Received March 30, 2019, accepted May 20, 2019, date of publication June 3, 2019, date of current version June 26, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2920405

Gecko: A Resilient Dispersal Scheme for Multi-Cloud Storage

MENG YAN, JIAQI FENG, TRENT G. MARBACH, REBECCA J. STONES, GANG WANG, AND XIAOGUANG LIU
Nankai-Baidu Joint Laboratory, College of Computer, Nankai University, Tianjin 300350, China

Corresponding author: Gang Wang (wgzwp@nbjl.nankai.edu.cn)

This work was supported in part by the National Science Foundation of China under Grant 61872201, 61702521, 61602266, and U1833114, the Science and Technology Development Plan of Tianjin under Grant 17JCYBJC15300, 16JCYBJC41900, 18ZXZNGX00140, and 18ZXZNGX00200, the Fundamental Research Funds for the Central Universities, SAFEA: Overseas Young Talents in Cultural and Educational Sector, KLMDASR, and KLNDST in Tianjin.

ABSTRACT We have entered an era where copious amounts of sensitive data are being stored in the cloud. To meet the rising privacy, reliability, and verifiability needs, we propose Gecko, a multi-cloud dispersal scheme where: (a) the key used to encrypt the data file is the secret in a Latin-square-autotopism secret-sharing scheme, (b) data files and encryption keys are dispersed separately to multiple clouds, and (c) a blockchain-based integrity-check protocol is devised to pinpoint faulty data. Gecko enables fast and thorough key renewal: when a portion of the key (the secret) is leaked, we replace all shares of the partially-leaked secret without replacing the secret itself; this immediately resists targeted attack to certain file without re-encrypting the data file itself. Key renewal is further accelerated by the blockchain-based integrity check. We evaluate Gecko theoretically and experimentally against the traditional AONT-RS dispersal scheme, drawing two conclusions: 1) Gecko admits powerful key renewal and identification of damaged data, with a minor transfer overhead; and 2) Gecko performs key renewal three to five times faster than AONT-RS hybrid-slice renewal (the closest thing AONT-RS has to key renewal).

INDEX TERMS Blockchain, data recovery, dispersal scheme, integrity check, Latin square, multi-cloud.

I. INTRODUCTION

Cloud storage is convenient but gives rise to multiple concerns, such as privacy [1], fault tolerance [2], and verifiable integrity [3], particularly when the stored data is sensitive and large-scale. As one example among many, CareCloud[1] stores information such as electronic health records using Amazon S3 and Amazon Glacier [4]; for privacy protection and accuracy of analysis, CareCloud requires the outsourced data to be kept confidential, retrievable, and intact.

For sensitive systems such as CareCloud, a multi-cloud dispersal scheme [2] is beneficial, in which a sensitive file is fragmented into *file slices* and each of file slices (with corresponding integrity credentials) is dispersed to a distinct cloud service provider (CSP). With proper redundancy, the file remains retrievable even when some CSPs fail or are malicious. To fragment data files, symmetric encryption can be combined with erasure codes to balance security and the consequential overhead [5]–[7]. Regrettably, key-protection methods in current schemes are insufficient.

Ideally an encryption key that has been dispersed should have stronger protection than an encrypted file does, so the two dispersals should be performed independently; thus, combining file and key then dispersing, as in [6], [7], is not the most secure option. In [5], secret sharing is utilized independently for encryption keys, which guarantees high-level key secrecy; however, key recovery in case of failures or accidents is not efficient due to the recovery procedure requiring too much computation. In addition, a leaked key in some schemes may be useful in a number of attacks.

In this paper, we propose a multi-cloud dispersal scheme called Gecko, featuring a fast and thorough key-recovery process called key renewal, in which all slices of the key in danger are promptly discarded and renewed, just like a gecko discarding its tail when attacked. Using key renewal, a leaked key slice becomes permanently useless and a broken key slice is rapidly cured; it does not involve replac-

The associate editor coordinating the review of this manuscript and approving it for publication was Jun Huang.

[1]`www.carecloud.com`

ing the encryption key, thus data files do not need to be re-encrypted. Our key renewal is based on a secret sharing scheme in which the secret is an autotopism of a Latin square, the secret shares do not reveal partial information of the secret, and fragmentation of the autotopism can be performed many times. A blockchain-based protocol called NAP-check is devised to pinpoint faulty downloaded slices, reinforcing and accelerating recovery process; by publishing double signatures on a blockchain, NAP-check achieves non-repudiation, accountability, and public verifiability without trusting any third party. For data files Gecko offers the same level of security compared to the state-of-the-art [6].

In summary, Gecko:

- offers different levels of protection for files and encryption keys, in terms of secrecy and recoverability;
- utilizes symmetric encryption combined with a Reed-Solomon code for files, offering file-slice recovery;
- utilizes Latin-square-autotopism secret sharing for encryption keys, and
- uses a blockchain-based integrity-check protocol, to give fast and thorough key renewal.

The rest of this paper is organized as follows. Section II reviews related work and background. Section III explains the secret-sharing scheme used in Gecko. Section IV introduces Gecko in detail. Section V analyzes the threats against and the security of Gecko. Latency is evaluated theoretically in Section VI-A and experimentally in Section VI-B, in a simulated multi-cloud storage system. Section VII concludes this paper with some ideas for extending this research.

II. RELATED WORK AND BACKGROUND

A. SINGLE-CLOUD VS. MULTI-CLOUD STORAGE

In recent years, established cloud servers such as Google Docs [8], Amazon S3 [9], Microsoft Azure [10], and iCloud [11] have incurred malfunctions, indicating that single-cloud storage is not completely secure or reliable. Multi-cloud storage, where we disperse data to different clouds, is a trending way to solve single-cloud problems such as availability failures, malicious insiders [12], and vendor lock-in [13]. Further, multi-cloud storage requires a lower level of trust: instead of trusting a single provider, users trust that several providers do not collude.

We propose a dispersal scheme based on the multi-cloud such that: 1) each cloud stores a piece of ciphertext and a piece of the encryption key, which secures confidentiality; and 2) a (k, n) erasure code keeps data retrievable even if $n - k$ clouds fail.

B. DISPERSAL SCHEMES

Dispersal schemes are generally applied to distributed systems for data security. An (n, k, r) *dispersal scheme* fragments data D into n slices, each of which is dispersed to a different server or a different cloud. Any k of n slices can be used to reconstruct D, and without at least r slices it is impossible to deduce any information about D.

Many dispersal schemes aim to balance security against overhead. Secret sharing is characterized by high security and high overhead [14], whereas information dispersal algorithms (IDAs) are the contrary; it has been proposed to combine these two techniques. For example, in SSMS [5] the encryption key is fragmented via Shamir's secret sharing scheme (SSSS) [15] to get key slices, and the encrypted file is encoded using Rabin's IDA [16] to get file slices. AONT-RS [6] achieved lower overhead than SSMS by combining the file and the encryption key together via all-or-nothing transforms (AONT) [17] to get an AONT package (a hybrid data block), which is encoded by systematic Reed-Solomon (RS) erasure codes [18] to get hybrid slices (some of which contain both a part of the file and a part of the encryption key).

Gecko is similar to SSMS in terms of dispersing encryption keys independently, but we use a novel Latin-square-autotopism secret sharing (LASS) instead of SSSS, to offer powerfully fast and thorough key renewal which is not found in either SSMS or AONT-RS.

Note that although many keyless dispersal schemes have been proposed to avoid key management [19]–[21], key-based methods are still the primary technique as long as we design key-protection strategies cautiously, as Gecko does. Table 1 presents the comparison.

C. SECRET SHARING SCHEMES

In 1979, Blakley [22] and Shamir [15] independently introduced the concept of secret sharing, used to protect important but small-sized secrets such as encryption keys. In Blakley's scheme, the secret is the unique intersection point of (suitably chosen) hyperplanes in a vector space, and the shares are the hyperplanes. Shamir's Secret Sharing Scheme (SSSS) is based on polynomial interpolation, which achieves a high level of security at the expense of computational complexity and storage cost. In general, a (k, n) secret sharing scheme splits a secret into n *shares*, any k of which can be used to reconstruct the secret.

An early Latin-square secret-sharing scheme was given by Cooper, Donovan, and Seberry [23] (see also [24]) based on critical sets in 1994, but it was subsequently harshly criticized [25]–[27]. Afterward, changing the secret from the Latin square to one of its autotopisms was suggested in [28] and developed in [29]. Compared with critical sets, autotopisms of Latin squares are better understood [30]–[33] and are more practical to work with (e.g., determining if a partial Latin square has a completion as required by the original Latin square secret-sharing scheme is NP-complete [34]).

This paper applies the autotopism-based scheme LASS[29] to the proposed dispersal scheme, for a high-level of key protection. We introduce the basic notions and properties of LASS, as well as why we use it in Section III.

D. BLOCKCHAINS AND INTEGRITY CHECK

A *blockchain* is a distributed and tamper-resistant database composed of numerous identical copies of its entries, each

TABLE 1. Building blocks and properties of various dispersal schemes. Note that "LASS" is short for Latin-square-autotopism secret sharing, "RS code" is short for Reed-Solomon code, and b^{key} and b are the number of bytes of the key and the data.

Keyless	Encryption	File Secrecy and Resilience		Security	Storage Blowup
SSSS	×	perfect secret sharing		$(n,k,k-1)$	n
CAONT-RS	✓	AONT + RS code		$(n,k,k-1)$	$\frac{n}{k}+\frac{n}{k}\cdot\frac{b^{\text{key}}}{b}$
Rabin's IDA	×	non-systematic erasure code		$(n,k,0)$	$\frac{n}{k}$
Key-based	Key Type	Key Secrecy	Resilience	Security	Storage Blowup
SSMS	random key	SSSS	Rabin's IDA	$(n,k,k-1)$	$\frac{n}{k}+n\cdot\frac{b^{\text{key}}}{b}$
AONT-RS	random key	AONT	RS code	$(n,k,k-1)$	$\frac{n}{k}+\frac{n}{k}\cdot\frac{b^{\text{key}}}{b}$
Gecko	autotopism	LASS	RS code	$(n,k,k-1)$	$\frac{n}{k}+n\cdot\frac{b^{\text{key}}}{b}$

copy being maintained by a node in the blockchain network. All honest nodes comply with a consensus protocol instead of following a central party, where: (1) nodes need not trust each other, (2) even if a substantial minority of the nodes are dishonest or out of order, consensus still succeeds.

Many researchers have proposed to utilize blockchains to reform third-party-based systems involving access control [35], [36], public key infrastructure [37], and integrity checks [38]. In [38] Liu et al proposed blockchain-based integrity check for IoT data, where data owners publish *encrypted data-block hashes* on Ethereum's blockchain while simultaneously uploading data to the cloud. During download from the cloud, the corresponding encrypted data-block hashes are used by the data owners (or authorized data consumers) to verify data-block integrity. However, Liu et al's scheme only offers client-managed verification and fails to satisfy non-repudiation, which is very important as an integrity check for outsourced data.

Compared to [38], the blockchain-based NAP-check in Gecko not only offers client-initiated and CSP-initiated integrity check, but also achieves non-repudiation, accountability, and public verifiability simultaneously (which is why we call it NAP-check). Compared to traditional hash-based or signature-based methods [39], [40], NAP-check utilizes a decentralized network instead of a single trusted party, thus features higher reliability.

In industry, the decentralized storage network called Filecoin [41] stores data-auditing proofs in a blockchain to ensure that data is actually stored. Gecko is similar to Filecoin in that it stores "signatures" in the blockchain. Also similar to Gecko, Storj [42] locally encrypts and splits client data via erasure codes, and distributes the pieces to peers (storage nodes in a peer-to-peer network). However, Storj does not use a blockchain for verification (although they use Ethereum for payments of "Storj" tokens), and instead use probabilistic challenges called "audits." Filecoin's and Storj's networks involve many (untrusted) storage participants, whereas Gecko is aimed at storing data on a small number of clouds. Although clouds are also untrusted, they are more reliable than network participants, and Gecko also inherits the security and erasure-tolerance functionality of the individual clouds.

III. LATIN-SQUARE-AUTOTOPISM SECRET SHARING

Here we introduce basic notions about Latin squares. Particularly, we give special properties of LASS, which explains the benefits of dispersing encryption keys based on LASS. To ease understanding, we give an example of a 6-order Latin square and an autotopism.

A. NOTIONS ABOUT LATIN SQUARES

A *Latin square* $L=(l_{i,j})$ of order r is an $r\times r$ matrix with cells filled with symbols from the set $\mathbb{Z}_r$, such that each symbol occurs exactly once in each row and each column. An *entry* $(i,j,l_{i,j})$ implies the symbol $l_{i,j}$ is in the cell (i,j) in Latin square L.

A *partial Latin square* is a generalized Latin square which allows the possibility of empty cells, denoted $\cdot$, and where we require only that symbols occur at most once in each row and each column.

Let S_r be the symmetric group acting on $\mathbb{Z}_r$. An *isotopism* $\theta=(\alpha,\beta,\gamma)\in S_r\times S_r\times S_r$ is a kind of mapping that permutes each entry $(i,j,l_{i,j})\mapsto\big(\alpha(i),\beta(j),\gamma(l_{i,j})\big)$, thereby permuting L to $\theta(L)$. If $\theta(L)=L$, we call θ an *autotopism* of L.

The *orbit* of any entry $(i,j,l_{i,j})$ under an autotopism (α,β,γ) is the set of entries $\{\big(\alpha^k(i),\beta^k(j),\gamma^k(l_{i,j})\big):k\geq 0\}$. Orbits of a Latin square are mutually exclusive (i.e., their intersections are empty sets). If we choose exactly one entry from each orbit, we obtain a partial Latin square called a *contour*. With a contour and its corresponding autotopism, we can reconstruct the Latin square.

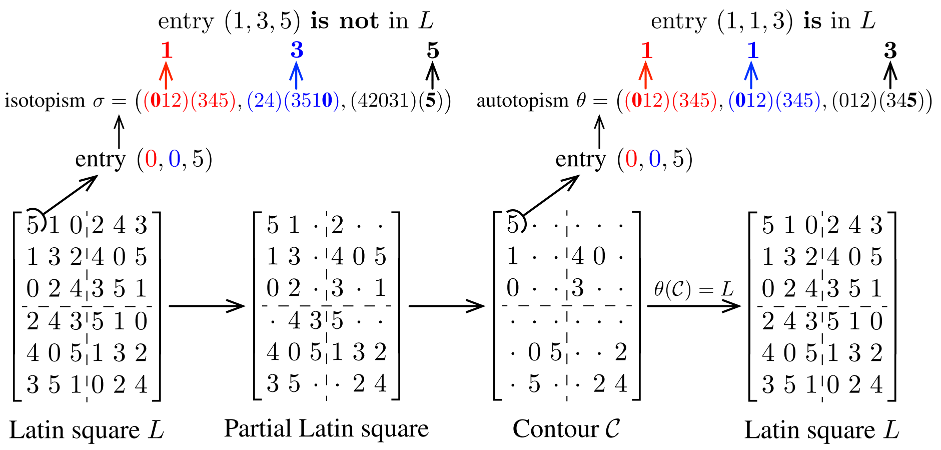
entry $(1,3,5)$ **is not** in L
entry $(1,1,3)$ **is** in L
isotopism $\sigma = ((012)(345), (24)(3510), (42031)(5))$
autotopism $\theta = ((012)(345), (012)(345), (012)(345))$
entry $(0,0,5)$
$\theta(\mathcal{C}) = L$
Latin square L
Partial Latin square
Contour $\mathcal{C}$
Latin square L

FIGURE 1. An example: the autotopism θ maps the entry (0, 0, 5) to the entry (1, 1, 3), which is in L; the isotopism σ (not an autotopism) maps the entry (0, 0, 5) to the entry (1, 3, 5), which is not in L. A contour $\mathcal{C}$ constructs a Latin square L under the respec autotopism θ.

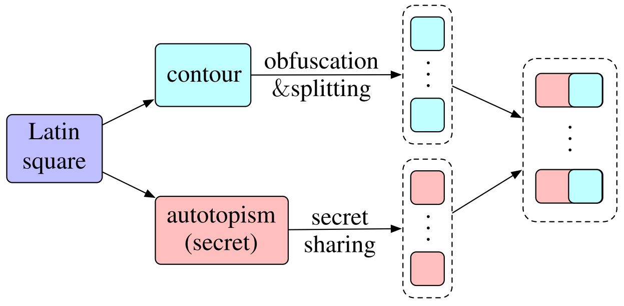

FIGURE 2. Fragmentation in LASS: a secret (an autotopism) is split into k secret shares, each of which is combined with a shard of contour to achieve verifiability.

B. A SUCCINCT EXAMPLE

Figure 1 gives an example based on a Latin square L. The entry $(0, 0, 5)$ denotes the element of Latin square L that is in row 0 and column 0 with the value of 5. The isotopism σ and autotopism θ both consist of three parts, each of which defines the permutation of row/column/value of an element. Under the autotopism θ, which is an automorphic permutation for Latin squares, the entry $(0, 0, 5)$ becomes entry $(1, 1, 3)$, which is an element that is also in Latin square L. However, under the isotopism σ, which is a permutation too, the same entry $(0, 0, 5)$ becomes entry $(1, 3, 5)$, which is not in Latin square L. The contour $\mathcal{C}$, which is a partial Latin square generated by selecting one entry from each orbit, constructs the Latin square L by three permutations under the autotopism θ, like $\theta(\mathcal{C}) = L$.

C. PROPERTIES OF LASS

In LASS, a secret θ is an autotopism. The k secret shares $\sigma_1, \ldots, \sigma_k$ are random isotopisms generated by the following steps: 1) randomly generate $k-1$ isotopisms $\sigma_1, \ldots, \sigma_{k-1}$; 2) compute $\sigma_k = {\sigma_{k-1}}^{-1}{\sigma_{k-2}}^{-1}\cdots{\sigma_1}^{-1}\theta$, so we have $\theta = \sigma_1\sigma_2\cdots\sigma_k$. Figure 2 plots the fragmentation process, in which a contour shard $\mathcal{C}_i$ is appended to each secret share. Reconstruction is the inverse process of fragmentation. Here we introduce some important properties of LASS.

1) VERIFIABILITY

With the corresponding autotopism θ, a contour $\mathcal{C}$ constructs a Latin square L; whereas, with a faulty autotopism θ, a contour $\mathcal{C}$ is unlikely to construct a Latin square L. Thus, during reconstruction we can verify that whether the autotopism θ' reconstructed from downloaded secret shares is correct or not. Verifiability is one of the primary benefits of LASS over both SSSS and Blakey's secret sharing scheme [29].

2) CONFIDENTIALITY

In Blakley's scheme, a leaked hyperplane permanently reveals partial information about the secret (i.e., a sub-hyperplane it belongs to). In LASS, a share σ_i is a random isotopism and does not reveal partial information about the secret θ. Thus, when a share σ_i of a secret θ is leaked, if we re-fragment θ again and replace all shares of θ, the leaked σ_i becomes permanently useless even if subsequent leaks occur. Confidentiality of LASS enables Gecko to resist targeted attacks (see Section V-A).

3) EFFICIENCY

Since reconstruction and re-fragmentation of secret and replacement of shares are very fast in LASS, we can efficiently and periodically perform replacement to provide a high-level of protection for the secret. Efficiency is another benefit of LASS compared to SSSS.

D. DISPERSION VIA LASS

Using an autotopism to generate an encryption key and using LASS to fragment the autotopism to give key slices, in Gecko:

1) a key slice does not reveal partial information of the encryption key;
2) we can verify whether the encryption key reconstructed from key slices is correct or not (whether $\theta' = \theta$ or not);

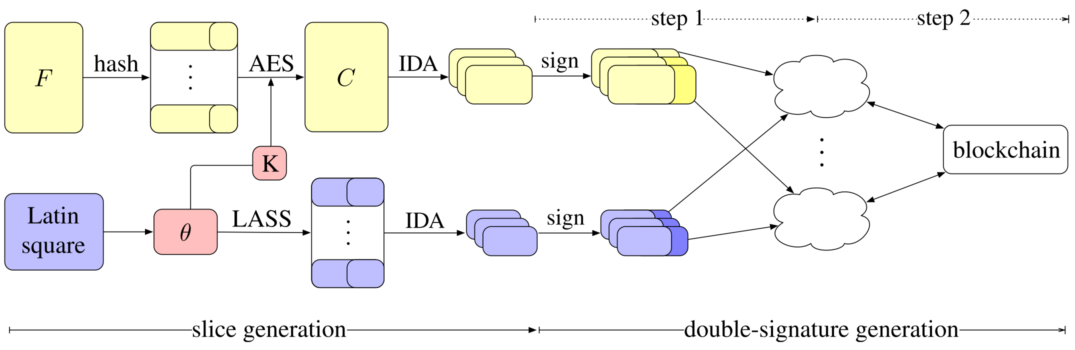

FIGURE 3. Uploading data under Gecko: we obtain *n* file slices and *n* key slices respectively, which are dispersed appended with signatures to *n* distinct CSPs. Double signatures are generated and uploaded to the blockchain by CSPs.

3) fast and thorough key renewal is possible, in which all slices of the key in danger are replaced.

However, if during key reconstruction we learn that $\theta' \neq \theta$ (implying an incorrect key slice exists), we cannot directly identify which key slice was incorrect due to tampering. Thus, we instead utilize blockchain-based NAP-check to identify incorrect slices exactly. Furthermore, LASS is a (k, k) scheme which does not offer fault tolerance. Thus, we combine a systematic Reed-Solomon code with LASS.

IV. GECKO DESIGN

In this section, we present the design details of Gecko, in which slices of the key in danger are discarded and renewed. In Gecko, uploading data includes slice generation and double-signature generation; downloading data involves reconstruction and possibly involves recovery. The NAP-check and key renewal are explained in detail. Gecko is utilized in a multi-cloud system with n CSPs in total, with a fault-tolerance of (k, n).

A. SLICE GENERATION

As Figure 3 plots, slice generation includes three stages as follows.

1) INITIALIZE THE ENCRYPTION KEY

We generate an autotopism θ and a contour $\mathcal{C}$ from a Latin square L, then convert the autotopism θ into a 256-bit key suitable for AES-256:

(a) convert θ into a string, e.g.

```
043,125*024,153*035,124;
```

(b) encode the above string as a positive integer, and use the SHA-256 hash function to generate a 256-bit encryption key $K :=$ SHA-256(pre-key).

2) GENERATE FILE SLICES

We split the plaintext F into k partial files $F_1, F_2, \ldots, F_k$ of approximately equal sizes. We compute a hash H_i for each F_i via SHA-256 and have

$$F' := [F_1|H_1|F_2|H_2|\cdots|F_k|H_k].$$

Here H_i can be used for integrity verification after recovering F' (if needed). We use AES-256 to encrypt file F' with the key K generated in Stage 1, yielding the ciphertext C. For fault tolerance we use an IDA based on systematic Reed-Solomon code RS(k, n) on C and give n file slices.

3) GENERATE KEY SLICES

According to LASS given in Figure 2, from an autotopism θ we generate k isotopisms $\sigma_1, \ldots, \sigma_k$, each of which is appended with a shard of contour $\mathcal{C}$ to get:

$$\hat{\theta} := [\sigma_1|\mathcal{C}_1|\sigma_2|\mathcal{C}_2|\cdots|\sigma_k|\mathcal{C}_k].$$

For fault tolerance, we use RS(k, n) on $\hat{\theta}$ and give n *key slices*, each of which does not reveal partial information about the encryption key K.

B. DOUBLE-SIGNATURE GENERATION

As Figure 3 plots, double-signature generation contains two steps and needs both the client and the CSP to participate. Double signatures are published on the blockchain for giving NAP-check. We first define two functions of a deterministic signature algorithm:

- Sign(sk, msg): to sign message msg using private key sk; it outputs signature sig.
- Veri(pk, sig, msg): to verify a signature sig using public key pk and the original message msg; it outputs 1 when verification fails and outputs 0 else.

As Algorithm 1 presents, the client signs each generated slice to get a signature. During dispersion, the generated n file slices along with signatures, and n key slices along with signatures, are dispersed independently to n distinct CSPs. Receiving a slice with the corresponding signature, a CSP stores the slice if verification succeeds, and signs the signatures to create double signatures, which are uploaded to the blockchain.

C. RECONSTRUCTION

To reconstruct a file, the client downloads k file slices and k key slices from the n CSPs, applying NAP-check to each slice. If any slice fails NAP-check, the client performs key renewal or file-slice recovery. Essentially this is a test-and-recover mechanism. When we obtain $2k$ correct slices, either at first or after the recovery process, then:

1) we decode the Reed-Solomon-encoded slices, to obtain $\hat{\theta}$ and the ciphertext C;
2) from $\hat{\theta}$, we compute the autotopism θ and the contour $\mathcal{C}$, and use θ and $\mathcal{C}$ to generate the corresponding Latin square, thereby verifying that the key is correct and untampered;
3) we regenerate the AES key K from θ, and decrypt the ciphertext C to give F' and thus the plaintext F; and
4) we may also verify that the decrypted file's contents are correct by using the hash H_i.

Algorithm 1 Double-Signature Generation

```
// step 1: client-side generation
$sig \leftarrow \mathrm{Sign}(sk^{\mathrm{CLIENT}}, slice)$
Disperse to CSPs
// step 2: CSP-side generation
if $\mathrm{Veri}(pk^{\mathrm{CLIENT}}, sig, slice)$ then
    SendToClient($errMsg$)
else
    Store($sig, slice$)
    $sig_d \leftarrow \mathrm{Sign}(sk^{\mathrm{CSP}}, sig)$
    $receipt \leftarrow \mathrm{SendToBlockchain}(sig_d)$
    SendToClient($receipt$)
```

D. KEY RENEWAL AND FILE-SLICE RECOVERY

During reconstruction, when a faulty key slice is detected through NAP-check, *key renewal* is performed, replacing all the slices of the key without changing the key itself, thereby avoiding re-encrypting files. The detailed steps are as follows:

1) discard a downloaded faulty key slice and download another key slice;
2) retain the downloaded key slice if correct, and proceed to download the next key slice;
3) continue step 1) and 2) until we obtain k correct key slices;
4) reconstruct the autotopism θ via LASS; and
5) re-disperse θ, including key-slice generation which is Stages 3 in Section IV-A, and double-signature generation (Section IV-B), thus to overwrite the previously stored key slices and corresponding signatures.

Thanks to NAP-check, which is used in step 1)–3), we avoid reconstructing θ with faulty key slices, which is a waste of time. In the experiment section we prove that although requesting double signatures entailing some additional time cost, key renewal in Gecko is still computationally fast since LASS is used. We envisage that it will be beneficial to perform key renewal periodically.

Having identified a corrupted file slice via NAP-check, we perform *file-slice recovery*:

1) download k correct file slices like the above step 1)–3);
2) compute the correct value of the corrupted file slice through Reed-Solomon coding; and
3) upload the corrected file slice along with its signature to the CSP, as in step 1 of Algorithem 1, replacing the corrupted file slice.

During this process, we do not decode the ciphertext C and do not replace the corresponding double signatures (as we do not renew the file slice).

E. BLOCKCHAIN-BASED INTEGRITY CHECK

In this part we present the workflow of NAP-check and explain how it meets non-repudiation, accountability, and public verifiability. NAP-check is useful in reconstruction; especially it accelerates key renewal and file-slice repair.

Algorithm 2 Client-Initiated NAP-Check

```
$sig' \leftarrow \mathrm{Sign}(sk^{\mathrm{CLIENT}}, slice')$
if $\mathrm{Veri}(pk^{\mathrm{CSP}}, sig_d, sig')$ then
    return fail
else
    return success
```

1) NAP-CHECK WORKFLOW

In client-initiated NAP-check, a cloud client downloads a slice *slice′* from the cloud and access the corresponding double signature sig_d from the blockchain, then perform Algorithm 2. If the output is "fail," the slice is possibly tampered or damaged so we perform either key renewal or file-slice recovery. Downloading a file slice may be time-consuming as the file slice size may be large. Thus, we prefer a client to perform NAP-check on file slices only when they access the file. Client-initiated NAP-check on key slices is efficient since key slices are small, which is suitable for periodical checks.

In CSP-initiated NAP-check, a CSP reads the to-be-checked slice *slice′* and its signature *sig′* stored on data servers, and accesses the corresponding double signature sig_d from the blockchain, then perform Algorithem 3. If the output is "fail," the CSP performs its own recovery mechanisms.

Algorithm 3 CSP-Initiated NAP-Check

```
if $\mathrm{Veri}(pk^{\mathrm{CSP}}, sig_d, sig')$ then
    return fail
else
    if $\mathrm{Veri}(pk^{\mathrm{CLIENT}}, sig', slice')$ then
        return fail
    NAP-check else
        return success
```

2) NAP-CHECK FEATURES

- **Non-repudiation.** A double signature is signed by both the client and the CSP, thus it is an integrity credential admitted by both parties, and is reliable as they are stored in a blockchain to guarantee they remain tamper-free. Utilizing such credentials, NAP-check results convince both parties. In other words, NAP-check achieves non-repudiation.

- **Accountability.** By generating and publishing a double signature, the CSP claims that the received slice is intact (there is no error during dispersion). Thus, if NAP-check fails, we can immediately conclude that the error was caused by the CSP's weak protection of data.
- **Public verifiability.** As mentioned, both of clients and CSPs can perform the NAP-check. The client can pinpoint incorrect slices when downloading them. CSPs can periodically check integrity to offer reliable storage service.

An additional feature granted due to the public verifiability of NAP-check is the ability to prove the authenticity of data to outside parties, such as insurance companies or regulatory organizations. In the scenario we grant access of any particular data to a third party, this third party can perform NAP-check and thereby verify the data's authenticity.

V. SECURITY ANALYSIS

We analyze the security of Gecko based on the threat model where:

1) there are n CSPs and the client utilizes a (k, n)-erasure code when generating key slices or file slices;
2) some CSPs may be temporarily lost due to accidents, whereas a proper (k, n) ensures the availability of data; and
3) data on servers of CSPs may be stolen, erased or corrupted (due to accidents or malicious operations).

A. THEFT

Under Gecko, a key slice does not reveal partial information about the key due to the use of LASS. A file slice reveals partial information of the ciphertext C, which we consider inconsequential. We conclude that with a Gecko slice (no matter whether a key slice or file slice), an attacker cannot infer any information. Thus, an attacker running targeted attack to steal a file has to steal k key slices and k file slices to access the plaintext. We assume that a targeted-attack attacker will at first attempt to reconstruct the encryption key, with which he or she then tries to decrypt file slice(s).

Let us focus on key theft. In a multi-cloud storage system, to steal a key an attacker needs to access at least k different CSPs using varying protocols and hardware, which is already an arduous task. Moreover, in Gecko we provide more powerful key protection: an attempt to access k key slices must access all k key slices before any key renewal is performed, and so is time-limited if key renewal is performed periodically. This is due to the fact that LASS shares are randomly generated and we can replace LASS shares $\sigma_1 \cdots \sigma_k$ without replacing the original autotopism θ, which means we can replace all slices of a leaked encryption key without replacing the key itself.

For comparison, in AONT-RS, each hybrid slice contains both a part of the file and a part of the encryption key, and any k slices are sufficient to obtain the plaintext, thereby requiring fewer successful attacks than Gecko. Furthermore, AONT-RS did not offer mechanism similar to key renewal, and we have to renew the AONT package (which is the combination of the ciphertext and the encryption key) to achieve the same protection as Gecko. AONT renewal requires a replacement of the encryption key and a re-encryption and upload of the file, which is much less flexible than Gecko.

In SSMS, since encryption keys are independently dispersed, key renewal is possible. However, key renewal based on SSSS is much less efficient than key renewal in Gecko; a key renewal that is slow gives attackers more chances to steal key slices.

In terms of files, Gecko offers the same level of protection as AONT-RS, in terms of secrecy and recovery: it uses AES to encrypt the file for confidentiality and uses Reed-Solomon codes to fix the incorrect file slices. As SSMS uses a 16-bit encryption key, file protection in SSMS is weaker than in both Gecko and AONT-RS.

B. ERASURE

In Gecko, data is only irretrievably lost when erasures (or corruption, tampering, etc.) occur to the relevant data on at least $n - k + 1$ CSPs simultaneously, which is unlikely given that CSPs themselves are independent and tolerate erasures through, e.g., data replication (i.e., storing multiple copies of data) or erasure coding, and run processes like data scrubbing for error detection. Thus, data loss through random erasures is a negligible concern.

Supposing deliberate or systematic erasure occurs (e.g., through an attack or collusion, or CSPs failing to renew or repair correctly), if $n - k$ or fewer CSPs selectively erase the client's data, the client is still capable of recovering the data via the other CSPs. Moreover, the inaccessible data enable the client to identify problematic CSPs and to take remedial steps.

If $n - k + 1$ or more CSPs selectively erase the client's data, then the client's data are irretrievably lost, so n and k should be chosen to suit the level of risk. (The client should also choose more trustworthy clouds.)

VI. LATENCY ANALYSIS

A. THEORETICAL LATENCY ANALYSIS

We derive mathematical formulas for the recovery latency of Gecko, and compare them against AONT-RS. Table 2 tabulates the variables we include. We make some simplifying assumptions: (a) the per-MB download and upload latency are equal, (b) latency scales linearly with the transfer volume, (c) the cloud-blockchain latency is the same for each cloud, and is the same as the client-blockchain latency, (d) we ignore the contribution of generating signatures, and (e) actions are performed sequentially. However, we allow the possibility that different CSPs have different client-cloud latencies.

TABLE 2. Latency variables. All sizes are in MB, and latencies are in seconds per MB.

variable	definition
$\lvert F\rvert$, $\lvert C\rvert$, $\lvert\hat{C}\rvert$	the plaintext file size, the ciphertext file size of Gecko, and of AONT-RS, respectively
$\lvert K\rvert$	the encryption-key size
$\mathbf{f}$, $\mathbf{k}$, $\mathbf{s}$	Gecko: the size of file slice, key slice, and signature (or double signature), respectively
$\hat{\mathbf{f}}$	AONT-RS: the slice size
e, d	Gecko: the slice generation, and reconstruction latency, respectively
$\hat{e}$, $\hat{d}$	AONT-RS: the slice generation, and reconstruction latency, respectively
c_i, B	the transfer latency of the i-th cloud and the blockchain, respectively
n	the number of CSPs; equal to the number of key slices or file slices
k	the Reed-Solomon parameter in $\mathrm{RS}(k, n)$

Here we present a running example to compare theoretical recovery latency of Gecko and AONT-RS. There are 5 CSPs $C_1, \ldots, C_5$ and we use the Reed-Solomon code RS(3, 5) for both Gecko and AONT-RS. We download 3 slices s_1, s_2, s_3 from C_1, C_2, and C_3 among which C_1 is dishonest.

In AONT-RS, only after reconstructing $|F|$ can we detect that there is incorrect downloaded slice(s) and we cannot know which one is incorrect. To continue reconstructing, we need to download other slices and try other combinations (such as s_1, s_3, and s_5). In worst case, we need to try $\binom{5}{3} = 10$ times, which gives the latency:

$$\hat{\ell}_{\text{step1}} = 10 \cdot \hat{d}|\hat{C}| + (c_1 + \cdots + c_5)\hat{\mathbf{f}}.$$

To renew all slices, we need to re-encrypt the file using a new key, then upload the five new slices to the CSPs, taking total time:

$$\hat{\ell}_{\text{step2}} = \hat{e}|F| + (c_1 + \cdots + c_5)\hat{\mathbf{f}}.$$

Thus, the renewal latency of AONT-RS is:

$$\hat{\ell}_{\text{renewal}} = 10\hat{d}|\hat{C}| + 2\sum_{i=1}^{5} c_i\hat{\mathbf{f}} + \hat{e}|F|.$$

In Gecko, we detect that s_1 is incorrect by NAP-check just after downloading. We discard s_1 and perform recovery. If s_1 is a file slice, we download three other slices, perform NAP-check and reconstruct F, which gives the latency:

$$\ell_{\text{file-step1}} = d|C| + (c_1 + \cdots + c_4)\mathbf{f} + 4 \cdot B\mathbf{s}.$$

We then fix the incorrect slice s_1 along with the corresponding signature on C_1, which takes time:

$$\ell_{\text{file-step2}} = e|F| + c_1(\mathbf{f} + \mathbf{s}).$$

Thus, the file-slice-recovery latency in Gecko is:

$$\ell_{\text{file-slice-recovery}} = d|C| + (2c_1 + \sum_{i=2}^{4} c_i)\mathbf{f} + e|F| + 4B\mathbf{s} + c_1\mathbf{s}.$$

If s_1 is a key slice, we download three other slices, perform NAP-check and reconstruct θ, which takes time:

$$\ell_{\text{key-step1}} = (c_1 + \cdots + c_4)\mathbf{k} + 4 \cdot B\mathbf{s}.$$

Here we ignore the latency of decoding the key, as key slices are small-sized. We then replace $s_1, \ldots, s_5$ and the five corresponding double signatures on the blockchain, which takes time:

$$\ell_{\text{key-step2}} = (c_1 + \cdots + c_5)(\mathbf{k} + \mathbf{s}) + 5 \cdot B\mathbf{s}.$$

Here we ignore the latency of re-encoding the key. Thus, the key-recovery latency in Gecko are:

$$\ell_{\text{key-renewal}} = \Big(2\sum_{i=1}^{4} c_i + c_5\Big)\mathbf{k} + 9B\mathbf{s} + \sum_{i=1}^{5} c_i\mathbf{s}.$$

Assuming $\mathbf{f}$ is large, the $\ell_{\text{key-renewal}}$ (which does not have an $\mathbf{f}$ term) is far smaller than $\hat{\ell}_{\text{renewal}}$ which is dominated by the term $2\sum_{i=1}^{5} c_i\hat{\mathbf{f}}$. Moreover, the $\ell_{\text{file-slice-recovery}}$ is dominated by $(2c_1 + \sum_{i=2}^{4} c_i)\mathbf{f}$, which we expect is smaller than $2(c_1 + \cdots + c_5)\mathbf{f} \simeq 2(c_1 + \cdots + c_5)\hat{\mathbf{f}}$. This indicates how key renewal and file-slice recovery is faster than hybrid-slice renewal in AONT-RS, and we expect this behavior to become more significant as $\mathbf{f}$ grows (or equivalently as $|F|$ grows). Compared to SSMS, key-slice generation for SSMS is $O(kn \cdot |K|)$ and for Gecko is $O\big((n-k)|K| + k\big)$, whereas double-signature generation in Gecko may add a small overhead.

We observe that the cloud latencies c_i along with the blockchain latency B play a significant role in how efficient Gecko is in practice. These latencies will be an essential consideration in the conclusions in this section and the subsequent experimental simulations in Section VI-B: one slow cloud may be a bottleneck for the whole system, and may radically affect these latencies.

B. EXPERIMENTAL LATENCY ANALYSIS

Motivated by the theoretical latency analysis, we experimentally evaluate: (a) that key renewal in Gecko is significantly more efficient than hybrid-slice recovery in AONT-RS, and (b) that the upload and download latency of Gecko is only slightly more than that of AONT-RS.

1) EXPERIMENTAL SETUP

We use one computer as the client and another as the cloud server, with FTP servers used to simulate multiple CSPs. The cloud client has an 8-core Intel i7-6700 processor (2.60 GHz) and 8 GB RAM running 64-bit Windows 10, and the cloud server has an AMD Phenom processor (2.60 GHz) and 4GB RAM running 32-bit Windows 7. The source code is implemented in C and C# and compiled by Visual Studio 2012. The network between the client and the server consists of a gigabit Ethernet switch, and we use a TL-WR847N router to limit the bandwidth to 100MB/s between the components to simulate a WAN connection. Various listener ports are set to accept the

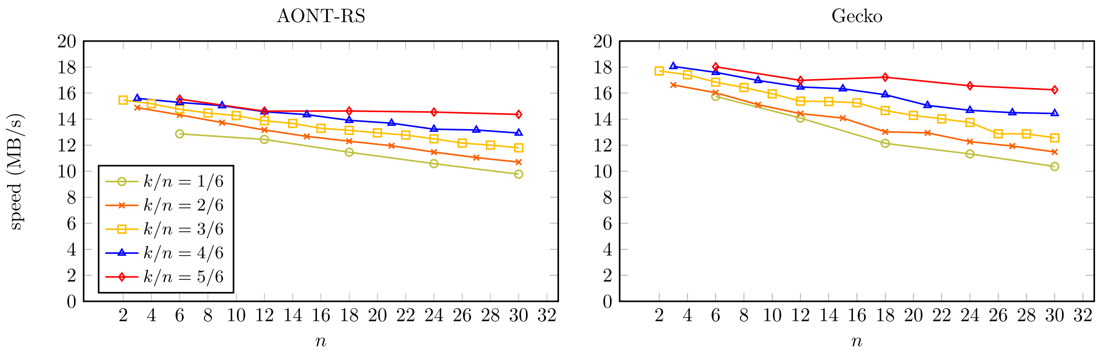

FIGURE 4. Slice-generation speed for AONT-RS and Gecko.

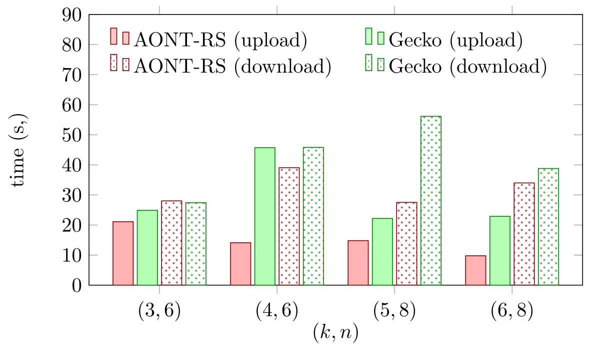

FIGURE 5. Upload and download time for AONT-RS and Gecko, for an 8MB unencrypted file (averaged over three runs).

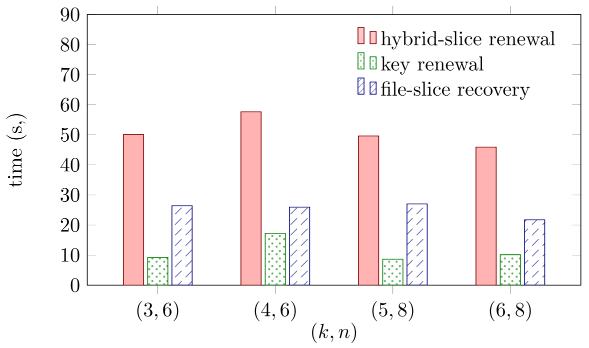

FIGURE 6. The hybrid-slice renewal time for AONT-RS, and key renewal and file-slice recovery time for Gecko (averaged over three runs).

data which are sent to different CSPs, where the server reads the data and verifies the signature from the client.

Encoding speeds when using Latin squares of differing orders makes a negligible difference, so we only present results for order-10 Latin squares. Where relevant, the file size is 8 MB. The IDA uses systematic Reed-Solomon RS(k, n) coding and we vary k and n. For Reed-Solomon coding, we use Luigi Rizzo's open source library [43] over GF(2^8). We use an elliptic curve digital signature algorithm (ECDSA) to generate signatures.

Gecko uses a public blockchain: public blockchains feature higher reliability than private blockchains. We use the API provided by Tierion [44] to join the Bitcoin blockchain; we post a double signature into Tierion, and it returns a record ID. When performing NAP-check, Tierion returns the double signature and its location in the Bitcoin blockchain to a requester who holds the corresponding record ID.

2) UPLOAD AND DOWNLOAD LATENCY

Figure 4 plots the speed of fragmenting a file into: 1) hybrid slices for AONT-RS, and 2) file slices and key slices for Gecko. Under various parameters (k, n), Gecko features a slightly higher speed than AONT-RS. In [29] Stones made a simple implementation of the LASS (which included everything involved in the secret sharing scheme), and its runtime was in the ballpark of milliseconds. Thus, we expect that most of the time spent on Gecko is on file-slice generation.

Figure 5 plots the upload and download time of AONT-RS and Gecko, in the simulated multi-cloud storage system. There is indeed additional overhead for Gecko in uploading and downloading, due to the generation of double signatures and transfer latency between CSPs and the blockchain. The additional overhead is the trade-off for performing NAP-check and efficient key renewal and file-slice recovery. The fluctuation under various parameters (k, n) may be due to: 1) the fluctuations in the blockchain response time, 2) the fluctuations in the cloud processing time, and 3) network speed.

3) RENEWAL LATENCY

Figure 6 plots the renewal time for AONT-RS (which requires that both the key and file be renewed), and the key renewal time and the file-slice recovery time for Gecko. We observe that the Gecko key renewal time is 3 to 5 times faster than renewal in AONT-RS, and the Gecko file-slice recovery time is about 2 times faster than renewal in AONT-RS. This observation is consistent with the theoretical analysis in Section VI-A. Importantly, AONT-RS does not have a mechanism for detecting faulty files, unlike Gecko (via NAP-check), so renewal in AONT-RS is mostly non-effective.

We further evaluate the time cost for NAP-check during key renewal. We split the key renewal time into two components: (a) the time spent on LASS: reconstructing θ by k correct key slices and then generating n new key slices; and (b) the time spent involving NAP-check: the NAP-check time for each downloaded key slice, and double-signature

TABLE 3. The key renewal time split into two parts: (a) the time spent on LASS and (b) the time spent involving NAP-check.

(k,n)	part (a)		part (b)	
$(3,6)$	4.47	(48%)	4.76	(52%)
$(4,6)$	2.70	(16%)	14.56	(84%)
$(5,8)$	1.71	(20%)	6.93	(80%)
$(6,8)$	2.38	(24%)	7.74	(76%)

generation for each new generated key slice. Table 3 tabulates the results.

We observe that NAP-check adds a noticeable overhead, especially when we download k correct key slices. However, as noted in Section VI-A, with NAP-check we avoid computation during the reconstruction of θ when incorrect slices exist. In summary, NAP-check accelerates key renewal and file-slice recovery.

VII. CONCLUDING REMARKS

In this paper, we proposed Gecko, a multi-cloud dispersal scheme that utilizes a Latin-square autotopism secret sharing scheme and blockchain technology. It improves on the traditional AONT-RS by having additional functionality: (a) distributed integrity verification, which is performed after a slice is downloaded, and (b) key renewal and file-slice repair, which are both faster than complete renewal. Experimental results indicate that Gecko performs key renewal 3 to 5 times faster than AONT-RS performs complete renewal, and the upload and download overhead is minor. It is also important to note that cloud latency and the blockchain latency plays a determining role in the latency of both Gecko and AONT-RS.

One challenge of this work is deciding how to store proof-of-data in the blockchain. We devise a double signature to achieve non-repudiation, accountability, and public verifiability. We envisage Gecko being used as a kind of storage ''middleman'' (such as a logical layer in a trusted server): users send their data (either encrypted or unencrypted) to the Gecko operator, who maintains the data on multiple clouds on behalf of the users. This benefits the users, who may not be particularly familiar with cloud storage, or may not wish to implement multi-cloud storage themselves.

We also suggest some other directions for future work: (a) the Latin-square-autotopism secret sharing scheme we use has no fault tolerance. Future work could focus on adapting this scheme to tolerate faults. One possibility is using the multi-level adaptation of the Latin-square secret sharing scheme described in [29]; and (b) we could explore constructing a blockchain specially designed for storing proof-of-data, which may accelerate the transmission speeds involved in using the blockchain.

REFERENCES

[1] H. Takabi, J. B. D. Joshi, and G.-J. Ahn, ''Security and privacy challenges in cloud computing environments,'' *IEEE Security Privacy*, vol. 8, no. 6, pp. 24–31, Nov./Dec. 2010.

[2] D. K. Bowers, A. Juels, and A. Oprea, ''HAIL: A high-availability and integrity layer for cloud storage,'' in *Proc. CCS*, Nov. 2008, pp. 187–198.

[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, ''Provable data possession at untrusted stores,'' in *Proc. CCS*, Oct. 2007, pp. 598–609.

[4] A. W. Services. (2019). *AWS Partner Story: CareCloud*. Accessed: Jan. 9. 2019. [Online]. Available: https://aws.amazon.com/cn/partners/success/carecloud/

[5] H. Krawczyk, ''Secret sharing made short,'' in *Proc. Annu. Int. Cryptol. Conf.* Berlin, Germany: Springer, 1993, pp. 136–146.

[6] J. K. Resch and J. S. Plank, ''AONT-RS: Blending security and performance in dispersed storage systems,'' in *Proc. FAST*, 2011, pp. 1–12.

[7] M. Li, C. Qin, P. P. C. Lee, and J. Li, ''Convergent dispersal: Toward storage-efficient security in a cloud-of-clouds,'' in *Proc. HotStorage*, Jul. 2014, pp. 1–5.

[8] C. Cachin, I. Keidar, and A. Shraer, ''Trusting the cloud,'' *ACM SIGACT News*, vol. 40, no. 2, pp. 81–86, Jun. 2009.

[9] J. Novet. (2017). *Microsoft Confirms Azure Storage Issues Around the World (Updated)*. Accessed: Feb. 15, 2018. [Online]. Available: https://venturebeat.com/2017/03/15/microsoft-confirms-azure-storage-issues-around-the-world/

[10] T. Robinson. (2018). *Open AWS S3 Bucket Exposes Private info on Thousands of Fedex Customers*. Accessed: Nov. 6, 2018. [Online]. Available: https://www.scmagazine.com/

[11] L. Razavi. (2014). *The IC Cloud Leak: Weak Security isn't Only a Problem for Apple's Backup Service*. Accessed: Sep. 2, 2016. [Online]. Available: https://www.newstatesman.com/sci-tech/

[12] M. A. AlZain, E. Pardede, B. Soh, and J. A. Thom, ''Cloud computing security: From single to multi-clouds,'' in *Proc. HICSS*, Jan. 2012, pp. 5490–5499.

[13] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, ''RACS: A case for cloud storage diversity,'' in *Proc. SoCC*, Jun. 2010, pp. 229–240.

[14] W. M. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti, ''POT-SHARDS: A secure, recoverable, long-term archival storage system,'' *ACM Trans. Storage*, vol. 5, no. 2, pp. 1–35, Jun. 2009.

[15] A. Shamir, ''How to share a secret,'' *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979.

[16] M. O. Rabin, ''Efficient dispersal of information for security, load balancing, and fault tolerance,'' *J. ACM*, vol. 36, no. 2, pp. 335–348, Apr. 1989.

[17] R. L. Rivest, ''All-or-nothing encryption and the package transform,'' in *Proc. 4th Int. Workshop Fast Softw. Encryption*, 1997, pp. 210–218.

[18] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. Amsterdam, The Netherlands: Elsevier, 1977.

[19] M. Li, C. Qin, and P. P. C. Lee, ''CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal,'' in *Proc. USENIX Annu. Tech. Conf.*, Jul. 2015, pp. 111–124.

[20] L. Shen, S. Feng, J. Sun, Z. Li, M. Su, G. Wang, and X. Liu, ''CloudS: A multi-cloud storage system with multi-level security,'' *IEICE Trans. Inf. Syst.*, vol. 99, no. 8, pp. 2036–2043, 2016.

[21] K. Kapusta, G. Memmi, and H. Noura, ''POSTER: A keyless efficient algorithm for data protection by means of fragmentation,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1745–1747.

[22] G. R. Blakley, ''Safeguarding cryptographic keys,'' in *Proc. NCC*, vol. 48, Jun. 1979, pp. 313–317.

[23] J. Cooper, D. Donovan, and J. Seberry, ''Secret sharing schemes arising from latin squares,'' *Bull. Inst. Combinatorics Appl.*, vol. 12, pp. 33–43, 1994.

[24] J. Cooper, D. Donovan, and J. Seberry, ''Latin squares and critical sets of minimal size,'' *Australas. J. Combin.*, vol. 4, pp. 113–120, 1991.

[25] D. M. Donovan, J. G. Lefevre, T. A. McCourt, N. J. Cavenagh, and A. Khodkar, ''Identifying flaws in the security of critical sets in latin squares via triangulations,'' *Australas. J. Combinatorics*, vol. 52, pp. 243–268, 2012.

[26] M. Tompa and H. Woll, ''How to share a secret with cheaters,'' *J. Cryptol.*, vol. 1, no. 3, pp. 133–138, Oct. 1989.

[27] J. Mike Grannell, S. Terry Griggs, and A. P. Street, ''A flaw in the use of minimal defining sets for secret sharing schemes,'' *Des., Codes Cryptogr.*, vol. 40, no. 2, pp. 225–236, Aug. 2006.

[28] R. M. F. Ganfornina, ''Latin squares associated to principal autotopisms of long cycles. Application in cryptography,'' *Transgressive Comput.*, pp. 213–230, 2006.

[29] R. J. Stones, M. Su, X. Liu, G. Wang, and S. Lin, ''A latin square autotopism secret sharing scheme,'' *Des., Codes Cryptogr.*, vol. 80, no. 3, pp. 635–650, Sep. 2016.

[30] J. Browning, D. S. Stones, and I. M. Wanless, "Bounds on the number of autotopisms and subsquares of a latin square," *Combinatorica*, vol. 33, no. 1, pp. 11–22, Feb. 2013.
[31] R. M. Falcón, "Cycle structures of autotopisms of the Latin squares of order up to 11," *Ars Combinatoria*, vol. 103, pp. 239–256, Oct. 2012.
[32] B. D. McKay, A. Meynert, and W. Myrvold, "Small latin squares, quasigroups, and loops," *J. Combinat. Des.*, vol. 15, no. 2, pp. 98–119, Mar. 2007.
[33] D. S. Stones and P. Vojt chovský, and I. M. Wanless, "Cycle structure of autotopisms of quasigroups and Latin squares," *J. Combinat. Des.*, vol. 20, no. 5, pp. 227–263, May 2012.
[34] C. J. Colbourn, "The complexity of completing partial latin squares," *Discrete Appl. Math.*, vol. 8, no. 1, pp. 25–30, Apr. 1984.
[35] N. Kaaniche and M. Laurent, "A blockchain-based data usage auditing architecture with enhanced privacy and availability," in *Proc. IEEE 16th Int. Symp. Netw. Comput. Appl. (NCA)*, Oct./Nov. 2017, pp. 1–5.
[36] H. Shafagh, L. Burkhalter, A. Hithnawi, and S. Duquennoy, "Towards blockchain-based auditable storage and sharing of IoT data," in *Proc. Cloud Comput. Secur. Workshop*, Nov. 2017, pp. 45–50.
[37] S. Matsumoto and R. M. Reischuk, "IKP: Turning a PKI around with decentralized automated incentives," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 410–426.
[38] B. Liu, X. L. Yu, S. Chen, X. Xu, and L. Zhu, "Blockchain based data integrity service framework for IoT data," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jun. 2017, pp. 468–475.
[39] A. Juels and B. S. Kaliski, Jr., "PORs: Proofs of retrievability for large files," in *Proc. CCS*, Oct. 2007, pp. 584–597.
[40] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proc. SecureComm*, Sep. 2008, p. 9.
[41] P. Labs. (2014). *Filecoin: A Decentralized Storage Network*. Accessed: Aug. 14, 2017. [Online]. Available: https://filecoin.io/filecoin.pdf
[42] Storj Labs, Inc. (2018). *Storj: A Decentralized Cloud Storage Network Framework*. Accessed: Oct. 30, 2018. [Online]. Available: https://github.com/storj/whitepaper
[43] L. Rizzo. (2006). *Reed-Solomon FEC Codes*. Accessed: Nov. 5, 2018. [Online]. Available: http://planete-bcast.inrialpes.fr/rubrique1edb.html?id_rubrique=10
[44] Tierion. (2018). *Hash API Documentation of Tierion*. Accessed: Nov. 5, 2018. [Online]. Available: https://v1.tierion.com/docs/hashapi

MENG YAN was born in Tianjin, China, in 1992. She received the B.S. degree in applied physics from the Tianjin University, Tianjin, China, in 2014.

She is a doctoral student of Nankai-Baidu Joint Laboratory, Nankai University, Tianjin, China. Her research interests include security and privacy of cloud-storage, blockchain technology, and data integrity.

JIAQI FENG was born in Anhui, China, in 1995. She received the B.S. degree in information and computing science from Anhui University, Hefei, China, in 2017. She is a graduate student of Nankai-Baidu Joint Laboratory, Nankai University, Tianjin, China. Her research interests include security and privacy of storage, and image processing.

TRENT G. MARBACH received the B.Sc., B.Sc. (Honours), and Ph.D. degrees in mathematics from The University of Queensland, Australia, in 2009, 2010, and 2016, respectively. He is currently a post-doctoral fellow with the College of Computer, Nankai University, Tianjin, China. His research interests include combinatorics and security.

REBECCA J. STONES received the Ph.D. degree from Monash University in pure mathematics, in 2010. Stones now has diverse research interests, including combinatorics and graph theory, codes, search engines and data storage, phylogenetics, and quantitative psychology.

GANG WANG received the B.Sc., M.Sc., and Ph.D. degrees in computer science from Nankai University, Tianjin, China, in 1996, 1999, and 2002, respectively.

He is currently a professor with the College of Computer, Nankai University, Tianjin, China. His research interests include storage systems and parallel computing.

XIAOGUANG LIU received the B.Sc., M.Sc., and Ph.D. degrees in computer science from Nankai University, Tianjin, China, in 1996, 1999, and 2002, respectively.

He is currently a professor at the College of Computer, Nankai University, Tianjin, China. His research interests include parallel computing, storage systems, and search engines.