DupLESS: Server-Aided Encryption for Deduplicated Storage

Mihir Bellare and Sriram Keelveedhi, *University of California, San Diego;*
Thomas Ristenpart, *University of Wisconsin—Madison*

DupLESS:
Server-Aided Encryption for Deduplicated Storage

Mihir Bellare

Sriram Keelveedhi

Thomas Ristenpart

University of California, San Diego

University of California, San Diego

University of Wisconsin–Madison

Abstract

Cloud storage service providers such as Dropbox, Mozy, and others perform deduplication to save space by only storing one copy of each file uploaded. Should clients conventionally encrypt their files, however, savings are lost. Message-locked encryption (the most prominent manifestation of which is convergent encryption) resolves this tension. However it is inherently subject to brute-force attacks that can recover files falling into a known set. We propose an architecture that provides secure deduplicated storage resisting brute-force attacks, and realize it in a system called DupLESS. In DupLESS, clients encrypt under message-based keys obtained from a key-server via an oblivious PRF protocol. It enables clients to store encrypted data with an existing service, have the service perform deduplication on their behalf, and yet achieves strong confidentiality guarantees. We show that encryption for deduplicated storage can achieve performance and space savings close to that of using the storage service with plaintext data.

1 Introduction

Providers of cloud-based storage such as Dropbox [3], Google Drive [7], and Mozy [63] can save on storage costs via deduplication: should two clients upload the same file, the service detects this and stores only a single copy. The savings, which can be passed back directly or indirectly to customers, are significant [50, 61, 74] and central to the economics of the business.

But customers may want their data encrypted, for reasons ranging from personal privacy to corporate policy to legal regulations. A client could encrypt its file, under a user's key, before storing it. But common encryption modes are randomized, making deduplication impossible since the SS (Storage Service) effectively always sees different ciphertexts regardless of the data. If a client's encryption is deterministic (so that the same file will al-

ways map to the same ciphertext) deduplication is possible, but only for that user. Cross-user deduplication, which allows more storage savings, is not possible because encryptions of different clients, being under different keys, are usually different. Sharing a single key across a group of users makes the system brittle in the face of client compromise.

One approach aimed at resolving this tension is message-locked encryption (MLE) [18]. Its most prominent instantiation is convergent encryption (CE), introduced earlier by Douceur et al. [38] and others (c.f., [76]). CE is used within a wide variety of commercial and research SS systems [1, 2, 5, 6, 8, 12, 15, 32, 33, 55, 60, 66, 71, 78, 79]. Letting M be a file's contents, hereafter called the message, the client first computes a key $K \leftarrow \mathsf{H}(M)$ by applying a cryptographic hash function H to the message, and then computes the ciphertext $C \leftarrow \mathsf{E}(K, M)$ via a deterministic symmetric encryption scheme. The short message-derived key K is stored separately encrypted under a per-client key or password. A second client B encrypting the same file M will produce the same C, enabling deduplication.

However, CE is subject to an inherent security limitation, namely susceptibility to offline brute-force dictionary attacks. Knowing that the target message M underlying a target ciphertext C is drawn from a dictionary $S = \{M_1, \ldots, M_n\}$ of size n, the attacker can recover M in the time for $n = |S|$ off-line encryptions: for each $i = 1, \ldots, n$, it simply CE-encrypts M_i to get a ciphertext denoted C_i and returns the M_i such that $C = C_i$. (This works because CE is deterministic and keyless.) Security is thus only possible when the target message is drawn from a space too large to exhaust. We say that such a message is *unpredictable*.

Bellare, Keelveedhi, and Ristenpart [18] treat MLE formally, providing a definition (semantic-security for unpredictable messages) to capture the best possible security achievable for MLE schemes in the face of the inherent limitation noted above. The definition is based

on previous ones for deterministic encryption, a primitive subject to analogous inherent limitations [16,17,27]. The authors go on to show that CE and other mechanisms achieve their definition in the random-oracle model.

The unpredictability assumption. The above-mentioned work puts security on a firm footing in the case messages are unpredictable. In practice, however, security only for unpredictable data may be a limitation for, and threat to, user privacy. We suggest two main reasons for this. The first is simply that data is often predictable. Parts of a file's contents may be known, for example because they contain a header of known format, or because the adversary has sufficient contextual information. Some data, such as very short files, are inherently low entropy. This has long been recognized by cryptographers [43], who typically aim to achieve security regardless of the distribution of the data.

The other and perhaps more subtle fear with regard to the unpredictability assumption is the difficulty of validating it or testing the extent to which it holds for "real" data. When we do not know how predictable our data is to an adversary, we do not know what, if any, security we are getting from an encryption mechanism that is safe only for unpredictable data. These concerns are not merely theoretical, for offline dictionary attacks are recognized as a significant threat to CE in real systems [77] and are currently hindering deduplication of outsourced storage for security-critical data.

This work. We design and implement a new system called DupLESS (Duplicateless Encryption for Simple Storage) that provides a more secure, easily-deployed solution for encryption that supports deduplication. In DupLESS, a group of affiliated clients (e.g., company employees) encrypt their data with the aid of a key server (KS) that is separate from the SS. Clients authenticate themselves to the KS, but do not leak any information about their data to it. As long as the KS remains inaccessible to attackers, we ensure high security. (Effectively, semantic security [43], except that ciphertexts leak equality of the underlying plaintexts. The latter is necessary for deduplication.) If both the KS and SS are compromised, we retain the current MLE guarantee of security for unpredictable messages.

Unlike prior works that primarily incorporate CE into new systems, our goal is to make DupLESS work transparently with existing SS systems. DupLESS therefore sits as a layer on top of existing simple storage interfaces, wrapping store, retrieve, and other requests with algorithms for encrypting filenames and data on the fly. This also means that DupLESS was built: to be as feature-compatible as possible with existing API commands, to not assume any knowledge about the systems implementing these APIs, to give performance very close to that of

using the SS without any encryption, and to achieve the same availability level as provided by the SS.

We implement DupLESS as a simple-to-use command-line client that supports both Dropbox [3] and Google Drive [7] as the SS. We design two versions of the KS protocol that clients can use while encrypting files. The first protocol uses a RESTful, HTTPS based, web interface, while the second is a custom protocol built over UDP. The first is simpler, being able to run on top of existing web servers, and the latter is optimized for latency, and capable of servicing requests at close to the (optimal) round-trip time of the network. These protocols and their implementations, which at core implement an oblivious pseudorandom function (OPRF) [64] service, may be of independent interest.

To evaluate end-to-end performance, we deploy our KS on Amazon EC2 [10] and experimentally evaluate its performance. DupLESS incurs only slight overheads compared to using the SS with plaintext data. For a 1 MB file and using Dropbox, the bandwidth overhead is less than 1% and the overhead in the time to store a file is about 17%. We compute storage overheads of as little as 4.5% across a 2 TB dataset consisting of over 2,000 highly dedupable virtual machine file system images that we gathered from Amazon EC2. All this shows that DupLESS is practical and can be immediately deployed in most SS-using environments. The source code for DupLESS is available from [4].

2 Setting

At a high level, our setting of interest is an enterprise network, consisting of a group of affiliated clients (for example, employees of a company) using a deduplicated cloud storage service (SS). The SS exposes a simple interface consisting of only a handful of operations such as storing a file, retrieving a file, listing a directory, deleting a file, etc.. Such systems are widespread (c.f., [1,3,7,11,63]), and are often more suitable to user file backup and synchronization applications than richer storage abstractions (e.g., SQL) [37,69] or block stores (c.f., [9]). An example SS API, abstracted from Dropbox, is detailed in Figure 5 (Section 6). The SS performs deduplication along file boundaries, meaning it checks if the contents of two files are the same and deduplicates them if so, by storing only one of them.

Clients have access to a key server (KS), a semi-trusted third party which will aid in performing dedupable encryption. We will explain further the role of the KS below. Clients are also provisioned with per-user encryption keys and credentials (e.g., client certificates).

Threat model. Our goal is to protect the confidentiality of client data. Attackers include those that gain access to the SS provider's systems (including malicious insiders working at the provider) and external attackers with access to communication channels between clients and the KS or SS. Security should hold for all files, not just unpredictable ones. In other words, we seek semantic security, leaking only equality of files to attackers.

We will also be concerned with compromise resilience: the level of security offered by the scheme to legitimate clients should degrade gracefully, instead of vanishing, should other clients or even the KS be compromised by an attacker. Specifically, security should hold at least for unpredictable files (of uncompromised clients) when one or more clients are compromised and when the KS is compromised.

We will match the availability offered by the SS, but explicitly do not seek to ensure availability in the face of a malicious SS: a malicious provider can always choose to delete files. We will, however, provide protection against a malicious SS that may seek to tamper with clients' data, or mount chosen-ciphertext attacks, by modifying stored ciphertexts.

Malicious clients can take advantage of an SS that performs client-side deduplication to mount a side-channel attack [46]. This arises because one user can tell if another user has already stored a file, which could violate the latter's privacy.[1] We will not introduce such side-channels. A related issue is that client-side deduplication can be abused to perform illicit file transfers between clients [73]. We will ensure that our systems can work in conjunction with techniques such as proofs-of-ownership [45] that seek to prevent such issues.

We will not explicitly target resistance to traffic analysis attacks that abuse leakage of access patterns [48] or file lengths [24, 31, 40, 47, 59, 65, 72], though our system will be compatible with potential countermeasures.

Our approaches may be used in conjunction with existing mechanisms for availability auditing [13, 41, 51, 70] or file replication across multiple services [26]. (In the latter case, our techniques will enable each service to independently perform deduplication.)

Design goals. In addition to our security goals, the system we build will meet the following functionality properties. The system will be *transparent*, both from the perspective of clients and the SS. This means that the system will be backwards-compatible, work within existing SS APIs, make no assumptions about the implementation details of the SS, and have performance closely matching that of direct use of the SS. In normal operation and for all clients of a particular KS, the space required to store

all encrypted data will match closely the space required when storing plaintext data. The system should never reduce storage availability, even when the KS is unavailable or under heavy load. The system will not require *any* client-side state beyond a user's credentials. A user will be able to sit down at any system, provide their credentials, and synchronize their files. We will however allow client-side caching of data to improve performance.

Related approaches. Several works have looked at the general problem of enterprise network security, but none provide solutions that meet all requirements from the above threat model. Prior works [42, 53, 54, 58, 75] which build a secure file system on top of a flat outsourced storage server break deduplication mechanisms and are unfit for use in our setting. Convergent encryption (CE) based solutions [8, 71], as we explored in the Introduction, provide security only for unpredictable messages even in the best case, and are vulnerable to brute-force attacks. The simple approach of sharing a secret key across clients with a deterministic encryption scheme [16, 68] fails to achieve compromise resilience. Using CE with an additional secret shared across all clients [76] does not work for the same reason.

3 Overview of DupLESS

DupLESS starts with the observation that brute-force ciphertext recovery in a CE-type scheme can be dealt with by using a key server (KS) to derive keys, instead of setting keys to be hashes of messages. Access to the KS is preceded by authentication, which stops external attackers. The increased cost slows down brute-force attacks from compromised clients, and now the KS can function as a (logically) single point of control for implementing rate-limiting measures. We can expect that by scrupulous choice of rate-limiting policies and parameters, brute-force attacks originating from compromised clients will be rendered less effective, while normal usage will remain unaffected.

We start by looking at secret-parameter MLE, an extension to MLE which endows all clients with a system-wide secret parameter *sk* (see Section 4). The rationale here is that if *sk* is unknown to the attacker, a high level of security can be achieved (semantic security, except for equality), but even if *sk* is leaked, security falls to that of regular MLE. A server-aided MLE scheme then is a transformation where the secret key is restricted to the KS instead of being available to all clients. One simple approach to get server-aided MLE is to use a PRF F, with a secret key K that never leaves the KS. A client would send a hash H of a file to the KS and receive back a *message-derived* key $K' \leftarrow \mathsf{F}(K, H)$. The other steps are as in CE. However, this approach proves unsatisfying

---

[1]The reader might be interested to note that our experience with the Dropbox client suggests this side channel still exists.

from a security perspective. The KS here becomes a single point of failure, violating our goal of compromise resilience: an attacker can obtain hashes of files after gaining access to the KS, and can recover files with brute-force attacks. Instead, DupLESS employs an oblivious PRF (OPRF) protocol [64] between the KS and clients, which ensures that the KS learns nothing about the client inputs or the resulting PRF outputs, and that clients learn nothing about the key. In Section 4, we propose a new server-aided MLE scheme DupLESSMLE which combines a CE-type base with the OPRF protocol based on RSA blind-signatures [20, 29, 30].

Thus, a client, to store a file M, will engage in the RSA OPRF protocol with the KS to compute a message-derived key K, then encrypt M with K to produce a ciphertext C_{data}. The client's secret key will be used to encrypt K to produce a key encapsulation ciphertext C_{key}. Both C_{key} and C_{data} are stored on the SS. Should two clients encrypt the same file, then the message-derived keys and, in turn, C_{data} will be the same (the key encapsulation C_{key} will differ, but this ciphertext is small). The DupLESS client algorithms are described in Section 6 along with how DupLESS handles filenames and paths.

Building a system around DupLESSMLE requires careful design in order to achieve high performance. DupLESS uses at most one or two SS API calls per operation. (As we shall see, SS API calls can be slow.) Because interacting with the KS is on the critical path for storing files, DupLESS incorporates a fast client-to-KS protocol that supports various rate-limiting strategies. When the KS is overloaded or subjected to denial-of-service attacks, DupLESS clients fall back to symmetric encryption, ensuring availability. On the client side, DupLESS introduces *dedup heuristics* (see Section 6) to determine whether the file about to be stored on the SS should be selected for deduplication, or processed with randomized encryption. For example, very small files or files considered particularly sensitive can be prevented from deduplication. We use deterministic authenticated encryption (DAE) [68] to protect, in a structure-preserving way, the path and filename associated to stored files. Here we have several choices along an efficiency/security continuum. Our approach of preserving folder structure leaks some information to the SS, but on the other hand, enables direct use of the SS-provided API for file search and moving folders.

DupLESS is designed for a simple SS API, but can be adapted to settings in which block-oriented deduplication is used, and to complex network storage and backup solutions that use NFS [62], CIFS [56] and the like, but we do not consider these further.

In the following sections we go into greater detail on the various parts of the DupLESS system, starting with the cryptographic primitives in Section 4, then moving
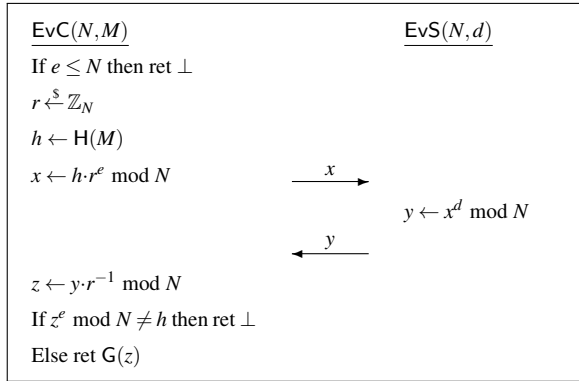
on to describing KS design in Section 5, and then on to the client algorithms in Section 6, followed by performance and security in Sections 7 and 8 respectively.

4 Cryptographic Primitives

A *one-time encryption scheme* SE with key space $\{0,1\}^k$ is a pair of deterministic algorithms (E, D). Encryption E on input a key $K \in \{0,1\}^k$ and message $M \in \{0,1\}^*$ outputs a ciphertext C. Decryption D takes a key and a ciphertext and outputs a message. CTR mode using AES with a fixed IV is such a scheme. An *authenticated encryption* (AE) scheme is pair of algorithms $AE = (EA, DA)$ [19, 67]. Encryption EA takes as input a key $K \in \{0,1\}^k$, associated data $D \in \{0,1\}^*$, and message $M \in \{0,1\}^*$ and outputs a ciphertext of size $|M| + \tau_d$, where τ_d is the ciphertext stretch (typically, 128 bits). Decryption DA is deterministic; it takes input a key, associated data, and a ciphertext and outputs a message or error symbol $\bot$. When encryption is deterministic, we call the scheme a deterministic authenticated encryption (DAE) scheme [68]. We use the Encrypt-then-MAC [19] scheme for AE and SIV mode [68] for DAE, both with HMAC[SHA256] and CTR[AES].

Oblivious PRFs. A (verifiable) oblivious PRF (OPRF) scheme [64] consists of five algorithms $OPRF = (Kg, EvC, EvS, Vf, Ev)$, the last two deterministic. Key generation $(pk, sk) \xleftarrow{\$} Kg$ outputs a public key pk which can be distributed freely among several clients, and a secret key sk, which remains with a single entity, the server. The evaluation protocol runs as follows: on the client-side, EvC starts with an input x and ends with output y such that $y = Ev(sk, x)$, while on the server-side, EvS starts with secret key sk and ends without output. Figure 1 gives an example. Verification $Vf(pk, x, y)$ returns a boolean. Security requires that (1) when keys are picked at random, $Ev(sk, \cdot)$ outputs are indistinguishable from random strings to efficient attackers without pk, and (2) no efficient attacker, given (pk, sk), can provide x, x', y such that $Vf(pk, x, y) = Vf(pk, x', y) = $ true, or $Vf(pk, x, y) = $ true but $Ev(sk, x) \neq y$, or $Vf(pk, x, y) = $ false but $Ev(sk, x) = y$, except with negligible probability. Moreover, in the OPRF protocol, the server learns nothing about client inputs or resulting PRF outputs, and the client learns nothing about sk.

Verifiable OPRF schemes can be built from deterministic blind signatures [29]. The RSA-OPRF[G, H] scheme based on RSA blind signatures [20, 30] is described as follows. The public RSA exponent e is fixed as part of the scheme. Key generation Kg runs RSAKg with input e to get N, d such that $ed \equiv 1 \mod \phi(N)$, modulus N is the product of two distinct primes of roughly equal length and $N < e$. Then, $(N, (N, d))$ is output as

```
EvC(N,M)                                EvS(N,d)
─────────                               ─────────
If e ≤ N then ret ⊥
r ←$ ℤ_N
h ← H(M)
x ← h·r^e mod N        ──── x ───→
                                        y ← x^d mod N
                       ←─── y ────
z ← y·r^{-1} mod N
If z^e mod N ≠ h then ret ⊥
Else ret G(z)
```

Figure 1: The RSA-OPRF protocol. The key generation Kg outputs PRF key N,d and verification key N. The client uses two hash functions $\mathsf{H}: \{0,1\}^* \to \mathbb{Z}_N$ and $\mathsf{G}: \mathbb{Z}_N \to \{0,1\}^k$.

the public key, secret key pair. The evaluation protocol $(\mathsf{EvC},\mathsf{EvS})$ with verification Vf is shown in Figure 1. The client uses a hash function $\mathsf{H}: \{0,1\}^* \to \mathbb{Z}_N$ to first hash the message to an element of $\mathbb{Z}_N$, and then blinds the result with a random group element r raised to the e-th power. The resulting blinded hash, denoted x, is sent to the KS. The KS signs it by computing $y \leftarrow x^d \bmod N$, and sends back y. Verification then removes the blinding by computing $z \leftarrow yr^{-1} \bmod N$, and then ensures that $z^e \bmod N$ is indeed equal to $\mathsf{H}(M)$. Finally, the output of the PRF is computed as $\mathsf{G}(z)$, where $\mathsf{G}: \mathbb{Z}_N \to \{0,1\}^k$ is another hash function.

This protocol can be shown to be secure as long as the map $f_e: \mathbb{Z}_N^* \to \mathbb{Z}_N^*$, defined by $f_e(x) = x^e \bmod N$ for all $x \in \mathbb{Z}_N^*$, is a permutation on $\mathbb{Z}_N^*$, which is assured by $\gcd(\varphi(N),e) = 1$. In particular, this is true if the server creates its keys honestly. However, in our setting, the server can cheat while generating the keys, in an attempt to glean something about $\mathsf{H}(M)$. This is avoided by requiring that $N < e$, which will be verified by the client. Given that e is prime, this standard technique ensures that $\gcd(\varphi(N),e) = 1$ even if N is maliciously generated, and thus ensures that f_e is a permutation. Since f_e is a permutation and the client checks the signature, even a malicious server cannot force the output $K = \mathsf{G}(z)$ to be a fixed value or force two keys output for distinct messages to collide, as long as G is collision-resistant.

MLE. A deterministic Message-Locked Encryption (MLE) scheme is a tuple $\mathsf{MLE} = (\mathsf{P},\mathsf{K},\mathsf{E},\mathsf{D})$ of algorithms, the last three deterministic[2]. Parameter generation outputs a public parameter $P \xleftarrow{\$} \mathsf{P}$, common to all users of a system. To encrypt M, one generates the message-derived key $K \leftarrow \mathsf{K}(P,M)$ and ciphertext

$C \leftarrow \mathsf{E}(P,K,M)$. Decryption works as $M \leftarrow \mathsf{D}(P,K,C)$. Security requires that no efficient attacker can distinguish ciphertexts of unpredictable messages from random strings except with negligible probability. Convergent encryption (CE) [38] is the most prominent MLE scheme. We use CE with parameters P set to random 128-bit strings, key generation returning the first 128 bits of $\mathsf{SHA256}(P\|M)$ on input M, and encryption and decryption being implemented with $\mathsf{CTR}[\mathsf{AES}]$.

In a secret-parameter MLE scheme SPMLE, parameter generation outputs a (system-wide) secret parameter sk along with a public parameter P. This secret parameter, which is provided to all legitimate users, is used to generate message-derived keys as $K \leftarrow \mathsf{K}(P,sk,M)$. In a server-aided MLE scheme, the secret parameter is provided only to a KS. Clients interact with the KS to obtain message-derived keys. A simple of way of doing this of course is that clients can send the messages to the KS which would then reply with message-derived keys. But, as we saw in the previous section, this is undesirable in the DupLESS setting, as the KS now becomes a single point of failure. Instead, we propose a new server-aided MLE scheme DupLESSMLE combining $\mathsf{RSA\text{-}OPRF}[\mathsf{G},\mathsf{H}] = (\mathsf{Kg},\mathsf{EvC},\mathsf{EvS},\mathsf{Vf},\mathsf{Ev})$ and $\mathsf{CTR}[\mathsf{AES}]$. Here parameter generation runs Kg to get $(N,(N,d))$, then outputs N as the public parameter and (N,d) as the secret parameter (recall that e is fixed as part of the scheme). From a message M, a key K is generated as $K \leftarrow \mathsf{Ev}((N,d),M) = \mathsf{G}(\mathsf{H}(M)^d \bmod N)$ by interacting with the KS using EvC and EvS. Encryption and decryption work as in CE, with $\mathsf{CTR}[\mathsf{AES}]$. We use RSA1024 with full-domain-hash using SHA256 in the standard way [22] to get H and G.

The advantage of server-aided MLE is the prospect of multi-tiered security. In DupLESSMLE in particular, when the adversary does not have access to the KS (but has access to ciphertexts and OPRF inputs and outputs), it has no knowledge of sk, and semantic-security similar to deterministic SE schemes follows, from the security of $\mathsf{RSA\text{-}OPRF}[\mathsf{G},\mathsf{H}]$ and $\mathsf{CTR}[\mathsf{AES}]$. When the attacker has access to the KS additionally, attacks are still constrained to be online and consequently slow, and subject to rate-limiting measures that the KS imposes. Security here relies on implementing the OPRF protocol correctly, and ensuring that the rate-limiting measures cannot be circumvented. We will analyze this carefully in Section 5. Even when sk is compromised to the attacker, DupLESSMLE provides the usual MLE-style security, conditioned on messages being unpredictable. Moreover, we are guaranteed that the clients' inputs are hidden from the KS, even if the KS is under attack and deviates from its default behavior, from the security of the $\mathsf{RSA\text{-}OPRF}[\mathsf{G},\mathsf{H}]$ protocol.

---

[2]We drop the tag generation algorithm which was part of the original MLE formulation [18]. Since we restrict attention to deterministic MLE schemes, we let ciphertexts work as tags.

5 The DupLESS KS

In this section we describe the KS side of DupLESS. This includes protocols for client-KS interaction which realize RSA-OPRF[G, H], and rate limiting strategies which limit client queries to slow down online brute-force attacks. We seek low-latency protocols to avoid degrading performance, which is important because the critical path during encryption includes interaction with a KS. Additionally, the protocol should be light-weight, letting the KS handle a reasonably high request volume.

We describe two protocols: OPRFv1, and OPRFv2, which rely on a CA providing the KS and clients with verifiable TLS certificates. In the following, we assume that each client has a unique certificate, and that clients can be identified by their certificates. Of course, the protocols can be readily converted to work with other authentication frameworks. We believe our OPRF protocols to be faster than previous implementations [36], and given the support for rate-limiting, we expect that they will be useful in other applications using OPRFs.

HTTPS based. In the first protocol, OPRFv1, all communication with the KS happens over HTTPS. The KS exposes an interface with two procedures: KSInit and KSReq. The first time a client uses the KS, it makes a KSInit request to obtain, and then locally cache, the KS's OPRF public key. Here the client must perform any necessary checks of the public key, which for our scheme is simply that $e > N$. When the client wants a key, say for a file it is about to upload, the client will make use of the KSReq interface, by sending an HTTPS POST of the blinded hash value. Now, the KS checks request validity, and performs rate-limiting measures which we describe below. Then, the KS computes the signature over the blinded hash value, and sends this back over the established HTTPS channel.

OPRFv1 has the benefit of extreme simplicity. With 20 lines of code (excluding rate limiting logic) in the form of a Web-Server Gateway Interface (WSGI) Python module, one can run the KS on top of most webservers. We used Apache 2.0 in our implementation.

Unfortunately, while simple, this is a high latency solution, as it requires four full round trips across the network (1 for TCP handshake, 2 for the TLS handshake, 1 for the HTTP request) to perform KSReq. While subsecond latency is not always critical (e.g., because of poor SS performance or because the KS and clients share a LAN), it will be critical in many settings, and so we would like to do better.

UDP based. We therefore turn to OPRFv2, which removes the slow per-request handshakes from the critical path of encryption. Here, the KSInit procedure starts with a TLS handshake with mutual authentication, initi-

ated by a client. The KS responds immediately following a valid handshake with the OPRF public key pk, a TLS identifier of a hash function H (by default SHA-256), a random session identifier $S \in \{0,1\}^{128}$, and a random session key $K_S \in \{0,1\}^k$ (we set $k = 128$ in our implementations). We shave off one round trip from KSInit by responding immediately, instead of waiting for an HTTP message as in OPRFv1. The KS also associates a sequence number with this session, initialized to zero. Internally the KS maintains two tables, one mapping session identifiers with keys, and a second which keeps track of sequence numbers. Each session lasts for a fixed time period (currently 20 minutes in our implementation) and table entries are removed after the session expires. The client caches pk, S and K_S locally and initializes a sequence number $N = 0$.

To make an OPRF request KSReq on a blinded value X, the client first increments the sequence number $N \leftarrow N + 1$, then computes a MAC tag using its session key, as $T \leftarrow \mathsf{HMAC}[\mathsf{H}](K_S, S \| N \| X)$ and sends the concatenation $S \| N \| X \| T$ to the KS in a single UDP packet. The KS recovers S, N, X and T and looks up K_S and N_S. It ensures that $N > N_S$ and checks correctness of the MAC T. If the packet is malformed or if some check fails, then the KS drops the packet without further action. If all the checks pass, the KS sends the OPRF protocol response in a single UDP packet.

The client waits for time t_R after sending a KSReq packet before triggering timeout behavior. In our implementation, this involves retrying the same request twice more with time t_R between the tries, incrementing the sequence number each time. After three attempts, the client will try to initiate a new session, again timing out after t_R units. If this step fails, the client believes the KS to be offline. This timeout behavior is based on DNS, and following common parameters, we set $t_R = 1$ second.

We implemented OPRFv2 in Python. It comes to 165 lines of code as indicated by the cloc utility, the bulk of which is in fact the rate limiting logic discussed below. Our current KS implementation is not yet optimized. For example it spawns and kills a new thread for each connection request (as opposed to keeping a pool of children around, as in Apache). Nevertheless the implementation is fully functional and performs well.

Rate limiting KS requests. We explore approaches for per-client rate limiting. In the first approach, called Bounded, the KS sets a bound q on the total number of requests a client can make during a fixed time interval t_E, called an epoch. Further queries by the client will be ignored by the KS, until the end of the epoch. Towards keeping the KS simple, a single timer controls when epochs start and end, as opposed to separate timers for each client that start when their client performs a ses-

sion handshake. It follows that no client can make more than $2q$ queries within a t_E-unit time period.

Setting q gives rise to a balancing act between online brute-force attack speed and sufficiently low-latency KS requests, since a legitimate client that exceeds its budget will have to wait until the epoch ends to submit further requests. However, when using these OPRF protocols within DupLESS, we also have the choice of exploiting the trade-off between dedupability and online brute-force speed. This is because we can build clients to simply continue with randomized encryption when they exceed their budgets, thereby alleviating KS availability issues for a conservative choice of q.

In any case, the bound q and epoch duration should be set so as to not affect normal KS usage. Enterprise network storage workloads often exhibit temporal self-similarity [44], meaning that they are periodic. In this case, a natural choice for the epoch duration is one period. The bound q can be set to the expected number of client requests plus some buffer (e.g., one or more standard deviations). Administrators will need to tune this for their deployment; DupLESS helps ease this burden by its tolerance of changes to q as discussed above.

We also considered two other mechanisms for rate limiting. The fixed delay mechanism works by introducing an artificial delay t_D before the KS responds to a client's query. This delay can either be a system-wide constant, or be set per client. Although this method is the simplest to implement, to get good brute-force security, the delay introduced would have to be substantially high and directly impacts latency. The exponential delay mechanism starts with a small delay, and doubles this quantity after every query. The doubling stops at an upper limit t_U. The server maintains synchronized epochs, as in the bounded approach, and checks the status of active clients after each epoch. If a client makes no queries during an entire epoch, its delay is reset to the initial value. In both these approaches, the server maintains an active client list, which consists of all clients with queries awaiting responses. New queries from clients in the active client list are dropped. Client timeout in fixed delay is $\max(t_D, t_R)$ and in exponential delay it is $\max(t_U, t_R)$.

To get a sense of how such rate-limiting mechanisms might work in real settings, we estimate the effects on brute-force attacks by deriving parameters from the characteristics of a workload consisting of about 2,700 computers running on an enterprise network at NetApp, as reported in [57]. The workload is periodic, with similar patterns every week. The clients together make 1.65 million write queries/week, but the distribution is highly skewed, and a single client could potentially be responsible for up to half of these writes. Let us be conservative and say that our goal is to ensure that clients making at most $825,000$ queries/week should be unaffected by rate-

Mechanism	Rate formula	NetApp Scenario
Bounded	$2q/t_E$	2.73
Fixed delay	$1/t_D$	1.36
Exp. delay	$2t_E/t_U$	2.73
None	3,200	3,200
Offline	120–12000	120–12000

Figure 2: Comparing brute-force rates in queries per second for different rate limiting approaches, no rate limiting (None), and hashes as computed using SHA-256 (Offline). The first column is the formula used to derive the rate as a function of the request limit q, epoch duration t_E, delay t_D, and upper limit t_U. The second column is the rates as for the NetApp workload. The None row does not include offline computation cost.

limiting. We set the epoch duration t_E as one week and query bound as $q = 825k$. The fixed delay would need to be set to 730 milliseconds (in order to facilitate $825k$ requests in one week), which is also the upper limit t_U for the exponential technique.

The maximum query rates in queries per second that an attacker who compromised a client can achieve are given in Figure 2, along with the formulas used to calculate them. The "None" row, corresponding to no rate limiting, gives as the rate the highest number of replies per second seen for OPRFv2 in the throughput experiment above. The offline brute force rate was measured by running Intel's optimized version of SHA256 [49] to get processing speed as 120 MBps on our client system, whose 7200-RPM hard disk has peak read speed of 121MBps (as measured by hdparm). The range then varies from the number of hashes per second for 1 MB files up to the number of hashes per second for 1 KB files, assuming just a single system is used.

Despite being generous to offline brute-force attacks (by just requiring computation of a hash, not considering parallelization, and not including in the online attacks any offline computational costs), the exercise shows the huge benefit of forcing brute-force attackers to query the KS. For example, the bounded rate limiting mechanism slows down brute-force attacks by anywhere from 43x for large files up to 4,395x for small files. If the attacker wants to identify a 1KB file which was picked at random from a set S of 2^{25} files, then the offline brute-force attack requires less than an hour, while the bounded rate limited attack requires more than twenty weeks.

We note that bounded rate-limiting is effective only if the file has enough unpredictability to begin with. If $|S| < q = 825k$, then the online brute-force attack will be slowed down only by the network latency, meaning that it will proceed at one-fourth the offline attack rate. Moreover, parallelization will speed up both online and offline attacks, assuming that this is permitted by the KS.

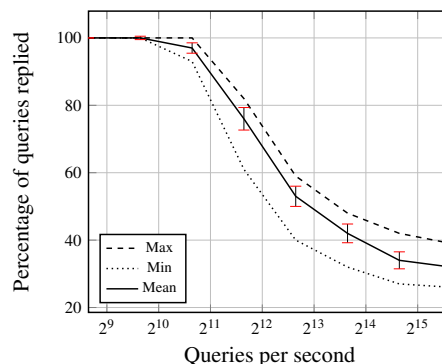| Operation	Latency (ms)
OPRFv1 KSReq (Low KS load)	374 ± 34
OPRFv2 KSInit	278 ± 56
OPRFv2 KSReq (Low KS load)	83 ± 16
OPRFv2 KSReq (Heavy KS load)	118 ± 37
Ping (1 RTT)	78 ± 01

Figure 3: The median time plus/minus one standard deviation to perform KSInit and KSReq operations over 1000 trials. Low KS load means the KS was otherwise idle, whereas Heavy KS load means it was handling 3000 queries per second.

Figure 4: Packet loss in OPRFv2 as a function of query rate. Packet loss is negligible at rates < 3k queries per second.

Performance. For the OPRF, as mentioned in Section 4, we implement RSA1024 with full-domain-hash using SHA256 in the standard way [22]. The PKI setup uses RSA2048 certificates and we fix the ECDHE-RSA-AES128-SHA ciphersuite for the handshake. We set up the two KS implementations (OPRFv1 and OPRFv2) on Amazon EC2 m1.large instances. The client machine, housed on a university LAN, had an x86-64 Intel Core i7-970 processor with a clockspeed fixed at 3201 MHz.

Figure 3 depicts the median times, in milliseconds, of various operations for the two protocols. OPRFv2 significantly outperforms OPRFv1, due to the reduced number of round trip times. On a lightly loaded server, a KS request requires almost the smallest possible time (the RTT to the KS). The time under a heavy KS load was measured while a separate m1.large EC2 instance sent 3000 requests per second. The KS request time for OPRFv2 increases, but is still three times faster than OPRFv1 for a low KS load. Note that the time reported here is only over successful operations; ones that timed out three times were excluded from the median.

To understand the drop rates for the OPRFv2 protocol on a heavily loaded server and, ultimately, the throughput achievable with our (unoptimized) implementation, we performed the following experiment. A client sent $100i$ UDP request packets per second (qps) until a total of 10,000 packets are sent, once for each of $1 \le i \le 64$. The number of requests responded to was then recorded. The min/max/mean/standard deviation over 100 trials are shown in Figure 4. At rates up to around 3,000 queries per second, almost no packets are dropped. We expect that with further (standard) performance optimizations this can be improved even further, allowing a single KS to support a large volume of requests with very occasional single packet drops.

Security of the KS protocols. Adversarial clients can attempt to snoop on, as well as tamper with, communications between (uncompromised) clients and the KS. With rate-limiting in play, adversaries can also attempt to launch denial-of-service (DOS) attacks on uncompro-

mised clients, by spoofing packets from such clients. Finally, adversaries might try to circumvent rate-limiting. A secure protocol must defend against all these threats.

Privacy of OPRF inputs and outputs follows from blinding in the OPRF protocol. Clients can check OPRF output correctness and hence detect tampering. In OPRFv1, every KSReq interaction starts with a mutual-authentication TLS handshake, which prevents adversaries from spoofing requests from other clients. In OPRFv2, creating a new session once again involves a mutual-authentication TLS handshake, meaning that an adversary cannot initiate a session pretending to be a uncompromised client. Moreover, an adversary cannot create a fresh KSReq packet belonging to a session which it did not initiate, without a successful MAC forgery (HMAC with SHA256 specifically). Packets cannot be replayed across sessions, due to session identifiers being picked at random and being included in the MAC, and packets cannot be replayed within a session, due to increasing sequence numbers. Overall, both protocols offer protecting against request spoofing, and neither of the two protocols introduce new denial-of-service vulnerabilities.

In the Bounded rate-limiting approach, the server keeps track of the total number of the queries made by each client, across all sessions in an epoch, and stops responding after the bound q is reached, meaning that even adversarial clients are restricted to q queries per epoch. In the fixed-delay and exponential-delay approaches, only one query from a client is handled at a time by the KS in a session through the active clients list. If a client makes a second query — even from a different session, while a query is in process, the second query is not processed by the KS, but simply dropped.

Command	Description
SSput(P,F,M)	Stores file contents M as P/F
SSget(P,F)	Gets file P/F
SSlist(P)	Gets metadata of P
SSdelete(P,F)	Delete file F in P
SSsearch(P,F)	Search for file F in P
SScreate(P)	Create directory P
SSmove(P_1,F_1,P_2,F_2)	Move P_1/F_1 to P_2/F_2

Figure 5: API commands exposed by the storage service (SS) used by DupLESS. Here F represents a filename and P is the absolute path in a directory hierarchy.

6 The DupLESS client

The Dupless client works with an SS which implements the interface described in Figure 5 (based on the Dropbox API [39]), and provides an analogous set of commands DLput, DLget, DLlist, etc. Figure 6 gives pseudocode for the DupLESS commands for storing and retrieving a file. We now explain the elements of these commands, and will then discuss how other API commands are handled.

Path and filename encryption. The SS provides a rudimentary file system abstraction. Clients can generate directories, use relative and absolute paths, move files from one directory to another, etc. Following our design goal of supporting as much of the base SS functionality as possible, DupLESS should also support paths, filenames, and related functionalities such as copying files. One option is to treat paths and filenames as non-private, and simply mirror in clear the directory hierarchy and filenames asked for by a user. This has the benefit of simplicity and no path-related overheads, but it relies on users guaranteeing that paths and filenames are, in fact, not confidential. A second option would be to hide the directory structure from the SS by using just a single directory, and storing the client's directory hierarchy and filenames in completely encrypted form using some kind of digest file. But this would increase complexity and decrease performance as one would (essentially) have to build a file system on top of the SS. For example, this would bar use of the SS API to perform filename searches on behalf of DupLESS.
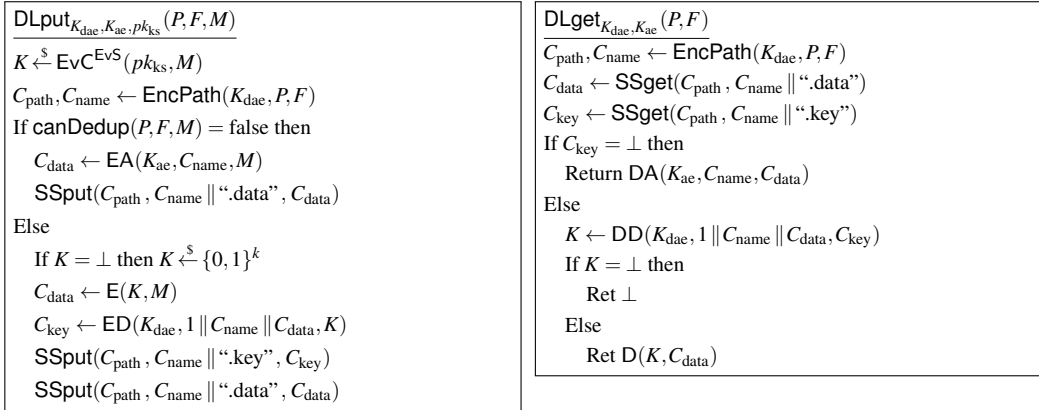
We design DupLESS to provide some security for directory and filenames while still enabling effective use of the SS APIs. To encrypt file and directory names, we use the SIV DAE scheme [68] $\mathsf{SIV} = (\mathsf{ED}, \mathsf{DD})$ with HMAC[SHA256] and CTR[AES]. The EncPath subroutine takes as input a DAE key K_{dae}, a path P (a sequence of directory names separated by '/'), and a filename F, and returns an encrypted path C_{path} and an encrypted filename F. It does so by encrypting each directory D

in P by way of $\mathsf{ED}(K_{\mathrm{dae}}, 0, D)$ and likewise encrypting F by $\mathsf{ED}(K_{\mathrm{dae}}, 0, F)$. (The associated data being set to 0 here will be used to distinguish this use from that of the key encapsulation, see below.) Being deterministic, twice encrypting the same file or directory name results in the same ciphertext. We will then use the ciphertexts, properly encoded into a character set allowed by the SS, as the directory names requested in calls to, e.g., SScreate. We note that the choice of encoding as well as the ciphertext stretch τ_d mean that the maximum filename length supported by DupLESS will be shorter than that of the SS. Should this approach prove limiting, an alternative approach would be to use format-preserving encryption [21] instead to reduce ciphertext expansion.

All this means that we will be able to search for file and directory names and have efficient file copy and move operations. That said, this approach does leak the structure of the plaintext directory hierarchy, the lengths of individual directory and file names, and whether two files have the same name. While length leakage can be addressed with padding mechanisms at a modest cost on storage overhead, hierarchy leakage cannot be addressed without adversely affecting some operations.

Store requests. To store a file with filename F and contents M at path P, the DupLESS client first executes the client portion of the KS protocol (see Section 5). The result is either a message-derived key K or an error message $\perp$. The client then runs a check canDedup to determine whether to use dedupable encryption or non-dedupable encryption. If $K = \perp$ or canDedup returns false, then a random key is selected and will be used in place of a message-derived key. In this case the resulting ciphertext will not be dedupable. We discuss canDedup more below. The client next encrypts M under K with CTR[AES] and a fixed IV to produce ciphertext C_{data}, and then wraps K using SIV to produce ciphertext C_{key}. We include the filename ciphertext C_{name} and C_{data} in order to cryptographically bind together the three ciphertexts. The client uploads to the SS via the SSput command the file "C_{name}.key" with contents C_{key} and C_{data} in file "C_{name}.data". DupLESS encodes the ciphertexts into character sets allowed by the SS API. Both files are uploaded in parallel to the SS. Usually, the SS might require the client to be authorized, and if this is the case, the authorization can be handled when the client starts.

The ".data" file contains only ciphertext C_{data}, and can be deduplicated by the SS assuming K was not replaced by a random value. The ".key" file cannot be deduplicated, its contents being essentially uniformly distributed, but requires only a fixed, small number of bits equal to $k + \tau_d$. With our instantiation choices, this is 384 bits, and does not lead to significant overheads as we show in Section 7.

$$\begin{array}{l}
\underline{\mathsf{DLput}_{K_{\mathrm{dae}},K_{\mathrm{ae}},pk_{\mathrm{ks}}}(P,F,M)}\\[4pt]
K \xleftarrow{\$} \mathsf{EvC}^{\mathsf{EvS}}(pk_{\mathrm{ks}},M)\\[2pt]
C_{\mathrm{path}},C_{\mathrm{name}} \leftarrow \mathsf{EncPath}(K_{\mathrm{dae}},P,F)\\[2pt]
\text{If } \mathsf{canDedup}(P,F,M) = \text{false then}\\[2pt]
\quad C_{\mathrm{data}} \leftarrow \mathsf{EA}(K_{\mathrm{ae}},C_{\mathrm{name}},M)\\[2pt]
\quad \mathsf{SSput}(C_{\mathrm{path}},C_{\mathrm{name}}\,\|\,\text{``.data''},C_{\mathrm{data}})\\[2pt]
\text{Else}\\[2pt]
\quad \text{If } K = \bot \text{ then } K \xleftarrow{\$} \{0,1\}^k\\[2pt]
\quad C_{\mathrm{data}} \leftarrow \mathsf{E}(K,M)\\[2pt]
\quad C_{\mathrm{key}} \leftarrow \mathsf{ED}(K_{\mathrm{dae}},1\,\|\,C_{\mathrm{name}}\,\|\,C_{\mathrm{data}},K)\\[2pt]
\quad \mathsf{SSput}(C_{\mathrm{path}},C_{\mathrm{name}}\,\|\,\text{``.key''},C_{\mathrm{key}})\\[2pt]
\quad \mathsf{SSput}(C_{\mathrm{path}},C_{\mathrm{name}}\,\|\,\text{``.data''},C_{\mathrm{data}})
\end{array}$$

$$\begin{array}{l}
\underline{\mathsf{DLget}_{K_{\mathrm{dae}},K_{\mathrm{ae}}}(P,F)}\\[4pt]
C_{\mathrm{path}},C_{\mathrm{name}} \leftarrow \mathsf{EncPath}(K_{\mathrm{dae}},P,F)\\[2pt]
C_{\mathrm{data}} \leftarrow \mathsf{SSget}(C_{\mathrm{path}},C_{\mathrm{name}}\,\|\,\text{``.data''})\\[2pt]
C_{\mathrm{key}} \leftarrow \mathsf{SSget}(C_{\mathrm{path}},C_{\mathrm{name}}\,\|\,\text{``.key''})\\[2pt]
\text{If } C_{\mathrm{key}} = \bot \text{ then}\\[2pt]
\quad \text{Return } \mathsf{DA}(K_{\mathrm{ae}},C_{\mathrm{name}},C_{\mathrm{data}})\\[2pt]
\text{Else}\\[2pt]
\quad K \leftarrow \mathsf{DD}(K_{\mathrm{dae}},1\,\|\,C_{\mathrm{name}}\,\|\,C_{\mathrm{data}},C_{\mathrm{key}})\\[2pt]
\quad \text{If } K = \bot \text{ then}\\[2pt]
\quad \quad \text{Ret } \bot\\[2pt]
\quad \text{Else}\\[2pt]
\quad \quad \text{Ret } \mathsf{D}(K,C_{\mathrm{data}})
\end{array}$$

Figure 6: DupLESS client procedures for storage and retrieval. They use our server-aided MLE scheme $\mathsf{DupLESSMLE} = (\mathsf{P},\mathsf{K},\mathsf{E},\mathsf{D})$, built with $\mathsf{RSA\text{-}OPRF}[\mathsf{G},\mathsf{H}] = (\mathsf{Kg},\mathsf{EvC},\mathsf{EvS},\mathsf{Vf},\mathsf{Ev})$ along with the DAE scheme $\mathsf{SIV} = (\mathsf{ED},\mathsf{DD})$, and the AE scheme $\mathsf{EtM} = (\mathsf{EA},\mathsf{DA})$. Instantiations are as described in text. The subroutine $\mathsf{canDedup}$ runs dedup heuristics while $\mathsf{EncPath}$ encrypts the path and file name using SIV.

Dedupability control. The $\mathsf{canDedup}$ subroutine enables fine-grained control over which files end up getting deduplicated, letting clients enforce polices such as not deduplicating anything in a personal folder, and setting a lower threshold on size. Our current implementation uses a simple length heuristic: files less than 1 KB in size are not deduplicated. As our experiments show in Section 7, employing this heuristic does not appear to significantly degrade storage savings.

By default, DLput ensures that ciphertexts are of the same format regardless of the output of $\mathsf{canDedup}$. However, should $\mathsf{canDedup}$ mark files non-dedupable based only on public information (such as file length), then we can further optimize performance by producing only a single ciphertext file (i.e. no C_{key}) using an authenticated-encryption scheme with a key K_{ae} derived from the client's secret key. We use AES in CTR mode with random IVs with HMAC in an Encrypt-then-MAC scheme. This provides a slight improvement in storage savings over non-deduped ciphertexts and requires just a single SSput call. We can also query the KS only if needed, which is more efficient.

When $\mathsf{canDedup}$'s output depends on private information (e.g., file contents), clients should always interact with the KS. Otherwise there exists a side channel attack in which a network adversary infers from the lack of a KS query the outcome of $\mathsf{canDedup}$.

Retrieval and other commands. The pseudocode for retrieval is given in Figure 6. It uses $\mathsf{EncPath}$ to recompute the encryptions of the paths and filenames, and then issues SSget calls to retrieve both C_{key} and C_{data}. It then proceeds by decrypting C_{key}, recovering K, and then using it to decrypt the file contents. If non-dedupable encryption was used and C_{key} was not uploaded, the second

SSget call fails and the client decrypts accordingly.

Other commands are implemented in natural ways, and we omit pseudocode for the sake of brevity. DupLESS includes listing the contents of a directory (perform an SSlist on the directory and decrypt the paths and filenames); moving the contents of one directory to another (perform an SSmove command with encrypted path names); search by relative path and filename (perform an $\mathsf{SSsearch}$ using the encryptions of the relative path and filename); create a directory (encrypt the directory name and then use $\mathsf{SScreate}$); and delete (encrypt the path and filename and perform a delete on that).

The operations are, by design, simple and whenever possible, one-to-one with underlying SS API commands. The security guarantees of SIV mean that an attacker with access to the SS cannot tamper with stored data. An SS-based attacker could, however, delete files or modify the hierarchy structure. While we view these attacks as out of scope, we note that it is easy to add directory hierarchy integrity to DupLESS by having $\mathsf{EncPath}$ bind ciphertexts for a directory or file to its parent: just include the parent ciphertext in the associated data during encryption. The cost, however, is that filename search can only be performed on full paths.

In DupLESS, only DLput requires interaction with the KS, meaning that even if the KS goes down files are *never* lost. Even DLput will simply proceed with a random key instead of the message-derived key from the KS. The only penalty in this case is loss of the storage savings due to deduplication.

Other APIs. The interface in Figure 5 is based on the Dropbox API [39]. Google Drive [7] differs by indexing files based on unique IDs instead of names. When a file is uploaded, SSput returns a file ID, which should be

provided to SSget to retrieve the file. The SSlist function returns a mapping between the file names and their IDs. In this case, DupLESS maintains a local map by prefetching and caching file IDs by calling SSlist whenever appropriate; this caching reduces DLget latency. When a file is uploaded, the encrypted filename and returned ID are added to this map. Whenever a local map lookup fails, the client runs SSlist again to check for an update. Hence, the client can start without any local state and dynamically generate the local map.

Supporting keyword search in DupLESS requires additional techniques, such as an encrypted keyword index as in searchable symmetric encryption [34], increasing storage overheads. We leave exploring the addition of keyword search to future work.

7 Implementation and Performance

We implemented a fully functional DupLESS client. The client was written in Python and supports both Dropbox [3] and Google Drive [7]. It will be straightforward to extend the client to work with other services which export an API similar to Figure 5. The client uses two threads during store operations in order to parallelize the two SS API requests. The client takes user credentials as inputs during startup and provides a command line interface for the user to type in commands and arguments. When using Google Drive, a user changing directory prompts the client to fetch the file list ID map asynchronously. We used Python's SSL and Crypto libraries for the client-side crypto operations and used the OPRFv2 KS protocol.

We now describe the experiments we ran to measure the performance and overheads of DupLESS. We will compare both to direct use of the underlying SS API (no encryption) as well as when using a version of DupLESS modified to implement just MLE, in particular the convergent encryption (CE) scheme, instead of DupLESSMLE. This variant computes the message-derived key K by hashing the file contents, thereby avoiding use of the KS. Otherwise the operations are the same.

Test setting and methodology. We used the same machine as for the KS tests (Section 5). Measurements involving the network were repeated 100 times and other measurements were repeated 1,000 times. We measured running times using the timeit Python module. Operations involving files were repeated using files with random contents of size 2^{2i} KB for $i \in \{0, 1, \ldots, 8\}$, giving us a file size range of 1 KB to 64 MB.

Dropbox exhibited significant performance variability in the course of our experiments. For example, the median time to upload a 1 KB file was 0.92 seconds, while the maximum observed was 2.64 seconds, with standard deviation at 0.22 seconds. That is close to 25% of the median. Standard deviation decreases as the file size increases, for example it is only 2% of the median upload time for 32 MB files. We never observed more than 1 Mbps throughput to Dropbox. Google Drive exhibited even slower speeds and more variance.

Storage and retrieval latency. We now compare the time to store and retrieve files using DupLESS, CE, and the plain SS. Figure 7 (top left chart) reports the median time for storage using Dropbox. The latency overhead when storing files with DupLESS starts at about 22% for 1 KB files and reduces to about 11% for 64 MB files.

As we mentioned earlier, Dropbox and Google Drive exhibited significant variation in overall upload and download times. To reduce the effect of these variations on the observed relative performance between DupLESS over the SS, CE over the SS and plain SS, we ran the tests by cycling between the three settings to store the same file, in quick succession, as opposed to, say, running all plain Dropbox tests first. We adopted a similar approach with Google Drive.

We observe that the CE (Convergent Encryption) store times are close to DupLESS store times, since the KSReq step, which is the main overhead of DupLESS w.r.t CE, has been optimized for low latency. For example, median CE latency overhead for 1 KB files over Dropbox was 15%. Put differently, the overhead of moving to DupLESS from using CE is quite small, compared to that of using CE over the base system.

Relative retrieval latencies (bottom left, Figure 7) for DupLESS over Dropbox were lower than the store latencies, starting at about 7% for 1 KB files and reducing to about 6% for 64 MB files.

Performance with Google Drive (Figure 7, top middle chart) follows a similar trend, with overhead for DupLESS ranging from 33% to 8% for storage, and 40% to 10% for retrieval, when file sizes go from 1 KB to 64 MB.

These experiments report data only for files larger than 1 KB, as smaller files are not selected for deduplication by canDedup. Such files are encrypted with non-dedupable, randomized encryption and latency overheads for storage and retrieval in these cases are negligible in most cases.

Microbenchmarks. We ran microbenchmarks on DLput storing 1MB files, to get a breakdown of the overhead. We report median values over 100 trials here. Uploading a 1 MB file with Dropbox takes 2700 milliseconds (ms), while time for the whole DLput operation is 3160 ms, with a 17% overhead. The KSReq latency, from Section 5, is 82 ms or 3%. We measured the total time for all DLput steps except the two SSput operations (refer to Figure 6) to be 135 ms, and uploading the content file on top of this took 2837 ms. Then, net overhead
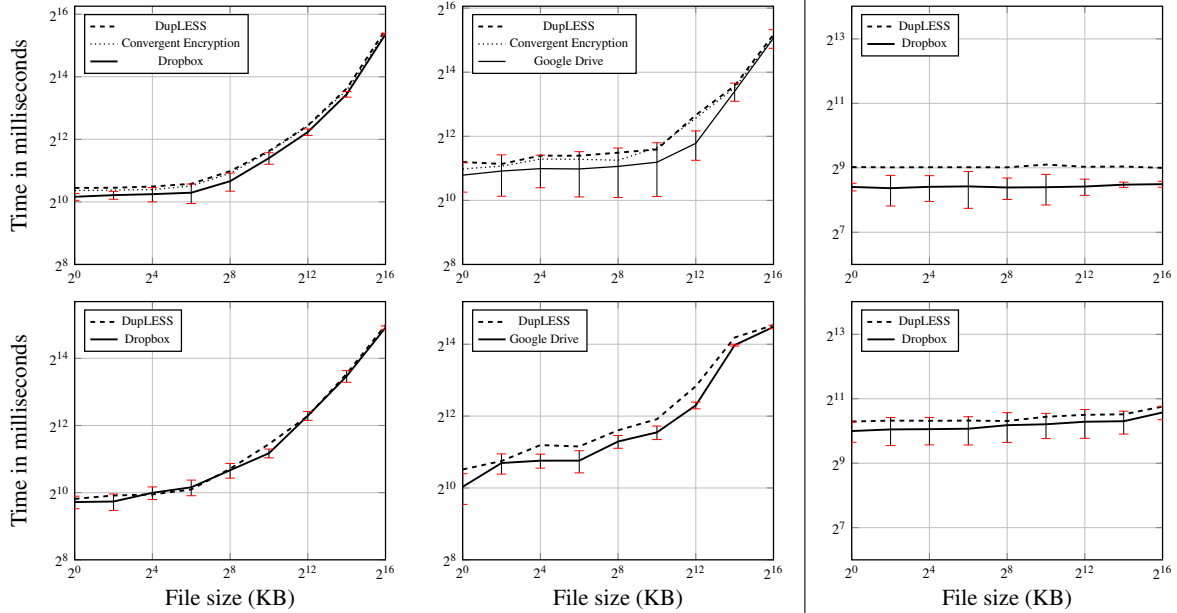
Figure 7: (Left) Median time to store (top two graphs) and retrieve (bottom two graphs) as a function of file size. **(Top Right)** Median time to delete a file as a function of file size. **(Bottom Right)** Median time to copy a file as a function of file size. All axes are log-scale and error bars indicate one standard deviation. Standard deviations are displayed only for base Dropbox/Google Drive times to reduce cluttering.

of KS and cryptographic operations is about 5%, while storing the key file accounts for 12%. Our implementation of DLput stores the content and key files simultaneously, by spawning a new thread for storing the key, and waiting for both the stores to complete before finishing. If DLput exits before the key store thread completes, i.e., if the key is uploaded asynchronously, then the overhead drops to 14%. On the other hand, uploading the files sequentially by storing the content file first, and then storing the key, incurs a 54% overhead (for 1 MB files).

Bandwidth overhead. We measured the increase in transmission bandwidth due to DupLESS during storage. To do so, we used tcpdump and filtered out all traffic unrelated to Dropbox and DupLESS. We took from this the total number of bytes (in either direction). For even very small files, the Dropbox API incurs a cost of about 7 KB per upload. Figure 8 (middle) shows the ratio of bandwidth used by DupLESS to that used by plain Dropbox as file size increases. Given the small constant size of the extra file sent by DupLESS, overhead quickly diminishes as files get larger.

Storage overhead. DupLESS incurs storage overhead, due to the encrypted file name, the MLE key, and the MAC. The sizes of these components are independent of the length of the file. Let n denote the length of the filename in bytes. Then, encrypting the filename with SIV and encoding the result with base64 encoding consumes $2n + 32$ bytes. Repeating the process for the content and key files, and adding extensions brings the file name overhead to $4n + 72 - n = 3n + 72$ bytes. The contents of the key file include the MLE key, which is 16 bytes long in our case, and the 32 byte HMAC output, and hence 48 bytes together. Thus, the total overhead for a file with an n-byte filename is $3n + 120$ bytes. Recall that if the file size is smaller than 1 KB, then canDedup rejects the file for deduplication. In this case, the overhead from encrypting and encoding the file name is $n + 32$ bytes, since only one file is stored. Randomized encryption adds 16 bytes, bringing the total to $n + 48$ bytes.

To assess the overall effect of this in practice, we collected a corpus of around 2,000 public Amazon virtual machine images (AMIs) hosting Linux guests. The AMIs were gathered using techniques similar to those used previously [14, 28], the difference being that we as well downloaded a snapshot of the full file system for each public AMI. There are 101,965,188 unique files across all the AMIs, with total content size of all files being 2,063 GB. We computed cryptographic hashes over the content of all files in the dataset, in order to simulate the storage footprint when using plain deduplication as well as when using DupLESS. This dataset has significant redundancy, as one would expect, given that many AMIs are derivative of other AMIs and so share common files. The plain dedup storage required for the file contents is just 335 GB. DupLESS with the dedupability
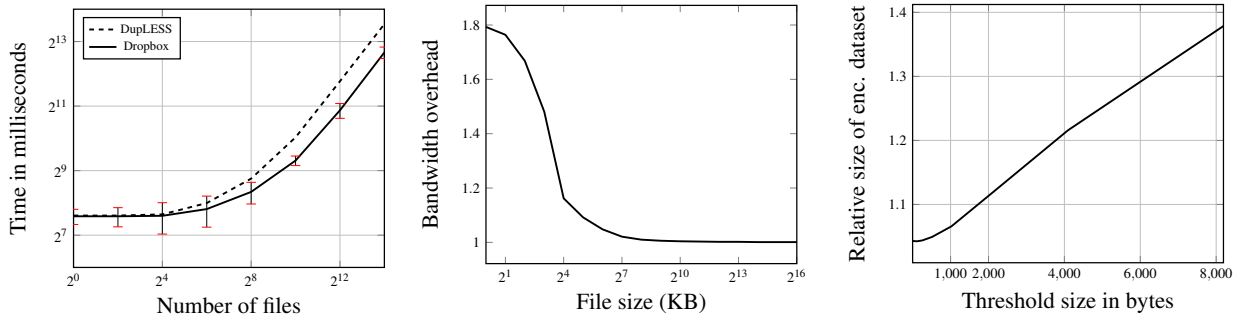
Figure 8: (Left) Median time to list a directory as a function of number of files in the directory. Both axes are logscale and error bars are one standard deviation. **(Middle)** Network bandwidth overhead of DupLESS as a function of file size (log-scale axis) for store operations. **(Right)** The ratio of space required when DupLESS is used for the AMI dataset and when plain dedup is used, as a function of the dedupable threshold length.

length threshold used by canDedup (see Section 6) set to zero (all files were dedupable) requires 350 GB, or an overhead of about 4.5%. In this we counted the size of the filename and path ciphertexts for the DupLESS estimate, though we did not count these in the base storage costs. (This can only inflate the reported overhead.)

We also measure the effect of higher threshold values, when using non-dedupable encryption. Setting the threshold to 100 bytes saves a few hundred megabytes in storage. This suggests little benefit from deduping small files, which is in line with previous observations about deduplication on small files [61].

Figure 8 plots the storage used for a wide range of threshold values. Setting a larger threshold leads to improved security (for those files) and faster uploads (due to one less SSput request) and appears to have, at least for this dataset, only modest impact on storage overheads for even moderately sized thresholds.

The above results may not extend to settings with significantly different workloads. For example, we caution when there is significantly less deduplication across the corpus, DupLESS may introduce greater overhead. In the worst case, when there is no deduplication whatsoever and all 1 KB files with long names of about 100 characters, the overhead will be almost 30%. Of course here one could have canDedup force use of non-dedupable encryption to reduce overhead for all files.

Overhead of other operations. The time to perform DLmove, DLdelete, and DLlist operations are reported in Figure 7 and Figure 8 for Dropbox. In these operations, the DupLESS overheads and the data sent over the network involve just the filenames, and do not depend on the length of the file. (The operations themselves may depend on file length of course.) The overhead of DupLESS therefore remains constant. For DLlist, DupLESS times are close to those of plain Dropbox for folders with twice as many files, since DupLESS stores an extra key

encapsulation file for each user file. We also measured the times for DLsearch and DLcreate, but in these cases the DupLESS overhead was negligible.

8 Security of DupLESS

We argued about the security of the KS protocols and client encryption algorithms in sections 5 and 6. Now, we look at the big picture, the security of DupLESS as a whole. DupLESS provides security that is usually significantly better than current, convergent encryption based deduplicated encryption architectures, and never worse. To expand, security is "hedged," or multi-tiered, and we distinguish three tiers, always assuming that the adversary has compromised the SS and has the ciphertexts.

The optimistic or best case is that the adversary does not have authorized access to the KS. Recall that both OPRFv1 and OPRFv2 need clients to authenticate first, before requesting queries, meaning that in this setting, the attacker cannot obtain any information about message-derived keys. These keys are effectively random to the attacker. In other words, all data stored on the SS is encrypted with random keys, including file contents, names and paths. The attacker can only learn about equality of file contents and the topology of the file system (including file sizes). Thus, DupLESS provides, effectively, semantic security. In particular, security holds even for predictable messages. By using the SIV DAE scheme, and generating tags over the file names, file contents and keys, DupLESS ensures that attempts by the SS to tamper with client data will be detected.

The semi-optimistic, or next best case is that the adversary, having compromised one or more clients, has remote access to the KS but does not have the KS's secret key. Here, security for completely predictable files is impossible. Thus, it is crucial to slow down brute-force attacks and push the feasibility threshold for the attacker. We saw in Section 5 that with the right rate-

limiting setup (Bounded, with appropriate parameters), brute-force attacks can be slowed down significantly. Importantly, attackers cannot circumvent the rate-limiting measures, by say, repeating queries.

Finally, the pessimistic case is that the adversary has compromised the KS and has obtained its key. Even then, we retain the guarantees of MLE, and specifically CE, meaning security for unpredictable messages [18]. Appropriate deployment scenarios, such as locating the KS within the boundary of a large corporate customer of a SS, make the optimistic case the most prevalent, resulting in appreciable security gains without significant increase in cost. The security of non-deduplicated files, file names, and path names is unaffected by these escalations in attack severity.

9 Conclusions

We studied the problem of providing secure outsourced storage that both supports deduplication and resists brute-force attacks. We design a system, DupLESS, that combines a CE-type base MLE scheme with the ability to obtain message-derived keys with the help of a key server (KS) shared amongst a group of clients. The clients interact with the KS by a protocol for oblivious PRFs, ensuring that the KS can cryptographically mix in secret material to the per-message keys while learning nothing about files stored by clients.

These mechanisms ensure that DupLESS provides strong security against external attacks which compromise the SS and communication channels (nothing is leaked beyond file lengths, equality, and access patterns), and that the security of DupLESS gracefully degrades in the face of comprised systems. Should a client be compromised, learning the plaintext underlying another client's ciphertext requires mounting an online brute-force attacks (which can be slowed by a rate-limited KS). Should the KS be compromised, the attacker must still attempt an offline brute-force attack, matching the guarantees of traditional MLE schemes.

The substantial increase in security comes at a modest price in terms of performance, and a small increase in storage requirements relative to the base system. The low performance overhead results in part from optimizing the client-to-KS OPRF protocol, and also from ensuring DupLESS uses a low number of interactions with the SS. We show that DupLESS is easy to deploy: it can work transparently on top of any SS implementing a simple storage interface, as shown by our prototype for Dropbox and Google Drive.

References

[1] Bitcasa, inifinite storage. http://www.bitcasa.com/.

[2] Ciphertite data backup. http://www.ciphertite.com/.

[3] Dropbox, a file-storage and sharing service. http://www.dropbox.com/.

[4] Dupless source code. http://cseweb.ucsd.edu/users/skeelvee/dupless.

[5] The Flud backup system. http://flud.org/wiki/Architecture.

[6] GNUnet, a framework for secure peer-to-peer networking. https://gnunet.org/.

[7] Google Drive. http://drive.google.com.

[8] ADYA, A., BOLOSKY, W., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J., HOWELL, J., LORCH, J., THEIMER, M., AND WATTENHOFER, R. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 1–14.

[9] AMAZON. Amazon Elastic Block Store (EBS). http://aws.amazon.com/ebs.

[10] AMAZON. Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2.

[11] AMAZON. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3.

[12] ANDERSON, P., AND ZHANG, L. Fast and secure laptop backups with encrypted de-duplication. In *Proc. of USENIX LISA* (2010).

[13] ATENIESE, G., BURNS, R. C., CURTMOLA, R., HERRING, J., KISSNER, L., PETERSON, Z. N. J., AND SONG, D. Provable data possession at untrusted stores. In *ACM CCS 07* (Alexandria, Virginia, USA, Oct. 28–31, 2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM Press, pp. 598–609.

[14] BALDUZZI, M., ZADDACH, J., BALZAROTTI, D., KIRDA, E., AND LOUREIRO, S. A security analysis of amazon's elastic compute cloud service. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (2012), ACM, pp. 1427–1434.

[15] BATTEN, C., BARR, K., SARAF, A., AND TREPETIN, S. pStore: A secure peer-to-peer backup system. *Unpublished report, MIT Laboratory for Computer Science* (2001).

[16] BELLARE, M., BOLDYREVA, A., AND O'NEILL, A. Deterministic and efficiently searchable encryption. In *CRYPTO 2007* (Santa Barbara, CA, USA, Aug. 19–23, 2007), A. Menezes, Ed., vol. 4622 of *LNCS*, Springer, Berlin, Germany, pp. 535–552.

[17] BELLARE, M., FISCHLIN, M., O'NEILL, A., AND RISTENPART, T. Deterministic encryption: Definitional equivalences and constructions without random oracles. In *CRYPTO 2008* (Santa Barbara, CA, USA, Aug. 17–21, 2008), D. Wagner, Ed., vol. 5157 of *LNCS*, Springer, Berlin, Germany, pp. 360–378.

[18] BELLARE, M., KEELVEEDHI, S., AND RISTENPART, T. Message-locked encryption and secure deduplication. In *EUROCRYPT 2013, to appear*. Cryptology ePrint Archive, Report 2012/631, November 2012.

[19] BELLARE, M., AND NAMPREMPRE, C. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT 2000* (Kyoto, Japan, Dec. 3–7, 2000), T. Okamoto, Ed., vol. 1976 of *LNCS*, Springer, Berlin, Germany, pp. 531–545.

[20] BELLARE, M., NAMPREMPRE, C., POINTCHEVAL, D., AND SEMANKO, M. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology 16*, 3 (June 2003), 185–215.

[21] BELLARE, M., RISTENPART, T., ROGAWAY, P., AND STEGERS, T. Format-preserving encryption. In *SAC 2009* (Calgary, Alberta, Canada, Aug. 13–14, 2009), M. J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini, Eds., vol. 5867 of *LNCS*, Springer, Berlin, Germany, pp. 295–312.

[22] BELLARE, M., AND ROGAWAY, P. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS 93* (Fairfax, Virginia, USA, Nov. 3–5, 1993), V. Ashby, Ed., ACM Press, pp. 62–73.

[23] BELLARE, M., AND YUNG, M. Certifying permutations: Noninteractive zero-knowledge based on any trapdoor permutation. *Journal of Cryptology 9*, 3 (1996), 149–166.

[24] BISSIAS, G., LIBERATORE, M., JENSEN, D., AND LEVINE, B. N. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Proceedings of the Privacy Enhancing Technologies Workshop* (May 2005), pp. 1–11.

[25] BONEH, D., GENTRY, C., HALEVI, S., WANG, F., AND WU, D. Private database queries using somewhat homomorphic encryption.

[26] BOWERS, K. D., JUELS, A., AND OPREA, A. HAIL: a highavailability and integrity layer for cloud storage. In *ACM CCS 09* (Chicago, Illinois, USA, Nov. 9–13, 2009), E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds., ACM Press, pp. 187–198.

[27] BRAKERSKI, Z., AND SEGEV, G. Better security for deterministic public-key encryption: The auxiliary-input setting. In *CRYPTO 2011* (Santa Barbara, CA, USA, Aug. 14–18, 2011), P. Rogaway, Ed., vol. 6841 of *LNCS*, Springer, Berlin, Germany, pp. 543–560.

[28] BUGIEL, S., NÜRNBERGER, S., PÖPPELMANN, T., SADEGHI, A., AND SCHNEIDER, T. Amazonia: when elasticity snaps back. In *ACM Conference on Computer and Communications Security – CCS '11* (2011), ACM, pp. 389–400.

[29] CAMENISCH, J., NEVEN, G., AND SHELAT, A. Simulatable adaptive oblivious transfer. In *EUROCRYPT 2007* (Barcelona, Spain, May 20–24, 2007), M. Naor, Ed., vol. 4515 of *LNCS*, Springer, Berlin, Germany, pp. 573–590.

[30] CHAUM, D. Blind signatures for untraceable payments. In *CRYPTO'82* (Santa Barbara, CA, USA, 1983), D. Chaum, R. L. Rivest, and A. T. Sherman, Eds., Plenum Press, New York, USA, pp. 199–203.

[31] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. SideChannel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2010), pp. 191–206.

[32] COOLEY, J., TAYLOR, C., AND PEACOCK, A. ABS: the apportioned backup system. *MIT Laboratory for Computer Science* (2004).

[33] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev. 36* (Dec. 2002), 285–298.

[34] CURTMOLA, R., GARAY, J. A., KAMARA, S., AND OSTROVSKY, R. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 06* (Alexandria, Virginia, USA, Oct. 30 – Nov. 3, 2006), A. Juels, R. N. Wright, and S. Vimercati, Eds., ACM Press, pp. 79–88.

[35] DE CRISTOFARO, E., LU, Y., AND TSUDIK, G. Efficient techniques for privacy-preserving sharing of sensitive information. In *Proceedings of the 4th international conference on Trust and trustworthy computing* (Berlin, Heidelberg, 2011), TRUST'11, Springer-Verlag, pp. 239–253.

[36] DE CRISTOFARO, E., SORIENTE, C., TSUDIK, G., AND WILLIAMS, A. Hummingbird: Privacy at the time of twitter. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 285–299.

[37] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 205–220.

[38] DOUCEUR, J., ADYA, A., BOLOSKY, W., SIMON, D., AND THEIMER, M. Reclaiming space from duplicate files in a serverless distributed file system. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on* (2002), IEEE, pp. 617–624.

[39] DROPBOX. Dropbox API Reference. https://www.dropbox.com/developers/reference/api.

[40] DYER, K., COULL, S., RISTENPART, T., AND SHRIMPTON, T. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 332–346.

[41] ERWAY, C. C., KÜPÇÜ, A., PAPAMANTHOU, C., AND TAMASSIA, R. Dynamic provable data possession. In *ACM CCS 09* (Chicago, Illinois, USA, Nov. 9–13, 2009), E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds., ACM Press, pp. 213–222.

[42] GOH, E., SHACHAM, H., MODADUGU, N., AND BONEH, D. Sirius: Securing remote untrusted storage. NDSS.

[43] GOLDWASSER, S., AND MICALI, S. Probabilistic encryption. *Journal of Computer and System Sciences 28*, 2 (1984), 270–299.

[44] GRIBBLE, S. D., MANKU, G. S., ROSELLI, D., BREWER, E. A., GIBSON, T. J., AND MILLER, E. L. Self-similarity in file systems. In *ACM SIGMETRICS Performance Evaluation Review* (1998), vol. 26, ACM, pp. 141–150.

[45] HALEVI, S., HARNIK, D., PINKAS, B., AND SHULMANPELEG, A. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 491–500.

[46] HARNIK, D., PINKAS, B., AND SHULMAN-PELEG, A. Side channels in cloud services: Deduplication in cloud storage. *Security & Privacy, IEEE 8*, 6 (2010), 40–47.

[47] HINTZ, A. Fingerprinting Websites Using Traffic Analysis. In *Proceedings of the Privacy Enhancing Technologies Workshop* (April 2002), pp. 171–178.

[48] ISLAM, M., KUZU, M., AND KANTARCIOGLU, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS12)* (2012).

[49] JIM GUILFORD, KIRK YAP, V. G. Fast SHA256 Implementations on Intel Architecture Processors. http://download.intel.com/embedded/processor/whitepaper/327457.pdf.

[50] JIN, K., AND MILLER, E. L. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, p. 7.

[51] JUELS, A., AND KALISKI JR., B. S. Pors: proofs of retrievability for large files. In *ACM CCS 07* (Alexandria, Virginia, USA, Oct. 28–31, 2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM Press, pp. 584–597.

[52] KAKVI, S., KILTZ, E., AND MAY, A. Certifying rsa. *Advances in Cryptology–ASIACRYPT 2012* (2012), 404–414.

[53] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), pp. 29–42.

[54] KAMARA, S., PAPAMANTHOU, C., AND ROEDER, T. Cs2: A searchable cryptographic cloud storage system. Tech. rep., Technical Report MSR-TR-2011-58, Microsoft, 2011.

[55] KILLIJIAN, M., COURTÈS, L., POWELL, D., ET AL. A survey of cooperative backup mechanisms, 2006.

[56] LEACH, P. J., AND NAIK, D. C. A Common Internet File System (CIFS/1.0) Protocol. http://tools.ietf.org/html/draft-leach-cifs-v1-spec-01.

[57] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference* (2008), pp. 213–226.

[58] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. *Secure untrusted data repository (SUNDR).* Defense Technical Information Center, 2003.

[59] LIBERATORE, M., AND LEVINE, B. N. Inferring the Source of Encrypted HTTP Connections. In *Proceedings of the ACM Conference on Computer and Communications Security* (November 2006), pp. 255–263.

[60] MARQUES, L., AND COSTA, C. Secure deduplication on mobile devices. In *Proceedings of the 2011 Workshop on Open Source and Design of Communication* (2011), ACM, pp. 19–26.

[61] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. *ACM Transactions on Storage (TOS) 7*, 4 (2012), 14.

[62] MICROSYSTEMS, S. NFS: Network File System Protocol Specification. http://tools.ietf.org/html/rfc1094.

[63] MOZY. Mozy, a file-storage and sharing service. http://mozy.com/.

[64] NAOR, M., AND REINGOLD, O. Number-theoretic constructions of efficient pseudo-random functions. In *38th FOCS* (Miami Beach, Florida, Oct. 19–22, 1997), IEEE Computer Society Press, pp. 458–467.

[65] PANCHENKO, A., NIESSEN, L., ZINNEN, A., AND ENGEL, T. Website Fingerprinting in Onion Routing-based Anonymization Networks. In *Proceedings of the Workshop on Privacy in the Electronic Society* (October 2011), pp. 103–114.

[66] RAHUMED, A., CHEN, H., TANG, Y., LEE, P., AND LUI, J. A secure cloud backup system with assured deletion and version control. In *Parallel Processing Workshops (ICPPW), 2011 40th International Conference on* (2011), IEEE, pp. 160–167.

[67] ROGAWAY, P. Authenticated-encryption with associated-data. In *ACM CCS 02* (Washington D.C., USA, Nov. 18–22, 2002), V. Atluri, Ed., ACM Press, pp. 98–107.

[68] ROGAWAY, P., AND SHRIMPTON, T. A provable-security treatment of the key-wrap problem. In *EUROCRYPT 2006* (St. Petersburg, Russia, May 28 – June 1, 2006), S. Vaudenay, Ed., vol. 4004 of *LNCS*, Springer, Berlin, Germany, pp. 373–390.

[69] SEARS, R., VAN INGEN, C., AND GRAY, J. To blob or not to blob: Large object storage in a database or a filesystem? *arXiv preprint cs/0701168* (2007).

[70] SHACHAM, H., AND WATERS, B. Compact proofs of retrievability. In *ASIACRYPT 2008* (Melbourne, Australia, Dec. 7–11, 2008), J. Pieprzyk, Ed., vol. 5350 of *LNCS*, Springer, Berlin, Germany, pp. 90–107.

[71] STORER, M., GREENAN, K., LONG, D., AND MILLER, E. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability* (2008), ACM, pp. 1–10.

[72] SUN, Q., SIMON, D. R., WANG, Y.-M., RUSSELL, W., PADMANABHAN, V. N., AND QIU, L. Statistical Identification of Encrypted Web Browsing Traffic. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2002), pp. 19–30.

[73] VAN DER LAAN, W. Dropship. https://github.com/driverdan/dropship.

[74] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST12)* (2012).

[75] WANG, W., LI, Z., OWENS, R., AND BHARGAVA, B. Secure and efficient access to outsourced data. In *Proceedings of the 2009 ACM workshop on Cloud computing security* (2009), ACM, pp. 55–66.

[76] WILCOX-O'HEARN, Z. Convergent encryption reconsidered, 2011. http://www.mail-archive.com/cryptography@metzdowd.com/msg08949.html.

[77] WILCOX-O'HEARN, Z., PERTTULA, D., AND WARNER, B. Confirmation Of A File Attack. https://tahoe-lafs.org/hacktahoelafs/drew_perttula.html.

[78] WILCOX-O'HEARN, Z., AND WARNER, B. Tahoe: The least-authority filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability* (2008), ACM, pp. 21–26.

[79] XU, J., CHANG, E.-C., AND ZHOU, J. Leakage-resilient client-side deduplication of encrypted data in cloud storage. Cryptology ePrint Archive, Report 2011/538, 2011. http://eprint.iacr.org/.