PerfectDedup: Secure Data Deduplication

Pasquale Puzio[1,2], Refik Molva[2], Melek Önen[2], Sergio Loureiro[1]

[1] SecludIT, Sophia Antipolis, FRANCE
{pasquale,sergio}@secludit.com,
http://secludit.com
[2] EURECOM, Sophia Antipolis, FRANCE
{puzio,molva,onen}@eurecom.fr,
http://www.eurecom.fr

Abstract. With the continuous increase of cloud storage adopters, data deduplication has become a necessity for cloud providers. By storing a unique copy of duplicate data, cloud providers greatly reduce their storage and data transfer costs. Unfortunately, deduplication introduces a number of new security challenges. We propose PerfectDedup, a novel scheme for secure data deduplication, which takes into account the popularity of the data segments and leverages the properties of Perfect Hashing in order to assure block-level deduplication and data confidentiality at the same time. We show that the client-side overhead is minimal and the main computational load is outsourced to the cloud storage provider.

Keywords: cloud,storage,deduplication,confidentiality,encryption,security, perfect hashing

1 Introduction

Cloud storage providers constantly look for techniques aimed to minimize redundant data and maximize space savings. We focus on deduplication, which is one of the most popular techniques and has been adopted by many major providers such as Dropbox[3]. The idea behind deduplication is to store duplicate data only once. Thanks to such a mechanism, space savings can reach 70% [7] and even more in backup applications. On the other hand, along with low costs, users also require the confidentiality of their data through encryption. Unfortunately, deduplication and encryption are two conflicting techniques. A solution which has been proposed to meet these two conflicting requirements is Convergent Encryption (CE) [4] whereby the encryption key is the result of the hash of the data segment. However, CE unfortunately suffers from various

[2] Partially supported by the TREDISEC project (G.A. no 644412), funded by the European Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme.
[3] https://www.dropbox.com

well-known weaknesses [9] including dictionary attacks. We propose to counter the weaknesses due to CE by taking into account the popularity [10] of the data segments. Data segments stored by several users, that is, popular ones, are only protected under the weak CE mechanism whereas unpopular data segments that are unique in storage are protected under semantically-secure encryption. This declination of encryption mechanisms lends itself perfectly to efficient deduplication since popular data segments that are encrypted under CE are also the ones that need to be deduplicated. This scheme also assures proper security of stored data since sensitive thus unpopular data segments enjoy the strong protection thanks to the semantically-secure encryption whereas the popular data segments do not actually suffer from the weaknesses of CE since the former are much less sensitive because they are shared by several users. Nevertheless, this approach raises a new challenge: the users need to decide about the popularity of each data segment before storing it and the mechanism through which the decision is taken paves the way for a series of exposures very similar to the ones with CE. The focus of schemes based on popularity then becomes the design of a secure mechanism to detect the popularity of data segments.

In this paper we suggest a new scheme for the secure deduplication of encrypted data, based on the aforementioned popularity principle. The main building block of this scheme is an original mechanism for detecting the popularity of data segments in a perfectly secure way. Users can lookup for data segments in a list of popular segments stored by the Cloud Storage Provider (CSP) based on data segment identifiers computed with a Perfect Hash Function (PHF). Thanks to this technique, there is no information leakage about unpopular data segments and popular data segments are very efficiently identified. Based on this new popularity detection technique, our scheme achieves deduplication of encrypted data at block level in a perfectly secure manner. The advantages of our scheme can be summarized as follows:

- our scheme allows for storage size reduction by deduplication of popular data;
- our scheme relies on symmetric encryption algorithms, which are known to be very efficient even when dealing with large data;
- our scheme achieves deduplication at the level of blocks, which leads to higher storage space savings compared to file-level deduplication [7];
- our scheme does not require any coordination or initialization among users;
- our scheme does not incur any storage overhead for unpopular data blocks;

2 Secure Deduplication Based on Popularity

Given the inherent incompatibility between encryption and deduplication, existing solutions suffer from different drawbacks. CE was considered to be the most convenient solution for secure deduplication but it has been proved that is is vulnerable to various types of attacks [9]. Hence, CE cannot be employed to protect data confidentiality and thus stronger encryption mechanisms are required.

We point out that data may need different levels of protection depending on its popularity [10] a data segment becomes "popular" whenever it belongs to more than t users (where t is the popularity threshold). The "popularity" of a block is viewed as a trigger for its deduplication. Similarly, a data segment is considered to be unpopular if it belongs to less than t users. This is the case for all highly sensitive data, which are likely to be unique and thus unlikely to be duplicated.

Given this simple distinction, we observe that popular data do not require the same level of protection as unpopular data and therefore propose different forms of encryption for popular and unpopular data. For instance, if a file is easily accessible by anyone on the Internet, then it is reasonable to consider a less secure protection. On the other hand, a confidential file containing sensitive information, such as a list of usernames and passwords, needs much stronger protection. Popular data can be protected with CE in order to enable source-based deduplication, whereas unpopular data must be protected with a stronger encryption. Whenever an unpopular data segment becomes popular, that is, the threshold t is reached, the encrypted data segment is converted to its convergent encrypted form in order to enable deduplication.

We propose to encrypt unique and thus unpopular data blocks (which cannot be deduplicated) with a symmetric encryption scheme using a random key, which provides the highest level of protection while improving the computational cost at the client. Whenever a client wishes to upload a data segment, we propose that she should first discover its popularity degree in order to perform the appropriate encryption operation. The client may first lookup for a convergent encrypted version of the data stored at the CSP. If such data segment already exists, then the client discovers that this data segment is popular and hence can be deduplicated. If such data segment does not exist, the client will encrypt it with a symmetric encryption scheme. Such a solution would greatly optimize the encryption cost and the upload cost at the client. However, a standard lookup solution for the convergent encrypted data segment would reveal the convergent encrypted data segment ID, that is the digest of the data computed under an unkeyed hash function like SHA-3, which would be a serious breach. Secure lookup for a data segment is thus a delicate problem since the ID used as the input to the lookup query can lead to severe data leakage as explained in [17] and [9]. Therefore, in such a scenario the main challenge becomes how to enable the client to securely determine the popularity of a data segment without leaking any exploitable information to the CSP. Also, the client needs to securely handle the "popularity transition", that is the phase triggered by a data segment that has just reached the popularity threshold t. More formally, the popularity detection problem can be defined as follows: given a data segment D and its ID ID_D, the client wants to determine whether ID_D belongs to the set P of popular data segment IDs stored at an untrusted CSP. It is crucial that if $ID_D \notin P$, no information must be leaked to the CSP. More generally, this problem can be seen as an instance of the Private Set Intersection (PSI) problem [26]. However, existing solutions are known to be costly in terms of computation and communication, especially when dealing with very large sets. Private Information Retrieval (PIR) [25] may also be a solution to this problem. However, using PIR raises two main issues: first, it would incur a significant communication overhead; second, PIR is designed to retrieve a single element per query, whereas an efficient protocol for the popularity check should allow to check the existence of multiple data segment IDs at once. Hence instead of complex cryptographic primitives like PSI and PIR we suggest a secure mechanism for popularity detection based on a lightweight building block called Perfect Hashing [11]. We aim at solving this problem by designing a novel secure lookup protocol, which is defined in next section, based on Perfect Hashing [11].

3 Basic Idea: Popularity Detection Based on Perfect Hashing

The popularity detection solution we propose makes use of the Perfect Hashing process which, given an input set of n data segments, finds a collision-free hash function, called the perfect hash function (PHF), that maps the input set to a set of m integers (m being larger than n by a given load factor). The CSP can run this process in order to generate the PHF matching the IDs of the convergent encrypted popular blocks that are currently stored at the CSP. The resulting PHF can be efficiently encoded into a file and sent to the client. Using the PHF received from the CSP, the client can lookup for new blocks in the set of encrypted popular block IDs stored at the CSP, as illustrated in Figure 1. For each new block D, the client first encrypts the block to get $CE(D)$, he then computes the ID thereof using an unkeyed hash function h like SHA-3. Finally, by evaluating the PHF over ID, the client gets the lookup index i for the new block. The integer i will be the input of the lookup query issued by the client. Once the CSP has received the lookup query containing i, he will return to the client the convergent encrypted popular block ID stored under i. At this point, the client can easily detect the popularity of his data segment by comparing the ID he computed with the one received from the CSP: if the two IDs match, then D is popular. As mentioned above, it is a crucial requirement to prevent the CSP from discovering the content of the block D when it is yet unpopular. We achieve so by introducing an enhanced and secure version of Perfect Hashing, which makes the generated PHF one-way, meaning that the CSP cannot efficiently derive the input of the PHF from its output i. This also implies that the PHF must yield well-distributed collisions for unpopular blocks.

However, even though the client is now able to securely detect the popularity of a block, he still needs to handle the popularity transition, that is the phase in which a block reaches the threshold t and the convergent encrypted block needs to be uploaded to the CSP. Since the client cannot be aware of other copies of the same block previously uploaded by other users, a mechanism to keep track of the unpopular data blocks is needed. Clearly, the client cannot rely on the CSP for this task, as the CSP is not a trusted component. Therefore, we propose to introduce a semi-trusted component called Index Service (IS), which is responsible for keeping track of unpopular blocks. If the result of a popularity check is negative, then the client updates the IS accordingly by sending the popular convergent encrypted block ID and the ID of the symmetrically encrypted block. As soon as a block becomes popular, that is reaches the threshold t, the popularity transition is triggered and the client is notified in order to let him upload the convergent encrypted block, which from now on will be deduplicated by the CSP. Upon a popularity transition, the IS will delete from its storage any information related to the newly popular block. Regarding the popularity threshold, we point out that users do not have to be aware of its value, since the popularity transition is entirely managed by the IS, that is responsible for determining the current value for t. For instance, the value of t may be either static or dynamic, as proposed in [15]. Indeed, our scheme is completely independent of the strategy used for determining the value of the popularity threshold.

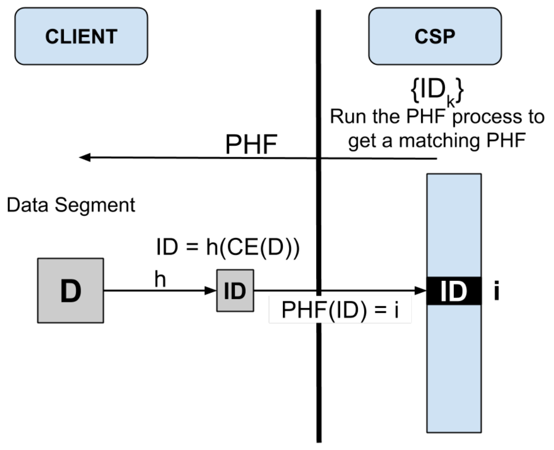

Fig. 1. The secure PHF allows users to detect popular blocks while preventing the CSP from discovering unpopular blocks

4 Background

4.1 Convergent Encryption

The idea of convergent encryption (CE) [4] is to derive the encryption key from the hash of the plaintext. A basic implementation of convergent encryption can be defined as follows: a user computes the encryption key using the message by applying a secure hash function H over M: $K = H(M)$; the message can then be encrypted with this key using a block cipher E: hence, $C = E(K, M) = E(H(M), M)$. Thanks to this technique, two users with two identical plaintexts will obtain two identical ciphertexts since the encryption key is the same and the encryption algorithm is deterministic. Despite its practicality, CE is known to be vulnerable to several weaknesses which undermine its capability of protecting confidential data and allow an attacker who has access to the storage server to perform offline dictionary attacks and discover predictable files. As shown in [9], CE is unfortunately exposed to the two following attacks: confirmation-of-a-file (COF) and learn-the-remaining-information (LRI). These attacks exploit the deterministic relationship between the plaintext and the encryption key and therefore can be successful in the verification whether a given plaintext has already been stored.

4.2 Perfect Hashing

A Perfect Hash Function (PHF) maps a set of arbitrary entries into a set of integers without collisions. Authors in [11] proposed a new algorithm that allows finding a perfect mapping for very large sets in a very efficient way. This algorithm, which is called CHD (Compress, Hash and Displace), achieves linear space and computational complexities (with respect to the size of the set). The main idea behind this algorithm is to split the input set into several buckets (subsets) with a few elements and find a collision-free mapping for each of these buckets separately. This approach has proved to be much more scalable than previous approaches. The mean number of elements per bucket is a parameter that can be tuned upon executing the generation algorithm. CHD also allows choosing a load factor, which is the fraction of non-empty positions in the hash table.

Although perfect hashing is widely adopted for efficient indexing in the field of relational databases [19], it has some desirable properties which make it an appropriate building block for our scheme. First, the computational complexity to build the PHF is linear and the PHF can be evaluated in constant time. Thanks to these properties, the system is scalable since the PHF generation remains feasible when dealing with very large datasets. In addition to that, the main computational load is outsourced to the CSP, while the client only has to perform very simple and lightweight operations such as evaluating the PHF on block IDs and symmetrically encrypting data blocks. Second, thanks to a special encoding and compression mechanism, the size of the PHF file is small and therefore it can easily be transferred to the client. Therefore, the performance impact is minimal and this approach can easily scale up to sets of millions of elements. Third, the resulting hash table is collision-free with respect to the elements of the input set (popular block IDs), meaning that any index is associated to at most one element of the input set. On the other hand, if the PHF is evaluated over the rest of the domain (unpopular block IDs) then collisions are well-distributed. This property is an important starting point to build our secure lookup protocol which must guarantee that an attacker is not able to determine on what input the PHF has been evaluated. Indeed, while an index in the hash table corresponds to a unique popular block ID, many unpopular block IDs are mapped to the same index. Therefore, given an index in the hash table, the CSP cannot determine the corresponding block ID. In our solution we propose to extend the existing PHF by replacing the underlying hash function with a one-way secure hash function such as SHA-3 [24]. Indeed, for the security of the scheme, it is crucial that the hash function used by the algorithm is one-way, meaning that it is easy to compute on a given input, but hard to invert given the image of a random input.

5 Our solution

5.1 Overview

We consider a scenario where users want to store their data (files) on a potentially untrusted Cloud Storage Provider (CSP) while taking advantage of source-based block-level deduplication and protecting the confidentiality of their data at the same time. Users run a client C which is a lightweight component with respect to both storage and computational capacity. CSP is assumed to be honest-but-curious and thus correctly stores users' data while trying to disclose the content thereof. Prior to uploading its data, C runs a secure lookup protocol to check whether the data are popular. The CSP is responsible for the generation of the PHF over the popular blocks and the storage of the resulting collision-free hash table. The proposed protocol introduces a trusted third party called Index Service (IS) which helps the client to discover the actual number of copies of a yet unpopular block. We stress the fact that IS only stores information on unpopular blocks and once a block becomes popular, all corresponding information are removed from its database, hence this component does not need to have a significant storage capacity.

The proposed solution is described under three different scenarios:

- Unpopular data upload (Scenario 1): if C finds out that the data is yet unpopular, it performs the upload to the CSP and updates the IS;
- Popularity transition (Scenario 2): if C finds out that the popularity degree of the data is $t-1$ (where t is the popularity threshold), then it performs the appropriate operations to upload the newly popular data. IS removes all information with respect to this specific data and CSP deletes all the encrypted copies previously stored;
- Popular data upload (Scenario 3): C only uploads metadata since it has detected that the requested data is popular, therefore deduplication can take place.

CSP stores a hash table for popular block IDs which is constructed with the previously introduced PHF. Each element of the hash table is defined by the couple $(PHF(h(CE(b_i))), h(CE(b_i)))$ where $h(CE(b_i))$ is the unkeyed secure hash of the convergent encrypted block. Before any operation, given the current set of popular blocks, CSP creates a corresponding secure PHF. This PHF is updated only when CSP needs to store new popular blocks. In the next sections, we first present the popularity check phase which is common to all three scenarios and then explain the following phases.

5.2 Popularity Check (Scenarios 1, 2 and 3)

Before uploading a file F, C splits F into blocks $F = \{b_i\}$, encrypts each of them with CE and computes their IDs. We point out that our scheme is completely independent of the underlying data-chunking strategy used for determining block boundaries, which is a problem that is out of the scope of this paper. The client fetches the PHF from the CSP and evaluates it over $\{h(CE(b_i))\}$. The result of this operation is a set of indices $I = \{PHF(h(CE(b_i)))\}$, where each index represents the position of the potentially popular block ID in the hash table stored at the CSP. These indices can be used to perform the popularity check without revealing the content of the blocks to the CSP. Indeed, given a set of indices obtained as above, the client can retrieve the corresponding block IDs stored in the hash table and then compare them with his own block IDs. Any block b_i such that $h(CE(b_i))$ is equal to the popular block ID retrieved from the CSP, is considered as popular, hence will be deduplicated. The index does not reveal any exploitable information on the block.

5.3 Popularity Transition (Scenarios 1 and 2)

If the popularity check reveals that a block is not popular, C needs to check whether it is going to trigger a popularity transition. A block becomes popular as soon as it has been uploaded by t users. In order to enable C to be aware of the change of the popularity status and perform the transition, C sends an update to the IS whenever the popularity check has returned a negative result for a given block ID. IS stores a list of block IDs and owners corresponding to each encrypted copy of the yet unpopular block. When the number of data owners for a particular block reaches t, the popularity transition protocol is triggered and IS returns to C the list of block IDs. In order to complete this transition phase, CSP stores the convergent-encrypted copy, removes the corresponding encrypted copies and updates the PHF. From now on, the block will be considered popular, therefore it will be deduplicated. We point out that this operation is totally transparent to the other users who uploaded the same block as unpopular. Indeed, during their upload phase, users also keep encrypted information about the convergent encryption key. This allows them decrypting the block when it becomes popular.

5.4 Data Upload (Scenarios 1, 2 and 3)

Once the client has determined the popularity of each block, he can send the actual upload request. The content of the request varies depending on the block status. If the block is unpopular, C uploads the block symmetrically encrypted with a random key. If the block is popular, C only uploads the block ID, so that the CSP can update his data structures. Optionally, in order to avoid to manage the storage of the encryption keys, C may rely on the CSP for the storage of the random encryption key and the convergent encryption key, both encrypted with a secret key known only by the client.

6 Security Analysis

In this section, we analyze the security of the proposed scheme, the CSP being considered the main adversary. The CSP is "honest-but-curious", meaning that it correctly performs all operations but it may try to discover the original content of unpopular data. We do not consider scenarios where the CSP behaves in a byzantine way. We assume that CSP cannot collude with the IS since this component is trusted. Since the goal of the malicious CSP is to discover the content of unpopular blocks, we analyze in detail whether (and how) confidentiality is guaranteed for unpopular data in all phases of the protocol. However, if the user wants to keep a file confidential even when it becomes popular, he may encrypt the file with a standard encryption solution and upload it to the cloud without following the protocol steps. Finally, we also analyze some attacks that may be perpetrated by users themselves and propose simple countermeasures against them.

Security of blocks stored at the CSP By definition, an unpopular block is encrypted using a semantically-secure symmetric encryption. The confidentiality of unpopular data segments thus is guaranteed thanks to the security of the underlying encryption mechanism.

Security during Popularity Check The information exchanged during the Popularity Check must not reveal any information that may leak the identity of an unpopular block owned by the user. The identity of an unpopular block is protected thanks to the one-wayness of the secure PHF: the query generated by the client does not include the actual unpopular block ID but an integer i that is calculated by evaluating the secure PHF on the block ID. Simple guessing by exploring the results of the secure hash function embedded in the PHF is not feasible thanks to the one-wayness of the underlying secure hash function (SHA-3 [24]). In addition to that, when the PHF is evaluated over an unpopular block ID, there is definitely a collision between the ID of the unpopular block and the ID of a popular block stored at the CSP. These collisions serve as the main countermeasure to the disclosure of the unpopular block ID sent to the CSP during the lookup. With a reasonable assumption, we can also consider that the output of the underlying secure hash function (SHA-3) is random. In case of a collision between an unpopular block ID and the ID of a popular block stored at the CSP, thanks to the randomness of the underlying secure hash function, the output of a PHF based on such a hash function is uniformly distributed between 0 and m. In the case of such a collision, the probability that the CSP guesses the unpopular block ID used as input to the PHF by the client thus is:

$$\frac{m}{|\bar{P}|} = \frac{|P|}{|\bar{P}| * \alpha} \tag{1}$$

where P is the set of popular block IDs stored at the CSP, $\bar{P}$ is the rest of the block ID domain including all possible unpopular block IDs, α is the load factor of the PHF such that $m = \frac{|P|}{\alpha}$.

Assuming that the cardinality of the entire domain is much larger than the cardinality of the set of popular block IDs (which is the case if popular block IDs are the result of a secure hash function), we can state that the number of collisions per index is large enough to prevent a malicious CSP from inferring the actual block ID used as input to the PHF. In a typical scenario using a PHF based on a secure hash function like SHA-3, whereby the complexity of a collision attack would be 2^{256}, and a popular block ID set with 10^9 elements, this probability will be ($\alpha = 0.81$):

$$\frac{10^9}{(2^{256} - 10^9) * 0.81} \approx 1.06 * 10^{-68} \tag{2}$$

Hence collisions can effectively hide the identity of unpopular blocks from an untrusted cloud provider while keeping the lookup protocol extremely efficient and lightweight for the users.

Security against potential protocol vulnerabilities We now consider a few additional attacks that may be perpetrated by the CSP. For each of them, we propose simple but effective countermeasures, which are easy to implement and do not significantly increase the computational and network overhead. First, we consider that the CSP may pre-build a PHF based on some specific data (derived for example from a dictionary) which have not been yet uploaded by users. Within such a scenario, clients would detect their requested block to be popular although it has never actually been uploaded by any user; such a block will then be stored with a lower level of protection. As a countermeasure to such an attack, we propose that the IS attaches a signature to each popular block ID upon the Popularity Transition. Therefore, the IS will sign popular block IDs before being stored at the CSP, enabling clients to verify the authenticity of these blocks when running the popularity check. Such a countermeasure would have a minimal impact on the performance of the system. Another attack we consider is related to the confirmation-of-file attack to which convergent encryption is also vulnerable [9]. Indeed, upon a Popularity Check, the CSP may compare the sequence of indices sent by the client with the sequence produced by a given popular file F. If the two sequences match, then there is a chance that the client is actually uploading F. In order to hide this information from the CSP, the client may add a number of random indices to the list of indices being sent upon the Popularity Check. Thanks to the resulting noise included in the index list, the identification of the target file by the CSP will be prevented. This countermeasure also prevents the CSP from running the learn-the-remaining-information attack. Moreover, the overhead due to this countermeasure is negligible both in terms of bandwidth and computation.

Security against users Users may force a popularity transition by repeatedly uploading random or targeted blocks. As a countermeasure, the popularity threshold may be set to a value $t' = t + u$, where u is the expectation of the maximum number of malicious users. As opposed to the proposal of [10], the threshold can be dynamically updated at any time of the system life. Indeed, this parameter is transparent to both users and the CSP, hence the Index Service can update it depending on the security needs. Users may also perpetrate a DoS attack by deleting random blocks stored at the cloud. This may happen upon a popularity transition: the client is asked to attach a list of block IDs that may not be the actual encrypted copies of the block being uploaded. We suggest making the Index Service sign the list of block IDs to be deleted so that the cloud can verify whether the request is authentic. This signature does not significantly increase the overhead since several schemes for short signatures [22] have been proposed in the literature.

7 Performance Evaluation

7.1 Prototype Implementation

In order to prove the feasibility of our approach, we implemented a proof-of-concept prototype consisting of the three main components, namely, the Client, the IS and the CSP. All components have been implemented in Python. Cryptographic functions have been implemented using the pycrypto library[4]. Both the Client and the IS run on an Ubuntu VM hosted on our OpenStack platform, while the CSP runs on an Ubuntu VM hosted on Amazon EC2 (EU Region). The IS uses REDIS[5] in order to store the information on unpopular blocks, which are encoded as lists. Metadata (block IDs, file IDs, files structures, encrypted keys) are stored in a MySQL database. Perfect Hashing has been implemented using the CMPH library[6] at both the Client and the CSP. In order to achieve one-wayness, we customized CMPH by replacing the internal hash function with SHA256 [20]. We stress the fact that this is a proof-of-concept implementation, therefore for the sake of simplicity the CSP has been deployed on a VM where data blocks are stored locally. In a production environment, the CSP service may be deployed on a larger scale and any storage provider such as Amazon S3[7] may be employed to physically store blocks.

We consider a scenario where the client uploads a 10MB file to the CSP prefilled with 10^6 random blocks. We propose to first evaluate the computational overhead of each single component and measure the total time a client needs to wait during each phase until the data upload has been completed. We then analyze the network overhead of the proposed solution. Our analysis considers the three previously described scenarios:

– Scenario 1 (Unpopular File): the file to be uploaded is still unpopular;
– Scenario 2 (Popularity Transition): the file has triggered a popularity transition hence is going to become popular;
– Scenario 3 (Popular File): the file to be uploaded is already popular.

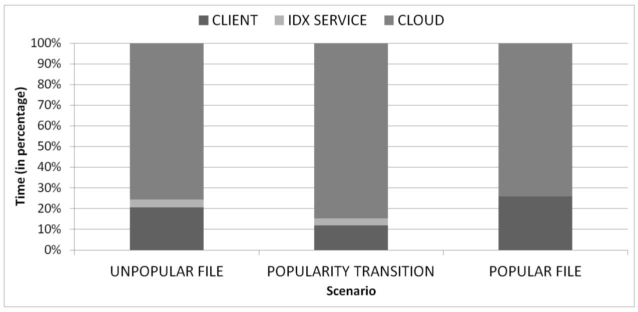

Fig. 2. Portion of the total computation time spent at each component in each scenario

7.2 Computational Overhead

In this section we present our measurements of the computational overhead at each component and then show the total time a client takes to upload a file. Figure 2 shows an aggregate measure of all computation-intensive operations each component performs. The results prove that, as expected, the computational overhead introduced in the CSP is much higher than the one affecting the client. Also, since the operations performed by the IS are extremely simple, its computational overhead is negligible.

Figure 3 shows more detailed results by highlighting which operations introduce a higher computational overhead. The results prove that:

- Symmetric encryption introduces a negligible computational overhead, hence it does not affect the system performance;
- The client-side Popularity Check is extremely lightweight and thus introduces a negligible computational overhead;
- The most computation-intensive operations (PHF generation, hash table storage, upload processing) are performed by the CSP, hence a big fraction of the computational overhead is outsourced to the CSP.

Figures 4 and 5 show the results of an in-depth study on the performance of the Perfect Hashing algorithm, both in terms of storage space and computation time for the generation of the PHF. The generation time also includes the time needed to store the hash table. We measured these quantities on a dataset of 10^6 random block IDs while varying the load factor and the bucket size. The former is a coefficient indicating the fraction of non-empty positions in the final collision-free hash table; the latter is the mean number of elements in each subset of the input set (see [11] for further details). As we can observe from Figures 4 and 5, the optimal bucket size is between 3 and 4 and the load factor should not

[4] https://pypi.python.org/pypi/pycrypto
[5] http://redis.io
[6] http://cmph.sourceforge.net/
[7] https://aws.amazon.com/s3

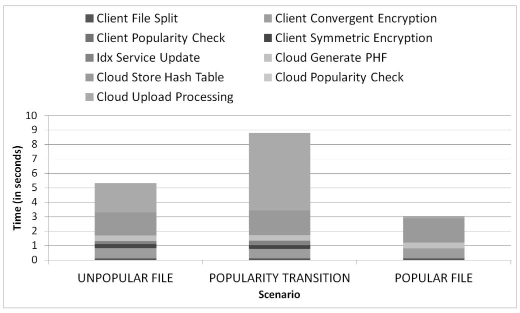

Fig. 3. Total time spent during each phase of the protocol in each scenario

be greater than 0.8. These parameters can be tuned depending on the scenario (e.g. bandwidth) in order to achieve the best performance.

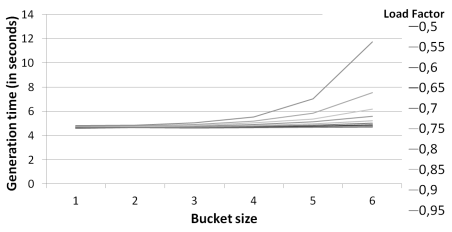

Fig. 4. Analysis of PHF generation time with varying parameters for a set containing 10^6 elements

Furthermore, as mentioned earlier, in order to improve the security of our novel lookup protocol, we replaced the default hash function employed by the CMPH library (Jenkins [21]) with SHA-3. This improvement is required for the following reason: using a non-secure hash function would allow an adversary such as the CSP to easily enumerate all block IDs mapped to a given index of the hash table. Such a threat may compromise the security of the whole system and make the popularity check protocol insecure.

Conclusion Figure 6 summarizes all measurements by showing the total time spent during each phase of the upload protocol within the three scenarios. These results show that despite the delay introduced by the Popularity Check phase, the user achieves a throughput of approximately 1MB per second even when a file does not contain any popular block.

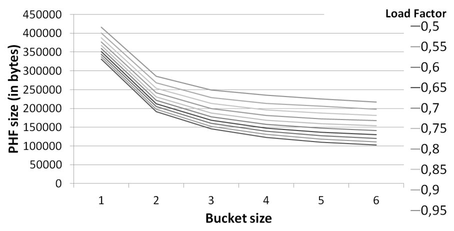

Fig. 5. Analysis of PHF size with varying parameters for a set containing 10^6 elements

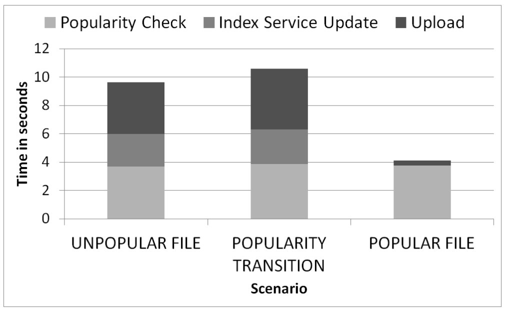

Fig. 6. Total time spent by all components when uploading a file (including Popularity Check) in each scenario

7.3 Communication Overhead

In this section we analyze the communication overhead of our scheme considering the same scenarios. The upload has been split into multiple sub-operations: PHF Download, Popularity Check, Index Service Update (not performed in Scenario 2) and the Upload. For each of these operations we analyze the size of all messages exchanged (both requests and responses). Table 1 regroups all the results expressed in MB. The PHF Download response size is linear with respect to the set of popular block IDs. The larger the set, the larger the response will be. However, as shown in [11], the size of PHF file is about 1.4 bits per popular block ID; hence this operation does not introduce a significant delay even when dealing with very large datasets. We point out that the PHF file does not have to be downloaded at every request, since the user can cache it. Furthermore, the size of the Popularity Check request and response is linear with respect to the number of blocks in the file that is being uploaded. The Popularity Check request contains a list of indices (one integer per block), while the response contains a list of block IDs (one per index) of 32 bytes each. The Index Service Update request is only sent for unpopular blocks. The request consists of two block IDs (32 bytes each) per block. The response size varies depending on whether the popularity transition occurs. If the file has triggered a popularity transition, then the response includes a list of block IDs, otherwise it is empty. As we can see from Table 1, requests and responses of the Popularity Check and the Index

Service Update operations have a negligible size with respect to the file size. Finally, the size of the Upload request varies depending on the block status. If a block is popular, the request only consists of the block ID and one key (32 bytes). If a block is not popular, the request contains the encrypted data, two keys (32 bytes each) and a few fields: the file ID (32 bytes), the user ID and the block status (1 byte). As shown in Table 1, the overhead introduced by the Upload is minimal and mainly depends on the encoding method used to transfer the encrypted binary data. For simplicity, we used JSON objects to pack encrypted blocks and keys and Base64 to encode binary data, which increases the size of the data by 1/3. To summarize, the preliminary operations performed in our scheme before the Upload introduce a negligible communication overhead. In addition, the scheme does not affect the gains in terms of storage space and bandwidth achieved thanks to deduplication.

	SCENARIO 1	**SCENARIO 2**	**SCENARIO 3**
PHF DOWNLOAD IN	0.67	0.67	0.67
POPULARITY CHECK REQUEST	0.004	0.004	0.004
POPULARITY CHECK RESPONSE	0.02	0.02	0.02
INDEX SERVICE UPDATE REQUEST	0.1	0.1	-
INDEX SERVICE UPDATE RESPONSE	0.009	0.04	-
UPLOAD REQUEST	13.51	13.47	0.09

Table 1. Communication overhead (in MB) introduced by each operation

8 Related Work

Secure deduplication for cloud storage has been widely investigated both in the literature and in the industry. Convergent encryption, has been proposed as a simple but effective solution to achieve both confidentiality and deduplication [1, 3, 4]. However, it is vulnerable to well-known attacks which put data confidentiality at risk [3, 4]. A relevant work on this topic is DupLESS [8], which is based on a privacy-preserving protocol running between the user and a trusted key server. If an attacker learns the secret stored at the key server, confidentiality can no longer be guaranteed. Recently, a system called ClouDedup [9] has been proposed, which achieves secure and efficient block-level deduplication while providing transparency for end users. However, the system relies on a complex architecture in which users have to trust an encryption gateway which takes care of encrypting/decrypting data. Similarly to DupLESS, the leakage of the secret key compromises confidentiality.Another relevant work is iMLE [2], which proposes an elegant scheme for secure data deduplication. However, the scheme is purely theoretical, hence cannot be adopted in real scenarios. In fact, it makes an extensive use of fully homomorphic encryption [23]. To the best of our knowledge, one of the most recent and relevant works in the field of secure data deduplication is [10], which is based on the idea of differentiating data protection depending on its popularity and makes use of a mixed cryptosystem combining convergent encryption and a threshold encryption scheme. However, this work suffers from a few drawbacks which we aim to solve. First, the system suffers from a significant storage and bandwidth overhead. Indeed, for each unpopular file the user uploads two encrypted copies, one encrypted with a random symmetric key and one encrypted with the mixed encryption scheme. In scenarios with a high percentage of unpopular files, the storage overhead will be significant and nullify the savings achieved thanks to deduplication. We propose to eliminate the storage overhead by storing one single copy for each data segment at a time, encrypted with either a random symmetric key or a convergent key. Second, the system proposed in [10] relies on a trusted component which provides an indexing service for all data, both popular and unpopular. We propose to limit the usage of this trusted component to unpopular data. In our scheme, popular data can be detected thanks to the secure lookup protocol, whereby [10] relies on the trusted component. Third, the effectiveness of the system proposed in [10] is limited to file-level deduplication, which is known to achieve lower space savings than block-level deduplication. Fourth, both the client and the CSP have to perform complex cryptographic operations based on threshold cryptography on potentially very large data. As opposed to this, our proposed scheme has been designed to perform only simple and lightweight cryptographic operations, which significantly lowers the cost for the client. Fifth, our scheme does not require any coordination or initialization among users as opposed to [10]'s requirement to setup and distribute key shares among users.

9 Conclusion and Future Work

We designed a system which guarantees full confidentiality for confidential files while enabling source-based block-level deduplication for popular files. The main building block of our system is our novel secure lookup protocol built on top of an enhanced version of Perfect Hashing. To the best of our knowledge, this is the first work that uses Perfect Hashing for a different purpose other than database indexing. Our system is not based on any key-management protocol, hence it does not require users to agree on a shared secret or trust a third party for storing encryption keys. A semi-trusted component is employed for the purpose of storing metadata concerning unpopular data and providing a support for detecting popularity transitions, meaning that a data block has just reached the popularity threshold. We also implemented a prototype of the proposed solution. Our measurements show that the storage, network and computational overhead is affordable and does not affect the advantage of deduplication. Also, we showed that the computational overhead is moved to the CSP, while the client has to perform very lightweight operations. As part of future work, PerfectDedup may be optimized in order to reduce the overhead due to the PHF generation and transmission.

References

1. Xu, Jia, Ee-Chien Chang, and Jianying 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, pp. 195-206. ACM, 2013.
2. Bellare, Mihir, and Sriram Keelveedhi. "Interactive message-locked encryption and secure deduplication." (2015).
3. Adya, Atul, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment." ACM SIGOPS Operating Systems Review 36, no. SI (2002): 1-14.
4. Douceur, John R., Atul Adya, William J. Bolosky, P. Simon, and Marvin Theimer. "Reclaiming space from duplicate files in a serverless distributed file system." In Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on, pp. 617-624. IEEE, 2002.
5. Perttula. Attacks on convergent encryption. `http://bit.ly/yQxyvl.`
6. Liu, Chuanyi, Xiaojian Liu, and Lei Wan. "Policy-based de-duplication in secure cloud storage." In Trustworthy Computing and Services, pp. 250-262. Springer Berlin Heidelberg, 2013.
7. Meyer, Dutch T., and William J. Bolosky. "A study of practical deduplication." ACM Transactions on Storage (TOS) 7, no. 4 (2012): 14.
8. Bellare, Mihir, Sriram Keelveedhi, and Thomas Ristenpart. "DupLESS: server-aided encryption for deduplicated storage." In Proceedings of the 22nd USENIX conference on Security, pp. 179-194. USENIX Association, 2013.
9. Puzio, Pasquale, Refik Molva, Melek Önen, and Sergio Loureiro. "ClouDedup: secure deduplication with encrypted data for cloud storage." In Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on, vol. 1, pp. 363-370. IEEE, 2013.
10. Stanek, Jan, Alessandro Sorniotti, Elli Androulaki, and Lukas Kencl. "A secure data deduplication scheme for cloud storage." In Financial Cryptography and Data Security, pp. 99-118. Springer Berlin Heidelberg, 2014.
11. Belazzougui, Djamal, Fabiano C. Botelho, and Martin Dietzfelbinger. "Hash, displace, and compress." In Algorithms-ESA 2009, pp. 682-693. Springer Berlin Heidelberg, 2009.
12. Cox, Landon P., Christopher D. Murray, and Brian D. Noble. "Pastiche: Making backup cheap and easy." ACM SIGOPS Operating Systems Review 36, no. SI (2002): 285-298.
13. Rabin, Michael O. Fingerprinting by random polynomials. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.
14. Wilcox-O'Hearn, Zooko, and Brian Warner. "Tahoe: the least-authority filesystem." In Proceedings of the 4th ACM international workshop on Storage security and survivability, pp. 21-26. ACM, 2008.
15. Harnik, Danny, Benny Pinkas, and Alexandra Shulman-Peleg. "Side channels in cloud services, the case of deduplication in cloud storage." IEEE Security & Privacy 8, no. 6 (2010): 40-47.
16. Is Convergent Encryption really secure? `http://bit.ly/Uf63yH`
17. Bellare, Mihir, Sriram Keelveedhi, and Thomas Ristenpart. "Message-locked encryption and secure deduplication." In Advances in CryptologyEUROCRYPT 2013, pp. 296-312. Springer Berlin Heidelberg, 2013.

18. Storer, Mark W., Kevin Greenan, Darrell DE Long, and Ethan L. Miller. "Secure data deduplication." In Proceedings of the 4th ACM international workshop on Storage security and survivability, pp. 1-10. ACM, 2008.
19. Olumofin, Femi, and Ian Goldberg. "Privacy-preserving queries over relational databases." In Privacy enhancing technologies, pp. 75-92. Springer Berlin Heidelberg, 2010.
20. Description of SHA256 `http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf`
21. Jenkins hash function `http://www.burtleburtle.net/bob/c/lookup3.c`
22. Boneh, Dan, Ben Lynn, and Hovav Shacham. "Short signatures from the Weil pairing." In Advances in CryptologyASIACRYPT 2001, pp. 514-532. Springer Berlin Heidelberg, 2001.
23. Gentry, Craig. "A fully homomorphic encryption scheme." PhD diss., Stanford University, 2009.
24. SHA-3. `http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf`
25. Chor, Benny, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. "Private information retrieval." Journal of the ACM (JACM) 45, no. 6 (1998): 965-981.
26. Freedman, Michael J., Kobbi Nissim, and Benny Pinkas. "Efficient private matching and set intersection." Advances in Cryptology-EUROCRYPT 2004. Springer Berlin Heidelberg, 2004.