SPARK

Secure Pseudorandom Key-based Encryption for Deduplicated Storage

Dave, Jay; Faruki, Parvez; Laxmi, Vijay; Zemmari, Akka; Gaur, Manoj; Conti, Mauro

DOI
10.1016/j.comcom.2020.02.037

Publication date
2020

Document Version
Final published version

Published in
Computer Communications

Computer Communications 154 (2020) 148–159

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

SPARK: Secure Pseudorandom Key-based Encryption for Deduplicated Storage

Jay Dave [a,*], Parvez Faruki [b], Vijay Laxmi [a], Akka Zemmari [c], Manoj Gaur [d], Mauro Conti [e,f]

[a] Malaviya National Institute of Technology Jaipur, India
[b] Government MCA College, Ahmedabad, India
[c] University of Bordeaux, Talence, France
[d] Indian Institute of Technology Jammu, India
[e] University of Padua, Italy
[f] Delft University of Technology, Netherlands

ARTICLE INFO

Keywords:
Deduplication
Encryption
Dictionary attacks
Tag inconsistency anomaly

ABSTRACT

Deduplication is a widely used technology to reduce the storage and communication cost for cloud storage services. For any cloud infrastructure, data confidentiality is one of the primary concerns. Data confidentiality can be achieved via user-side encryption. However, conventional encryption mechanism is at odds with deduplication. Developing a user-side encryption mechanism with deduplication is a vital research topic. Existing state-of-the-art solutions in security of deduplication are vulnerable to dictionary attacks and tag inconsistency anomaly.

In this paper, we present SPARK, a novel approach for secure pseudorandom key-based encryption for deduplicated storage. SPARK achieves semantic security along with deduplication. Security analysis proves that SPARK is secure against dictionary attacks and tag inconsistency anomaly. As a proof of concept, we implement SPARK in realistic environment and demonstrate its efficiency and effectiveness.

1. Introduction

Cloud Storage has become an essential part of various network applications due to its location independent, low-cost, and scalable online storage services. Cisco Global Cloud Index foresights that the size of digital data on cloud storage will increase up to 19.5 Zettabytes in 2021 [1]. Such explosive growth of data on cloud storage promotes the demand for an approach that reduces the storage cost. Deduplication is an approach to optimize the utilization of storage resources. In the following, we discuss the process of deduplication approach.

Deduplication: In this approach, when a user requests data upload, storage server checks whether this data exists on its storage. Storage server stores the data only if it is not present. In this way, deduplication avoids storing multiple copies of the same data. The advantage of deduplication is measured by space reduction ratio [2]. Space reduction ratio is $(\frac{size\ of\ input\ data}{size\ of\ data\ to\ be\ stored})$, the size of input data divided by size of data to be stored. Hence, deduplication advantage percentage is $\{1 - (\frac{1}{space\ reduction\ ratio})\} \times 100$.

Deduplication can be categorized on the basis of (1) Granularity, (2) Intra-Inter user, (3) Locality, and (4) Architecture as discussed in [3]. Based on granularity, deduplication is further categorized into File level and Block level deduplication. (i) *File level*: Storage is scanned filewise to detect the existence of identical file. (ii) *Block level*: File is divided into blocks. Storage is scanned block by block. In block level deduplication, block size can be either fixed size or variable size. In terms of user levels, deduplication is categorized into Intra user-Inter user deduplication. (iii) *Intra User*: Deduplication is applied in context of the user's individual data only. (iv) *Inter User*: Deduplication is applicable to the data of all users of the storage server. Based on locality, deduplication is further classified into Client side and Server side deduplication. (v) *Client side*: User first sends some tag of data (i.e., data's hash value) to the storage server for detecting redundancy. User sends the data only if it is not present at storage. (vi) *Server side*: User is not aware of whether deduplication will take place to her data. She just outsources the data, and the storage server further executes deduplication on received data. Communication overhead is higher in server side deduplication as compared to client side deduplication because the user needs to send the data in server side deduplication even if it is present on storage. And finally, based on architecture, deduplication can be categorized into (vii) *Single cloud deduplication architecture* in which a single cloud service provider is considered as many commercial cloud service provider do, and (viii) *multi-cloud deduplication architecture* in which multiple cloud service providers are considered, and users' data is split and dispersed across these service

* Corresponding author.
E-mail address: jaydaveadms@gmail.com (J. Dave).

https://doi.org/10.1016/j.comcom.2020.02.037
Received 3 June 2019; Received in revised form 29 December 2019; Accepted 11 February 2020
Available online 13 February 2020

providers. In this paper, we consider file level, client side, intra-inter user, single cloud deduplication for our proposal.

From a security perspective, shared access to users' data in deduplication system unfolds data confidentiality vulnerabilities. For preserving data confidentiality, user encrypts the data before outsourcing it on cloud storage. However, conventional encryption is at odds with deduplication. Deduplication exploits data similarity to detect redundancy. On the other hand, conventional encryption achieves indistinguishability between two ciphertext through randomization. In other words, encryption of identical data by different users generates different ciphertext. For instance, user A encrypts the data D with secret key k_A and uploads ciphertext C_A to the storage. Subsequent uploader, user B encrypts same data D with secret key k_B and requests to store C_B.

This unfolds two queries: (i) how will the storage server be able to identify that C_A and C_B are ciphertext of same data D, (ii) even if it detects this successfully, how can it allow both users to access the stored ciphertext (C_A or C_B). One simple solution is: encrypt the data with public key of storage server. In this way, storage server will be able to decrypt ciphertext with its private key and detect redundancy. However, this solution breaches data confidentiality by revealing users' data to storage server.

As a solution to aforementioned problem, Douceur et al. proposed Convergent Encryption (CE) that provides confidentiality along with deduplication [4]. In CE, users encrypt data with the convergent key, i.e., hash value of data. Users upload ciphertext to storage server and keep key with them. Since CE is deterministic, it outputs same ciphertext for identical data. Thus, storage server can perform deduplication on ciphertext. Variants of Convergent Encryption [5–11] were proposed in different state-of-the-art for secure deduplicated storage. However, CE is vulnerable to dictionary attacks on predictable data [12], since it is a deterministic encryption approach. In the following, we briefly discuss the dictionary attacks.

Dictionary attack: An adversary who has infinite access to users' outsourced data (e.g., Storage server) can carry out offline dictionary attack. This adversary may have certain knowledge about the data. And, she tries to predict the unknown portion within finite attempts. In this way, adversary constructs the possible versions of data, encrypts them with convergent key and identifies the encrypted version which is matching with stored ciphertext. On the other hand, a malicious user of storage server executes online dictionary attack. She may have knowledge about certain portion of data. And, she guesses the unknown portion within finite attempts. In this way, adversary constructs the predicted versions, encrypts them with convergent keys, uploads ciphertext versions to storage server, and identifies the version which causes deduplication.

In [10], authors proposed randomized CE as a solution to this problem. In this approach, data is encrypted with a random key for securing it against dictionary attacks. However, this random key is encrypted with the convergent key and outsourced with ciphertext of data. Hence, encrypted random key is still vulnerable to offline dictionary attack. On the other hand, an adversary can perform online dictionary attack by computing and uploading deduplication tag (e.g., Hash(Hash(data))) of predicted versions. Some solutions for deduplication security [9,11–15] recommend employing the additional independent key server(s) which is a cost-inefficient assumption. It conflicts with the idea of outsourcing data to cloud for cost reduction. In the following, we briefly discuss the tag inconsistency anomaly.

Tag inconsistency anomaly: In randomized encryption, an adversary can disrupt the deduplication process by uploading false combination of deduplication tag and ciphertext. Since deduplication tag is generated from plaintext, storage server is not able to detect such inconsistency between deduplication tag and ciphertext. In this case, when a genuine subsequent uploader requests for data upload with same deduplication tag, storage server permits access to false data and sends successful upload confirmation. After receiving the confirmation, most of users deletes the data from their local site. As a consequence, a genuine user gets false content when she downloads her uploaded data.

In this paper, we propose a novel approach "SPARK", Secure Pseudorandom Key-based Encryption for Deduplicated Storage which provides security against online-offline dictionary attacks and also tag inconsistency anomaly. SPARK computes a random encryption key using a secure key generation mechanism which is combination of a random *seed* value, cryptographic hash function and secure pseudorandom generator. Moreover, SPARK is equipped with semantically secure encryption mechanism. Since SPARK leverages a random encryption key and semantically secure encryption mechanism, it is computationally hard to perform successful offline dictionary attack on outsourced ciphertext. On the other hand, adversary tries to launch online-offline dictionary attacks by guessing deduplication tags of data. Since SPARK uses a trimmed hash value as deduplication tag, multiple files will be linked with same deduplication tag. Hence, indistinguishability of a file on the basis of the tag ensures that the adversary is not able to perform dictionary attacks using deduplication tags.

In addition, SPARK prevents adversary who is having knowledge about *seed* value, from learning the key for unpredictable files. SPARK key generation mechanism requires entire file along with *seed* value to generate the key. Thus, adversary cannot learn the key without knowledge of entire file. Moreover, SPARK adopts rate limiting strategy per user to slow down the flooding attack. For security against tag inconsistency anomaly, storage server asks hash value of ciphertext before giving access to existing data. If received hash value differs from hash value of existing data, then storage server asks the user to upload data. In this way, tag inconsistency anomaly will not disturb the deduplication process of SPARK.

Our contributions. This paper makes the following contributions.

- We propose a novel approach of secure pseudorandom key-based encryption mechanism to achieve semantic security along with deduplication.
- Security analysis theoretically proves that our approach is secure against online-offline dictionary attacks and tag inconsistency anomaly.
- We implement SPARK considering four basic storage services: Upload, Download, Delete, and Update file. Performance evaluation shows that SPARK incurs minimal overhead in realistic environment.

2. Preliminaries

In the following, we briefly discuss the cryptographic notions, i.e., Symmetric Encryption and Pseudorandom Generator, which are used in SPARK.

Symmetric encryption. Symmetric Encryption leverages same key K for both encryption and decryption procedures. The symmetric encryption mechanism comprises following primitive algorithms:

- $KeyGen(seed) \rightarrow K$: It generates a symmetric key K based on seed value.
- $Enc(K, File) \rightarrow C_{File}$: It takes symmetric key K and plaintext $File$ as input and outputs ciphertext C_{File}.
- Dec(K, C_{File}) $\rightarrow$ File: It takes symmetric key K and ciphertext C_{File} as input and yields plaintext $File$.

Pseudorandom generator. Security of an encryption mechanism depends on randomness of its encryption key. Pseudorandom Generator (PRG) produces an unpredictable string that can be used as encryption key. PRG appears as random string generator, however it follows a deterministic pattern to generate the output string.

PRG is a polynomial time deterministic function, PRG: $\{0,1\}^l \rightarrow \{0,1\}^{l'}$, where $l' \geq l$ and $l' - l$ is stretch of PRG. A PRG is secure if no efficient adversary can distinguish between pseudorandom value generated by PRG and pure random value with non-negligible advantage.

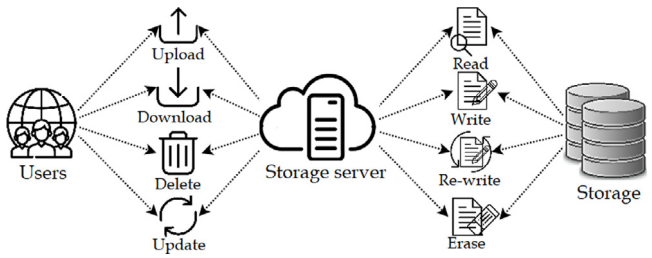


Fig. 1. System model.

3. Problem statement

This section discusses the architecture of SPARK system. Subsequently, we illustrate adversaries and their capabilities. And finally, we describe security assumptions, and security goals of our proposal.

3.1. System model

Fig. 1 illustrates the model of our proposal SPARK, which comprises following entities:

1. `User`($\mathcal{U}$): User is referred to as the client of storage server. $\mathcal{U}$ is the owner of data and stores it on the storage server. $\mathcal{U}$ invites deduplication process to reduce the storage and communication cost. For confidentiality of data, $\mathcal{U}$ encrypts data before outsourcing it to storage server.

2. `Storage Server`($\mathcal{SS}$): $\mathcal{SS}$ provides following storage services:
   (a) *Upload*: $\mathcal{U}$ computes deduplication tag (i.e., trimmed hash value of file F) and sends it to $\mathcal{SS}$. $\mathcal{SS}$ checks whether deduplication tag is present or not. If not, then $\mathcal{SS}$ asks $\mathcal{U}$ to upload the file. $\mathcal{U}$ generates a random key, encrypt the file using it and uploads the ciphertext to $\mathcal{SS}$. Otherwise, $\mathcal{SS}$ allows $\mathcal{U}$ to learn the random key from existing file owners. Further, $\mathcal{SS}$ permits $\mathcal{U}$ to access the file.
   (b) *Download*: $\mathcal{U}$ sends download request with deduplication tag and file tag (i.e., hash value of ciphertext). $\mathcal{SS}$ verifies whether $\mathcal{U}$ is the owner of the file. If yes, then $\mathcal{SS}$ sends ciphertext file. Further, $\mathcal{U}$ decrypts it with the secret key.
   (c) *Delete*: $\mathcal{U}$ requests to delete a file. $\mathcal{SS}$ verifies whether $\mathcal{U}$ is the owner of the file. $\mathcal{SS}$ removes entry of $\mathcal{U}$ from owners' list. In case, if owners' list is empty, $\mathcal{SS}$ erases ciphertext and associated metadata.
   (d) *Update*: Update consists of following operations: (i) "Delete" operation on existing version of file, and (ii) "Upload" operation on updated version of file.

3. `Storage`: It is a physical storage system that provides *Read*, *Write*, *Re-write*, and *Erase* operations.

3.2. Threat model and assumptions

In this paper, we deal with following malicious entities: (i) honest-but-curious $\mathcal{SS}$; (ii) malicious $\mathcal{U}$.

- `Honest-but-curious` $\mathcal{SS}$: We assume $\mathcal{SS}$ as honest-but-curious entity. That means, $\mathcal{SS}$ honestly executes the operations requested by $\mathcal{U}$s. But, it is curious to learn about plaintext from outsourced content (ciphertext, deduplication tags, and file tags). Secondly, $\mathcal{SS}$ may perform offline dictionary attack since it has infinite access to outsourced ciphertext, deduplication tags, and file tags. In addition, $\mathcal{SS}$ may also be aware of a portion of the plaintext file. $\mathcal{SS}$ constructs the versions of deduplication tag, file tag, or ciphertext by guessing unknown portion of file. Next, it compares them with stored content and identifies the version which gets matched.
  For unpredictable file, $\mathcal{SS}$ may be aware of *seed* value (discussed in Section 4), hash of plaintext, and a portion of the file along with outsourced content. $\mathcal{SS}$ tries to compute the encryption key using this knowledge.
- `Malicious User`: Malicious $\mathcal{U}$ may have knowledge about deduplication tag, file tag, and ciphertext of particular file. She attempts to learn about plaintext from this knowledge.
  Secondly, $\mathcal{U}$ may perform online dictionary attack. In particular, she may be aware of a fraction of the file. She constructs the versions of file by predicting unknown portion of file. She uploads them to $\mathcal{SS}$ and observes the deduplication. Alternately, she may compromise $\mathcal{SS}$ and perform offline dictionary attack as discussed above. For unpredictable file, $\mathcal{U}$ may know *seed* value, hash value, and a fraction of the plaintext file. $\mathcal{U}$ tries to generate the encryption key using this knowledge.
  Thirdly, malicious $\mathcal{U}$ may launch tag inconsistency attack. For example, she uploads ciphertext of file B (C_{FileB}) with deduplication tag of File A. If deduplication tag is not computed from ciphertext, then $\mathcal{SS}$ fails to detect inconsistency between uploaded ciphertext and deduplication tag. As result, when a genuine $\mathcal{U}$ sends upload request with deduplication tag of File A, $\mathcal{SS}$ will permit access to C_{FileB}. Furthermore, when she requests for File A, she gets $C_{FileB}-$ her File A is lost.
  And lastly, a malicious $\mathcal{U}$ may flood large number of messages to disrupt working of $\mathcal{U}$s or $\mathcal{SS}$.

3.2.1. Assumptions

1. $\mathcal{U}$s and $\mathcal{SS}$ have pairs of public key and secret key.
2. $\mathcal{SS}$ is always online and has significantly high storage and computational capacity.
3. The communication channels in SPARK are secure.
4. $\mathcal{U}$ is capable of generating a secure pseudo random key value.
5. $\mathcal{U}$ keeps the secret key and file encryption keys in protected local storage.
6. There is no direct communication among $\mathcal{U}$s. They interact with one another via $\mathcal{SS}$.

3.3. Security goals

1. `Confidentiality`: The primary goal of SPARK is to deliver confidentiality of outsourced files. For that, $\mathcal{U}$ encrypts the file using a semantically secure encryption and a random key. As a result, the adversary cannot successfully perform online-offline dictionary attacks on ciphertext. Moreover, SPARK leverages trimmed hash value as deduplication tag of file for protection against online-offline dictionary attacks via deduplication tag.
2. `Security against tag inconsistency`: In SPARK, $\mathcal{SS}$ asks for file tag (hash of ciphertext) at time of subsequent upload. If it does not match with existing file tags, then $\mathcal{SS}$ requests $\mathcal{U}$ to upload the file. In this way, $\mathcal{U}$ will get access to correct file, even if tag inconsistency is there.

4. Proposed method

In this section, we discuss our approach "SPARK". It consists of following basic storage operations (1) *Upload*; (2) *Download*; (3) *Delete*; and (4) *Update File*.

1. **File Upload**: For file upload, $\mathcal{U}$ computes DedupTag, i.e., a trimmed hash value containing first t bits of Hash(File), where b is size of Hash digest and t tends to $b-1$. Therefore, it is possible that more than one file has same DedupTag implying

that DedupTag is not collision resistant. The maximum number of files that can be linked with DedupTag is 2^{b-t}. After computing DedupTag, $\mathcal{U}$ will send it to $\mathcal{SS}$. Here, there are two possibilities: (a) DedupTag is not present on $\mathcal{SS}$, or (b) DedupTag is present on $\mathcal{SS}$.

(a) If DedupTag is not present on $\mathcal{SS}$, $\mathcal{U}$ generates a random *seed* value. Then, she computes an encryption key "*Key*" by executing KeyGen procedure (as discussed in Algorithm 2). Using this "*Key*", $\mathcal{U}$ encrypts the file. Next, she generates "*FileTag*", i.e., hash of ciphertext. $\mathcal{U}$ uploads (DedupTag, FileTag, C_{File}) as described in Algorithm 1. $\mathcal{U}$ keeps *Key* and *seed* secret in local storage. $\mathcal{SS}$ verifies the integrity of received file by checking hash of C_{File} with $FileTag$ and stores (DedupTag, FileTag, C_{File}) to storage (as discussed in Algorithm 1: server_fupload).

(b) If DedupTag is present on $\mathcal{SS}$, then $\mathcal{SS}$ asks $\mathcal{U}$ to learn the secret *seed* from existing file owners. After learning *seed*, $\mathcal{U}$ computes "*Key*" using KeyGen as discussed in Algorithm 2 and encrypts the file. $\mathcal{U}$ computes "*FileTag*" and sends it to $\mathcal{SS}$. $\mathcal{SS}$ gives access to such C_{File} whose $DedupTag$ and $FileTag$ are matching with received $DedupTag$ and $FileTag$. In case, if $\mathcal{SS}$ finds it different than existing filetags, then it asks $\mathcal{U}$ to upload the file.

Algorithm 1 File Upload

procedure User: User_Upload($File$)
  $DedupTag \leftarrow$ Extract(Hash($File$),t)
  if VerifyDedup($DedupTag$) = TRUE **then**
    $seed \leftarrow$ LearnKey(PKs, Hash($File$))
    $Key \leftarrow$ KeyGen($seed$, $File$)
    $C_{File} \leftarrow$ Enc(Key, $File$)
    $FileTag \leftarrow$ Hash(C_{File})
    if Server_SUpload($DedupTag$, $FileTag$) = TRUE **then**
      Return
    else
      goto step 13
  else
    $seed \xleftarrow{\$} \{0,1\}^b$
    $Key \leftarrow$ KeyGen($seed$, $File$)
    $C_{File} \leftarrow$ Enc(Key, $File$)
    $FileTag \leftarrow$ Hash(C_{File})
    Server_FUpload($DedupTag$, $FileTag$, C_{File})
procedure Storage Server: VerifyDedup($DedupTag$)
  if $DedupTag \in$ DedupTagList **then**
    Return "True"
  else
    Return "False"
procedure Storage Server: Server_FUpload($DedupTag$, $FileTag$, C_{File})
  if Hash(C_{File}) $\neq FileTag$ **then**
    Return
  else
    Files($DedupTag$) $\leftarrow$ Files($DedupTag$) $\cup$ C_{File}
    FileTags($DedupTag$) $\leftarrow$ FileTags($DedupTag$) $\cup$ $FileTag$
    Owners($DedupTag$, $FileTag$) $\leftarrow$ Owners($DedupTag$, $FileTag$) $\cup$ $\mathcal{U}$
procedure Storage Server: Server_SUpload($DedupTag$, $FileTag$)
  if $FileTag \in$ FileTags($DedupTag$) **then**
    Owners($DedupTag$, $FileTag$) $\leftarrow$ Owners($DedupTag$, $FileTag$) $\cup$ $\mathcal{U}$
    Return "True"
  else
    Return "False"

KeyGen: In this process, $\mathcal{U}$ splits the file into 512 bits blocks. $\mathcal{U}$ extracts n^2 blocks from them, where $n = \left\lfloor \sqrt{no.\ of\ file\ blocks} \right\rfloor$.

A matrix M_{File} of size $n \times n$ is formed from these blocks.
$\mathcal{U}$ computes hash of each row of M_{File}, i.e., for ith row: B_i = Hash($M_{File}[i][1] \parallel M_{File}[i][2] \ldots \parallel M_{File}[i][n]$), $|B_i| = b$. $\mathcal{U}$ generates a set of pseudorandom values, Y_i = PRG(($P_{i-1} \oplus Q_{i-1}$) $\parallel B_i$), where $|Y_i| = 2b + x$, $P_0 = seed$, $Q_0 = \{0\}^b$, and $|seed| = b$. Here, PRG takes input of $2b$ bits and outputs Y_is of $2b + x$ bits. We consider first b left most bits of Y_i as P_i, following b bits as Q_i, and remaining x bits as X_i. KeyGen discards X_i and leverages P_i and Q_i for *seed* value to generate next pseudorandom value. In this way, key generation process outputs ($P_n \oplus Q_n$) as random encryption key as shown in Fig. 2. Here, we can say that only a $\mathcal{U}$ who owns complete file, is able to compute the correct encryption key.

Algorithm 2 KeyGen: User

procedure User: KeyGen($seed$, $File$)
  {Blocks} $\leftarrow$ Split($File$)
  $n \leftarrow \left\lfloor \sqrt{no.\ of\ blocks} \right\rfloor$
  $M_{File} \leftarrow Matrix(\{Blocks\}_{n \times n})$
  for i=1 to n **do**
    $B_i \leftarrow$ Hash($M_{File}[i][1] \ldots \parallel M_{File}[i][n]$)
  $P_0 \leftarrow seed$, $Q_0 \leftarrow \{0\}^b$
  for i=1 to n **do**
    $Y_i \leftarrow$ PRG(($P_{i-1} \oplus Q_{i-1}$) $\parallel B_i$)
    $P_i \parallel Q_i \parallel X_i \leftarrow Y_i$
    Discard(X_i)
  Return($P_n \oplus Q_n$)

Learn Key: When DedupTag is present on storage, $\mathcal{U}$ needs to learn *seed* value of keys corresponding to this DedupTag as mentioned in Algorithm 3.

Algorithm 3 LearnKey: User

procedure Requesting User: LearnKey(PKs, Hash($File$))
  $r \xleftarrow{\$} \{0,1\}^\lambda$
  $x_i \leftarrow Hash(File)^{PK} \parallel r^{PK}$
  Send(PKs,x_i)
  $y_i \leftarrow$ Receive()
  $z_i \leftarrow y_i^{SK}$
  if $z_i[2] = r$ **then**
    $seed \leftarrow z_i[1]$
  else
    $seed \leftarrow$ NULL
  Return $seed$
procedure Owner: LearnKey($seed$,Hash($File$),PK)
  $x_i \leftarrow$ Receive()
  $w_i \leftarrow x_i^{SK_i}$
  if $w_i[1]$ = Hash(File) **then**
    $y_i \leftarrow seed^{PK} \parallel w_i[2]^{PK}$
  else
    $y_i \leftarrow$ NULL $\parallel w_i[2]^{PK}$
  Return y_i

As stated earlier, a DedupTag can be linked with multiple files. So, $\mathcal{U}$ may need to communicate with a large number of file owners and it incurs high communication overhead.
To minimize the communication overhead, SPARK classifies the owners per file. $\mathcal{SS}$ sends public keys PKs of one owner per file to the requesting $\mathcal{U}$. In this way, $\mathcal{U}$ needs to communicate with maximum 2^{b-t} number of file owners, where $t \rightarrow b-1$. Further, $\mathcal{U}$ generates a random nonce r. She encrypts the Hash($File$) and r using PK and sends it to corresponding owners via SS. Each

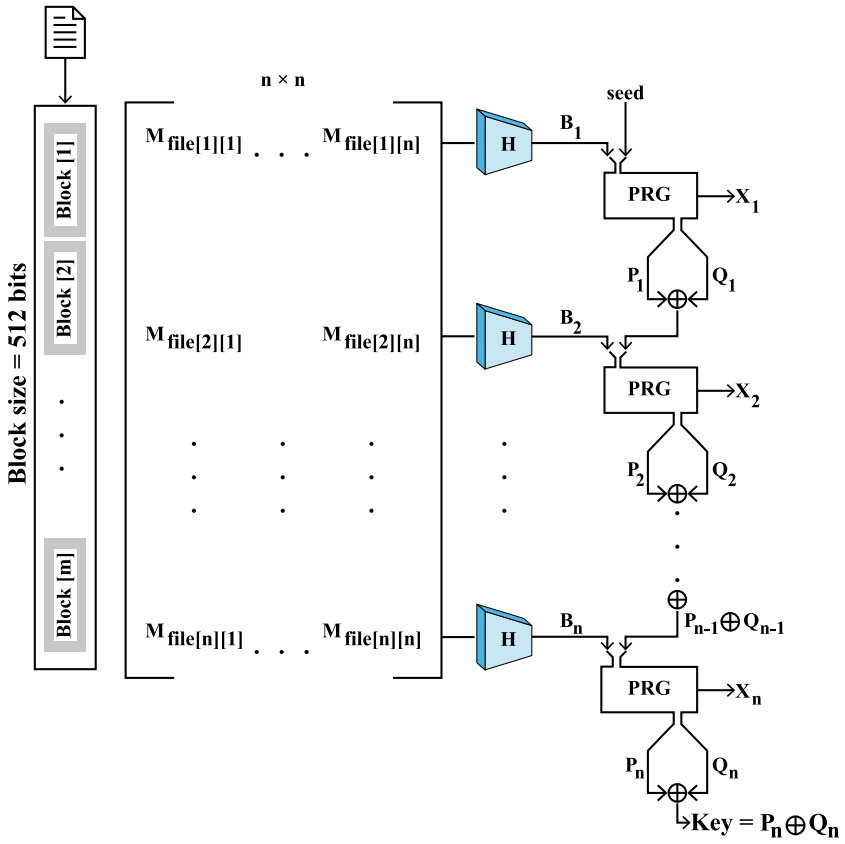


Fig. 2. KeyGen.

owner decrypts the received message using secret key SK_i. If she possesses same Hash(File) as in received message, then she encrypts *seed* and *r* using $PK_{\mathcal{U}}$ and sends it $\mathcal{U}$. Otherwise, owner returns NULL value with *r*. Next, $\mathcal{U}$ decrypts the received message, verifies the correctness of *r* and learns *seed*. Nevertheless, adversary can exploit this procedure for flooding attack against $\mathcal{U}$ or $\mathcal{SS}$. We adopt rate limiting strategy [12] to avoid flooding attack.

2. **File Download**: $\mathcal{U}$ sends DedupTag and FileTag of a file to $\mathcal{SS}$. $\mathcal{SS}$ checks whether requesting $\mathcal{U}$ resides in owners' list or not. If $\mathcal{U}$ is an owner of the file, then $\mathcal{SS}$ sends C_{File} along with Hash(C_{File}). $\mathcal{U}$ verifies integrity of received data and decrypts C_{File} using secret key.
3. **File Delete**: $\mathcal{U}$ sends DedupTag and FileTag of a file to $\mathcal{SS}$. $\mathcal{SS}$ checks whether requesting $\mathcal{U}$ is owner of file or not. If $\mathcal{U}$ is an owner, then $\mathcal{SS}$ removes entry of $\mathcal{U}$ from owners' list. If owners' list is empty, then $\mathcal{U}$ erases C_{File} and corresponding meta-data from storage.
4. **File Update**: When $\mathcal{U}$ requests to update a file, SPARK executes Delete operation on existing file and Upload operation on updated file.

5. Security analysis

This section presents the security of SPARK in terms of "Confidentiality" and "Tag consistency".

5.1. Confidentiality

In SPARK, we assume $\mathcal{U}$ and $\mathcal{SS}$ as untrusted entities. Additionally, we contemplate $\mathcal{U}$ and $\mathcal{SS}$ having partial knowledge of file. They launch online-offline dictionary attacks as discussed in Threat model.

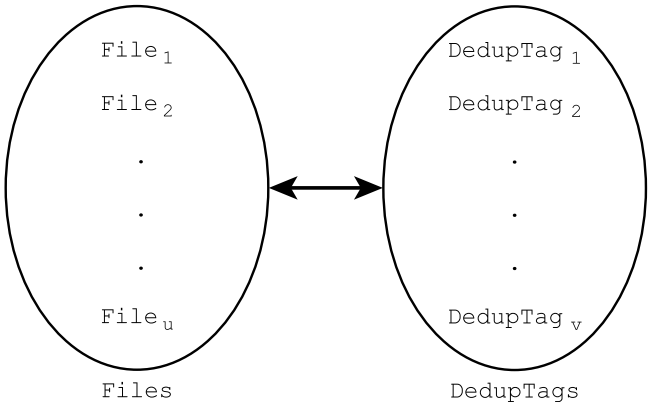


Fig. 3. Mapping between files and DedupTags.

To achieve data confidentiality, SPARK encrypts the data with a random key during initial upload. Moreover, SPARK leverages semantically secure encryption algorithm. Hence, it is computationally hard to perform successful offline dictionary attack on ciphertext. In particular, malicious $\mathcal{SS}$ cannot compute ciphertext versions of predicted plaintext versions, since it is unable to guess random encryption key. Hence, a malicious $\mathcal{SS}$ is not able to launch successful offline dictionary attack with non-negligible advantage. Similarly, a malicious $\mathcal{U}$ who compromise $\mathcal{SS}$ for a while, is also not successfully able to launch offline dictionary attack.

Online-offline dictionary attacks can be performed by guessing the DedupTags of files. We describe the mapping between files and DedupTags stored at cloud storage in Fig. 3. If DedupTag is generated using a collision resistant hash function, then each distinct file has a unique DedupTag. In other words, $v = u$. Therefore, an adversary who has

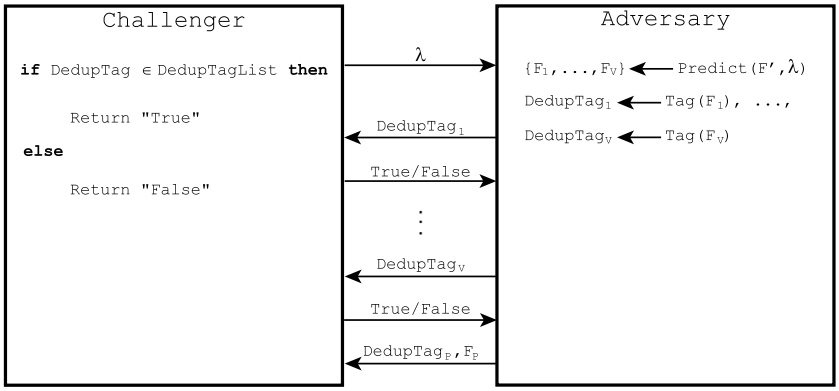


Fig. 4. Security game for online-offline dictionary attacks using DedupTag.

partial information about the file, will predict the dictionary versions, compute the DedupTag of each version, upload them, and observes the deduplication process. She identifies the version which causes deduplication.

For protection against this security threat, SPARK leverages trimmed hash value as DedupTag. In particular, DedupTag is first t bits of hash value, which is not collision resistant. In this case, $v = \frac{u}{2^{(b-t)}}$, where b = |hash value|hash value and t = |trimmed hash value| trimmed hash value |. That means, it is possible that multiple files will have the same DedupTag. As a result, an adversary will get confused with multiple files while executing the online-offline dictionary attacks.

In the following, we discuss the security game for the adversary who mounts the online-offline dictionary attacks using DedupTags of predicted versions of a file, as described in Fig. 4. In the security game, the challenger provides security parameter λ to the adversary. The adversary predicts the versions $\{F_1, \ldots, F_V\}$ of a file F using her knowledge F' about the file F. Then she computes DedupTags of every versions and sends them to the challenger. Challenger executes VerifyDedupTag procedure (as discussed in Algorithm 2) for each query. If a DedupTag is present on storage, then it returns "True", otherwise "False". Now, the adversary decides which of the predicted versions is correct based on the received responses of the queries. The advantage of the adversary to successfully predict the file is

$$DedupTag_{Adv} = \frac{1}{2^{(b-t)}} \tag{1}$$

Let us assume that F_P is the correct guess among $\{F_1, \ldots, F_V\}$. In other words, $F_P = F$. If Tag is a collision resistant hash function, then each distinct file has a unique DedupTag. Here, since F_P (= F) is present on cloud storage, DedupTag of F_P will get matched to DedupTag of F. And, since other predicted versions are not matching, challenger returns "False" for their queries. In this way, $DedupTag_{Adv} = \frac{1}{2^{(b-t)}} = \frac{1}{2^{(b-b)}} = 1$, when Tag is a collision resistant hash function. On the other hand, in SPARK, Tag is not a collision resistant hash function. As a result, it is possible that multiple files will have the same DedupTag. In this case, $v < u$ as described in Fig. 3, and hence DedupTags of predicted versions including F_P may get matched with DedupTags of other files on storage. Therefore, $DedupTag_{Adv} = \frac{1}{2^{(b-t)}}$, where $t < b$.

For unpredictable files, adversary is aware of DedupTag, FileTag and some portion of plaintext and unable to predict unknown portion. Further, she may learn Hash($File$) and $seed$ by compromising a file owner. Still, it is computationally difficult to learn the encryption key, since SPARK key generation procedure requires knowledge of complete file. In particular, KeyGen procedure takes complete plaintext file and $seed$ as input and outputs an encryption key using combination of secure pseudorandom generator and cryptographic hash function. Thus, we can say that, the knowledge of DedupTag, FileTag, portion of file, Hash($File$) and $seed$ will not be advantageous for adversary to learn the encryption key.

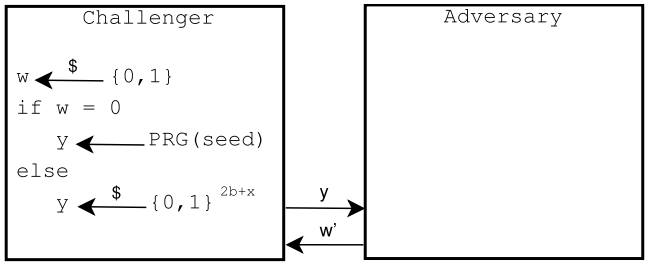


Fig. 5. Security game against pseudorandom generator.

The adversary can try to disturb the working of $\mathcal{U}$s and SS by flooding large number of messages to them. For protection against such attacks on $\mathcal{U}$s, SPARK allows all communication among $\mathcal{U}$s via SS only. Furthermore, we limit the number of attempts by adversary using rate limiting strategy [12]. This strategy slows down the message flooding attack on SS.

Moreover, we assume that an adversary may be aware of DedupTag-FileTag. With knowledge of DedupTag-FileTag, adversary can learn ciphertext file. However, adversary is not able to learn anything about plaintext from ciphertext since file is encoded using semantically secure encryption and secure random key. Next, we prove that SPARK KeyGen is having indistinguishability from random string generator with underlying secure PRG. In Fig. 5, we discuss security game for adversary mounting attack against pseudorandom generator as discussed in [16, 17].

Advantage of the adversary to win in the experiment is as follows:

$$PRG_{Adv} = |Pr[E_1] - Pr[E_0]|, \tag{2}$$

where E_0 = Event that adversary yields w' = 1 when w = 0, and E_1 = Event that adversary yields w' = 1 when w = 1.

Definition 1. A pseudorandom generator is secure if no efficient adversary wins the security game with negligible advantage.

Theorem 1. *If PRG is a secure pseudorandom generator, then KeyGen is also a secure pseudorandom generator.*

Proof. SPARK KeyGen is composed with secure pseudorandom generator, PRG. In this theorem, we will prove that if PRG is secure then KeyGen is also secure.

We denote Adv_1 as the adversary that plays security game with respect to KeyGen. At first, we define $n+1$ number of intermediate security games, $inter_0$, $inter_1$, ..., $inter_n$, where $n = \left\lfloor \sqrt{no.\,of\ file\ blocks} \right\rfloor$ as discussed in Section 4.

For j = 0, 1, …, n, $inter_j$ is a security game played between Adv_1 and challenger. In each game, challenger forms a set of j random values preceded by $n-j$ pseudorandom values which are described as follows.

- $Y_1 \xleftarrow{\$} \{0,1\}^{2b+x}, \ldots, Y_j \xleftarrow{\$} \{0,1\}^{2b+x}$,
- $Y_{j+1} \leftarrow PRG((P_j \oplus Q_j) \parallel B_{j+1}), \ldots, Y_n \leftarrow PRG((P_{n-1} \oplus Q_{n-1}) \parallel B_n)$.

Challenger sends following sets of values to Adv_1 in the intermediate games.

$$Inter_0 : Y_1 \leftarrow PRG(seed \parallel B_1), \ldots, Y_n \leftarrow PRG((P_{n-1} \oplus Q_{n-1}) \parallel B_n)$$

$$Inter_1 : Y_1 \xleftarrow{\$} \{0,1\}^{2b+x} \ldots, Y_n \leftarrow PRG((P_{n-1} \oplus Q_{n-1}) \parallel B_n)$$

$$\ldots$$

$$Inter_n : Y_1 \xleftarrow{\$} \{0,1\}^{2b+x} \ldots, Y_n \xleftarrow{\$} \{0,1\}^{2b+x}.$$

We denote Pr_j as probability of event that Adv_1 outputs 1 in $Inter_j$ security game. Moreover, Pr_0 is equal to probability of event where Adv_1 outputs 1 in security game with $w = 0$. Similarly, Pr_n is equal to probability of event where Adv_1 outputs 1 in security game with $w = 1$. Thus, advantage of Adv_1 is $PRG_{adv}[Adv_1, KeyGen] = |Pr_n - Pr_0|$.

Now, we denote Adv_2 as the adversary which plays security game with PRG. When Adv_2 receives a value y, Adv_2 plays role of Adv_2's challenger as follows.

Security game between Adv_2 and challenger

$z \xleftarrow{\$} \{1, \ldots, n\}$
$Y_1 \xleftarrow{\$} \{0,1\}^{2b+x}, \ldots, Y_{z-1} \xleftarrow{\$} \{0,1\}^{2b+x}$,
$Y_z \leftarrow y$,
$Y_{z+1} \leftarrow PRG((P_z \oplus Q_z) \parallel B_{z+1}), \ldots, Y_n \leftarrow PRG((P_{n-1} \oplus Q_{n-1}) \parallel B_n)$
Send $(Y_1, \ldots, Y_n)$ to Adv_1

Here, Adv_2 outputs whatever Adv_1 yields.

We define an event E_0 is that Adv_2 outputs 1 in security game with $w = 0$. Similarly, event E_1 is that Adv_2 outputs 1 in security game with $w = 1$.

Here, for $z = j$ and $w = 0$, security game between Adv_2 and challenger is identical to intermediate security game $Inter_{j-1}$ where $j-1$ random values and $n-j+1$ pseudorandom values are sent to adversary.

Similarly, for $z = j$ and $w = 1$, security game between Adv_2 and challenger is identical to intermediate security game $Inter_j$ with j random values and $n-j$ pseudorandom values.

Hence, probability of event E_0 occurs when $z = j$ is

$$Pr[E_0|z=j] = \sum_{j=1}^{n} Pr[E_0|z=j] \times Pr[z=j]$$
$$= \frac{1}{n}\sum_{j=1}^{n} Pr[E_0|z=j] = \frac{1}{n}\sum_{j=1}^{n} Pr_{j-1}. \quad (3)$$

Similarly, probability of event E_1 occurs when $z = j$ is

$$Pr[E_1|z=j] = \sum_{j=1}^{n} Pr[E_1|z=j] \times Pr[z=j]$$
$$= \frac{1}{n}\sum_{j=1}^{n} Pr[E_1|z=j] = \frac{1}{n}\sum_{j=1}^{n} Pr_j. \quad (4)$$

Advantage of Adv_2 is

$$PRG_{adv}[Adv_2, PRG] = |Pr[E_1] - Pr[E_0]|$$
$$= |\frac{1}{n} \times \sum_{j=1}^{n} Pr_j - \frac{1}{n} \times \sum_{j=1}^{n} Pr_{j-1}|$$
$$= \frac{1}{n}|\sum_{j=1}^{n} Pr_j - \sum_{j=1}^{n} Pr_{j-1}|$$
$$= \frac{1}{n}|Pr_n - Pr_0|. \quad (5)$$

In this way,

$$PRG_{adv}[Adv_2, PRG] = \frac{1}{n} \times PRG_{adv}[Adv_1, KeyGen]$$
$$\Rightarrow PRG_{adv}[Adv_1, KeyGen] = n \times PRG_{adv}[Adv_2, PRG]. \quad (6)$$

Since PRG is a secure pseudorandom generator, $PRG_{adv}[Adv_2, PRG]$ is negligible. In addition, n is poly-bounded value. Hence, $PRG_{adv}[Adv_1, KeyGen]$ is also negligible.

In other words, "If PRG is a secure pseudorandom generator, then KeyGen is also a secure pseudorandom generator". □

Let us now examine the security of SPARK against dictionary attacks via Theorem 2. In SPARK system, KeyGen plays pivotal role for security against dictionary attacks. Hence, we prove security of SPARK in terms of security of KeyGen.

Theorem 2. *If KeyGen is secure against online-offline dictionary attacks, then SPARK is secure against online-offline dictionary attacks.*

Proof. We prove this theorem by contrapositive method. Contrapositive statement of the theorem is as follows: "If SPARK is not secure against online-offline dictionary attacks, then KeyGen is not secure against online-offline dictionary attacks".

In other words, if there exist a Probabilistic Polynomial Time (PPT) adversary A which breaks security of SPARK, then we can construct PPT adversary B which breaks KeyGen. We discuss security game of A and B as follows.

As shown in Fig. 6, B is PPT adversary of KeyGen mechanism. Challenger of B provides security parameter λ. B simulates challenger for A. B forwards security parameter λ to A. A is PPT adversary of SPARK. A sends two files F_0 and F_1. B selects value of w randomly over 0 and 1 and forwards F_w to her challenger. By querying KeyGen oracle, challenger of B generates K_w for F_w. Challenger also generates random string R_w of same length as $|K_w|$. Challenger randomly chooses over K_w and R_w and returns K_w'. B encrypts F_w by K_w' and forwards ciphertext C_w. A receives C_w and she already knows F_0 and F_1. Based on such knowledge, she tries to learn C_w is ciphertext of F_0 or F_1. A returns $w' = 0$ if she believes C_w is ciphertext of F_0, otherwise $w' = 1$. If $w' \neq w$, then A is not able to learn any information from C_w. The reason is that file may be encrypted with a pure random key which is not related to plaintext file. In this case, B replies R to her challenger. On the other hand, if $w' = w$, then A is able to learn information from C_w. That means file may be encrypted with a key which is generated from the plaintext file. In this case, B returns K to her challenger.

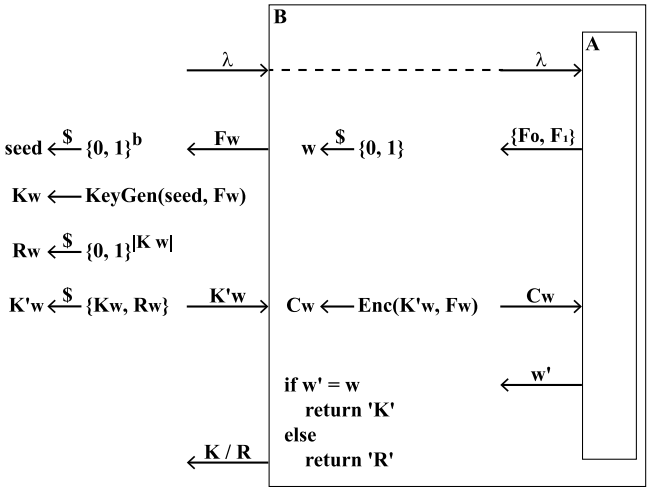


Fig. 6. Construction of PPT adversary B.

Lemma 1. *If A is PPT algorithm, then B is also PPT algorithm.*

Proof. We can also state the lemma as follows: "If A make polynomially many queries, then B also makes polynomially many queries". B's running time = Time for p(n) queries + A's running time + other.

Suppose, B executes $p(n)$ queries to her challenger. Each query takes constant time for execution. i.e. O(1). Secondly, B executes A inherently. Running time of A is $q(n)$ is considered in time to execute B. For all $p(n)$ queries, B selects w randomly, encrypts F_w and checks whether $w' = w$. These operations consumes $r(n)$ polynomial time. B consumes 1 unit time for forwarding security parameter and 1 unit time for forwarding result to the challenger. Hence, B's running time is

$$\begin{aligned} B_{time} &= (p(n) \times O(1)) + q(n) + r(n) + 2 \\ &= O(max(n^a, n^b, n^c)) = O(n^d) \\ &= polynomial\ time \end{aligned} \tag{7}$$

Here p, q and r are polynomial functions, $p(n) = O(n^a)$, $q(n) = O(n^b)$, $r(n) = O(n^c)$, $d = max(a, b, c)$.

It implies that B is PPT algorithm. □

Lemma 2. *B is indistinguishable from real challenger of A.*

Proof. B provides security parameter λ to A. Subsequently, B accepts $\{F_0, F_1\}$ from B. For each query $\{F_0, F_1\}$, B responses C_w and accepts guess w' of A. Challenger of A works same as B. Hence, we can say that B is indistinguishable from real challenger of A. □

Lemma 3. *If A wins with non negligible probability, then B wins with non negligible probability.*

Proof. If A wins with non negligible probability, then

$$Pr[w' = w] = \frac{1}{2} + \epsilon(n), where\ \epsilon(n)\ is\ non - negligible. \tag{8}$$

If B wins with non negligible probability, then

$$\begin{aligned} &Pr[K'_w \leftarrow K_w | K'_w = K] = Pr[K'_w \leftarrow K_w | K'_w = R] + \epsilon(n) \\ &, where\ \epsilon(n)\ is\ non - negligible \end{aligned} \tag{9}$$

Hence, as per Eq. (2), A's winning probability is $\frac{1}{2}$ + negligible. It implies that winning probability of A is negligible. It proves contrapositive statement of Lemma 3.

$$\neg(Pr[B\ wins] > negl(n)) \Rightarrow \neg(Pr[A\ wins] > negl(n)) \quad \square \tag{10}$$

These lemmas imply that "If KeyGen is secure against online-offline dictionary attacks, then SPARK is secure against online-offline dictionary attacks". □

5.2. Security against tag inconsistency

An adversary can try to deviate the deduplication process via inserting tag inconsistency anomaly. Adversary uploads DedupTag of *File P* with ciphertext of *File Q* (C_{FileQ}). Since SPARK leverages randomized encryption, SS is unable to detect the inconsistency between DedupTag and ciphertext of file. For protection against this anomaly, SPARK SS asks for file tag(hash of ciphertext) during subsequent upload. Therefore, when a genuine subsequent uploader wants to upload *File P*, she submits DedupTag of *File P*. Now, she communicates with adversary owner and gets secret *seed*. She generates encryption key using *seed* and plaintext. She encrypts the file and computes Hash(C_{FileP}). Next, she uploads it to SS. However, SS finds Hash(C_{FileP}) and Hash(C_{FileQ}) not matching. It asks genuine user to upload C_{FileP} and make a separate entry for DedupTag and FileTag of C_{FileP} in its lookup table. As result, when genuine user requests to download C_{FileP} in future, she will get the correct file. In this way, tag inconsistency anomaly will not disturb the deduplication process in SPARK.

6. Performance analysis

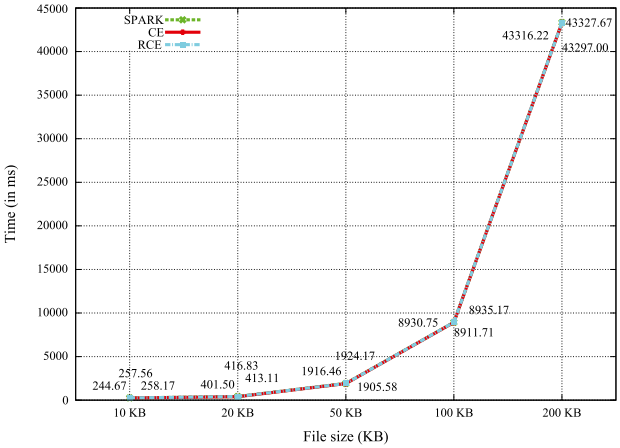
We have implemented SPARK system and evaluated performance of it. In the following, we discuss performance analysis of SPARK system in realistic environment.

6.1. Experimental setup

We have implemented SPARK User's code in C# Microsoft visual studio with ASP .NET web services on Intel i5 machine with 2.40 GHz and 4 GB RAM. We have developed SPARK Storage Server's code in C# Microsoft visual studio with Microsoft SQL Database services on Intel i5 machine with 3.19 GHz, 8 GB RAM, and 1 TB HDD. We have used collision resistant SHA-256 as cryptographic hash function and AES as symmetric cipher. We execute SPARK system over files of size 10 kB, 20 kB, 50 kB, 100 kB, and 200 kB, respectively. We observe upload, download, delete, and update time for these files. We consider size of DedupTag is 254 bits for our experiments.

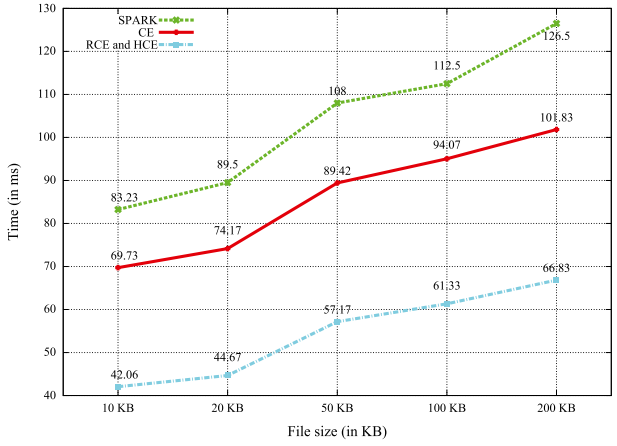
6.2. Evaluation

- Upload time: For file upload, $\mathcal{U}$ first sends DedupTag to SS. If DedupTag is not present, then SS asks to upload the file. To preserve the file confidentiality, $\mathcal{U}$ encrypts the file before outsourcing it. $\mathcal{U}$ generates a random encryption key using SPARK key generation mechanism. Next, $\mathcal{U}$ encrypts the file and sends it to SS. In brief, we can say that execution time of first upload is summation of time taken by key generation, encryption, and upload procedure.
  In Fig. 7, we compare execution time of first upload in SPARK with two generic secure deduplication schemes, Convergent Encryption (CE) [4] and Randomized Convergent Encryption (RCE) [10]. We observe that the execution time of first upload in these schemes is approximately equal to one another. KeyGen is a secure pseudorandom generator as it is composed with secure *seed* value, cryptographic Hash function and secure PRG. It implies that B wins with negligible probability. Therefore, $\epsilon(n)$ is negligible.
  At time of subsequent upload, SS asks $\mathcal{U}$ to communicate with file owners for retrieving *seed* value as discussed in Algorithm 3. Next, $\mathcal{U}$ computes the key using Algorithm 2. $\mathcal{U}$ encrypts the file using key and sends DedupTag and FileTag to SS. In this way,



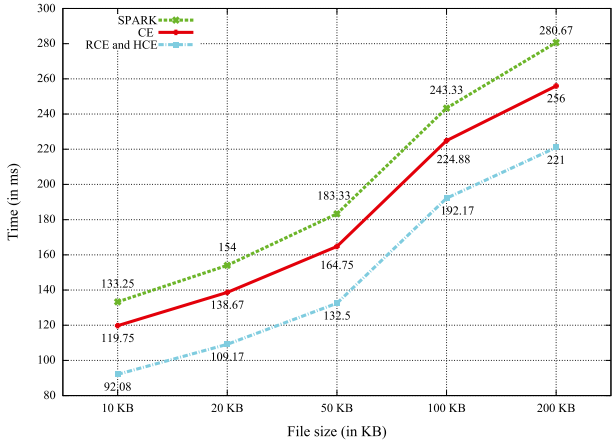
	10 KB	20 KB	50 KB	100 KB	200 KB
SPARK	258.17	416.83	1924.17	8935.17	43327.67
CE	244.67	401.50	1905.58	8911.71	43297.00
RCE	257.56	413.11	1916.46	8930.75	43316.22

Fig. 7. First upload time.



SPARK	83.23	89.5	108	112.5	126.5
CE	69.73	74.17	89.42	94.07	101.83
RCE	42.06	44.67	57.17	61.33	66.83

Fig. 8. Subsequent upload time.



SPARK	133.25	154	183.33	243.33	280.67
CE	119.75	138.67	164.75	224.88	256
RCE	92.08	109.17	132.5	192.17	221

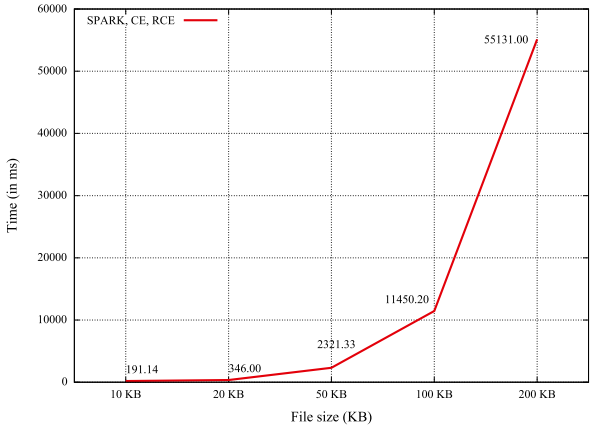
Fig. 10. Update time.

the execution time of subsequent upload is summation of time taken by random key retrieval, key generation, and encryption procedure.

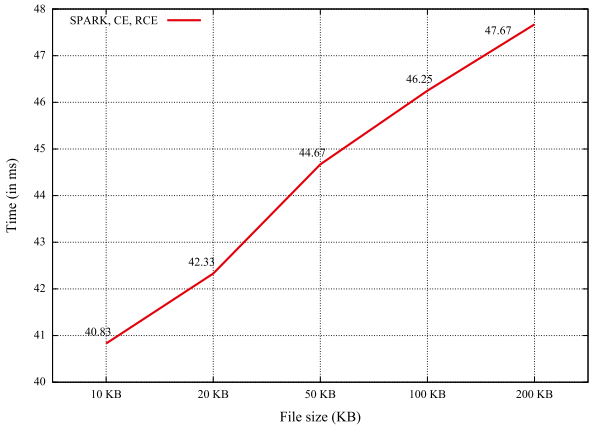
In CE [4], $\mathcal{U}$ encrypts the file using convergent key, computes DedupTag which is Hash(ciphertext file) and sends it to $\mathcal{SS}$. $\mathcal{SS}$ replies "True" and allows $\mathcal{U}$ to ciphertext file. The execution time of CE's subsequent upload is lesser than SPARK's subsequent upload as shown in Fig. 8. The reason behind such a difference in execution time is that CE's deterministic key generation process takes lesser time than SPARK's random key exchange and key generation process. However, CE is vulnerable to online-offline dictionary attacks due to deterministic key generation process. On the other hand, SPARK is secure against the online-offline dictionary attacks, as discussed in Section 5.

In RCE [10], $\mathcal{U}$ computes DedupTag, which is Hash(File), and sends it to $\mathcal{SS}$. $\mathcal{SS}$ replies "True" and allows $\mathcal{U}$ to access ciphertext of file and ciphertext of key. Execution time of RCE's subsequent upload is lesser than CE's and SPARK's subsequent upload. The reason behind such a difference in execution time is that $\mathcal{U}$ does not need to encrypt the file for checking duplication. However, RCE is vulnerable to tag inconsistency anomaly because $\mathcal{SS}$ is not able to verify the consistency between DedupTag and ciphertext during first and subsequent upload. On the other hand, SPARK provides security against tag inconsistency anomaly as discussed in Section 5.

- Download time: $\mathcal{U}$ sends DedupTag and FileTag to download a file. $\mathcal{SS}$ verifies $\mathcal{U}$'s ownership of the file and sends ciphertext file. $\mathcal{U}$ decrypts the ciphertext using encryption key. In download process of CE [4] and RCE [10] ciphertext is retrieved from $\mathcal{SS}$ and decrypted using encryption key, same as SPARK. In Fig. 9, we discuss analyzed execution time of file download for various file sizes.
- Delete time: $\mathcal{U}$ sends DedupTag and FileTag to delete a file. $\mathcal{SS}$ verifies $\mathcal{U}$'s ownership of the file and removes her record from file's ownership list. In CE [4] and RCE [10], delete operation is executed same as SPARK. Fig. 9 shows the execution time of delete procedure in our experiments.
- Update time: In SPARK, file update procedure is combination of delete and upload procedure. When $\mathcal{U}$ requests to update a file, $\mathcal{SS}$ removes her access from old version of file and executes upload operation for new version of file. In Fig. 10, we describe the execution time of update operation on various sized files. The difference in execution times of update procedure in CE, RCE and SPARK is because of difference in execution times of subsequent upload procedure in these schemes.



(a) Download time



(b) Delete time

Fig. 9. Download time, delete time.

6.3. Minimizing communication overhead

SPARK classifies the owners per file to minimize the communication overhead incurred by key learning process. In particular, SS creates and maintains a lookup table "DedupTag-FileTag-Owners-File pointer". When a $\mathcal{U}$ requests for subsequent upload, SS extracts FileTags corresponding to requested DedupTag. Next, SS selects one of the owners for each FileTag and sends their public keys PKs to $\mathcal{U}$. In this way, $\mathcal{U}$ needs to communicate with only x number of owners, where x is number of FileTags corresponding to requested DedupTag. Maximum number of FileTags corresponding to requested DedupTag can be 2^{b-t}, where t is $|DedupTag|$ and t tends to $b-1$.

7. Related work

We reviewed various existing approaches which addresses the confidentiality and tag inconsistency anomaly in deduplication system. We observe that many of existing proposals leverages CE or assistance of additional server. Whereas SPARK provides security against online-offline dictionary attacks and tag inconsistency anomaly without additional servers. Table 1 describes the comparison of SPARK with the recent state-of-the-art secure deduplication schemes.

For confidentiality of deduplication system, Douceur et al. [4] proposed the first encryption mechanism "Convergent Encryption" (CE). In CE, file is encrypted with a convergent key which is the hash value of file. CE is adopted by various state-of-the-art as follows. Storer et al. [5] proposed Authenticated Model and Anonymous Model based on CE. However, Authenticated model reveals the information about owners of particular file to server. And anonymous model allows a user to leverage the storage services as Content Distribution Network. In [6], authors proposed CE based efficient chunking for deduplication. In this scheme, they leveraged Two Threshold, Two Divisors (TTTD) algorithm for chunking and B + Tree for searching.

Puzio et al. [7] proposed "CloudDedup", an encryption scheme which equips additional deterministic encryption layer followed by CE. In this proposal, user encrypts the data with CE and sends to additional server. The server further encrypts received data with own secret deterministic key. Hence, data is vulnerable to dictionary attacks until additional server re-encrypts it. Xu et al. [18] proposed a deduplication scheme in which file is encrypted with random key. And random key is encrypted with output of keyed hash function. This scheme detects poisoned entry of file and tag at server.

Bellare et al. [10] proposed three formal models of CE. Authors proposed Random CE (RCE) in which file is encrypted with random key. And random key is encrypted with convergent key. In [19], Bellare et al. extended MLE theory to interactive MLE with upload and download protocols. In [20], authors presented a new term "deduplication consistency" which allows servers to verify whether ciphertext is well-formed for deduplication or not. The scheme uses non-interactive zero knowledge proof for verifiability.

Rongmao et al. [8] extended the concept MLE to block level MLE. The updatable block level MLE is proposed in [25]. In these schemes, block level convergent encryption generates large number of convergent keys. Hence, key management is a critical issue for users. In [9], authors proposed efficient and reliable key management scheme. In this scheme, a key is split into key shares and distributed across multiple key servers using Ramp secret sharing scheme. In [11], authors proposed server side deduplication with additional key management server. In this scheme, user sends hash of file blocks and key management server responds encryption keys for them. Authors assumed that key management server is fully trusted and communication links between users and key management server are secure.

In [26], Harnik et al. outlined following weaknesses of convergent encryption: (i) Adversary is able to find the presence of particular file on storage server. (ii) Adversary can perform dictionary attacks on predictable files. (iii) Adversary can establish a covert channel in CE based deduplication system. As a solution to these weaknesses, authors proposed threshold based deduplication. In the following, we discuss various proposals that overcome the weaknesses of CE. Bellare et al. proposed DupLESS [12] which allows users to compute the encryption key by executing oblivious PRF with key server. Oblivious PRF allows users to compute keys without revealing information about plaintext file. For security against dictionary attacks, DupLESS consists of per-user request limit. Each user can send particular number of requests in a time-slot. In this scheme, if storage server and key server are compromised, then security of scheme reduces to simple CE's security. Puzio et al. [15] proposed PerfectDedup approach. In this approach, file is initially encrypted with random keys. Index server maintains popularity index for each file. When a file reaches popularity index, stored ciphertext is replaced by CE based ciphertext.

Jian et al. [21] proposed random encryption mechanism to defend the brute-force attacks. In this scheme, subsequent user retrieves the encryption key from former uploader of file by PAKE protocol. This scheme incurs communication overhead at user side. In particular, key exchange protocol incurs significant communication overhead, since user needs to communicate with all existing owners of a file. Secondly, an adversary having hash(file) is able to learn encryption key from file owners. In [27], authors proposed a formal security model for the aforementioned PAKE based approach. To protect the popularity information of the file from Storage Server, Stanek et al. [23] proposed a multilayer encryption scheme(inner layer = CE, outer layer = semantically secure encryption scheme) with aid of additional index repository service provider. Ramp secret sharing (RSSS) [13] allows high reliability of storage.

In [28], authors proposed secure random encryption mechanism for deduplicated storage. Target of the proposal is to protect the deduplicated storage against dictionary attacks on ciphertext. To achieve the goal, authors propose random key based encryption mechanism. In the proposal, first uploader selects a random key, encrypts the file with it and uploads to Storage Server. Moreover, first uploader encrypts the random key using set of hash values generated from the file as follows: $C_{Key} = Key \oplus B_1 \oplus B_2 ... \oplus B_n$. First uploader uploads the ciphertext of key along with ciphertext of file. As result, a subsequent uploader who possesses the same file is able to generate hash values $B_1, B_2, \ldots, B_n$ using the file and learn the encryption key as follows: $Key = C_{Key} \oplus B_1 \oplus B_2 ... \oplus B_n$.

We find security limitations in this approach as follows. This approach is vulnerable to tag inconsistency anomaly. First uploader can upload an inconsistent combination of {deduplication tag($File_A$), Ciphertext($File_B$)}, since deduplication tag is not derived from ciphertext. Secondly, XOR operation is used for encryption/decryption of the key. Suppose, adversary has knowledge about $x\%$ of file blocks in M_{File}. Hence, she is able to compute n × $x\%$ hash values using known blocks. If the remaining $(100 - x)\%$ of file blocks have identical data, then generated hash values will also be identical. XOR operation among such identical values cancels themselves and results 0s. Thirdly, adversary can launch online dictionary attack by uploading deduplication tags of predicted versions of file and observing deduplication.

CDStore scheme [24] provides security of deduplication using Convergent Dispersal, a secret sharing scheme, on multiple servers. Armknecht et al. proposed ClearBox [29], a transparent deduplication mechanism which advantages users by transparent storage cost reduction. Zheng et al. [22] proposed secure deduplication scheme for big data in Cloud. In this scheme, data is encrypted with random key and random key is encrypted with public key of trusted third party. When a user requests to upload same data, trusted party decrypts the random encryption key and sends to requesting user after challenge-response process. In [14], authors introduced hybrid architecture having public and private cloud for privileges based deduplication.

Convergent encryption is unable to provide forward and backward secrecy. Forward secrecy prevents access of user to file after deletion or modification of file. Backward secrecy prevents access of user who

Table 1
Comparison with recent state-of-the-art secure deduplication schemes.

State-of-the-art	CE	Additional server	Other limitations
Secure data deduplication [5]	✓	✗	–
A secure data deduplication framework [6]	✓	✗	–
ClouDedup [7]	✓	✓	–
Weak leakage-resilient client-side deduplication [18]	✗	✗	Key is encrypted by Hash(File)
Message-locked encryption and secure deduplication [10]	✓	✗	–
BL-MLE [8]	✓	✗	–
Lamassu [11]	✓	✓	Server side deduplication
PerfectDedup [15]	✓	✓	When popularity reaches threshold value, data is encrypted by only CE
Secure deduplication of encrypted data without additional independent servers [21]	✗	✗	Key exchange incurs high communication overhead
A hybrid cloud approach for secure authorized deduplication [14]	✗	✓	Priori knowledge about privileges and keys is required.
DupLESS [12]	✗	✓	If storage server and key server are compromised, then security of scheme is reduced to CE's security.
Secure deduplication with efficient and reliable convergent key management [9]	✓	✓	–
Deduplication on encrypted big data in cloud [22]	✗	✓	Assumption: third party consisting all encryption keys is fully trusted
Enhanced secure thresholded data deduplication [23]	✓	✓	–
CDStore [24]	✗	✓	–
SPARK	✗	✗	*None*

has not requested file upload yet. Hur et al. [30] proposed dynamic key management. In this scheme, server provides the forward and backward secrecy using dynamic group key based re-encryption. SecCloud [31] encrypts the file with random key and shares the key using RSSS. SecCloud$^+$ [32] introduces additional auditing authority with SecCloud to maintain integrity of stored data.

8. Conclusion and future work

In this paper, we address two important security threats on inter user client side deduplication system: dictionary attacks and tag inconsistency anomaly. We propose a novel approach Secure Pseudorandom Key-based Encryption for Deduplicated Storage, "SPARK". This approach encrypts the data using semantically secure cipher and the secure random key. We prove that SPARK is secure against the online-offline dictionary attacks. SPARK also guarantees security against tag inconsistency. For that, SPARK storage server asks for hash value of ciphertext before allowing access to outsourced ciphertext. If received hash value does not match with hash of existing ciphertext, then storage server asks user to upload the ciphertext. In this way, user will get access to correct the file even if tag inconsistency exists. We implement SPARK and analyze the efficiency of our approach in a realistic environment.

As a future work, SPARK can be extended with block level deduplication. We plan to develop a secure pseudorandom key-based encryption for block level deduplication along with key management policies.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Jay Dave: Conceptualization, Methodology, Software, Writing - original draft. **Parvez Faruki:** Validation, Investigation, Data curation. **Vijay Laxmi:** Visualization, Resources, Writing - review & editing. **Akka Zemmari:** Formal analysis, Writing - review & editing. **Manoj Gaur:** Supervision, Project administration, Funding acquisition. **Mauro Conti:** Visualization, Supervision, Writing - review & editing.

References

[1] C.V. Networking, Cisco Global Cloud Index: Forecast and Methodology, 2016–2021, White paper, Cisco Public, San Jose, 2016.
[2] M. Dutch, Understanding data deduplication ratios, in: SNIA Data Management Forum, 2008, p. 7.
[3] Y. Shin, D. Koo, J. Hur, A survey of secure data deduplication schemes for cloud storage systems, ACM Comput. Surv. (CSUR) 49 (4) (2017) 74:4–74:5.
[4] J.R. Douceur, A. Adya, W.J. Bolosky, P. Simon, M. Theimer, Reclaiming space from duplicate files in a serverless distributed file system, in: 22nd International Conference on Distributed Computing Systems, 2002. Proceedings, IEEE, 2002, pp. 617–624.
[5] M.W. Storer, K. Greenan, D.D. Long, E.L. Miller, Secure data deduplication, in: Proceedings of the 4th ACM International Workshop on Storage Security and Survivability, ACM, 2008, pp. 1–10.
[6] F. Rashid, A. Miri, I. Woungang, A secure data deduplication framework for cloud environments, in: 2012 Tenth Annual International Conference on Privacy, Security and Trust (PST), IEEE, 2012, pp. 81–87.
[7] P. Puzio, R. Molva, M. Onen, S. Loureiro, Cloudedup: secure deduplication with encrypted data for cloud storage, in: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom), Vol. 1, IEEE, 2013, pp. 363–370.
[8] R. Chen, Y. Mu, G. Yang, F. Guo, BL-MLE: block-level message-locked encryption for secure large file deduplication, IEEE Trans. Inf. Forensics Secur. 10 (12) (2015) 2643–2652.
[9] J. Li, X. Chen, M. Li, J. Li, P.P. Lee, W. Lou, Secure deduplication with efficient and reliable convergent key management, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1615–1625.
[10] M. Bellare, S. Keelveedhi, T. Ristenpart, Message-locked encryption and secure deduplication, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, Berlin, Heidelberg, 2013, pp. 296–312.
[11] P. Shah, W. So, Lamassu: Storage-efficient host-side encryption, in: USENIX Annual Technical Conference, 2015, pp. 333–345.
[12] M. Bellare, S. Keelveedhi, T. Ristenpart, DupLESS: Server-aided encryption for deduplicated storage, in: IACR Cryptology ePrint Archive, 2013, p. 429, 2013.
[13] G.R. Blakley, C. Meadows, Security of ramp schemes, in: Workshop on the Theory and Application of Cryptographic Techniques, Springer, Berlin, Heidelberg, 1984, pp. 242–268.
[14] J. Li, Y.K. Li, X. Chen, P.P. Lee, W. Lou, A hybrid cloud approach for secure authorized deduplication, IEEE Trans. Parallel Distrib. Syst. 26 (5) (2015) 1206–1216.
[15] P. Puzio, R. Molva, M. Önen, S. Loureiro, PerfectDedup: secure data deduplication, in: Data Privacy Management, and Security Assurance, Springer, Cham, 2015, pp. 150–166.
[16] J. Katz, Y. Lindell, Introduction to Modern Cryptography, Chapman and Hall/CRC, 2014, p. 70.
[17] D. Boneh, V. Shoup, A graduate course in applied cryptography, 2015, p. 47, Draft 0.2.

[18] J. Xu, E.C. Chang, J. Zhou, Weak leakage-resilient client-side deduplication of encrypted data in cloud storage, in: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ACM, 2013, pp. 195–206.
[19] M. Bellare, S. Keelveedhi, Interactive message-locked encryption and secure deduplication, in: IACR International Workshop on Public Key Cryptography, Springer, Berlin, Heidelberg, 2015, pp. 516–538.
[20] S. Canard, F. Laguillaumie, M. Paindavoine, Verifiable message-locked encryption, in: International Conference on Cryptology and Network Security, Springer, Cham, 2016, pp. 299–315.
[21] J. Liu, N. Asokan, B. Pinkas, Secure deduplication of encrypted data without additional independent servers, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 874–885.
[22] Z. Yan, W. Ding, X. Yu, H. Zhu, R.H. Deng, Deduplication on encrypted big data in cloud, IEEE Trans. Big Data 2 (2) (2016) 138–150.
[23] J. Stanek, L. Kencl, Enhanced secure thresholded data deduplication scheme for cloud storage, IEEE Trans. Dependable Secure Comput. (2016).
[24] M. Li, C. Qin, J. Li, P.P. Lee, CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal, IEEE Internet Comput. 20 (3) (2016) 45–53.
[25] Y. Zhao, S.S. Chow, Updatable block-level message-locked encryption, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ACM, 2017, pp. 449–460.
[26] D. Harnik, B. Pinkas, A. Shulman-Peleg, Side channels in cloud services: Deduplication in cloud storage, IEEE Secur. Priv. 8 (6) (2010) 40–47.
[27] J. Liu, L. Duan, Y. Li, N. Asokan, Secure deduplication of encrypted data: Refined model and new constructions, in: Cryptographers' Track at the RSA Conference, Springer, Cham, 2018, pp. 374–393.
[28] J. Dave, S. Saharan, P. Faruki, V. Laxmi, M.S. Gaur, Secure random encryption for deduplicated storage, in: International Conference on Information Systems Security, Springer, Cham, 2017, pp. 164–176.
[29] F. Armknecht, J.M. Bohli, G.O. Karame, F. Youssef, Transparent data deduplication in the cloud, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 886–900.
[30] J. Hur, D. Koo, Y. Shin, K. Kang, Secure data deduplication with dynamic ownership management in cloud storage, IEEE Trans. Knowl. Data Eng. 28 (11) (2016) 3113–3125.
[31] Y. Zhou, D. Feng, W. Xia, M. Fu, F. Huang, Y. Zhang, C. Li, SecDep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management, in: 2015 31st Symposium on Mass Storage Systems and Technologies (MSST), IEEE, 2015, pp. 1–14.
[32] J. Li, J. Li, D. Xie, Z. Cai, Secure auditing and deduplicating data in cloud, IEEE Trans. Comput. 65 (8) (2016) 2386–2396.