Mitigating Traffic-based Side Channel Attacks in Bandwidth-efficient Cloud Storage

Pengfei Zuo, Yu Hua✉, Cong Wang†, Wen Xia, Shunde Cao, Yukun Zhou, Yuanyuan Sun
Wuhan National Laboratory for Optoelectronics
School of Computer, Huazhong University of Science and Technology, Wuhan, China
†City University of Hong Kong, Hong Kong, China
✉Corresponding author: Yu Hua (csyhua@hust.edu.cn)

Abstract—**Data deduplication is able to effectively identify and eliminate redundant data and only maintain a single copy of files and chunks. Hence, it is widely used in distributed storage systems and cloud storage to save the users' network bandwidth for uploading files. However, the occurrence of deduplication can be easily identified by monitoring and analyzing network traffic, which leads to the risk of user privacy leakage. An attacker can carry out a very dangerous side channel attack, i.e., learn-the-remaining-information (LRI) attack, to reveal users' privacy information by exploiting the side channel of network traffic in deduplication. Existing work addresses the LRI attack at the cost of the high bandwidth consumption. In order to address this problem, we propose a simple yet effective scheme, called randomized redundant chunk scheme (RRCS), to significantly mitigate the risk of the LRI attack while maintaining the high bandwidth efficiency of deduplication. The idea behind RRCS is to add randomized redundant chunks to mix up the real deduplication states of files used for the LRI attack, which effectively obfuscates the view of the attacker, who attempts to exploit the side channel of network traffic for the LRI attack. Our security analysis shows that RRCS significantly mitigates the risk of the LRI attack. We have implemented the RRCS prototype and evaluated it by using three real-world datasets. Experimental results demonstrate RRCS significantly outperforms existing work in terms of bandwidth efficiency.**

I. Introduction

According to an International Data Corporation (IDC) report [1], the amount of worldwide digital data created and replicated reaches 4.4 Zettabytes in 2013, while it is expected to exceed 44 Zettabytes in 2020. IDC analysis also shows that nearly 75% data has a copy, which indicates a large amount of data redundancy existing in our digital world. Moreover, Microsoft Research collects the file data from 857 desktop computers with the size of 162TB, and observes that there exist nearly 40% duplicate data in personal data and nearly 68% duplicate data in the shared data among users [2]. The data redundancy causes large consumption of storage capacity and network bandwidth in distributed file and storage systems as well as cloud storage.

In order to save network bandwidth and storage space, data deduplication [3]–[8] identifies data redundancy and maintains a single copy of files or chunks, which has been widely used in cloud storage services [9]–[15]. In general, deduplication may occur either at the source (client) or the target (server). In the source-based deduplication, before uploading files (or chunks), their fingerprints are first uploaded to the server. If the fingerprints exist in the index of the server, the corresponding files will not be uploaded. In the target-based deduplication, files are directly uploaded to the server, and then deduplicated. The former can obtain both bandwidth and storage savings, while the latter only saves storage space. Moreover, duplicates can be detected among the files owned by a single user or cross users. The single-user deduplication only identifies redundant data in a single user. Based on the single-user deduplication, further using the cross-user deduplication can identify more redundant data among users, thus obtaining significant space savings [2]. Hence, current cloud storage systems typically perform cross-user source-based deduplication for higher storage and bandwidth efficiency [12], [13].

Although the cross-user source-based deduplication significantly improves storage and bandwidth utilizations, the occurrence of deduplication can be easily identified by monitoring and analyzing network traffic, resulting in the risk of user privacy leakage. By exploiting the side channel of network traffic in deduplication, the attacker can carry out a very dangerous side channel attack, i.e., learn-the-remaining-information (LRI) attack, to obtain user privacy, which is detailed in Section II-B. Harnik et al. [16] perform tests and find that the LRI attack can occur in the popular cloud storage services such as Dropbox [9] and Mozy [10]. Unfortunately, the LRI attack in deduplication is difficult to be addressed due to the following challenges.

- **The Limitations Using CE or MLE.** To protect data confidentiality in deduplication, convergent encryption (CE) is used to encrypt data [17]. CE proposed by Douceur et al. [18] uses the hash of files to encrypt the files so that the repeated files always generate identical ciphertexts. Thus deduplication can be done over the encrypted data. Bellare et al. [19] formalize CE and its variants as a cryptographic primitive, called message-locked encryption (MLE). However, even if data are encrypted by CE/MLE in cryptography deduplication systems, there still exists the risk of the LRI attack. Because the attacker could always carry out the LRI attack based on the side channel of network traffic to perceive whether deduplication occurs without probing the data themselves transmitted in the network.
- **Deduplication Inefficiency.** There are two baseline solutions to defend against the LRI attack. The first solution is to use encryption to avoid cross-user deduplication. Before uploading files to the cloud server, a client encrypts the files using the users' personal keys, and the duplicate files cross

users will produce different ciphertexts via encryption with different keys. This solution prevents the cross-user deduplication in the server, but substantially increases bandwidth and storage overheads. The second solution is to perform target-based deduplication. Files are directly uploaded to the server and then deduplicated. This solution has no bandwidth saving and only reduces the storage overhead compared with source-based deduplication. Both solutions substantially decrease the deduplication efficiency. Hence, it is nontrivial to defend against the LRI attack while ensuring the deduplication efficiency.

Several schemes have been proposed to defend against the LRI attack. Heen et al. [20] propose a gateway-based deduplication model that uses a gateway (i.e., home router) as the third entity in deduplication systems to improve the resistance to the LRI attack. However, the solution needs an extra gateway provided by the Network Service Provider [20], which is not always possible in practical settings. Harnik et al. [16] propose the randomized threshold solution (RTS) without the need of an extra gateway. However, RTS causes huge bandwidth overhead due to uploading redundant data, and has the risk of leaking privacy with a certain probability.

To address these challenges, this paper proposes a bandwidth-efficient randomized redundant chunk scheme (R-RCS) to mitigate the risk of the LRI attack in cloud storage while maintaining the high bandwidth efficiency of deduplication. By carefully adding randomized chunk-level redundancy for each uploaded file, RRCS can mix up the real deduplication states of files used for the LRI attack, and effectively obfuscate the view of the attacker, who attempts to exploit the side channel of network traffic for the LRI attack. Specifically, the main contributions of this paper include:

- We propose RRCS, a simple yet effective scheme to defend against the LRI attack. In RRCS, when a client uploads the non-duplicate chunks of a file to the server, a small amount of redundant data chunks are also uploaded, which obfuscate the attacker's view on the network traffic. The number of the redundant chunks is chosen at random. The randomness of redundant chunks in RRCS mixes up the real deduplication states of files to defend against the LRI attack.
- We present an in-depth security analysis for RRCS. In the security analysis, we first show that all possible variants of the deduplication detection method are not effective in RRCS, and then demonstrate that RRCS can significantly reduce the risk of the LRI attack.
- We have implemented the RRCS prototype in a deduplication system, and examined the real performance of RRCS by using multiple large-scale real-world datasets, including Fslhomes [21], MacOS [21], and Onefull [22]. Extensive experimental results demonstrate that RRCS has much less bandwidth overhead than RTS [16].

II. Background and Motivation

A. System and Threat Models

We consider a general cloud storage service model that includes two entities, i.e., the user and cloud storage server. In the threat model of the side channel attack, the attack is launched by the users who aim to steal the privacy information of other users [16], [20], [23]. The attacker can act as a user via its own account or use multiple accounts to disguise as multiple users. The cloud storage server communicates with the users through Internet. The connections from the clients to the cloud storage server are encrypted by Secure Socket Layer (SSL) [24] or Transport Layer Security (TLS) protocol [25]. Hence, the attacker can monitor and measure the amount of network traffic between the client and server but cannot intercept and analyze the contents of the transmitted data. The attacker can then perform the sophisticated traffic analysis with sufficient computing resources. For example, the user A is the victim who has uploaded his/her file with privacy information to the cloud storage server. The user B is the attacker who can upload any number of files to the same cloud storage server. During the file uploads, the user B monitors the amount of their network traffic to determine the duplication states of files and then infers the privacy information in the file uploaded by the user A, as the method described in Section II-B.

In summary, this paper mainly focuses on the side channel of traffic information[1], like existing work [16], [20], [23] on side channel attacks. Thus the attacker could only infer/probe the privacy by observing the amount of network traffic between the client and server. The variants of the deduplication detection method are discussed in details in Section IV-A.

B. The LRI Attack in Deduplication

An attacker can easily identify whether deduplication occurs for a file via monitoring and analyzing the network traffic, and further carries out the learn-the-remaining-information (LRI) attack to reveal user privacy information as presented in the following.

The LRI Attack: In the LRI attack, the attacker knows a large part of the target file in the cloud and tries to learn the remaining unknown parts of the target file via uploading all possible versions of the file's content, i.e, m files. As shown in Figure 1, the attacker knows all the contents of the target file X except the sensitive information θ. To learn the sensitive information, the attacker needs to upload m files $(F_1, F_2, ..., F_m)$ with all possible values of θ $(\theta_1, \theta_2, ..., \theta_m)$, respectively. If a file F_k with the value θ_k is deduplicated and other files are not, the attacker knows that the information $\theta = \theta_k$. Note that the attacker knows that, for the m files, only one file is the same as the file X and the remaining $m-1$ files are similar to the file X since only a small part of their contents are different from file X. The different parts of their contents are the sensitive information, such as the PIN [16], the password of bank account [13], and the salary number, which can usually be represented as a small number of bits and easily covered in one-chunk size (about 8kB) in the chunk level.

[1] Note that if the attacker has the ability to control the SSL encryption or memory sniffing, etc., a new kind of attack can be formed, whereby the attacker could potentially obtain the deduplication state of a file. However, such attack is much harder than the side channel of traffic information, and is beyond the scope of the threat models we consider.

The LRI attack can be applied in extensive application scenarios whenever the privacy information is in *a moderately sized domain*, i.e., the number of possible versions of the target file is moderate [16], [20]. We use two examples to show how the LRI attack is used to obtain the other users' private information in practice.

• **Stealing the salary information.** Alice and Bob belong to the same company. Alice knows Bob's employee number and other personal information about Bob. The salary of the company is in the range of 5,000 to 15,000, and a multiple of 1,000. If Alice wants to know Bob's salary, she can backup 11 ($m = 11$) versions of the payroll with Bob's name, Bob's employee number and the salary ranging from 5,000 to 15,000 to the same server in which Bob has backed up his payroll. Thus Alice can know the salary of Bob that is in the payroll version in which the deduplication occurs.

• **Stealing the medical test results.** Bob backups the result of his medical test to a cloud storage service. The medical test results are presented in the standard document templates that are public. Moreover, some medical test results usually come from a small domain, such as, a yes or no result for a pregnancy test, and a hundred likely range for a cholesterol test. Thus Alice can obtain the Bob's test result via backing up a small number of documents to the same cloud storage service that Bob uses.

Note that in the LRI attack, the server cannot catch or sanction the attacker. That is because the attacker uploads his/her own files to the server regardless of the number of uploaded files and thus the behavior of the attacker is the same as that of a normal user [16].

C. Related Work on the LRI Attack

The security issues of cross-user deduplication in cloud storage services have been widely studied, including data confidentiality [17], [19], [26], side channel attacks [16], [20], and the proofs of ownership [27], [28]. Convergent encryption [19] is proposed to ensure the data confidentiality in deduplication systems. However, even with data encryption, deduplication still leaks the sensitive information of users via the LRI attack [16], [20]. Existing work addressing the LRI attack can be divided into two categories.

The first category is based on a special deduplication system model, i.e., gateway-based system model. The model consists of three entities, i.e., the user, the gateway provided by the Network Service Provider, and the storage server. Heen et al. [20] assume that the gateway is installed in the attacker's home network, and propose to use the gateway to mix up the traffic of the cloud storage service with that of other services. Shin el al. [23] assume that the gateway is shared by multiple users, and propose to leverage the gateway to mix up the traffic among multiple users. These solutions avoid the attacker to learn the occurrence of deduplication by monitoring the network traffic of clients, thus improving the resistance to the LRI attack. However, an extra gateway provided by the Network Service Provider is needed, which is not always possible in practical settings.

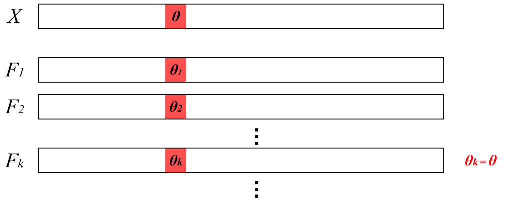


Fig. 1. The LRI attack.

The second category addresses the LRI attack in the general deduplication system model including two entities, i.e., the user and the storage server. The general system model is widely used in current cloud storage systems [9]–[11]. Harnik et al. [16] propose the randomized threshold solution (RTS). For each file X, the server sets a threshold t_X which is chosen uniformly from the range $[2, d]$ at random (d might be a public parameter). The server keeps a counter c_X to count the number of previously uploaded copies of file X. When a new copy of file X is uploaded, RTS checks the counter c_X. If c_X is smaller than t_X, the file is uploaded and deduplicated in the server. Otherwise it is deduplicated in the client. Harnik et al. show that RTS has a risk of privacy leakage with probability $\frac{1}{d-1}$. Because t_X is chosen uniformly at random, when $t_X = 2$, the attacker uploads one copy of file X and can learn that deduplication occurs. Moreover, RTS assigns thresholds to all files which consumes high bandwidth overhead in the practical deduplication as presented in Section V.

Unlike the gateway-based solution [20], [23], RRCS does not require an extra gateway that is not always available. Compared with RTS [16], RRCS can obtain $2 \sim 10$ times higher redundancy elimination ratio as shown in Section V-D, due to exploiting fine-grained redundancy to defend against the LRI attack.

D. Motivation

1) File-level vs. Chunk-level Deduplication: From the identification granularity of the duplicate data, the deduplication is divided into two categories, i.e., file-level and chunk-level deduplication. Specifically, file-level deduplication considers the whole file as a unit to eliminate redundant data. Chunk-level deduplication divides the entire file into chunks (fixed-sized [5] or variable-sized [6], [29]), and then considers the chunk as a unit to eliminate redundant data. Compared with file-level deduplication, the chunk-level deduplication not only identifies the identical files, but also eliminates the identical chunks among the similar files. Consequently, chunk-level deduplication can obtain higher deduplication ratio, and thus has been widely used in backup systems [5], [6], [22] and cloud storage systems [12], [13].

For file-level deduplication, there are two deduplication states for a file in a given storage system, i.e., duplicate and non-duplicate. The client does not upload the duplicate-detected files in the former case. In the latter case, the client needs to upload the non-duplicate files. In the LRI attack, for m files, only the file F_k with correct sensitive information is the same as the target file X and thus not uploaded. Other files $F_i (i \in [1, m] \& i \neq k)$ with incorrect information are uploaded.

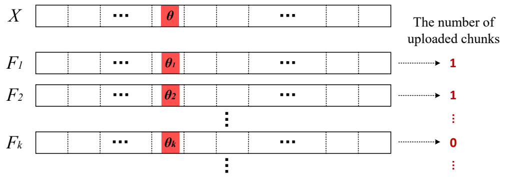


Fig. 2. The LRI attack in chunk-level deduplication.

If we want to mix up the deduplication states of the file F_k and other files to defend against the LRI attack, we need to upload the whole file regardless of whether deduplication occurs, like RTS [16], which incurs high bandwidth overhead.

This paper focuses on defending against the LRI attack in chunk-level deduplication. Chunk-level deduplication deals with duplicate files based on their redundant level. Specifically, there are three deduplication states for a file: (1) Full deduplication (D_{full}). A client uploads a file X_a to the server. If an existing file X_b is completely identical to the file X_a, X_a will be deduplicated without the need of uploading. (2) Partial deduplication (D_{part}). A file X_c in the server is similar (partially identical) to file X_a to be uploaded, meaning that they share some duplicate chunks. The client only uploads the non-duplicate chunks. (3) No deduplication (D_{no}). If no identical/similar files exist in the server, the whole file X_a needs to be uploaded.

As shown in Figure 2, in the LRI attack, for the m files, the file F_k with correct sensitive information is completely identical to the target file X, i.e, D_{full}, whose uploading traffic is zero. Other files have $N-1$ duplicate chunks and one non-duplicate chunk with the value θ_i (as described in Section II-B), belonging to D_{part}, whose uploading traffics are equal to one-chunk size. To defend against the LRI attack, we can explore leveraging chunk-level redundancy rather than the whole-file redundancy, to mix up the deduplication states of the file F_k and other files $F_i (i \in [1, m] \& i \neq k)$ via uploading some redundant chunks in each file.

2) Deterministic Chunk-level Redundancy for Defending against the LRI Attack: For the m files used for the LRI attack, the uploading traffic of the file F_k with correct sensitive information is zero and the uploading traffic of the other $m-1$ files are the size of one chunk. To mix up the m files in terms of the uploading traffic, a simple solution is to add a fixed number of redundant chunks to ensure that the traffic of each file is always more than one-chunk size. Specifically, for a file with non-duplicate chunks, we upload its non-duplicate chunks. For a file without non-duplicate chunks, i.e., the whole file is duplicate, we randomly choose one chunk of the file to upload. Thus one chunk is uploaded for F_k in the solution. Hence, the m files are indistinguishable in terms of uploading traffic, since the traffic of each file is equal to the size of one chunk.

However, in fact, the solution is easily broken. The attacker can append one non-duplicate chunk in each file to break the solution, as shown in Figure 3. The non-duplicate chunk can be randomly generated. Since the average chunk size is about 8 KB, a randomly generated chunk is unlikely to exist in the server since there are $2^{2^{16}}$ possible chunks. By doing so, the traffic of F_k is the size of one chunk and the traffics of other files are the total size of two chunks. Thus F_k with correct sensitive information is easily identified from the m files according to the traffic.

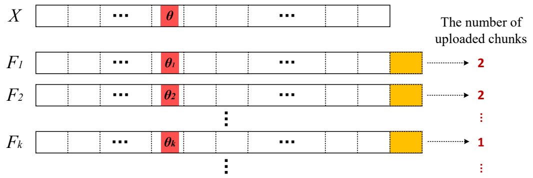


Fig. 3. Appending Chunks Attack. (The appended non-duplicate chunk is marked by yellow in the figure).

To enhance the simple solution, we can add more redundant chunks to ensure that the traffic of each file is always more than the size of l chunks ($1 < l < N$), e.g., $l = N/2$. However, the attacker can also append more than l non-duplicate chunks in each file. The traffic of F_k is the size of one chunk less than the traffics of other files, which breaks the enhanced solution. We name the method that appends one or multiple non-duplicate chunks in each file to assist the LRI attack as Appending Chunks Attack (ACA). In summary, using deterministic chunk-level redundancy fails to mitigate the risk of the LRI attack.

III. Design and Implementation

As mentioned above that using deterministic chunk-level redundancy fails to mitigate the risk of the LRI attack, we present the Randomized Redundant Chunk Scheme (RRCS) which explores and exploits randomized chunk-level redundancy to mitigate the risk of the LRI attack.

A. The Randomized Redundant Chunk Scheme

The idea behind RRCS is to explore and exploit randomized chunk-level redundancy to obfuscate the view of the attacker, who attempts to measure the uploading traffics of files for executing the LRI attack.

In RRCS, the basic idea of adding redundant chunks is to choose the number of the redundant chunks from a range uniformly at random. The redundant chunks are randomly chosen from all the duplicate chunks of the file. By doing so, RRCS can significantly weaken the correlation between the deduplicated and the existing files in the server from the network traffic point of view, and effectively prevent the potential attacker that observes the network traffic from accurately determining whether deduplication occurs.

1) The Overview of RRCS: RRCS determines the uploaded chunks based on the real deduplication states of files via mixing the redundant chunks. Figure 4 shows the framework of RRCS. RRCS includes three key function modules, range generation (RG), secure bounds setting (SBS), security-irrelevant redundancy elimination (SRE). When uploading the random-number redundant chunks, RRCS first uses RG to generate a fixed range in which the random number is chosen. However, the fixed range may cause a security issue. SBS is used to deal with the bounds of the fixed range to avoid the security issue. There may exist security-irrelevant redundant chunks in

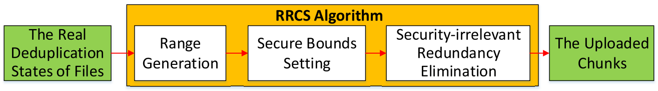

The Real Deduplication States of Files
RRCS Algorithm
Range Generation
Secure Bounds Setting
Security-irrelevant Redundancy Elimination
The Uploaded Chunks

Fig. 4. The framework of the randomized redundant chunk algorithm.

RRCS. SRE reduces the security-irrelevant redundant chunks to improve the deduplication efficiency. The modules are detailed as follows.

2) Range Generation: For each file, RRCS first assigns a range $[0, \lambda N]$ ($\lambda \in (0, 1]$), in which the number of redundant chunks R is chosen uniformly at random. N is the total number of chunks in a file, which the attacker can obtain by chunking the file using the chunking algorithm. λ is a parameter assigned by the deduplication system, which might be public. How to set the parameter λ for the system is a tradeoff between the security and bandwidth efficiency, which we will discuss in Section IV and V. If λN is not an integer, $\lambda N = \lceil \lambda N \rceil$.

Security Analysis for the Range. As described in Section II-B, m files $(F_1, F_2, ..., F_m)$ are used for executing the LRI attack, in which the file F_k has the correct sensitive information. We add $R_i (i = 1, 2, ..., m)$ redundant chunks for file F_i, and R_i is randomly chosen form the range $[0, \lambda N]$. Thus the number of actually uploaded chunks in F_k is in the range $[0, \lambda N]$, due to no non-duplicate chunks. The numbers of actually uploaded chunks in other $m - 1$ files $F_i (i \in [1, m] \& i \neq k)$ are in the range $1 + [0, \lambda N] = [1, \lambda N + 1]$, due to one non-duplicate chunk. Hence, the file F_k and other files have different ranges in terms of the uploading traffic, which is not secure enough for the LRI attack. There are two events causing privacy leakage.

- If R_k for the file F_k happens to be 0 with probability $\frac{1}{\lambda N+1}$, the uploading traffic of F_k is zero. Thus the attacker can easily distinguish F_k from the m files since the uploading traffic of the other $m - 1$ files is always more than zero.
- If $R_i (i \in [1, m] \& i \neq k)$ for all the $m - 1$ files with incorrect sensitive information happen to be λN with probability ${\frac{1}{\lambda N+1}}^{m-1}$, the uploading traffics of all the $m - 1$ files are equal to the size of $\lambda N + 1$ chunks. Thus the attacker can determine the remaining one file is F_k since the uploading traffic of F_k is always no more than the size of λN chunks.

In summary, assigning the same range of the number of the redundant chunks to the m files results in the risk of privacy leakage with probability $\frac{1}{\lambda N+1} + {\frac{1}{\lambda N+1}}^{m-1}$.

3) Secure Bounds Setting: When R happens to be the bound of the fixed range $[0, \lambda N]$, the attacker can identify the file F_k with correct sensitive information, resulting in privacy leakage with a certain probability. In the following, we aim to set the secure bounds to avoid the privacy leakage.

Form the above discussion, we argue that the problem of the bounds can be avoided only when the numbers of actually uploaded chunks in all the m files are in the same range. We show how to avoid the problem below. Since the server can clearly know that each uploaded file is completely identical (D_{full}) or partially identical (D_{part}) to the files in the server, different R ranges can be set for different cases. For example, For the file belonging to D_{full}, R is randomly chosen from $[1, \lambda N + 1]$. For the file belonging to D_{part}, R is randomly chosen from $[0, \lambda N]$. Thus the number of actually uploaded chunks in F_k which belongs to D_{full} is in the range $[1, \lambda N + 1]$. The numbers of actually uploaded chunks in other $m - 1$ files which belongs to D_{part} are also in the range $1 + [0, \lambda N] = [1, \lambda N + 1]$.

TABLE I
NOTATIONS USED IN THE PAPER

Label	Description
N	The total number of chunks in the file
K	The number of non-duplicate chunks after deduplication
R	The number of redundant chunks added by RRCS
R'	The number of redundant chunks after eliminating the security-irrelevant chunks
U	The number of actually uploaded chunks ($= K + R'$)
H	The set which R is randomly chosen from H_{full} in D_{full}, H_{part} in D_{part}

Overall, we denote that R is randomly chosen from the set H_{full} in the case of D_{ful}, and randomly chosen from the set H_{part} in the case of D_{part}. In order to mix up the two deduplication states in the m files used for the LRI attack, it is easy to get the equation:

$$H_{part} + 1 = H_{full} \tag{1}$$

Note that the equation means adding 1 to each element in set H_{part} to form the set H_{full}.

4) Security-irrelevant Redundancy Elimination: For a file with N chunks, due to adding the redundant chunks, the number of uploaded chunks, U, is possibly larger than N. It is not necessary to upload more than N chunks, since the $U - N$ redundant chunks become the security-irrelevant redundant chunks without contributions to the security. We hence upload N chunks by reducing the number of redundant chunks, R, when U is larger than N.

5) The RRCS Algorithm: We summarize the RRCS algorithm in Algorithm 1. First, the server assigns the range $[0, \lambda N]$ as the set H_{part} for a file. RRCS algorithm generates set H_{full} by the Equation 1: $H_{part} + 1 = H_{full}$. The two sets are used for two real deduplication states of files, i.e., D_{part} and D_{full}, respectively. RRCS algorithm then judges which deduplication state the file belongs to by checking the number of its non-duplicate chunks K. $K = 0$ means the file is completely identical to a file in the server. RRCS algorithm further configures the set $H = H_{full}$. Moreover, $0 < K < N$ means the file will be partially identical (similar) to files in the server, and we have the set $H = H_{part}$. Otherwise, $K = N$ means the file has no duplicate chunks in the server, and we have the set $H = \{0\}$. The number of redundant chunks R is randomly chosen from the set H. If $R + K > N$, RRCS algorithm sets $R' = N - K$. Otherwise, $R' = R$. Finally, RRCS algorithm generates R' redundant chunks by choosing from the duplicate chunks.

Algorithm 1 The RRCS Algorithm

```
Input: The system parameter $\lambda$; the total number of chunks in a
    file, $N$; and the number of non-duplicate chunks in the file, $K$;
Output: The chunks which need to be uploaded for the file;
$\lambda N = \lceil \lambda N \rceil$;
$H_{part} = [0, \lambda N]$;
$H_{full} = H_{part} + 1 = [1, \lambda N + 1]$;
if ($K == 0$) then
    $H = H_{full}$;
else if ($0 < K < N$) then
    $H = H_{part}$;
else
    $H = \{0\}$;
end if
$R$ is randomly chosen from the set $H$;
if ($R + K > N$) then
    $R' = N - K$;
else
    $R' = R$;
end if
Generate $R'$ redundant chunks by choosing from the duplicate
    chunks;
```

From the RRCS algorithm, we can see that the number of the chunks which need to be uploaded $U(= K + R')$ meets $1 \leq U \leq N$. For a special case that a file only has one chunk, i.e., $N = 1$, the file is directly uploaded in RRCS algorithm.

B. Implementation

In the subsection, we present how to implement RRCS in the chunk-level deduplication system.

As shown in Figure 5, in chunk-level deduplication, the real deduplication states of files include full deduplication (D_{full}), partial deduplication (D_{part}), and no deduplication (D_{no}) (described in Section II-D). D_{full} consists of two cases, i.e., single-user duplicate files and cross-user duplicate files. The single-user duplicate file means that a file uploaded by a user is identical to the file previously uploaded by the user, and thus observing the occurrence of D_{full} for the single-user duplicate file does not cause privacy leakage, as demonstrated in [16]. Hence, RRCS directly deduplicates the single-user duplicate files in the client to obtain the bandwidth savings. The cross-user duplicate file means that a file uploaded by a user is identical to the file previously uploaded by other users. Observing the occurrence of D_{full} for the cross-user duplicate file can be used to reveal other users' privacy. Hence, RRCS mixes up the case with D_{part} using the RRCS algorithm. We directly upload the files occurring in D_{no}.

The RRCS algorithm is implemented in the server. For a file to be uploaded, the client first divides the file into chunks using fixed-sized [5] or variable-sized [6], [29] chunking algorithms and then uploads the fingerprints of all chunks to the server. After receiving the fingerprints, the server can know the deduplication state of the file via querying the fingerprint index. The server employs the RRCS algorithm to determine the chunks needing to be uploaded which include the non-duplicate chunks and mixed redundant chunks, and then responds to the client. The client finally uploads these chunks but cannot distinguish the redundant and non-duplicate chunks.

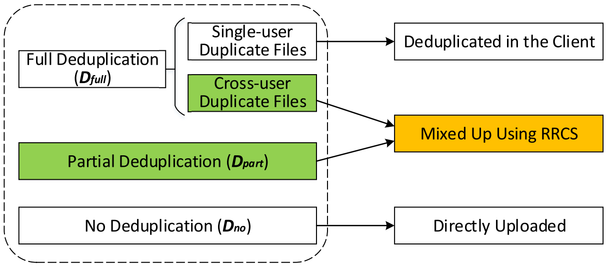

Full Deduplication (D_{full})
Single-user Duplicate Files
Cross-user Duplicate Files
Partial Deduplication (D_{part})
No Deduplication (D_{no})
Deduplicated in the Client
Mixed Up Using RRCS
Directly Uploaded

Fig. 5. The implementation of RRCS in deduplication systems.

IV. Security Analysis

In this section, we first discuss all variants of the deduplication detection method and analyze whether the variants are effective in RRCS. We then analyze the security properties of RRCS for the LRI attack.

A. The Variants of the Deduplication Detection

In order to comprehensively evaluate the solutions in resisting the LRI attack, we first elaborate below the baseline deduplication detection method from the attacker and its possible variants. As shown in Section II-A, the attacker's goal is to exploit/identify the occurrence of deduplication to launch the LRI attack. To launch the attack, the attacker can pass the file to the client to upload to the deduplication server. By measuring the uploading traffic, i.e., the side channel, the attacker could attempt to infer/probe the occurrence of deduplication. There are several variants of the above detection method, but we show below that those variants can all be reduced to the above baseline detection method. Thus, later in the next subsection we will only focus on the defense of above baseline case.

The variants include: 1) The attacker might upload the same file multiple times. However, only the first upload could be deemed useful for the attacker. This is because the file will be stored in the server after its first uploaded (regardless of whether there was an old copy of the file or not in the server), and thus all the subsequent upload of the same file will be always identical to the attacker's own file. Such reasoning could be extended to the case where the attacker might use multiple accounts to disguise as multiple users to upload the same file. 2) The attacker can also try to upload a file to the server and then immediately delete the file. By repeating the operations of uploading and deleting the file, in theory the attacker can perform multiple uploadings. However, this is not feasible in practice. As pointed out by Harnik et al. [16], many online storage services, such as DropBox, Memopal and Mozy, need to keep copies of the deleted files for a period of at least 30 days, either for the purpose of storage resilience or version recovery. Users hence can restore to past versions. Therefore, the execution of each iteration of the attack has to last at least 30 days. The need of long term execution and the fact that the target file status in cloud could be easily changed during the long period due to normal application requests would render such attack practically useless to the attacker. Again, only the first uploaded file is useful for the attacker in RRCS.

B. Security Strength of RRCS

We analyze the security of RRCS for the LRI attack in the general case. We then analyze the security of RRCS for the LRI attack assisted by the Appending Chunks Attack (presented in Section II-D2).

1) The LRI Attack in the General Case: For the LRI attack, the attacker knows a big part of the targeted file X and tries to determine the remaining unknown parts of file X via uploading all possible versions of the file's content. All possible versions are m files in which only one file is the same as file X and the remaining $m-1$ files are similar to file X since only a small part of their contents are different from file X, as the background described in Section II-B. The sizes of different contents are smaller than that of one data chunk. The attacker could observe the client's upload of m similar files F_i $(i = 1, 2, ..., m)$ via chunk-level deduplication and measure the uploading traffic.

In general, by observing the results of measuring the uploading traffic, the attacker can find that the uploading traffic of one file F_k is zero, and the uploading traffics of other files are equal to the size of one chunk. The attacker hence confirms the content of the file F_k is the same as the target file X.

In order to prove that RRCS can address the LRI attack in the general case, we demonstrate in *Theorem 1* that m files should be indistinguishable in RRCS.

Theorem 1: In the general case, the m files used for the LRI attack are indistinguishable from the attacker's view in RRCS.

Proof 1: Initially, the target file X exists in the server. m files $(F_1, F_2, ..., F_m)$ are uploaded for the LRI attack, in which file F_k is the same as file X. Due to adding randomized redundant chunks in RRCS, the uploading traffic of file F_k is equal to the size of R_k chunks. The uploading traffics of the other $m-1$ files F_i $(i \in [1, m], i \neq k)$ are equal to the size of $1 + R_i$ $(i \in [1, m], i \neq k)$ chunks. Since F_k belongs to D_{full} and the other $m-1$ files belong to D_{part}, we have that R_k is randomly chosen from the set H_{full}, and R_i $(i \in [1, m], i \neq k)$ are randomly chosen from H_{part}, as shown in Section III-A3. We thus obtain $R_k \in H_{full}$ and $1 + R_i \in 1 + H_{part}$[2] $(i \in [1, m], i \neq k)$. According to Equation 1, we have $H_{full} = 1 + H_{part}$. Hence, the identical file F_k and other $m-1$ similar files have the same range of uploading traffic, from the attacker's view. Hence, the attacker cannot distinguish between the identical file F_k and the other $m-1$ files F_i $(i \in [1, m], i \neq k)$.

In summary, RRCS defends against the LRI attack by making the m files used for executing the LRI attack indistinguishable from the attacker's view in the general case.

2) The LRI Attack Assisted by the ACA: To execute the Appending Chunks Attack (ACA), the attacker can append one or multiple non-duplicate chunks to each file in the m files used for the LRI attack. In the following, we analyze the security of RRCS for the LRI attack assisted by the ACA.

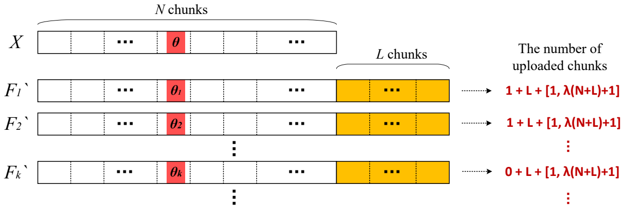


Fig. 6. RRCS under the Appending Chunks Attack. (The appended non-duplicate chunks are marked by yellow in the figure).

Initially, the target file X exists in the server. m files $(F_1, F_2, ..., F_m)$ are uploaded for the LRI attack, in which each file has N chunks and the file F_k is the same as file X. By executing the ACA, $L (L \geq 1)$ non-duplicate chunks are appended to each file. We denote the m files appended non-duplicate chunks as $F'_1, F'_2, ..., F'_m$, which have $N+L$ chunks. Due to being appended by non-duplicate chunks, all the m new files belong to D_{part}, in which $R_i (i \in [1, m])$ are randomly chosen from the range $[1, \lambda N + 1]$ in RRCS. Thus the range of the number of actually uploaded chunks in the file F'_k is $0 + L + [1, \lambda N + 1] = [L+1, \lambda N + L + 1]$, and the ranges in the other $m-1$ files are $1 + L + [1, \lambda N + 1] = [L+2, \lambda N + L + 2]$, as shown in Figure 6. The file F'_k and other $m-1$ files have different ranges in terms of the uploading traffic.

To analyze security, we demonstrate in *Theorem 2* that RRCS leaks no information with high probability for ACA.

Theorem 2: For the LRI attack assisted by the Appending Chunks Attack, RRCS leaks no information which prevents the attacker from accurately identifying the file with the correct sensitive information from the m files, with the probability of $1 - \frac{1}{\lambda(N+L)+1} - \frac{1}{\lambda(N+L)+1}^{m-1}$.

Proof 2: We consider all four events in RRCS where the attacker wants to identify F'_k with correct sensitive information from the m files appended by non-duplicate chunks.

1) The attacker uploads the m files. If observing the uploading traffic of one file is equal to the size of $L+1$ chunks, the attacker can determine that the file is F'_k, since $L+1$ only belongs to the range of the number of actually uploaded chunks in F'_k.
2) If the traffics of $m-1$ files are equal to the size of $\lambda(N+L) + L + 2$ chunks [3], the attacker can determine that the remaining one file is F'_k, since $\lambda(N+L) + L + 2$ only belongs to the ranges of the number of actually uploaded chunks in the $m-1$ files with incorrect sensitive information.
3) If the traffics of all m files are between the sizes of $L+2$ and $\lambda(N+L) + L + 1$ chunks, the attacker fails to determine which file is F'_k. This is because the traffics of all m files can cover the range of $[L+2, \lambda(N+L) + L + 1]$ chunks size. The m files are indistinguishable from the attacker's view in RRCS, based on the proof in Section IV-B1.
4) If the traffics of n files are the size of $\lambda(N+L) + L + 2$ chunks and the traffics of the remaining $m-n$ files are between the sizes of $L+2$ chunks and $\lambda(N+L) +$

[2] $1 + H_{part}$ means adding 1 to each element in set H_{part}.

[3] If $\lambda(N+L)$ is not an integer, $\lambda(N+L) = \lceil \lambda(N+L) \rceil$.

$L + 1$ chunks, the attacker can determine that F'_k is not in the n files but still cannot identify F'_k from the remaining $m-n$ files. Thus the $m-n$ files containing F_k are indistinguishable from the attacker's view in RRCS, based on the proof in Section IV-B1.

The first event that leaks information, occurs with probability $\frac{1}{\lambda(N+L)+1}$.[4] The second event leaking information occurs with probability $\frac{1}{\lambda(N+L)+1}^{m-1}$. Whereas the third and fourth events, which do not leak information, occur with probability $1 - \frac{1}{\lambda(N+L)+1} - \frac{1}{\lambda(N+L)+1}^{m-1}$.

Remark. How to set λ for the server is a tradeoff between the security and bandwidth efficiency. The larger λ is, the higher the probability of leaking no information is. But larger λ also leads to larger range of R, which would naturally result in more potential bandwidth overhead. Nevertheless, even when $\lambda = 1$, RRCS provides the best security guarantee while can also obtain good bandwidth efficiency as demonstrated in Section V-D.

V. Performance Evaluation

A. Setup and Datasets

To evaluate the performance of RRCS, we implement a prototype of cross-user source-based deduplication with RRCS in the distributed cloud system. Each client is equipped with the Ubuntu 12.04 operating system running on a quad-core Intel Core i5-4460 CPU at 3.20 GHz, with a 16GB RAM and a 2TB hard disk. Each server has a 16-core CPU, a 32GB RAM and a 10TB hard disk. The RRCS prototype is written in C language in a Linux environment.

We examine the performance of RRCS using three real-world trace-based datasets, i.e., Fslhomes [21], MacOS [21], and Onefull [22]. We explore the characteristics of the datasets in Section V-B and summarize them in Table II.

- Fslhomes was collected in the File system and Storage Lab (FSL) at Stony Brook University, which contains the snapshots of students' home directories from a shared network file system. The files contain virtual machine images, office documents, source code, binaries and other miscellaneous files.
- MacOS was collected from a MacOS X Enterprise Server that holds 247 users and provides multiple services: email, webservers, calendar server, mailman for mailing lists, wiki server, mySQL, and a trouble-ticketing server.
- Onefull is a subset of the trace reported by Xia et al. [22], which was collected from the personal computers of 15 graduate students in our research group.

As described in Section III-B, single-user duplicate files do not cause privacy leakage. We eliminate single-user duplicate files in the source (client), which obtains significant bandwidth savings in RRCS and RTS. RRCS and RTS hence exhibit the same bandwidth efficiency, i.e., no bandwidth overhead, in eliminating the single-user redundancy. On the other hand, for cross-user deduplication, RRCS and RTS add different-granularity redundancy (i.e., chunk and file) for defending against the side channel attacks. Therefore, we examine the performance of eliminating the cross-user redundancy in RRCS and RTS. In the performance evaluation, we eliminate single-user duplicate files in the client and evaluate the bandwidth efficiency of cross-user deduplication as shown in Section V-D.

TABLE II
THE CHARACTERISTICS OF DATASETS

	Fslhome	MacOS	Onefull
Total size	5.1TB	1.9TB	219GB
Avg. chunk size	8kB	8kB	10kB
Avg. file size	1530kB	683kB	622kB
Cross-user redundancy ratio	39%	48%	25%
The total number of files	3.663M	3.058M	378K
The number of unique files	2.238M	1.600M	283K
The number of > 3 copies unique files	0.316M (8.4%)	0.281M (7.4%)	7.8K (2.8%)
The number of > 5 copies unique files	0.068M (4.8%)	0.011M (0.7%)	2.0K (0.7%)
The number of > 10 copies unique files	0.017M (0.9%)	0.003M (0.2%)	0 (0)

B. The Characteristics of the Datasets

Before evaluating the performance of RRCS, we explore and analyze the characteristics of cross-user file redundancy in the three real-world datasets owning many users. We count the number of the files that have k copies ($k = 1, 2, 3, ...$), while k is the number of users sharing the file.

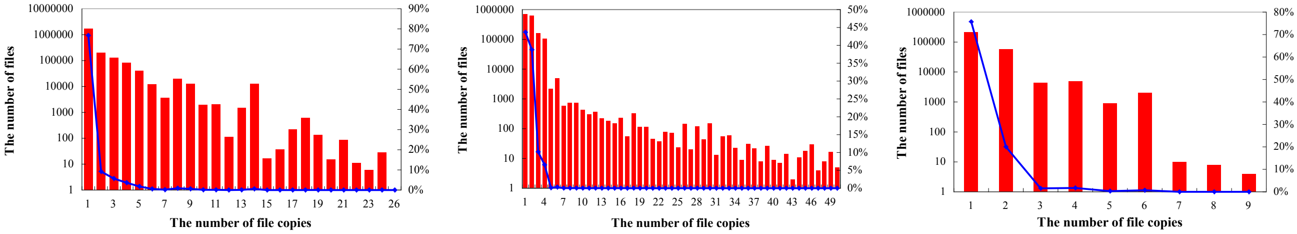
The relationships between the number of files and their copies are shown in Figure 7. The number of files exponentially decreases as a function of the number of file copies. *We can observe that most files only have a few copies (i.e., shared by a few users).* We summarize the results in Table II (M is 10^6, and K is 10^3 in the Table). For Fslhomes dataset, the number of unique files containing more than 5 copies only accounts for 4.8% of the total number of the unique files. For MacOS dataset, the number of unique files containing more than 5 copies only accounts for 0.7% of the total number of unique files. We also investigate the redundancy characteristics in chunk-level which show the similar results.

As a result, most files only have a few copies (or shared by a few users) in the real-world datasets. RTS [16] performs target-based deduplication when the number of file copies is smaller than a pre-defined threshold (detailed in Section II-C). However, since the files having a few copies account for a significant proportion as shown in Figure 7, most files are performed target-based deduplication in RTS. Therefore, RTS becomes bandwidth-inefficient in the real-world datasets.

C. Uploading a Single File Multiple Times

We mainly consider five deduplication schemes, including source-based deduplication, target-based deduplication, file-level RTS, chunk-level RTS, and RRCS. Based on the file-level RTS described in Section II-C, we develop the chunk-level RTS for comparisons, in which a random threshold T is set for each chunk. The five deduplication schemes have the same

[4] Note that since the average size of personal files is over 600kB in the real-world datasets as shown in Table II and thus the average number of chunks N is large enough ($N > 600kB/10kB = 60$), the probability of leaking information $\frac{1}{\lambda(N+L)+1}$ is very small.



(a) Fslhomes

(b) MacOS

(c) Onefull

Fig. 7. The characteristics of datasets. (The blue lines show the percentage of the number of files having k copies in the total number of files.)

space savings in the storage server, but different bandwidth savings (i.e., the reduced amount of the transmitted data by the above five deduplication schemes).

In order to intuitively compare the characteristic of the five deduplication schemes in bandwidth overhead, we first consider a simple situation that the same file is uploaded multiple times by different users. We use an 800kB-size file, which is divided into 100 chunks with the average chunk size of 8kB. We upload the file k times and observe the changes of the total amount of the transmitted data among the above mentioned four schemes. Specifically, file-level (chunk-level) RTS uses the target-based deduplication when the number of the uploaded copies of the file (chunk) is smaller than the threshold T that is chosen uniformly from the range $[2, d]$. We use the parameter setting in their paper [16], i.e., $d = 20$. RRCS needs to upload the randomized redundant chunks for defending against the LRI attack.

Figure 8 shows the changes of the total amount of the transmitted data (i.e., the total traffic) with the increase of the file upload number k. For the target-based deduplication, the total traffic of uploading file k times is equal to k times the size of the file. For file-level RTS, the total traffic is equal to k times the size of the file when k is smaller than the threshold T, and the file is deduplicated in the client when k is larger than T. $T = 11$ in the Figure 8, which is selected by the average value in the range $[2, 20]$. Other cases that the T is set to other numbers are easy to understand. For chunk-level RTS, the total traffic increases slower than that of file-level RTS. When the number of file uploads is high (i.e., 17), file-level and chunk-level RTS have the near-same total traffic, since setting a threshold to a file has the same expectation of the total traffic as setting a threshold to each chunk in the file. For RRCS, the total traffic grows slowly due to adding chunk-level redundancy, and the curve shows a fluctuation since the number of redundant chunks is at random. Compared with RTS, when the file uploading times k is quite large (more than 42 in Figure 8), the total traffic in RRCS may be more than RTS. However, we argue that the files containing many copies are very few in the real-world datasets as shown in Section V-B. Thus RRCS can obtain significant bandwidth saving compared with RTS in the real-world datasets, as we demonstrate in the next subsection.

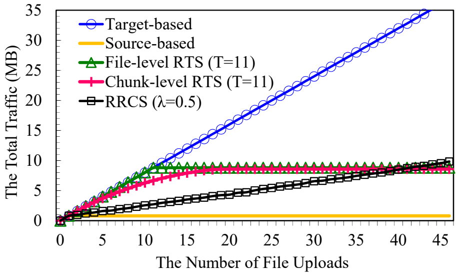


Fig. 8. The total traffic with the increase of the number of file uploads.

D. Bandwidth Overhead

We compare these deduplication schemes in terms of bandwidth overhead in cross-user deduplication, using the three real-world datasets mentioned above. Specifically, in file-level (chunk-level) RTS, we also use the range $[2, 20]$ in which the threshold of each file (chunk) is uniformly chosen at random, as the parameter setting in their paper [16]. In RRCS, we respectively set the system parameter $\lambda = 0.5$ and $\lambda = 1$ to show how the different λ impacts the bandwidth efficiency.

Figure 9 shows the normalized bandwidth overhead of five schemes. The bandwidth overhead of target-based deduplication is equal to the total file size. Compared with target-based deduplication, source-based deduplication reduces $25\% - 48\%$ of bandwidth overhead in the three datasets, due to eliminating all redundancy in the client. File-level (chunk-level) RTS reduce $3.2\% - 6.6\%$ ($4.6\% - 7.9\%$) of bandwidth overhead, due to only obtaining the bandwidth saving of the files (chunks) that have many copies. In fact, these files (chunks) having many copies are quite few as discussed in Section V-B. RRCS with $\lambda = 0.5$ reduces $20.0\% - 32.3\%$ of bandwidth overhead and RRCS with $\lambda = 1$ reduces $13.4\% - 23.0\%$ of bandwidth overhead. We observe that with the increase of λ, the bandwidth overhead of RRCS increases, since larger λ provides better security guarantee while consuming more bandwidth overhead, as discussed in Section IV-B2. Other cases that the λ is set to other numbers are easy to understand. Even though in the worst case where $\lambda = 1$ in terms of bandwidth overhead, RRCS still consumes much less bandwidth overhead than RTS.

Figure 10 shows the redundancy elimination (RE) ratios of the five schemes. RE ratio is defined as the ratio of the size of eliminated redundancy data to that of all redundancy data. Source-based deduplication eliminates 100% data redundancy which however has no security guarantee. File-level (chunk-level) RTS only eliminates $8.1\% - 16.8\%$ ($9.8\% - 20.3\%$) of redundancy, due to only eliminating the redundancy of the files (chunks) that have many copies. RRCS with $\lambda = 0.5$ eliminates $76.1\% - 78.0\%$ of redundancy and RRCS with $\lambda =$

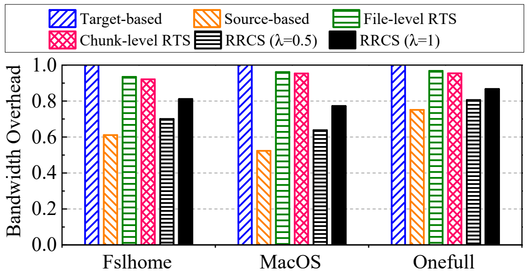


Fig. 9. Normalized bandwidth overhead. (The Y-axis represents the ratio of the transmission bandwidth overhead to the total file size.)

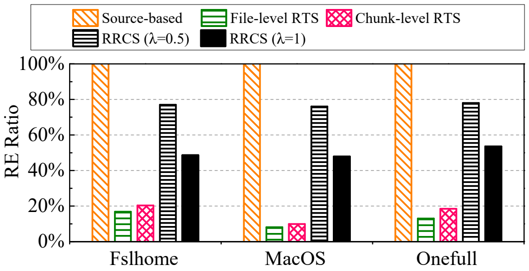


Fig. 10. Redundancy elimination (RE) ratio.

1 eliminates $47.9\% - 53.6\%$ of redundancy. Compared with RTS, RRCS can eliminate 2 to 10 times data redundancy.

From Figure 10, we also observe that RRCS achieves approximately the same RE ratios for the three datasets with the same λ, even though these datasets have different total sizes as shown in Table II. This is because RRCS adds the per-file redundancy to defend against the LRI attack and the amount of added redundancy is only related to λ and independent with the number of files in a dataset. Therefore, RRCS demonstrates good scalability in terms of dataset sizes.

VI. Conclusion

This paper proposes a simple yet effective scheme called RRCS to address an important security issue that deduplication can be exploited to carry out the LRI attack to steal user privacy in cloud storage services. RRCS mixes up the real deduplication states of files used for the LRI attack by adding the randomized redundant chunks, which prevents the attacker from accurately identifying the file with correct sensitive information and thus significantly mitigates the risk of the LRI attack. RRCS also allows the system to control the tradeoff/balance between the security and bandwidth efficiency by a configurable parameter λ. A larger λ results in higher security but lower deduplication efficiency. When $\lambda = 1$, RRCS provides the optimal security guarantee while also obtains a relatively high redundancy elimination ratio, i.e., about 50%. Based on the real RRCS prototype, experimental results from using three real-world datasets demonstrate that RRCS has much less bandwidth overhead than RTS.

Acknowledgement

This work is supported by National Key Research and Development Program of China under Grant 2016YFB1000202, National Natural Science Foundation of China (NSFC) under Grant No. 61772212 and No. 61502190.

References

[1] V. Turner, J. F. Gantz, R. David, and M. Stephen, “The digital universe of opportunities: Rich data and the increasing value of the internet of things,” *IDC iView: IDC Analyze the Future*, 2014.

[2] D. T. Meyer and W. J. Bolosky, “A study of practical deduplication,” *Proc. USENIX FAST*, 2011.

[3] T. Yang, H. Jiang, D. Feng, and Z. Niu, “DEBAR: A scalable high-performance de-duplication storage system for backup and archiving,” in *Proc. IPDPS*, 2010.

[4] L. Xu, A. Pavlo, S. Sengupta, J. Li, and G. R. Ganger, “Reducing replication bandwidth for distributed document databases,” in *Proc. SoCC*, 2015.

[5] S. Quinlan and S. Dorward, “Venti: A new approach to archival storage,” *Proc. USENIX FAST*, 2002.

[6] B. Zhu, K. Li, and R. H. Patterson, “Avoiding the disk bottleneck in the data domain deduplication file system.” in *Proc. FAST*, 2008.

[7] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, “Design tradeoffs for data deduplication performance in backup workloads,” in *Proc. USENIX FAST*, 2015.

[8] B. Mao, H. Jiang, S. Wu, and L. Tian, “POD: Performance Oriented I/O Deduplication for Primary Storage Systems in the Cloud,” in *Proc. IPDPS*, 2014.

[9] “Dropbox: Dropbox-simplify your life,” https://www.dropbox.com/.

[10] “Mozy: Award-winning cloud backup, sync, and mobile access for computers and servers,” http://mozy.com/.

[11] “Spideroak: Zero-knowledge data backup, sync, access, storage and share from any device,” https://spideroak.com/.

[12] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl, “Dark clouds on the horizon: Using cloud storage as attack vector and online slack space.” in *USENIX Security Symposium*, 2011.

[13] P. Puzio, R. Molva, M. Onen, and S. Loureiro, “Cloudedup: secure deduplication with encrypted data for cloud storage,” *CloudCom*, 2013.

[14] D. Frey, A.-M. Kermarrec, and K. Kloudas, “Probabilistic deduplication for cluster-based storage systems,” in *Proc. SoCC*, 2012.

[15] P. Zuo, Y. Hua, X. Liu, D. Feng, W. Xia, S. Cao, J. Wu, Y. Sun, and Y. Guo, “BEES: Bandwidth-and Energy-Efficient Image Sharing for Real-Time Situation Awareness,” in *Proc. ICDCS*, 2017.

[16] D. Harnik, B. Pinkas, and A. Shulman-Peleg, “Side channels in cloud services: Deduplication in cloud storage,” *IEEE Security & Privacy*, vol. 8, no. 6, pp. 40–47, 2010.

[17] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, “Secure data deduplication,” *Proc. ACM StorageSS*, 2008.

[18] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer, “Reclaiming space from duplicate files in a serverless distributed file system,” *Proc. IEEE ICDCS*, 2002.

[19] M. Bellare, S. Keelveedhi, and T. Ristenpart, “Message-locked encryption and secure deduplication,” *Proc. Springer EUROCRYPT*, 2013.

[20] O. Heen, C. Neumann, L. Montalvo, and S. Defrance, “Improving the resistance to side-channel attacks on cloud storage services,” *Proc. IEEE NTMS*, 2012.

[21] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok, “Generating realistic datasets for deduplication analysis,” *Proc. USENIX ATC*, 2012.

[22] W. Xia, H. Jiang, D. Feng, and Y. Hua, “Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput,” *Proc. USENIX ATC*, 2011.

[23] Y. Shin and K. Kim, “Differentially private client-side data deduplication protocol for cloud storage services,” *Security and Communication Networks*, vol. 8, no. 12, pp. 2114–2123, 2015.

[24] “The openssl program,” http://www.openssl.org/.

[25] T. Dierks, “The transport layer security (tls) protocol version 1.2,” 2008.

[26] J. Li, X. Chen, M. Li, J. Li, P. P. Lee, and W. Lou, “Secure deduplication with efficient and reliable convergent key management,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1615–1625, 2014.

[27] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, “Proofs of ownership in remote storage systems,” *Proc. ACM CCS*, 2011.

[28] R. Di Pietro and A. Sorniotti, “Boosting efficiency and security in proof of ownership for deduplication,” in *Proc. ASIACCS*, 2012.

[29] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang, “Fastcdc: a fast and efficient content-defined chunking approach for data deduplication,” in *Proc. USENIX ATC*, 2016.