AccNimbus: Scalable Proofs of Data Possession for Cloud Storage

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

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

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

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

Abstract

As organizations increasingly store large volumes of data in the cloud, there is a growing need for efficient auditing mechanisms that verify data integrity without requiring full data downloads. This paper presents AccNimbus, a cloud-native provable data possession (PDP) system based on recent advances in RSA-based cryptographic accumulators. Like prior PDP schemes, AccNimbus uses a challenge-response protocol to probabilistically audit a random sample of stored data. However, unlike existing approaches that rely on clients to perform audits, AccNimbus shifts this responsibility to a trusted cloud service. To protect sensitive audit metadata and ensure trustworthy execution, AccNimbus operates within an AMD SEV-SNP trusted execution environment. Our evaluation on Google Cloud Storage shows that AccNimbus introduces minimal overhead and is a practical service, auditing a 1 GB storage bucket (100,000 objects) in less than five minutes.

CCS Concepts

• **Information systems → Cloud based storage**; • **Security and privacy → Denial-of-service attacks**; • **Theory of computation → Cryptographic protocols**.

Keywords

Proof of Data Possession, Cryptographic Accumulators, Trusted Hardware

ACM Reference Format:
Collin MacDonald, Pankaj Niroula, Aashutosh Poudel, and Stephen Herwig. 2025. AccNimbus: Scalable Proofs of Data Possession for Cloud Storage. In *Hardware and Architectural Support for Security and Privacy 2025 (HASP 2025), October 19, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3768725.3768733

1 Introduction

Organizations increasingly rely on cloud object storage services—such as Amazon S3 [37], Google Cloud Storage [19], and Azure Blob Storage [33]—to manage large volumes of data. These services have become the dominant form of cloud storage due to their scalability, high availability, and pay-as-you-go pricing.

While existing cryptographic techniques ensure the privacy and integrity of cloud-stored data, this paper addresses a complementary concern: enabling users to verify that cloud storage providers have not deleted or tampered with their storage objects prior to retrieval. Such auditing is increasingly important as organizations outsource data storage for information that is accessed only sporadically (such as archival backups), or which must be retained for regulatory compliance (such as financial records).

This capability, known as *Provable Data Possession (PDP)* [6], allows a client to verify that an untrusted server still holds the original data—without downloading it. PDP schemes use probabilistic proofs: the client (verifier) maintains a small, constant amount of (secret) metadata and randomly samples a number of objects (or portions of objects) from the server (prover). The client then issues a challenge, and the server responds with a proof of possession for each object, which the client checks against its metadata.

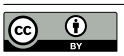
Several prior works propose efficient PDP techniques. For example, Filho and Barreto [18] and Ateniese et al. [6] use RSA-based (homomorphic) hash functions to generate integrity tags for each object. Erway et al. [15] instead employ RSA-based authenticated dictionaries to verify object store integrity. Our approach continues these RSA-centric efforts, but differs in two key ways. First, we use a much simpler cryptographic primitive—cryptographic accumulators—that can be implemented succinctly using standard library functions. Second, unlike prior work that treats auditing as a client-side responsibility, we integrate it as a first-class, cloud-native service.

We present AccNimbus, a cloud-native gateway for dynamic cloud storage that offers PDP-as-a-service. To our knowledge, AccNimbus is the first PDP scheme based directly on cryptographic accumulators,[1] which compactly represent sets and support efficient membership proofs. To protect the accumulator's trapdoor and sensitive audit metadata from the untrusted cloud, AccNimbus runs within a *Trusted Execution Environment (TEE)*. Our proof-of-concept uses AMD SEV-SNP [2, 26, 27], though the design is compatible with other TEEs, such as Intel SGX [23, 31].

In designing AccNimbus, we leverage recent advances in batching and aggregating RSA-based cryptographic accumulators [10] to further reduce the verifier's bandwidth and computational costs. We also address important edge cases, such as ensuring that object

*Both authors contributed equally to this research.

[1]We note that the PDP scheme of Erway et al. [15] uses the related concept of authenticated dictionaries.



HASP 2025, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2198-4/25/10
https://doi.org/10.1145/3768725.3768733

sampling is performed faithfully and without adversarial interference.

Contributions. We make the following contributions:

- We design a PDP scheme, AccNimbus, that combines trusted hardware with cryptographic accumulators to achieve efficient auditing of cloud storage buckets.
- We implement AccNimbus on the Google Cloud Platform, including several optimizations to reduce bandwidth and verifier computation.
- We evaluate AccNimbus's performance during auditing, updating, and re-initialization, and quantify the impact of each optimization. Our results show that AccNimbus audits a 1 GB cloud bucket with 100,000 objects in less than five minutes.

Paper Organization. This paper is organized as follows. In §2, we provide background on PDP, cryptographic accumulators, and TEEs We state our threat model and goals in §3. In §4, we describe the design and operations of the system, including the aggregation optimizations. We sketch a security analysis of AccNimbus in §5, and a performance evaluation on Google cloud in §6. We discuss future work in §7, and conclude in §8.

2 Background

2.1 Proof of Data Possession

A *Proof of Data Possession (PDP)* [6, 15, 43, 44] is a cryptographic protocol that enables a client to efficiently verify that a remote storage service still holds its data, thereby providing an audit mechanism against unauthorized deletion or modification. While a naïve solution might involve the client downloading the data and verifying a digital signature, PDPs aim to perform this audit efficiently—without requiring the client to retrieve the full storage, or the server to access the entire data store. PDPs can be viewed as a kind of (non-zero-knowledge) proof of knowledge, where the verifier knows the content being proven, and the goal is to minimize computational and communication overhead. We formalize these properties in §3.2.

Relation to Proof of Retrievability. *Proofs of Retrievability (PoRs)* [3–5, 11, 21, 25, 29, 38, 40, 42, 49, 50] are closely related to, and often conflated with, PDPs. PDPs are probabilistic protocols that verify whether storage remains largely intact by detecting significant corruption. In contrast, PoRs offer stronger guarantees of full storage retrievability by incorporating redundancy mechanisms, such as erasure coding or error-correcting codes. These mechanisms ensure that even if the audit misses some corruption, the client can still recover the complete storage (with high probability) from the verified portions. In this paper, we develop a PDP protocol, and leave PoR guarantees for future work.

2.2 Cryptographic Accumulators

A *cryptographic accumulator*, introduced by Benaloh and de Mare [9] and later formalized by Barić and Pfitzmann [8], enables the aggregation of a set of values into a compact, fixed-length digest called the *accumulator value*, while supporting constant-size membership proofs known as *witnesses*. A verifier can efficiently verify that an element is part of the set using only the membership witness and the accumulator value. Accumulators can be constructed using the strong RSA assumption in groups of unknown order (e.g., RSA groups) [8–10, 13] bilinear maps [34, 41], or Merkle hash trees [32].

There are several different types of accumulators, and in this work we deal with *dynamic accumulators* in the *accumulator manager* setting. A dynamic accumulator [7, 13, 47] allows updates to the set of accumulated elements, but requires that the previously issued witnesses be updated accordingly. The accumulator manager setting means that a trusted manager knows the trapdoor (e.g., the RSA modulus factorization) for the accumulator, which enables the manager to efficiently delete elements from the accumulator.

High-level algorithms. A dynamic cryptographic accumulator in the trusted manager settting consists of the following high-level algorithms:

- $\mathbf{Acc.Setup}(1^\lambda) \rightarrow \{A_0, \mathbf{pp}, \mathbf{sk}\}$ Given the initial security parameters 1^λ, initialize the (empty) accumulator value A_0, and generate the public parameters pp (e.g., the public key) and trapdoor sk. We assume pp is an implicit parameter to the remaining algorithms.
- $\mathbf{Acc.Add}(A_t, x) \rightarrow \{A_{t+1}, w_{(t+1,x)}, u_{t+1}\}$: Add a value x to the accumulator. Returns the updated accumulator value A_{t+1}, the membership witness $w_{(t+1,x)}$ for proving $x \in A_{t+1}$, and the update value u_{t+1} for updating witnesses created before $t+1$.
- $\mathbf{Acc.Del}(\mathbf{A_t}, \mathbf{x}) \rightarrow \{A_{t+1}, u_{t+1}\}$: Delete a value x from the accumulator. Returns the updated accumulator A_{t+1} and the update value u_{t+1} for updating witnesses created before $t+1$. The manager's trapdoor sk is an implicit parameter.
- $\mathbf{Acc.VerifyMemWit}(A_t, x, w_{(t,x)}) \rightarrow \{0, 1\}$: Verify a membership witness.
- $\mathbf{Acc.UpdateMemWit}(w_{(t,x)}, u_{t+1}) \rightarrow w_{(t+1,x)}$: Update a membership witness.

RSA instantiation. In this paper, we use the widely adopted RSA-based cryptographic accumulator [8, 9, 13]. To guarantee unique, collision-resistant encoding of sets, RSA accumulators require elements to be prime. We thus define a collision-resistant hash function HashToPrime from $\{0,1\}^*$ to the odd prime domain that closely follows Boneh et al.'s [10] implementation. Algorithm 1 shows the instantiation of an RSA accumulator. Note that the input x to Acc.Add, Acc.Del, and Acc.VerifyMemWit is a prime number from HashToPrime. Note also that a witness w for x is simply the accumulator value with the x exponent removed.

2.3 Trusted Execution Environments

Trusted Execution Environments (TEEs) are hardware-based security features that provide strong isolation for applications from the rest of the system—including privileged software like the OS or hypervisor—and some physical attacks. Their core capability is memory isolation: application memory is encrypted and integrity-protected in DRAM and decrypted only within the CPU package. This isolated memory region (as well as the application proper) is often referred to as an *enclave*.

TEEs assume that the CPU, microcode, and security processors are trusted, while all other components—BIOS, hypervisor, drivers,

Algorithm 1 RSA-Based Accumulator

```
function HashToPrime(data)
    ▷ H is a collision-resistant hash function ◁
    x ← H(data)
    if x is even then
        x ← x + 1
    while x is not prime do
        x ← H(x)
        if x is even then
            x ← x + 1
    return x
function Acc.Setup(1^λ)
    pk, sk ← RSAKeyGen(1^λ)
    return {A_0, pk, sk}
function Acc.Add(A_t, x)
    return {A_t^x, A_t, x}
function Acc.Del(A_t, x)
    return {A_t^{x^{-1}}, x^{-1}}
function Acc.VerifyMemWit(A_t, x, w_(t,x))
    return A_t ≟ w_(t,x)^x
function Acc.UpdateMemWit(w_(t,x), u_{t+1})
    return w_(t,x)^{u_{t+1}}
```

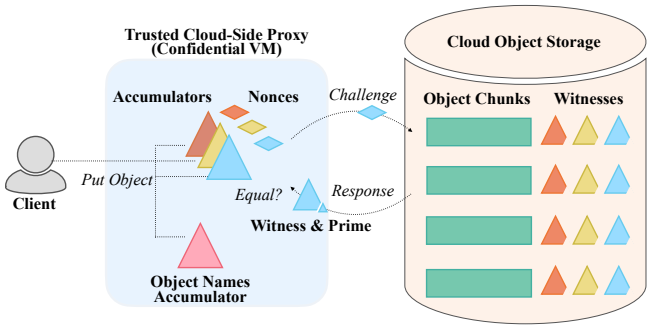


Figure 1: AccNimbus architecture. A client uploads content via a trusted proxy, which hashes it with nonces, records the keyed hashes in cryptographic accumulators, and stores the object with accumulator membership witnesses. To detect tampering, the proxy periodically challenges the untrusted provider to return a keyed hash and verifies its membership.

co-resident VMs, and hardware peripherals—are untrusted. AccNimbus relies on a TEE because it operates cloud-side as a service, but does not trust the cloud provider (the principal that it is auditing). AccNimbus handles sensitive data, including an RSA private key and other audit-related cryptographic material, and a TEE provides security guarantees that the untrusted provider cannot observe or interfere with these values.

Intel SGX [23, 31] was the first widely adopted, general-purpose TEE. It supports enclaves at the granularity of a part of a process, promoting small *trusted computing bases (TCBs)*, but limiting compatibility with legacy applications. In contrast, newer TEEs such as AMD SEV-SNP [2, 26, 27], Intel TDX [24], and Arm CCA [30] support enclaves at the virtual machine level (*confidential VMs*), enabling unmodified applications to run securely. Our proof-of-concept implementation of AccNimbus uses an AMD SEV-SNP confidential VM, though its design is compatible with other TEEs, including Intel SGX.

3 Assumptions

3.1 Threat Model

We assume an attacker capable of tampering with a client's cloud object store. This includes modifying or deleting data, as well as launching attacks on freshness, such as discarding object modifications, or overwriting new data with old versions. The attacker may be a malicious cloud provider, an insider, or an external cloud-side adversary. The attacker's goal is to prevent the storage client from detecting the tampering. We trust that the hardware and firmware underlying AccNimbus is fully patched and free of vulnerabilities, such as side-channel attacks [12, 14, 16, 35, 36, 45, 46], and vulnerabilities that could undermine security-sensitive operations such as random number generation [1, 20, 22].

3.2 Goals

In designing AccNimbus, we aim to achieve the following functional requirements, which follow directly from the core properties of PDP schemes:

- **F1** The verifier's state must be O(1), rather than scale with the number of objects in the store.
- **F2** The server (cloud provider) should not process the entire store during an audit.
- **F3** The server should not send an object's content during an audit.
- **F4** The additional storage overhead required for each object to support proof-of-possession should be minimal. If we let $\tilde{F}$ represent the object F along with its proof metadata, then the expansion factor $\tilde{F}/F$ must be small.
- **F5** There should be no restriction on the number of times the verifier can challenge the server to prove data possession.

In short, the goals for a PDP scheme attempt to minimize the storage state and bandwidth resource requirements for both the server and verifier.

Non-goals. This paper does not aim to address data confidentiality. Confidentiality is an orthogonal concern, and AccNimbus's design is agnostic to whether the objects contain plaintext or ciphertext. Data availability is also a non-goal: AccNimbus focuses on detecting data corruption, not restoring lost data.

4 Design & Implementation

Figure 1 illustrates the high-level architecture of AccNimbus, which involves three main parties: *(1)* a set of *storage clients* sharing a cloud bucket, *(2)* a *trusted proxy* that mediates access to the bucket, and *(3)* an *untrusted cloud storage provider*. The core of the system is the trusted proxy, which runs in a trusted execution environment

Algorithm 2 Proxy Initialization

```
function Proxy.Init
    A ← []                                   ▷ Array of accumulators
    nonces ← []                              ▷ nonce for each accumulator
    pk ← []                                  ▷ RSA public key for each accumulator
    sk ← []                                  ▷ RSA private key for each accumulator
    ▷ Accumulator of all object (chunk) names ◁
    A_names, pk_names, sk_names ← Acc.Setup(1^λ)
    for all i ∈ [12] do
        A_i, pk_i, sk_i ← Acc.Setup(1^λ)
        nonces_i ←$ {0,1}^256
```

and: *(1)* proxies client I/O requests to the cloud bucket, *(2)* manages the cryptographic accumulators by storing them and updating their state upon client I/O operations (add, delete, modify), and *(3)* periodically audits the bucket's integrity by verifying proofs of possession from the cloud storage provider.

Notation.

- $\boldsymbol{a}$ is a vector of elements and a_i is the i^{th} element
- $\text{Len}(\boldsymbol{a})$ is the number of elements in vector $\boldsymbol{a}$
- $a \parallel b$ denotes concatenation for strings and byte arrays
- $[\ell]$ is the set of integers $\{0, 1, \ldots, \ell - 1\}$
- $x \xleftarrow{\$} S$ denotes sampling a uniformly random element $x \in S$
- $\boldsymbol{x} \xleftarrow{\$,n} S$ denotes sampling n uniformly random elements from S without replacement
- $\{0,1\}^\ell$ is the set of bit strings of length ℓ
- $\perp$ represents an error, exception, or failure

4.1 Initialization

Algorithm 2 outlines the proxy's initialization procedure. The proxy creates a set of RSA-based accumulators $\boldsymbol{A}$ and for each accumulator A_i, it generates a secret nonce nonce_i. Additionally, as an integrity check of the bucket's object names, the proxy maintains a separate accumulator A_{names} that concisely represents the names of all existing objects. The object accumulators $\boldsymbol{A}$, their nonces **nonces**, the object name accumulator A_{names}, and the public and private RSA keys for all of the accumulators constitute the state the proxy must maintain. In our implementation, we generate 12 accumulators, allowing for one audit per month, and requiring the proxy to reinitialize its accumulator set on a yearly basis.

4.2 I/O Operations

Algorithm 3 describes how the proxy handles client requests to create and delete storage objects (for space considerations, we do not show a request to modify an object, as AccNimbus effectively handles it as a composition of the delete and create operations).

When a client requests the creation of a new object, the proxy runs Proxy.CreateObject. This function splits the object's content into an array of chunks and adds each chunk to every accumulator A_i, incorporating the chunk's data, name, and the accumulator's nonce into the value it adds. The nonce is used for the audit challenge (see §4.3), while including the object name prevents swap

Algorithm 3 I/O Operations

```
function Proxy.CreateObject(objName, objData)
    chunks ← Chunk(objData)
    names ← []                               ▷ Chunk object names
    attrs ← []                               ▷ Attributes for each chunk object
    ▷ Updates to apply to witnesses of existing objects ◁
    updates ← [1, ..., 1]
    for all chunk_i ∈ chunks do
        name_i ← objName ‖ "_" ‖ i
        u ← []                               ▷ Updates from this chunk
        w ← []                               ▷ Witnesses for this chunk
        for all A_j ∈ A do
            x ←
            HashToPrime(nonce_j ‖ name_i ‖ chunk_i)
            A_j, w_j, u_j ← Acc.Add(A_j, x)
            updates_j ← updates_j · u_j
        attrs_i.witnesses ← w
        if i = 0 then
            attrs_i.numChunks ← Len(chunks)
        ▷ Update the witnesses for the prior chunks ◁
        for all k < i do
            for all w_j ∈ attrs_k.witnesses do
                Acc.UpdateMemWit(w_j, u_j)
    ▷ Update the witnesses of the existing objects in the store ◁
    Proxy.UpdateWitnesses(updates)
    for all i ∈ [Len(chunks)] do
        Bucket.PutObject(name_i, chunk_i, attrs_i)
        x ← HashToPrime(name_i)
        A_names, _, _ ← Acc.Add(A_names, x)
function Proxy.DeleteObject(objName)
    chunks ← GetObjectChunks(objName)
    ▷ Each element is a product of the values to remove for that
      accumulator ◁
    batchX ← [1, ..., 1]
    for all chunk_i ∈ chunks do
        for all A_j ∈ A do
            w_j ← chunk_i.attrs.witnesses_j
            data ← nonce_j ‖ chunk_i.Name ‖ chunk_i.Data
            x ← HashToPrime(data)
            ▷ Ensure chunk integrity by verifying first witness ◁
            if j = 0 ∧ ¬ Acc.VerifyMemWit(A_j, x, w_j) then
                return ⊥
            batchX_j ← batchX_j · x
        x ← HashToPrime(chunk_i.Name)
        A_names, _ ← Acc.Del(A_names, x)
        Bucket.DeleteObject(chunk_i.Name)
    ▷ Updates to apply to witnesses of existing objects ◁
    updates ← [1, ..., 1]
    for all A_i ∈ A do
        A_i, updates_i ← Acc.Del(A_i, batchX_i)
    Proxy.UpdateWitnesses(updates)
```

attacks—ensuring the provider does not hash object A when the audit targets object B. The proxy puts each chunk object in the bucket,

and sets as its attribute metadata the chunk's witness for each accumulator. Additionally, the proxy must update the witnesses of the prior existing objects in the bucket (Proxy.UpdateWitnesses) to account for the new chunk objects being added. This entails the proxy retrieving and updating the metadata for each existing object. Finally, in order to have an integrity guarantee on the names of all objects in the bucket (which is required for auditing), the proxy adds each chunk object's name to the proxy's A_{names} accumulator.

Deleting an object involves performing the inverse of the creation process. The proxy first retrieves all chunk objects associated with the given object name and verifies each chunk's integrity by using the first witness w_0 from the chunk's attribute metadata, and calling Acc.VerifyMemWit to confirm the chunk's membership in A_0. It then deletes the chunk objects from the bucket and removes their entries from the accumulator set $\boldsymbol{A}$. The proxy also updates the witnesses of the remaining objects to reflect these changes. Finally, it deletes each chunk name from A_{names}.

4.3 Audits

Every month, the proxy chooses a random sample of objects from the bucket and audits their integrity and availability. Algorithm 4 shows the challenge-response audit protocol. The proxy first uses the storage API to retrieve a listing of all objects in bucket. Since the untrusted cloud provider services this API call, and could thus remove objects from the listing, the proxy creates a fresh accumulator A_{tmp} with the same RSA key as A_{names}, adds each object name in the listing to A_{tmp}, and checks that the accumulator's resulting value matches A_{names}. A mismatch indicates that the cloud provider tampered with the listing. The proxy then chooses n random object names from the list (see later in this section for guidance on choosing n), and sends these n object names, along with one of the nonces nonce_i to the provider.

Upon receiving the audit request, the provider executes Svr.Response, which retrieves each object in the sample, and hashes the concatenation of the nonce_i, the object's name, and the object's data to a prime x. Additionally, the provider retrieves from the object's metadata attributes the witness w that corresponds to the accumulator A_i that uses nonce_i. The provider then returns the witness and prime for each of the sampled objects back to the proxy.

Once the proxy receives the provider's response, it simply invokes Acc.VerifyMemWit to verify that each $w^x = A_i$. After the proxy completes the audit, the proxy can no longer use nonce_i, and by association no longer uses (and can thus delete) A_i.

Sample size. The proxy needs to choose an audit sample size n to detect provider misbehavior with high probability. Suppose a bucket has N file chunks, of which m are corrupted. The probability of detecting i corrupted chunks when performing an audit of n chunks is a hypergeometric random variable X:

$$P\{X = i\} = \frac{\binom{m}{i}\binom{N-m}{n-i}}{\binom{N}{n}} \quad i = 0, 1 \ldots, n$$

Table 1 shows the number of blocks the verifier needs to audit to detect at least one corrupted block (with probability 95%, 99%, and 99.9%) when the provider has corrupted 1% of the storage.

Algorithm 4 Audit Operations

```
function Proxy.Challenge(n, i)
    chunkNames ← Bucket.List
    ▷ Create a fresh accumulator with the same secret key as A_names ◁
    A_tmp ← Acc.NewWithSecretKey(sk_names)
    ▷ Verify the bucket listing ◁
    for all name ∈ chunkNames do
        x ← HashToPrime(name)
        A_tmp, _, _ ← Acc.Add(A_tmp, x)
    if A_tmp ≠ A_names then
        return ⊥
    sample ←$,n chunkNames
    ▷ Send the cloud provider the randomly sampled object names and one of the nonces ◁
    Send(sample, nonce_i)
    ▷ Receive from the cloud provider the witness and hash (prime exponent) for each object in the sample ◁
    w, x ←Recv
    ▷ Verify the witnesses ◁
    for all j ∈ [Len(sample)] do
        if ¬ Acc.VerifyMemWit(A_i, x_j, w_j) then
            return ⊥

function Svr.Response(names, nonce_i)
    w ← []
    x ← []
    for all j ∈ [Len(names)] do
        data, attrs ← Bucket.GetObject(names_j)
        x_j ← HashToPrime(nonce_i, names_j, data)
        w_j ← attrs.witnesses_i
    ▷ Return to the proxy the witnesses and hashes (prime exponents) for the requested objects ◁
    Send(w, x)
```

Table 1: Number of File Chunks to Audit when Bucket has 1% Corruption

Chunks in Bucket	95%	99%	99.9%
100	95	99	100
1,000	258	368	497
10,000	294	448	665
100,000	298	458	685
1,000,000	299	459	688

4.4 Re-initialization

When the proxy has one nonce remaining, it must use it to verify the entire store and reinitialize a new set of accumulators with fresh nonces. This operation is expensive but infrequent. The reinitialization process begins similarly to Proxy.Challenge: the proxy lists all objects in the bucket and verifies the list against A_{names}. It then creates 12 new accumulators, each initialized with a new random nonce. For each object, the proxy retrieves its chunks, computes their hashes, and verifies their last unused witness against A_{11}. It then re-hashes each chunk, adds the corresponding exponent

Algorithm 5 Shamir Trick [39]

```
function ShamirTrick(w1, w2, x, y)
    if w1^x ≠ w2^y then
        return ⊥
    ▷ Given co-primes x and y, Bezout computes the Bezout coefficients a and b that satisfy ax + by = 1. ◁
    a, b ← Bezout(x, y)
    return w1^b w2^a
```

to each accumulator, updates the chunk's metadata with the new witnesses, and writes the chunk's metadata back to the bucket.

4.5 Optimizations

Proof of exponentiation. To reduce the computational costs on the proxy during an audit, we use Wesolowski's (non-zero-knowledge) proof of exponentiation (PoE) in group $\mathbb{G}$ [48], as generalized by Boneh et al. [10] to support all exponents (not just powers of two). In this scheme, both the prover (cloud provider) and verifier (proxy) have (w, x, A), and the prover seeks to convince the verifier that $A = w^x$. The advantage is that the verifier can check the claim with significantly less work than computing w^x directly. This is particularly useful when $x \in \mathbb{Z}$ is much larger than $|\mathbb{G}|$.

The protocol works as follows:

Protocol 1. *Proof of Exponentiation (PoE) [48]*

1. *Verifier sends a random odd prime ℓ to the prover.*
2. *Prover computes the quotient $q = \lfloor x/\ell \rfloor \in \mathbb{Z}$ and residue $r \in [\ell]$ such that $x = q\ell + r$. Prover sends $Q \leftarrow w^q \in \mathbb{G}$ to the verifier.*
3. *Verifier computes $r \leftarrow (x \bmod \ell) \in [\ell]$ and accepts if $Q^\ell w^r = A$.*

The protocol can be adapted easily to a non-interactive PoE (NIPoE) through the Fiat-Shamir heuristic [17].

Witness aggregation. Boneh et al. [10] also introduce a technique for aggregating RSA accumulator witnesses—compressing n individual membership witnesses into a single, constant-sized *aggregate witness*. The technique is a straightforward adaptation of the *Shamir Trick* [39] (see Algorithm 5), which computes an xy-th root of a group element g from an x-th root of g and a y-root of g (note that a witness is simply a root of the accumulator). Applying the trick iteratively, in a fold or reduce-like fashion, allows the aggregation of an arbitrary number of witnesses.

Combining techniques. The untrusted cloud provider (prover) can apply both NIPoE and witness aggregation to reduce an audit's required bandwidth as well as the computational costs of the verifier. The audit proceeds exactly as in Algorithm 4, but the server instead aggregates the witnesses $\boldsymbol{w}$ into a single witness $\hat{w}$, and computes a NIPoE π that $\hat{w}^{\prod \boldsymbol{x}} = A$. The server then sends $\{\hat{w}, \boldsymbol{x}, \pi\}$ to the proxy. Instead of invoking Acc.VerifyMemWit, the verifier verifies the proof π as per step *(3)* of Protocol 1.

4.6 Implementation

We implement AccNimbus using Go v1.24. With the exception of two Google Cloud libraries, AccNimbus uses only the Go standard library. We developed the trusted proxy and the untrusted provider's prover as standard Go web servers, configured with mutual TLS for an extra layer of security. We use Go's lightweight threads (goroutines) to parallelize much of the code, including uploading file chunks and metadata (trusted proxy), processing audit requests (provider), and handling audit verifications (trusted proxy). Additionally, when creating new witnesses or updating existing witnesses, we leverage goroutines to compute witnesses in parallel across all 12 accumulators.

Hashing to prime. Our HashToPrime implementation uses SHA-256 as the hash function H (see Algorithm 1). This choice reflects tradeoffs among prime size (affecting how quickly a prime is found), the bit-width of the prime's inverse (impacting the performance of Acc.Del and Acc.Update), and the likelihood of hash collisions.

5 Security Analysis

In this section, we present a security proof sketch for AccNimbus showing that an attacker who tampers with data cannot still pass the audit, except with negligible (or explicitly bounded) probability.

Adversary's goal. We consider a PPT adversary $\mathcal{A}$ controlling the untrusted cloud provider. The goal of $\mathcal{A}$ is to tamper with (modify or delete) at least one stored chunk and yet pass the audit.

Assumptions. We have the following assumptions.:

- HashToPrime is collision-resistant hash function that maps byte strings to the odd primes.
- The RSA accumulator has security under the strong-RSA assumption, which implies tha forging a valid membership witness for an element not accumulated is infeasible without the trapdoor.
- The TEE protects the proxy's secrets (trapdoor, nonces) from the provider.
- Sampling is uniform over the set of chunk names; the proxy detects tampering with the listing via the A_{name} equality check.

We use $\mathsf{negl}(\lambda)$ to denote an unspecified negligible function in the security parameter λ. Intuitively, a negligible function vanishes faster than the reciprocal of any polynomial in λ.

Proof Sketch.

Theorem 1 (Soundness of AccNimbus). *Let N be the number of stored chunks, and $m \geq 1$ the number of tampered chunks. Under the collision resistance of* HashToPrime *and the strong-RSA security assumption of RSA accumulators, the probability that an adversary convinces the proxy to accept an audit after tampering is at most*

$$Pr[accept] \leq \frac{\binom{N-m}{n}}{\binom{N}{n}} + negl(\lambda),$$

where n is the audit sample size and λ the security parameter.

Proof. During an audit, the proxy first validates the provider's bucket listing against A_{name}. Any omission or equivocation would yield either a hash collision or a forged accumulator witness, both

negligible under our assumptions. The proxy then selects n object chunks uniformly at random and reveals a fresh nonce. For each chunk, the provider must return a valid witness for $x = \text{HashToPrime}(\text{nonce} \parallel \text{name} \parallel \text{data})$. If the data was tampered with, the adversary has four options:

(1) Guess the nonce (negligible, since nonces are 256 bits).
(2) Find a hash collision so that tampered data maps to the same prime (negligible).
(3) Forge a membership witness for a prime never accumulated (negligible under Strong RSA).
(4) Manipulate the name set to hide tampered objects (detected by the A_{name} check).

Thus, conditioned on a tampered chunk being sampled, the adversary succeeds only with negligible probability. The only remaining way to evade detection is if all n are unmodified. This occurs with probability $\binom{N-m}{n} / \binom{N}{n}$, the standard hypergeometric bound (see Table 1). Therefore, by selecting n large enough, AccNimbus detects tampering with probability $1 - \epsilon$, where ϵ is negligible plus the sampling error. □

6 Evaluation

6.1 System Benchmarks

We evaluate AccNimbus's performance in terms of its overhead for auditing, updating witnesses, and re-initializing accumulators. We run these experiments on Google Cloud Platform, using an *nd2-standard-2* confidential VM with AMD SEV-SNP [2] to host the trusted proxy and, for our proof of concept implementation, the untrusted prover. The VM has 2 vCPUs and 8 GB of RAM. The VM runs in the same region and zone as the storage buckets.

Auditing. Figure 2 shows the average time required to perform an audit operation for different bucket sizes. An audit operation measures the time spent by the proxy to perform a full audit of the object storage, as outlined in Section 4.3.

We compare three different implementations of the audit operation performed by the proxy for a bucket with 1,000 chunks. The *Audit* variant does not use any optimizations and reflects the baseline performance. The *Audit Batch Optimized* variant achieves the lowest latency by creating batches of aggregated witnesses and is 5× faster than the baseline. The *Audit Optimized* variant, while reducing data transfer by 90% compared to other variants (Section 4.5), performs the worst in terms of latency, incurring a 2.1× overhead compared to the baseline. The average time is measured as a 10% weighted mean of the durations from 10 runs.

Updating witnesses. In this evaluation, we measure the average time spent by the proxy to add a new file to object storage[2]. When a file is uploaded, the proxy must update the state for each of its accumulators, for each chunk in the uploaded file. In addition to updating its own state, the proxy applies the updates for the new object(s) to the witnesses of all existing objects.

Because of this, the update time for a new file is dependent on the number of existing objects in the storage bucket (Figure 3). The overhead of the member witness update method (*Acc.UpdateMemWit* in Table 2) and its dependency on existing objects account for the higher latencies observed during witness updates. The average time is measured as a 10% weighted mean of the durations from 10 runs.

[2]A single file uploaded by a client can result in many objects being created in the storage bucket, as uploaded files are chunked for more efficient auditing.

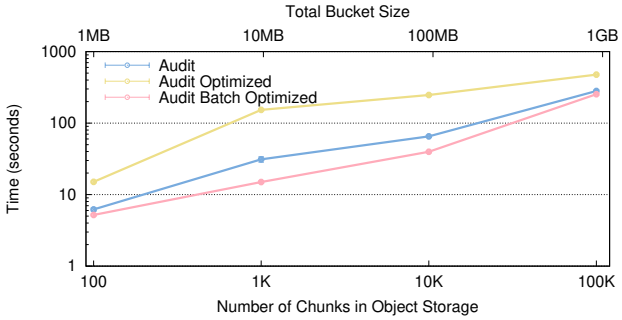


Figure 2: Audit time vs. number of chunks

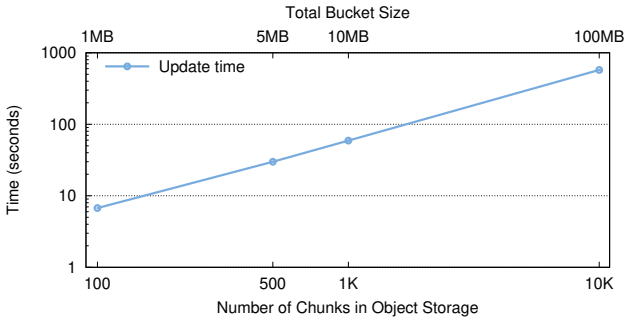


Figure 3: Update time vs. number of chunks

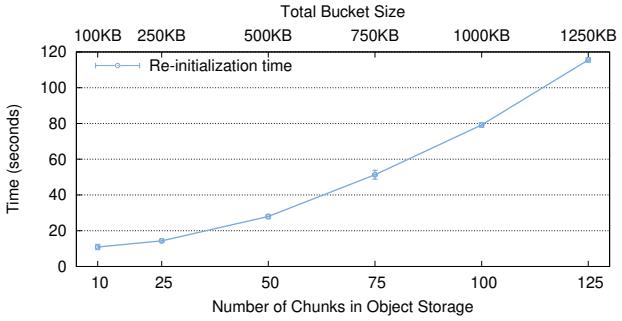


Figure 4: Re-initialization time vs. number of chunks

Re-initializing accumulators. Figure 4 shows the time required to re-initialize accumulators as a function of the number of chunks stored in the system. Re-initialization involves computing fresh accumulators and nonces for all stored data, which scales with the amount of data in the system. The measurements are averaged over 10 runs with 12 accumulators and show that re-initialization time grows approximately linearly with the number of chunks, taking around 79.2 seconds for 100 chunks (1000 KB of data) and 115.5 seconds for 125 chunks (1250 KB of data). Error bars represent one standard deviation across the 10 runs. These results represent a naive re-initialization approach without optimizations such as parallelization or batch processing of accumulator operations.

Table 2: Time (microseconds) for Accumulator Operations and Overhead Relative to SHA-256

Operation	Confidential VM	
HashToPrime	2,516	(3,302×)
Acc.Add	2,574	(3,377×)
Acc.Del	13,103	(17,196×)
Acc.VerifyMemWit	952	(1,249×)
Acc.UpdateMemWit	6,054	(7,945×)
NIPoE	1,799	(2,360×)
VerifyNIPoE	1,932	(2,536×)

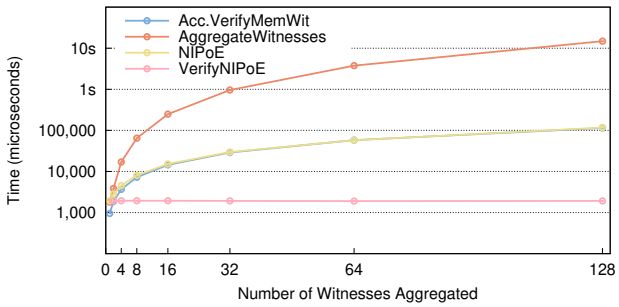


Figure 5: Scaling of witness verification, aggregation, and NIPoE generation and verification as the number of aggregated witnesses increases. The plots for Acc.VerifyMemWit **and** NIPoE **are nearly identical.**

6.2 Microbenchmarks

Cryptographic operations. Table 2 shows the microbenchmark performance overhead of each cryptographic operation, using Go's benchmarking tool. We conduct the benchmarks on the confidential VM (a Google Cloud *nd2-standard-2*). For comparison, the numbers in parentheses are the overheads relative to computing a SHA-256 hash of 1 K worth of data on each machine. Note that, in our implementation, Acc.Add and Acc.Del directly take the object contents and internally call HashToPrime, rather than merely taking as input the prime hash as in Algorithm 1. Each object is 1 K in size.

Figure 5 shows how the witness aggregation and PoE optimizations scale with the number of aggregated witnesses, again using Go's benchmarking tool. We observe that Acc.VerifyMemWit, AggregateWitnesses, and NIPoE are $O(n)$ operations, though the AggregateWitnesses operation has a larger constant multiplier. In contrast, VerifyNIPoE is $O(1)$. Critically, when compared to Table 2, VerifyNIPoE is more performant than individual witness verifications with Accc.VerifyMemWit after merely aggregating 2–3 witnesses.

Bandwidth usage. Figure 6 compares the amount of data received by the trusted proxy from the provider during a standard audit, an audit with witness aggregation and PoE optimizations, and an audit with batched witness aggregation and PoE optimizations. By using witnesses and PoE optimizations, the amount of data received by the trusted proxy can be reduced by upwards of 90%. When batched, the amount of data used is more than the non-batched version with full witnesses and PoE optimizations, but still considerably less than the standard audit.[3]

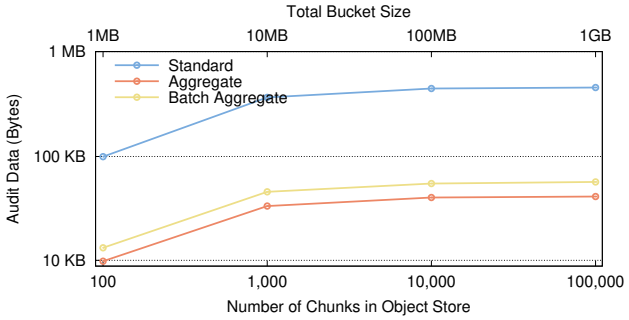


Figure 6: Data required to verify audit vs. number of chunks

7 Future Work

We believe that there are many ways that future research can build upon the work presented in this paper. We discuss several below:

Expanded Evaluation. The sensitivity of several configuration options, such as chunk and aggregation size, could be varied to better evaluate AccNimbus and reveal potential areas for optimizations.

Alternative Designs. Alternative accumulator designs could be explored to further optimize auditing, and different ways of overcoming the 12-accumulator limit could be investigated. Different hash functions, such as eXtendable-Output Functions (XOFs), which support arbitrary output lengths, as well as recent work [28] that constructs RSA-based accumulators without requiring prime elements, albeit with added complexity, could be explored too.

Additional Features. As discussed in §3.2, AccNimbus focuses on *Proof of Data Possession* not *Proof of Retrievability*. We believe AccNimbus could be expanded to support *Proof of Retrievability*, which provides stronger guarantees to the clients of AccNimbus that their data is recoverable.

8 Conclusion

In this paper, we present AccNimbus, a provable data possession scheme that combines recent advances in cryptographic accumulators with trusted hardware to deliver an efficient, cloud-native service. Our evaluations on Google Cloud Storage show that AccNimbus supports low-overhead, fast audits. To promote further research into provable data possession, our code is publicly available at https://github.com/etclab/accnimbus.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. The authors were supported by NSF grant CNS-2348130.

[3] In early experiments, the non-batched witnesses and PoE optimizations take twice as long compared to a standard audit. In contrast, the batched witnesses and PoE optimizations take half as long compared to a standard audit.

References

[1] Advanced Micro Devices, Inc. 2025. *AMD SEV Confidential Computing Vulnerability.* https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3019.html

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

[3] Gaspard Anthoine, Jean-Guillaume Dumas, Mélanie de Jonghe, Aude Maignan, Clément Pernet, Michael Hanling, and Daniel S. Roche. 2021. Dynamic Proofs of Retrievability with Low Server Storage. In *USENIX Security Symposium.*

[4] Frederik Armknecht, Jens-Matthias Bohli, David Froelicher, and Ghassan Karame. 2017. Sharing Proofs of Retrievability across Tenants. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS).*

[5] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O. Karame, Zongren Liu, and Christian A. Reuter. 2014. Outsourced Proofs of Retrievability. In *ACM Conference on Computer and Communications Security (CCS).*

[6] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable Data Possession at Untrusted Stores. In *ACM Conference on Computer and Communications Security (CCS).*

[7] Man Ho Au, Patrick P. Tsang, Willy Susilo, and Yi Mu. 2009. Dynamic Universal Accumulators for DDH Groups and Their Application to Attribute-Based Anonymous Credential Systems. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA).*

[8] Niko Bari and Birgit Pfitzmann. 1997. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT).*

[9] Josh Benaloh and Michael de Mare. 1993. One-way Accumulators: A Decentralized Alternative to Digital Signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT).*

[10] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2019. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In *International Cryptology Conference (CRYPTO).*

[11] Kevin D. Bowers, Ari Juels, and Alina Oprea. 2009. Proofs of Retrievability: Theory and Implementation. In *ACM Workshop on Cloud Computing Security (CCSW).*

[12] 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).*

[13] Jan Camenisch and Anna Lysyanskaya. 2002. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In *International Cryptology Conference (CRYPTO).*

[14] 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).*

[15] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. 2009. Dynamic Provable Data Possession. In *ACM Conference on Computer and Communications Security (CCS).*

[16] 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).*

[17] Amos Fiat and Adi Shamir. 1986. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *International Cryptology Conference (CRYPTO).*

[18] Décio Luiz Gazzoni Filho and Paulo Sérgio Licciardi Messeder Barreto. 2006. Demonstrating Data Possession and Uncheatable Data Transfer. In *International Conference on Security and Cryptography for Networks (SCN).*

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

[20] Google, LLC. 2025. *GCP-2025-007.* https://cloud.google.com/confidential-computing/confidential-vm/docs/security-bulletins#gcp-2025-007

[21] Chaowen Guan, Kui Ren, Fangguo Zhang, Florian Kerschbaum, and Jia Yu. 2015. Symmetric-Key Based Proofs of Retrievability Supporting Public Verification. In *European Symposium on Research in Computer Security (ESORICS).*

[22] Iain Thomson. 2025. *How to make any AMD Zen CPU always generate 4 from RDRAND.* https://www.theregister.com/2025/02/04/google_amd_microcode/

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

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

[25] Ari Juels and Burton S. Kaliski. 2007. PORs: Proofs of Retrievability for Large Files. In *ACM Conference on Computer and Communications Security (CCS).*

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

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

[28] Victor Youdom Kemmoe and Anna Lysyanskaya. 2024. RSA-Based Dynamic Accumulator without Hashing into Primes. In *ACM Conference on Computer and Communications Security (CCS).*

[29] Tung Le, Pengzhi Huang, Attila A. Yavuz, Elaine Shi, and Thang Hoang. 2023. Efficient Dynamic Proof of Retrievability for Cold Storage. In *Network and Distributed System Security Symposium (NDSS).*

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

[31] 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).*

[32] Ralph Merkle. 1979. *Secrecy, Authentication, and Public Key Systems.* Ph. D. Dissertation. http://www.ralphmerkle.com/papers/Thesis1979.pdf.

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

[34] Lan Nguyen. 2005. Accumulators from Bilinear Pairings and Applications. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA).*

[35] 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.*

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

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

[38] Hovav Shacham and Brent Waters. 2013. Compact Proofs of Retrievability. *Journal of Cryptology* 26, 3 (July 2013).

[39] Adi Shamir. 1983. On the Generation of Cryptographically Strong Pseudorandom Sequences. *ACM Transactions on Computer Systems* 1, 1 (Feb. 1983).

[40] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. 2013. Practical Dynamic Proofs of Retrievability. In *ACM Conference on Computer and Communications Security (CCS).*

[41] Shravan Srinivasan, Ioanna Karantaidou, Foteini Baldimtsi, and Charalampos Papamanthou. 2022. Batching, Aggregation, and Zero-Knowledge Proofs in Bilinear Accumulators. In *ACM Conference on Computer and Communications Security (CCS).*

[42] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. 2012. Iris: A Scalable Cloud File System with Efficient Integrity Checks. In *Annual Computer Security Applications Conference (ACSAC).*

[43] Yuzhe Tang, Ting Wang, Ling Liu, Xin Hu, and Jiyong Jang. 2014. Lightweight Authentication of Freshness in Outsourced Key-Value Stores. In *Annual Computer Security Applications Conference (ACSAC).*

[44] Stephen R. Tate, Roopa Vishwanathan, and Lance Everhart. 2013. Multi-User Dynamic Proofs of Data Possession Using Trusted Hardware. In *ACM Conference on Data and Application Security and Privacy (CODASPY).*

[45] 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.*

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

[47] Giuseppe Vitto and Alex Biryukov. 2022. Dynamic Universal Accumulator with Batch Update over Bilinear Groups. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA).*

[48] Benjamin Wesolowski. 2020. Efficient Verifiable Delay Functions. *Journal of Cryptology* 33, 4 (Oct. 2020).

[49] Jiawei Yuan and Shucheng Yu. 2013. Proofs of Retrievability with Public Verifiability and Constant Communication Cost in Cloud. In *International Workshop on Security in Cloud Computing (CloudComputing).*

[50] Qingji Zheng and Shouhuai Xu. 2011. Fair and Dynamic Proofs of Retrievability. In *ACM Conference on Data and Application Security and Privacy (CODASPY).*