Akeso: Bringing Post-Compromise Security to Cloud Storage

Lily Gloudemans
William & Mary
Virginia, USA
algloudemans@wm.edu

Pankaj Niroula
William & Mary
Virginia, USA
pniroula@wm.edu

Aashutosh Poudel
William & Mary
Virginia, USA
apoudel01@wm.edu

Collin MacDonald
William & Mary
Virginia, USA
cmacdonald01@wm.edu

Stephen Herwig
William & Mary
Virginia, USA
smherwig@wm.edu

Abstract

Although cloud providers offer many options for encrypting object storage and rotating the encryption key, the cloud ultimately possesses the key, leaving data vulnerable to insider attacks, legal demands, and storage bugs. Moreover, current key rotation methods do not re-encrypt existing objects, exposing the data indefinitely to adversaries with stolen keys. This paper introduces Akeso, the first cloud storage system to achieve post-compromise security, thus restoring data confidentiality after a key compromise. For efficient key rotation, Akeso adapts the asynchronous group key agreement protocols of messaging applications to storage clients. For scalable object re-encryption, Akeso makes novel use of a cloud-side enclave to coordinate an updatable encryption scheme among untrusted cloud functions. Our evaluations demonstrate that Akeso re-encrypts a 10 G bucket 2.5× faster than a naïve method that fetches and re-encrypts each object, with a monthly expense that is only 15.6–19.3% higher than the current, less secure, provider encryption options.

Keywords

Cloud Storage, Post-Compromise Security, Confidential Computing

1 Introduction

Organizations increasingly store tremendous amounts of data in the cloud using object storage services such as Amazon S3 [108], Google Cloud Storage (GCS) [56], and Azure Blob Storage [93]. These object stores are the dominant cloud persistence paradigm due to their scalability, availability, and pay-as-you-go billing. However, as these stores grow in importance, they are also a prime target for multiple threat actors. For instance, in several data leaks [5, 7–11], adversaries have exploited bugs [70] in the storage stack of the cloud provider to gain access to customer data. At the same time, cloud provider insider threat incidents [104] and disclosures of customer data to law enforcement [1, 3, 4, 16] call into question the data stewardship of otherwise trustworthy providers.

Security & privacy problem. While cloud providers offer various options for encrypting stored data (see §2) these mechanisms ultimately rely on provider-managed, cloud-side encryption, where the storage infrastructure is responsible for generating and managing *data encryption keys (DEKs)*. Although this model simplifies key management for users, it places full control over data confidentiality and privacy in the hands of the provider. As a result, the provider retains persistent access to plaintext data and unilaterally determines how that data may be used or disclosed. For example, terms of service may permit data reuse for purposes such as targeted advertising, third-party analytics, or training proprietary AI models—often without meaningful user consent [54, 66, 120].

In addition to privacy concerns, provider-controlled encryption limits customer agency in key lifecycle management and regulatory compliance. While customers may rotate keys, such rotations only change the *key encryption key (KEK)* used to wrap the DEK in storage; they do not re-encrypt existing data with a new DEK. Consequently, if a DEK is compromised, rotating the KEK offers no protection against retroactive decryption of stored objects.

This is particularly concerning in backup-oriented storage use cases, where adversaries may collect large volumes of encrypted data and later exfiltrate a single compromised key. To address this, cloud encryption mechanisms should support *forward secrecy*—ensuring that a compromised key cannot be used to decrypt historical data—and *post-compromise security (PCS)*—limiting the time window during which a stolen key remains valid. Indeed, several high-profile data breaches (affecting Capital One [79], Dropbox [2], LastPass [51, 76] and Microsoft [6, 95]) involved stolen credentials or compromised keys that enabled prolonged data exfiltration.

PCS is especially important as cloud storage becomes tightly integrated with large production ecosystems, such as smartphone applications [122, 124] and automated processing pipelines. In the case of smartphone data synced to the cloud, physical theft or forensic access to a device can expose authentication material. Without PCS, this can enable indefinite access to a user's cloud data. Likewise, in machine learning (ML) workflows, cloud storage is often used to stage datasets across training and inference phases. A key compromised early in the pipeline (e.g., during training) could otherwise expose data or model behavior from later stages.

Research question. These limitations motivate the need for system designs that support client-controlled encryption models with strong key lifecycle guarantees, while preserving usability and performance at scale. In this paper, we treat two related problems. *First, how can we prevent the cloud from leaking (whether through accident or injunction) plaintext cloud storage? Second, in the event that an*

Proceedings on Privacy Enhancing Technologies 2025(4), 446–464

https://doi.org/10.56553/popets-2025-0139

adversary compromises a storage encryption key, how can we recover the confidentiality and integrity of a customer's data, and thus limit the overall damages?

Insufficiency of alternative approaches. There are two broad approaches to this problem. At one end of the spectrum, storage clients can encrypt the data client-side before uploading it to the cloud. This approach, however, is costly, as the organization would pay at least 4× more in data egress and ingress fees (see Table 5) to download, re-encrypt, and re-upload the entire bucket on each key rotation, compared to re-encrypting the data in the cloud. Additionally, the most straightforward approach to distributing and refreshing a group encryption key relies on centralized management, which introduces scalability challenges and concentrates risk in a single authority.

At the other end of the spectrum, the organization could use cloud provider offerings for trusted execution environments (TEEs), like Intel SGX [91] or AMD SEV [13, 77, 78], to run a storage proxy in the cloud that is shielded from the provider and cloud-side adversary. However, prior studies have shown that TEEs have an appreciable I/O performance tax [86, 97], and thus that using a confidential VM to re-encrypt a bucket could incur up to 30% additional latency. Attempting to improve performance through horizontal scaling may be expensive, as a confidential VM costs 20% more than its normal VM counterpart [55] (see also Table 4). Finally, while a confidential VM enhances security by safeguarding the storage encryption key, clients still need to authenticate to this VM. In other words, such an approach merely shifts the problem from key leakage to the equally challenging problem of credential leakage.

This paper. Our insight is to leverage recent advances in *continuous group key agreement (CGKA)* protocols, along with a limited (and optional) use of a cloud-side TEE, to achieve a middle ground to these two extremes. In this paper, we present the design and implementation of Akeso, a cloud storage system that ensures the confidentiality of the data from the cloud provider, and which admits recovery of this confidentiality in the event of key compromise.[1] Akeso has the following three important properties:

1. Post-compromise security. Akeso adapts the CGKA protocols [24, 42] that underpin end-to-end encrypted group messaging [21] to establish a shared encryption key for a group of storage clients. The clients use the group key for client-side encryption prior to uploading an object to their cloud bucket. These CGKA protocols notably enable efficient and post-compromise secure group key rotation: if an adversary learns the group key, this key is only valid until the next key rotation. Effectively, for an adversary to indefinitely compromise the group, the adversary faces the daunting task of eavesdropping on every key update.

2. Efficient re-encryption. On a key rotation, Akeso re-encrypts the entire bucket. To autoscale this intensive operation to the bucket size, Akeso uses a practical updatable encryption scheme [25], whereby untrusted cloud functions re-encrypt the data without learning the plaintext. This updatable encryption scheme requires a trusted principal to perform an initial, less demanding, setup step for each re-encryption. Akeso supports using either a single cloud-side TEE to perform this step, or a trusted on-premise machine, though our evaluations in §5 show that a TEE is both more performant and cost-effective.

3. Compatibility. In designing Akeso, we place immediate deployability as a first-level concern. To that end, Akeso does not require provider-level changes; rather it is compatible with the current cloud storage services. Additionally, its usage is transparent to the client-side applications.

Contributions. We make the following contributions:

- We present the design of Akeso, the first system (to our knowledge) that achieves post-compromise security for shared, persistent storage.
- We implement an Akeso prototype on the Google Cloud Platform (GCP), and specifically the Google Cloud Storage (GCS) service. Our prototype integrates the Asynchronous Ratcheting Tree [42] CGKA protocol, updatable encryption, and a small enclaved service into an elegant cloud-native system.
- We comprehensively evaluate the performance and costs of Akeso under a set of diverse workloads, and compare these overheads to existing GCS options for object encryption. Our benchmarks show that Akeso has little effect on client I/O latencies, can re-encrypt a bucket in minutes, and costs less than $10 more per month compared to provider options for 1 TB and larger buckets.

Paper organization. We review existing cloud storage encryption options, as well as our threat model and goals, in §2. In §3, we describe the existing techniques that underpin Akeso, and in §4 describe how Akeso adapts and composes these techniques for the cloud storage setting. We present a comprehensive evaluation of Akeso's cost and performance in §5. We discuss configuration choices for Akeso in §6, related work in §7, and conclude in §8.

2 Motivation

2.1 Current Cloud Storage Encryption

We begin by summarizing the current encryption options available to customers for their cloud storage and the limitations in these methods. In Table 1 we summarize the advantages and disadvantages of each approach in comparison to Akeso. For concreteness, we focus on Google Cloud Storage (GCS), but note that Amazon S3 [109] and Azure Blob Storage [94] have similar offerings.

As Figure 1 shows, GCS offers several options for encrypting cloud storage, and all rely on a *key-wrapping* scheme: GCS generates a *data encryption key* (DEK) to encrypt the object's content, and Google encrypts (wraps) the DEK with a *key encryption key* (KEK). GCS stores the wrapped DEK with the corresponding object. GCS encrypts an object when it is stored in a bucket, decrypts the object before it serves it to a client, and automatically rotates the DEK every time the client modifies the object. By default, Google uses AES-256 in Galois Counter Mode (GCM), as implemented by the Tink cryptographic library [61].

For managing KEKs, Google uses its *Cloud Key Management Service* (KMS), which supports software- and hardware-backed KEKs, as well as externally managed KEKs. A KEK never leaves the KMS; thus, when GCS needs to wrap or unwrap a DEK, GCS makes a remote procedure call to the KMS to perform this operation.

[1] Akeso (alternatively, *Aceso*) is the Greek goddess of the healing (recovery) process.

Table 1: Properties of Akeso and Alternative Approaches

Approach	Latency	Scalablity (Performance)	Scalability (Key Management)	Cost ($)	Achieves PCS	Re-encryption Speed
Default (CMEK)	↑	⇑	⇑	⇑	○	↔
TEE Proxy	↓	↓	⇑	↑	⊙	↔
Client-side Encryption	↔	↓	↓	↓	⊙	↓
Akeso	↔	↑	↑	↑	●	⇑

Great (⇑), Good (↑), OK (↔), Bad (↓), Terrible (⇓)
Yes (●), No (○), Depends (⊙)

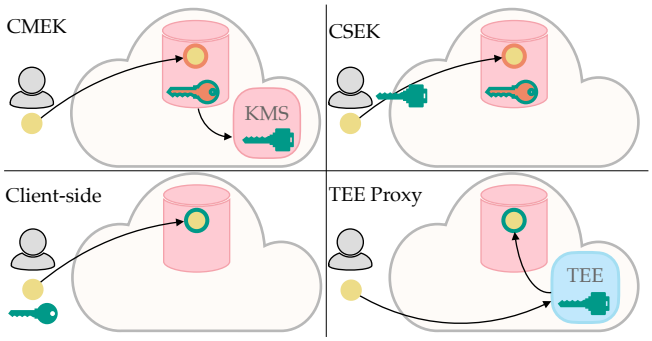

Figure 1: Comparison of GCS encryption options (*CMEK* and *CSEK*) and other alternatives (*TEE Proxy* and *Client-side*). The KEK (teal key) encrypts the DEK (orange key), which in turn encrypts the object (yellow circle). The arrows show the interactions between the client and the cloud, and within the cloud subsystems, when the client downloads the object. Note that the TEE proxy and client-side encryption would also result in GCS applying a layer of encryption (not shown).

Default encryption. By default, Google encrypts (at no charge) all storage object content server-side before it writes the content to disk. For this default encryption, Google generates and manages the corresponding KEKs: Google may use a single KEK to wrap DEKs across objects and across customers. Periodically, the KMS rotates a KEK, and uses the new key version for all subsequent object creations and modifications. The KMS continues to store prior keys so that the customer can access previously existing objects.

Customer-managed encryption keys. To comply with diverse customer policies for KEK management, GCS offers several (non-free) options for *customer-managed encryption keys*, or CMEKs. Here, *managed* refers to the customer's ability to set permissions on keys, monitor key usage, configure different keys for individual objects, and specify the mechanisms for generating and storing keys. These options reflect a spectrum in the balance of key control between Google and the customer, and include:

- **Software-generated keys:** Google generates the KEK.
- **Hardware-generated keys:** Google generates the KEK in a Google-managed and -owned hardware security module, referred to as *Cloud HSM*, and the KMS forwards key operations to the HSM. Google uses Marvell LiquidSecurity HSMs, which comply with FIPS 140-2 Level 3-validation [57].
- **Imported keys:** The customer generates their KEK and imports it into the KMS (or Cloud HSM). This option satisfies *Bring-Your-Own-Key* policies, such as where a regulatory requirement stipulates that key generation occur in a specific environment.
- **Externally managed keys:** The customer uses a compatible key manager that is external to Google Cloud to create and control their KEKs, and configures the KMS to forward key operations to this external key manager. This option satisfies *Hold-Your-Own-Key* policies, such as requirements that restrict key custody.

Customer-supplied encryption keys. With a *customer-supplied encryption key* (CSEK), the customer provides the KEK as an input argument to each storage operation, and GCS uses the KEK to wrap the GCS-generated DEKs. Google does not permanently store a CSEK, but rather purges this key from its system after each operation. To ensure key consistency, GCS stores a hash of the CSEK with the object; as such, a CSEK associates with an individual object, rather than an entire bucket.

Limitations. Despite the array of options, in all approaches the cloud provider generates the DEK, which allows them to decrypt the data. Even in the case of a trustworthy provider, the DEK is still (for some time) accessible on the cloud storage system, thereby exposing it to an infiltrator. Although the CMEK options allow a customer to set a key rotation schedule, this simply changes the KEK that GCS uses for newly created objects; it does not re-encrypt the contents of existing objects with a new DEK. Thus, an adversary with access to the DEK can decrypt the object until it is modified, at which point GCS changes the object's DEK. Unfortunately, as we show in Appendix C, customers heavily use cloud storage for read-only data, such as media files; for such objects, the DEK does not change.

2.2 Alternative Approaches

Client-side encryption. Customers may also choose to encrypt their data client-side using a cipher suite of their choice before uploading it to GCS. In this setup, the customer is fully responsible for key management, including storage and rotation. Since GCS is

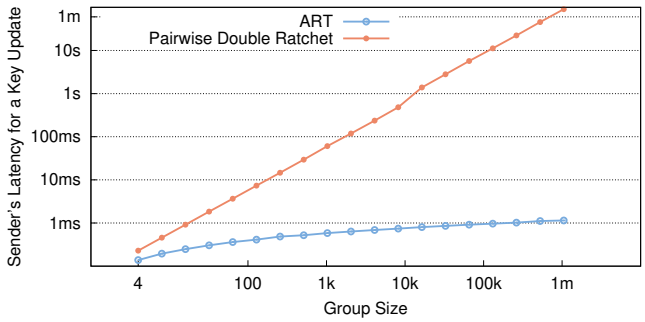

Figure 2: The sender's workload (here, measured as latency) for a key update operation using ART and using a pairwise Double Ratchet key transport. Both axes are log-scale. ART scales as $O(\log(N))$, whereas pairwise Double Ratchet scales $O(N)$ (where N is the group size).

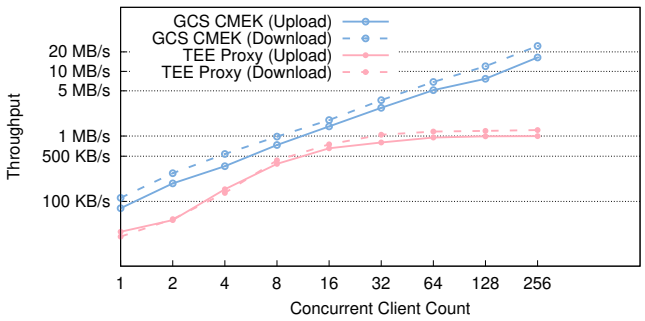

Figure 3: The throughput of GCS CMEK and our TEE Proxy for file uploads and downloads. For both GCS CMEK and the TEE Proxy, we use their respective REST APIs.

unaware that the data is already encrypted, it applies its default server-side encryption, resulting in double encryption. As noted in §1, this approach can lead to significantly higher data egress fees compared to performing re-encryption in the cloud.

This design also assumes that clients share a group key. In the simplest case, each client interacts with a trusted client-side key manager to obtain the new key upon rotation. However, traditional key transport—where the key manager encrypts the key to each client's public key—does not provide PCS guarantees, as an attacker who steals a client's secret identity key can decrypt future key transport messages. To achieve PCS, the key transport must use a protocol like the *Double Ratchet* algorithm [99].

The main drawback of a pairwise Double Ratchet approach is that the sender's workload (in this case, the key manager's) scales linearly with the number of clients, whereas a tree-based key agreement protocol achieves logarithmic scaling. To experimentally confirm this, we develop Go-based versions of the Double Ratchet algorithm and the Asynchronous Ratcheting Tree (ART) key agreement protocol (see §4 for details), and use Go's benchmark support to measure the sender's latency for a key update as the group size increases (to measure the key manager's work, the key manager serially processes each member's key transport message). Figure 2 confirms the computational complexity of the two approaches, and that pairwise Double Ratchet is an impractical key management strategy for large groups.

TEE encryption proxy. As we mentioned in §1, an alternative approach to ensure key confidentiality and support key rotation is to deploy an encryption proxy within a TEE in the cloud (see §3.3 for background on TEEs). In this design, the clients send queries to the proxy over attested and mutually-authenticated TLS. The proxy, which is the only component with access to the encryption key, forwards requests to GCS and handles encryption or decryption as needed. For re-encryption, the TEE proxy generates a new encryption key, and retrieves and re-encrypts each object.

While prior studies [86, 97] noted appreciable I/O overheads when using TEEs (upwards of 30% additional latency), we conduct our own I/O performance measurements to specifically compare the latency and scalability of the TEE-proxy approach to GCS's CMEK encryption option. To do this, we implement a simple TEE proxy as an HTTPS gateway to a storage bucket. The gateway supports two operations: `PUT` (write an entire object) and `GET` (read an entire object). We wrote the TEE proxy in Go and create a separate goroutine for each client request. Figure 3 compares the throughput performance of the TEE proxy and the CMEK option as the number of clients increases, showing that the TEE proxy is both slower and less scalable.

2.3 Threat Model

Our setting consists of three principals: *(1)* the *storage clients*, who share access to an encrypted cloud store; *(2)* the *cloud provider*, who operates the cloud services, and *(3)* an *adversary*, whose goal is to learn the plaintext of the cloud storage objects. We now detail our trust assumptions and threat model for each of these principals.

Storage clients. We assume that the storage clients mutually trust one another (for instance, they belong to the same company): they follow the protocol and do not intentionally leak the plaintext or the key material. A client may be either cloud-side or on-premise. If the client is hosted in the cloud (for instance, the client is a microservice), we assume that the client runs in a TEE so as to maintain the confidentiality of the storage data it accesses.

TEEs. Akeso uses TEEs in two capacities: to shield cloud-side clients, and to host a special storage service called Akesod (see §4). We assume that the cloud-side clients may contain vulnerabilities, thus allowing an adversary to extract their keys and credentials despite having the added protection of a TEE. In contrast, we assume that the Akesod software is vetted and bug-free, and thus immune to exploitation or accidental key leakage. Note that for clarity and consistency, the remainder of this paper assumes that clients are hosted outside the cloud. We assume that side-channel attacks [30, 37, 50, 106, 107, 117, 119] are out-of-scope; we trust the CPU designers to patch such vulnerabilities.

Cloud provider. We assume the cloud provider is *semi-trusted*. Concretely, the provider faithfully follows the protocols and service-level agreements, and does not persist old copies of a storage object. Specifically, if a client updates a storage object, the provider overwrites or otherwise removes the old version (we discuss relaxing

this assumption in §6). The provider may comply with law enforcement demands to disclose customer data (including a customer's storage), and may be susceptible to insider threats.

Adversary. We assume a non-persistent adversary with temporary access to either the cloud provider's storage stack or the client. This adversary can eavesdrop on or intercept client-provider communications as a man-in-the-middle. The adversary can fully compromise a client—whether inside a TEE or not—allowing them to extract secrets, key material, and long-term cloud credentials. With these, they can impersonate the client, interact with cloud services on their behalf, or monitor their communications.

In summary, we assume that encryption key compromises are isolated incidents rather than ongoing events. However, the adversary's access to (encrypted) cloud data—whether through stolen cloud credentials or insider privileges—may be permanent.

2.4 Goals

In designing Akeso, we have three primary goals:

G1: Post-compromise security. Informally, post-compromise security (PCS) means that if an adversary temporarily obtains the system's secret keys, the adversary will lose access after a key refresh (assuming the refresh process is not compromised), allowing the system to regain confidentiality. Cohn-Gordon et al. [41] formally define PCS within the setting of authenticated key exchange, and Lehmann and Tackmann [85] provide formalisms within the setting of updatable encryption. We state the PCS property in the cloud storage setting using a simplified notation that closely follows that of the TreeKEM paper [24].

We assume that the global state of the storage system S consists of the following elements:

- $G = \{m_0, m_1, \dots\}$: the storage group G, consisting of members (storage clients) m_i.
- $O = \{o_0, o_1, \dots\}$: the storage bucket O consisting of storage objects o_i. Each storage object o_i has a name, contents, and user-specified metadata, called *attributes*.
- $\mathbf{public}(G \mid m_i)$: when applied to the group G, the public data of the group; when applied to a member m_i, the public data of that member.
- $\mathbf{secret}(G \mid m_i)$: when applied to the group G, the secret data shared by all group members (e.g., a group key); when applied to a member m_i, the secrets specific to that member (e.g., the member's private key(s)).

The global state evolves over time as sequence of epochs:

$$S_0 \rightarrow S_1 \rightarrow \dots$$

where a state transitions from S_k to S_{k+1} via the following operation:

> $\mathbf{Update}_k(m_i)$: member m_i refreshes its local state and generates a fresh group key, hence updating $\text{secret}_{k+1}(G)$ and $\text{secret}_{k+1}(m_i)$.

Note that, for clarity, we may suffix an operation with the epoch in which it occurs. Within a state S_k, any member may invoke one of the following cloud storage I/O operations:

- $\mathbf{Cloud.Put}_k(\mathbf{name}, \mathbf{contents}, [\mathbf{attr}])$: Put (write) an object. If the object name does not already exist, the system creates the object; otherwise, the system overwrites the existing object. If the client provides the optional attr argument, then this call also updates the object's attributes.
- $\mathbf{Cloud.PutAttr}_k(\mathbf{name}, \mathbf{attrs})$: Update an object's attributes.
- $\mathbf{Cloud.Get}_k(\mathbf{name})$: Get an object and its attributes.
- $\mathbf{Cloud.GetAttr}_k(\mathbf{name})$: Get the attributes for an object.
- $\mathbf{Cloud.Pub}_k(\mathbf{msg})$: Publish message msg to the system.

Additionally, within a state S_k, an adversary $\mathcal{A}$ can call the following operation to compromise a member:

> $\mathbf{Compromise}_k(m_i)$: the adversary compromises member m_i, thereby learning $\text{secret}_k(G)$ and $\text{secret}_k(m_i)$.

Using this notation, we define:

Definition 1 (*Post-Compromise Security*)**.** If an adversary $\mathcal{A}$ executes $\text{Compromise}_k(m_i)$, and in some subsequent state S_{k+x} either *(1)* m_i executes $\text{Update}_{k+x}(m_i)$ or *(2)* some other member m_j executes $\text{Update}_{k+x}(m_j)$ for which $\mathcal{A}$ does not learn $\text{secret}_{k+x+1}(G)$, then starting in state S_{k+x+1} the objects O are again confidential.

Informally, if $\mathcal{A}$ compromises m_i in epoch k, then the system regains the confidentiality of its storage when either m_i refreshes the group key (thus preventing $\mathcal{A}$ from learning m_i's contribution to the group key) or some m_j initiates a group key update that $\mathcal{A}$ does not observe (eavesdrop upon), thus preventing $\mathcal{A}$ from deriving the new group key, and hence all subsequent keys (until a next compromise).

Note that we seek only to provide (and regain) the confidentiality and integrity of the storage object's content. In particular, we do not aim to conceal object metadata or access patterns from an adversary. Metadata-hiding storage [36, 38, 39, 112] is an important and active area of research, but is complementary to Akeso.

G2: Performance and cost. For practicality, the system's performance (such as upload and download latency) and costs (as for cloud services and data ingress and egress) should be competitive with the cloud provider's current cloud storage encryption offerings.

G3: Compatibility and transparency. To promote adoption, we assume that we cannot modify the cloud provider's hardware or software. Thus, our system may only modify the client-side software, and utilize the available cloud services. Moreover, our client-side modifications must also be compatible and transparent: they should extend the conventional storage client software, and not impose alterations on any higher-level applications.

3 Building Blocks

Akeso's high-level contribution is composing multiple existing cryptographic techniques with TEEs in a setting distinct from their original target use case. In this section, we briefly review the underlying cryptographic and TEE concepts that Akeso uses, and in §4 we detail our approach to integrating these components for the shared cloud storage setting.

3.1 Continuous Group Key Agreement

Group key establishment is a critical operation for Akeso, as the clients encrypt data client-slide and periodically rotate the joint encryption key. *Continuous group key agreement (CGKA)* protocols are natural extensions of two-party continuous key agreement

protocols—such as Double Ratchet [99]—that gained popularity in messaging applications like Signal [40], to the group setting. A CGKA protocol generates a sequence of symmetric group keys—derivable by each member of the group—, with each new key marking the start of a new epoch. From a security perspective, CGKA protocols aim to provide *forward secrecy* and *post-compromise security*, limiting the impact of a key compromise to the affected epoch. Important functional properties include whether the protocol is asynchronous (allowing clients to update their cryptographic state independently of others, which may be offline), and decentralized, (operating without a trusted central authority). Many CGKA protocols also support dynamic group membership, enabling members to be added or removed over time.

Asynchronous ratcheting tree. If a member must store information about all members of the group, and if each group modification operation involves a distinct message for each member, the storage, bandwidth, and computational requirements for every operation is linear in the group size, which is impractical for large groups (see Figure 2). As a result, many CGKA protocols (and by extension, Akeso) organize their members into a tree-based data structure [31, 32, 46, 80, 81, 98, 111]. The first proposal for a CGKA with PCS guarantees was the *Asynchronous ratcheting tree (ART)*, which implements all operations using $O(\log n)$ storage, bandwidth, and complexity, both at the sender and the receiver. As Figure 4 shows, each tree node in ART is a Diffie-Hellman (DH) key pair [47]; all non-leaf nodes contain two children, and each leaf corresponds to the key pair for a specific group member. Starting from the leaves, the private DH key of each parent node is the result of a DH computation between its two children; this process continues recursively until yielding the *root key*, the secret half of which is input to a key derivation function (KDF) to form the symmetric *group key*. The core property of this data structure is that, to compute the *root key*, a member must know one secret leaf key, as well as all public keys on the leaf's *copath* (the list of sibling nodes along the leaf's path to the tree root). To rotate the group key, a group member m_i generates a new DH key pair for their leaf, computes the new tree, and broadcasts a `KeyUpdate` message containing the tree's new public keys; each member then recomputes their local tree.

TreeKEM. TreeKEM [24], which forms the basis of the IETF Messaging Layer Security (MLS) standard [21], builds on the structure of ART by using a similar tree-based design but a different cryptographic construction that reduces computational overhead for recipients. While ART and TreeKEM share the same group structure and local state, TreeKEM moves beyond Diffie-Hellman and allows each tree node to use any keypair that supports key encapsulation (KEM). Additionally, TreeKEM improves usability by supporting dynamic group membership and enabling more flexible consistency through mergeable operations.

3.2 Updatable Encryption

A simple approach to re-encrypt all storage during key rotation involves downloading the storage objects, re-encrypting them, and uploading the new ciphertext back to the cloud. However, this method faces two major challenges: it does not scale with the bucket size, and incurs significant data egress and ingress costs

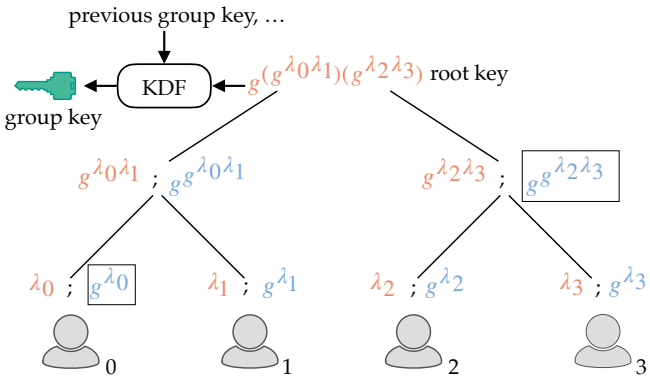

Figure 4: Asynchronous Ratcheting Tree. The red keys (left key in each pair) are private; the blue are public. The two boxed public keys represent the copath for group member #1. The private root key is an input to a key derivation function (KDF) that outputs the symmetric group key.

if the re-encryption occurs on-premise, outside the cloud. To overcome these issues, Akeso must both autoscale object re-encryption, and perform this process within an untrusted cloud environment. Akeso solves this by using *updatable encryption*, a symmetric-key encryption scheme that allows key rotation by an untrusted third party, such as the cloud. In such schemes, the client sends a small *update token* to the cloud, which uses the token to rotate the encrypted storage from the old key to the new one without learning the plaintext.

Previous research has introduced various updatable encryption schemes, often focusing on theoretical aspects like ciphertext unlinkability after re-encryption [35], unidirectionality [92, 96], resilience to token corruption [45], and metadata leakage [48]. Akeso's design directly addresses some of these properties, such as update token integrity, while others (e.g., unlinkability, unidirectionality, metadata leakage) fall outside its use case and threat model. Additionally, updatable encryption schemes can be categorized into ciphertext-independent [27, 49] and ciphertext-dependent schemes [29, 82, 85], where the latter requires the client to download a small portion of the ciphertext, known as the *ciphertext header*, to generate the update token. Despite this minor overhead, ciphertext-dependent schemes are more efficient, and Akeso employs such a construction.

3.3 Confidential Computing

Although CGKA and updatable encryption provide robust key management and encryption in adversarial settings, Akeso faces the challenge of protecting two remaining security sensitive operations: managing group membership, and generating the re-encryption tokens. As we describe in §4, Akeso uses a small orchestrating process called Akesod for these operations, and requires that Akesod run in a trusted environment. While Akeso's design provides flexibility in hosting Akesod either on-premise or in the cloud, for the cloud hosting, the customer must deploy Akesod using the provider's offerings for confidential computing.

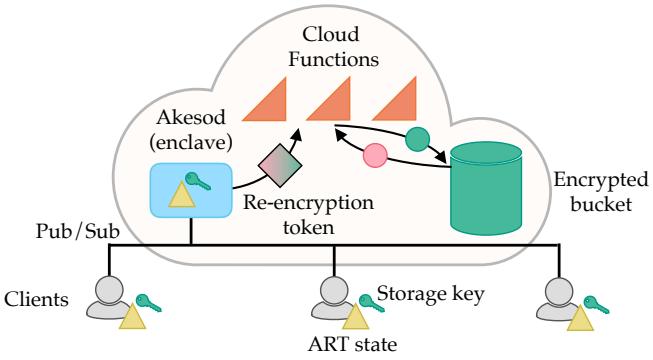

Figure 5: High-level architecture of Akeso. The clients and Akesod trust one another, and we assume that an adversary cannot breach Akesod.

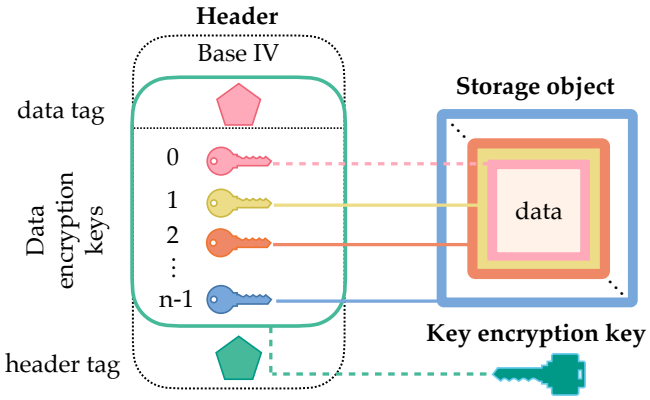

Figure 6: Updatable encryption using nested AES. The key encryption key (KEK) as well as an object's first data encryption key (DEK) are sensitive. Subsequent DEKs in the header serve as update tokens, each of which is the AES key for a layer of encryption. Dashed lines from a key denote encryption with AES-GCM, and solid lines denote AES-CTR.

Confidential computing is executing a workload on an untrusted third-party machine (the cloud) in a way that "shields" the workload from the third-party: the third-party cannot alter, observe, or interfere with the computation or its data (though, in practice, side channels [28, 33, 87–89, 117, 118] weaken these guarantees). Confidential computing relies on hardware trusted execution environments (TEEs) to ensure memory isolation from the rest of the system: an application's memory is encrypted and integrity-protected when in DRAM, and decrypted only when the memory enters the trusted CPU package. When a workload is running within a TEE, we say that the workload is running within an *enclave*; as shorthand, we often simply call the workload an enclave.

Process- and VM-based enclaves. Intel SGX [74, 91] was the first general approach to confidential computing. By representing enclaves at the granularity of a part of a process, Intel SGX promotes a small trusted computing base, but at the expense of incompatibility with legacy software, though several middleware solutions [18, 22, 71, 115] attempt to ease this burden. To allow for unmodified applications to run within an enclave, the current generation of TEEs, such as AMD SEV [13, 77, 78], Intel TDX [75], Arm CCA [90], and IBM's PEF [73], implements enclaves at the granularity of a virtual machine (called *confidential VMs*). Due to their ease of use and broad provider support, our implementation uses AMD SEV enclaves, though the Akeso design supports using either a process- or VM-based enclave for Akesod.

Attestation. A core property of all enclaves is attestation [14]: a hardware root-of-trust signs a digest of the initial launch state of the enclave, forming an *attestation report*; the customer can retrieve this report to verify the correct software is running in an enclave. To allow an attestation to bind some runtime value, the enclave can include a small amount of *user-data* in the report; the user-data is opaque to the secure hardware, but otherwise covered by the report's signature.

4 Design & Implementation

In this section, we present a prose-style description of the design and implementation of our Akeso proof-of-concept. Appendix B sketches a security proof that Akeso fulfills Definition 1 (PCS).

We implement Akeso using Google Cloud Platform (GCP). In translating the underlying techniques of §3 to the cloud storage setting, we confront four challenges:

- **C1:** Scaling updatable encryption for practical post-compromise security
- **C2:** Adapting CGKA from transient messaging to persistent storage (specifically, ensuring PCS applies to both new and existing data)
- **C3:** Maintaining backward-compatibility with the existing cloud infrastructure and client software
- **C4:** Ensuring the consistency of the shared group data

Figure 5 depicts a high-level overview of the Akeso architecture. For a cloud-native solution, Akeso organizes the clients and orchestrating enclave (Akesod) into a CGKA group, and uses GCP's Pub/Sub channels for broadcasting the CGKA messages. Additionally, on a key update, Akesod triggers GCP to launch untrusted cloud functions, which receive the corresponding update token and re-encrypt the objects in the bucket. Our Akeso prototype uses ART as its CGKA protocol so as to focus on the core CGKA operation of key update; we leave the other pragmatic features of group management to future work. We now describe in detail how Akeso addresses each of the four challenges.

4.1 Scaling Updatable Encryption

Akeso implements a practical version of ciphertext-dependent updatable encryption based on a performant, nested-AES scheme from Boneh et al. [25]. In this scheme, the ciphertext header is a list of DEKs, and the ART group key serves as a KEK that encrypts this list. On each key rotation, Akesod generates a single new DEK for the entire bucket; Akesod decrypts each object's ciphertext header with the old group key, appends this new DEK to the list, and re-encrypts the list with the new group key. The new DEK serves as the re-encryption token: Akesod triggers the cloud functions to

use this DEK to apply another layer of encryption to each object's contents.

Intuitively, if an adversary learns the group key (KEK) at epoch k, but is unable to eavesdrop on the cloud messages for some later key update at epoch $k + x$, the adversary is unable to derive the KEK for epoch $k + x + 1$, and thus cannot decrypt an object's ciphertext header to reveal the list of DEKs. Appendix B crystallizes this intuition by providing a proof sketch of PCS that performs a case analysis with respect to an object's life cycle, and further shows that knowledge of the re-encryption token still preserves the PCS property.

We next describe in further detail how we integrate ART with nested AES in an object's life cycle. (Appendix A lists pseudocode for these major operations.)

Object creation. When a client creates an object, the client randomly generates an AES-256 DEK and encrypts the object content using AES-GCM (for authenticated encryption with additional data). This initial DEK is secret and unique to the object (in contrast, the subsequent DEKs for re-encryption are update tokens that Akesod shares with the cloud and applies to all objects). For AES-GCM's additional data, the client uses the object's name, which defends against object-swapping attacks where an adversary renames object A to B and vice-versa. The client constructs the ciphertext header by generating a random 16-byte IV (the *Base IV*), and using AES-GCM to encrypt the DEK and the data's authentication tag (the *data tag*). The client uses the first 12-bytes of *Base IV* as the nonce to the AES-GCM encryption of the DEK and data tag.

GCS supports attaching 8 K of arbitrary metadata (in the form of key-value pairs) to each cloud object [60]. The client therefore sets the ciphertext header as the value for metadata key `akeso_header`, and uploads the new object with its metadata to cloud storage.

Object re-encryption. A client initiates a group key update by broadcasting a `KeyUpdate` message to the group's Pub/Sub topic. Upon pulling this message from its topic subscription, Akesod saves the old group key, updates its local ART state to derive the new group key, and generates a random DEK. Akesod then creates a notification to cloud storage that includes this DEK, and which triggers GCP to launch a cloud function when an object's metadata changes. This notification uses a distinct channel from the ART Pub/Sub channel; a client's cloud credential does not permit access to this channel.

Next, Akesod fetches the metadata for each object in the bucket and extracts its `akeso_header` value from the metadata. Using the old group key, Akesod decrypts the encrypted portion of the header and appends the new DEK. Akesod encrypts the list of DEKs (along with the data tag) using AES-GCM with an incremented nonce value (an incremented *Base IV*).[2] Figure 6 shows the complete structure of a ciphertext header after $n - 1$ key rotations.

After updating the header, Akesod makes a GCS request to update the object's metadata, which subsequently launches the cloud function. Each cloud function instance receives as input the DEK and a specific object name. The function instance uses the DEK to apply another layer of encryption to the object's content. As the header's data tag captures the integrity of the content, the function instance uses the simpler (unauthenticated) AES-CTR mode of encryption. Similar to the new encryption of the header, the cloud function uses an increment of the full 16-byte *Base IV* for the IV.[3]

Maximum encryption layers. As Boneh et al. note [25], a drawback of using nested AES for updatable encryption is that the time for a client to encrypt and decrypt an object grows linearly with the number of re-encryptions. In practice, Akesod easily counters this concern: to prevent an unbounded overhead, Akesod enforces a (configurable) maximum number of layers of encryption. When fetching object A's metadata on a key update, if Akesod determines that the new DEK would exceed A's maximum number of layers, Akesod downloads the entire object, and resets A's DEK list to a size of one. Since an object's initial DEK is secret, Akesod must randomly generate and encrypt A with a fresh initial DEK, rather than use as the initial DEK the DEK that it generated as an update token for the other objects. Section §4.3 also describes a simple client optimization to reduce growth in the number of layers of encryption.

4.2 Adapting ART to Storage

The need for object re-encryption is the fundamental distinction between post-compromise security in the persistent storage setting and post-compromise security for transient messaging. The remaining challenge is the trustworthy construction of the initial ART group (see Appendix D). As we trust Akesod to initiate object re-encryption, it is natural to also designate Akesod as the ART initiator, which must be trusted to create the ART group.

We assume that the clients trust the Akesod software (as through a software vetting process) and possess the hardware measurement that reflects a faithful launch of the software. A key technical hurdle is to bind Akesod's attestation to the initial `GroupSetup` message, thereby providing clients with assurance about the integrity of ART's initial state. Our approach, which is similar to efforts [17, 53, 83, 116] to bind an attestation with the TLS handshake [102], is for Akesod to generate its long-term key pair in the enclave, and include a hash of the public key as *user-data* in the attestation report. Akesod then includes this report within the `GroupSetup` message. Since Akesod signs the message with its long-term identity key, the other members have assurance that Akesod is running within an enclave.

4.3 Preserving Software Compatibility

For the storage clients, we modify Google's open source Cloud Storage FUSE (Filesystem in Userspace) adapter [58, 62]—`gcsfuse`—which allows users to mount a storage bucket as a local filesystem. Specifically, we integrate the ART protocol into `gcsfuse`, and amend `gcsfuse` to use Google's Pub/Sub [64] for key update notifications. We extend `gcsfuse`'s caching layer to invoke the nested decryption operations when fetching an object from cloud storage, and to re-apply these layers when uploading a modified object. In all, we add ~1200 lines of Go code to `gcsfuse` to implement an

[2] Using a different nonce value is not strictly necessary since the KEK—the header encryption key—has changed; we do this as a simple means to increase robustness to brute force attacks.

[3] The IV is: base IV + len(DEKs) − 1

Akeso client (see Table 7 in Appendix E for the total lines of code in all Akeso components).

By default, with file caching enabled [59], on a partial, random read operation, `gcsfuse` does not always fetch the entire backing object from the cloud bucket to the local cache. Similarly, on a write operation which only appends to a file and does not overwrite any of its current data, `gcsfuse` does not always re-upload the entire file contents. Due to Akeso's need for authenticated encryption, we override these optimizations and ensure that upon opening a file, the `gcsfuse` downloads and decrypts the entire object into the local file cache. Similarly, upon closing (or explicitly syncing a file), we modify `gcsfuse` to encrypt and upload the entire file to the cloud object.

Since any write already requires downloading and re-uploading the entire object, we treat all writes as new object creations. Specifically, when a client modifies a file, gcsfuse generates a new random DEK and uploads the file with a single layer of encryption, regardless of any prior nesting. This approach reduces decryption overhead for future reads in write-heavy applications.

4.4 Maintaining Object Consistency

Concurrent object accesses and key updates open the possibility for an object to become inconsistent with respect to its metadata. We detail the two cases where an access and key update operation may conflict, and how Akeso handles these situations.

Case 1 (reads). Suppose the `gcsfuse` client processes an `open` system call to fetch an object during a key update. The challenge is that there is a race condition between three events: when Akesod processes the key update, when the client processes the update, and when the client fetches the object. To enforce an ordering on these operations, Akesod adds two additional metadata values along with the `akeso_header`: `akeso_kek_hash` and `akeso_reencrypt`. The former is simply a hash of the new ART group key. When a `gcsfuse` client fetches an object, it compares the hash of its group key to this metadata value; there are two possibilities:

The hashes differ. This implies that either `gcsfuse` updated its key and fetched the object before Akesod processed the key update, or that Akesod processed the key update but the client has not. In either case, `gcsfuse` returns an error for the system call, indicating that the application should retry.

The hashes are equal. This implies that Akesod and the client are in sync with respect to the header; the only question is whether the object has the number of layers of encryption as the header implies, or if the object is awaiting a cloud function to add the last layer. To detect this scenario, we use a simple locking scheme: Akesod sets `akeso_reencrypt` to `true`; upon re-encrypting the object, the cloud function sets this to `false`. If a `gcsfuse` client fetches an object that is awaiting re-encryption, it returns an error for the system call, indicating that the application should retry.

We note that our implementation chooses simplicity by placing the onus of retries on the application. An alternative approach could modify `gcsfuse` to use a backoff-and-retry strategy (and only return an error to the application after a set timeout), or briefly maintain and try the old group key.

Case 2 (writes). This case deals with conflicts in writes, as when a client downloads an object and modifies it during a key update. Akeso handles this situation using GCS's automatic metadata for an object's *generation* and *metageneration* value. An object's generation is a unique number that changes (but does not necessarily increment) when the object's content is modified—it logically identifies an object version. In contrast, the metageneration value resets to 1 for each generation, and increments for each modification to an object's metadata.

GCS enables requests to include preconditions for these values to prevent object corruption during read-modify-write updates. We modify `gcsfuse` to use a precondition when syncing an object to the cloud storage to enforce that the object has not been modified since the client opened it. If the object has been modified, the precondition fails, and `gcsfuse` returns an error for the system call (either a `close` or `fsync`) that triggered an attempt to write the object to the cloud. We leave optimizations to this behavior, in which `gcsfuse` attempts to resolve the conflict, to future work.

5 Evaluation

In this section, we compare the performance of Akeso to the options that Google Cloud Platform currently provides for encrypting cloud storage, as well as to simpler client-side encryption schemes. We aim to demonstrate that Akeso has comparable performance to existing alternatives of weaker security properties, and better performance than naïve approaches that provide the same security properties. We first review these specific alternatives, and describe for each *(1)* any changes we made to `gcsfuse` to use this alternative, and *(2)* how we effect a key rotation.

CMEK. With CMEK, a bucket uses a software-generated key with the GCS customer-managed encryption key feature, as described in §2.1. To re-encrypt a bucket encrypted with CMEK, Akesod first generates a new KEK in the Cloud KMS. Akesod then reads each object and updates its metadata to specify this KEK. When GCS observes the update, it re-encrypts the object with a new DEK, and wraps this DEK with the new KEK.

CMEK-HSM. CMEK-HSM uses a hardware-generated key with the GCS customer-managed encryption key feature, as described in §2.1. Re-encryption in CMEK-HSM works similar to CMEK, the difference being the key is managed by an HSM.

CSEK. CSEK uses the GCS customer-supplied encryption key feature, as described in §2.1. We modify `gcsfuse` so that the KEK it supplies to GCS for all storage operations is the ART group key. On a key rotation, Akesod uses a GCS API to migrate the object's KEK from the old group key to the new one; the DEK itself does not change.

Akeso-keywrap. In this option, each object is encrypted using a simple client-side encryption scheme similar to CSEK, but one which does not share the DEK or KEK with the cloud. Concretely, to create an object, a client first randomly generates a DEK and nonce, and encrypts the object's data with the DEK using AES-GCM before uploading it to GCS, similar to Akeso. The ART group key serves as a KEK, and the client uses this key with AES-GCM to encrypt the DEK, the random nonce, and the data's GCM tag. Our `gcsfuse` client adds this value to the object's metadata. On a key rotation,

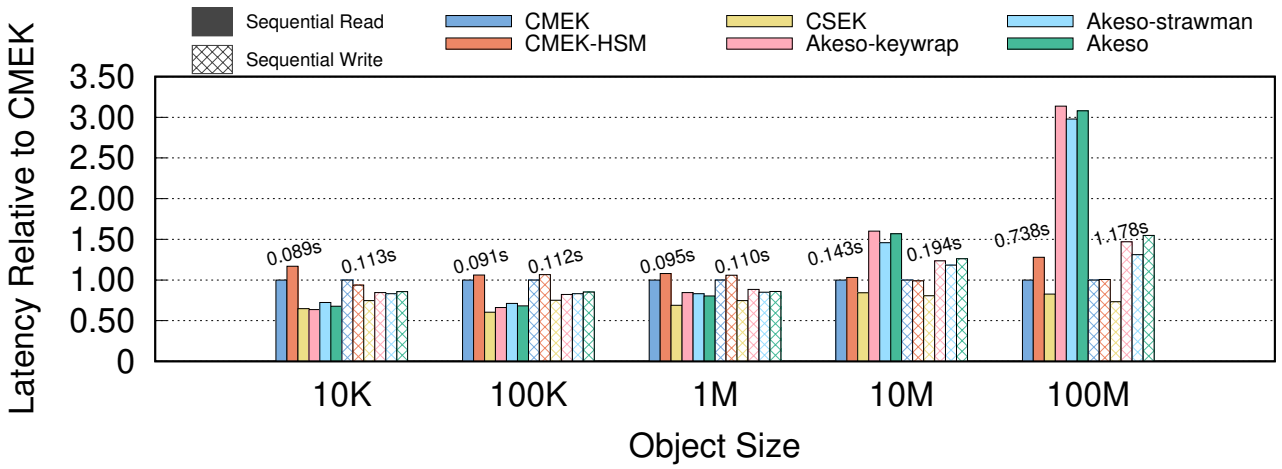

Figure 7: Latency to read and write an entire object using encrypted cloud storage, relative to the CMEK option. The numbers above the bars are the absolute latencies for the CMEK cases.

Akesod downloads the metadata of each object, re-encrypts the header with the new group key, and uploads the header back to the object's metadata. As in CSEK, the object's DEK and ciphertext do not change.

Akeso-strawman. Akeso-strawman provides the same security guarantees as Akeso, but uses a more naïve approach. In this option, each object is encrypted using AES-GCM, with the key being the ART group key. We modify `gcsfuse` to use this key when reading and writing objects. On a key rotation, Akesod downloads the entire object data, decrypts it using the old key, encrypts it using the new key, and re-uploads it.

5.1 Experimental Setup

Unless otherwise noted, we run Akesod in a cloud enclave, specifically GCP's N2D compute node (*base variant - N2D-standard-2*)—an AMD SEV confidential VM. We configure Google Cloud Functions with 512 M memory. The Google Pub/Sub Service handles the asynchronous aspect of Akeso for sending and receiving the `GroupSetup` and `KeyUpdate` messages for the storage group, as well as the `MetadataUpdate` messages for the cloud functions. We run all cloud services in the `us-east1` region. Our clients run in the cloud (N2D VMs, but without AMD-SEV) in the same region. [4]

5.2 I/O Performance

We first evaluate Akeso's filesystem I/O performance; that is, how each encryption option performs from the perspective of an application reading and writing the bucket objects through a `gcsfuse` mount, as if they were normal files in a filesystem. To assess this, we use the official benchmarks in the `gcsfuse` GitHub repo [62] to measure two common operations: sequentially reading an entire file from the cloud, and sequentially writing an entire file to the cloud. We modify the benchmarking tools to additionally include the calls to `open` and `close` in the total elapsed time.

Figure 7 shows the 10% trimmed mean of running each workload 50 times under each encryption option, varying the object sizes from 10 K up to 100 M. Although the distribution of object sizes differs slightly by provider (see Figure 13 in Appendix C), we note that across GCP, AWS, and Azure: 10 K—100 K bounds the median object size of all providers and captures a third of all objects; 1 M–2 M tightly bounds the 90th percentile; 10 M is roughly the 95th percentile; 100 M is the 99th percentile. For Akeso, we apply only a single layer of encryption (we analyze the impact of increasing the layers later in this section). The results show that the performance of Akeso is similar to the existing encryption options despite providing stronger security guarantees. This is not unexpected, as each option performs a similar AES-GCM operation of the file contents. The read latencies consistently show a variability of 3–11% relative to their mean. In contrast, the write latencies show a wider variability of 5–50% relative to their mean. We attribute this to the default write path in `gcsfuse`, where files are written locally as temporary files before being written out to cloud storage [63]

To demonstrate the variation in measurements, Figure 8 shows a CDF of Akeso's read and write latency for a 10 M object. Sequential read operations with Akeso are typically faster. In contrast, sequential writes show more variability and higher tail latencies.

Overhead of nested-AES layers. A property of nested-AES updatable encryption is that the work to decrypt the data increases with the number of re-encryptions (that is, the number of layers of encryption). To evaluate the rate of this increase, we use Go's benchmarking apparatus to develop benchmarks for the nested-decrypt function in our Go package. We run this benchmark on the same client cloud VM, varying the data size and the number of layers of encryption. Table 2 shows the times and associated overheads compared to the 1-layer baseline. Note these times only include the cryptographic operations, and do not include the I/O overhead of retrieving the object. While exact timings vary by machine (our

[4]We initially used local machines for the clients, but switched to cloud machines to promote reproducibility. Regardless, our evaluations showed identical performance and network latencies across all strategies in both setups.

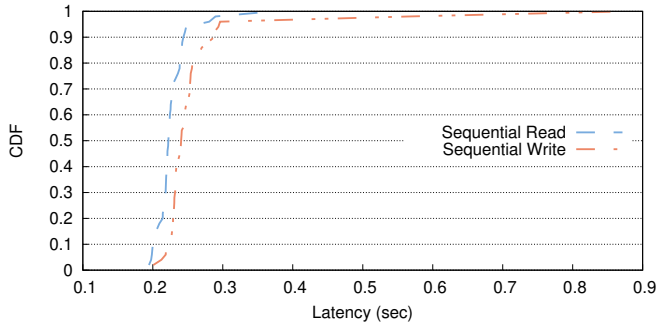

Figure 8: CDF of latencies to read and write a 10 M object with Akeso.

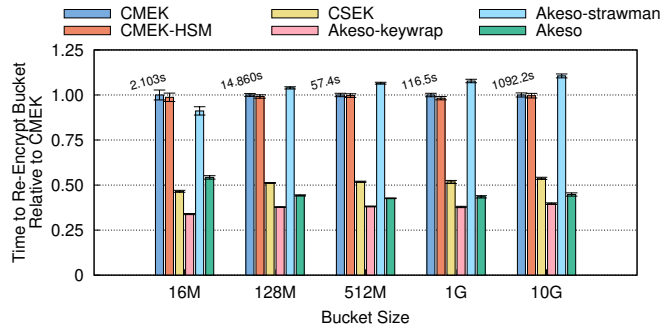

Figure 9: Time to re-encrypt a bucket of varying sizes, where each bucket object is 2 M, relative to the CMEK option. The numbers above the bars are the absolute latencies for the CMEK cases.

Table 2: Time (ms) to Decrypt Objects with Varying Layers of Encryption

Layers / Size	1	10		50		100	
10 K	0.02	0.04	(2.0×)	0.13	(6.5×)	0.24	(12.0×)
100 K	0.08	0.24	(3.0×)	1.02	(12.8×)	1.97	(24.6×)
1 M	1.49	3.34	(3.9×)	12.01	(17.2×)	21.62	(33.6×)
10 M	10.58	27.65	(2.6×)	105.78	(10.0×)	207.52	(19.6×)
100 M	90.89	310.90	(3.4×)	1245.90	(13.7×)	2010.10	(22.1×)

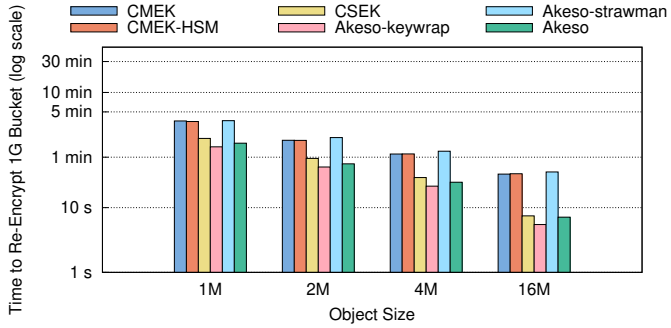

Figure 10: Time to re-encrypt a 1 G bucket, varying the size of the objects in the bucket.

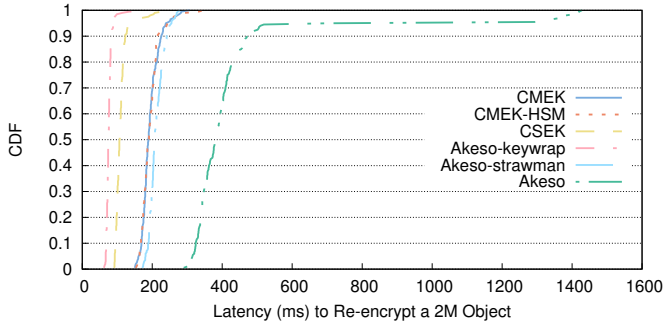

Figure 11: Latency to re-encrypt a 2 M object, varying the re-encryption method. The latency includes the network overheads of each method.

cloud VM includes the AES-NI hardware extension), the results demonstrate Akeso's flexibility to tailor the maximum encryption layers to the bucket's profile while keeping overhead low.

5.3 Key Rotation

We evaluate key rotation performance by measuring the time required to rotate the encryption key for all objects in the cloud store under various encryption options.

Time to re-encrypt bucket. We first evaluate buckets of varying sizes, but where each object is 2 M. As Figure 13 of Appendix C shows, 2 M reflects the 90th percentile of real-world object sizes. Our range of buckets sizes (16 M up to 10 G) spans nearly the 50th through 90th percentiles of our bucket size estimates for GCP, AWS, and Azure in Table 6 of Appendix C.

Figure 9 shows the mean re-encryption time relative to CMEK across 10 runs, with error bars representing one standard deviation. As bucket size increases, Akeso demonstrates superior scalability compared to other client-side strategies. For the 10 G bucket, Akeso (0.4477 ± 0.0093) outperforms Akeso-strawman (1.1060 ± 0.0108) by 2.5×, due to scalability of cloud functions for re-encryption. Akeso performs comparably to Akeso-keywrap (0.3967 ± 0.0044), despite providing stronger security guarantees through full data re-encryption rather than merely rewrapping keys.

Next we evaluate the re-encryption time for a 1 G bucket (which closely corresponds to the 80th percentile of bucket sizes as per Table 6), but vary the object sizes to capture 80% of the object counts, as per Figure 12 in Appendix C. Figure 10 presents mean times across 10 runs, with error bars omitted for readability. Storing data in fewer, larger objects proves more efficient: Akeso requires 98.7 ± 6.6 seconds for 1024 1 M objects but only 7.1 ± 0.3 seconds for 64 16 M objects (14× improvement). Akeso consistently outperforms alternatives while providing stronger security guarantees, achieving 4.6× faster re-encryption than CMEK for 16 M objects.

Table 3: Comparison of Time (s) to Re-encrypt a 1 G Bucket for Different VM Types and Locations

Location	VM Type	Akeso	Strawman
Same region	Non-Confidential	45.955	132.412
	AMD SEV	48.860	135.094
	AMD SEV-SNP	52.716	135.883
Different region	Non-Confidential	147.769	479.875
	AMD SEV	151.027	484.801
	AMD SEV-SNP	153.983	514.898

Time to re-encrypt an object. In addition to the time to encrypt an entire bucket, we are also interested in the time to encrypt a single object; as we described in §4.4, this time reflects a period during which a client object access may fail. Figure 11 shows that despite Akeso's superior performance demonstrated in the previous two experiments, it is consistently slower than the other encryption options for re-encrypting a single object. This is largely due to the latency between Akesod updating the metadata and the cloud function starting.

Overhead of Akesod enclave. To measure the overhead of running Akesod within an enclave, we perform a re-encryption operation using three identically configured AMD EPYC Milan VMs on Google Cloud: a non-confidential version, a confidential VM with AMD SEV, and a confidential VM with AMD SEV-SNP. As Table 3 shows, re-encrypting a 1 G bucket containing 2 M objects incurs an overhead factor in the range of 1.02–1.06× for AMD SEV and 1.04–1.15× for AMD SEV-SNP.

5.4 Monthly Costs

We use Google's Cloud Pricing Calculator [55] to estimate the monthly costs for running the four cloud storage encryption options: CMEK, CMEK-HSM, CSEK and Akeso. Specifically, we look at the costs for key rotation. We assume an object size of 1 M, and choose bucket sizes of 10 G, 100 G, 1 T and 10 T. All the cloud components are in the same region. We also assume key rotation happens every 30 days, and that the storage group has 32 members.

Table 4 shows the monthly cost of running the four encryption options. The difference between the estimates is that CMEK, CMEK-HSM, and CSEK use a regular *Compute Engine*, while Akeso uses a *Compute Engine* with AMD SEV support. Akeso is modestly more expensive than the non-secure alternatives, ranging from 15.6–19.3% more for a 10 G bucket, and only 2.4–14.4% more for a 10 T.

We additionally estimate the difference between running Akeso in a cloud-side enclave and running Akeso in a secure but non-confidential on-premises server. We assume the compute cost of running Akeso on-premises is negligible, the bucket size is 10 T, objects are 1 M, and that a single key rotation occurs each month. The on-premise Akesod performs the exact same functions as the cloud enclave version, but, as Table 5 shows, incurs over 4× the monthly expenses due to the cost of data egress [101] during a key rotation. Nevertheless, organizations may have special needs that require an on-premise Akesod; for instance, while side channels are outside the scope of our threat model, an organization that is wary of such attacks may opt for an on-premise deployment.

Table 4: Comparison of Monthly Costs (USD) for Running Akeso and Alternatives on an *N2D (n2d-standard-2)* Machine

Bucket Size	CMEK	CMEK-HSM	CSEK	Akeso
10 G	63.96	65.84	63.78	76.12
100 G	68.31	70.19	67.58	80.38
1 T	87.33	89.21	84.06	98.67
10 T	299.96	301.84	268.38	307.04

Table 5: Monthly Cost (USD) of Akeso Running on a Cloud Enclave vs. an On-Premises Server

Akeso	Cloud Enclave	On-Premises
Compute	75.97	0
Storage	204.7	204.7
Cloud Function	26.37	26.37
Data Egress	0	1126.4
Total	307.04	1357.47

6 Discussion

Frequency of key rotations. To ensure regular rotation of the ART group key by all clients, Akeso defaults to a simple round-robin system. The rate of key rotation (one week by default) is a global configuration, and clients take turns initiating a key update.

Unfortunately, a *rushing attacker* may try to exfiltrate all data before a re-encryption operation. Based on Figure 9, Akeso re-encrypts a bucket at ∼0.02 GB/s. To simulate an attacker, we launched a VM in the same region as the bucket, and measured that the attacker could download data from the bucket at nearly 0.15 GB/s. Rather than use a fixed epoch, Akeso could dynamically determine the epoch length by monitoring the volume of data access. This is based on the principle that higher data access volumes correlate with increased security risk.

Using the setup in Table 5 as an example, a 10 T bucket costs $26.37 (10 M req/month) for cloud functions. With egress at $0.11/G, clients (or an attacker) would access ∼240 G ($26.37/$0.11) before reaching this re-encryption cost threshold. Using these metrics, a simple rule of thumb is to re-encrypt when egress costs equal the re-encryption cost, as ∼1/20 of the total bucket volume would have been accessed, and Akeso could still encrypt roughly 13% of the bucket before the attacker finishes. Despite the speed advantage of a rushing attack, we note that Akeso is also valuable for ensuring data integrity—namely, that the attacker is eventually locked out from modifying existing data.

Data and crash consistency. Akeso relies on Google Cloud's guarantees for write consistency. Specifically, the cloud functions that re-encrypt the data update the ciphertext in a single write operation, and Google Cloud ensures this write either succeeds or fails and thus that the data is not left in an inconsistent state. If a cloud function crashes before updating the ciphertext, the `akeso_reencrypt` metadata value (the flag that Akesod sets and the cloud function unsets upon completion) will still be set. Although currently unimplemented, Akesod could inspect this flag and retry the operation. Existing systems like Ariadne [113] or Nimble [15] could handle the crash fault tolerance of Akesod itself.

Relaxing assumptions on cloud provider. In our threat model (§2.3), we make the strong assumption that the cloud provider adheres to data removal requests and does not persist versions of an object. Akeso makes this assumption to ensure an adversary with an old key cannot retrieve the corresponding old ciphertext. Note that even if the cloud provider's data removal is not perfect (e.g., old data exists in caches), the bar for an adversary is still quite high, as the adversary would have to compromise both a client (to learn a KEK) and then the cloud infrastructure (to recover old versions of the objects).

Dynamic group membership. Like the original ART paper [42], our Akeso prototype does not yet support dynamic group membership. We note that the TreeKEM [24] and MLS [21] protocols that build upon ART prominently feature the ability for groups to add or remove members. We leave these extensions to future work.

7 Related Work

There is considerable research in securing remote storage. Our intent is not to survey this vast field, but rather contrast our work with prior efforts that similarly incorporate TEEs or prioritize key rotation.

TEEs. Several works integrate TEEs—notably Intel SGX [72, 91]—into the cloud's storage stack to ensure the confidentiality and integrity of the data, as well as an array of other properties. For instance, systems like Pesos [84] and SeGShare [52] use a TEE as a secure cloud-side proxy for managing encryption keys and enforcing fine-grained access controls. SPEICHER [20] instead stores data directly in the TEE (using the TEE as a secure caching layer), and develops advanced techniques for persisting the data with freshness guarantees. KVSEV [123] is a confidential key-value store built on AMD SEV, incorporating special measures to defend against attackers with physical access to DRAM who may attempt active DRAM corruption or replay attacks. Finally, rkt-io [114] aims to improve the storage performance of individual cloud applications by leveraging userspace I/O libraries within an SGX enclave for kernel-bypass I/O.

Although TEEs play a central role in previous works, Akeso's use of a TEE is much more limited, and even optional. The Akesod enclaved service functions merely as an ART group member, with the added responsibilities of managing group membership and updating object metadata during key updates. A storage group could instead run Akesod on a trusted on-premise server, and this decision fundamentally represents a trade-off in renting computing power (a confidential VM) or paying for data ingress and egress (the on-premise server).

Provisioning TEEs. For completeness, we also highlight the use of cloud storage for non-interactively provisioning a confidential VM with sensitive data; this use case is orthogonal to Akeso, despite the overlap in some terminology. Specifically, Google's Split-Trust Encryption Tool (STET) [65] uses Shamir secret sharing [110] to distribute a DEK across multiple key management systems, eliminating the need for unilateral trust in a single key manager. Customers can use STET to encrypt data on the client side, ensuring that only an attested confidential VMs can reconstruct the DEK and access the data.

Group key management. In the group storage setting, many systems use pairing-based cryptography [26] to cryptographically enforce access controls among the group members, and rotate keys on membership changes. The work of Piretti et al. [100] was perhaps the first to use Attribute-Based Encryption (ABE) [23, 67] in the context of a distributed, shared filesystem. Later, Excalibur [105] and Sieve [121] use ABE for acess control for cloud-side and client-side applications respectively. More recently, IBBE-SGX [44] applies Identity-Based Broadcast Encryption (IBBE) [103] to manage access controls in the face of group membership changes, and uses SGX to reduce the computational complexity. In contrast, Akeso's access control is fundamentally bucket-granular: Akeso uses the ART protocol to rotate keys *within* a bucket, while relying on the cloud provider's existing Identity and Access Management (IAM) controls for managing access *across* buckets.

Untrusted third-party re-encryption. Ateniese et al. [19] use proxy re-encryption (the public-key counterpart to updatable encryption) to develop a distributed filesystem with fine-grained access controls. Their system uses a key-wrapping scheme where a symmetric key encrypts a file, and a user's public key encrypts this symmetric key. A (mostly) untrusted server interposes on file access, and rewraps the key to the requestor's public key if the requestor has suitable permissions. Sieve [121] instead uses an ABE-based hybrid approach to cryptographically enforce access controls and revoke permissions, and a key homomorphism [27] to safely outsource the symmetric re-encryption of the data. Other systems use re-encryption to translate ciphertext to a more performant scheme [69], or simply explore techniques for improving the speed of re-encryption altogether [12, 25, 49, 85]. Akeso is unique not in its application or scheme for updatable re-encryption, but rather in its use of cloud-native services to scale the re-encryption of an *entire* cloud store.

8 Conclusion

We have presented Akeso, the first cloud storage system that guarantees the post-compromise security of a customer's data. Akeso notably adapts the continuous group key agreement protocols from messaging applications to the cloud setting, granting cloud users great control over their data's confidentiality and privacy. A distinct challenge in the cloud setting is the need to re-encrypt all storage on a key rotation. To scale this intensive operation, Akeso leverages an updatable encryption scheme, thus allowing untrusted cloud functions to perform the re-encryption without learning the plaintext. We demonstrated that Akeso scales re-encryption at modest monthly costs while maintaining overall I/O performance. To assist further research in developing secure and private cloud systems, we make our code publicly available at https://github.com/etclab/akeso-artifact.

Acknowledgments

We thank the anonymous reviewers and the revision editor for their helpful feedback. The authors of this work were supported, in part, by NSF grant CNS-2348130. Lily Gloudemans was supported by an REU supplement to that grant. The authors used ChatGPT-4o to revise the text in this paper to correct typos, grammatical errors, and awkward phrasing.

A Pseudocode Protocol Definitions

We list here the pseudocode for a group member to create a storage object, and for Akeso and the cloud functions to jointly re-encrypt the storage. These operations build upon the notation of §2.4 and follow the prose-level descriptions in §4.1.

As preliminaries, the system uses:

Symmetric encryption scheme:

- **k ← KeyGen**: Generate a key k.
- **ct ← Enc(k, o)**: Encrypt object o using key k.
- **o ← Dec(k, ct)**: Decrypt ciphertext ct using key k.

Authenticated encryption with additional data (AEAD):

- **k ← AEAD.KeyGen**: Generate a key k.
- **ct, tag ← AEAD.Enc(k, pt, ad)**: Encrypt plaintext pt using key k, producing the ciphertext ct and authentication tag tag. The tag also covers the additional data ad.
- **pt ← AEAD.Dec(k, ct, ad, tag)**: Decrypt ciphertext ct using key k, additional data ad, and authentication tag tag. If authentication succeeds, produces the plaintext pt; otherwise $\perp$.

Cryptographic hash function:

digest ← **H(data)**

Other primitives. We abstract the nested AES decryption operation of §4.1 with the function:

pt ← **NestedDecrypt(k, ct_hdr, ct)**: Decrypt the nested-encrypted ciphertext ct using key k and ciphertext header ct_hdr. On success, returns the plaintext pt; otherwise $\perp$.

Major Akeso algorithms. Following the notation in §2.4, we use $\mathcal{K}_k$ as an alias for the group encryption key at epoch k:

$$\mathcal{K}_k \leftarrow \text{secrets}_k(G)$$

We present Akeso's main algorithms in Algorithms 1, 2, and 3.

Algorithm 1 Member.CreateObject$_k$

```
Input o, oName                                  ▷ object content and name
  dek ← AEAD.KeyGen
  ct, dataTag ← AEAD.Enc(dek, o, oName)
  hdr, hdrTag ← AEAD.Enc(K_k, dataTag ‖ dek, nil)
  attrs ← {}
  attrs['akeso_header'] ← hdr ‖ hdrTag
  attrs['akeso_kek_hash'] ← H(K_k)
  Cloud.Put(oName, ct, attrs)
```

B Security Analysis

In this section, we sketch a proof that Akeso satisfies Definition 1 (PCS) using the notation from §2.4 and Akeso's major algorithms from Appendix A.

Algorithm 2 Akesod.Update$_k$

```
token ← KeyGen
▷ This Cloud.Pub triggers a cloud function to process an object
  when its metadata (but not content) changes. The cloud function
  receives the token as input. ◁
Cloud.Pub(token)
for all oName ∈ S do
    attrs ← Cloud.GetAttrs(oName)
    hdr, hdrTag ← attrs['akeso_header']
    deks, dataTag ← AEAD.Dec(K_k, hdr, nil, hdrTag)
    if Len(deks) = Config.MaxLayers then
        ▷ Reset to a single encryption layer ◁
        o, attrs ← Cloud.Get(oName)
        pt ← NestedDecrypt(K_k, attrs['akeso_header'], o)
        dek ← AEAD.KeyGen
        ct, dataTag ← AEAD.Enc(dek, pt, oName)
        hdr, hdrTag ← AEAD.Enc(K_{k+1}, dataTag ‖ dek, nil)
        attrs ← {}
        attrs['akeso_header'] ← hdr ‖ hdrTag
        Cloud.Put(oName, ct, attrs)
    else
        deks ← append(deks, token)
        hdr, hdrTag ← AEAD.Enc(K_{k+1}, datatag ‖ deks, nil)
        attrs['akeso_header'] ← hdr ‖ hdrTag
        attrs['akeso_kek_hash'] ← H(K_{k+1})
        attrs['akeso_reencrypt'] ← true
        Cloud.PutAttr(oName, attrs)
```

Algorithm 3 CloudFunction.Update$_k$

```
Input oName, token                              ▷ object name, re-encryption token
  ct, attrs ← Cloud.Get(oName)
  ct ← Enc(token, ct)
  attrs['akeso_reencrypt'] ← false
  Cloud.Put(oName, ct, attrs)
```

B.1 Assumptions

For simplicity, we assume the deployment does not place a limit on the maximum number of encryption layers. Additionally, we assume that a compromise occurs at the start of an epoch, and that any exfiltration of data simultaneously occurs at the start of that epoch.

B.2 Proof Sketch

Assume the adversary $\mathcal{A}$ compromises Akeso group member m_i in epoch k; that is, $\mathcal{A}$ invokes Compromise$_k(m_i)$, thereby learning secret$_k(G)$ and secret$_k(m_i)$. Concretely, for Akeso, we have that:

$$\text{secret}_k(G) = \{\mathcal{K}_k\},$$
$$\text{secret}_k(m_i) = \{\lambda_{i,k}, C_i\}$$

where $\mathcal{K}_k$ is the CGKA group key for epoch k, $\lambda_{i,k}$ is m_i's secret for epoch k (e.g., its ART leaf key), and C_i is m_i's long-term authentication credential. By our threat model assumptions in §2.3, we assume that $\mathcal{A}$ is nonpersistent, and thus there is some earliest epoch $k' > k$ for which $\mathcal{A}$ does not invoke Compromise. Without

loss of generality, assume that $k' = k + 1$. By construction, the storage system S only advances to epoch $k + 1$ if a member invokes Update_k. There are two cases:

Case 1 – $\text{Update}_k(m_i)$**:** If m_i invokes Update_k, then m_i updates $\text{secret}_{k+1}(G)$ and $\text{secret}_{k+1}(m_i)$. For Akeso, this means the creation of a new group key $\mathcal{K}_{k+1}$ and new member secret $\lambda_{i,k+1}$. The call to $\text{Update}(\text{m}_\text{i})_k$ in turn causes m_i to invoke Cloud.Pub(`KeyUpdate`), which broadcasts the new, public CGKA data (e.g., the updated ART public keys).

Upon receiving, `KeyUpdate`, Akesod updates its local CGKA state, deriving $\mathcal{K}_{k+1}$. Akesod then invokes Akesod.Update_k (see Algorithm 2); this appends a new DEK d' to each object's ciphertext header, and calls AEAD.Enc to encrypt this header with $\mathcal{K}_{k+1}$. The DEK d' serves as the updatable encryption token; Akesod invokes $\text{Cloud.Pub}(d')$ to broadcast the token to the untrusted cloud functions, which in turn call $\text{CloudFunction.Update}_k$ to re-encrypt all objects with d'.

From the update, $\mathcal{A}$ can learn the public CGKA state by using C_i to receive Cloud.Pub(`KeyUpdate`). Additionally, if $\mathcal{A}$ infiltrated the cloud backend—say, as an insider—$\mathcal{A}$ can observe $\text{Cloud.Pub}(d')$, and hence d'. Regarding confidentiality, for a given object o_x with name oName_x there are four cases:

- **(A)** $\mathcal{A}$ did not exfiltrate o_x ($\mathcal{A}$ did not call $\text{Cloud.Get}_k(\text{oName}_x)$). By the security of the CGKA protocol, $\mathcal{A}$ cannot derive $\mathcal{K}_{k+1}$. This implies that $\mathcal{A}$ cannot decrypt the o_x's ciphertext header, and thus cannot learn the DEKs for any inward layers of encryption. Thus, o_x is confidential □
- **(B)** $\mathcal{A}$ called $\text{Cloud.Get}_k(\text{oName}_x)$ and some group member called $\text{Cloud.Put}_k(\text{oName}_x, \ldots)$. For Akeso, when a member calls $\text{Cloud.Put}_k(\text{oName}_x, \ldots)$, the member generates a new ciphertext header with an object-specific, secret, initial DEK d_{0,o_x} The update then triggers Akesod.Update to append d' to the header's list of DEKs, and encrypt this header with $\mathcal{K}_{k+1}$. Since $\mathcal{A}$ cannot derive $\mathcal{K}_{k+1}$, $\mathcal{A}$ cannot learn d_{0,o_x}. Thus, o_x is confidential □
- **(C)** $\mathcal{A}$ called $\text{Cloud.Get}_k(\text{oName}_x)$ and the object has not since changed (no member of G called $\text{Cloud.Put}_k(\text{oName}_x, \ldots)$). In this case, if $\mathcal{A}$ learns d', then $\mathcal{A}$ has the complete list of DEKs for o_x and can decrypt each of its layers. However, since o_x's contents are no different, there is no loss of confidentiality beyond that of the initial breach □
- **(D)** A member calls $\text{Member.CreateObject}_k$ to create a new object o_x. This case reduces to the same argument as subcase **B**. Thus, o_x is confidential □

Case 2 – $\text{Update}_k(m_j)$**:** If m_j (where $j \neq i$) invokes Update_k, then $\mathcal{A}$ may use C_i (or its insider cloud access) to eavesdrop on Cloud.Pub(`KeyUpdate`) and $\text{Cloud.Pub}(d')$. Here, the `KeyUpdate` message (e.g., the updated ART public keys), combined with the unchanged $\lambda_{k,i}$, allow $\mathcal{A}$ to derive $\mathcal{K}_{k+1}$, and read any object with NestedDecrypt(...). By our threat model assumptions, $\mathcal{A}$ is non-persistent, and thus there is some epoch $k + n$ where $\mathcal{A}$ is unable to eavesdrop on Cloud.Pub messages. Specifically, if any member invokes Update_{k+n}, then by the CGKA protocol, $\mathcal{A}$ is unable to derive $\mathcal{K}_{k+n+1}$. This case now reduces to subcases **A**–**D** of **Case 1**, and the objects are confidential □

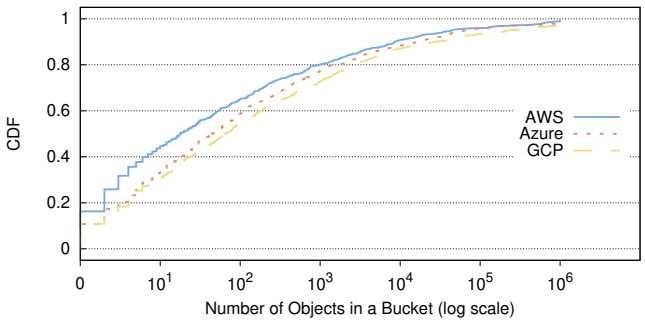

Figure 12: CDF of the number of objects in a bucket for a sample of 1,000 open buckets from each cloud provider.

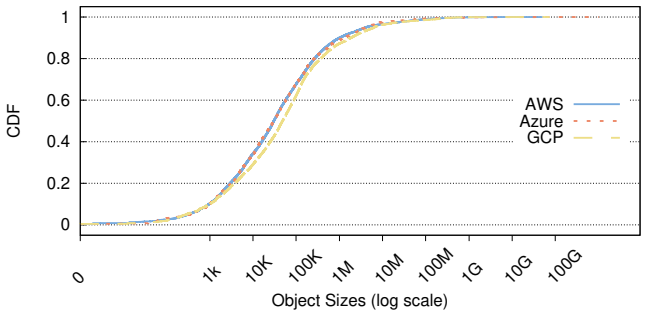

Figure 13: CDF of object sizes from a random sample of 1,000 open buckets from each cloud provider.

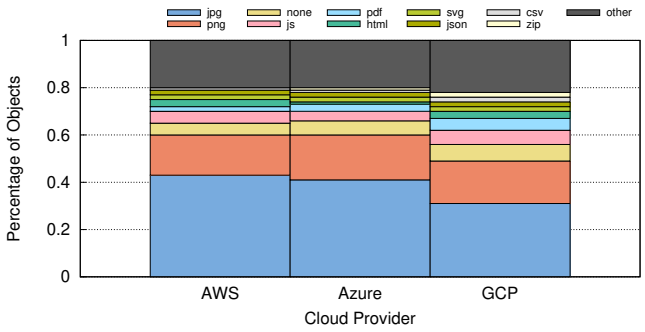

Figure 14: Stacked histogram of the ten most popular file extensions in a random sample of 1,000 GCP open buckets, and the percentage of objects in the sample with that extensions. The figure also shows the corresponding prevalence of these ten extensions in AWS and Azure. The category *none* indicates files without an extension, and *other* includes all other extensions.

C Pre-evaluation: Bucket Measurements

To add in evaluating Akeso, we conduct a brief measurement study to analyze the characteristics of real-world cloud storage. By examining the distribution of bucket sizes, object sizes, and object types,

Table 6: Bucket Size Estimates

Provider \ Size	16 M	128 M	512 M	1 G	10 G	100 G	1 T
GCP	46.4%	63.1%	73.9%	78.1%	88.8%	94.6%	98.6%
AWS	57.3%	73.4%	82.1%	86.1%	93.3%	96.7%	99.3%
Azure	43.4%	62.2%	72.2%	76.1%	89.9%	96.4%	99.3%

we can parameterize our experiments to more accurately reflect actual cloud storage usage.

We collect our measurements using Grayhat Warfare [68], a platform previously utilized in large-scale studies on cloud bucket misconfigurations [34, 43]. Grayhat Warfare indexes publicly accessible buckets belonging to AWS, Azure Blob Storage, and GCP, and provides a RESTful API for searching for buckets and the objects they contain. As of October 2024, Grayhat Warfare had indexed over 12.6 billion objects belonging to 327k AWS, 56k Azure, and 81k GCP buckets.

Grayhat Warfare offers both free and paid accounts; we registered for a free account, which has the following limits:

- A user can only search 22k–36k buckets for a given cloud provider (about 3.5–5.3% of a provider's buckets)
- A bucket listing limits its results to 1,000 buckets
- A listing of objects within a bucket limits its results to 2,000 objects (though such a listing also includes the *total count* of objects in that bucket)

We acknowledge that publicly accessible buckets may not fully represent all buckets. Despite this caveat, as well as the limitations of the free-tier account, our analysis still includes sample sizes large enough to achieve a 95% confidence level that our sample statistics accurately reflect the population of Grayhat's Warfare's large bucket index.

For each cloud provider, we list 1,000 buckets, and for each bucket we list up to the limit of 2,000 objects. Figure 12 shows that, regardless of provider, roughly 80% of buckets have fewer than 1,000 objects, and 90% have fewer than 10,000. Regarding the objects themselves, Figure 13 shows a log-normal distribution of sizes, with median object sizes varying between 31–49k, depending on provider. Based on the sampled bucket listings, Table 6 shows an estimate of the distribution of bucket sizes across provider (for instance, 46.4% of GCP buckets contain 16 M or less). These bucket size estimates compute the mean size of a bucket's listed objects (recall that Grayhat Warfare limits a listing to 2,000 objects), and then multiplies this average by the bucket's total object count. Finally, Figure 14 presents the top ten most common object (file) extensions for GCP, and the prevalence of these extensions among the three providers. The results indicate that media files account for more than 50% of objects, and that PDFs account for 2–5%. In other words, more than half of objects are read-only.

D Asychronous Ratcheting Tree

In this section, we provide further details on ART, namely its group setup operation.

Table 7: Lines of Code (Go) for each Akeso Component

Component	Lines of Code
`gcsfuse` **(lines added)**	1,206
Akesod	1,164
ART	1,414
Nested AES	235
Utilities	286
Total	4,305

Group setup. To create the initial tree, ART designates one group member as the *initiator*. On startup, the initiator generates an ephemeral DH key pair (suk, SUK) called the *setup key*, as well as its initial ART leaf key λ_0.[5] Each group member sends to the initiator their public long-term *identity key IK*, as well as an ephemeral public *prekey EK*. With each prekey EK_i, the initiator performs a DH computation with its secret *suk* to arrive at each member's initial ART leaf key λ_i. Using these λ_i, the initiator constructs the complete DH tree. The initiator then broadcasts a signed `GroupSetup` message consisting of the public prekeys EK_i and identity keys IK_i of each group member, the public setup key, and the tree T of public keys.

Upon receiving the `GroupSetup` message, a client performs a DH computation using the public setup key SUK and their private prekey ek_i, yielding their private leaf key λ_i. A client then extracts the keys from T that are on their λ_i's copath, and iteratively performs a DH computation with each copath node until deriving the root key, and ultimately the shared group key.

E Lines of Code

Table 7 shows the lines of code that we wrote (or added) for each component of Akeso.

References

[1] 2014. US court forces Microsoft to hand over personal data from Irish server. https://www.theguardian.com/technology/2014/apr/29/us-court-microsoft-personal-data-emails-irish-server.

[2] 2016. Dropbox hack 'affected 68 millions users'. https://www.bbc.com/news/technology-37232635.

[3] 2020. The police want your phone data. Here's what they can get — and what they can't. https://www.vox.com/recode/2020/2/24/21133600/police-fbi-phone-search-protests-password-rights.

[4] 2022. Amazon gave Ring videos to police without owners' permission. https://www.politico.com/news/2022/07/13/amazon-gave-ring-videos-to-police-without-owners-permission-00045513.

[5] 2022. Old Services, New Tricks: Cloud Metadata Abuse by UNC2903. https://cloud.google.com/blog/topics/threat-intelligence/cloud-metadata-abuse-unc2903/.

[6] 2023. Analysis of Storm-0558 Techniques for Unauthorized Email Access. https://www.microsoft.com/en-us/security/blog/2023/07/14/analysis-of-storm-0558-techniques-for-unauthorized-email-access/.

[7] 2023. Anatomy of a Cloud Incident | SentinelOne's Vigilance vs. IceFire Ransomware. https://www.sentinelone.com/blog/anatomy-of-a-cloud-incident-sentinelones-vigilance-v-icefire-ransomware/.

[8] 2023. Microsoft comes under blistering criticism for "grossly irresponsible" security. https://arstechnica.com/security/2023/08/microsoft-cloud-security-blasted-for-its-culture-of-toxic-obfuscation/.

[9] 2023. New Attack Vector In The Cloud: Attackers caught exploiting Object Storage Services. https://www.securityjoes.com/post/new-attack-vector-in-the-cloud-attackers-caught-exploiting-object-storage-services.

[5] We use lowercase names for private keys, and uppercase for public keys.

[10] 2023. The attack on ONUS – A real-life case of the Log4Shell vulnerability. https://cystack.net/research/the-attack-on-onus-a-real-life-case-of-the-log4shell-vulnerability.

[11] 2024. Public Cloud Security Breaches. https://www.breaches.cloud/.

[12] Calvin Abou Haidar, Benoit Libert, and Alain Passelègue. 2022. Updatable Public Key Encryption from DCR: Efficient Constructions With Stronger Security. In *ACM Conference on Computer and Communications Security (CCS)*.

[13] AMD. 2020. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More.* Technical Report. AMD.

[14] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[15] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. 2023. Nimble: Rollback Protection for Confidential Cloud Services. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[16] Inc. Apple. 2025. Apple can no longer offer Advanced Data Protection in the United Kingdom to new users. https://support.apple.com/en-us/122234.

[17] Frederik Armknecht, Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno, and Davide Vernizzi. 2008. An Efficient Implementation of Trusted Channels Based on OpenSSL. In *Workshop on Scalable Trusted Computing (STC)*.

[18] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux containers with Intel SGX. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[19] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. 2005. Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage. In *Network and Distributed System Security Symposium (NDSS)*.

[20] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *USENIX Conference on File and Storage Technologies (FAST)*.

[21] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. 2023. The Messaging Layer Security (MLS) Protocol. RFC 9420. https://www.rfc-editor.org/info/rfc9420

[22] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[23] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-Policy Attribute-Based Encryption. In *IEEE Symposium on Security and Privacy*.

[24] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. 2018. *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups: A protocol proposal for Messaging Layer Security (MLS).* Research Report. Inria Paris. https://inria.hal.science/hal-02425247

[25] Dan Boneh, Saba Eskandarian, Sam Kim, and Maurice Shih. 2020. Improving Speed and Security in Updatable Encryption Schemes. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*.

[26] Dan Boneh and Matthew K. Franklin. 2001. Identity-Based Encryption from the Weil Pairing. In *International Cryptology Conference (CRYPTO)*.

[27] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. 2013. Key Homomorphic PRFs and Their Applications. In *International Cryptology Conference (CRYPTO)*.

[28] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security Symposium*.

[29] Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Yao Jiang. 2020. Fast and Secure Updatable Encryption. In *International Cryptology Conference (CRYPTO)*.

[30] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[31] Timo Brecher, Emmanuel Bresson, and Mark Manulis. 2009. Fully Robust Tree-Diffie-Hellman Group Key Exchange. In *International Conference on Cryptology and Network Security (CANS)*.

[32] Emmanuel Bresson, Olivier Chevassut, David Pointcheval, and Jean-Jacques Quisquater. 2001. Provably Authenticated Group Diffie-Hellman Key Exchange. In *ACM Conference on Computer and Communications Security (CCS)*.

[33] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *ACM Conference on Computer and Communications Security (CCS)*.

[34] Jack Cable, Drew Gregory, Liz Izhikevich, and Zakir Durumeric. 2021. Stratosphere: Finding Vulnerable Cloud Storage Buckets. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. The first three authors made substantial contributions.

[35] Christian Cachin, Jan Camenisch, Eduarda Freire-Stögbuchner, and Anja Lehmann. 2017. Updatable Tokenization: Formal Definitions and Provably Secure Constructions. In *International Conference on Financial Cryptography and Data Security (FC)*.

[36] Chengjun Cai, Yichen Zang, Cong Wang, Xiaohua Jia, and Qian Wang. 2022. Vizard: A Metadata-hiding Data Analytic System with End-to-End Policy Controls. In *ACM Conference on Computer and Communications Security (CCS)*.

[37] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.

[38] Weikeng Chen, Thang Hoang, Jorge Guajardo, and Attila A. Yavuz. 2022. Titanium: A Metadata-Hiding File-Sharing System with Malicious Security. In *Network and Distributed System Security Symposium (NDSS)*.

[39] Weikeng Chen and Raluca Ada Popa. 2020. Metal: A Metadata-Hiding File-Sharing System. In *Network and Distributed System Security Symposium (NDSS)*.

[40] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2017. A Formal Security Analysis of the Signal Messaging Protocol. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.

[41] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. 2016. On Post-compromise Security. In *IEEE Cloud Security Foundations Symposium (CSF)*.

[42] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. 2018. On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees. In *ACM Conference on Computer and Communications Security (CCS)*.

[43] Andrea Continella, Mario Polino, Marcello Pogliani, and Stefano Zanero. 2018. There's a Hole in that Bucket! A Large-scale Analysis of Misconfigured S3 Buckets. In *Annual Computer Security Applications Conference (ACSAC)*.

[44] Stefan Contiu, Rafael Pires, Sebastien Vaucher, Marcelo Pasin, Pascal Felber, and Laurent Reveillere. 2018. IBBE-SGX: Cryptographic Group Access Control Using Trusted Execution Environments. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[45] Alex Davidson, Amit Deo, Ela Lee, and Keith Martin. 2019. Strong Post-Compromise Secure Proxy Re-Encryption. In *Australasian Conference on Information Security and Privacy (ACISP)*.

[46] Yvo Desmedt, Tanja Lange, and Mike Burmester. 2007. Scalable authenticated tree based group key exchange for ad-hoc groups. In *International Conference on Financial Cryptography and Data Security (FC)*.

[47] W. Diffie and M. Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* 22, 6 (1976).

[48] Betül Durak, Gwangbae Choi, and Serge Vaudenay. 2021. Post-Compromise Security in Self-Encryption. In *Conference on Information-Theoretic Cryptography (ITC)*.

[49] Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. 2017. Key Rotation for Authenticated Encryption. In *International Cryptology Conference (CRYPTO)*.

[50] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[51] Chris Farris. 2023. Public Cloud Security Breaches: LastPass. https://www.breaches.cloud/incidents/lastpass/.

[52] Benny Fuhry, Lina Hirschoff, Samuel Koesnadi, and Florian Kerschbaum. 2020. SeGShare: Secure Group File Sharing in the Cloud using Enclaves. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[53] Kenneth Goldman, Ronald Perez, and Reiner Sailer. 2006. Linking Remote Attestation to Secure Tunnel Endpoints. In *Workshop on Scalable Trusted Computing (STC)*.

[54] Dan Goodin. 2025. Copilot exposes private GitHub pages, some removed by Microsoft. https://arstechnica.com/information-technology/2025/02/copilot-exposes-private-github-pages-some-removed-by-microsoft/.

[55] Google. 2023. Google Cloud Pricing Calculator. https://cloud.google.com/products/calculator.

[56] Google. 2023. Google Cloud Storage. https://cloud.google.com/storage.

[57] Google. 2024. Cloud HSM architecture. https://cloud.google.com/docs/security/cloud-hsm-architecture.

[58] Google. 2024. Cloud Storage Fuse. https://cloud.google.com/storage/docs/gcs-fuse.

[59] Google. 2024. Cloud Storage Fuse Caching. https://cloud.google.com/storage/docs/gcsfuse-cache.

[60] Google. 2024. Cloud Storage Quotas & Limits. https://cloud.google.com/storage/quotas.

[61] Google. 2024. Default encryption at rest. https://cloud.google.com/docs/security/encryption/default-encryption.

[62] Google. 2024. gcsfuse. https://github.com/GoogleCloudPlatform/gcsfuse.

[63] Google. 2024. gcsfuse - Read/Writes Semantics. https://github.com/googlecloudplatform/gcsfuse/blob/master/docs/semantics.md#writes.

[64] Google. 2024. Overview of the Pub/Sub service. https://cloud.google.com/pubsub/docs/pubsub-basics.

[65] Google. 2024. Split-trust Encryption Tool. https://cloud.google.com/confidential-computing/docs/split-trust-encryption-tool.

[66] Google LLC. 2024. Google Cloud Terms of Service. https://cloud.google.com/terms.

[67] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. 2006. Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data. In *ACM Conference on Computer and Communications Security (CCS)*.

[68] GrayhatWarfare 2024. Public Buckets by GrayhatWarfare. https://buckets.grayhatwarfare.com/.

[69] Matthew Green, Susan Hohenberger, and Brent Waters. 2011. Outsourcing the Decryption of ABE Ciphertexts. In *USENIX Security Symposium*.

[70] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *ACM Symposium on Cloud Computing (SOCC)*.

[71] Stephen Herwig, Christina Garman, and Dave Levin. 2020. Achieving Keyless CDNs with Conclaves. In *USENIX Security Symposium*.

[72] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[73] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. 2021. Confidential computing for OpenPOWER. In *European Conference on Computer Systems (EuroSys)*.

[74] Intel 2014. *Intel Software Guard Extensions Programming Reference.* Intel.

[75] Intel. 2021. *Intel Trust Domain Extensions White Paper.* Technical Report. Intel.

[76] Matt Kapko. 2023. LastPass Breach Timeline: How a monthslong Cyberattack Unraveled. https://www.cybersecuritydive.com/news/lastpass-cyberattack-timeline/643958/.

[77] David Kaplan. 2017. *Protecting VM Register State with SEV-ES.* Technical Report. AMD.

[78] David Kaplan, Jeremy Powell, and Tom Woller. 2021. *AMD Memory Encryption.* Technical Report. AMD.

[79] Shaharyar Khan, Ilya Kabanov, Yunke Hua, and Stuart Madnick. 2022. A Systematic Analysis of the Capital One Data Breach: Critical Lessons Learned. *ACM Transactions on Privacy and Security* 26, 1 (Nov. 2022).

[80] Yongdae Kim, Adrian Perrig, and Gene Tsudik. 2000. Simple and Fault-Tolerant Key Agreement for Dynamic Collaborative Groups. In *ACM Conference on Computer and Communications Security (CCS)*.

[81] Yongdae Kim, Adrian Perrig, and Gene Tsudik. 2004. Tree-based group key agreement. *ACM Transactions on Information and System Security (TISSEC)* 7, 1 (Feb. 2004).

[82] Michael Klooß, Anja Lehmann, and Andy Rupp. 2019. (R)CCA Secure Updatable Encryption with Integrity Protection. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[83] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2019. Integrating Remote Attestation with Transport Layer Security. https://arxiv.org/abs/1801.05863.

[84] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy Enhanced Secure Object Store. In *European Conference on Computer Systems (EuroSys)*.

[85] Anja Lehmann and Björn Tackmann. 2018. Updatable Encryption with Post-Compromise Security. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[86] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. 2023. Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines. In *USENIX Annual Technical Conference (ATC)*.

[87] Mengyuan Li. 2022. *Understanding and Exploiting Design Flaws of AMD Secure Encrypted Virtualization.* Ph. D. Dissertation. https://etd.ohiolink.edu/.

[88] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *USENIX Security Symposium*.

[89] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security Symposium*.

[90] Arm Limited. 2021. *Arm Confidential Computer Architecture.* Technical Report. Arm Limited.

[91] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[92] Peihan Miao, Sikhar Patranabis, and Gaven Watson. 2023. Unidirectional Updatable Encryption and Proxy Re-encryption from DDH. In *International Workshop on Public Key Cryptography (PKC)*.

[93] Microsoft. 2023. Azure Blob Storage. https://azure.microsoft.com/en-us/products/storage/blobs.

[94] Microsoft. 2024. Azure Storage encryption for data at rest. https://learn.microsoft.com/en-us/azure/storage/common/storage-service-encryption.

[95] Lily Hay Newman. 2023. The Comedy of Errors That Let China-Backed Hackers Steal Microsoft's Signing Key. https://www.wired.com/story/china-backed-hackers-steal-microsofts-signing-key-post-mortem/.

[96] Ryo Nishimaki. 2022. The Direction of Updatable Encryption Does Matter. In *International Workshop on Public Key Cryptography (PKC)*.

[97] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *European Conference on Computer Systems (EuroSys)*.

[98] Adrian Perrig, Dawn Song, and J. D. Tygar. 2001. ELK, a New Protocol for Efficient Large-Group Key Distribution.

[99] Trevor Perrin and Moxie Marlinspike. 2016. The Double Ratchet Algorithm. https://www.signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[100] Matthew Pirretti, Patrick Traynor, Patrick McDaniel, and Brent Waters. 2006. Secure Attribute-Based Systems. In *ACM Conference on Computer and Communications Security (CCS)*.

[101] Google Cloud Platform. 2024. General Network Usage | Pricing | Cloud Storage | Google Cloud. https://cloud.google.com/storage/pricing#network-egress.

[102] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://www.rfc-editor.org/info/rfc8446

[103] Ryuichi Sakai and Furukawa Jun. 2007. Identity-based broadcast encryption. Cryptology ePrint Archive, Paper 2007/217. https://eprint.iacr.org/2007/217 https://eprint.iacr.org/2007/217.

[104] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. 2009. Towards Trusted Cloud Computing. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[105] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. 2012. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *USENIX Security Symposium*.

[106] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. 2024. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *IEEE Symposium on Security and Privacy*.

[107] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. HECKLER: Breaking Confidential VMs with Malicious Interrupts. In *USENIX Security Symposium*.

[108] Amazon Web Services. 2023. Amazon S3. https://aws.amazon.com/s3/.

[109] Amazon Web Services. 2024. Protecting data with encryption - Amazon Simple Storage Service. https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingEncryption.html.

[110] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (nov 1979).

[111] D. G. Steer, L. Strawczynski, Whitfield Diffie, and Michael J. Wiener. 1988. A Secure Audio Teleconference System. In *International Cryptology Conference (CRYPTO)*.

[112] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *ACM Conference on Computer and Communications Security (CCS)*.

[113] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *USENIX Security Symposium*.

[114] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. 2021. rkt-io: A Direct I/O Stack for Shielded Execution. In *European Conference on Computer Systems (EuroSys)*.

[115] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (ATC)*.

[116] Hannes Tschofenig, Yaron Sheffer, Paul Howard, Ionuț Mihalcea, and Yogesh Deshpande. 2023. *Using Attestation in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS).* Internet-Draft draft-fossati-tls-attestation-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-fossati-tls-attestation/03/ Work in Progress.

[117] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.

[118] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Symposium on Security and Privacy*.

[119] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX Fails in Practice. https://sgaxeattack.com/.

[120] Laurel Wamsley. 2017. Google Says It Will No Longer Read Users' Emails To Sell Targeted Ads. https://www.npr.org/sections/thetwo-

way/2017/06/26/534451513/google-says-it-will-no-longer-read-users-emails-to-sell-targeted-ads.

[121] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. 2016. Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[122] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. 2023. Credit Karma: Understanding Security Implications of Exposed Cloud Services through Automated Capability Inference. In *USENIX Security Symposium*.

[123] Junseung You, Kyeongryong Lee, Hyungon Moon, Yeongpil Cho, and Yunheung Paek. 2023. KVSEV: A Secure In-Memory Key-Value Store with Secure Encrypted Virtualization. In *ACM Symposium on Cloud Computing (SOCC)*.

[124] Maximilian Zinkus, Tushar M. Jois, and Matthew Green. 2022. SoK: Cryptographic Confidentiality of Data on Mobile Devices. In *Privacy Enhancing Technologies Symposium (PETS)*.