Secure Deduplication of Encrypted Data without Additional Independent Servers

Jian Liu
Aalto University
jian.liu@aalto.fi

N. Asokan
Aalto University and
University of Helsinki
asokan@acm.org

Benny Pinkas
Bar Ilan University
benny@pinkas.net

Abstract

Encrypting data on client-side before uploading it to a cloud storage is essential for protecting users' privacy. However client-side encryption is at odds with the standard practice of deduplication. Reconciling client-side encryption with cross-user deduplication is an active research topic. We present the first secure cross-user deduplication scheme that supports client-side encryption *without requiring any additional independent servers*. Interestingly, the scheme is based on using a PAKE (password authenticated key exchange) protocol. We demonstrate that *our scheme provides better security guarantees than previous efforts*. We show both the effectiveness and the efficiency of our scheme, via simulations using realistic datasets and an implementation.

Categories and Subject Descriptors

E.3 [**Data Encryption**]

Keywords

Cloud Storage; Deduplication; Semantically Secure Encryption; PAKE

1. INTRODUCTION

Cloud storage is a service that enables people to store their data on a remote server. With a rapid growth in user base, cloud storage providers tend to save storage costs via *cross-user deduplication*: if two clients upload the same file, the storage server detects the duplication and stores only a single copy. Deduplication achieves high storage savings [23] and is adopted by many storage providers. It is also adopted widely in backup systems for enterprise workstations.

Clients who care about privacy prefer to have their data encrypted on the client-side using *semantically secure* encryption schemes. However, naïve application of encryption thwarts deduplication since identical files are uploaded as completely independent ciphertexts. Reconciling deduplication and encryption is an active research topic. Current solutions either use *convergent encryption* [13], which is susceptible to *offline brute-force attacks*, or require the aid of additional *independent* servers [4, 26, 28], which is a strong assumption that is very difficult to meet in commercial contexts. Furthermore, some schemes of the latter type are susceptible to *online brute-force attacks*.

Our Contributions. We present the first *single-server* scheme for secure cross-user deduplication with client-side encrypted data. Our scheme allows a client uploading an existing file to securely obtain the encryption key that was used by the client who has previously uploaded that file. The scheme builds upon a well-known cryptographic primitive known as *password authenticated key exchange* (PAKE) [7], which allows two parties to agree on a session key iff they share a short secret ("password"). PAKE is secure even if the passwords have low entropy. In our deduplication scheme, a PAKE-based protocol is used to compute identical keys for different copies of the same file. Specifically, a client uploading a file first sends a short hash of this file (10-20 bits long) to the server. The server identifies other clients whose files have the same short hash, and lets them run a single round PAKE protocol (routed through the server) with the uploader using the (long, but possibly low entropy) hashes of their files as the "passwords". At the end of the protocol, the uploader gets the key of another client iff their files are identical. Otherwise, it gets a random key.

Our scheme uses a *per-file rate limiting* strategy to prevent online brute-force attacks. Namely, clients protect themselves by limiting the number of PAKE instances they will participate in for each file. Compared with the commonly used *per-client rate limiting* (used in DupLESS [4]), which limits the number of queries allowed for each client (during a time interval), our scheme is significantly more resistant to online brute-force attacks by an adversary who has compromised multiple clients, or by the storage server. Per-client rate limiting is not fully effective against such attacks because the adversary can use different identities.

At a first glance, it seems that our scheme incurs a high communication and computation overhead because a client uploading a file is required to run PAKE many times due to the high collision rate of the short hash. In fact, the number of PAKE runs for an upload request is limited to a certain level by the rate limiting strategy. For a requested short hash, the server only checks a subset of existing files (in descending order of file popularity) that have the same short hash. This implies that our scheme may fail to find duplicates for some requests, and this will certainly reduce the deduplication effectiveness. Nonetheless, our simulations in Section 6 show that this negative effect is very small. The reason is that the file popularity distribution is far from being uniform, and popular files account for most of the benefit from deduplication. Our scheme can almost always find duplicates for these popular files.

To summarize, we make the following contributions:

- Presenting the **first single-server scheme for cross-user deduplication** that enables client-side semantically secure encryption (except that, as in all deduplication schemes, ciphertexts leak the equality of the underlying files), and proving its security in the **malicious model** (Section 5);

- Showing that our scheme has **better security guarantees** than previous work (Section 5.2). As far as we know, our proposal is the first scheme that can prevent online brute-force attacks by compromised clients or server, *without* introducing an identity server;
- Demonstrating, via simulations with realistic datasets, that our scheme provides its privacy benefits while retaining a utility level (in terms of deduplication effectiveness) on **par with standard industry practice**. Our use of per-file rate limiting implies that an incoming file will *not* be checked against all existing files in storage. Notably, our scheme still achieves high deduplication effectiveness (Section 6);
- **Implementing** our scheme to show that it incurs **minimal overhead** (computation/communication) compared to traditional deduplication schemes (Section 7).

2. PRELIMINARIES

2.1 Deduplication

Deduplication strategies can be categorized according to the basic data units they handle. One strategy is *file-level deduplication* which eliminates redundant files [13]. The other is *block-level deduplication*, in which files are segmented into blocks and duplicate blocks are eliminated [27]. In this paper we use the term "file" to refer to both files and blocks.

Deduplication strategies can also be categorized according to the host where deduplication happens. In *server-side deduplication*, all files are uploaded to the storage server, which then deletes the duplicates. Clients are unaware of deduplication. This strategy saves storage but not bandwidth. In *client-side deduplication*, a client uploading a file first checks the existence of this file on the server (by sending a hash of the file). Duplicates are not uploaded. This strategy saves both storage and bandwidth, but allows a client to learn if a file already exists on the server.

The effectiveness of deduplication is usually expressed by the *deduplication ratio*, defined as the "the number of bytes input to a data deduplication process divided by the number of bytes output" [15]. In the case of a cloud storage service, this is the ratio between total size of files uploaded by all clients and the total storage space used by the service. The *dedpulication percentage* (sometimes referred to as "space reduction percentage") [15, 19] is $1 - \frac{1}{\text{deduplication ratio}}$. A *perfect deduplication* scheme will detect all duplicates.

The level of deduplication achievable depends on a number of factors. In common business settings, deduplication ratios in the range of 4:1 (75%) to 500:1 (99.8%) are typical. Wendt et al. suggest that a figure in the range 10:1 (90%) to 20:1 (95%) is a *realistic expectation* [29].

Although deduplication benefits storage providers (and hence, indirectly, their users), it also constitutes a privacy threat for users. For example, a cloud storage server that supports client-side, cross-user deduplication can be exploited as an oracle that answers "did anyone upload this file?". An adversary can do so by uploading a file and observing whether deduplication takes place [19]. For a *predictable file* that has low entropy, the adversary can construct all possible files, upload them and observe which file causes deduplication. Harnik et al. [19] propose a randomized threshold approach to address such online brute-force attacks. Specifically, for each file F, the server keeps a random threshold t_F ($t_F \geq 2$) and a counter c_F that indicates the number of clients that have previously uploaded F. Client-side deduplication happens only if $c_F \geq t_F$. Otherwise, the server does a server-side deduplication.

In Section 8, we survey the state-of-the-art of deduplication with encrypted data.

2.2 Hash Collisions

A *hash function* $H\colon \Phi \rightarrow \{0,1\}^n$ is a deterministic function that maps a binary string in Φ of arbitrary length to a binary string h of fixed length n. The term *cryptographic hash function* is used to denote that a hash function has random outputs, being one-way and infeasible to find collisions. We model the hash function H as a random oracle[1].

A cryptographic hash function with a long output is collision resistant, whereas a hash function with a short output has many collisions. As we will see in Section 4 and 5, we use a *short hash* function SH to improve efficiency and privacy.

2.3 Additively Homomorphic Encryption

A public key encryption scheme is *additively homomorphic* if given two ciphertexts $c_1 = Enc(pk, m_1; r_1)$ and $c_2 = Enc(pk, m_2; r_2)$, it is possible to efficiently compute $Enc(pk, m_1 + m_2; r)$ with a new random value r (which depends on r_1 and r_2 but cannot be determined by anyone who knows only r_1 or r_2), even without knowledge of the corresponding private key. Examples of such schemes are Paillier's encryption [24], and lifted ElGamal encryption [16] where plaintexts are encoded in the exponent. The Lifted ElGamal Encryption is described as follows:

- $Gen(1^\lambda)$ returns a generator g of a cyclic group $\mathbb{G}$ of order p, and an integer $h = g^x$ where x is a random value in $\mathbb{Z}_p$. The tuple $(\mathbb{G}, p, g, h)$ is the public key pk and $(\mathbb{G}, p, g, x)$ is the private key sk;
- $Enc(pk, m)$ chooses a random value y in $\mathbb{Z}_p$ and returns the ciphertext $c = (c_1, c_2)$ where $c_1 = g^y$ and $c_2 = g^m \cdot h^y$;
- $Dec(sk, c)$ returns $\frac{c_2}{c_1^x} = \frac{g^m \cdot h^y}{g^{xy}} = g^m$. Note that, in general, calculating the discrete logarithm of g^m is hard, but knowing g^m is enough in our application.

We use $E()/D()$ to denote symmetric encryption/decryption, and use $Enc()/Dec()$ to denote additively homomorphic encryption/decryption. We abuse the notation and use $Enc(pk, m)$ to denote $Enc(pk, m; r)$ where r is chosen uniformly at random. In addition, we use $\oplus$ and $\ominus$ to denote homomorphic addition and subtraction respectively.

2.4 Password Authenticated Key Exchange

Password-based protocols are commonly used for user authentication. However, such protocols are vulnerable to offline brute-force attacks (also referred to as *dictionary attacks*) since users tend to choose passwords with relatively

[1]Our deduplication scheme (see Section 5) will be based on a PAKE protocol (see Section 2.4) that uses the random oracle model. Therefore we choose this model for all our analyses.

Inputs:

- Alice's input is a password pw_a;
- Bob's input is a password pw_b.

Outputs:

- Alice's output is k_a;
- Bob's output is k_b.

where if $pw_a = pw_b$ then $k_a = k_b$, and if $pw_a \neq pw_b$ then Bob (resp. Alice) cannot distinguish k_a (resp. k_b) from a random string of the same length.

Figure 1: The ideal functionality $\mathcal{F}_{pake}$ for password authenticated key exchange.

low entropy that are hence guessable. Bellovin and Merritt [7] were the first to propose a password authenticated key exchange (PAKE) protocol, in which an adversary making a password guess cannot verify the guess without an online attempt to authenticate itself with that password. The protocol is based on using the password as a symmetric key to encrypt the messages of a standard key exchange protocol (e.g., Diffie-Hellman [12]), so that two parties with the same password successfully generate a common session key without revealing their passwords. If the passwords are different then neither party can learn anything about the key output by the other party (namely, cannot distinguish that key from a random key).

Following this seminal work, many protocols were proposed to improve PAKE in several aspects, e.g., achieving provably security [6, 8], weakening the assumption (i.e., working in standard model without random oracles) [18, 3], achieving a stronger proof model [11, 10] and improving the round efficiency [21, 6, 22].

The ideal functionality of PAKE, $\mathcal{F}_{pake}$, is shown in Figure 1. We use it as a building block, and require the following properties in addition to the ideal functionality:

- *Implicit key exchange:* At the end of the protocol, neither party learns if the passwords matched or not. (In fact, many PAKE protocols were designed to be *explicit* so that parties can learn this information.)
- *Single round:* The protocol must be single-round so that it can be easily facilitated by the storage server.
- *Concurrent executions:* The protocol must allow multiple PAKE instances to run in parallel. There are two common security notions for such PAKE protocols. One stronger notion is "UC-secure PAKE" [11], which guarantees security for composition with arbitrary protocols. The other notion is "concurrent PAKE", defined by [6, 8], where each party is able to concurrently run many invocations of the protocol. Implementations of this notion are much more efficient than implementations of UC-secure PAKE. We therefore use a concurrent PAKE protocol in our work.

Our deduplication scheme (see Section 5) uses the SPAKE2 protocol of Abdalla and Pointcheval [1], which is described in Figure 2. This protocol is secure in the concurrent setting and random oracle model. It requires each party to compute only three exponentiations, and send just a single group element to the other party. Theorem 5.1 in [1] states that this protocol is secure assuming that the computational Diffie-Hellman problem is hard in the group used by the protocol.

Inputs:

- Alice's input is a password pw_a;
- There are n parties $P_1, \ldots, P_n$, with passwords $pw_1, \ldots, pw_n$, respectively.

Outputs:

- Alice's output is $k_{a,1}, \ldots, k_{a,n}$;
- P_i's output is $k_{b,i}$.

where $\forall i \in [1, n]$ if $pw_a = pw_i$ then $k_{a,i} = k_{b,i}$, and otherwise P_i (resp. Alice) cannot distinguish $k_{a,i}$ (resp. $k_{b,i}$) from a random string of the same length.

Figure 3: The ideal functionality $\mathcal{F}_{same-input-pake}$.

Same-Input-PAKE. In our deduplication protocol, one client (Alice) runs multiple PAKE instances with other clients. The protocol must ensure that the client uses the *same input* in all these PAKE instances. We define this requirement in the functionality of *same-input-PAKE* described in Figure 3.

We list three possible methods for implementing the same-input-PAKE functionality: (1) The protocol can be based on the SPAKE2 protocol, where Alice uses the same first message X^* in her interactions with all clients, thus using the same input in all these instances.[2] We do not know how to prove security for this variant of PAKE, and leave it as a heuristic solution. (2) Alice can run independent SPAKE2 instances, with a different first message in each instance, and in addition prove in zero-knowledge that her inputs to all instances are identical. The proof can be based on standard sigma protocols for Diffie-Hellman tuples and the Fiat-Shamir heuristic (see, e.g. [20], Chap. 7), and requires only one additional exponentiation from Alice and two exponentiations from each other party. (3) The protocol can use generic protocols for non-interactive secure computation (NISC) [2]. These protocols are single round and secure against malicious adversaries. A variant called *multi-sender NISC* [2], has one party sending the first message of the protocol (committing to its input) and then has multiple other parties independently answering this message with messages encoding their input. The drawback of this approach in terms of performance is that the protocol requires an oblivious transfer for each input bit of Alice, and is therefore less efficient than protocols based on SPAKE2 or similar specific PAKE protocols.

3. PROBLEM STATEMENT

3.1 General Setting

The generic setting for cloud storage systems consists of a storage server ($\mathcal{S}$) and a set of clients ($\mathcal{C}$s) who store their

[2]This change is similar to the transformation from the (interactive) Diffie-Hellman key exchange protocol to the (non-interactive) ElGamal encryption: The latter can be considered as being generated from Diffie-Hellman key exchange, where Alice's public key is her first message.

Public information: A finite cyclic group G of prime order p generated by an element g. Public elements $M_u \in G$ associated with user u. A hash function H modeled as a random oracle.
Secret information: User u has a password pw_u.
The protocol is run between Alice and Bob:

1. Each side performs the following computation:
   - Alice chooses $x \in_R Z_p$ and computes $X = g^x$. She defines $X^* = X \cdot (M_A)^{pw_A}$.
   - Bob chooses $y \in_R Z_p$ and computes $Y = g^y$. He defines $Y^* = Y \cdot (M_B)^{pw_B}$.
2. Alice sends X^* to Bob. Bob sends Y^* to Alice.
3. Each side computes the shared key:
   - Alice computes $K_A = (Y^*/(M_B)^{pw_A})^x$. She then computes her output as $SK_A = H(A, B, X^*, Y^*, pw_A, K_A)$.
   - Bob computes $K_B = (X^*/(M_A)^{pw_B})^y$. He then computes his output as $SK_B = H(A, B, X^*, Y^*, pw_B, K_B)$.

Figure 2: The SPAKE2 protocol of Abdalla and Pointcheval [1].

files on $\mathcal{S}$. $\mathcal{C}$s never communicate directly, but exchange messages with $\mathcal{S}$, and $\mathcal{S}$ processes the messages and/or forwards them as needed. Additional independent servers ($\mathcal{IS}$s) can be introduced to assist deduplication [4, 28, 26]. But they are unrealistic in commercial settings[3] and can be bottlenecks for both security and performance. We do not require any $\mathcal{IS}$s to take part in our scheme.

We assume that the parties communicate through secure channels, so that an adversary ($\mathcal{A}$) cannot eavesdrop and/or tamper with any channel.

We introduce new notations as needed. A summary of notations appears in Appendix A.

3.2 Security Model

Ideal Functionality. We define the ideal functionality $\mathcal{F}_{dedup}$ of deduplicating encrypted data in Figure 4. There are three types of participants: the storage server $\mathcal{S}$, the *uploader* $\mathcal{C}$ attempting to upload a file and existing clients $\{\mathcal{C}_i\}$ who have already uploaded a file. A protocol implementing $\mathcal{F}_{dedup}$ is secure if it implements $\mathcal{F}_{dedup}$. Specifically, the protocol leaks no information about $\mathcal{C}$s' files and keys, and only $\mathcal{S}$ knows whether deduplication happens or not. Furthermore, we require the protocol to be secure in the malicious model [17] where participants can behave arbitrarily (are allowed to refuse to participate in the protocol, substitute their inputs with other values, and abort the protocol prematurely).

Inputs:

- The uploader $\mathcal{C}$ has input F;
- Each existing client $\mathcal{C}_i$ has inputs F_i and k_{F_i};
- $\mathcal{S}$'s input is empty.

Outputs:

- $\mathcal{C}$ gets an encryption key k_F for F. If F is identical to an existing file F_i then $k_F = k_{F_i}$. Otherwise k_F is random;
- Each $\mathcal{C}_i$'s output is empty;
- $\mathcal{S}$ gets the ciphertext $E(k_F, F)$. If there is a ciphertext $E(k_{F_j}, F_j)$ that is equal to $E(k_F, F)$ in its storage, it learns j as well. Otherwise, $\mathcal{S}$ learns that F is not in the storage.

Figure 4: The ideal functionality $\mathcal{F}_{dedup}$ of deduplicating encrypted data.

Threat Model. An adversary $\mathcal{A}$ might compromise $\mathcal{C}$, $\mathcal{S}$, any subset of $\{\mathcal{C}_i\}$, or any collusion of these parties. The security of a single upload procedure is captured by requiring that the protocol implements $\mathcal{F}_{dedup}$ according to the ideal/real model security definitions. However, additional attacks are possible when considering the long-term operation of the system:

- *Online brute-force attack by a compromised* active *uploader:* as we described in Section 2.1, for a predictable file, an uploader can construct all candidate files, upload them and observe which one causes deduplication;
- *Offline brute-force attack by a compromised* passive $\mathcal{S}$*:* if $\mathcal{S}$ gets a deterministic representation (e.g., cryptographic hash or convergent encryption) of a predictable file, it can construct all candidate files and verify them offline;
- *Online brute-force attack by a compromised* active $\mathcal{S}$*:* $\mathcal{S}$ can also masquerade as $\mathcal{C}$s running the protocol for every "guess" and checking if deduplication occurs.

Other than the brute-force attacks, a compromised $\mathcal{S}$ can easily detect whether a certain file is in the storage by running the deduplication protocol once. We claim that no known deduplication scheme can prevent this attack, as $\mathcal{S}$ always knows that deduplication happens. This attack is not included in our threat model.

3.3 Design Goals

Security goals. We define the following security goals for our scheme:

S1 Realize $\mathcal{F}_{dedup}$ in malicious model;

S2 Prevent online brute-force attacks by compromised active uploaders;

[3]It is difficult to find business justification for an independent party to run an $\mathcal{IS}$ solely for improving privacy in cloud storage services.

S3 Prevent offline brute-force attacks by compromised passive $\mathcal{S}$;

S4 Prevent online brute-force attacks by compromised active $\mathcal{S}$ (masquerading as multiple $\mathcal{C}$s).

Functional goals. In addition, the protocol should also meet certain functional goals:

F1 Maximize deduplication effectiveness (exceed realistic expectations, as discussed in Section 2.1);

F2 Minimize computational and communication overhead (i.e., the computation/communication costs should be comparable to storage systems without deduplication).

4. OVERVIEW OF THE SOLUTION

Overview. We first motivate salient design decisions in our scheme before describing the details in Section 5.

When an uploader $\mathcal{C}$ wants to upload a file F to $\mathcal{S}$, we need to address two problems: (a) determining if $\mathcal{S}$ already has an encrypted copy of F in its storage and (b) if so, securely arranging to have the encryption key transferred to $\mathcal{C}$ from some $\mathcal{C}_i$ who uploaded the original encrypted copy of F.

In traditional client-side deduplication, when $\mathcal{C}$ wants to upload F to $\mathcal{S}$, it first sends a cryptographic hash h of F to $\mathcal{S}$ so that it can check the existence of F in $\mathcal{S}$'s storage. Naïvely adapting this approach to the case of encrypted storage is insecure since a compromised $\mathcal{S}$ can easily mount an offline brute-force attack on the hash h if F is predictable. Therefore, instead of h, we let $\mathcal{C}$ send a short hash $sh = SH(F)$. Due to the high collision rate of $SH()$, $\mathcal{S}$ cannot use sh to reliably guess the content of F offline.

Now, suppose that another $\mathcal{C}_i$ previously uploaded $E(k_{F_i}, F_i)$ using k_{F_i} as the symmetric encryption key for F_i and that $sh = sh_i$. Our protocol needs to determine if this happened because $F = F_i$ and, in that case, arrange to have k_{F_i} securely transferred from $\mathcal{C}_i$ to $\mathcal{C}$. We do this by having $\mathcal{C}$ and $\mathcal{C}_i$ engage in an oblivious key sharing protocol which allows $\mathcal{C}$ to receive k_{F_i} iff $F = F_i$, and a random key otherwise. We say that $\mathcal{C}_i$ plays the role of a *checker* in this protocol.

The oblivious key sharing protocol could be implemented using generic solutions for secure two-party computation, such as versions of Yao's protocol [31], which express the desired functionality as a boolean circuit. Protocols of this type have been demonstrated to be very efficient, even with security against malicious adversaries. In our setting the circuit representation is actually quite compact, but the problem in using this approach is that the inputs of the parties are relatively long (say, 288 bits long, comprising of a full-length hash value and a key), and known protocols require an invocation of oblivious transfer, namely of public-key operations, for each input bit. There are known solutions for oblivious transfer extension, which use a preprocessing step to reduce the online computation time of oblivious transfer. However, in our setting the secure computation is run between two $\mathcal{C}$s that do not have any pre-existing relationship, and therefore preprocessing cannot be computed before the protocol is run.

Our solution for an efficient oblivious key sharing is having $\mathcal{C}$ and $\mathcal{C}_i$ run a PAKE protocol, using the hash values of their files, namely h and h_i, as their respective "passwords". The protocol results in $\mathcal{C}_i$ getting k_i and $\mathcal{C}$ getting k'_i, which are equal if $h = h_i$ and are independent otherwise. The next step of the protocol uses these keys to deliver a key k_F to $\mathcal{C}$, which is equal to k_{F_i} iff $k_i = k'_i$. $\mathcal{C}$ uses this key to encrypt its file, and $\mathcal{S}$ can deduplicate that file if the ciphertext is equal to the one uploaded by $\mathcal{C}_i$. Several additional issues need to be solved:

1. *How to prevent uploaders from learning about stored files?* Our protocol supports client-side deduplication, and as such informs $\mathcal{C}$ whether deduplication takes place. In order to solve the problem, we use the randomized threshold strategy of [19] (see Section 5.1).
2. *How to prevent a compromised $\mathcal{S}$ from mounting an online brute-force attack where it initiates many interactions with $\mathcal{C}/\mathcal{C}_i$ to identify F/F_i.* Each interaction essentially enables a single guess about the content of the target file. Clients therefore use a per-file rate limiting strategy to prevent such attacks. Specifically, they set a bound on the maximum number of PAKE protocols they would service as a checker or an uploader for each file. (See Section 5.2.) Our simulations with realistic datasets in Section 6 show that this rate limiting does not affect the deduplication effectiveness.
3. *What if there aren't enough checkers?* If $\mathcal{S}$ has a large number of clients, it is likely to find enough online checkers who have uploaded files with the required short hash. If there are not enough checkers, we can let the uploader run PAKE with the currently available checkers and with additional dummy checkers to hide the number of available checkers (See Section 5.3). Again, our experiments in Section 6 show that this does not affect the deduplication effectiveness (since the scheme is likely to find checkers for popular files).

Relaxing $\mathcal{F}_{dedup}$. The protocol we described implements the $\mathcal{F}_{dedup}$ functionality of Figure 4 with the following relaxations: (1) $\mathcal{S}$ learns a short hash of the uploaded file F (in our simulations we set the short hash to be 13 bits long). (2) F_{dedup} is not applied between the uploader and all existing clients, but rather between the uploader and clients which have uploaded files with the same short hash as F. Therefore these clients learn that a file with the same short hash is being uploaded.

We observe that in a large-scale system a short hash matches many files, and uploads of files with any specific short hash happen constantly. Therefore these relaxations leak limited information about the uploaded files. For example, since the short hash is random and short (and therefore matches many uploaded files), the rate with which a client who uploaded a file is asked to participate in the protocol is rather independent of whether the same file is uploaded again.

5. DEDUPLICATION PROTOCOL

In this section we describe our deduplication protocol in detail. The data structure maintained by $\mathcal{S}$ is shown in Figure 5. $\mathcal{C}$s who want to upload a file also upload the corresponding short hash sh. Since different files may have the same short hash, a short hash sh is associated with the list of different encrypted files whose plaintext maps to sh. $\mathcal{S}$ also keeps track of clients ($\mathcal{C}_1, \mathcal{C}_2, \ldots$) who have uploaded the same encrypted file.

Figure 6 shows the *basic* deduplication protocol. When $\mathcal{C}_i$ stores the $E(H_1(k_{F_i}), F_i)$ at $\mathcal{S}$, it also stores the short hash sh_i of F_i. (Step 0). Note that k_{F_i} is a full-length element in

0. For each previously uploaded encrypted file $E(H_1(k_{F_i}), F_i)$,[a] $\mathcal{S}$ also stores the corresponding short hash sh_i (which is $SH(F_i)$).

1. Before uploading a file F, the uploader $\mathcal{C}$ calculates both the cryptographic hash h (which is $H_2(F)$) and the short hash sh (which is $SH(F)$), and sends sh to $\mathcal{S}$.

2. $\mathcal{S}$ finds the checkers $\{\mathcal{C}_i\}$ who have uploaded files $\{F_i\}$ with the same short hash sh. Then it asks $\mathcal{C}$ to run the same-input-PAKE protocol with $\{\mathcal{C}_i\}$[b]. $\mathcal{C}$'s input is h and $\mathcal{C}_i$'s input is h_i.

3. After the invocation of the same-input-PAKE protocol, each $\mathcal{C}_i$ gets a session key k_i and $\mathcal{C}$ gets a set of session keys $\{k'_i\}$ corresponding to the different $\mathcal{C}_i$'s.[c]

4. Each $\mathcal{C}_i$ first uses a pseudorandom function to extend the length of k_i and then splits the result to $k_{iL}||k_{iR}$[d]. It sends k_{iL} and $(k_{F_i} + k_{iR})$ to $\mathcal{S}$.[e]

5. For each k'_i, $\mathcal{C}$ extends and splits it to $k'_{iL}||k'_{iR}$ in the same way as $\mathcal{C}_i$ does. Then it sends $\mathcal{S}$ its public key pk, $\{k'_{iL}\}$ and $\{Enc(pk, k'_{iR} + r)\}$ where r is a random element chosen by $\mathcal{C}$ from the plaintext group.

6. After receiving these messages from $\{\mathcal{C}_i\}$ and $\mathcal{C}$, $\mathcal{S}$ checks if there is an index j such that $k_{jL} = k'_{jL}$.

   (a) If so, $\mathcal{S}$ uses the homomorphic properties of the encryption to compute $e = Enc(pk, k_{F_j} + k_{jR}) \ominus Enc(pk, k'_{jR} + r) = Enc(pk, k_{F_j} - r)$, and sends e back to $\mathcal{C}$;

   (b) Otherwise it sends $e = Enc(pk, r')$, where r' is a random element chosen by $\mathcal{S}$ from the plaintext group.

7. $\mathcal{C}$ calculates $k_F = Dec(sk, e) + r$, and sends $E(H_1(k_F), F)$ to $\mathcal{S}$.

8. $\mathcal{S}$ deletes $E(H_1(k_F), F)$ if it is equal to a stored $E(H_1(k_{F_j}), F_j)$, and then allows $\mathcal{C}$ to access $E(H_1(k_{F_j}), F_j)$. Otherwise, $\mathcal{S}$ stores $E(H_1(k_F), F)$.

[a] k_{F_i} is a full-length element in the plaintext group of the additively homomorphic encryption scheme used by the protocol. We use a cryptographic hash function H_1 to hash down it to the length of the keys used by $E()$.
[b] All communication is run via $\mathcal{S}$. There is no direct interaction between $\mathcal{C}$ and any $\mathcal{C}_i$. $\mathcal{C}$'s input to the same-input-PAKE protocol was sent together with sh in Step 1.
[c] With overwhelming probability, $k_i = k'_i$ iff $F_i = F$.
[d] After extension, the result must be long enough to be divided into $k_{iL}||k_{iR}$ that satisfies: (1) k_{iL} is long enough so that the probability of two random instances of this key having the same value is small (namely, $|k_{iL}| >> \log N$ where N is the number of clients participating in the protocol); (2) k_{iR} is a full-length element in the plaintext group of the additively homomorphic encryption scheme.
[e] Addition is done in the plaintext group of the additively homomorphic encryption scheme.

Figure 6: The deduplication protocol.

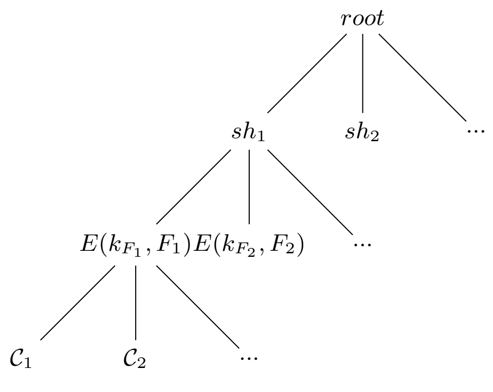


Figure 5: $\mathcal{S}$'s record structure.

the plaintext group of the additively homomorphic encryption scheme and we use a cryptographic hash function H_1 to hash down it to the length of the keys used by $E()$. When an uploader $\mathcal{C}$ wishes to upload a file F, it sends the short hash sh of this file to $\mathcal{S}$ (Step 1). $\mathcal{S}$ identifies the checkers $\{\mathcal{C}_i\}$ who have uploaded files with the same short hash (Step 2), and runs the following protocol with each of them.

Consider a specific $\mathcal{C}_i$ who has uploaded F_i. $\mathcal{C}$ runs a PAKE protocol with $\mathcal{C}_i$, where their inputs are the cryptographic hash values of F and F_i respectively, and their outputs are keys k'_i and k_i respectively (Step 3). The protocol must ensure that $\mathcal{C}$ uses the same input (hash value) in the PAKE instances that it runs with all $\{\mathcal{C}_i\}$. Therefore, we use a protocol implementing the same-input-PAKE functionality defined in Section 2.4. Both parties use a pseudorandom function to extend the output of the same-input-PAKE protocol.

Each $\mathcal{C}_i$ sends k_{iL} and $(k_{F_i} + k_{iR})$ to $\mathcal{S}$. $\mathcal{C}$ sends $\mathcal{S}$ the set of pairs $\{k'_{iL},\ Enc(pk, k'_{iR} + r)\}$, where r is chosen at random, the encryption is additively homomorphic and the private key is known to $\mathcal{C}$ (Steps 4-5). After receiving these messages from all $\mathcal{C}_i$s, $\mathcal{S}$ looks for a pair i for which $k_{iL} = k'_{iL}$. This equality happens iff $F_i = F$ (except with negligible probability). If $\mathcal{S}$ finds such a pair it sends $\mathcal{C}$ the value $e = Enc(pk, (k_{F_i} + k_{iR}) - (k'_{iR} + r)) = Enc(pk, k_{F_i} - r)$, computed using the homomorphic properties. Otherwise it sends $e = Enc(pk, r')$, where r' is chosen randomly by $\mathcal{S}$ (Step 6). $\mathcal{C}$ calculates $k_F = Dec(sk, e) + r$, and sends $E(H_1(k_F), F)$ to $\mathcal{S}$

(Step 7). If F was already uploaded to $\mathcal{S}$ then $E(H_1(k_F), F)$ is equal to the previously stored encrypted version of the file.

Note that after the PAKE protocol (in Step 3), a naïve solution would be to just have $\mathcal{C}_i$ send $E(k_i, k_{F_i})$ to $\mathcal{C}$. However, this would enable a subtle attack by $\mathcal{C}$ to identify whether F has been uploaded already.[4] Note also in Step 5 $\mathcal{C}$ uses r inside the encryption (rather than sending it in the clear) to prevent a malicious $\mathcal{S}$ from being able to set the value of k_F.

Implementation notes. We use the first option of implementing the same-input-PAKE protocol as described in Section 2.4. We use AES as the pseudorandom function used in Step 3. Most importantly, in order to improve the performance of the additively homomorphic encryption, we implement it using lifted ElGamal encryption as described in Section 2.3. The encryption is modulo a prime p of length 2048 bits (although encryption can also be done using ECC). As a result, k_F, k_{F_i}, k_{iR} and k'_{iR} are 2048 bits elements. This requires us to apply the following modifications to the protocol in our implementation:

- In Step 4, $\mathcal{C}_i$ sends $g^{(k_{F_i}+k_{iR})}$ instead of $(k_{F_i} + k_{iR})$;
- In Step 5, $\mathcal{C}$ sends $\mathcal{S}$ a lifted ElGamal encryption of $k'_{iR} + r$, i.e. an ElGamal encryption of $g^{k'_{iR}+r}$;
- In Step 6a, $Enc(pk, k_{F_j} + k_{jR})$ is a lifted encryption of $(k_{F_j} + k_{jR})$, i.e. an ElGamal encryption of $g^{(k_{F_j}+k_{jR})}$;
- Similarly, in Step 6b, $Enc(pk, r')$ is a lifted ElGamal encryption of r';
- In Step 7, $\mathcal{C}$ calculates $g^{k_F} = Dec(sk, e) \cdot g^r$, where $Dec(sk, e)$ is $g^{k_{F_j}-r}$ or $g^{r'}$. Then, $\mathcal{C}$ uses $H_1(g^{k_F})$ as the encryption key for F, and uses g^{k_F} as the input when it acts as a checker. Note that $\mathcal{C}$ knows nothing about k_F (similarly, $\mathcal{C}_i$ knows nothing about k_{F_i}).

Theorem 1. *The deduplication protocol in Figure 6 implements $\mathcal{F}_{dedup}$ with security against malicious adversaries, if the same-input-PAKE protocol is secure against malicious adversaries, the additively homomorphic encryption is semantically secure and the hash function H_2 is modeled as a random oracle.*

Proof. (sketch) We will show that the execution of the deduplication protocol in the real model is computationally indistinguishable from the execution of $\mathcal{F}_{dedup}$ in the ideal model. We construct a simulator that can both access $\mathcal{F}_{dedup}$ in ideal model and obtain messages that the corrupt parties would send in real model. The simulator will generate a message transcript of the ideal model execution (*IDEAL*) that is computationally indistinguishable from that of the real model execution (*REAL*). For the purpose of the proof we assume that the same-input-PAKE protocol is implemented as an oracle to which the parties send their inputs and receive their outputs.

[4]The problem with this approach is that $\mathcal{C}$ learns keys k_i for multiple other $\mathcal{C}$s, and should send $\mathcal{S}$ information about each key, $\mathcal{S}$ then tells $\mathcal{C}$ which key index to use (and chooses a random index if no match is found). A corrupt $\mathcal{C}$ might replace some keys with dummy values. If it is then told by $\mathcal{S}$ to use an index of one of these keys then it knows that no match was found. The protocol must therefore send back to $\mathcal{C}$ a key without specifying the index to which this key corresponds.

A corrupt uploader $\mathcal{C}$: We first assume that $\mathcal{S}$ and $\mathcal{C}_i$s are honest and construct a simulator for the uploader $\mathcal{C}$. The simulator operates as follows: it records the calls that $\mathcal{C}$ makes to the hash function H_2 (random oracle), and records tuples of the form $\{(F^j, h^j, sh^j)\}$. On receiving sh from $\mathcal{C}$, it observes the call that $\mathcal{C}$ makes to the same-input-PAKE protocol. If $\mathcal{C}$ uses in that call a value h that appears in a tuple together with sh, the simulator invokes $\mathcal{F}_{dedup}$ with the file F appearing in that tuple. Otherwise, it invokes $\mathcal{F}_{dedup}$ with a random value. In either case, the simulator will receive a key k_F. (If F has been uploaded by any $\mathcal{C}_j$, k_F is the key k_{F_j} for that file, otherwise k_F is random.)

The simulator also records the output set $\{k'_i\}$ that $\mathcal{C}$ receives from the same-input-PAKE protocol. On receiving a set of pairs $\{(k'_{iL}, Enc(pk, k'_{iR} + r))\}$ from $\mathcal{C}$, it chooses a random index l, and checks if k'_{lL} is equal to the left part of k'_l. If so, it calculates $e' = Enc(pk, k_F - r)$ and sends it back to $\mathcal{C}$; otherwise it sends $e' = Enc(pk, r')$ where r' is a random value.

We now show that $IDEAL_{\mathcal{C}} = \langle\{k'_i\}, e'\rangle$ and $REAL_{\mathcal{C}} = \langle\{k'_i\}, e\rangle$ are identically distributed. (1) If F exists and $\mathcal{C}$ behaves honestly, then $k_F = k_{F_j}$ and consequently e' is equal to e (Step 6a). (2) If F does not exist, then k_F is a random value. As a result, e' is an encryption of a random value like e (Step 6b). (3) If $\mathcal{C}$ deviates from the protocol then the only action it can take, except for changing its input, it to replace some elements of $\{k'_{iL}, Enc(pk, k'_{iR} + r)\}$ that it sends to $\mathcal{S}$. We assume the size of this set is N and x elements are replaced by $\mathcal{C}$. In the real model, the execution will change if there is an index j such that $k_{jL} = k'_{jL}$ but k'_{jL} (and/or $Enc(pk, k'_{jR} + r)$) is replaced by $\mathcal{C}$. As a result, $\mathcal{C}$ will get a random value even though it inputs an existing file. The probability for this event is $\frac{x}{N}$. In the ideal model, the same result will be caused by the event that a replaced element is chosen by the simulator. The probability for this event is also $\frac{x}{N}$. Based on (1) (2) (3), we can conclude that $IDEAL_{\mathcal{C}}$ and $REAL_{\mathcal{C}}$ are identically distributed.

A corrupt checker $\mathcal{C}_i$: We prove security with relation to the relaxed functionality, where C_i also learns whether the uploaded file has the same short hash as F_i. The simulator needs to extract $\mathcal{C}_i$'s inputs: F_i and k_{F_i}, but since $\mathcal{C}_i$ has previously uploaded $E(H_1(k_{F_i}), F_i)$, it only needs to extract k_{F_i}.

The simulator first observes whether sh matches the short hash of $\mathcal{C}_i$. If not, it provides a random k_{F_i} to $\mathcal{F}_{dedup}$. Otherwise, the simulator observes $\mathcal{C}_i$'s input h_i to the same-input-PAKE protocol, its output k_i, and the message (α, β) that $\mathcal{C}_i$ sends to $\mathcal{S}$ (if $\mathcal{C}_i$ is honest then (α, β) is equal to $(k_{iL}, k_{iR} + k_{F_i})$). The simulator checks if α is equal to k_{iL}. If so, it extracts k_{F_i} as $k_{F_i} = \beta - k_{iR}$ and sends it to $\mathcal{F}_{dedup}$. Otherwise, it sends a random k_{F_i}.

To show that the simulation is accurate, observe that (1) If $\mathcal{C}_i$ behaves honestly, then $IDEAL_{\mathcal{C}_i}$ and $REAL_{\mathcal{C}_i}$ are obviously indistinguishable. (2) If $\mathcal{C}_i$ deviates from the protocol, then the only operation it can do is to send wrong values of (α, β). If $\alpha \neq k_{iL}$ then in both the real and ideal executions k_F is assigned a random value. If $\alpha = k_{iL}$ then in both the real ideal executions k_F is set to be $\beta - k_{iR}$. Based on (1) (2), we can conclude that $IDEAL_{\mathcal{C}_i}$ and $REAL_{\mathcal{C}_i}$ are identically distributed.

A corrupt server $\mathcal{S}$: We prove security with relation to the relaxed functionality. The simulator first sends a short hash sh to $\mathcal{S}$. $\mathcal{S}$ selects a set of clients to participate in the same-input-PAKE protocol. Let $\{\mathcal{C}_i\}$ be the set of clients who have uploaded to $\mathcal{S}$ files with the same short hash. $\mathcal{C}_i$ is marked as "OK" if it is selected by $\mathcal{S}$ as a checker, i.e., $\mathcal{S}$ has sent it a request to participate in the same-input-PAKE protocol, and forwarded its reply to the uploader (if $\mathcal{S}$ is honest then all clients in $\{\mathcal{C}_i\}$ are "OK"). The simulator then pretends to be each selected $\mathcal{C}_i$, sending $\mathcal{S}$ random values for $(k_{iL}, k_{iR} + k_{F_i})$.

Next, the simulator invokes $\mathcal{F}_{dedup}$. It sets a bit $b = 1$ if it receives $E(H_1(k_F), F)$ and an index j from $\mathcal{F}_{dedup}$ (in the case of a file match); otherwise, it sets $b = 0$. If $b = 1$ but client $\mathcal{C}_j$ is not "OK", the simulator changes b to 0. It then pretends to be the uploader $\mathcal{C}$. If $b = 1$, it sets the message with respect to $\mathcal{C}_j$ to be $(k_{jL}, Enc(pk, x_j))$ (where k_{jL} is the same as before and x_j is random), and random values for the other pairs. Otherwise, it sets random values for all pairs.

The simulator should now receive an encryption $Enc(y)$ from $\mathcal{S}$. If $b = 0$, it sends $\widehat{F} = E(H_1(k_F), F)$ (that was received from $\mathcal{F}_{dedup}$) to $\mathcal{S}$. If $b = 1$, it decrypts $Enc(y)$ and checks if $y = (k_{jR} + k_{F_j}) - x_j$. If so, it sends $\widehat{F} = E(H_1(k_F), F)$ and the index j to $\mathcal{S}$. Otherwise, it sends to $\mathcal{S}$ a random string $\widehat{F}$ of the same length with $E(H_1(k_F), F)$.

We now show that $IDEAL_{\mathcal{S}} = \langle\ sh, \{(k_{iL}, k_{iR} + k_{F_i}\}, \{(k'_{iL}, Enc(pk, x_j))\}, \widehat{F}\ \rangle$ and $REAL_{\mathcal{S}} = \langle\ sh, \{(k_{iL}, k_{iR} + k_{F_i})\}, \{(k'_{iL}, Enc(pk, k'_{iR} + r_i))\}, E(H_1(k_F), F)\ \rangle$ are identically distributed. (1) Since $\{k_i, k'_i\}$ are random keys output by the PAKE protocol, $\mathcal{S}$ cannot distinguish $\{(k_{iL}, k_{iR}+k_{F_i}\}$ and $\{(k'_{iL}, Enc(pk, x_j))\}$ in the ideal model from those in real model. (2) If F exists and $\mathcal{S}$ behaves honestly, it will get the same $E(H_1(k_F), F)$ and j in the real model and the ideal model. (3) If F does not exist, then k_F is a random key in the real model, and consequently $\mathcal{S}$ cannot distinguish $E(H_1(k_F), F)$ from a random string. This also happens in the ideal model: what the simulator gets from $\mathcal{F}_{dedup}$ is an encryption of F under a random key. (4) If $\mathcal{S}$ deviates from the protocol, it can choose to select a subset of the $\{\mathcal{C}_i\}$ and/or send a wrong value of $Enc(y)$. Assume that the size of $\{\mathcal{C}_i\}$ is M and that z clients are not chosen by $\mathcal{S}$. Deduplication will fail if the owner of F is not chosen by $\mathcal{S}$, which happens in both the real and the ideal model with the same probability ($\frac{z}{M}$). (5) If $\mathcal{S}$ sends a wrong value of $Enc(y)$, $\mathcal{C}$ will obtain a random key in the real model, and it will send $Enc(k_F, F)$ that is indistinguishable from a random string (like $\widehat{F}$ in the ideal model). Based on (1)-(5) we can conclude that $IDEAL_{\mathcal{S}}$ and $REAL_{\mathcal{S}}$ are identically distributed.

A collusion between a corrupt uploader and a corrupt server: The simulator in this case can invoke $\mathcal{F}_{dedup}$ once, pretending to be both $\mathcal{C}$ and $\mathcal{S}$, and providing $\mathcal{C}$'s input F ($\mathcal{S}$ has no input to $\mathcal{F}_{dedup}$). It then receives the outputs of both parties, namely the key k_F, $E(H_1(k_F), F)$, and an index j (if there is a file match).

The simulation is similar to the case of a corrupted uploader, except that $\mathcal{S}$ might choose a subset of $\{\mathcal{C}_i\}$ to run the same-input-PAKE protocol. Therefore, the simulation begins as the proof of a corrupt $\mathcal{C}$ and the simulator extracts $\mathcal{C}$'s input F from the random oracle. Then it invokes $\mathcal{F}_{dedup}$ with input F. If a match was found, the simulator observes the operation of $\mathcal{S}$ and checks if $\mathcal{C}_j$ is "OK" (as was defined in the proof for a corrupt $\mathcal{S}$). If so, it uses (F, k_F) as $\mathcal{C}_j$'s input and random values for other checkers' inputs.

A collusion between corrupt $\mathcal{C}_i$s and a corrupt $\mathcal{S}$: The simulation is similar to the case of a corrupted $\mathcal{S}$, except that the simulator does not need to pretend to be checkers sending messages to $\mathcal{S}$. Instead, it can get these messages from the same-input-PAKE protocol, which will be invoked by each corrupt checker. The rest of the simulation is exactly the same as in the case of a corrupt $\mathcal{S}$. □

Based on Theorem 1, we conclude that compromised parties cannot run the computation in a way that is different than is defined by (relaxed) $\mathcal{F}_{dedup}$. This makes our scheme satisfy the requirement **S1** and **S3**. We now discuss several extensions to the basic protocol to account for the types of issues we alluded to in Section 4.

5.1 Randomized Threshold

The protocol in Figure 6 is for server-side deduplication. To save bandwidth, we transform it to support client-side deduplication. In order to satisfy requirement **S2** and protect against a corrupt uploader, we use the randomized threshold approach from Harnik et al. [19]: for each file F, $\mathcal{S}$ maintains a random threshold t_F ($t_F \geq 2$), and a counter c_F that indicates the number of $\mathcal{C}$s that have previously uploaded F.

In step 6 of the deduplication protocol,

- In the case of a match (6a), if $c_{F_i} < t_{F_i}$, $\mathcal{S}$ tells $\mathcal{C}$ to upload $E(k_F, F)$ as if no match occurred (but $\mathcal{S}$ does not store this copy). Otherwise, $\mathcal{S}$ informs $\mathcal{C}$ that the file is duplicated and there is no need to upload it;
- In the case of a no match (6b), $\mathcal{S}$ asks $\mathcal{C}$ to upload $E(k_F, F)$.

5.2 Rate Limiting

A compromised active $\mathcal{S}$ can apply online brute-force attacks against $\mathcal{C}$ or $\mathcal{C}_i$. Specifically, if F_i is predictable, $\mathcal{S}$ can pretend to be an uploader sending PAKE requests to $\mathcal{C}_i$ to guess F_i. $\mathcal{S}$ can also pretend to be a checker sending PAKE responses to $\mathcal{C}$ to guess F. Therefore both uploaders and checkers should limit the number of PAKE runs for each file in their respective roles. This per-file rate limiting strategy can both improve security (see below) and reduce overhead (namely the number of PAKE runs) without damaging the deduplication effectiveness (as shown in Section 6).

We use RL_c to denote the rate limit for checkers, i.e., each $\mathcal{C}_i$ can process at most RL_c PAKE requests for F_i and will ignore further requests. Similarly, RL_u is the rate limit for uploaders, i.e., a $\mathcal{C}$ will send at most RL_u PAKE requests to upload F. Suppose that n is the length of the short hash and m is the min-entropy of a predictable file F, and x is the number of clients who potentially hold F. Our scheme can prevent online brute-force attacks from $\mathcal{S}$ if

$$2^m > 2^n \cdot x \cdot (RL_u + RL_c) \qquad (1)$$

because $\mathcal{S}$ can run PAKE with all owners of F to confirm its guesses. So the uncertainty in a guessable file (2^{m-n}) must be larger than x times per-file rate limit ($RL_u + RL_c$).

As a comparison, DupLESS [4] uses a *per-client* rate limiting strategy to prevent such online brute-force attacks from any single $\mathcal{C}$. The rate limit must still support a client $\mathcal{C}$ that needs to legitimately upload a large number of files within a brief time interval (such as backing up a local file system). Therefore the authors of DupLESS chose a large bound (825 000) for the total number of requests a single $\mathcal{C}$

can make during one week. The condition for DupLESS to resist online brute-force attack by a single $\mathcal{C}$ is

$$2^m > y \cdot RL \tag{2}$$

where y is the number of compromised clients and RL is the rate limit (825 000/week) for each client. Recall also, that a compromised active $\mathcal{S}$ can masquerade as any number of $\mathcal{C}$s that is needed, and therefore y could be as large as possible. Consequently, DupLESS cannot fully deal with an online-brute force attack by a compromised $\mathcal{S}$.

To prevent a compromised active $\mathcal{S}$ from masquerading multiple $\mathcal{C}$s, the authors of [28] introduce another independent party called *identity server*. When $\mathcal{C}$s first join the system, the identity server is responsible for verifying their identity and issuing credentials to them. However, it is hard to deploy an *independent* identity server in a real world setting. As far as we know, our scheme is the first deduplication protocol that can prevent online brute-force attacks (satisfying requirement **S4**) without the aid of an identity server.

5.3 Checker Selection

If an uploader is required to run only a few (or none) PAKE instances, due to no short hash matches, it will learn that it is less likely that its file is already in $\mathcal{S}$'s storage. To avoid this leakage of information, $\mathcal{S}$ fixes the number of PAKE runs (i.e., RL_u) for an upload request to be constant. For a requested short hash sh, checkers are selected according to the following procedure:

1. $\mathcal{S}$ selects the most popular file among the files whose short hash is sh and which were not already selected with respect for the current upload (popularity is measured in terms of the number of $\mathcal{C}$s who own the file).

2. $\mathcal{S}$ selects a checker for that file in ascending order of engagement among the $\mathcal{C}$s that are currently online (in terms of the number of PAKE requests they have serviced so far for that specific file).

3. If the number of selected files is less than RL_u, repeat Step 1-3.

4. If the total number of selected files for which there are online clients is smaller than RL_u, $\mathcal{S}$ uses additional dummy files and clients, until reaching RL_u files.

Then, $\mathcal{S}$ lets the uploader run PAKE instances with the selected RL_u clients ($\mathcal{S}$ itself runs as dummy clients).

6. SIMULATION

Our use of rate limiting can impact deduplication effectiveness. In this section, we use realistic simulations to study the effect of various parameter choices in our protocol on deduplication effectiveness.

Datasets. We want to consider two types of storage environments. The first consists predominantly of media files, such as audio and video files from many users. We did not have access to such a dataset. Instead, we use a dataset comprising of Android application prevalence data to represent an environment with media files. This is based on the assumption that the popularity of Android applications is likely to be similar to that of media files: both are created and published by a small number of authors (artists/developers), made available on online stores or other distribution sites, and are acquired by consumers either for free or for a fee. We call this the *media dataset*. We use a publicly available dataset[5]. It consists of data collected from 77 782 Android devices. For each device, the dataset identifies the set of (anonymized) application identifiers found on that device. We treat each application identifier as a "file" and consider the presence of an app on a device as an "upload request" to add the corresponding "file" to the storage. This dataset has 7 396 235 "upload requests" in total, of which 178 396 are for distinct files.

The second is the type of storage environments that are found in enterprise backup systems. We use data gathered by the Debian Popularity Contest[6] to approximate such an environment. The *popularity-contest* package on a Debian device regularly reports the list of packages installed on that device. The resulting data consists of a list of debian packages along with the number of devices which reported that package. We took a snapshot of this data on Nov 27, 2014. It consists of data collected from 175 903 Debian users. From this data we generated our *enterprise dataset*: it has 217 927 332 "upload requests" (debian package installations) of which 143 949 are for distinict files (unique packages).

Figure 7 shows the file popularity distribution (i.e., the number of upload requests for each file) in logarithmic scale for both datasets. We map each dataset to a stream of upload requests by generating the requests in random order, where a file that has x copies generates x upload requests at random time intervals.

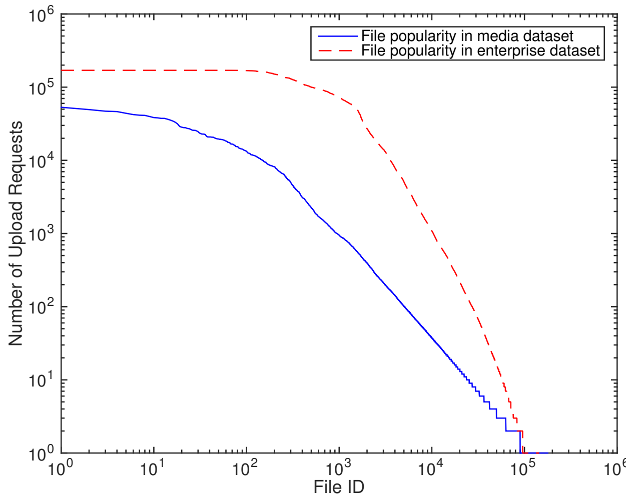


Figure 7: File popularity in both datasets.

Parameters. To facilitate comparison with DupLESS [4], we set the min-entropy to log(825000). We then set the length of the short hash $n = 13$, and $(RL_u + RL_c) = 100$ (i.e., a $\mathcal{C}$ will run PAKE at most 100 times for a certain file as both uploader and checker), so that we achieve the bound in inequality 2 in Section 5.2: $\mathcal{A}$ cannot uniquely identify a file within the rate limit. We use these parameters in our simulations.

[5]`https://se-sy.org/projects/malware/`
[6]`http://popcon.debian.org`

We measure overhead as the average number of PAKE runs[7], which can be calculated as:

$$\mu = \frac{Total\ number\ of\ \text{PAKE}\ runs}{Total\ number\ of\ upload\ requests} \quad (3)$$

We measure deduplication effectiveness using the deduplication percentage (Section 2.1). We assume that all files are of equal size so that the deduplication percentage ρ is:

$$\rho = (1 - \frac{Number\ of\ all\ files\ in\ storage}{Total\ number\ of\ upload\ requests}) \cdot 100\% \quad (4)$$

Rate limiting. We first assume that all $\mathcal{C}$s are online during the simulation, and study the impact of rate limits. Having selected RL_u+RL_c to be 100, we now see how selecting specific values for RL_u and RL_c affects the average number of PAKE runs and the deduplication effectiveness. Figure 8 shows the average number of PAKE runs resulting from different values of RL_u (and hence RL_c) in both datasets. Both values are very low, in the range 1.3-1.75. We also ran the simulation without any rate limits, which led to an average of 26.88 PAKE runs in the media dataset and 13.19 PAKE runs in the enterprise dataset. These numbers are significantly larger than the results with rate limiting. Figure 9 shows ρ resulting from different rate limit choices in both datasets. We see that setting $RL_u = 30$ (and hence setting $RL_c = 70$), maximizes ρ to be (97.58% and 99.9332%, respectively. These values are extremely close to the perfect deduplication percentages in both datasets (97.59% and 99.9339% respectively). A major conclusion is that rate limiting can improve security and reduce overhead without negatively impacting deduplication effectiveness.

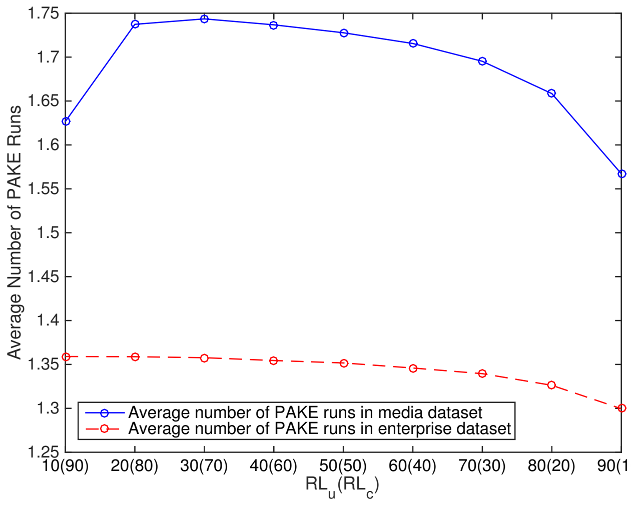


Figure 8: Average number of PAKE runs VS. rate limits.

Offline rate. The possibility of some $\mathcal{C}$s being offline may adversely impact deduplication effectiveness. To estimate this impact, we assign an *offline rate* to each $\mathcal{C}$ as its probability to be offline during one upload request. Using the chosen rate limits ($RL_u = 30$ and $RL_c = 70$), we measured ρ by varying the offline rate. The results for both datasets are shown in Figure 10. It shows that ρ is still reasonably high if the offline rate is lower than 70%. But drops quickly beyond that. We can solve this by introducing *deferred check*. Specifically, we split RL_u to $RL_{u1} + RL_{u2}$. $\mathcal{S}$ will let the uploader run RL_{u1} times PAKE before uploading, and later ask it to run further RL_{u2} PAKE instances when some $\mathcal{C}$s who are previously offline, come online. If $\mathcal{S}$ finds a match after uploading, it checks the counter and random threshold for the matched file. If the counter has exceeded the threshold, $\mathcal{S}$ deletes the previously uploaded file and asks the uploader to change the encryption key to match the detected duplicate. The only issue for this solution is that the uploader needs to keep the randomness of all PAKE runs of offline check. Otherwise, $\mathcal{S}$ keeps the messages for that PAKE instance until the threshold being crossed. Figure 11 shows that this method can significantly improve the deduplication effectiveness when offline rate is high.

Evolution of deduplication effectiveness. Figure 12

[7]We do not include fake PAKE runs by $\mathcal{S}$ (Section 5.3) since we are interested in estimating the average number of real PAKE runs.

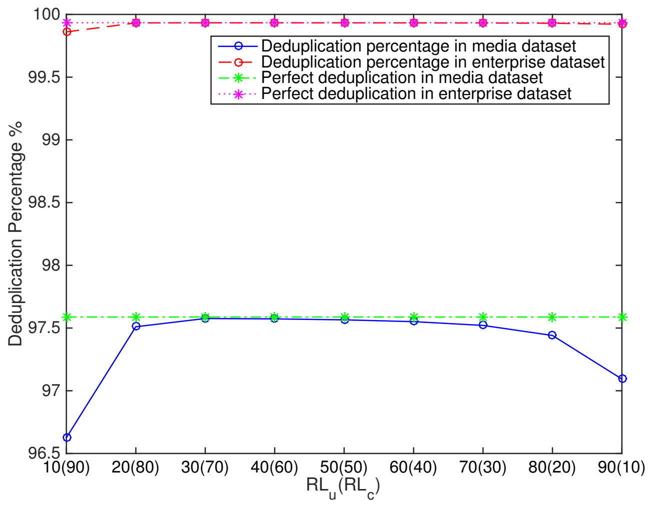


Figure 9: Dedup. percentage VS. rate limits.

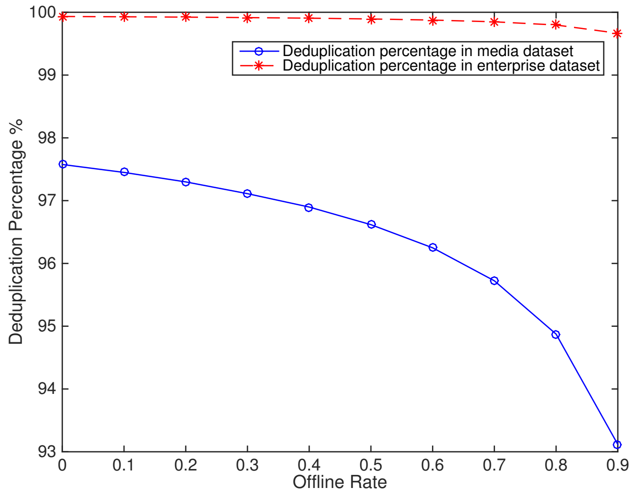


Figure 10: Dedup. percentage VS. offline rates.

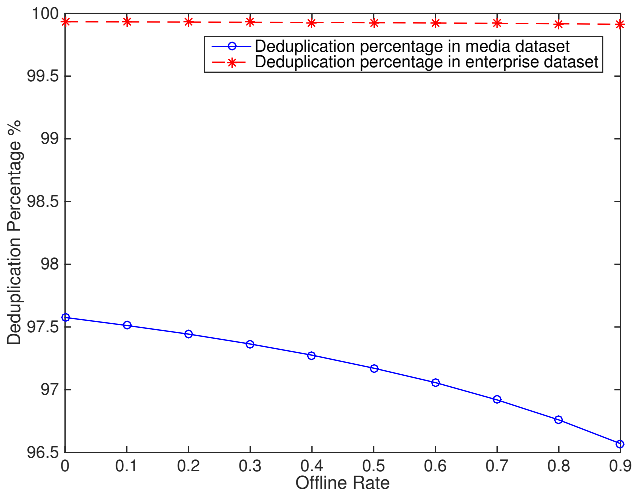


Figure 11: Dedup. percentage VS. offline rates.

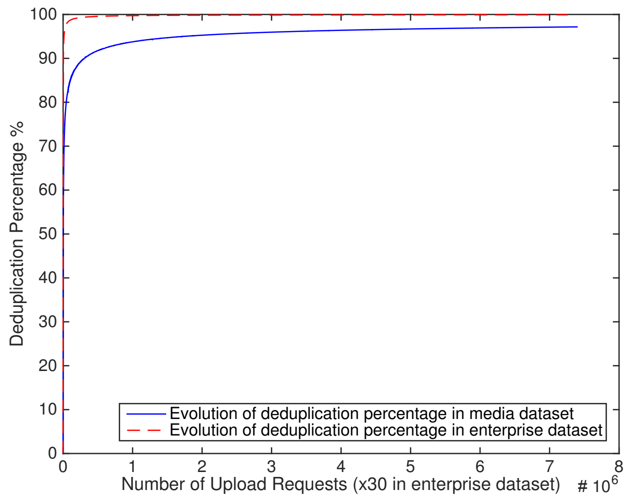


Figure 12: Dedup. percentage VS. number of upload requests.

shows that the ρ achieved by our scheme increases as more files are added to the storage, and it meets the realistic expectation (95%) quickly: after receiving 304 160 (4%) upload requests in the media dataset, and 121 110 (0.05%) upload requests in the enterprise dataset. Given that the deduplication effectiveness of our scheme is close to that of perfect deduplication and exceeds typical expected values, we can conclude that it satisfies functionality goal **F1**. Using rate limits implies that ρ increases more slowly in our scheme than in perfect deduplication. Figure 13 shows that this difference stablizes as the number of upload requests increases.

Figure 16 (in Appendix B) shows the second order difference in deduplication effectiveness compared to perfect deduplication. The second order difference vanishes as more files are uploaded to the storage.

Explanation. The fact that our scheme achieves close to perfect deduplication even in the presence of rate limits may appear counter-intuitive at first glance. But this phenomenon can be explained by *Zipf's law* [32]. As seen from Figure 7, beyond the initial plateau, the file popularity distribution is a straight line and thus follows a *power law distribution* (also known as *Zipf distribution*). The initial plateau does not impact our system. This is evident when we account for the use of short hash function. Figure 14 shows the file popularity in both datasets for some specific, but randomly selected, short hash values (of length 10).

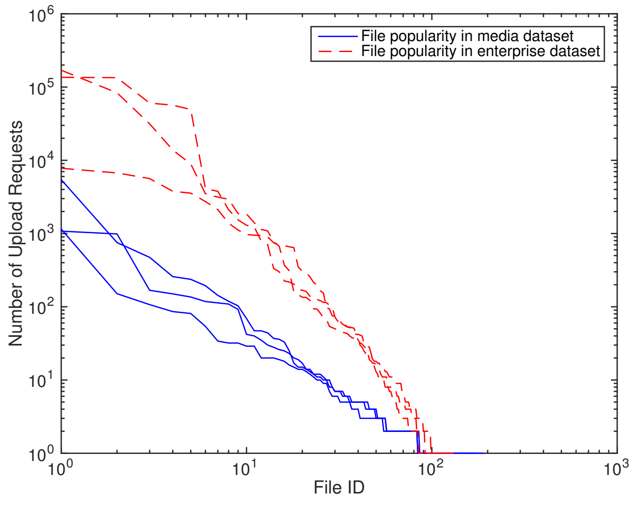


Figure 14: File popularity for six short hashes.

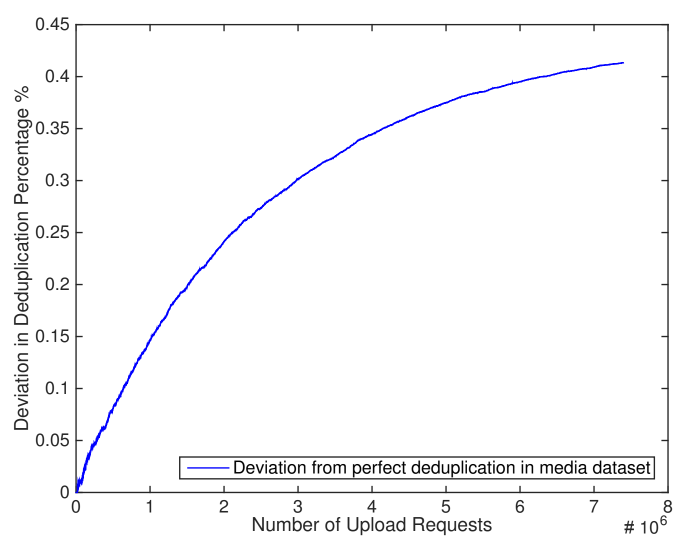
Even though we use rate limits, $\mathcal{S}$ always selects files based on descending order of popularity (step 1 in Section 5.3). Since file popularity follows the Zipf distribution, selecting files based on popularity ensures that popular uploaded files have a much higher likelihood of being selected and thus deduplicated. There are other examples of using the Zipf distribution to design surprisingly efficient systems. *Web proxy caching proxies* are such an example [9]. Breslau et al. observe that the distribution of page requests follows Zipf's law. Consequently, proxies use their limited storage to only cache popular files but still achieve significant bandwidth savings. The frequency of a request for the m^{th} most popular page can be calculated as $\frac{1/m^{\alpha}}{\sum_{i=1}^{N}(1/i^{\alpha})}$, where N is the size of the cache, and α is the value of the exponent characterising the distribution[9]. As a result, most of the requested pages can be found in the cache. Similarly, in our case, most of the upload requests for files that have already been uploaded can find a matched file within the rate limit.

7. PERFORMANCE EVALUATION

Our deduplication scheme incurs some extra computation and communication due to the number of PAKE runs. In this section, we demonstrate that the overhead is negligible for large files by implementing a proof-of-concept prototype.

Prototype. Our prototype consists of two parts: (1) a server program which simulates $\mathcal{S}$ and (2) a client program which simulates $\mathcal{C}$ (performing file uploading/downloading, encryption/decryption, and assisting $\mathcal{S}$ in deduplication). We used *Node.js*[8] for the implementation of both parties,

[8]`http://nodejs.org`



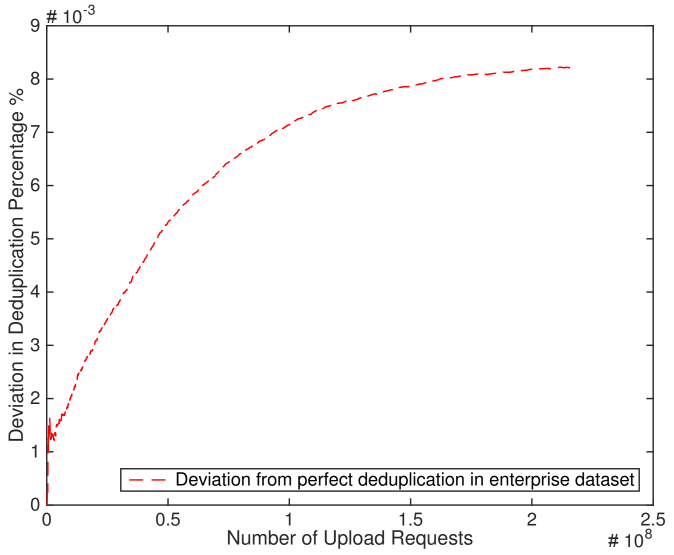


Figure 13: Deviation from perfect deduplication VS. Number of upload requests.

and *Redis*[9] for the implementation of $\mathcal{S}$'s data structure. We used SHA-256 as the cryptographic hash function and AES with 256-bit keys as the symmetric encryption scheme, both of which are provided by the *Crypto* module in *Node.js*. We used the *GNU multiple precision* arithmetic library[10] to implement the public key operations. The additively homomorphic encryption scheme used in the protocol is ElGamal encryption where the plaintext is encoded in the exponent, see Section 5.

Test setting and methodology. We ran the server-side program on a remote server (Intel Xeon with 4 2.66 GHz cores) and the client-side program on an Intel Core i7 machine with 4 2.2 GHz cores. We measured the running time using the *Date* module in Javascript and measured the bandwidth usage using *TCPdump*..

As the downloading phase in our protocol is simply downloading an encrypted file, we only consider the uploading phase. We set the length of short hash to be 13, and set $RL_u = 30$. We considered the case where the uploader $\mathcal{C}$ runs PAKE with 30 checkers $\mathcal{C}_i$. So we simulate the uploading phase in our protocol as:

1. $\mathcal{C}$ sends the short hash of the file it wants to upload to the server $\mathcal{S}$;
2. $\mathcal{S}$ forwards requests to 30 checkers $\mathcal{C}_i$ and lets them run PAKE with $\mathcal{C}$;
3. $\mathcal{S}$ waits for responses in all instances back from $\mathcal{C}$ and $\{\mathcal{C}_i\}$;
4. $\mathcal{S}$ chooses one instance and sends the result to $\mathcal{C}$;
5. $\mathcal{C}$ uses the resulting key to encrypt its file with AES and uploads it to $\mathcal{S}$.

We measured both running time and bandwidth usage during the whole procedure above. Network delay for all parties was included in the final results. We compare the results to two baselines: (1) uploading without encryption and (2) uploading with AES encryption. As in [4], we repeat our experiment using files of size 2^{2i} KB for $i \in \{0, 1, ..., 8\}$, which provides a file size range of 1KB to 64 MB. For each file, we upload it 100 times and calculate the mean. For files that are larger than the computer buffer, we do loading, encryption and uploading at the same time by pipelining the data stream. As a result, uploading encrypted files uses almost the same amount of time as uploading plain files.

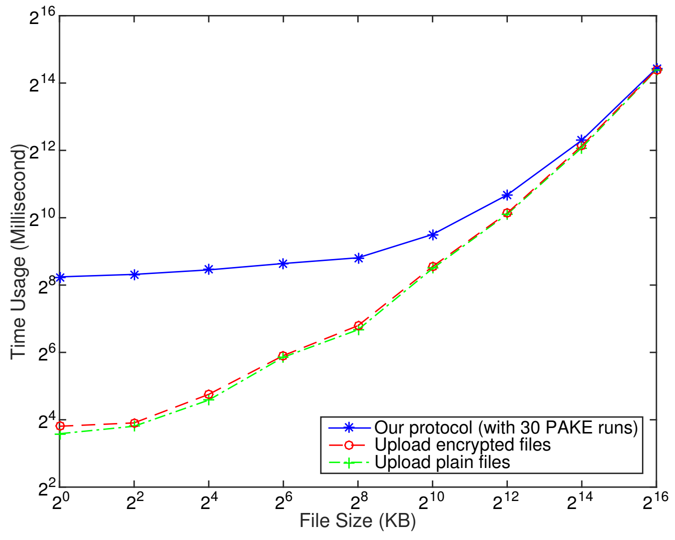
Results. Figure 15 reports the uploading time and bandwidth usage in our protocol compared to the two baselines. For files that are smaller than 1 MB, the overhead introduced by our deduplication protocol is relatively high. For example, it takes 15 ms (2 508 bytes) to upload a 1 KB encrypted file, while it takes 319 ms (145 359 bytes) to upload the same file in our protocol. However, the overhead introduced by our protocol is independent of the file size (about 10 ms for each PAKE run), and becomes negligible when the file is large enough. For files that are larger than 64 MB file, the time overhead is below 2%, and the bandwidth overhead is below 0.16%. So our scheme meets **F2**.

8. RELATED WORK

There are several types of schemes that enable deduplication with client-side encrypted data. The simplest approach (which is used by most commercial products) is to encrypt $\mathcal{C}$s' files using a global key which is encoded in the client-side software. As a result, different copies of F result in the same ciphertext and can therefore be deduplicated. This approach is, of course, insecure if $\mathcal{S}$ is untrusted.

Another approach is *convergent encryption* [13], which uses $H(F)$ as a key to encrypt F, where $H()$ is a publicly known cryptographic hash function. This approach ensures that different copies of F result in the same ciphertext. However, a compromised passive $\mathcal{S}$ can perform an offline brute-force attack if F has a small (or medium) entropy. Bellare et al. proposed *message-locked encryption* (MLE), which uses a semantically secure encryption scheme but produces a deterministic tag [5]. So it still suffers from the same attack.

[9]`http://redis.io`
[10]`https://gmplib.org`



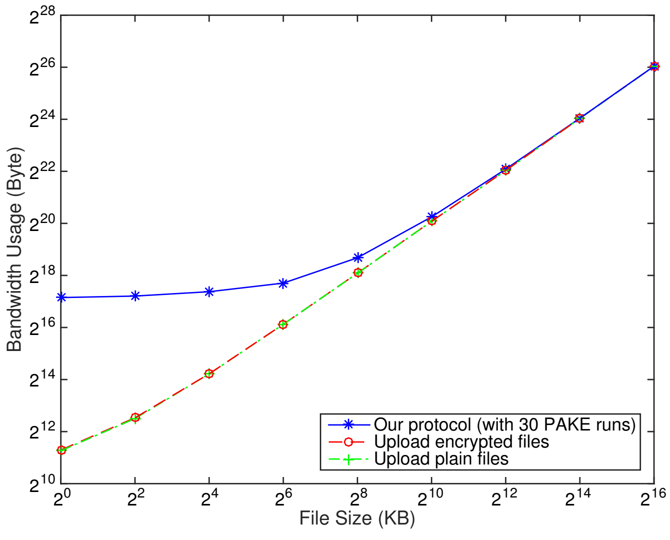


Figure 15: Time (left) and bandwidth usage (right) VS. file size.

Other solutions are based on the aid of additional independent servers ($\mathcal{IS}$s). For example, *Cloudedup* is a deduplication system that introduces an $\mathcal{IS}$ for encryption and decryption [26]. Specifically, $\mathcal{C}$ first encrypts each block with convergent encryption and sends the ciphertexts to $\mathcal{IS}$, who then encrypts them again with a key only known by itself. During file retrieval, blocks are first decrypted by $\mathcal{IS}$ and sent back to $\mathcal{C}$. In this scheme, a compromised active $\mathcal{S}$ can easily perform an online brute-force attack by uploading guessing files and see if deduplication happens.

Stanek et al. propose a scheme that only deduplicates popular files [28]. $\mathcal{C}$s encrypt their files with two layers of encryption: the inner layer is obtained through convergent encryption, and the outer layer is obtained through a semantically secure threshold encryption scheme with the aid of an $\mathcal{IS}$. $\mathcal{S}$ can decrypt the outer layer of F iff the number of $\mathcal{C}$s who have uploaded F reaches the threshold, and thus perform a deduplication. In addition, they introduce another $\mathcal{IS}$ as an identity server to prevent online brute-force attacks by multiple compromised $\mathcal{C}$s.

Both [26] and [28] are vulnerable to offline brute-force attacks by compromised $\mathcal{IS}$s. To prevent this, Bellare et al. propose DupLESS that enables $\mathcal{C}$s to generate file keys by running an *oblivious pseudorandom function* (OPRF) with $\mathcal{IS}$. Specifically, in the key generation process of convergent encryption, they introduce another secret which is provided by $\mathcal{IS}$ and identical for all $\mathcal{C}$s. The OPRF enables $\mathcal{C}$s to generate their keys without revealing their files to $\mathcal{IS}$, and without learning anything about $\mathcal{IS}$'s secret. To prevent the online brute-force attacks from compromised active $\mathcal{S}$. DupLESS uses a per-client rate limiting strategy to limit the number of requests that a $\mathcal{C}$ can send to $\mathcal{IS}$ during each epoch. We have identified the limitations for this strategy in Section 5.2. In addition, if $\mathcal{A}$ compromises both $\mathcal{S}$ and $\mathcal{IS}$, it can get the secret from $\mathcal{IS}$, and the scheme is reduced to normal convergent encryption.

Duan proposes a scheme that uses the same idea as DupLESS, but distributes the task of $\mathcal{IS}$ [14], where a $\mathcal{C}$ must interact with a threshold of other $\mathcal{C}$s to generate the key. So this scheme is only suitable for peer-to-peer paradigm: a threshold number of $\mathcal{C}$s must be online and interact with one another. While improving availability and security compared to DupLESS, this scheme is still susceptible to online brute-force attacks by compromised active $\mathcal{S}$, and it is unclear how to apply any rate-limiting strategy to it.

In Table 1, we summarize the resilience of these schemes with respect to the design goals from Section 3.3.

Threat / Schemes	Compromised				
	$\mathcal{C}$	$\mathcal{S}$ (pas.)	$\mathcal{S}$ (act.)	$\mathcal{IS}$s	$\mathcal{S}$, $\mathcal{IS}$s
[13], [5]	√	X	X	–	–
[26]	√	√	X	X	X
[28]	√	√	√	X	X
[4]	√	√	X	√	X
[14]	√	√	X	–	–
Our work	√	√	√	–	–

Table 1: Resilience of deduplication schemes.

9. DISCUSSION

Incentives. In our scheme $\mathcal{C}$s have to run several PAKE instances as both uploaders and checkers. This imposes a cost on each $\mathcal{C}$. $\mathcal{S}$ is the direct beneficiary of deduplication. $\mathcal{C}$s may indirectly benefit in that effective deduplication makes the storage system more efficient and can thus potentially lower the cost incurred by each $\mathcal{C}$. Nevertheless, it is desirable to have more direct incentive mechanisms to encourage $\mathcal{C}$s to do do PAKE checks. For example, if a file uploaded by $\mathcal{C}$ is found to be shared by other $\mathcal{C}$s, $\mathcal{S}$ could reward the $\mathcal{C}$s owning that file by giving them small increases in their respective storage quotas.

User involvement. A simple solution to the problem of deduplication vs. privacy is to have the user identify sensitive files. The client-side program can then use convergent encryption for non-sensitive files and semantically secure encryption for sensitive files. This approach has three drawbacks: it is too burdensome for average users, it reveals which files are sensitive, and it foregoes deduplication of sensitive files altogether.

Deduplication effectiveness. We can improve deduplication effectiveness by introducing additional checks. For example, an uploader can indicate the (approximate) file size (which will be revealed anyway) so that $\mathcal{S}$ can limit the selection of checkers to those whose files are of a similar size. Similarly, $\mathcal{S}$ can keep track of similarities between $\mathcal{C}$s based on the number of files they share and use this information while selecting checkers by prioritizing checkers who are similar to the uploader. Nevertheless, as discussed in Figure 12, our scheme exceeds what is considered as realistic levels of deduplication early in the life of the storage system. Whitehouse [30] reported that when selecting a deduplication scheme, enterprise administrators rated considerations such as ease of deployment and of use being more important than deduplication ratio. Therefore, we argue that the very small sacrifice in deduplication ratio is offset by the significant advantage of ensuring user privacy without having to use independent third party servers.
Block-level deduplication. Our scheme can be applied for both file-level and block-level deduplication. Applying it for block-level deduplication will incur more overhead.
Datasets. Deduplication effectiveness is highly dependent on the dataset. Analysis using more realistic datasets can shed more light on the efficacy of our scheme.
Realistic modeling of offline status. In our analysis of how deduplication effectiveness is affected by the offline rate (Figure 10), we assumed a simple model where the offline status of clients is distributed uniformly at a specified rate. In practice the offline status is influenced by many factors like geography and time of day.

10. CONCLUSIONS

In this paper, we dealt with the dilemma that cloud storage providers want to use deduplication to save cost, while users want their data to be encrypted on client-side. We designed a PAKE-based protocol that enables two parties to privately compare their secrets and share the encryption key. Based on this protocol, we developed the first single-server scheme that enables cross-user deduplication of client-side encrypted data.

Acknowledgments

This work was supported in part by the "Cloud Security Services" project funded by the Academy of Finland (283135), the EU 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE) and a grant from the Israel Ministry of Science and Technology. We thank Ivan Martinovic for suggesting the analogy between our system and web-caching proxies. We thank Billy Brumley, Kaitai Liang, and the reviewers for their valuable feedback.

11. REFERENCES

[1] M. Abdalla and D. Pointcheval. Simple password-based encrypted key exchange protocols. In A. Menezes, editor, *CT-RSA*, volume 3376 of *LNCS*, pages 191–208. Springer, 2005.

[2] A. Afshar, P. Mohassel, B. Pinkas, and B. Riva. Non-interactive secure computation based on cut-and-choose. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT*, volume 8441 of *LNCS*, pages 387–404. Springer, 2014.

[3] B. Barak, R. Canetti, Y. Lindell, R. Pass, and T. Rabin. Secure computation without authentication. In V. Shoup, editor, *CRYPTO*, volume 3621 of *LNCS*, pages 361–377. Springer, 2005.

[4] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *USENIX Security*, pages 179–194. USENIX Association, 2013.

[5] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *EUROCRYPT*, volume 7881 of *LNCS*, pages 296–312. Springer, 2013.

[6] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In Preneel [25], pages 139–155.

[7] S. M. Bellovin and M. Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84. IEEE Computer Society, 1992.

[8] V. Boyko, P. D. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using diffie-hellman. In Preneel [25], pages 156–171.

[9] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM*, volume 1, pages 126–134, Mar 1999.

[10] R. Canetti, D. Dachman-Soled, V. Vaikuntanathan, and H. Wee. Efficient password authenticated key exchange via oblivious transfer. In M. Fischlin, J. Buchmann, and M. Manulis, editors, *PKC*, volume 7293 of *LNCS*, pages 449–466. Springer, 2012.

[11] R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT*, pages 404–421, 2005.

[12] W. Diffie and M. E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[13] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS*, pages 617–624. IEEE, 2002.

[14] Y. Duan. Distributed key generation for encrypted deduplication: Achieving the strongest privacy. In *CCSW*, pages 57–68. ACM, 2014.

[15] M. Dutch. Understanding data deduplication ratios. SNIA Data Management Forum, 2008. `http://storage.ctocio.com.cn/imagelist/2009/222/l3pm284d8r1s.pdf`.

[16] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. Blakley and D. Chaum, editors, *CRYPTO*, volume 196 of *LNCS*, pages 10–18. Springer, 1985.

[17] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2004.

[18] O. Goldreich and Y. Lindell. Session-key generation using human passwords only. In J. Kilian, editor, *CRYPTO*, volume 2139 of *LNCS*, pages 408–432. Springer, 2001.

[19] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud

storage. *IEEE Security & Privacy*, 8(6):40–47, Nov 2010.

[20] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010.

[21] I. Jeong, J. Katz, and D. Lee. One-round protocols for two-party authenticated key exchange. In M. Jakobsson, M. Yung, and J. Zhou, editors, *ACNS*, volume 3089 of *LNCS*, pages 220–232. Springer, 2004.

[22] J. Katz and V. Vaikuntanathan. Round-optimal password-based authenticated key exchange. *Journal of Cryptology*, 26(4):714–743, 2013.

[23] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *USENIX FAST*, pages 1–1. USENIX Association, 2011.

[24] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *EUROCRYPT*, volume 1592 of *LNCS*, pages 223–238. Springer, 1999.

[25] B. Preneel, editor. *EUROCRYPT*, volume 1807 of *LNCS*. Springer, 2000.

[26] P. Puzio, R. Molva, M. Önen, and S. Loureiro. Cloudedup: Secure deduplication with encrypted data for cloud storage. In *CloudCom*, pages 363–370. IEEE Computer Society, 2013.

[27] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *USENIX FAST*, pages 7–7. USENIX Association, 2002.

[28] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl. A secure data deduplication scheme for cloud storage. In N. Christin and R. Safavi-Naini, editors, *FC*, volume 8437 of *LNCS*, pages 99–118. Springer, 2014.

[29] J. M. Wendt. Getting Real About Deduplication Ratios. `http://www.dcig.com/2011/02/getting-real-about-deduplication.html`, 2011.

[30] L. Whitehouse. Understanding data deduplication ratios in backup systems. TechTarget article, May 2009. `http://searchdatabackup.techtarget.com/tip/Understanding-data-deduplication-ratios-in\-backup-systems`.

[31] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, Oct 1986.

[32] G. K. Zipf. Relative frequency as a determinant of phonetic change. *Harvard studies in classical philology*, pages 1–95, 1929.

APPENDIX

A. NOTATION TABLE

A table of notations is shown in Table 2.

Notation	**Description**
Entities	
$\mathcal{C}$	Client
$\mathcal{S}$	Server
$\mathcal{A}$	Adversary
$\mathcal{IS}$	Independent Server
Cryptographic Notations	
$E()$	Symmetric key encryption
$D()$	Symmetric key decryption
k	Symmetric encryption/decryption key
$Enc()$	Additively homomorphic encryption
$Dec()$	Additively homomorphic decryption
$\oplus$	Additively homomorphic addition
$\ominus$	Additively homomorphic subtraction
$H()$	Cryptographic hash function
h	Cryptographic hash
$SH()$	Short hash function
sh	Short hash
PAKE	Password Authenticated Key Exchange
$\mathcal{F}_{pake}$	Ideal functionality of PAKE
$\mathcal{F}_{dedup}$	Ideal functionality of deduplication protocol
Parameters	
F	File
m	Entropy of a predictable file
n	Length of the short hash
RL_u	Rate limit by uploaders
RL_c	Rate limit by checkers
t_F	Random threshold for a file
c_F	Counter for a file
ρ	Deduplication Percentage

Table 2: Summary of notations

B. SECOND ORDER DEVIATION FROM PERFECT DEDUPLICATION

Figure 16 how the second order difference in deduplication percentage (between perfect deduplication and our scheme) evolves as more files are uploaded to the storage server. This second order difference essentially vanishes as more files are added to the server.

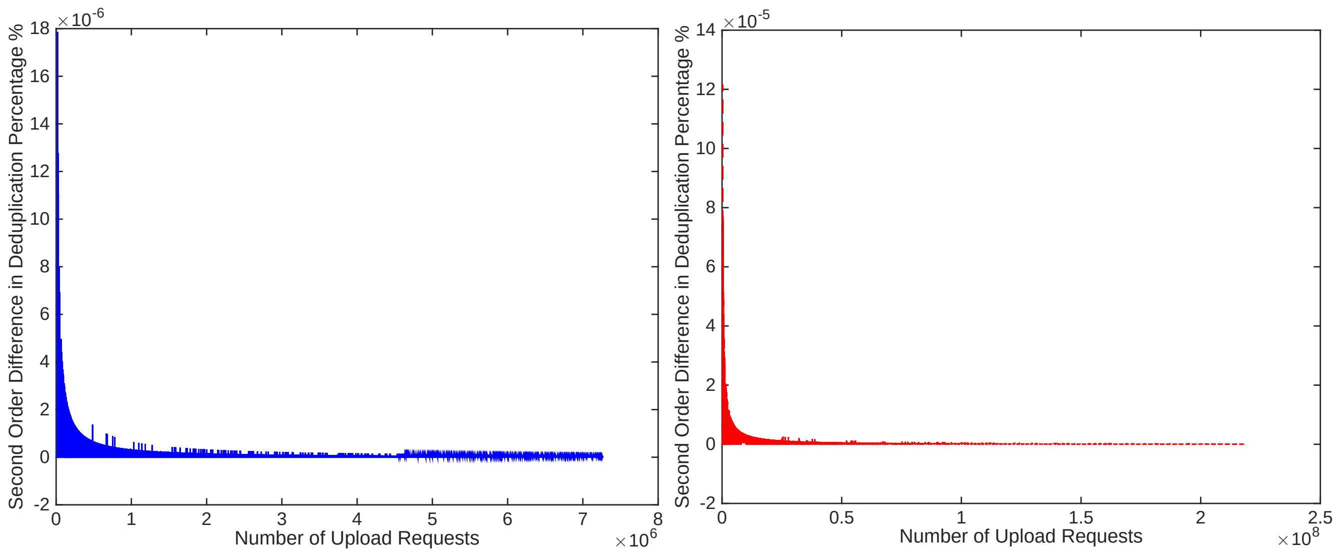


Figure 16: Second order difference in deduplication percentage: perfect deduplication VS. Number of upload requests. Left: Media dataset; Right: Enterprise dataset