus leads to totally $d + \lfloor \frac{k+e}{2} \rfloor$ errors, which can be efficiently corrected by the syndrome decoding in the retrieving algorithm, if the following is satisfied:

$$d + \lfloor \frac{k+e}{2} \rfloor = \lfloor \frac{2d+k+e}{2} \rfloor \leq \lfloor \frac{s-1}{2} \rfloor \tag{19}$$

This establishes $2d + e \leq s - k - 1$ as a sufficient condition for recovering $\mathbb{S}_1, \ldots, \mathbb{S}_k$ from the retrieved messages.

Next, we show that the critical data $\mathbb{F}$ remains completely unknown to attacks. Without loss of generality, we assume that the first c messages $\mathbb{M}_1, \ldots, \mathbb{M}_c$ are revealed to attackers. According to the proposed algorithm, attackers have c linear equations

$$\mathbb{M}_i = \sum_{j=1}^{k} \mathbb{S}_j g_{j,i} + \sum_{j=1}^{t} \mathbb{X}_j g_{k+j,i}, \ \forall i = 1, \ldots, c \tag{20}$$

for analyzing $\mathbb{S}_1, \ldots, \mathbb{S}_k$ and $\mathbb{X}_1, \ldots, \mathbb{X}_t$ from the revealed messages. Because the dual error control code $(k + n, n-t, s')$ has distance s', coding theory shows that any $s'-1$ columns of the G matrix are linearly independent. We can then prove that the coefficients for $\mathbb{X}_1, \ldots, \mathbb{X}_t$ in the c linear equations (20) form a full rank matrix, i.e.

$$\text{Rank} \begin{bmatrix} g_{k+1,1} & \cdots & g_{k+1,c} \\ \vdots & \ddots & \vdots \\ g_{k+t,1} & \cdots & g_{k+t,c} \end{bmatrix} = c. \tag{21}$$

This implies that if we regard $\mathbb{M}_1, \ldots, \mathbb{M}_c$ and $\mathbb{S}_1, \ldots, \mathbb{S}_k$ as known constants in (20), then (20) gives a set of c linear independent equations (due to (21)) with $t > c$ unknowns, i.e. $\mathbb{X}_1, \ldots, \mathbb{X}_t$. Therefore, when vectors $\mathbb{X}_1, \ldots, \mathbb{X}_t$ are generated randomly from a uniform distribution independent of the information vectors, for any observation $[\mathbb{M}_1, \ldots, \mathbb{M}_c] = \hat{M}$ and realization $\hat{S}$, we have

$$\begin{aligned} &\text{Prob}\left\{ [\mathbb{S}_1, \ldots, \mathbb{S}_k] = \hat{S} \middle| [\mathbb{M}_1, \ldots, \mathbb{M}_c] = \hat{M} \right\} \\ &= \text{Prob}\left\{ [\mathbb{S}_1, \ldots, \mathbb{S}_k] = \hat{S} \right\}. \end{aligned} \tag{22}$$

From (22), we conclude that given messages $\mathbb{M}_1, \ldots, \mathbb{M}_c$, all critical data vectors $[\mathbb{S}_1, \ldots, \mathbb{S}_k]$ remains equally likely. Let $I(\cdot)$ be the mutual information function and $H(\cdot)$ be the entropy function. This means that

$$\begin{aligned} &I([\mathbb{S}_1, \ldots, \mathbb{S}_k], [\mathbb{M}_1, \ldots, \mathbb{M}_c]) \\ &\quad = H([\mathbb{S}_1, \ldots, \mathbb{S}_k]) - H([\mathbb{S}_1, \ldots, \mathbb{S}_k] \,|\, [\mathbb{M}_1, \ldots, \mathbb{M}_c]) \\ &\quad = 0. \end{aligned}$$

In other words, attacks can obtain absolutely no information about the critical data $\mathbb{F}$, which is unconditionally confidential. ■