Accelerating Encrypted Deduplication via SGX

Yanjing Ren and Jingwei Li, *University of Electronic Science and Technology of China;*
Zuoru Yang and Patrick P. C. Lee, *The Chinese University of Hong Kong;*
Xiaosong Zhang, *University of Electronic Science and Technology of China*

https://www.usenix.org/conference/atc21/presentation/ren-yanjing

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the
2021 USENIX Annual Technical Conference
is sponsored by USENIX.

Accelerating Encrypted Deduplication via SGX

Yanjing Ren[†], Jingwei Li[†,*], Zuoru Yang[‡], Patrick P. C. Lee[‡], and Xiaosong Zhang[†]
[†]*University of Electronic Science and Technology of China* [‡]*The Chinese University of Hong Kong*

Abstract

Encrypted deduplication preserves the deduplication effectiveness on encrypted data and is attractive for outsourced storage. However, existing encrypted deduplication approaches build on expensive cryptographic primitives that incur substantial performance slowdown. We present SGXDedup, which leverages Intel SGX to speed up encrypted deduplication based on server-aided message-locked encryption (MLE), while preserving security via SGX. SGXDedup implements a suite of secure interfaces to execute MLE key generation and proof-of-ownership operations in SGX enclaves. It also proposes various designs to support secure and efficient enclave operations. Evaluation on synthetic and real-world workloads shows that SGXDedup achieves significant speedups and maintains high bandwidth and storage savings.

1 Introduction

Outsourcing storage management to the cloud is a common practice for clients (enterprises or individuals) to save the overhead of self-managing massive data, and *security* and *storage efficiency* are two major goals for practical outsourced storage. To satisfy both goals, we explore *encrypted deduplication*, a paradigm that combines encryption and deduplication by always encrypting duplicate plaintext chunks (from the same or different clients) into duplicate ciphertext chunks with a key derived from the chunk content itself; for example, the key can be the cryptographic hash of the corresponding chunk [25]. Thus, any duplicate ciphertext chunk can be eliminated via deduplication for storage efficiency, while all outsourced chunks are encrypted against unauthorized access. Encrypted deduplication is particularly suitable for backup applications, which carry high content redundancy [59] and are attractive use cases for outsourced storage [35, 38, 58].

Existing encrypted deduplication approaches often incur high performance overhead to achieve security guarantees. We use DupLESS [14], a state-of-the-art encrypted deduplication system, as a representative example to explain the performance concerns (see §2.1 for elaboration). First, to prevent adversaries from inferring the content-derived keys, DupLESS employs *server-aided key management*, in which it deploys a dedicated key server to manage key generation requests from clients. However, server-aided key management requires expensive cryptographic operations to prevent the key server from knowing the plaintext chunks and the keys during key generation. Second, to prevent adversaries from obtaining unauthorized access to ciphertext chunks by inferring the deduplication pattern (a.k.a. side-channel attacks [32, 33]), DupLESS can take one of the following approaches: it either (i) performs *target-based deduplication* (i.e., uploading all ciphertext chunks and letting the cloud remove any duplicate ciphertext chunks) so that the deduplication pattern is protected from any (malicious) client, or (ii) performs *source-based deduplication* (i.e., removing all duplicate ciphertext chunks on the client side without being uploaded to the cloud) and additionally proves to the cloud that it is indeed the owner of the ciphertext chunks (i.e., has access to the full contents of the corresponding plaintext chunks) and is authorized to perform deduplication on the ciphertext chunks. The former incurs extra communication bandwidth for uploading duplicate ciphertext chunks, while the latter incurs expensive cryptographic operations for proving that the client is the owner of the ciphertext chunks. Although various protocol designs have been proposed to address the performance issues of encrypted deduplication, they often weaken security [23, 41, 60], add bandwidth overhead [33, 42], or degrade storage efficiency [41, 51, 62] (see §6 for details).

The advances of hardware-assisted trusted execution [1–3, 57] provide new opportunities to improve the performance of encrypted deduplication. In particular, we focus on Intel Software Guarded Extensions (SGX), which provides a *trusted execution environment (TEE)*, called an *enclave*, for processing code and data with confidentiality and integrity guarantees [13]. Given that SGX achieves reasonably high performance with proper configurations [34], we are motivated to offload the expensive cryptographic operations of encrypted deduplication by directly running sensitive operations in enclaves, so as to improve the performance of encrypted deduplication, while maintaining its security, bandwidth efficiency, and storage efficiency.

We propose SGXDedup, a high-performance SGX-based encrypted deduplication system. SGXDedup builds on server-aided key management as in DupLESS [14], but executes efficient cryptographic operations inside enclaves. Realizing the design of SGXDedup has non-trivial challenges. First, it is critical to securely bootstrap enclaves for hosting trusted code and data, yet attesting the authenticity of enclaves incurs significant delays. Second, each client needs to communicate via a secure channel with the enclave inside the key server, yet the management overhead of the secure channels increases with the number of clients. Finally, clients may renew or revoke cloud service subscriptions, so allowing dynamic client authentication is critical. To this end, we implement three

*Corresponding author: Jingwei Li (jwli@uestc.edu.cn)

major building blocks for SGXDedup:

- *Secure and efficient enclave management*: It protects against the compromise of the key server and allows a client to quickly bootstrap an enclave after a restart.
- *Renewable blinded key management*: It generates a blinded key for protecting the communication between the enclave inside the key server and each of the clients based on key regression [30], such that the blinded key is renewable for dynamic client authentication.
- *SGX-based speculative encryption*: It offloads the online encryption/decryption overhead of secure channel management via speculative encryption [27].

We evaluate our SGXDedup prototype using synthetic and real-world [5, 45] workloads. It achieves significant speedups by offloading cryptographic operations to enclaves (e.g., a 131.9× key generation speedup over the original key generation scheme in DupLESS [14]). It also achieves 8.1× and 9.6× speedups over DupLESS [14] for the uploads of non-duplicate and duplicate data, respectively, and has up to 99.2% of bandwidth/storage savings in real-world workloads.

We release the source code of our SGXDedup prototype at: **http://adslab.cse.cuhk.edu.hk/software/sgxdedup**.

2 Background and Problem

We present background details and formal definitions for encrypted deduplication (§2.1) and Intel SGX (§2.2). We also present the threat model addressed in this paper (§2.3).

2.1 Encrypted Deduplication

Basics. We consider *chunk-based deduplication* [45, 59, 63], which is widely deployed in modern storage systems to eliminate content redundancy. It works by partitioning an input file into non-overlapping *chunks*. For each chunk, it computes the cryptographic hash of the chunk content (called the *fingerprint*). It tracks the fingerprints of all currently stored chunks in a *fingerprint index*. It only stores a physical copy of the chunk if the fingerprint is new to the fingerprint index, or treats the chunk as a duplicate if the fingerprint has been tracked, assuming that fingerprint collisions are highly unlikely in practice [17].

Encrypted deduplication extends chunk-based deduplication with encryption to provide both data confidentiality and storage efficiency for outsourced cloud storage. A *client* encrypts each *plaintext chunk* of the input file with some symmetric secret key into a *ciphertext chunk* and stores all ciphertext chunks in the *cloud* (or any remote storage site), which manages deduplicated storage for ciphertext chunks. To support file reconstruction, the client creates a *file recipe* that lists the fingerprints, sizes, and keys of the ciphertext chunks. It encrypts the file recipe with its own master secret key and stores the encrypted file recipe in the cloud.

Message-locked encryption (MLE) [15] formalizes a cryptographic primitive for encrypted deduplication. It specifies how the symmetric secret key (called the *MLE key*) is derived from the *content* of a plaintext chunk (e.g., its popular instantiation *convergent encryption (CE)* [25] uses the cryptographic hash of a plaintext chunk as the MLE key). Thus, it encrypts duplicate plaintext chunks into duplicate ciphertext chunks, so that deduplication remains viable on ciphertext chunks.

Server-aided MLE. CE is vulnerable to *offline brute-force attacks*, as its MLE key (i.e., the hash of a plaintext chunk) can be publicly derived. Specifically, an adversary infers the input plaintext chunk from a target ciphertext chunk (without knowing the MLE key) by enumerating the MLE keys of all possible plaintext chunks to check if any plaintext chunk is encrypted to the target ciphertext chunk.

Server-aided MLE [14] is a state-of-the-art cryptographic primitive that strengthens the security of encrypted deduplication against offline brute-force attacks. It deploys a dedicated *key server* for MLE key generation. To encrypt a plaintext chunk, a client first sends the fingerprint of the plaintext chunk to the key server, which returns the MLE key via both the fingerprint and a *global secret* maintained by the key server. If the global secret is secure, an adversary cannot feasibly launch offline brute-force attacks; otherwise, if the global secret is compromised, the security reduces to that of the original MLE. Server-aided MLE further builds on two security mechanisms. First, it uses the *oblivious pseudorandom function (OPRF)* [47] to allow a client send "blinded" fingerprints of the plaintext chunks, such that the key server can still return the same MLE keys for identical fingerprints without learning the original fingerprints. Second, it rate-limits the key generation requests from the clients to protect against *online brute-force attacks*, in which malicious clients issue many key generation requests to the key server, in order to find a target MLE key.

Proof-of-ownership. For bandwidth savings, encrypted deduplication can apply source-based deduplication, instead of target-based deduplication, to remove duplicate ciphertext chunks on the client side without being uploaded to the cloud (§1). The client sends the fingerprints of ciphertext chunks to the cloud, which checks if the fingerprints are tracked by the fingerprint index (i.e., the corresponding ciphertext chunks have been stored). The client then uploads only the non-duplicate ciphertext chunks to the cloud. However, source-based deduplication is vulnerable to *side-channel attacks* [32, 33] when some clients are compromised. One side-channel attack is that a compromised client can query the existence of any target ciphertext chunk (e.g., if the ciphertext chunk corresponds to some possible password [33]) by sending the fingerprint of the ciphertext chunk to the cloud, so as to identify the sensitive information from other clients. Another side-channel attack is that a compromised client can obtain unauthorized access to the stored chunks of other clients. Specifically, it can use the fingerprint of any target ciphertext chunk to convince the cloud that it is the owner of the corresponding ciphertext chunk with full access rights [32].

Proof-of-ownership (PoW) [32] is a cryptographic approach that augments source-based deduplication with protection against side-channel attacks, while maintaining the bandwidth savings of source-based deduplication. Its idea is to let the cloud verify that a client is indeed the owner of a ciphertext chunk and is authorized with the full access to the ciphertext chunk. This ensures that a compromised client cannot query for the existence of other clients' chunks. Specifically, in PoW-based source-based deduplication, a client attaches each fingerprint being sent to the cloud with a *PoW proof*, through which the cloud can verify if the client is the real owner of the corresponding ciphertext chunk. The cloud only responds upon the successful proof verification, thereby preventing any compromised client from identifying the ciphertext chunks owned by other clients.

Limitations. Recall from §1 that existing encrypted deduplication implementations require expensive cryptographic protection. Server-aided MLE necessitates the OPRF protocol [47] to protect the fingerprint information against the key server, yet the OPRF protocol involves expensive public-key cryptographic operations. For example, our evaluation (§5.1) shows that the OPRF-based MLE key generation achieves only up to 25 MB/s (Exp#1) and limits the overall encrypted deduplication performance to 20 MB/s (Exp#4). Also, the existing PoW implementation is based on the Merkle-tree protocol [32], which achieves only 37 MB/s (Exp#3) due to frequent hash computations for chunk-level PoW. In a 1 GbE LAN environment, the computational overhead of PoW even *negates* the performance gain of eliminating the uploads of duplicate data in source-based deduplication. Although we can mitigate PoW computations by applying PoW on a per-file basis (i.e., a client proves its ownership of a file), the cloud cannot verify if a chunk belongs to a file under chunk-based deduplication. Existing solutions that improve the performance of server-aided MLE or PoW often sacrifice security [23, 41, 60], bandwidth efficiency [33, 42], or storage efficiency [41, 51, 62] (§6).

2.2 Intel SGX

We explore hardware-assisted trusted execution to mitigate the performance overhead of encrypted deduplication, while preserving security, bandwidth efficiency, and storage efficiency. In this work, we focus on Intel SGX [3], a suite of security-related instructions built into modern Intel CPUs. SGX shields the execution of code and data in a hardware-protected environment called an *enclave*. In the following, we highlight three security features of an enclave related to our work: isolation, attestation, and sealing.

Isolation. An enclave resides in a hardware-guarded memory region called the *enclave page cache (EPC)* for hosting any protected code and data. An EPC comprises 4KB pages, and any in-enclave application can take up to 96 MB [34]. If an enclave has a larger size than the EPC, it encrypts unused pages and evicts them to the unprotected memory. In this work, we deploy enclaves in each client and the key server to protect sensitive operations (§3.1). We also limit the size of in-enclave contents to mitigate the paging overhead (§3.4).

An enclave provides an interface, namely *enclave call (ECall)*, such that an outside application can issue ECalls to securely access in-enclave contents. Note that ECalls incur context switching overhead for accessing the enclave memory [34]. We reduce the number of ECalls by batching contents for processing (§4).

Attestation. SGX supports *remote attestation* to authenticate a target enclave via a remote entity (e.g., the cloud). In the remote attestation process (see [3] for elaboration), the remote entity needs to contact the attestation service operated by Intel to check the integrity of the enclave information provided by the target enclave. Then the remote entity verifies the target enclave by comparing its enclave information with the trusted configuration expected in the target enclave. We use remote attestation to ensure that the correct code and data are loaded into each enclave in the first bootstrap.

Sealing. SGX protects enclave contents when they are stored outside an enclave via sealing. It uses a secret *sealing key* to encrypt the data before being evicted. The sealing key can be derived from either the *measurement hash* (i.e., a SHA256 hash on the enclave contents) or the author identity of the enclave, so that only the corresponding enclave can access the sealing key and decrypt the sealed data. Since remote attestation incurs significant delays (Exp#7), we use sealing to eliminate remote attestation after the first bootstrap of an enclave (§3.2).

Remarks. We do not consider memory encryption-based TEEs (e.g., AMD SEV [1] and MK-TME [2]), since they need a large trusted computing base and expose a broad attack surface [46]. Also, AMD SEV [1] does not protect memory integrity, and is vulnerable to the attack that a privileged adversary can manipulate encrypted memory pages [46].

2.3 Threat Model

Adversarial capabilities. We start with the server-aided MLE architecture [14] with multiple clients, the key server, and the cloud. Our major security goal is to achieve data confidentiality for outsourced cloud storage [14] against a *semi-honest* adversary that infers the original plaintext chunks via the following malicious actions:

- The adversary can compromise the key server and learn the MLE key generation requests issued by each client. It can also access the global secret to infer the original chunks in outsourced storage via offline brute-force attacks [14].
- The adversary can compromise one or multiple clients and send arbitrary MLE key generation requests to query the MLE keys of some target chunks [14]. It can also launch side-channel attacks against some target chunks [33] (§2.1), so as to infer the original plaintext chunks owned by other non-compromised clients.

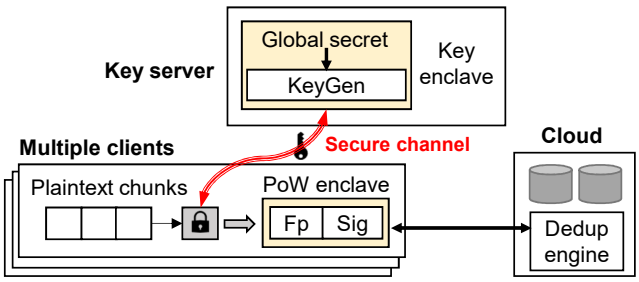


Figure 1: Overview of SGXDedup architecture: a key enclave and a PoW enclave are deployed in the key server and each client, respectively.

Assumptions. We make the following threat assumptions. (i) All communications among the clients, the key server, and the cloud are protected against tampering (e.g., via SSL/TLS). (ii) SGX is trusted and reliable; denial-of-service or side-channel attacks against SGX [18, 48] are beyond our scope. (iii) We can achieve both *integrity* via remote auditing [11, 36] and *fault tolerance* via a multi-cloud approach [42]. (iv) We do not consider traffic analysis [64], frequency analysis [41] and chunk size leakage [54], while the related defenses [41,54,64] are compatible to our design.

3 SGXDedup Design

SGXDedup is a high-performance SGX-based encrypted deduplication system designed for outsourced storage, particularly for backup workloads with high content redundancy. It aims for the following design goals: (i) *confidentiality*, which protects the outsourced chunks and keys against unauthorized access even when the key server or any client are compromised, as in server-aided MLE in DupLESS [14] and source-based deduplication with PoW [32]; (ii) *bandwidth/storage efficiency*, which removes all duplicate chunks across multiple clients before uploads, as in source-based deduplication. (iii) *computational efficiency*, which mitigates the computational overhead of the cryptographic operations and achieves significantly higher performance than existing software-based encrypted deduplication designs.

3.1 Overview

Architecture. Figure 1 presents the architecture of SGXDedup, which introduces two enclaves: the *key enclave* and the *PoW enclave*. SGXDedup deploys a key enclave in the key server to manage and protect the global secret of server-aided MLE against a compromised key server. To perform MLE key generation, both the key enclave and a client first establish a secure channel based on a shared *blinded key* (see §3.3 for details how a blinded key is formed). The client then submits the fingerprint of a plaintext chunk via the secure channel. The key enclave computes the MLE key as the cryptographic hash of both the global secret and the fingerprint. It returns the MLE key via the secure channel.

The key enclave benefits both performance and security. It avoids the expensive OPRF protocol [14] of server-aided MLE during MLE key generation. Also, it protects the fingerprints and MLE keys via a secure channel based on a shared blinded key, such that the key server cannot learn any information from the MLE key generation process. Furthermore, it protects the global secret in the enclave memory, and preserves security even if the key server is compromised (note that the security of the original server-aided MLE degrades when the key server is compromised; see §2.1).

SGXDedup deploys a PoW enclave in each client to prove the authenticity of the ciphertext chunks in source-based deduplication. The PoW enclave first sets up a shared *PoW key* with the cloud (we currently implement the key agreement using Diffie-Hellman key exchange (DHKE); see §4). After MLE key generation, the client encrypts each plaintext chunk into a ciphertext chunk. The PoW enclave takes the ciphertext chunk as input, computes the corresponding fingerprint, and creates a signature of the fingerprint using the shared PoW key with the cloud. The client then uploads both the fingerprint and the signature to the cloud. The cloud verifies the authenticity of the fingerprint based on the corresponding signature and the PoW key. Only when the fingerprint is authenticated, the cloud proceeds to check if the fingerprint corresponds to any already stored duplicate ciphertext chunk. Note that we verify the client's ownership of ciphertext chunks rather than that of plaintexts (e.g., [32]), so as to protect the original information from the cloud. Ensuring the ownership of ciphertext chunks is enough for security, since MLE applies one-to-one mappings and the ownership of a ciphertext chunk is consistent with that of the corresponding plaintext chunk.

The PoW enclave again benefits both performance and security. It avoids the computational overhead for the cryptographic PoW constructions. It also protects the deduplication patterns against malicious clients, as the patterns are only returned given the authenticated fingerprints.

Note that prior studies [24, 31, 37] perform key generation and encryption inside an enclave, while we opt to perform encryption in unprotected memory. The main reason is that both the original plaintext chunks and the encryption process are co-located within a client. Moving the encryption process to the enclave does not improve the security, as compromising the client can also access its plaintext chunks, yet it adds significant computational overhead to the enclave.

Questions. Realizing SGX-based encrypted deduplication efficiently is non-trivial, since we need to mitigate the potential performance overhead of SGX that would otherwise negate the overall performance benefits. Here, we pose three questions, which we address based on a suite of ECalls for the key enclave and the PoW enclave (Table 1).

- How should the enclaves be securely and efficiently bootstrapped? (§3.2)
- How should the key enclave and each client establish a secure channel? (§3.3)

ECall Name	Description
Key enclave	
Secret generation	Generate a global secret (§3.2)
Rekeying	Renew a blinded key (§3.3)
Nonce checking	Check the uniqueness of a nonce (§3.4)
Key generation	Return the MLE keys (§3.4)
Mask generation	Pre-compute masks (§3.4)
PoW enclave	
Key unsealing	Unseal a PoW key (§3.2)
Key sealing	Seal a PoW key into disk (§3.2)
Proof generation	Sign the ciphertext chunk fingerprints (§4)

Table 1: Major ECalls in SGXDedup.

- How should the key enclave reduce its computational overhead of managing the secure channels of clients? (§3.4)

3.2 Enclave Management

SGXDedup establishes trust in all enclaves via the cloud when it is first initialized. Before SGXDedup is deployed, we first compile the enclave code into shared objects [3], append each shared object with a signature (for integrity verification), and distribute the shared objects to the key server and each of the clients. The cloud also hosts the shared objects for subsequent verification. The key server creates the key enclave, while each client creates its own PoW enclave by loading the corresponding shared object. The cloud authenticates each enclave via remote attestation (§2.2) to ensure that the correct code is loaded. Here, we address two specific management issues: (i) how the global secret (§3.1) is securely bootstrapped into the key enclave; and (ii) how each client efficiently bootstraps its PoW enclave after a restart.

Key enclave management. Instead of bootstrapping the global secret in entirety, SGXDedup generates the global secret in the key enclave based on two *sub-secrets* respectively owned by the cloud and the key server, so as to prevent either of them from learning the whole global secret.

To generate the global secret, we hard-code the cloud's sub-secret into the key enclave code, and deliver the code (as a shared object) to the key server during SGXDedup's initialization. We also implement a *secret generation ECall* for the key enclave, so as to allow the key server to provide its own sub-secret. The ECall can only be issued by the key server after the cloud's sub-secret is included into the key enclave. It takes the key server's sub-secret as its single input, and hashes the concatenation of the key server's sub-secret and the cloud's sub-secret to form the global secret. Note that the key server cannot access the enclave code and hence cannot learn the cloud's sub-secret hard-coded inside the enclave (assuming that reverse engineering is impossible). Thus, even if the key server is compromised, the global secret remains secure, so the security of server-aided MLE is preserved. If both the key server and the cloud are simultaneously compromised, the security of SGXDedup reduces to that of the original MLE (§2.1).

PoW enclave management. When a client bootstraps its PoW enclave, it needs to attest the authenticity of the PoW enclave. However, remote attestation generally incurs a very large latency (e.g., about 9 s; see §5.1) to connect to the Intel service. Unlike the key enclave, whose remote attestation is only done once during initialization, the client needs to bootstrap and terminate the PoW enclave each time it joins and leaves SGXDedup, respectively. If remote attestation were used each time when the client joins, its substantial overhead will hurt usability.

SGXDedup leverages sealing to avoid remote attestation after the first bootstrap of the PoW enclave. Recall that the PoW enclave shares a PoW key with the cloud, such that the cloud can verify the authenticity of fingerprints (§3.1). Our idea is to seal the PoW key based on the measurement hash of the PoW enclave. Thus, when the client bootstraps again its PoW enclave, it unseals the PoW key into the bootstrapped PoW enclave. As long as the PoW key is recovered successfully, the authenticity of the bootstrapped PoW enclave is verified.

Specifically, the client first checks whether any sealed PoW key is locally available in its physical machine. If a sealed PoW key is not available (the first bootstrap), the client attests the PoW enclave via remote attestation and exchanges a PoW key with the cloud; otherwise if a sealed PoW key is available (after the first bootstrap), the client creates a new PoW enclave by loading the shared object, and calls the *key unsealing ECall* of the new PoW enclave to unseal the PoW key. The key unsealing ECall takes the address of the sealed PoW key as input. It derives a sealing key based on the measurement hash of the new PoW enclave, decrypts the sealed PoW key, and keeps it in the new PoW enclave.

When the client leaves SGXDedup, its PoW enclave needs to be terminated. The client issues the *key sealing ECall* to seal the PoW key. The key sealing ECall encrypts the PoW key based on the measurement hash of the PoW enclave, and stores the result in the address provided by the client.

3.3 Renewable Blinded Key Management

Each client securely communicates with the key enclave via a shared *blinded* key to prevent the eavesdropping by the key server (§3.1). To form the blinded key, a straightforward approach is to directly implement a key agreement protocol between the key enclave and each client. However, the key enclave needs to authenticate the client on-the-fly (e.g., a client may renew or revoke its cloud service subscription). Such dynamic authentication puts performance burdens on the key enclave.

SGXDedup manages blinded keys with the help of the cloud. During initialization, the cloud hard-codes a blinded secret κ into the key enclave code. Each client downloads a *key state* (derived from κ; see below) from the cloud, and generates its blinded key (based on the key state) for the secure communication with the key enclave. Our rationales are two-fold. First, when the client issues any download request

from the cloud, the cloud can check if the client is authorized. Second, we can derive a sequence of renewable blinded keys from κ (instead of directly using κ), so as to prevent the revoked/compromised clients from persistently accessing the key enclave. As a side benefit, the renewable key management also prevents online brute-force attacks (§2.1) without actively slowing down the key generation rate [14].

SGXDedup uses *key regression* [30] to derive renewable blinded keys, while ensuring that the blinded keys in each client and the key enclave are consistent. Specifically, key regression works on a sequence of key states $S[1], S[2], \ldots, S[m]$, each of which can be used to derive a key (e.g., via hashing). It allows the key enclave and the cloud to perform *rekeying* to derive a new state from an old state (e.g., deriving $S[2]$ from $S[1]$) using a *key regression secret*, such that the client cannot learn any information about new states without knowing the key regression secret. It also allows each client to derive any old state from the new state (e.g., deriving $S[1]$ from $S[2]$).

To realize key regression in SGXDedup, we use κ as the key regression secret shared between the cloud and the key enclave for deriving the new states and keys. In each upload, the client first downloads the up-to-date key state $S[i]$ from the cloud, and requests the current version number j of the blinded key accepted by the key enclave. Given that the key enclave may not renew the blinded key in time (e.g., it is busy with serving key generation exactly in the scheduled rekeying time), j is typically smaller than i (i.e., $S[j]$ is prior to $S[i]$). Then the client derives $S[j]$ from $S[i]$ and the corresponding blinded key $K[j]$, and communicates with the key enclave based on the same $K[j]$. Note that the cloud can derive $K[j]$, but it cannot eavesdrop the communication between each client and the key enclave, since the communication is additionally protected via the SSL/TLS-based channel between the client and the key server (see our assumptions in §2.3).

SGXDedup renews blinded keys in both the cloud and the key enclave. The cloud implements a timer to trigger rekeying over periodic time intervals. The key server issues the *rekeying ECall* to trigger rekeying when a scheduled rekeying time is reached.

Currently, we implement the hash-based key regression scheme [30] for high key derivation performance. Specifically, we define a parameter n (now set as 2^{20} [30]) to indicate the maximum number of affordable rekeying times. The cloud and the key enclave compute the i-th key state as $S[i] = \mathbf{H}^{n-i+1}(\kappa)$, and each client derives the corresponding blinded key as $K[i] = \mathbf{H}(S[i]||0^8)$, where $\mathbf{H}^{n-i+1}()$ iteratively calls the cryptographic hash function $\mathbf{H}()$ by $n-i+1$ times, and $||$ is the concatenation operator. To derive old states, the client downloads $S[i]$ and recovers $S[i-1] = \mathbf{H}(S[i])$.

3.4 SGX-Based Speculative Encryption

Given the shared blinded key (§3.3), the key enclave manages a secure channel with each client to protect the transferred fingerprints/keys during MLE key generation (§3.1). However, managing the secure channels, particularly with many clients, incurs high encryption/decryption costs in the key enclave. SGXDedup augments the secure communication channel management using *speculative encryption* [27] in the context of SGX, so as to offload the encryption/decryption overhead of the key enclave.

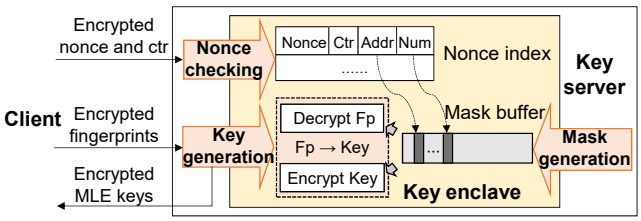


Figure 2: Overview of SGX-based speculative encryption.

Speculative encryption. Speculative encryption [27] adopts encryption/decryption in *counter mode* [6] and pre-computes partial encryption/decryption results in an offline procedure, so as to reduce the online computational overhead in standalone cryptographic file systems. To encrypt a plaintext M, we first partition M into a sequence of plaintext blocks $b_1, b_2, \ldots, b_m$ (e.g., each block has a fixed size of 16 bytes). For each client, we pick a unique *nonce* θ that can be used only once by the encryption with the same key. We then compute the *mask* for the i-th plaintext block as $e_i = \mathbf{E}(K, \theta||i)$, where $\mathbf{E}()$ is a symmetric encryption function, K is the key, i is the *counter* (for counter-mode encryption), and '$||$' is the concatenation operator. Finally, we compute each ciphertext block $c_i = e_i \oplus b_i$, where $\oplus$ is the bitwise XOR operator, and form the whole ciphertext $C = c_1||c_2||\ldots||c_m$. To decrypt the ciphertext, we generate the mask e_i for each block like above, recover the corresponding plaintext block $b_i = e_i \oplus c_i$ and hence the original plaintext M. Since the mask generation step is independent of each target plaintext/ciphertext, we can pre-compute the masks offline, followed by applying the lightweight XOR operation for online encryption/decryption.

Integration. To realize speculative encryption in SGXDedup, we need to address the nonce management issue. A unique nonce serves as an unpredictable "one-time-pad" that makes each counter-mode encryption output look random [6]. If counter-mode encryption is used by multiple clients, we need to associate each client with a unique nonce in its encryption/decryption operations. However, since different clients are isolated, it is challenging to ensure that the nonces associated with the clients are unique.

To ensure the uniqueness of a nonce, SGXDedup manages a centralized key-value store, called the *nonce index*, in the key enclave. Each entry of the nonce index maps a stored nonce (12 bytes) to three fields: its counter (4 bytes), the starting address (8 bytes) of the corresponding mask, and the number (4 bytes) of its available masks. Also, SGXDedup implements a *nonce checking ECall* (Figure 2) that can be called by the key server to compare the nonce submitted by

each client with the previously stored nonces in the nonce index and inform the client to re-pick a new one if a duplicate nonce is found. Note that the in-enclave nonce index can be effectively managed, since it can serve up to 112,000 clients (assuming one nonce per client) with only 3 MB EPC space (default configuration of SGXDedup).

SGXDedup applies speculative encryption to establish a secure channel between a client and the key enclave for the protection of MLE key generation. To initialize the secure channel, the client synchronizes its self-picked nonce θ and the corresponding counter i with the key enclave, where i is initialized as zero if θ has not been used for any encryption/decryption by the client. Specifically, it encrypts θ and i with the up-to-date blinded key (§3.3), computes a message authentication code (MAC) based on the resulting ciphertexts, and submits both the ciphertexts and the MAC to the key server. Here, we adopt *encrypt-then-MAC* [16] to detect any outdated blinded key by checking the MAC for any information transferred between the client and the key enclave.

The key server issues the nonce checking ECall, which takes the client's uploads as input, decrypts θ and i, and checks the decrypted θ and i with the nonce index:

- Case I: If θ is duplicate and $i = 0$, this indicates the re-use of an existing nonce, and the ECall returns a signal to inform the client to re-pick a new nonce.
- Case II: If θ is duplicate and $i \neq 0$, this implies that the nonce has been stored. The ECall updates the stored counter and marks the corresponding pre-computed masks (to be used for follow-up processing, as elaborated below).
- Case III: If θ is unique, this implies that the nonce is new, and the ECall adds θ into the nonce index.

For Cases II and III, the ECall accepts the communication and asks the client to transfer fingerprints for MLE key generation (§3.1). The client encrypts the fingerprints based on θ and i, and uploads the results; after encrypting each fingerprint block, the client increments i by one to prevent replay attacks. As shown in Figure 2, the key server issues the *key generation ECall* to process the encrypted fingerprints. The ECall checks if some masks are marked for the current client. If found (i.e., Case II), it uses the marked masks to decrypt the fingerprints and encrypt the generated MLE keys. Otherwise (i.e., Case III), it computes the masks online for decryption and encryption.

Mask pre-computation. To speed up encryption/decryption, the key enclave performs mask pre-computation when a new blinded key is applied (i.e., all existing masks are invalid) or a client has connected again after the last mask generation (i.e., some of its masks have been consumed). The key enclave calls the *mask generation ECall* (Figure 2), which pre-computes the masks for a number (e.g., three in our case) of most-recently-used nonces and writes the results into a *mask buffer*. By default, we configure the mask buffer with up to 90 MB. Suppose that each mask takes 16 bytes and the average chunk size is 8 KB. The MLE key generation of a fingerprint consumes four masks: two are for decrypting the 32-byte fingerprint and the other two are for encrypting the resulting 32-byte MLE key. Thus, the pre-computed masks in the mask buffer can be used to process the fingerprints of up to 11.25 GB of data.

4 Implementation

We build an SGXDedup prototype in C++ using OpenSSL 1.1.1g [49], Intel SGX SDK 2.7 [3] and Intel SGX SSL [7]. Our prototype implements fingerprinting operations for plaintext and ciphertext chunks via SHA256, and chunk-based encryption via AES256. It contains about 14.2 K LoC.

Setup. To bootstrap the key enclave, the cloud hard-codes both the cloud's sub-secret (§3.2) and the blinded secret (§3.3) into the key enclave code. Alternatively, SGXDedup can also provision both secrets (using the secret provisioning functionality of SGX [3]) after authenticating the key enclave, at the expense of incurring extra bootstrapping overhead. The key enclave uses SHA256 to generate the global secret and implement the hash function in key regression. Each PoW enclave implements DHKE in NIST P-256 elliptic curve to share a PoW key with the cloud.

Key generation. Each client implements Rabin fingerprinting [52] for content-defined chunking. We fix the minimum, average, and maximum chunk sizes in Rabin fingerprinting at 4 KB, 8 KB, and 16 KB, respectively. To implement SGX-based speculative encryption (§3.4), we fix the nonce and the counter at 12 bytes and 4 bytes, respectively, and implement MACs using HMAC-SHA256. The key enclave generates the MLE key of each plaintext chunk via SHA256.

Deduplication. SGXDedup realizes source-based deduplication coupled with PoW. The PoW enclave implements a *proof generation ECall* to compute the fingerprints of ciphertext chunks and generate a signature based on the resulting fingerprints using AES-CMAC. The cloud implements the fingerprint index (§2.1) as a key-value store based on LevelDB [4]. To mitigate network and disk I/O costs, we store (non-duplicate) ciphertext chunks in 8 MB containers as units of transfers and storage [43].

Optimization. To reduce context switching and SGX memory encryption/decryption overhead, each client batches multiple fingerprints (4,096 by default) to transfer in the secure communication channel with the key enclave (§3.4). The key enclave processes each received batch of fingerprints. It accesses the batch via a pointer without copying the contents into the enclave [34]. Similarly, the PoW enclave processes the ciphertext chunks on a per-batch basis (4,096 by default) without content copy. We also use multi-threading to boost performance. Each client parallelizes the processing of chunking, fingerprinting of plaintext chunks, encryption, PoW, and uploads via multi-threading, while the key enclave and the cloud serve multiple connections in different threads.

5 Evaluation

We configure a LAN cluster of machines for multiple clients, the key server, and the cloud. Each machine has a quad-core 3.0 GHz Intel Core i5-7400 CPU, a 1 TB 7200 RPM SATA hard disk, and 8 GB RAM. All machines run Ubuntu 18.04 and are connected with 10 GbE. We evaluate SGXDedup using both synthetic (§5.1) and real-world (§5.2) workloads. We summarize the main results as follows.

- SGXDedup achieves high MLE key generation performance in both single-client (Exp#1) and multi-client (Exp#2) cases. For example, it achieves a 131.9× speedup over OPRF-RSA adopted by DupLESS's MLE key generation [14] in the single-client case.
- SGXDedup has 2.2× and 8.2× computational PoW speedups over universal hash-based PoW [60] (that only achieves weaker security) and Merkle-tree-based PoW [32] (Exp#3).
- SGXDedup has high overall performance in single-client (Exp#4 and Exp#5) and multi-client (Exp#6) cases. We also provide a time breakdown of SGXDedup in uploads (Exp#7). For example, in a 10 GbE LAN testbed, SGXDedup incurs only a slowdown of 17.5% in uploads compared to a plain deduplication system without any security protection; in real-cloud deployment (Exp#5), SGXDedup incurs a slowdown of 13.2% and its performance is bounded by the Internet bandwidth.
- SGXDedup is efficient for processing real-world workloads (Exp#8 and Exp#9). For example, its upload performance overhead over plain deduplication (without security protection) is within 22.0%; it also achieves high bandwidth savings over existing approaches [33, 42], by an absolute difference of up to 91.4%.

5.1 Evaluation on Synthetic Workloads

We generate a synthetic dataset with a set of files, each of which comprises globally non-duplicate chunks. By default, we set the file size as 2 GB (except for Exp#2, in which we stress-test the MLE key generation performance). A client uploads or downloads a file via the cloud. To avoid the performance impact of disk I/O, we store the file data in memory rather than on disk (except for Exp#5, in which we process on-disk files in real-cloud deployment). We plot the average results over 10 runs. We also include the 95% confidence intervals from *Student's t-Distribution* into bar charts (for brevity, we exclude them from line charts).

Exp#1 (Single-client MLE key generation). We evaluate MLE key generation in two rounds. First, a client creates the plaintext chunks of a 2 GB file via Rabin fingerprinting (§4) and issues MLE key generation requests. It then repeats the MLE key generation process for the chunks of a different 2 GB file. The difference is that the second round uses the pre-computed masks for MLE key generation (§3.4).

We compare the single-client MLE key generation speed of SGXDedup with state-of-the-arts. We consider two OPRF-based key generation approaches, namely *OPRF-BLS* [10] and *OPRF-RSA* [14], which implement the OPRF primitive (§2.1) based on blind BLS and blind RSA signatures, respectively. We also consider two relaxed key generation approaches, namely *MinHash encryption* [51] and *TED* [41], which trade storage efficiency and security for performance. Specifically, MinHash encryption generates MLE keys using OPRF-RSA on a per-segment basis, where the average segment size is configured as 1 MB. TED generates the MLE key for each chunk based on the sketch-based frequency counting of the short hashes of the chunk (c.f. §3.3 in [41]).

Figure 3 shows the results. SGXDedup outperforms all baseline approaches, by avoiding the expensive cryptographic primitives in OPRF-BLS, OPRF-RSA, and MinHash encryption and the frequency counting computations in TED. Its first round achieves 1,583× and 131.9× speedups over OPRF-BLS and OPRF-RSA, respectively. The speedups are 9.4× and 3.7× over MinHash encryption and TED, respectively, even though the latter two sacrifice storage efficiency and security. Speculative encryption in the second round improves the MLE key generation speed of the first round by 67.8%.

Exp#2 (Multi-client MLE key generation). We evaluate multi-client MLE key generation. For stress-testing, we configure a single machine that runs multiple threads, each of which simulates a client, to simultaneously issue MLE key generation requests to the key server (which runs on a different machine). Recall that the pre-computed masks in the key enclave can be used to efficiently process at most 11.25 GB of data (§3.4). To enable speculative encryption for each simulated client in the second round, we configure each thread to generate 40,960 fingerprints (i.e., 320 MB of raw data for 8 KB chunks) and configure the key enclave to equally pre-compute masks for each simulated client after the first round of key generation. We measure the *aggregate* MLE key generation speed on processing all MLE key generation requests from all simulated clients.

Figure 4 shows the results. The aggregate key generation speeds of both rounds initially increase with the number of simulated clients. At peak, the first and second rounds achieve 8.5×10^5 keys/s and 29×10^5 keys/s for five and ten simulated clients, respectively. After ten clients, the aggregate speeds drop due to aggravated context switching overhead. On average, speculative encryption in the second round achieves 4.4× aggregate key generation speedup over the first round.

Exp#3 (Computational PoW). We evaluate the PoW performance. We consider a single client that performs PoW on a 2 GB file. The client creates plaintext chunks from the file, encrypts each plaintext chunk, and issues PoW requests to the cloud. We measure the *computational PoW speed* based on the total computational time of all chunks in both the client (where the PoW enclave performs fingerprinting on the ciphertext chunks and signs the resulting fingerprints) and the cloud (which verifies the authenticity of the fingerprints); note that we exclude the network transfer time between the

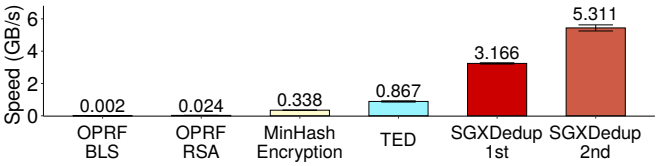


Figure 3: (Exp#1) Single-client MLE key generation.

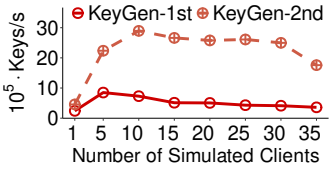


Figure 4: (Exp#2) Multi-client MLE key generation.

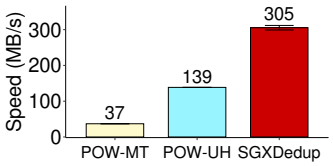


Figure 5: (Exp#3) Computational PoW.

client and the cloud in our speed calculation (we consider the network transfer time in Exp#11 in Appendix).

We compare SGXDedup with two state-of-the-art PoW approaches: (i) *PoW-MT* [32] (a.k.a. the basic version in [32]), a Merkle-tree-based PoW approach that encodes the chunks with erasure coding and builds a Merkle tree over the encoded content for PoW; and (ii) *PoW-UH* [60], which builds on universal hashing but trades security for performance. For fair comparisons, we implement both PoW-MT and PoW-UH in C++ by ourselves. Note that there are improved PoW approaches [32], but they incur even higher performance overhead for low memory usage.

Figure 5 shows the results. SGXDedup dramatically outperforms PoW-MT, since it avoids erasure coding and Merkle tree construction in the client, as well as Merkle-tree-based verification in the cloud. It achieves an 8.2× speedup over PoW-MT. It also achieves a 2.2× speedup over PoW-UH.

Exp#4 (Single-client uploads and downloads). We consider a single client, and compare the upload and download performance of SGXDedup with two baseline systems: (i) *PlainDedup*, which disables the key generation, encryption and PoW operations of SGXDedup, and hence realizes source-based deduplication without any security protection; and (ii) *DupLESS* [14], which generates per-chunk MLE keys via OPRF-RSA, performs encryption and uploads all ciphertext chunks to the cloud for deduplication. Since the original implementation of DupLESS does not provide a deduplicated storage backend (it assumes that Dropbox is used), we implement DupLESS in C++ based on the design described in [14] by ourselves. Note that PlainDedup retrieves files based on file recipes that are not encrypted, and differs from the two-round downloads in encrypted deduplication systems (i.e., both SGXDedup and DupLESS); in the latter, the client first downloads and decrypts the file recipe, followed by downloading the chunks to reconstruct the file (§2.1).

We evaluate the upload and download speeds in three steps: (i) a client first uploads a 2 GB file; (ii) the client restarts and then uploads another 2 GB file that is identical to the previous one; and (iii) the client downloads the file. Note that the second upload performs source-based deduplication (for both PlainDedup and SGXDedup), and leverages the pre-computed masks to accelerate key generation (only for SGXDedup).

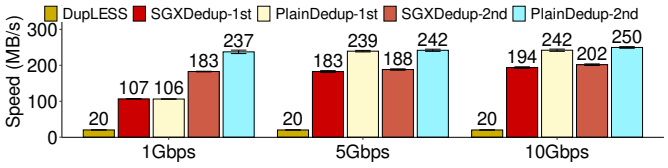
Figure 6(a) shows the upload speeds for different network bandwidths controlled by `trickle` [28]. For the first upload, when the network bandwidth is 1 Gbps, the upload speeds of both SGXDedup (106.6 MB/s) and PlainDedup (106.2 MB/s) are bound by the network speed, while the performance bottleneck of DupLESS (20.1 MB/s) is the OPRF-RSA-based key generation (Exp#1). When the network bandwidth increases to 10 Gbps (the default), the upload speeds of SGXDedup and PlainDedup achieve 193.6 MB/s and 242.0 MB/s, respectively, while that of DupLESS keeps stable at 20.0 MB/s. For the second upload, DupLESS achieves the same speed as the first upload due to its key generation performance bottleneck. The upload speeds of both SGXDedup and PlainDedup are less influenced by the network bandwidth since they do not need to transfer data. On average, SGXDedup achieves 8.1× and 9.6× speedups over DupLESS in the first and the second uploads, respectively. Even compared with the insecure PlainDedup, SGXDedup only incurs about 17.5% and 21.4% drops of the corresponding upload speeds. The overhead comes from the security mechanisms of SGXDedup, including key generation, encryption, and PoW.

Figure 6(b) shows the download speeds. As the network bandwidth increases to 10 Gbps, both SGXDedup and DupLESS achieve 323.1 MB/s, a 44.2% drop from PlainDedup. The reason is that they serially retrieves and decrypts the file recipe, followed by downloading the ciphertext chunks.

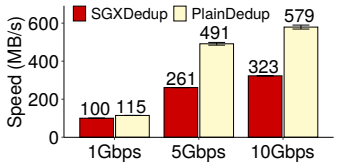
Exp#5 (Real-cloud uploads and downloads). We now extend Exp#4 and evaluate the upload and download speeds in a real-cloud deployment. Specifically, we deploy the client and the key server in our LAN testbed (§5), and connect the client via the Internet to the *Alibaba Cloud*, in which we rent a virtual machine *ecs.c6e.xlarge* to run the cloud. The cloud machine is equipped with a quad-core 3.2 GHz CPU (Intel Xeon Cascade Lake Platinum 8269CY in its host platform), 8 GB memory. We mount the cloud with *Alibaba General Purpose NAS* as the storage backend. The NAS can achieve up to 20000 IOPS for 4 K random reads and writes.

We use on-disk data files for uploads (as opposed to Exp#4, which loads files into client's memory before uploads), and allow the cloud to store the received data files in NAS. We also use `scp` to upload the whole data file (i.e., 2 GB) from the client to the cloud, so as to provide a data transfer benchmark in the Internet environment.

Table 2 shows the results. In the first upload, the performance of all systems (11.4 MB/s for SGXDedup, 11.6 MB/s for PlainDedup and 10.8 MB/s for DupLESS) is bounded by Internet bandwidth (11.9 MB/s). In the second upload, SGXDedup achieves 104.3 MB/s, 9.7× speedup over DupLESS and 13.2% drop compared to PlainDedup. Note that the performance differences are smaller than those in Exp#4, since



(a) Upload



(b) Download

Figure 6: (Exp#4) Single-client uploads and downloads. We exclude the second upload speed of DupLESS (which has the same performance in two uploads) and the download speed of DupLESS (which is identical to that of SGXDedup).

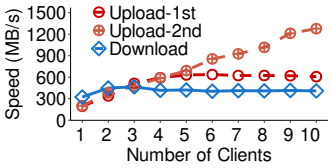


Figure 7: (Exp#6) Multi-client uploads and downloads.

Approach	First Upload	Second Upload	Download
Transfer	11.9 ± 0.03		
SGXDedup	11.4 ± 0.3	104.3 ± 1.2	10.1 ± 0.1
PlainDedup	11.6 ± 0.1	120.1 ± 1.4	11.3 ± 0.3
DupLESS	10.8 ± 0.2		10.1 ± 0.1

Table 2: (Exp#5) Real-cloud upload and download (unit: MB/s).

SGXDedup and PlainDedup are now bounded by client-side disk I/Os. In download, the performance of all three systems is bounded by Internet bandwidth again, while SGXDedup and DupLESS degrade the download speed of PlainDedup by 10.6% due to their serial retrieval and decryption (Exp#4).

Exp#6 (Multi-client uploads and downloads). We now consider multiple clients that issue uploads/downloads concurrently. We focus on SGXDedup, and configure the key enclave to equally pre-compute masks for each client after the first upload. We fix the network bandwidth at 10 Gbps, and evaluate the *aggregate* upload and download speeds for all clients to complete the uploads/downloads.

Figure 7 shows the results with up to ten clients. The aggregate upload speed in the second round increases with the number of clients, and reaches 1277.1 MB/s. On the other hand, the aggregate upload speed in the first round increases to 637.0 MB/s for seven clients, followed by dropping to 620.3 MB/s for ten clients due to the write contention across clients. Similarly, the aggregate download speed finally drops to 408.8 MB/s for the read contention of multiple clients.

Exp#7 (Time breakdown). We present a time breakdown of SGXDedup to study the performance of different steps. Suppose that the key enclave has been started and we focus on the initialization and the upload procedures of a single client. The initialization procedure bootstraps the PoW enclave of the client. The upload procedure includes the following steps: (i) *chunking*, which partitions an input file into plaintext chunks; (ii) *fingerprinting-p*, which computes the fingerprints of plaintext chunks; (iii) *key generation*, which generates MLE keys from the key enclave; (iv) *encryption*, which encrypts the plaintext chunks; (v) *fingerprinting-c*, in which the PoW enclave computes the fingerprints of ciphertext chunks; (vi) *signing*, in which the PoW enclave computes the signature of the fingerprints; (vii) *verification*, in which the cloud verifies the authenticity of received fingerprints; (viii) *deduplication*, in which the cloud detects duplicate ciphertext chunks and informs the client; and (ix) *transfer*, which uploads the non-duplicate ciphertext chunks and the file recipe.

Procedure/Step		First Upload	Second Upload
Initialization		9.38 ± 2.72 s	0.80 ± 0.004 s
Chunking		3.77 ± 0.15 ms	
Fingerprinting-p		3.24 ± 0.28 ms	
Key generation		0.31 ± 0.01 ms	0.18 ± 0.01 ms
Encryption		2.47 ± 0.10 ms	
PoW	Fingerprinting-c	3.28 ± 0.01 ms	
	Signing	0.01 ± 0.00004 ms	
	Verification	0.005 ± 0.00003 ms	
Deduplication		0.38 ± 0.03 ms	0.48 ± 0.03 ms
Transfer		1.29 ± 0.09 ms	0.05 ± 0.01 ms

Table 3: (Exp#7) Time breakdown per 1 MB of file data processed: fingerprinting-p and fingerprinting-c are operated on plaintext and ciphertext chunks, respectively.

Table 3 presents the results (per 1 MB of file data processed). The initialization procedure is time-consuming in the first upload since it needs to contact the Intel attestation service for checking the integrity of the PoW enclave. When the client restarts again, it no longer needs to execute remote attestation and reduces the setup time of the first round by 91.5%. Note that the time overhead of the initialization procedure can be amortized across multiple uploads and downloads.

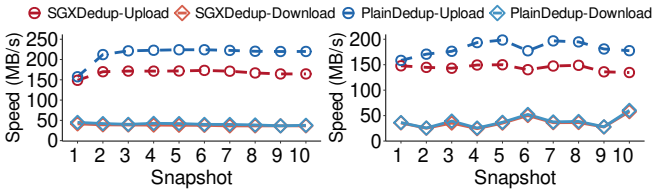
The key generation step is efficient, and takes up to 2.1% of the overall upload time. With speculative encryption, SGXDedup further reduces the key generation time of the first upload by 41.9%. Also, the overall PoW step takes up to 24.4% of the overall upload time, and the dominant computation in PoW is the fingerprinting of ciphertext chunks, which is necessary for finding duplicates in encrypted deduplication. With the lightweight signing and verification steps (that take up to 0.1% of the overall upload time), we protect source-based deduplication against side-channel attacks, while reducing the transfer time of the first upload by 96.1%.

Additional experiments. In our technical report [53], we study the impact of batch size on key generation and PoW performance, as well as the rekeying latency.

5.2 Evaluation on Real-world Workloads

We evaluate SGXDedup using real-world workloads (which contain duplicates). We consider two real-world datasets in our evaluation.

- The first dataset, called *FSL*, contains the backup snap-



(a) FSL dataset (b) MS dataset

Figure 8: (Exp#8) Trace-driven upload and download performance.

shots of user home directories from a shared file system at the File systems and Storage Lab (FSL) [5, 56]. Each snapshot lists the 48-bit fingerprints of the chunks of an average chunk size of 8 KB. We select all snapshots (under `fslhomes`) from January 22 to June 17, 2013, covering a total of 56.2 TB of pre-deduplicated data. The dataset reduces to 431.9 GB after deduplication (i.e., a 133.2× deduplication ratio).

- The second dataset, called *MS*, contains the Windows file system snapshots from Microsoft [45]. Each snapshot lists the 40-bit fingerprints of the chunks of an average chunk size of 8 KB. We sample 140 snapshots from the original 857 snapshots, such that each snapshot has a size of about 100 GB. Our dataset contains 14.4 TB of pre-deduplicated data. It reduces to 2.4 TB after deduplication (i.e., a 6× deduplication ratio).

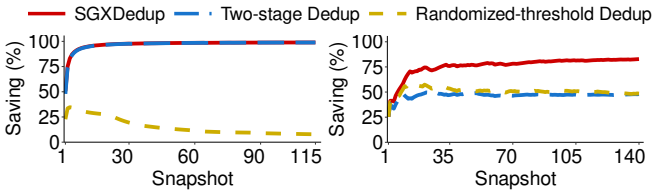
Exp#8 (Trace-driven performance). We conduct trace-driven evaluation on the upload and download performance of SGXDedup. We choose ten snapshots from FSL and MS each as follows. For FSL, we pick the snapshots from the same user to have high cross-snapshot redundancies; for MS, we pick the snapshots that have the most intra-snapshot redundancies. The chosen FSL and MS snapshots take 1.3 TB and 1.0 TB of pre-deduplicated data, respectively. Since our snapshots only contain fingerprints without actual data, we reconstruct each plaintext chunk by repeatedly writing its fingerprints into a spare chunk until reaching the specified chunk size, so the same (distinct) fingerprint returns the same (distinct) chunk. We use a single client to upload the snapshots one by one, followed by downloading them in the same order of uploads. We compare SGXDedup with PlainDedup (Exp#4).

Figure 8(a) shows the upload and download speeds across FSL snapshots. Both SGXDedup and PlainDedup achieve high upload speeds, especially when uploading the subsequent snapshots (e.g., at least 164.5 MB/s for SGXDedup and 212.2 MB/s for PlainDedup) after the first snapshot (e.g., 148.8 MB/s for SGXDedup and 157.5 MB/s for PlainDedup). The reason is that the FSL dataset has high redundancies across snapshots, and both systems can upload less data for subsequent snapshots. On average, SGXDedup incurs an upload performance drop of 22.0% compared to PlainDedup. Note that the overhead is slightly larger than that (17.5-21.4%) in our performance evaluation (Exp#4) using synthetic workloads. The reason is that chunking is now disabled in trace-driven evaluation, and the bottleneck for SGXDedup switches to PoW (see Table 3), while the bottleneck for PlainDedup is fingerprinting. The download speed decreases from 41.3 MB/s to 36.4 MB/s for SGXDedup, and from 45.0 MB/s to 37.9 MB/s for PlainDedup, mainly due to chunk fragmentation [43]. We can mitigate chunk fragmentation via re-writing and caching [19, 43].

Figure 8(b) shows the upload and download speeds across MS snapshots. Both systems have lower upload speeds than in the FSL dataset, since the MS dataset has more non-duplicate chunks (e.g., 28.7 M for MS versus 18.3 M for FSL) and aggravates the access overhead of the fingerprint index. On average, SGXDedup incurs an upload slowdown of 21% compared to PlainDedup. Note that the download speeds fluctuate (e.g., 24.3-57.5 MB/s for SGXDedup and 25.6-60.3 MB/s for PlainDedup), since some MS snapshots have more non-duplicate chunks and are likely to be stored in consecutive regions that can be accessed quickly via sequential reads (i.e., less chunk fragmentation [43]).

Exp#9 (Bandwidth savings). We evaluate the bandwidth efficiency of SGXDedup. We consider two existing approaches that defend against side-channel attacks using both source-based deduplication (i.e., the client performs deduplication and uploads only non-duplicate ciphertext chunks to the cloud) and target-based deduplication (i.e., the client uploads all ciphertext chunks to the cloud, which performs deduplication on the received ciphertext chunks): (i) *two-stage deduplication* [42], which applies source-based deduplication on individual users, followed by target-based deduplication across users; and (ii) *randomized-threshold deduplication* [33], which performs either source-based deduplication or target-based deduplication based on a randomly chosen threshold. We choose the upper and lower bounds of the threshold in randomized-threshold deduplication as 20 and 2, respectively, as in [33]. For FSL, we aggregate the same-day snapshots of different users, and add each aggregate snapshot into the storage in the order of its creation time. For MS, we add each snapshot based on its snapshot ID (we assume that each MS snapshot is from a distinct user). We measure the *bandwidth savings* as the fraction of data reduction in transmission compared to the pre-deduplicated data. Here, we do not consider the bandwidth overhead due to metadata.

Figure 9 shows the cumulative bandwidth saving after uploading each snapshot. After uploading all snapshots, SGXDedup achieves 99.2% and 83.2% of bandwidth savings in FSL and MS, respectively. Since SGXDedup performs source-based deduplication, its bandwidth savings are also translated to the storage savings. Two-stage deduplication has almost identical bandwidth savings to SGXDedup in FSL, since the FSL dataset includes a large volume of intra-user redundancies. On the other hand, in MS, two-stage deduplication only has 47.9% of bandwidth savings (less than SGXDedup by an absolute difference of 35.3%). Randomized-threshold deduplication has varying bandwidth savings, with 48.5% in MS



(a) FSL dataset (b) MS dataset

Figure 9: (Exp#9) Cumulative bandwidth savings after each snapshot is stored.

but only 7.8% in FSL (less than SGXDedup by an absolute difference of 34.7% and 91.4%, respectively).

6 Related Work

MLE key management. Traditional MLE-based [15] encrypted deduplication systems (e.g., [8,20,55]) are vulnerable to offline brute-force attacks [14]. DupLESS [14] proposes server-aided MLE to perform MLE key generation in a dedicated key server. Follow-up studies on server-aided MLE focus on deduplication pattern attestation [10], cross-user deduplication [62], and MLE key renewal [51].

Some studies mitigate the overhead of chunk-based MLE key generation at the expense of degrading deduplication effectiveness [51, 62] or weakening security [40]. SGXDedup outperforms these approaches in performance, while preserving both deduplication effectiveness and security (§5.1). Some other MLE key generation approaches include threshold-based key management [26] and decentralized key management [44], but they build on the cryptographic primitives (e.g., threshold signature [26] and password-authenticated key exchange [44]) that are theoretically proven but not readily implemented.

Defenses against side-channel attacks. Source-based deduplication is bandwidth-efficient but vulnerable to side-channel attacks [33]. Prior studies [33, 42] combine source-based deduplication and target-based deduplication to defend against side-channel attacks, while SGXDedup achieves significantly more bandwidth savings by purely performing source-based deduplication (§5.2) and using PoW to protect against side-channel attacks. Also, SGXDedup is much more efficient than Merkle-tree-based PoW (§5.1). Some other studies make PoW efficient by relaxing security (e.g., [23, 60]), while SGXDedup uses client-side SGX to preserve the security of PoW.

SGX-based storage. SGX [3] has been widely used for securing storage systems. PESOS [39] enforces the access policies of object storage with SGX. OBLIVIATE [9] enhances the security of SGX-based file systems against privileged side-channel attacks. EnclaveDB [50] and ObliDB [29] protect outsourced databases against information leakage via SGX. NEXUS [24] enables fine-grained access control with SGX over untrusted cloud storage. On the performance side, Harnik *et al.* [34] propose guidelines on mitigating the performance overhead of SGX implementations. ShieldStore [37] implements application-specific data management to limit the enclave memory usage. SPEICHER [12] is an SGX-based LSM-based key-value store with efficient I/O operations.

All the above studies do not consider deduplication. Dang *et al.* [22] propose proxy-based protocols for bandwidth-efficient encrypted deduplication, but the protocols do not address the key generation performance overhead and have no implementation. SPEED [21] leverages deduplication to make SGX computations efficient, but SGXDedup improves the performance of encrypted deduplication with SGX. Other studies use a cloud-side enclave for PoW verification [61] and secure file-based deduplication [31], while SGXDedup uses a client-side enclave for efficient PoW proof generation and supports more fine-grained chunk-based deduplication.

7 Conclusion

This paper addresses the performance overhead of encrypted deduplication via SGX. We present SGXDedup, which implements a set of ECalls to run sensitive operations in SGX enclaves, so as to accelerate encrypted deduplication while preserving security. SGXDedup incorporates three key designs: (i) the secure and efficient enclave management, (ii) the renewable blinded key management, and (iii) the design of SGX-based speculative encryption for lightweight computations. Evaluation on our SGXDedup prototype demonstrates its high performance in synthetic and real-world workloads.

Acknowledgments

We thank our shepherd, Russell Sears, and the anonymous reviewers for their valuable comments. We thank Changchun Li for his help in the prototype evaluation, and Cheng Li for his feedbacks on the paper draft. This work was supported in part by grants by the National Natural Science Foundation of China (61972073), the Key Research Funds of Sichuan Province (2020YFG0298, 2021YFG0167), Sichuan Science and Technology Program (2020JDTD0007), the Fundamental Research Funds for Chinese Central Universities (ZYGX2020ZB027), Innovation and Technology Fund (ITS/315/18FX), and CUHK Direct Grant 2020/21 (4055148).

References

[1] AMD secure encrypted virtualization (SEV). `https://developer.amd.com/sev/`.

[2] Intel architecture memory encryption technologies specification. `https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf?source=techstories.org`.

[3] Intel software guard extensions SDK. `https://software.intel.com/en-us/sgx/sdk`.

[4] LevelDB. `https://github.com/google/leveldb`.

[5] Traces and snapshots public archive. http://tracer.filesystems.org.

[6] Using advanced encryption standard (AES) counter mode with ipsec encapsulating security payload (ESP). https://tools.ietf.org/html/rfc3686.

[7] Intel software guard extensions SSL. https://github.com/intel/intel-sgx-ssl, 2017.

[8] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of ACM OSDI*, 2002.

[9] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. Obliviate: A data oblivious file system for Intel SGX. In *Proc. of NDSS*, 2018.

[10] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef. Transparent data deduplication in the cloud. In *Proc. of ACM CCS*, 2015.

[11] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. of ACM CCS*, 2007.

[12] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing LSM-based key-value stores using shielded execution. In *Proc. of USENIX FAST*, 2019.

[13] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *Proc. of USENIX OSDI*, 2014.

[14] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: server-aided encryption for deduplicated storage. In *Proc. of USENIX Security*, 2013.

[15] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *Proc. of EuroCrypt*, 2013.

[16] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Proc. of AsiaCrypt*, 2000.

[17] J. Black. Compare-by-hash: A reasoned analysis. In *Proc. of USENIX ATC*, 2006.

[18] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proc. of USENIX Security*, 2018.

[19] Z. Cao, H. Wen, F. Wu, and D. H. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proc. of USENIX FAST*, 2018.

[20] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. of ACM OSDI*, 2002.

[21] H. Cui, H. Duan, Z. Qin, C. Wang, and Y. Zhou. SPEED: Accelerating enclave applications via secure deduplication. In *Proc. of ICDCS*, 2019.

[22] H. Dang and E.-C. Chang. Privacy-preserving data deduplication on trusted processors. In *Proc. of IEEE CLOUD*, 2017.

[23] R. Di Pietro and A. Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *Proc. of ACM ASIACCS*, 2012.

[24] J. B. Djoko, J. Lange, and A. J. Lee. NEXUS: Practical and secure access control onuntrusted storage platforms using client-side SGX. In *Proc. of IEEE/IFIP DSN*, 2019.

[25] 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 *Proc. of IEEE ICDCS*, 2002.

[26] Y. Duan. Distributed key generation for encrypted deduplication: Achieving the strongest privacy. In *Proc. of ACM CCSW*, 2014.

[27] V. Eduardo, L. C. E. de Bona, and W. M. N. Zola. Speculative encryption on GPU applied to cryptographic file systems. In *Proc. of USENIX FAST*, 2019.

[28] M. A. Eriksen. Trickle: A userland bandwidth shaper for UNIX-like systems. In *Proc. of USENIX ATC*, 2005.

[29] S. Eskandarian and M. Zaharia. ObliDB: Oblivious query processing for secure databases. In *Proc. of ACM VLDB*, 2019.

[30] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proc. of NDSS*, 2006.

[31] B. Fuhry, L. Hirschoff, S. Koesnadi, and F. Kerschbaum. SeGShare: Secure group file sharing in the cloud using enclaves. In *Proc. of IEEE/IFIP DSN*, 2020.

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

[33] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[34] D. Harnik, E. Tsfadia, D. Chen, and R. Kat. Securing the storage data path with SGX enclaves. https://arxiv.org/abs/1806.10883, 2018.

[35] R. Hasan, W. Yurcik, and S. Myagmar. The evolution of storage service providers: techniques and challenges to outsourcing storage. In *Proc. of ACM StorageSS*, 2005.

[36] A. Juels and B. S. Kaliski, Jr. PORs: Proofs of retrievability for large files. In *Proc. of ACM CCS*, 2007.

[37] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. ShieldStore: Shielded in-memory key-value storage with SGX. In *Proc. of ACM Eurosys*, 2019.

[38] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *Proc. of USENIX ATC*, 2007.

[39] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. PESOS: Policy enhanced secure object store. In *Proc. of ACM EuroSys*, 2018.

[40] J. Li, P. Lee, C. Tan, C. Qin, and X. Zhang. Information leakage in encrypted deduplication via frequency analysis: Attacks and defenses. *ACM Transactions on Storage*, 16(1):4:1–4:30, 2020.

[41] J. Li, Z. Yang, Y. Ren, P. Lee, and X. Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proc. of ACM Eurosys*, 2020.

[42] M. Li, C. Qin, and P. Lee. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proc. of USENIX ATC*, 2015.

[43] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. of USENIX FAST*, 2013.

[44] J. Liu, N. Asokan, and B. Pinkas. Secure deduplication of encrypted data without additional independent servers. In *Proc. of ACM CCS*, 2015.

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

[46] S. Mofrad, F. Zhang, S. Lu, and W. Shi. A comparison study of Intel SGX and AMD memory encryption technology. In *Proc. of ACM HASP*, 2018.

[47] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. *Journal of the ACM*, 51(2):231–262, 2004.

[48] O. Oleksenko, B. Trach, R. Krahn, A. Martin, C. Fetzer, and M. Silberstein. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proc. of USENIX ATC*, 2018.

[49] OpenSSL. Cryptography and SSL/TLS toolkit. `https://www.openssl.org/`.

[50] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A secure database using SGX. In *Proc. of IEEE S&P*, 2018.

[51] C. Qin, J. Li, and P. Lee. The design and implementation of a rekeying-aware encrypted deduplication storage system. *ACM Transactions on Storage*, 13(1):9:1–9:30, 2017.

[52] M. O. Rabin. Fingerprint by random polynomials. Technical report.

[53] Y. Ren, J. Li, Z. Yang, P. P. C. Lee, and X. Zhang. Accelerating encrypted deduplication via SGX. Technical report, CUHK, 2021. `http://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_sgxdedup.pdf`.

[54] H. Ritzdorf, G. O. Karame, C. Soriente, and S. Čapkun. On information leakage in deduplicated storage systems. In *Proc. of ACM CCSW*, 2016.

[55] P. Shah and W. So. Lamassu: Storage-efficient host-side encryption. In *Proc. of USENIX ATC*, 2015.

[56] Z. Sun, N. Xiao, G. Kuenning, S. Mandal, E. Zadok, P. Shilane, and V. Tarasov. A long term user-centric analysis of deduplication patterns. In *Proc. of IEEE MSST*, 2015.

[57] A. S. Technology. Building a secure system using TrustZone® technology. `https://static.docs.arm.com/genc009492/c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, 2009.

[58] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. In *Proc. of USENIX FAST*, 2009.

[59] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proc. of USENIX FAST*, 2012.

[60] J. Xu, E.-C. Chang, and J. Zhou. Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. In *Proc. of ACM ASIACCS*, 2013.

[61] W. You and B. Chen. Proofs of ownership on encrypted cloud datavia Intel SGX. In *Proc. of ACNS*, 2020.

[62] Y. Zhou, D. Feng, W. Xia, M. Fu, F. Huang, Y. Zhang, and C. Li. SecDep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management. In *Proc. of IEEE MSST*, 2015.

[63] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of USENIX FAST*, 2008.

[64] P. Zuo, Y. Hua, C. Wang, W. Xia, S. Cao, Y. Zhou, and Y. Sun. Mitigating traffic-based side channel attacks in bandwidth-efficient cloud storage. In *Proc. of IEEE IPDPS*, 2018.