Aegon: Self-Auditable Key Transparency

Hossein Hafezi* Alireza Shirzad* Benedikt Bünz

Kevin Lewi Dillon George Joseph Bonneau

Abstract

Key transparency enables a centralized encrypted messaging provider to publicly commit to the public keys it distributes, allowing clients to detect potentially malicious keys. Recent deployments by WhatsApp and iMessage demonstrate the promise of this approach, but they rely on third-party global auditors to detect misbehavior by the key server. No existing system supports auditing efficiently enough to be done by lightweight end users while also providing scalability to billions of users and short epoch latency.

We present Aegon, a key transparency scheme designed for global-scale encrypted messaging. Building on ideas from IronDict, Aegon avoids per-epoch work that scales with the full dictionary size: its server computation depends only on the number of updates in the current epoch, eliminating global invariance proofs and enabling epoch latency of under a minute (500× reduction compared to IronDict). Aegon further introduces a sharded dictionary design that reduces global parameters to shard-dependent sizes and enables horizontal scaling. To control long-term storage, Aegon uses proof caching to safely discard historical dictionary snapshots, so storage grows only with retained history.

We provide a production-grade Rust implementation of Aegon and demonstrate practical scalability to a dictionary with 4 billion entries, comparing it against the public codebase of WhatsApp Key Transparency (AKD). At the throughput of 1,250 updates per second, Aegon produces constant-size auditor proofs of under 30 KB, verifiable in under 65 ms and independent of the number of updates per epoch or of the directory fill. At a fully-populated 2^{32}-entry directory this is roughly an 80,000× reduction in audit proof size and a 370× reduction in verify time relative to AKD. All other server and client operations remain highly efficient and comparable to AKD, while Aegon achieves stronger privacy guarantees.

*Equal contribution.

Hossein Hafezi, University of Cambridge. Alireza Shirzad, University of Pennsylvania. Benedikt Bünz, NYU. Joseph Bonneau, NYU.

Contents

1 Introduction

End-to-end encrypted (E2EE) messaging platforms protect users' confidential communication from network adversaries and even from the service provider itself. However, this guarantee rests on a fragile foundation: clients must obtain the correct public keys for parties they wish to communicate with, typically relying on a central key server run by the very provider whom E2EE is meant to protect against. Traditionally, E2EE applications mitigate the risk of a provider-mounted *man-in-the-middle* (MitM) attack by require users to compare public keys out-of-band, for example by scanning QR codes or manually compare long key fingerprints. While they represent a gold standard for user security in key distribution, experience has shown these manual methods are cumbersome, error-prone, and rarely utilized.

In addition to manual key verification, some providers have adopted *key transparency* [Mel+15] to make their key distribution publicly observable. Transparent systems ensure that all actions of a central server are publicly observable and verifiable, allowing clients to monitor server behavior. In the secure messaging context [Ung+15], a key transparency log guarantees that clients can view all public keys a server has disseminated for their username, even though it cannot prevent the server from publishing maliciously chosen keys. Users are only alerted if the server published an expected key for their account and are freed from any reasoning about their contacts' keys, leading to a much simpler user experience. In general, transparency systems aim to detect, but not prevent, misbehavior, though it is often assumed this is sufficient to prevent misbehavior of a potentially *malicious-but-cautious* service provider.

Hence, while the goals are weaker than manual verification, KT provides a baseline level of assurance for (the vast majority of) users who in practice never perform manual verification. As a result, KT has gained significant traction in both industry and academia [Mel+15; Cha+19; Hu+21; Tya+22; Tzi+22; Len+24; Mal+23; Ros+24; Haf+26], with an ongoing IETF standardization effort [Int25]. Major messaging providers—including WhatsApp [LL23] and iMessage [App23]—have recently deployed KT. WhatsApp's system builds on the academic proposals SEEMless [Cha+19] and Parakeet [Mal+23], while iMessage's approach is based on CONIKS [Mel+15]. Among all transparency applications, KT is especially challenging due to the massive scale of the required data structures, going up to billions of users.

A *transparent dictionary*[1]—the core underlying primitive of KT—involves three logical entities, all of which have access to a shared public bulletin board on which the server publishes commitments:

- *Server:* A centralized service provider that maintains a mutable dictionary mapping *labels* (e.g. usernames or addresses) to *values* (e.g. public keys) and periodically publishes commitments to a public bulletin board. Publication occurs at regular intervals, called *epochs* (e.g., once every minute). The server is also responsible for answering queries from both clients and auditors.

- *Auditors:* Independent parties that monitor the bulletin board to verify that the dictionary is well-formed and that global invariants are preserved across epochs.

- *Clients (users):* End users who issue lookup queries to the server to retrieve values associated with specific labels of interest (e.g., public keys of contacts they wish to message). Clients also monitor their own labels and may optionally perform auditing of invariants.

[1] Also referred to in prior work as a verifiable key directory (VKD) or an auditable key directory (AKD).

1.1 Practical Deployment of Key Transparency

We focus on four key goals for KT implementations that, to the best of our knowledge, have not been simultaneously satisfied in a single work: *lightweight auditability*, *scalability*, *privacy*, and *short epoch latency*.

Lightweight Auditability. As previously noted, *auditors* are parties that observe the bulletin board and ensure that certain global invariants hold between consecutive epochs. Typically, the server publishes a proof of invariance together with a commitment to the new state of the dictionary, and each auditor runs a verification algorithm to check the proof. The soundness of the transparent dictionary critically depends on the existence of at least one honest auditor per epoch to detect any possibly misbehavior. While any party can perform auditing and today's implementations assume relatively powerful third-party auditors do so for altruistic reasons, there is no clear incentive for them to do so nor proof that they have done so correctly. Ideally, the auditing process should be lightweight enough that everyday users, even those using resource-constrained devices such as mobile phones, can perform auditing themselves.

Scalability. As mentioned earlier, key transparency poses unique challenges because of the potentially enormous userbases and resulting enormous dictionary sizes. For any key transparency system to be practical, the underlying architecture must be scalable, meaning it can be easily parallelized and capacity can increase as needed to handle global-scale deployments, i.e., supporting billions of entries, by adding computing resources linearly as the userbase grows. Importantly, at such large scales, the server should be distributed across multiple machines, while the computational overhead for clients and auditors must remain low to ensure usability and accessibility.

Short Epoch Latency. The delay between consecutive epochs is crucial as it determines how quickly a client's value registration or update becomes publicly visible. Long epochs can create a consistency gap as clients request changes between epochs. The server is left with two unappealing options. The server might continue responding with a stale value until the next epoch, which represents an obvious loss of security as the old value might be a compromised or revoked key. Alternately, the server might respond with the new value not yet consistent with the published commitment. In this case however, the server must provide some additional assurances (such as a digital signature) that the provided value will be reflected in the next epoch.

Short epochs (e.g., on the order of a seconds) mitigate these security problems, but typically pose performance challenges. Auditors and clients may be unable to fetch, verify, and pin a commitment before it is superseded by the next one, effectively making each epoch stale before it can be processed. In practice, an epoch duration of roughly 30–60 seconds offers a good balance and matches the regime in which deployed systems such as WhatsApp Key Transparency operate.

Privacy. Finally, privacy is an important requirement for key transparency, as messaging platforms typically do not reveal their entire user database (unlike some other transparency systems such as certificate transparency). At a high level, privacy means that the published commitments and proofs must not reveal more than what is strictly necessary. This restriction is important because any unnecessary data can carry sensitive signals. For example, if the server were to reveal patterns of public key updates, an adversary could deduce which users frequently rotate their keys and which do not, potentially identifying less cautious users or devices. Furthermore, if an attacker compromises a device's key and observes that the key remains unchanged, they could infer that the device owner is unaware of the breach, enabling prolonged exploitation.

1.2 A Deployed Baseline: WhatsApp Key Transparency

To ground our deployment goals, we consider WhatsApp's Key Transparency (AKD) [LL23], which is the largest production KT system to date and offers an open-source implementation.[2] As described in their whitepaper [Wha23], AKD builds on the academic proposals SEEMless [Cha+19] and Parakeet [Mal+23]. We emphasise that none of the data we report on AKD is private: every figure is derived from public sources, such as the WhatsApp Key Transparency whitepaper [Wha23], Meta's announcements [LL23; LL25], and Cloudflare's public auditor dashboard [Clo25].

As of this writing, WhatsApp AKD uses epochs that last roughly 30–60 seconds, each typically containing approximately 50,000–100,000 updates, yielding an average throughput of about 1,250 updates per second. This figure can be inferred from the audit proof size, which consists of the number of updates in an epoch. The underlying Merkle tree currently contains over 100 billion nodes, each representing an individual update. Its size can be estimated easily from the tree's *average depth*, which is itself revealed to every user during lookups—through the membership and non-membership proofs returned on each query—and to auditors during audit proofs. The publicly observable average depth of approximately 36.5 therefore indicates a tree of roughly 100 billion nodes, accumulated across approximately 1.74 million epochs since deployment.

- *Auditability.* Because AKD inherits the append-only Merkle tree of SEEMless, auditor work scales linearly with the number of updates per epoch. Concretely, auditing an epoch with $|\Delta|$ updates requires fetching and hashing $|\Delta| \cdot \text{tree depth} \cdot 32$ bytes, amounting to nearly 80 MB every minute. The main bottleneck is not hashing but bandwidth and the need to stay continuously online: auditing every epoch requires downloading over 100 GB per day which is prohibitive for typical clients. Consequently, WhatsApp currently relies on third-party auditors such as Cloudflare [CG24; Clo25]; the audit interface, however, is publicly accessible and any party may perform audits [Fac25].

- *Scalability.* The append-only nature of the underlying tree means historical nodes are retained indefinitely, even when no longer relevant. With 100 billion nodes already accumulated and the tree growing by roughly 75,000 entries per minute, storage cost and I/O overhead increase monotonically over the lifetime of the deployment.

- *Privacy.* AKD adopts the leakage-function privacy notion of SEEMless rather than a strong privacy guarantee. Each value lookup reveals auxiliary information such as the version number and the epoch of the last update.

- *Epoch latency.* AKD's epoch duration of 30–60 seconds matches the regime we identified as desirable in Section 1.1, and Aegon targets the same range.

In summary, AKD demonstrates that KT can be deployed at planetary scale, but also exposes concrete shortfalls along three of the four goals introduced in Section 1.1. These shortfalls motivate the design of Aegon whose contributions we describe next.

1.3 Contributions

We introduce Aegon, the first transparent dictionary, to our knowledge, that simultaneously achieves all four goals in Section 1.1: lightweight auditability, scalability, privacy, and short epoch latency. Aegon is built generically from a polynomial commitment scheme (PCS), and we provide an efficient

[2] https://github.com/facebook/akd

instantiation with the KZH [Kad+25] polynomial commitment scheme. Aegon is lightweight on the server side, enabling short epoch latency (under a minute) and natural support for horizontal scalability, allowing the server to be deployed across multiple machines. Aegon provides efficient lookup and consistency proof mechanisms for both servers and clients, and guarantees *strong privacy* when the underlying PCS is zero-knowledge.

Finally, Aegon enables lightweight auditing with only constant work required from the auditor per epoch. This efficiency further enables Aegon to support *fast-forwarding* via *incremental verifiable computation (IVC)*. We elaborate on these properties and the techniques that enable them below.

Constant-Cost Auditing via Client-Delegated Index Uniqueness. The server efficiency of Aegon stems from eliminating IronDict's per-epoch *global* index-uniqueness proof. The key insight is that index immutability, like value immutability, is a *per-client* property: each client only cares that its own index has not changed. Since clients already monitor their own values, delegating index monitoring to them adds no new trust assumption, yet it replaces the global index-uniqueness proof with a local check of constant cost per client. Concretely, this reduces the auditor's per-epoch task to constant size.

Formalization of Security and Privacy. We formally define security (completeness and soundness) and privacy notions for transparent dictionaries. In particular, we provide the first UC-based definition of privacy for transparent dictionaries, and we prove that Aegon satisfies it when instantiated with a zero-knowledge PCS. Beyond the conceptual value of a precise definition, the exercise also pays off concretely: by pinpointing which polynomials genuinely require zero-knowledge openings and avoiding zero-knowledge masking on the rest, we improve the efficiency of every lookup on the server-side by a factor of roughly $2\times$ compared to IronDict.

Sharding. Aegon scales to dictionaries of billions of entries via a *dictionary sharding* technique that splits the dictionary into η independent shards. Sharding yields three complementary benefits: each shard requires only a PCS setup sized for its own slice rather than the full dictionary; every server-side operation parallelizes naturally across shards, enabling horizontal scaling across machines; and dictionary construction is accelerated by running open addressing independently within each shard.

Optimal Server Storage. Aegon is storage-optimal: the server's storage grows only with the amount of historical data it retains. Instead of storing full snapshots of the dictionary at every epoch, the server maintains a collection of lookup and consistency proofs whose total size is proportional to the number of updates performed during that epoch. After these proofs are recorded, the previous dictionary state can be safely discarded. As a result, the server's storage overhead scales with the number of updates per epoch rather than with the total dictionary size. Because the cached proofs are independent across clients, Aegon further supports per-client retention policies—for instance, one user may keep the last six months of versions while another keeps only the last two.

Extensions for Practical Deployment. We extend Aegon with several additional features that are important for practical deployment. These enhancements include:

- *Dictionary expansion:* enabling the dictionary size to grow over time without compromising correctness or soundness.

- *Parameter updates:* supporting secure rotation or upgrade of underlying cryptographic parameters, such as the PCS setup and the verifiable pseudorandom function (VRF) keys.

These extensions make Aegon robust, flexible, and suitable for long-term deployment.

End-to-End Implementation. We implement Aegon and demonstrate that it scales to dictionaries with billions of entries, matching the demands of modern secure messaging applications, while supporting update rates of tens of thousands per second and horizontal scalability. We conduct our experiments on a cluster of 131 machines (128 shards, one coordinator, one label–value backend, and one masking server), with each shard provisioned with 16 vCPUs and 64 GB of memory. For a dictionary supporting 4 billion entries, our system publishes a 65,536-update batch in about 4 seconds and sustains a cluster-wide fill throughput of up to 18k entries per second. Auditor proofs remain under 30 KB and can be verified in under 65 ms on a commodity core, independent of dictionary fill. Client queries—including lookups and consistency checks—produce proofs of approximately 5 KB, which can be verified in under 7 ms.

2 Previous Work

2.1 Overview of Prior Work

We categorize existing work on transparent dictionaries into two main classes: *Merkle tree–based approaches* and *algebraic accumulator–based approaches*. Most constructions, including all currently deployed schemes, fall into the first category [Kim+13; Bas+14; Mel+15; Cha+19; Hu+21; Mal+23; Len+24]. Merkle tree–based designs offer concrete efficiency for both servers and clients, relying only on a collision-resistant hash function. Algebraic accumulator–based approaches include solutions built on bilinear accumulators [Tom+19] and RSA accumulators, such as VeRSA [Tya+22]. Their main limitation is a lack of scalability. For example, Tomescu et al. [Tom+19] report that supporting a dictionary of size 2^{20} would require hundreds of gigabytes of storage. Similarly, generating lookup proofs with VeRSA is computationally expensive and scales linearly with the dictionary size, restricting its use to dictionaries of only a few million entries. Due to their limited scalability, we do not focus on these works in this paper. A more recent work, IronDict [Haf+26], constructs a transparent dictionary generically from a polynomial commitment scheme. When instantiated with KZH polynomial commitment [Kad+25], IronDict achieves high efficiency and can scale to dictionaries of up to 1 billion entries.

Next, we review the existing work according to the criteria outlined earlier—namely, audit cost, scalability, privacy, and epoch latency—and finally present an asymptotic comparison table between prior schemes and our proposed scheme, Aegon.

Audit Cost. A limitation of Merkle tree–based approaches such as SEEMless [Cha+19], Parakeet [Mal+23] and OPTIKS [Len+24] is that auditors' cost grows linearly with the number of updates in the epoch. In practice, this requires auditors to perform nearly as much computation as the server itself, making client-side auditing infeasible in large-scale deployments. For instance, under the current throughput of WhatsApp KT, auditing is equivalent to hashing approximately $80\,\text{MB} / 60\,\text{s} \approx 1.33\,\text{MB/s}$, as reported in the Key Transparency dashboard [Clo25]. Consequently, current systems rely on third-party auditors, often non-profit watchdog organizations [CG24]. Another work, Merkle^2 [Hu+21], directly builds a transparent dictionary in which the values are tailored to public keys, and modifies the trust model to improve auditors' performance. They reduce the auditor's per-epoch workload to logarithmic in the dictionary's size by requiring only a single Merkle extension proof per epoch. However, Merkle^2 relies on the signature chain assumption, that clients maintain a secure key to sign their own key updates, which we consider unrealistic for secure messaging applications as we will discuss in Section 3.

Some proposals, such as Hekaton [Ros+24] and Verdict [Tzi+22], attempt to offload the auditor's work to the server by employing general-purpose proof systems. However, in practice, such approaches remain impractical due to the prohibitive cost of general-purpose proof systems. For example, because of the high cost of SHA-256 in an arithmetic circuit, Hekaton achieves a throughput of only 10 updates per second, even when running on a powerful cluster of 4,096 cores.

Motivated by the high auditor cost of Merkle tree–based approaches, recent work, IronDict, introduced an alternative formulation of transparent dictionaries based on polynomial commitment schemes. For a dictionary of size 1 billion, IronDict has an auditor-proof size of 8 KB per epoch, independent of the number of updates in the epoch.

Scalability. The main scalability issue of tree–based approaches such as SEEMless, Parakeet, and OPTIKS is their unbounded storage growth. These approaches rely on an append-only tree, an ever-growing data structure. The enormous size of these trees, coupled with their append-only nature,

creates significant challenges. Specifically, due to the append-only property, the tree must retain the entire history of all users' public keys—even outdated or irrelevant data. As a naive mitigation, OPTIKS proposes a *resetting mechanism*, i.e., periodically resetting the tree and preserving only the most recent value of each user and discarding prior history. However, this solution introduces a security caveat: for soundness, every user must be online at least once per reset period. Otherwise, a malicious server could modify a user's value during the period and restore it at the next reset, leaving the user unaware of the tampering due to the absence of a history. Despite these drawbacks, Merkle trees remain appealing due to their efficiency and hierarchical structure, which allows for horizontal scalability.

As demonstrated by its implementation, IronDict can scale to dictionary sizes of up to 10^9 entries; however, it still faces fundamental barriers to further scalability:

- *Not horizontally scalable.* Distributing the server's workload across multiple machines is challenging in IronDict, as some of its core operations (e.g., *the sumcheck protocol* [Lun+92]) are inherently non-distributable.

- *Large SRS size.* When instantiated with KZH, IronDict requires a structured reference string (SRS) whose size grows linearly with the dictionary size. For example, for a dictionary of size 1 billion, the SRS already occupies around 256 GB, and extending to 4 billion clients pushes the requirement to nearly 1 TB. Such an SRS is extensively large because the server must store it entirely in RAM. Furthermore, the SRS must be generated in a trusted manner—typically through a sequential multiparty ceremony [WCB25]—whose cost increases with the SRS size, making the setup prohibitively expensive.

- *Large storage overhead.* Persistent storage across epochs is another bottleneck in IronDict. Storing a full dictionary snapshot per epoch is prohibitively expensive, roughly 250 GB per epoch for a dictionary of size 4 billion. Such storage is unnecessarily wasteful, since consecutive epochs differ only in the applied updates. IronDict proposes a naive solution that exploits the homomorphic structure of KZH to store only inter-epoch differences together with an auxiliary input of size $O(N^{\frac{k-1}{k}})$ when instantiated with KZH-k. This optimization, however, has the following drawbacks: (i) the auxiliary input itself is extensively large, e.g., one quarter of the dictionary at $N = 10^9$ with $k = 16$; (ii) lookups in the latest epoch require field operations linear in the total updates across all retained epochs, trading storage for computation; and (iii) the approach depends on KZH's homomorphic structure and sublinear auxiliary input, and does not generalize to arbitrary PCS.

Privacy. CONIKS introduced an informal notion of privacy, which SEEMless later formalized as a leakage function. This function specifies exactly what information may be revealed, ensuring that the data published by the server—namely, commitments posted on the bulletin board together with the corresponding proofs—does not disclose anything beyond what the leakage function permits. For example, in SEEMless, the leakage function for value queries unnecessarily reveals both the value's version number and the epoch of its last update. Other tree-based systems, such as Parakeet and OPTIKS, adopt similar privacy definitions based on leakage functions.

IronDict is the first system to achieve what can be called *strong privacy*[3]. Instead of relying on a leakage function, it defines privacy using a zero-knowledge-based definition, without any proper formalization.

[3]The original paper uses the term *perfect privacy*, but we avoid such a loaded term.

Epoch Latency. In early systems such as CONIKS, a trade-off was created between epoch latency and the monitoring cost of clients. Since these systems lacked explicit support for *proofs of invariance*, a client had to naïvely verify all states of the dictionary to audit their value. Consequently, shorter epochs increased the client's cost within a fixed time window, raising the client's computational burden; conversely, longer epochs reduced this cost. This limitation was addressed in subsequent systems such as SEEMless, Parakeet, and OPTIKS, which introduced invariance proofs. In these designs, clients no longer need to check all dictionary states, allowing the use of much shorter epoch durations—on the order of a minute, as demonstrated in SEEMless.

In IronDict, updates are classified into two categories: *index assignment* for new labels and *value updates* for existing labels. The second case is efficient, requiring only n group operations, where n is the number of values updated within an epoch. In contrast, index assignments are more expensive, requiring a linear number of field operations in the dictionary size. Concretely, the *append-only proof* for index assignment involves a sumcheck over the entire dictionary, and sumcheck protocols do not parallelize efficiently. As reported by IronDict, for a dictionary of size 10^9, the sumcheck takes roughly 15 minutes. IronDict mitigates this cost by observing that the two update types rely on distinct underlying polynomials, which allows them to be processed asynchronously. As a result, value updates can adopt short epoch delays—similar to SEEMless—ensuring that clients do not receive stale responses. Meanwhile, index assignments can proceed on longer intervals, reflecting their higher computational cost. However, this design introduces a complication for new users: because their registration is delayed by about 15 minutes, they require a *promise message* from the server. This additional mechanism complicates the system and is something we aim to avoid in this work. This delay grows linearly with the size of the dictionary. For instance, with a dictionary of size 4 billion, the expected delay would be approximately 60 minutes, indicating that IronDict does not scale efficiently to extensive dictionaries.

Concurrent Work. Forget-me-not Trees [Kap26] is a concurrent work that builds a transparent dictionary supporting lightweight, mass-scale auditing. By embedding Bloom filters into the nodes of a Merkle tree, they obtain concretely small audit proofs (roughly 30–60 KB) from hash functions. We view their approach as complementary to Aegon, but it differs along three axes. First, their audit proof grows linearly with the number of updates in an epoch, whereas Aegon offers constant-sized audit proofs. Second, and more importantly, their privacy guarantees are substantially weaker (even compared to the other tree-based scheme): because the per-epoch Bloom-filter leakage is deterministic, an auditor can link multiple updates related to one index/label across epochs. By contrast, Aegon's audit proofs reveal nothing to the auditor beyond an upper bound on the size of the dictionary. Finally, their construction incurs a substantial server-side memory footprint: with a Bloom-filter size of $m = 2^{27}$ (used throughout their evaluation), a tree with 2^{33} leaves naively requires ≈ 144 petabytes, which they propose to amortize via an (unimplemented) time–space tradeoff backed by a database layer. They further note that their prototype is intended for academic evaluation rather than deployment, whereas Aegon provides production-grade code.

2.2 Asymptotic Comparison Cost

Figure 1 presents an asymptotic efficiency comparison of state-of-the-art approaches, including our proposed scheme, Aegon. Among all existing constructions, only VeRSA, Merkle^2, IronDict and Aegon (both instantiated with zk-KZH) achieve auditor computation and communication costs that are *sublinear* in the number of updates within an epoch. However, each of these prior schemes has limitations. VeRSA is not scalable: its server lookup time is linear in the dictionary size. Similarly, Merkle^2 relies on the signature-chain assumption, which we consider impractical for KT

Protocol	Lookup		History		Update Cost	
	Client	Server	Client	Server	Server	Auditor
SEEMless [Cha+19]	$\log n^*$	$\log n^*$	$(u + \log T) \log n$	$(u + \log T) \log n$	$b \cdot \log(n + b)^*$	$b \cdot \log(n + b)^*$
Parakeet [Mal+23]	$\log n^*$	$\log n^*$	$(u + \log T) \log n$	$(u + \log T) \log n$	$b \cdot \log(n + b)^*$	$b \cdot \log(n + b)^*$
OPTIKS [Len+24]	$u \log n^*$	$u \log n^*$	$u \log n$	$u \log n$	$b \cdot \log(n + b)^*$	$b \cdot \log(n + b)^*$
Merkle^2 [Hu+21]	$\log^2 n$	$\log^2 n$	$\log^2 n$	$\log^2 n$	$\log^2 n^*$	$\log n^*$
VeRSA-IVC [Tya+22]	1	$(n/m + \log m)^*$	u	$(n/m + \log m)^*$	b	$\log T$
CONIKS [Mel+15]	$\log n^*$	$\log n^*$	naive	naive	$b \cdot \log(N + b)^*$	$b \cdot \log(N + b)^*$
IronDict [Haf+26]	$\log N$	$\log^2 N$	$u \cdot \log N$	$u \cdot \log^2 N$	index assignment: N value update: $\log N \cdot b$	$\log N$
Aegon	$\log N$	$\log^2 N$	$u \cdot \log N$	$\log^2 N$	$\log N \cdot b$	1

b: the number of elements added per epoch
n: the number of items in the dictionary
N: maximum dictionary capacity
T: the number of epochs since the operation was last invoked
u: value's version
m: batch size (in case of VeRSA supporting batch lookups)
naive indicates that verifying history requires performing a lookup for each epoch.
Complexities with star $(*)$ indicate the average case.

Figure 1: Asymptotic efficiency comparison of the state-of-the-art transparent dictionaries. Both IronDict and Aegon are privacy-preserving variants, i.e., instantiated with the zero-knowledge variant of KZH-$\frac{1}{2}\log N$. In Aegon, the server's computation for historical values is independent of the version number, since lookups for old versions are precomputed and stored in the database. The auditor's communication and computation costs are constant, depending only on the (constant) number of *dictionary shards*.

deployments, and as previously mentioned, IronDict has scalability issues and high epoch latency. In contrast to previous work, Aegon is the only work to achieve *constant* auditor cost. Moreover, considering the server-side workload, Aegon is significantly lightweight: its per-epoch computation depends only on the number of updates occurring in that epoch. This allows Aegon to support short epoch durations in practice, without sacrificing scalability.

Remark 1. The auditor's cost in Aegon technically scales with the number of shards η rather than being strictly constant. In practice, however, η is a small deployment-time parameter independent of the dictionary size, the number of updates per epoch, and the number of elapsed epochs. For example, our planetary-scale deployment supports a dictionary of 4 billion entries using only $\eta = 128$ shards. We therefore treat the auditor cost as effectively constant throughout this paper.

3 System Model

We briefly describe the three roles in our system—namely the *server*, *client*, and *auditor*. We also assume existence of a *public bulletin board*. These roles follow the standard model used in prior work on transparent dictionaries and key transparency [Mel+15].

Bulletin Board. Like prior work, we assume the existence of a public bulletin board, e.g., a publicly accessible and append-only ledger that allows the server to publish arbitrary strings along with authentication tags for verification. The use of a bulletin board prevents *split-view attacks*, where a malicious server provides different clients with inconsistent states of the dictionary at the same epoch n. The bulletin board abstraction can be realised in various ways, e.g., a trusted party, a public blockchain or a gossip protocol [Syt+16; Mei+20; FHP26].

At the time of writing, WhatsApp implements this bulletin board by relying on Cloudflare as a trusted party. Concretely, the WhatsApp server in each epoch sends the tree root to Cloudflare, which then signs it. This signature serves as a publicly verifiable proof of publication on the bulletin board. In practice, this trust model can be strengthened by distributing trust across multiple parties, for example, via a multisignature scheme. In our implementation, to ensure compatibility with the existing system, we also model the bulletin board as a signature issued by a trusted entity.

Server. The main entity in our system is a *server* that provides access to a queryable dictionary (a label-value map) and periodically updates it (once per *epoch*) while maintaining some invariants. With each update, the server publishes a new commitment to the dictionary. We expect the server to provide verifiable answers to two types of queries:

- *Lookup:* What value is assigned to label ℓ in epoch n?
- *Consistency:* Has the value associated with label ℓ changed between epochs s_0 and s_1? More broadly, a consistency proof enables retrieval of historical values, allowing a client to query all previous versions of its value.

Performing lookups on the values assigned to a label ℓ at epochs s_0 and s_1 does not suffice as a consistency proof between s_0 and s_1. The server might attempt an *oscillation attack* [Mei+20] by changing the mapping for a targeted label at one epoch and then switching it back in the next epoch. Hence, a consistency proof must rule out changes in any epoch between s_0 and s_1. In some systems, the server commits to maintaining additional invariants. For example, a version invariant [Mel+15] attaches a version number v to each value and requires that it be incremented in any epoch in which the value changes. Given this invariant, checking that a value has the same version number in epochs s_0 and s_1 is equivalent to a consistency proof. A stronger invariant ensures that each value is append-only—that is, the dictionary retains all historical versions of a value, so that a lookup at epoch n reveals the entire version history up to that point. Other invariants are application-specific; for example, CONIKS [Mel+15] proposed marking some labels as *strict*, meaning any changes must be authorized by a signature. In our work, similar to IronDict, we don't adopt any such special invariants and instead build efficient consistency proofs directly.

Client. A set of *clients* (or users) are able to query the dictionary via the server. Typically, each client is represented by a label ℓ, which is a human-readable identifier such as a username, email address, or phone number.[4] The client will typically *monitor* this label by manually verifying any

[4]Later, we distinguish between human-readable labels and *indices* which are assigned to each user but not exposed at the UX level.

changes to the value mapped to their label, i.e., by querying historical values or proof of consistency. Clients will also request lookup proofs of other labels of interest, such as peers in a secure messaging system that they wish to communicate with. However, the clients *do not* monitor other labels over time. The assumption is that the label's owner will monitor and detect any unauthorized changes. There may be restrictions on which labels a user can look up; for example, rate limits or a requirement to be marked as contacts within the system, we leave these out of scope. Clients can also request updates to the value mapped to their label (for example, to change their public key after losing a device). Notably, we do not require cryptographic authentication (such as a signature) to update any mapping, unlike some prior work [Hu+21]. The downside of such assumptions is that they require users to maintain uninterrupted access to cryptographic keys over time. In practice, users often lose their devices or reinstall applications, thereby losing access to previous secret keys. Recovering from such scenarios inherently requires the server to update any label's value based on non-cryptographic authentication (e.g., passwords or 2FA).

Remark 2. In our standard threat model, we assume that clients monitor only their own labels and do not track other users' labels. However, in some settings this may not hold. For example, in the context of KT, more cautious clients can track their contacts' public keys by verifying that a contact's key has not changed since their last interaction, confirming that they are still communicating with the same identity. Such designs are orthogonal to the construction of the transparent dictionary and instead rely on the server's API and policy choices.

Auditor. *Auditors* monitor the server's updates and verify the correctness of a proof of invariance at each epoch. At a high level, this proof ensures that the transition from one epoch to the next preserves the desired invariants. The efficiency of auditing depends directly on the size and verification cost of invariance proofs. For example, consider the simple case where the server publishes all value updates directly on the bulletin board. In this setting, auditors must be *almost* as powerful as the server to recompute the updates and verify their correctness. However, requiring auditors to be computationally expensive means that most clients cannot audit themselves and must depend on a quorum of external auditors. Even though one honest auditor can reveal any misbehavior, it is still unclear what incentives exist to perform these costly audits. Instead, similar to IronDict, we aim for a *self-auditing* model in which even lightweight clients can perform auditing.

Fast-forwarding via recursive proofs refers to a special scenario in which auditing proofs are verified recursively using a recursive proof system—such as in incremental verifiable computation (IVC) [Val08]. In this setting, verifying the recursive proof for a given epoch automatically guarantees the correctness of all prior updates. This mechanism allows a new auditor, or one who has been offline for an extended period, to validate a single proof instead of checking proofs from multiple missed epochs. For an in-depth discussion of fast-forwarding, we refer the reader to Section 6 of IronDict [Haf+26].

3.1 Threat Model

Transparency systems are generally reactive rather than proactive. A malicious server may carry out various short-term attacks, but these will eventually be detected. The model assumes that the server has a public reputation to protect and thus behaves as a malicious-but-cautious attacker, avoiding attacks that are guaranteed to be uncovered. Among detectable attacks, some are provably detectable because they produce irrefutable cryptographic evidence. For example, any incorrect update that violates the required invariants produces a verifiable proof of misbehavior that auditors can detect. To ensure such provably detectable attacks are broadcast, we assume the presence of a

whistle-blowing mechanism. Throughout our analysis, we further assume that each epoch contains at least one honest auditor.

A second class of attacks is unprovable, such as modifying a user's public key without a legitimate request. These attacks are typically detectable by the affected users—assuming they monitor their own records—but cannot be proven cryptographically. This limitation arises whenever updates are authenticated through non-cryptographic means. For instance, if account recovery is done via passwords or SMS, the server can always claim that the user requested the update. Conversely, a user may falsely accuse the server of performing an unauthorized update. Such disputes must be resolved socially or legally; if enough reputable users publicly report misbehavior, this may count as convincing—though not cryptographically irrefutable—evidence.

Finally, the system still relies on the server for availability and privacy; for instance, the server must not reveal a user's public key to clients outside that user's contact list. We emphasize that privacy in a key transparency system is designed to limit what clients and auditors learn, not to protect against the server itself, which necessarily learns every label and value.

4 Preliminaries

Notation. We denote vectors using the arrow notation $\vec{w}$, and use w_i to refer to the i-th entry of the vector, indexing from 0. Let $\mathbb{F}$ be a finite field and $\mathbb{G}$ a group with scalars in $\mathbb{F}$, with additive notation. For $a \in \mathbb{F}$ and $\mathsf{G} \in \mathbb{G}$, the scalar multiplication of G by a is denoted as $a \times \mathsf{G}$. A function $f(x)$ is negligible if, for any polynomial $p(x)$, there exists a positive integer N such that for all $x > N$, we have $f(x) < \frac{1}{p(x)}$. When we say an event happens with overwhelming probability (abbreviated by $o.w.p$), we mean it occurs with a probability of $1-\epsilon(\lambda)$, where $\epsilon(\lambda)$ is a negligible function. We use $\mathbb{N}$ to denote the set of positive integers (natural numbers). For a non-negative integer $n \in \mathbb{N} \cup \{0\}$, we use $[n]$ to denote the set $\{0, 1, \ldots, n\}$. Finally, for any integers $n_1, n_2 \in \mathbb{N} \cup \{0\}$ with $n_1 < n_2$, we denote by $[n_1, n_2]$ and $[n_1, n_2)$ the sets $\{n_1, n_1+1, \ldots, n_2\}$ and $\{n_1, n_1+1, \ldots, n_2-1\}$, respectively.

Multilinear Polynomials. A multivariate polynomial is said to be *multilinear* if the individual degree of any variable X_i is at most 1. We use $\mathbb{F}_{\mu}^{\leq 1}[\vec{X}]$ to denote the space of all μ-variate multilinear polynomials. A multilinear polynomial $p \in \mathbb{F}_{\mu}^{\leq 1}[\vec{X}]$ of size $N = 2^{\mu}$ is uniquely defined by (or uniquely extends) its N evaluations on the Boolean hypercube.

General Dictionary. We denote by $\mathsf{D} : \{0,1\}^{\star} \to \{0,1\}^{\star}$ a *general dictionary* that maps arbitrary bit strings to arbitrary bit strings. The dictionary is defined as a set of label–value pairs $\{(\ell_i, \upsilon_i)\}_i$ with *unique* labels. When we write a dictionary as $\mathsf{D} : A \to B$, we mean that the dictionary's labels belong to the set A, and its values belong to the set B.

Modeling Dictionaries from Multilinear Polynomials. There is a specific class of dictionaries that admits a one-to-one correspondence with multilinear polynomials. Let $m, N \in \mathbb{N}$, and assume that N is a power of two. Consider a dictionary $\mathsf{D} : \{0,1\}^{\log_2 N} \to \mathbb{F} \setminus \{0\}$, represented as the set of label–value pairs $\{(\vec{x}_i, \upsilon_i)\}_{i \in [m]}$. One can model this dictionary as a multilinear polynomial $p \in \mathbb{F}_{\log_2 N}^{\leq 1}$ such that $p(\vec{x}_i) = \upsilon_i$ for each stored label–value pair, and $p(\vec{x}) = 0$ for all labels $\vec{x} \in \{0,1\}^{\log_2 N}$ that do not appear in the dictionary; that is,

$$p(\vec{X}) = \begin{cases} \upsilon_i & \text{if } \vec{X} = \vec{x}_i, \\ 0 & \text{if } \vec{X} \in \{0,1\}^{\log_2 N} \setminus \{\vec{x}_i\}_{i \in [m]}. \end{cases}$$

This representation is well-defined and unique, since a multilinear polynomial is uniquely determined by its evaluations over the Boolean hypercube. Finally, given a hash function $\mathcal{H} : \{0,1\}^{\star} \to \mathbb{F}$ that is both collision-resistant and preimage-resistant, a dictionary of the form $\mathsf{D} : \{0,1\}^{\log_2 N} \to \{0,1\}^{\star}$ can be equivalently represented by a dictionary $\mathsf{D} : \{0,1\}^{\log_2 N} \to \mathbb{F} \setminus \{0\}$. Importantly, for the security of this transformation, we require the field $\mathbb{F}$ to be sufficiently large (e.g., $|\mathbb{F}| > 2^{2\lambda}$ achieves λ bit security) to rule out birthday attacks.

Schwartz–Zippel Lemma. The *Schwartz–Zippel lemma* [Zip79; Sch80] states that if a polynomial evaluates to zero at a random point chosen from the domain, then with overwhelming probability the polynomial must be identically zero. More formally, let $f \in \mathbb{F}_{\mu}^{\leq d}$ be a non-zero polynomial with μ variables of degree at most d, then we have:

$$\Pr_{r \leftarrow \mathbb{F}^{\mu}} \left[f(r) = 0 \right] \;\leq\; \frac{d}{|\mathbb{F}|}.$$

In particular, when $|\mathbb{F}|$ is sufficiently large compared to d, a single random evaluation suffices to test whether f is the zero polynomial except with negligible error probability.

Polynomial Commitment Scheme (PCS). PCS is a cryptographic primitive that enables a prover to *commit* to a (multilinear) polynomial succinctly and later prove the correctness of its evaluation at some chosen points from the domain, without revealing the full polynomial. When a verifier requests an opening at a point $\vec{x}$, the prover returns both the claimed evaluation $y = p(\vec{x})$ and a proof π attesting to its correctness. The verifier can efficiently check that y is the correct value of the committed polynomial at $\vec{x}$ using the proof π. Finally, a PCS is said to be *zero-knowledge* if a commitment to a polynomial reveals no information about the underlying polynomial (e.g., it is statistically independent of it), and an opening proof at a point $\vec{x}$ reveals nothing beyond the evaluation $p(\vec{x})$. A formal definition of PCS and its required properties is provided in Appendix A.

4.1 KZH: PCS with Optimal Opening

Similar to IronDict, we instantiate the PCS in Aegon using KZH [Kad+25]. KZH is a family of multilinear PCSs built over a pairing-friendly elliptic curve. A distinguishing feature of KZH (and its zero-knowledge variant) is that it achieves sublinear opening time for the prover[5] and sublinear proof size and verification time for the verifier. KZH is parameterized by an integer k. In the KZH-k instantiation, both the proof size and the verification time are $O(k \cdot N^{1/k})$, where N is the size of the Boolean hypercube that the polynomial is defined on, and consequently, $\log_2 N$ is the number of variables of the multilinear polynomial. Throughout this work, we instantiate the scheme with $k = \frac{1}{2} \cdot \log_2 N$, which results in proof size and verification time of $O(\log N)$.

A detailed description of the KZH-k construction is provided in Appendix E. Throughout the paper, when we refer to KZH, we mean the KZH-$(\frac{1}{2} \cdot \log_2 N)$ variant, which we may denote simply by KZH for brevity. The security of KZH holds in the algebraic group model (AGM) [FKL18] under the (q_1, q_2)-*discrete logarithm* and the *setup-find-representation* assumptions.

Advantages of KZH for our scheme. Apart from sublinear proof size and verifier time, the KZH family has the following key advantages that make it particularly well-suited as the underlying PCS for Aegon:

- *Free opening at Boolean points.* Opening Boolean points (non-zero-knowledge variant) using KZH-k family is essentially *free*, meaning the server is only required to read k *arrays*[6] of size $N^{1/k}$ from memory. The term *free* here means that the server does not need to perform any computation to generate the proof; it only needs to read and return the relevant auxiliary data. This is analogous to Merkle trees, where the server does not need to perform any computation to generate the membership proof. In Aegon, the server is only required to open the polynomials representing the underlying dictionary at Boolean points. Therefore, the cost of openings at Boolean points is particularly important.
- *Homomorphic commitments and auxiliary input.* The KZH family is homomorphic, and its auxiliary input used for opening is homomorphic as well. Here, *homomorphism* means the following: let $(\mathcal{C}_1, \mathsf{aux}_1)$ and $(\mathcal{C}_2, \mathsf{aux}_2)$ be the commitment–auxiliary pairs for polynomials f_1 and f_2, respectively. For any scalar $\alpha \in \mathbb{F}$, the pair $(\alpha \times \mathcal{C}_1 + \mathcal{C}_2,\ \alpha \times \mathsf{aux}_1 + \mathsf{aux}_2)$ is a valid commitment and auxiliary input for the polynomial $\alpha \cdot f_1 + f_2$, where $\cdot$ denotes scalar field multiplication. Consequently, updating n evaluation points of a polynomial requires only $k \cdot n$ group operations to update both the commitment and the auxiliary data of KZH-k. Although Aegon does not

[5] Excluding the cost of evaluating the polynomial itself.

[6] These arrays can be implemented as contiguous memory blocks (e.g., flat slices or static arrays), rather than dynamically allocated structures like linked lists or hash maps. This makes memory access highly efficient in practice due to spatial locality and reduced overhead.

necessarily require homomorphism, homomorphism significantly improves the efficiency of our protocol (see Table 1).

- *Cheap zero-knowledge compiler.* IronDict demonstrates how to construct a zero-knowledge variant of KZH-k using existing zero-knowledge compilers [Bün+21] for homomorphic polynomial commitment schemes, while requiring only a sublinear amount of randomness. Concretely, achieving zero-knowledge for a single opening increases the prover's (server's) cost by $O(k^2 \cdot N^{1/k})$ additional group operations, which becomes $O(\log^2 N)$ when instantiated with $k = \frac{1}{2}\log_2 N$. This modification incurs *almost* no additional overhead for the verifier (client). We include more details on this compiler in Appendix E.1.

KZH Trusted Setup. One limitation of KZH is its requirement for a *trusted* SRS, similar to the powers-of-tau used in KZG [KZG10]. The SRS must be generated via a *trusted setup*, which introduces important security considerations. In practice, such setups are typically carried out using multi-party computation (MPC) ceremonies [BGM17], where multiple participants sequentially contribute randomness to the SRS. As long as *at least one participant acts honestly*, the resulting SRS remains secure. On the positive side, like KZG, the KZH setup is *updatable*. This means that an existing SRS can be refreshed by adding new randomness, without compromising security, i.e., the resulting SRS remains secure as long as *either* the previous SRS was generated honestly or at least one participant in the update contributed honestly. MPC ceremonies of powers-of-tau have been successfully employed by real-world systems such as Ethereum [Fou25]. A summary of current best practices of such ceremonies is provided in [WCB25].

4.2 Transparent Dictionary

We adopt the definition of a transparent dictionary from IronDict, with two important modifications. To provide context, Aegon models the dictionary as consisting of two mappings. The first mapping, index, assigns each user label (e.g., an email address) to a unique index—represented as a binary string in $\{0,1\}^{\log_2 N}$, where N denotes the capacity of the dictionary. This assignment is fixed and does not change over time. The second mapping value associates each user index with a corresponding value (e.g., a hash of the user's public key). To look up their value, a user first retrieves their index from their label via index, and then queries the dictionary to obtain the value associated with that index. This type of mapping is also employed in tree-based constructions that rely on a sparse Merkle tree, where the index is essentially derived from the hash of the label. However, due to the simplicity of the index derivation process, these constructions do not distinguish between two separate algorithms: one for retrieving the index and another for retrieving the corresponding value. Our definition of a transparent dictionary diverges from that of IronDict in the following ways.

- We explicitly separate index lookup from value-related operations, namely value lookup and value consistency. Concretely, to retrieve the value associated with a label ℓ, the client first performs an index lookup to obtain the index corresponding to ℓ, and then uses this index to retrieve the value stored at that position. The same decomposition applies to value consistency: the client first retrieves the index of ℓ and then invokes a value-consistency functionality parameterised by that index. While this two-step lookup is also present in IronDict construction, we make this separation explicit in the definition because it is required for our subsequent privacy definitions.

- We relax the server's index assignment model. In IronDict, the server assigns an index to each label and, at every epoch, provides a succinct proof that all previously assigned indices remain

unchanged; the auditors verify this proof. In contrast, we relax this requirement and allow the server to change indices maliciously. To compensate, we introduce additional functionalities, Server.IndexConsis and Client.VerifyIndexConsis, which—analogous to value consistency—enable clients to verify whether their assigned index has remained unchanged. For example, the client checks whether their index has remained consistent from the time they registered to the latest epoch.

Definition 1 (Transparent dictionary)**.** A *transparent dictionary protocol* is defined with respect to three parties: Server, Auditor, and Client, all of whom have access to a public bulletin board BB. Because of the existence of BB, we assume all parties know the current epoch number n. We assume that the server is stateful and stores the state of past epochs. Each party supports the following algorithms:

- $\mathsf{Setup}(\lambda, \mu) \to (\mathsf{key}_{\mathsf{Server}}; \mathsf{key}_{\mathsf{Auditor}}, \mathsf{key}_{\mathsf{Client}})$: Given a security parameter λ and log of maximum supported size μ, this algorithm respectively outputs participants' keys, namely: $\mathsf{key}_{\mathsf{Server}}$, $\mathsf{key}_{\mathsf{Auditor}}$ and $\mathsf{key}_{\mathsf{Client}}$. We assume algorithms corresponding to each party take their corresponding key as an implicit input; we omit it for brevity. If the setup requires secret randomness, we assume it is executed by a trusted party (e.g., implemented via a ceremony).

- $\mathsf{Server.Init}() \to (\mathsf{state}_0, \mathcal{C}_0)$: Initializes the dictionary state as state_0 and computes a succinct commitment $\mathcal{C}_0$ to this state. The commitment $\mathcal{C}_0$ is then published on the bulletin board BB.

- $\mathsf{Server.Update}(\mathsf{state}_n, \Delta) \to (\mathsf{state}_{n+1}, \mathcal{C}_{n+1}, \pi_{n+1})$: Updates the current state state_n using a set of changes Δ to obtain a new state state_{n+1}. It then computes and publishes a new commitment $\mathcal{C}_{n+1}$ to BB, along with an invariance proof π_{n+1}.

- $\mathsf{Server.LookupIndex}(\ell, \mathsf{state}_n) \to (\mathsf{index}, \pi_{\text{Index}})$: Given a label ℓ and epoch state state_n, returns the index index associated with ℓ in the dictionary, along with an index lookup proof π_{Index}. If state_n does not exist or ℓ is invalid, the algorithm returns $\perp$.

- $\mathsf{Client.VerifyIndex}(\ell, \mathcal{C}_n, \mathsf{index}, \pi_{\text{Index}}) \to 0/1$: Given a label ℓ, epoch commitment $\mathcal{C}_n$, index index, and the index lookup proof π_{Index}, this algorithm verifies the proof and returns 1 (accept) or 0 (reject).

- $\mathsf{Server.LookupValue}(\mathsf{index}, \mathsf{state}_n) \to (\upsilon, \pi_{\text{Val}})$: Given an index index and epoch state state_n, returns the value υ at index in dictionary state state_n, along with a proof π_{Val}. If state_n does not exist or the index is invalid, the algorithm returns $\perp$.

- $\mathsf{Client.VerifyValue}(\mathsf{index}, \mathcal{C}_n, \upsilon, \pi_{\text{Val}}) \to 0/1$: Given an index index, epoch commitment $\mathcal{C}_n$, value υ, and a lookup proof π_{Val}, this algorithm verifies the proof and returns 1 (accept) or 0 (reject).

- $\mathsf{Server.IndexConsis}(\ell, \mathsf{state}_{s_0}, \mathsf{state}_{s_1}) \to \pi_{\text{IndexConsis}}$: Given a label ℓ and two states state_{s_0} and state_{s_1} corresponding to epochs s_0 and s_1 with $s_0 < s_1$, the server returns a proof $\pi_{\text{IndexConsis}}$ attesting that the index associated with ℓ remained unchanged throughout epochs s_0 to s_1. If the index changed during this interval or ℓ is invalid, it returns $\perp$.

- $\mathsf{Client.VerifyIndexConsis}(\ell, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\text{IndexConsis}}) \to 0/1$: Client verifies validity of a consistency proof $\pi_{\text{IndexConsis}}$ for the index corresponding to label ℓ between epochs s_0 and s_1, returning 1 if the proof is accepted, and 0 otherwise.

- $\mathsf{Server.ValueConsis}(\mathsf{index}, \mathsf{state}_{s_0}, \mathsf{state}_{s_1}) \rightarrow \pi_{\text{ValueConsis}}$: Given an index index and two states state_{s_0} and state_{s_1} corresponding to epochs s_0 and s_1 with $s_0 < s_1$, the server returns a proof $\pi_{\text{ValueConsis}}$ attesting that the value of index remained unchanged throughout epochs s_0 to s_1. If the value changed during this interval or index is invalid, it returns $\perp$.

- $\mathsf{Client.VerifyValueConsis}(\mathsf{index}, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\text{ValueConsis}}) \rightarrow 0/1$: Client verifies validity of a consistency proof $\pi_{\text{ValueConsis}}$ for index between epochs s_0 and s_1, returning 1 if the proof is accepted, and 0 otherwise.

- $\mathsf{Auditor.VerifyInvariance}(\mathcal{C}_n, \mathcal{C}_{n+1}, \pi_{n+1}) \rightarrow 0/1$: Given commitments $\mathcal{C}_n$ and $\mathcal{C}_{n+1}$ for epochs n and $n+1$ along with proof of invariance π_{n+1}, verifies its validity and returns 1 (accept) or 0 (reject).

Security definition. A transparent dictionary is secure if it is *complete* and *sound*.

- Completeness ensures that when the setup is performed honestly and the server behaves correctly, the system behaves as expected. In particular, consistency correctness guarantees that if a client requests an index (value) consistency between epochs s_0 and s_1 for an index (value) that has not changed, then running $\mathsf{Client.VerifyIndexConsis}$ ($\mathsf{Client.VerifyValueConsis}$) on the proof will lead to acceptance, and any query for the index (value) within the interval $[s_0, s_1]$ returns the same result. Similarly, audit correctness ensures that at each epoch, an honest auditor running $\mathsf{Auditor.VerifyInvariance}$, outputs 1.

- Soundness, on the other hand, ensures security except with negligible probability. Index (value) lookup soundness, also known as binding, guarantees that given a state commitment $\mathcal{C}_n$ and a label (index), it is computationally infeasible for the server to produce two distinct indices (values) with valid proofs that both pass verification under $\mathsf{Client.VerifyIndex}$ ($\mathsf{Client.VerifyValue}$). Consistency soundness ensures that if at least one honest auditor accepts the proof for each epoch, then for any two epochs s_0 and s_1, the server cannot generate a proof of consistency for an index (or value) that changes during the interval $[s_0, s_1]$ that will be accepted by the client.

Formal definitions of completeness and soundness are provided in Appendix B.

Comparison with the Definition of SEEMless. To compare our definition with SEEMless and its variants such as Parakeet and OPTIKS, we replace the *historical value query* with a *value consistency query*. As we show below, any historical value query can be expressed as a combination of lookup queries and consistency proofs.

Remark 3 (Equivalence of Historical Values with Value Consistency)**.** One can see that value consistency is equivalent to retrieving historical values, a concept primarily used in tree-based approaches. Consider a label with its historical values $\{(n_i, v_i)\}_{i \in [t]}$, where n_i denotes the epoch at which the value changes to v_i. Given the historical values, it is straightforward to verify whether the value has remained unchanged between any two desired epochs. Conversely, if a value has changed at epochs $n_0, n_1, \ldots, n_t$, the server can prove the full value history by showing that the value did not change within each interval $[n_i, n_{i+1})$ and by revealing the value at each epoch n_i. This requires a linear number of lookups and consistency checks proportional to the number of value changes.

5 Building Blocks

In this section, we introduce the building blocks underlying Aegon: a two-polynomial representation of the dictionary, an open-addressing scheme for assigning indices to labels, and a randomized-polynomial technique to prove a number of polynomials agree on an evaluation point.

5.1 Modeling a Dictionary from Two Polynomials

In Aegon, we model a general dictionary of the form $\mathsf{D} : \{0,1\}^\star \to \{0,1\}^\star$ using a pair of multilinear polynomials. As a first step, and as described in Section 4, one can model a dictionary of the form $\mathsf{D} : \{0,1\}^{\log_2 N} \to \mathbb{F} \setminus \{0\}$ via a multilinear polynomial. Provided that the underlying field $\mathbb{F}$ is sufficiently large, this dictionary is equivalent to a dictionary of the form $\mathsf{D} : \{0,1\}^{\log_2 N} \to \{0,1\}^\star$ by incorporating a collision-resistant and preimage-resistant hash function $\mathcal{H}^{\mathbb{F}} : \{0,1\}^\star \to \mathbb{F}$. However, this representation alone is insufficient to capture a general dictionary of the form $\mathsf{D} : \{0,1\}^\star \to \{0,1\}^\star$. The approach of using a hash function to map arbitrary strings into the Boolean hypercube $\{0,1\}^{\log_2 N}$ fails in this context. Here, N denotes the dictionary capacity, and even setting N to 100 billion is far too small to achieve collision resistance. Consequently, no collision-resistant hash function with range $\{0,1\}^{\log_2 N}$ exists. To support arbitrary label–value mappings of bit strings, we introduce two auxiliary dictionaries index and value as defined below, such that their composition (denoted by $\circ$) models a general bit string to bit string dictionary.

$$\mathsf{index} : \{0,1\}^{\log_2 N} \to \{0,1\}^\star,\ \mathsf{value} : \{0,1\}^{\log_2 N} \to \{0,1\}^\star$$

$$\implies \mathsf{value} \circ \mathsf{index}^{(-1)} : \{0,1\}^\star \to \{0,1\}^\star.$$

Importantly, both the index and value dictionaries can each be modeled independently by a multilinear polynomial, so the whole dictionary is represented using two such polynomials. The subtlety is that we use the *inverse* mapping $\mathsf{index}^{(-1)} : \{0,1\}^\star \to \{0,1\}^{\log_2 N}$, which maps each label back to its index. For this inverse to be well-defined, every label must be assigned a unique index; that is, no two distinct labels may share the same index. We refer to this requirement as the *invertibility* of index.

To summarize how a lookup works in the dictionary: each label ℓ (e.g., a phone number) is assigned a *unique* Boolean index $\vec{x} \in \{0,1\}^{\log_2 N}$, so that $\mathsf{index}(\vec{x}) = \ell$ (or, equivalently, a hash of ℓ). The value associated with ℓ is stored at the same index in the value dictionary, giving $\mathsf{value}(\vec{x}) = v$, where v may be a public key, its hash, or other related data.

Because a dictionary of the form $\mathsf{D} : \{0,1\}^{\log_2 N} \to \{0,1\}^\star$ corresponds uniquely to its multilinear representation, we use index and value to refer interchangeably to the dictionaries themselves or their polynomial representations. Modeling a general dictionary using two multilinear polynomials $(\mathsf{index}, \mathsf{value})$ has a significant advantage: it allows us to leverage a multilinear polynomial commitment scheme to commit to the dictionary and use its algebraic properties, e.g., homomorphism or selectively opening the underlying polynomials at points of interest. Concretely, at every epoch, the server commits to the polynomials $(\mathsf{index}, \mathsf{value})$ as $(\mathrm{com}(\mathsf{index}), \mathrm{com}(\mathsf{value}))$ and publishes these commitments to the bulletin board. Now, when a user queries the value associated with a label ℓ, assuming the client already knows the corresponding index $\vec{x}$, the server only needs to open the commitment $\mathcal{C}_v$ at point $\vec{x}$ to reveal $\mathsf{value}(\vec{x})$. Retrieving the index corresponding to a label is more involved, and we will elaborate on this later in this section.

This design of a general dictionary using two auxiliary dictionaries raises two key design questions, as outlined below, which determine the structure of the invariance proofs and the efficiency of the parties.

1. *Index assignment:* How is a unique index assigned to each label, and how does the server prove that no label is mapped to multiple indices?

2. *Value consistency:* How does the server prove to the client that their values have remained consistent across epochs?

5.2 Index Assignment

Our index assignment algorithm in Aegon follows IronDict, using an open addressing technique. The key difference lies in how we prove to clients that their index is unique and has not been maliciously modified: Aegon replaces IronDict's globally expensive proof with locally cheap, per-client checks, in which each client independently verifies that its own index has remained consistent. We first review the open addressing technique and how labels are assigned an index.

As mentioned earlier, the index domain $\{0,1\}^{\log_2 N}$ is relatively small given that N is the dictionary capacity. Consequently, there does not exist a collision-resistant hash function that maps arbitrary-length binary strings to $\{0,1\}^{\log_2 N}$. This raises an important question: *how can the server verifiably assign unique indices to labels?* In other words, the server must prove that there is exactly one valid index per label. We address this problem using an *open addressing* technique [con25]. Assume we have two hash functions $\mathcal{H}^{\text{bits}} : \{0,1\}^\star \to \{0,1\}^{\log_2 N}$ and $\mathcal{H}^{\mathbb{F}} : \{0,1\}^\star \to \mathbb{F} \setminus \{0\}$, such that $\mathcal{H}^{\text{bits}}$ is *not* collision-resistant while $\mathcal{H}^{\mathbb{F}}$ is both collision-resistant and preimage-resistant. In practice, we use VRFs to instantiate both $\mathcal{H}^{\mathbb{F}}$ and $\mathcal{H}^{\text{bits}}$. To assign a label ℓ an index, the server proceeds as follows: (i) Compute $\vec{x}_0 := \mathcal{H}^{\text{bits}}(0,\ell)$. (ii) If this index is unoccupied, i.e., $\mathsf{index}(\vec{x}_0) = 0$, then assign it to label ℓ by setting $\mathsf{index}(\vec{x}_0) = \mathcal{H}^{\mathbb{F}}(\ell)$. (iii) Otherwise, compute $\vec{x}_1 := \mathcal{H}^{\text{bits}}(1,\ell)$ and repeat the same procedure. The formal algorithm for index assignment is provided in Figure 2. There are two critical considerations for the efficiency and security of this approach.

- *Overprovisioning.* Open addressing generally requires *overprovisioning* the dictionary capacity. Without such slack, as the load factor approaches 1, locating an empty slot requires an increasing number of hash probes. In Aegon, we set the dictionary capacity to be four times the expected number of stored elements. With this load factor of $\alpha^{-1} = 1/4$, the expected number of probes needed to find an empty slot is $(1 - \frac{1}{\alpha})^{-1} = \frac{4}{3}$.

- *Use of VRF to instantiate $\mathcal{H}^{bits}$ and $\mathcal{H}^{\mathbb{F}}$.* The other consideration is that, in practice, $\mathcal{H}^{\text{bits}}$ and $\mathcal{H}^{\mathbb{F}}$ are instantiated using a VRF. This prevents an adversary from predicting the distribution of data in the dictionary by locally evaluating $\mathcal{H}^{\text{bits}}$ and $\mathcal{H}^{\mathbb{F}}$. Moreover, due to using open addressing, this also mitigates attacks where an adversary chooses a label ℓ such that the assigned index $\mathcal{H}^{\text{bits}}(\mathsf{ctr}_0,\ell)$ has a high counter ctr_0. For further details, we refer the reader to Section 4 of IronDict.

Another crucial requirement for the index dictionary is that the server must prove that each label is assigned a unique index. If a server could make clients accept different indices $\vec{x}_1$ and $\vec{x}_2$ for the same label ℓ, it could assign different values to $\mathsf{value}(\vec{x}_1)$ and $\mathsf{value}(\vec{x}_2)$, effectively opening multiple, inconsistent values for the same label. The open addressing technique for index assignment provides a natural way to enforce uniqueness. Suppose a user is assigned an index $\vec{x}_{\mathsf{ctr}_0} := \mathcal{H}^{\text{bits}}(\mathsf{ctr}_0,\ell)$, and let $\vec{x}_{\mathsf{ctr}} := \mathcal{H}^{\text{bits}}(\mathsf{ctr},\ell)$ for $\mathsf{ctr} < \mathsf{ctr}_0$. As long as the evaluations $\mathsf{index}(\vec{x}_{\mathsf{ctr}})$ remain unchanged for all $\mathsf{ctr} \in [\mathsf{ctr}_0]$, the index $\vec{x}_{\mathsf{ctr}_0}$ is guaranteed to be the unique index that the client accepts for label ℓ. Importantly, according to the index assignment algorithm, $\mathsf{index}(\vec{x}_{\mathsf{ctr}}) \neq 0$ is required for all $\mathsf{ctr} \leq \mathsf{ctr}_0$. In Aegon, the server proves to each client individually that their index has not been modified, and avoids an expensive global proof.

(1) Set $\mathsf{ctr} := 0$, now given label ℓ, the server computes $\vec{x} := \mathcal{H}^{\text{bits}}(\mathsf{ctr}, \ell)$ and checks if the index $\vec{x}$ is not already assigned, in other words, $\mathsf{index}(\vec{x}) = 0$. If so, it sets $\mathsf{index}(\vec{x}) = \mathcal{H}^{\mathbb{F}}(\ell)$ and returns $\vec{x}$ as the assigned index.

(2) If $\mathsf{index}(\vec{x}) \neq 0$, the server updates $\mathsf{ctr} := \mathsf{ctr} + 1$ and then recompute $\vec{x} := \mathcal{H}^{\text{bits}}(\mathsf{ctr}, \ell)$ and repeats the procedure. This continues until an unassigned index is found.

Figure 2: Index assignment algorithm by the server

5.3 Randomized Polynomials for Cross-Epoch Equality

As a central building block, we use a random-linear-combination technique for proving that a sequence of multilinear polynomials all agree at a single point [Haf+26]. IronDict introduces this technique to ensure value consistency. In Aegon, we extend this idea further by additionally proving the correctness of the label's index via this property.

Suppose a prover has committed to a sequence of multilinear polynomials $p_1, p_2, \ldots, p_n$ and wishes to convince a verifier that they all agree at a point $\vec{x}$, i.e., $p_1(\vec{x}) = p_2(\vec{x}) = \cdots = p_n(\vec{x})$. The naive approach—opening every p_i at $\vec{x}$—requires n PCS openings. Instead, the prover defines a sequence of *randomized polynomials* $\mathsf{rand}_1, \ldots, \mathsf{rand}_n$ recursively as

$$\mathsf{rand}_1 := 0, \qquad \mathsf{rand}_i := \mathsf{rand}_{i-1} + r_i \cdot (p_i - p_{i-1}) \quad \text{for } i \geq 2,$$

where each $r_i \in \mathbb{F}$ is sampled uniformly at random (in practice via Fiat–Shamir [FS87], by hashing r_{i-1} together with the commitment to p_i). Equivalently,

$$\mathsf{rand}_n = \sum_{k=2}^{n} r_k \cdot (p_k - p_{k-1}).$$

To check agreement at $\vec{x}$, it now suffices to verify a single equality: $\mathsf{rand}_1(\vec{x}) = \mathsf{rand}_n(\vec{x})$, requiring only two PCS openings regardless of n. Soundness follows from the Schwartz–Zippel lemma: if any $p_i(\vec{x}) \neq p_{i-1}(\vec{x})$, the equality holds only with negligible probability over the choice of $\{r_i\}$. Importantly, for this reduction to be sound, the verifier must be assured that each rand_i is correctly derived from rand_{i-1} and p_i. In the key transparency setting, this task is delegated to the auditor, who at every epoch recomputes the Fiat–Shamir challenge r_n and checks the homomorphic relation

$$\text{com}(\mathsf{rand}_n) = \text{com}(\mathsf{rand}_{n-1}) + r_n \cdot \big(\text{com}(p_n) - \text{com}(p_{n-1})\big). \tag{1}$$

As long as the underlying PCS is homomorphic, this check costs only a constant number of hashes and homomorphic operations per epoch. Finally, we emphasize that the technique does not fundamentally rely on homomorphism: the same identity can be verified non-homomorphically via fingerprinting (see Remark 4).

6 Aegon: A Scalable Transparent Dictionary

In this section, we present Aegon, a transparent dictionary built generically over a multilinear polynomial commitment scheme. We begin in Section 6.1 with the technique that underpins Aegon's constant auditor cost: we delegate index-uniqueness verification to clients. The key observation is that index immutability, like value immutability, is a per-client property: each client only cares that its own index has not changed. Since clients must already monitor their own values, delegating index monitoring to them adds no new assumption to the threat model, yet it collapses what would otherwise be a per-epoch global proof into a local check of constant cost per client.

We then present our construction in Section 6.2 as a generic protocol over any multilinear PCS. While Aegon does not inherently require the PCS to be homomorphic, such a property significantly improves the efficiency of the scheme.

In Section 6.3, we turn to a formal treatment of privacy. We give the first formal definition of privacy for transparent dictionaries in the UC framework, and we prove that Aegon satisfies this definition when instantiated with a zero-knowledge PCS.

Finally, in Section 6.4 we discuss the concrete instantiation of Aegon with a PCS. Following prior work IronDict, we instantiate Aegon with the KZH polynomial commitment. We emphasize, however, that Aegon is generic over the choice of PCS, and identifying alternatives to KZH—for instance, schemes that avoid a trusted setup or that are post-quantum secure—remains an interesting direction for independent study.

6.1 Index Uniqueness

Suppose a label ℓ is assigned an index $\vec{x}_{\mathsf{ctr}_0}$ such that:

$$\vec{x}_{\mathsf{ctr}_0} := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}_0, \ell) \text{ and for all } \mathsf{ctr} < \mathsf{ctr}_0 \text{ define: } \vec{x}_{\mathsf{ctr}} := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}, \ell)$$

According to the index assignment algorithm, $\mathsf{index}(\vec{x}_{\mathsf{ctr}_0}) = \mathcal{H}^{\mathbb{F}}(\ell)$, while all earlier evaluations $\mathsf{index}(\vec{x}_{\mathsf{ctr}})$ for $\mathsf{ctr} < \mathsf{ctr}_0$ are non-zero. In IronDict, the server proves that all non-zero evaluation points remain unchanged, ensuring that $\mathsf{index}(\vec{x}_{\mathsf{ctr}_0})$ remains the unique index corresponding to label ℓ. In our approach, we shift this check to the client: the owner of the label ℓ locally verifies that the evaluation points

$$\mathsf{index}(\vec{x}_0), \mathsf{index}(\vec{x}_1), \ldots, \mathsf{index}(\vec{x}_{\mathsf{ctr}_0})$$

have not changed from the epoch in which the label was registered up to the current epoch. To prove index consistency to clients, we use the same randomization technique as for the value polynomials. Define the incremental change of the index polynomial at epoch n as $\Delta_n^{(\mathrm{I})} := \mathsf{index}_n - \mathsf{index}_{n-1}$. At each epoch n, the server derives a random value $r_n^{(\mathrm{I})}$ non-interactively via the Fiat–Shamir heuristic and constructs the randomized index polynomial recursively:

$$\mathsf{rand}_n^{(\mathrm{I})} := \mathsf{rand}_{n-1}^{(\mathrm{I})} + r_n^{(\mathrm{I})} \cdot \Delta_n^{(\mathrm{I})} \implies \mathsf{rand}_n^{(\mathrm{I})} = \sum_{k=1}^{n} r_k^{(\mathrm{I})} \cdot \Delta_k^{(\mathrm{I})}.$$

As in the value consistency argument, if two randomized index polynomials agree at a point $\vec{x}$, then with overwhelming probability all intermediate index polynomials also agree at that point:

$$\mathsf{rand}_{s_0}^{(\mathrm{I})}(\vec{x}) = \mathsf{rand}_{s_1}^{(\mathrm{I})}(\vec{x}) \xRightarrow{\text{o.w.p.}}$$

$$\mathsf{index}_{s_0}(\vec{x}) = \mathsf{index}_{s_0+1}(\vec{x}) = \cdots = \mathsf{index}_{s_1}(\vec{x}).$$

Now consider a client with label ℓ who registered at epoch s_0 and the system is currently at epoch s_1. Let its current index be $\vec{x}_{\mathsf{ctr}_0} := \mathcal{H}^{\text{bits}}(\mathsf{ctr}_0, \ell)$, and for all $\mathsf{ctr} \in [\mathsf{ctr}_0 - 1]$ define $\vec{x}_{\mathsf{ctr}} := \mathcal{H}^{\text{bits}}(\mathsf{ctr}, \ell)$. For the client to verify that its index has remained unchanged from epoch s_0 to s_1, it must check the following:

$$\forall\, \mathsf{ctr} \in [\mathsf{ctr}_0] : \mathsf{rand}^{(\mathrm{I})}_{s_0}(\vec{x}_{\mathsf{ctr}}) = \mathsf{rand}^{(\mathrm{I})}_{s_1}(\vec{x}_{\mathsf{ctr}}).$$

This consistency check requires verifying $2 \cdot (\mathsf{ctr}_0 + 1)$ PCS openings. With an overprovisioning factor of 4, the expected number of PCS openings is $\frac{8}{3}$, which remains concretely efficient.

6.2 Construction

In this section, we present our construction of a transparent dictionary, summarized in Figure 3 and Figure 4. The construction for a dictionary of size $N := 2^\mu$, uses multilinear polynomials with μ variables defined over a field $\mathbb{F}$ such that $2^{2\lambda} < |\mathbb{F}|$, where λ is the bit security parameter. Aegon is generic with respect to the underlying PCS. We assume that all parties have access to a random oracle $\mathcal{O}$. As described earlier, our construction relies on two hash functions: $\mathcal{H}^{\mathbb{F}} : \{0,1\}^\star \to \mathbb{F}$ and $\mathcal{H}^{\text{bits}} : \{0,1\}^\star \to \{0,1\}^{\log_2 N}$, such that $\mathcal{H}^{\mathbb{F}}$ is preimage-resistant and collision-resistant. In practice, we instantiate them using a VRF. In this case, the client cannot evaluate $\mathcal{H}^{\text{bits}}$ and $\mathcal{H}^{\mathbb{F}}$ locally; instead, the server provides the evaluation of $\mathcal{H}^{\text{bits}}$ (highlighted red in Figure 4) together with a proof of correctness generated by the VRF, which we omit for simplicity.

Theorem 1. Let PCS be a secure polynomial commitment scheme for multilinear polynomials with μ variables over a field $\mathbb{F}$, where $|\mathbb{F}| > 2^{2\lambda}$. Let $\mathcal{H}^{\mathbb{F}} : \{0,1\}^\star \to \mathbb{F}$ be a collision-resistant and preimage-resistant hash function, and let $\mathcal{H}^{\text{bits}} : \{0,1\}^\star \to \{0,1\}^{\log_2 N}$ be an efficiently computable mapping that is not required to be collision-resistant. Finally, let $\mathcal{O} : (\mathbb{C} \cup \mathbb{F}^\mu)^\star \to \mathbb{F}$ be a random oracle that takes as input sequences of polynomial commitments (from the commitment space $\mathbb{C}$) and elements of $\mathbb{F}^\mu$, and outputs a field element. Then, the transparent dictionary construction described in Figure 3 and Figure 4 is complete and sound. Additionally, if PCS is zero-knowledge, the resulting transparent dictionary satisfies privacy.

Proof. We defer the security proof to Appendix C, and the privacy proof to Appendix D. □

Note that the construction of Aegon could be initialized using a homomorphic vector commitment scheme with zero-knowledge openings, rather than a multilinear polynomial commitment scheme. In particular, KZH-k, equipped with zero-knowledge openings, yields a vector commitment scheme with efficient opening time while preserving zero-knowledge. Nevertheless, we describe our construction using multilinear polynomials and a PCS to keep the scheme generic. Polynomial representations provide greater expressive power; for example, homomorphic relations can be verified without relying on commitment homomorphism by opening the committed polynomial at randomly chosen points. For this reason, and to maintain generality, we formulate the construction over multilinear polynomials and polynomial commitment schemes.

Remark 4 (Checking Homomorphic Relations on Non-Homomorphic Commitments.)**.** While homomorphic (i.e., degree one) relations can be verified directly using a homomorphic PCS, they can also be checked non-homomorphically via a *fingerprinting* approach, namely by evaluating both sides at a random point. The security of this method follows from the Schwartz–Zippel lemma [Zip79; Sch80]. Consider verifying a degree-one homomorphic relation of the form $G(f_1, \ldots, f_n) = f$, where $\{f_i\}_{i\in[1,n]}$ is a set of polynomials, f is another polynomial, and G specifies a degree-one relation over the entire set $\{f_i\}_{i\in[1,n]}$. Rather than checking this relation homomorphically, it suffices to verify it at a single random point $\vec{r}$. Concretely, the verifier is given commitments to all polynomials in the

- $\mathsf{Setup}(\lambda, \mu) \to (\mathsf{key}_{\mathsf{Server}}; \mathsf{key}_{\mathsf{Auditor}}, \mathsf{key}_{\mathsf{Client}})$: Given a security parameter λ and a parameter μ defining a capacity $N := 2^{\mu}$, run $\mathsf{Setup}_{\mathsf{PCS}}(\lambda, \mu)$ which outputs $(\mathsf{vk}, \mathsf{pk})$, i.e., the verifier key and the prover key. Set $\mathsf{key}_{\mathsf{Server}} := \mathsf{pk}$, $\mathsf{key}_{\mathsf{Auditor}} := \mathsf{vk}$, and $\mathsf{key}_{\mathsf{Client}} := \mathsf{vk}$.

- $\mathsf{Server.Init}(\mu) \to (\mathsf{state}_0, \mathcal{C}_0)$:
  - Initialise zero multilinear polynomials index_0, value_0, $\mathsf{rand}_0^{(\mathrm{I})}$, and $\mathsf{rand}_0^{(\mathrm{V})}$ of size N.
  - Compute commitments to these polynomials, forming $\mathcal{C}_0$.
  - Set $\mathsf{state}_0 = (\mathsf{index}_0, \mathsf{value}_0, \mathsf{rand}_0^{(\mathrm{I})}, \mathsf{rand}_0^{(\mathrm{V})})$ and output $(\mathsf{state}_0, \mathcal{C}_0)$.

- $\mathsf{Server.Update}(\mathsf{state}_n, \Delta) \to (\mathsf{state}_{n+1}, \mathcal{C}_{n+1}, \pi_{n+1})$: Parse Δ as $(\Delta_{\mathrm{index}}, \Delta_{\mathrm{value}})$ and perform updates as follows:
  - *Index dictionary:* Parse $\Delta_{\mathrm{index}} = \{\ell_i\}_i$ and ensure that all labels ℓ_i are unassigned.
    1. Assign indices $\vec{x}_i$ according to Figure 2, and compute com(index_{n+1}), such that:
    $$\Delta^{(\mathrm{I})}(\vec{x}) = \begin{cases} \mathcal{H}^{\mathbb{F}}(\ell_i) & \text{if } \vec{x} = \vec{x}_i, \\ 0 & \text{otherwise.} \end{cases} \implies \mathsf{index}_{n+1} = \mathsf{index}_n + \Delta^{(\mathrm{I})}$$
    2. Derive randomness $r_n^{(\mathrm{I})} \leftarrow \mathcal{O}(r_{n-1}^{(\mathrm{I})}, \mathrm{com}(\mathsf{index}_{n+1}))$ and compute com($\mathsf{rand}_{n+1}^{(\mathrm{I})}$), such that:
    $$\mathsf{rand}_{n+1}^{(\mathrm{I})} = \mathsf{rand}_n^{(\mathrm{I})} + r_n^{(\mathrm{I})} \cdot \Delta^{(\mathrm{I})}$$
  - *Value dictionary:* Parse $\Delta_{\mathrm{value}} = \{(\ell_i, \upsilon_i)\}_i$ and retrieve the associated indices $\vec{x}_i$.
    1. Compute com(value_{n+1}) such that:
    $$\Delta^{(\mathrm{V})}(\vec{x}) = \begin{cases} \upsilon_i & \text{if } \vec{x} = \vec{x}_i, \\ 0 & \text{otherwise.} \end{cases} \implies \mathsf{value}_{n+1} = \mathsf{value}_n + \Delta^{(\mathrm{V})}$$
    2. Derive randomness $r_n^{(\mathrm{V})} \leftarrow \mathcal{O}(r_{n-1}^{(\mathrm{V})}, \mathrm{com}(\mathsf{value}_{n+1}))$ and compute com($\mathsf{rand}_{n+1}^{(\mathrm{V})}$) such that:
    $$\mathsf{rand}_{n+1}^{(\mathrm{V})} = \mathsf{rand}_n^{(\mathrm{V})} + r_n^{(\mathrm{V})} \cdot \Delta^{(\mathrm{V})}$$

  The algorithm outputs the updated state state_{n+1} and its corresponding commitment $\mathcal{C}_{n+1}$. If the PCS is homomorphic, computing new commitments (denoted by blue) will be done homomorphically, hence the proof π_{n+1} is empty (i.e., $\pi_{n+1} = \perp$). Otherwise, π_{n+1} consists of a non-homomorphic proof attesting to the correctness of the commitments to $\mathsf{rand}_{n+1}^{(\mathrm{I})}$ and $\mathsf{rand}_{n+1}^{(\mathrm{V})}$. We omit the details, as the procedure follows directly from the standard fingerprinting argument (see Remark 4).

Figure 3: Aegon setup, init, and update functionality

- $\mathsf{Server.LookupIndex}(\ell, \mathsf{state}_n) \to (\vec{x}, \pi_{\mathrm{Index}})$:
  - Determine the label counter $\mathsf{ctr}_0 \in \mathbb{N} \cup \{0\}$ associated with ℓ.
  - For each $\mathsf{ctr} \in [\mathsf{ctr}_0]$, define $\vec{x}_{\mathsf{ctr}} := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}, \ell)$ and compute a PCS opening π_{ctr} attesting to the evaluation $\mathsf{index}_n(\vec{x}_{\mathsf{ctr}})$.
  - Output $\pi_{\mathsf{index}} := (\mathsf{ctr}_0, \{\pi_{\mathsf{ctr}}\}_{\mathsf{ctr} \in [\mathsf{ctr}_0]})$.
- $\mathsf{Client.VerifyIndex}(\ell, \mathcal{C}_n, \vec{x}, \pi_{\mathrm{Index}}) \to 0/1$: The client wishes to verify that the index $\vec{x}$ corresponds to the label ℓ.
  - Parse $\pi_{\mathsf{Index}} = (\mathsf{ctr}_0, \{\pi_{\mathsf{ctr}}\}_{\mathsf{ctr} \in [\mathsf{ctr}_0]})$. For each $\mathsf{ctr} \in [\mathsf{ctr}_0]$, recompute $\vec{x}_{\mathsf{ctr}} := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}, \ell)$ and verify the corresponding PCS opening π_{ctr}.
  - Check $\forall\, \mathsf{ctr} \in [\mathsf{ctr}_0 - 1] : \mathsf{index}_n(\vec{x}_{\mathsf{ctr}}) \notin \{0, \mathcal{H}^{\mathbb{F}}(\ell)\}$, $\mathsf{index}_n(\vec{x}_{\mathsf{ctr}_0}) = \mathcal{H}^{\mathbb{F}}(\ell)$, and $\vec{x} = \vec{x}_{\mathsf{ctr}_0}$.
- $\mathsf{Server.LookupValue}(\vec{x}, \mathsf{state}_n) \to (\upsilon, \pi_{\mathrm{Val}})$: Compute a PCS opening π_{Val} attesting to the evaluation $\mathsf{value}_n(\vec{x}) = \upsilon$, and output π_{Val}.
- $\mathsf{Client.VerifyValue}(\vec{x}, \mathcal{C}_n, \upsilon, \pi_{\mathrm{Val}}) \to 0/1$: Verify the PCS proof π_{Val}, attesting $\mathsf{value}_n(\vec{x}) = \upsilon$.
- $\mathsf{Server.IndexConsis}(\ell, \mathsf{state}_{s_0}, \mathsf{state}_{s_1}) \to \pi_{\mathrm{IndexConsis}}$: Run $\pi_{\mathrm{Index}} \leftarrow \mathsf{Server.LookupIndex}(\ell, \mathsf{state}_{s_1})$ to retrieve the set of indices $\{\vec{x}_{\mathsf{ctr}}\}_{\mathsf{ctr} \in [\mathsf{ctr}_0]}$. Then, for each $\mathsf{ctr} \in [\mathsf{ctr}_0]$, compute the PCS openings $(\pi_{\mathsf{ctr},s_0}, \pi_{\mathsf{ctr},s_1})$ attesting to the evaluations $\mathsf{rand}^{(\mathrm{I})}_{s_0}(\vec{x}_{\mathsf{ctr}})$ and $\mathsf{rand}^{(\mathrm{I})}_{s_1}(\vec{x}_{\mathsf{ctr}})$. Finally, output $\pi_{\mathrm{IndexConsis}} := (\pi_{\mathrm{Index}}, \{(\pi_{\mathsf{ctr},s_0}, \pi_{\mathsf{ctr},s_1})\}_{\mathsf{ctr} \in [\mathsf{ctr}_0]})$.
- $\mathsf{Client.VerifyIndexConsis}(\ell, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\mathrm{IndexConsis}}) \to 0/1$: Parse

$$(\mathsf{ctr}_0, \{\pi_{\mathsf{ctr}}\}_{\mathsf{ctr} \in [\mathsf{ctr}_0]}) \leftarrow \pi_{\mathrm{Index}},\ (\pi_{\mathrm{Index}}, \{(\pi_{\mathsf{ctr},s_0}, \pi_{\mathsf{ctr},s_1})\}_{\mathsf{ctr} \in [\mathsf{ctr}_0]}) \leftarrow \pi_{\mathrm{IndexConsis}}$$

  Recompute $\vec{x}_{\mathsf{ctr}} := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}, \ell)$ for all $\mathsf{ctr} \in [\mathsf{ctr}_0]$, and then verify π_{Index} by running $\mathsf{Client.VerifyIndex}(\ell, \mathcal{C}_{s_1}, \vec{x}_{\mathsf{ctr}_0}, \pi_{\mathrm{Index}})$. Next, verify the PCS openings $(\pi_{\mathsf{ctr},s_0}, \pi_{\mathsf{ctr},s_1})$ attesting to the evaluations of $\mathsf{rand}^{(\mathrm{I})}_{s_0}(\vec{x}_{\mathsf{ctr}})$ and $\mathsf{rand}^{(\mathrm{I})}_{s_1}(\vec{x}_{\mathsf{ctr}})$, and check that:

$$\mathsf{rand}^{(\mathrm{I})}_{s_0}(\vec{x}_{\mathsf{ctr}}) = \mathsf{rand}^{(\mathrm{I})}_{s_1}(\vec{x}_{\mathsf{ctr}})$$

- $\mathsf{Server.ValueConsis}(\vec{x}, \mathsf{state}_{s_0}, \mathsf{state}_{s_1}) \to \pi_{\mathrm{ValueConsis}}$: Computes PCS openings $\pi_{\mathrm{ValueConsis}} = (\pi_{s_0}, \pi_{s_1})$ attesting to the evaluations $\mathsf{rand}^{(\mathrm{V})}_{s_0}(\vec{x})$ and $\mathsf{rand}^{(\mathrm{V})}_{s_1}(\vec{x})$, and output $\pi_{\mathrm{ValueConsis}}$.
- $\mathsf{Client.VerifyValueConsis}(\vec{x}, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\mathrm{ValueConsis}}) \to 0/1$: Parse $(\pi_{s_0}, \pi_{s_1}) \leftarrow \pi_{\mathrm{ValueConsis}}$, verify the PCS openings attesting to $\mathsf{rand}^{(\mathrm{V})}_{s_0}(\vec{x})$ and $\mathsf{rand}^{(\mathrm{V})}_{s_1}(\vec{x})$, and finally check that $\mathsf{rand}^{(\mathrm{V})}_{s_0}(\vec{x}) = \mathsf{rand}^{(\mathrm{V})}_{s_1}(\vec{x})$.
- $\mathsf{Auditor.VerifyInvariance}(\mathcal{C}_n, \mathcal{C}_{n+1}, \pi_{n+1}) \to 0/1$: Recompute the challenges $r^{(\mathrm{I})}_n$ and $r^{(\mathrm{V})}_n$ using the Fiat–Shamir heuristic:

$$r^{(\mathrm{I})}_n \stackrel{?}{=} \mathcal{O}(r^{(\mathrm{I})}_{n-1}, \mathrm{com}(\mathsf{index}_{n+1})),\ r^{(\mathrm{V})}_n \stackrel{?}{=} \mathcal{O}(r^{(\mathrm{V})}_{n-1}, \mathrm{com}(\mathsf{value}_{n+1}))$$

  Verify that the following relations hold among the corresponding commitments:

$$\mathrm{com}(\mathsf{rand}^{(\mathrm{V})}_{n+1}) \stackrel{?}{=} \mathrm{com}(\mathsf{rand}^{(\mathrm{V})}_n) + r^{(\mathrm{V})}_n \cdot \big(\mathrm{com}(\mathsf{value}_{n+1}) - \mathrm{com}(\mathsf{value}_n)\big),$$

$$\mathrm{com}(\mathsf{rand}^{(\mathrm{I})}_{n+1}) \stackrel{?}{=} \mathrm{com}(\mathsf{rand}^{(\mathrm{I})}_n) + r^{(\mathrm{I})}_n \cdot \big(\mathrm{com}(\mathsf{index}_{n+1}) - \mathrm{com}(\mathsf{index}_n)\big).$$

  If the PCS is homomorphic, these identities are verified directly on the commitments. Otherwise, verification proceeds non-homomorphically (see Remark 4).

Figure 4: Aegon lookup, consistency, and audit functionalities

set $\{f_i\}_{i\in[1,n]}$ and to f, along with a description of G. The verifier samples a random point $\vec{r}$, and the prover returns the evaluations $\{f_i(\vec{r})\}_{i\in[1,n]}$ and $f(\vec{r})$, together with valid opening proofs. After verifying these PCS proofs, the verifier checks whether

$$G\big(f_1(\vec{r}), f_2(\vec{r}), \ldots, f_n(\vec{r})\big) \stackrel{?}{=} f(\vec{r}).$$

In practice, the randomness $\vec{r}$ is typically derived via the Fiat–Shamir heuristic, e.g., by hashing the commitments to all polynomials in the set $\{f_i\}_{i\in[1,n]}$ along with f.

6.3 Privacy

As discussed previously, privacy is an important consideration when a transparent dictionary is used in KT. Prior works before IronDict either did not address privacy or formalised it through a *leakage function* that captures the information inherently leaked by the construction. For example, SEEMless reveals the last time a value was updated during a value lookup. Such forms of leakage are non-standard and inherent to the construction.

IronDict informally defines a notion of *strong privacy* and avoids using the leakage function unlike prior work. More specifically, IronDict informally argues that: *For both client queries and the information posted on the bulletin board, the server reveals only polynomial commitments, PCS openings, and sumcheck proofs. If each underlying component is zero-knowledge, then the protocol as a whole preserves privacy.* While this intuition is appealing, it overlooks several subtleties. For example, when the server opens the index corresponding to a label, it may reveal some unintended information about the indices of other labels with non-negligible probability (see Appendix D for more details). Although this leakage does not expose sensitive information about users' values, it nonetheless illustrates the importance of a precise and formal treatment of privacy.

We introduce formal definitions of privacy for different server functionalities. At a high level, the server must respond to two types of queries: (I) *index lookup and consistency* and (II) *value lookup and consistency*, which also capture historical value queries, since historical value queries can be expressed as combinations of value lookups and value consistency. In addition, at every epoch, the server publishes a commitment to the dictionary along with auxiliary data (i.e., commitment to the *rand polynomials*) on the bulletin board.

Our notion of privacy captures data published on the bulletin board, as well as query type (II), akin to standard definitions of zero-knowledge proofs. In contrast, index lookup and consistency queries do not directly reveal sensitive information: they merely map user labels to internal static indices. For this reason, we do not require strict zero-knowledge for index lookup and index consistency. Nevertheless, similar to previous work [Mel+15; Cha+19; Haf+26], we use standard techniques such as employing verifiable pseudorandom functions (VRFs) [MRV99] to map labels to indices, which ensures that an adversary cannot predict or exploit the distribution of labels within the database. We defer our formal definitions of privacy to Appendix D.

Size leakage. There are additional notions of privacy one might consider for KT, such as preventing an adversary from learning the total number of users, the number of currently active users or the number of updates within an epoch. In Aegon, the setup phase publicly includes an upper bound on the number of users, since most polynomial commitment schemes (including KZH) require such a size parameter. Comparable size information is also leaked by Merkle tree–based constructions. However, on the positive side, unlike Merkle tree–based approaches such as SEEMless, Parakeet and OPTIKS, where the invariance proof reveals the number of updates in an epoch, in Aegon, the information published on the bulletin board does not reveal the number of updates in that epoch.

In Aegon, an adversary attempting to infer the number of users could perform probing attacks by registering multiple labels and using their assigned identifiers (e.g., the number of hashes required to locate an empty slot) to gather statistical information about the system's current load. A detailed analysis of these attacks is left for future work; importantly, they do not contradict our privacy definition.

6.4 Concrete Instantiation

Similar to IronDict, we initialize Aegon with KZH to benefit from the efficiency advantages discussed in Section 4.1, including free openings at Boolean points and inexpensive zero-knowledge transformations. These benefits come at the cost of a trusted setup. In practice, however, we do not view this requirement as a bottleneck. As discussed earlier, the setup can be executed via an MPC ceremony, guaranteeing security as long as at least one participant behaves honestly. Concretely, the ceremony can be carried out by a small number of independent servers operated by different organizations. For example, an SRS supporting polynomials of size 2^{32} can be generated by 10 servers from different companies, with the cost per server remaining below 10 dollars on a commodity cloud instance and the computation completing in under 30 minutes per server. As a result, the entire setup can be completed in less than one hour in practice. We leave more details to Section 9.

Checking Homomorphic Relations Over zk-KZH. Our protocols are generic with respect to the choice of multilinear PCS, regardless of whether the scheme is zero-knowledge. However, this generality introduces a subtle issue that must be handled carefully. When using a zero-knowledge variant such as zk-KZH, homomorphic relations between *blinded* commitments cannot be verified directly by the auditor without additional input from the server. To illustrate the issue, recall that a standard KZH commitment to a multilinear polynomial with evaluation points $(x_1, \ldots, x_n)$ has the form $\sum_{i\in[1,n]} x_i \cdot g_i$. To make the commitment hiding—analogous to Pedersen commitments—we introduce an additional generator h and sample a random blinding factor $r \in \mathbb{F}$. The resulting hiding commitment becomes $r \cdot h + \sum_{i\in[1,n]} x_i \cdot g_i$. The difficulty arises when verifying a homomorphic relation between two such commitments: the hiding factors on the two sides of the relation are generally different. Concretely, suppose we are given two PCS commitments com_1 and com_2, each using independent blinding randomness. The problem reduces to verifying that these commitments correspond to the same underlying polynomial despite their different hiding terms. A straightforward solution is to open both commitments at a randomly chosen evaluation point and check that the evaluations coincide. By the Schwartz–Zippel lemma, equality at a random point implies equality of the polynomials with high probability. However, this approach requires two PCS openings, which can be computationally expensive. A more efficient alternative is to consider the commitment difference $\mathrm{com}_1 - \mathrm{com}_2$ and verify that it opens to the zero polynomial. Equivalently, we check that $\mathrm{com}_1 - \mathrm{com}_2 = c \cdot h$ for some $c \in \mathbb{F}$, where h is the hiding generator. This statement can be proven using a simple Σ-protocol that requires only a constant number of group operations. In conclusion, verifying homomorphic identities between blinded commitments necessitates an additional Σ-protocol. Importantly, this incurs only constant overhead for both the server and the auditor. More details of zk-KZH can be found in Appendix E.1.

Instantiation With Other Polynomial Commitments. Aegon is generic over the choice of PCS and can be instantiated with a variety of alternatives, including those that offer a *transparent setup* [Zha+21]. Such alternatives typically involve trade-offs, such as larger proof sizes, slower

Party	Operation	Homomorphic PCS	Non-Homomorphic PCS
Client	Lookup Time / Communication	$O(1)$ PCS.V / $O(1)$ PCS.P	$O(1)$ PCS.V
	Consistency Time / Communication	$O(1)$ PCS.V / $O(1)$ PCS.P	$O(1)$ PCS.V
Server	Lookup	$O(1)$ PCS.OB	$O(1)$ PCS.OB
	Consistency	$O(1)$ PCS.OB	$O(1)$ PCS.OB
	Invariance	$O(n)$ (PCS.AuxH + PCS.ComH)	$8 \times$ PCS.O
Auditor	Computation	$O(1)$ PCS.ComH	$8 \times$ PCS.V
	Communication	$O(1)$	$8 \times$ PCS proofs

PCS.P : PCS proof size
PCS.V : PCS verification time
PCS.AuxH : Homomorphic operation on PCS auxiliary data
PCS.O : PCS opening at a random point
PCS.ComH : Homomorphic operation on PCS commitments
PCS.OB : PCS opening at a boolean point

Table 1: Comparison of the efficiency of Aegon using homomorphic and non-homomorphic PCS instantiations: The main bottleneck for non-homomorphic PCS lies in the server's computation to generate the proof of invariance, which requires opening the underlying polynomials at random points and thus takes linear time in the size of the dictionary. Using a homomorphic PCS, this cost is $O(n)$ and independent (sublinear) of the dictionary size, where n is the number of updates in the epoch. For simplicity, we ignore the zero-knowledge cost, which is incurred only on lookup or consistency proofs for values, and only when the dictionary is required to achieve privacy.

opening procedures, or increased costs for achieving zero-knowledge. Consequently, selecting an appropriate PCS requires carefully balancing efficiency against trust assumptions. Identifying efficient transparent alternatives to KZH remains an interesting direction for future work.

A key property of KZH that Aegon heavily leverages is its support for free openings over the evaluation domain (e.g., the Boolean hypercube in the multilinear setting), together with an inexpensive zero-knowledge compiler. These features enable a server to answer client queries efficiently. HydraProofs [PPP25] is a concurrent work to KZH that introduces a multilinear PCS supporting free openings over the Boolean hypercube and explores several additional applications of this property. However, unlike KZH, HydraProofs does not provide zero-knowledge, and its proof size scales with the square root of the size of the Boolean hypercube. Designing polynomial commitment schemes that simultaneously achieve these properties remains an interesting direction for future work. Another key property of KZH that boosts the efficiency of Aegon is homomorphism. Table 1 summarizes the concrete costs of Aegon when instantiated with either homomorphic or non-homomorphic PCS variants.

7 Planetary-Scale Deployment of Aegon

In this section, we introduce two main techniques that allow us to implement Aegon at a planetary scale. In Section 7.1, we enable horizontal scalability by sharding the dictionary. We shard the full dictionary into η independent sub-dictionaries such that all operations of a shard will be independent of the other shards. Sharding yields several benefits.

First, because each shard is committed separately, the PCS setup need only be sized for a single shard of capacity N instead of the full $\eta \cdot N$; for a dictionary of capacity 4 billion, when Aegon is instantiated with KZH, this shrinks the SRS from roughly 1 TB to under 10 GB per shard at $\eta = 128$. Second, because the shards are independent, every server-side operation parallelizes across them: shards can be distributed over multiple machines, each managing its own database, so client throughput scales with the number of shards. Third, sharding accelerates dictionary construction by parallelizing open addressing. Index assignment via open addressing is an inherently sequential operation, i.e., elements are added one by one. We enable a construction where the open addressing on each shard is independent of the other shards.

As an independent by-product of our analysis, we obtain a notion of *sharded open addressing*. Open addressing is classically a *global* procedure over a single address space, whereas sharding partitions that space across machines. We bridge this gap by decomposing the global probe sequence into independent per-shard probes while preserving the uniqueness guarantee of the original scheme. We believe this primitive is of independent interest and elaborate on it in Appendix F.

Finally, in Section 7.2, we further enable optimal server storage by caching a bounded set of PCS opening proofs per epoch and discarding prior dictionary snapshots. Because the number of updates in an epoch is orders of magnitude smaller than the dictionary size, this strategy yields a server whose storage grows only with the retained history, rather than with the dictionary size or the number of elapsed epochs. It further supports per-client retention policies, since the cached proofs for different clients are independent.

7.1 Dictionary Sharding

In this section, we introduce our dictionary sharding technique. Concretely, we replace the one global dictionary of capacity $\eta \cdot N$ with η *independent* sub-dictionaries, each a full single-shard Aegon instance of capacity N. Assume we have η shards, indexed by $0, 1, \ldots, \eta - 1$. In the single-shard setting, a label ℓ is first mapped to an index by the index polynomial, and this index is then mapped to a value by the value polynomials. In the sharded setting we prepend one step: the server holds a VRF $\mathcal{H}_\eta \colon \{0,1\}^\star \to \{0, \ldots, \eta - 1\}$ that assigns each label a shard $j := \mathcal{H}_\eta(\ell)$. This assignment is *static*: it depends only on ℓ, so a label's shard never changes, exactly as its index never changes. All remaining dictionary operations are then handled *entirely within shard* j by the unmodified single-shard protocol of Section 6.2: inserting ℓ runs that shard's local open addressing, a value lookup opens that shard's value polynomial, and a consistency proof opens that shard's randomized polynomials across the two epochs. Because $\mathcal{H}_\eta$ is a VRF, the server returns the shard index together with a proof of correct evaluation, and the client only ever queries (and monitors) shard j, rejecting any proof served from a different shard.

A naive approach would publish all η shard commitments directly on the bulletin board. To lower both client communication and the bulletin board overhead, we instead commit to the shard commitments via a Merkle tree whose leaves are the per-shard commitment tuples: each shard j produces its single-shard commitment $\mathcal{C}_n^{(j)}$, and the server builds a Merkle tree over the leaves $(\mathcal{C}_n^{(0)}, \ldots, \mathcal{C}_n^{(\eta-1)})$, publishing only the root on the bulletin board. Each client now interacts with a single shard, so its overhead drops to logarithmic in η: it fetches only its own shard commitment

together with a Merkle authentication path of length $\log_2 \eta$ attesting that this leaf lies under the published root.

The auditor's cost, by contrast, is unchanged by this optimization. Since the auditor must verify the invariance of *every* shard, it still incurs work linear in η: it recomputes the Merkle root from all η revealed leaves to confirm the tree was constructed correctly, and then runs one invariance check per shard. Sharding does, however, reduce server and client lookup and consistency costs, since each PCS opening is now over a dictionary of size N rather than $\eta \cdot N$. Finally, the bulletin board communication itself is unaffected: the server publishes a single Merkle root per epoch regardless of η, serving the relevant leaves and their authentication paths out of band on request.

Uneven Distribution of Labels Among Shards. A subtlety of static, per-label shard assignment is that a label cannot migrate: given $j := \mathcal{H}_\eta(\ell)$, the label ℓ can only ever be assigned to shard j. We rely on this determinism for security, so that a client, or the owner of the label, can deterministically know which shard holds their label and monitor only that shard. The risk of this approach is that a single shard may exhaust its capacity while others remain underutilized, forcing a dictionary expansion (Section 10.1) before the global dictionary is truly full. This is an inter-shard load-balancing question, which we now analyze.

We model the assignment as a *balls-into-bins* process: the m registered labels are the balls, the η shards are the bins, and each label lands in the bin $\mathcal{H}_\eta(\ell)$. The relevant quantity is the maximum load, the fill of the most-loaded shard, since the first shard to overflow is what triggers an expansion. Crucially, the throws are uniform and independent: $\mathcal{H}_\eta$ is a VRF rather than a public hash, so an adversary registering labels cannot evaluate it locally and therefore cannot grind labels into a shard of its choosing. The shard a label lands in is unpredictable until the server reveals it, so the uniform balls-into-bins model holds even under adaptive, adversarial registration. The following lemma bounds how far the heaviest shard can deviate from the mean load.

Lemma 1 (Shard balance)**.** Let m labels be assigned to η shards, where each label's shard is $\mathcal{H}_\eta(\ell)$ and $\mathcal{H}_\eta$ is modeled as a random function into $\{0, \ldots, \eta - 1\}$. Let X_j be the load of shard j, so $X_j \sim \mathrm{Bin}(m, 1/\eta)$ with mean $\nu := m/\eta$. Then for every $\delta \in (0, 1)$,

$$\Pr\Big[\max_{0 \le j < \eta} X_j \;\ge\; (1+\delta)\,\nu\Big] \;\le\; \eta \cdot \exp\Big(-\tfrac{\delta^2 \nu}{3}\Big).$$

Fixing a failure probability $2^{-\lambda}$ and solving for δ yields $\delta(\lambda) = \sqrt{3\big(\lambda \ln 2 + \ln \eta\big)/\nu}$.

Proof. Each X_j is a sum of m independent Bernoulli$(1/\eta)$ indicators, so the standard multiplicative Chernoff bound gives $\Pr[X_j \ge (1+\delta)\nu] \le e^{-\delta^2 \nu/3}$ for $\delta \in (0, 1)$. A union bound over the η shards gives the stated inequality; setting the right-hand side to $2^{-\lambda}$ and solving for δ gives $\delta(\lambda)$. □

Concrete numbers. Take a planetary deployment with $\eta = 128$ shards, each of true capacity $N = 2^{25}$ labels (backed by a polynomial of 2^{27} evaluations, so the over-provisioning factor $\alpha = 4$ is already folded into N). The system thus supports a total of $\eta \cdot N = 2^{32}$ labels. When the global dictionary nears this capacity, each shard expects a load of $N = 2^{25}$ labels. By Lemma 1, the most-loaded shard exceeds this mean by at most a factor $1 + \delta(\lambda)$, where λ is the desired bit-security level; at $\lambda = 128$ this gives $\delta(128) \approx 1.4 \times 10^{-3}$. Equivalently, the first shard reaches its capacity N only once the system already holds more than $\eta \cdot N \cdot (1 - \gamma)$ labels, such that:

$$\eta \cdot N \cdot (1 - \gamma) = \eta \cdot N \cdot (1 - \Big[\frac{3(\lambda \ln 2 + \ln \eta)}{N}\Big]^{\frac{1}{2}})$$

Thus, except with probability 2^{-128}, over 99.8% of the $\eta \cdot N = 2^{32}$ total capacity is usable before any single shard overflows. The VRF assignment therefore wastes only a negligible fraction of capacity, and the decision to expand is governed by the *global* fill rather than by a single prematurely saturated shard.

Sharded Open Addressing. As an interesting by-product of the construction above, we obtain a notion of *sharded open addressing*, which partitions a global address space into η shards and runs an open-addressing procedure on each shard independently, while preserving the uniqueness guarantee of the original scheme. We elaborate on this primitive in Appendix F.

Generality of Dictionary Sharding. The sharding technique above is not tied to Aegon: the same VRF-based partitioning applies to other key transparency schemes, such as WhatsApp's AKD. One simply instantiates η independent copies of the scheme and uses a VRF to assign each label to a shard, $j := \mathcal{H}_\eta(\ell)$, exactly as before; every operation on a label is then served entirely by its assigned instance, splitting the global dictionary into η local shards. For AKD this partitioning comes at essentially no cost to any party—server, client, or auditor. Specifically for the auditor, its per-epoch cost is linear in the number of key updates published in that epoch. Sharding only redistributes those updates across the η instances without changing their total number, so the auditor's aggregate communication and computation are unchanged whether or not the dictionary is sharded.

7.2 Historical Values

We rely on two key observations that allow the server to cache proofs for historical values and safely discard the previous dictionary states. This significantly improves both the server's storage and its efficiency in retrieving historical values.

The first observation is that the number of updates per epoch is much smaller than the total dictionary size. For instance, the dictionary may contain billions of entries, while an application such as WhatsApp processes roughly 1250 updates per second at the time of writing this paper [Fac25]. With an epoch duration of one minute, this translates to about 75,000 updates per epoch, which is negligible compared to the total number of entries. Next, assume that $|\Delta_i|$ updates occur during epoch i. The second observation is that the server can precompute and store $O(|\Delta_i|)$ PCS openings at each epoch, allowing it to discard all previous dictionary states safely. That is, the server only needs to retain the latest dictionary state along with these precomputed proofs, resulting in optimal server storage.

High-Level Overview of the Protocol. As a reminder, suppose a client queries their historical values, represented as $\{(n_i, v_i)\}_{i\in[t]}$, meaning the value changed to v_i at epoch n_i. To prove correctness, the server must show two facts for each interval $[n_i, n_{i+1})$: (1) the value remained unchanged throughout this interval, and (2) the value at epoch n_i is indeed v_i. Therefore, as long as the server caches the lookup proofs for each (n_i, v_i) pair and the consistency proofs for every interval $[n_i, n_{i+1})$, it no longer needs to retain past dictionary states. The only persistent data required is the sequence of commitments, which is already available on the bulletin board. These observations allow the server to reduce its storage overhead substantially.

Concrete Approach. Consider a client with label ℓ who registered at epoch n_0. Let the associated index be $\vec{x}_{\mathsf{ctr}_0} := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}_0, \ell)$, and for each $\mathsf{ctr} \in [\mathsf{ctr}_0 - 1]$ define $\vec{x}_{\mathsf{ctr}} := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}, \ell)$. Suppose

we are currently at epoch n, and let $\{(n_i, v_i)\}_{i\in[t]}$ denote its historical values, which are ordered such that

$$1 \le n_0 < n_1 < \cdots < n_t < n.$$

Next, we demonstrate how the server can only store ctr_0+3t+5 polynomial opening proofs to enable efficient proof of the historical values, along with proof of index correctness. Specifically, when the client registers at epoch n_0, the server stores the PCS openings for $\mathsf{index}_{n_0}(\vec{x}_{\mathsf{ctr}})$ and $\mathsf{rand}^{(\mathrm{I})}_{n_0}(\vec{x}_{\mathsf{ctr}})$ for all $\mathsf{ctr} \in [\mathsf{ctr}_0]$. Additionally, for each change epoch n_i, the server stores PCS openings for the following polynomial evaluations:

$$\forall\, i \in [t] :\ \mathsf{value}_{n_i}(\vec{x}), \qquad \forall\, i \in [t] :\ \mathsf{rand}^{(\mathrm{V})}_{n_i-1}(\vec{x}),\ \mathsf{rand}^{(\mathrm{V})}_{n_i}(\vec{x}).$$

These cached proofs, together with the latest state ($\mathsf{index}_{n_t},\ \mathsf{value}_{n_t},\ \mathsf{rand}^{(\mathrm{I})}_{n_t},\ \mathsf{rand}^{(\mathrm{V})}_{n_t}$) are sufficient for the server to prove the entire history of the client:

- *Index correctness.* Firstly, to verify that the index has not changed since epoch n_0, the server sends the stored PCS openings for $\{\mathsf{rand}^{(\mathrm{I})}_{n_0}(\vec{x}_{\mathsf{ctr}})\}_{\mathsf{ctr}\in[\mathsf{ctr}_0]}$ and computes PCS opening for evaluations $\{\mathsf{rand}^{(\mathrm{I})}_{n}(\vec{x}_{\mathsf{ctr}})\}_{\mathsf{ctr}\in[\mathsf{ctr}_0]}$ from the latest state polynomial $\mathsf{rand}^{(\mathrm{I})}_{n}$. The client first verifies the received PCS proofs and then checks the following to ensure that its index has remained unchanged:

  $$\forall\, \mathsf{ctr} \in [\mathsf{ctr}_0] :\quad \mathsf{rand}^{(\mathrm{I})}_{n_0}(\vec{x}_{\mathsf{ctr}}) = \mathsf{rand}^{(\mathrm{I})}_{n}(\vec{x}_{\mathsf{ctr}})$$

  Then, the server sends PCS openings for $\{\mathsf{index}_{n_0}(\vec{x}_{\mathsf{ctr}})\}_{\mathsf{ctr}\in[\mathsf{ctr}_0]}$, which serve as proof of the correctness of the index for the client.

- *Historical values.* Next, the stored PCS openings $\mathsf{value}_{n_i}(\vec{x}_{\mathsf{ctr}_0})$ provide lookup proofs at epochs $\{n_i\}_{i\in[t]}$ and the PCS openings for evaluations

  $$\{\mathsf{rand}^{(\mathrm{V})}_{n_i-1}(\vec{x}_{\mathsf{ctr}_0}),\ \mathsf{rand}^{(\mathrm{V})}_{n_i}(\vec{x}_{\mathsf{ctr}_0})\}_{i\in[t]},\ \mathsf{rand}^{(\mathrm{V})}_{n}(\vec{x}_{\mathsf{ctr}_0})$$

  provide consistency proofs for the intervals $\{[n_i, n_{i+1})\}_{i\in[t-1]}$ and $[n_t, n]$.

Flexible Policy Over Historical Values. Unlike all prior schemes, an important advantage of the design above is that it gives the server complete flexibility in managing the historical values. Since lookup proofs for different clients are independent, in principle, the server can apply distinct policies for each client. For instance, one client may request storing only the last two versions of their value, while another may require versions spanning the past five years. This flexibility allows the server to save storage by discarding unnecessary history.

For example, the server may choose to retain historical values only for the past year and safely remove older entries. This contrasts with Merkle tree–based approaches, such as SEEMless and Parakeet (the baseline for WhatsApp Key Transparency), where the server maintains an append-only tree and its history grows indefinitely, forcing it to store all past entries—even those that are no longer relevant.

Discarding old history introduces a soundness caveat: if the server retains only the last T epochs of a client's history, then a malicious server could have tampered with that client's value during any earlier period and erased the evidence. To preserve soundness under a retention window of T, we therefore require that each client come online at least once per interval of length T and verify the consistency of its value. This is the same kind of liveness assumption made by the resetting

mechanism of OPTIKS [Len+24], which periodically rebuilds the directory and keeps only the most recent value of each label. The mechanisms differ, however, in what they ask of the server and of clients. OPTIKS requires the server to recompute the directory tree from scratch at every reset, and clients must explicitly re-verify that their latest value is present in the new tree. In Aegon, neither step is needed: the server simply ceases to retain proofs older than the retention window, while imposing no additional burden on clients.

8 System Implementation and Optimizations

We implement and benchmark Aegon in roughly 50,000 lines of Rust.[7] Unlike IronDict, our implementation is production-grade and exposes the same API as WhatsApp's AKD[8].

Schema and Library Choices. We instantiate our construction using the KZH-k polynomial commitment scheme, reusing the KZH-k implementation from the IronDict codebase[9]. All cryptographic operations are implemented using the arkworks library [con22]. Following IronDict, we instantiate KZH over the BN254 curve, a standard pairing-friendly curve widely adopted in industry [But14]. BN254 provides at least 100 bits of security. Moreover, since the size of its scalar field is larger than 2^{200}, it satisfies our requirement that $2^{2\lambda} \leq |\mathbb{F}|$.

We use ECVRF-EDWARDS25519-SHA512-TAI as our verifiable random function specified in RFC 9381 [Gol+23]. This is the same Schnorr-style construction [Sch90] deployed by AKD. Following IronDict, we use a provisioning factor of $\alpha = 4$, meaning that the dictionary's nominal capacity is set to $4\times$ its true capacity.

Hardware and Deployment. A central design goal of our implementation is that every server-side process (shards, coordinator, masking server, and label–value database) must run on *standard commodity servers* rather than on specialised hardware, so that a Aegon deployment can be provisioned from any major cloud provider without bespoke procurement. Section 9 lists the specific instance types used for our planetary-scale evaluation and the resulting machine count.

8.1 Server Architecture

An Aegon deployment is composed of three distinct kinds of processes, illustrated in Figure 5: (i) a single *coordinator* that mediates every client request; (ii) η *shard servers*, where η is a power of two, each owning one disjoint slice of the directory polynomials *and* a co-located label–value database that persists that shard's own value bytes, per-label value-history lists, placement records, and cached history openings; and (iii) m *masking servers* that pre-compute the zero-knowledge masking packages the shards consume during lookup.

Two-Layer Open Addressing. Routing is two-layer: the coordinator applies a first-layer hash $\mathcal{H}_\eta$ to each label to pick the owning shard, and the shard itself applies a second-layer hash $\mathcal{H}^{\text{bits}}$ plus open addressing to pick a slot within its own polynomial.

Coordinator. The coordinator is the only process clients talk to. It holds $O(\eta)$ state independent of the label count, consisting of the epoch-commitment chain and a per-shard fullness counter. On any incoming request it applies $\mathcal{H}_\eta$ to the label to pick the owning shard. Lookups return that shard's response wrapped in a Merkle authentication path against the current sharded epoch commitment. Publishes partition the batch across shards by $\mathcal{H}_\eta$, collect the new commitment each shard returns, derive a single shared challenge by hashing all of those commitments together, and hand that challenge back so every shard applies the same update.

[7] Our codebase can be found here: https://github.com/alireza-shirzad/aegon
[8] https://github.com/facebook/akd
[9] https://github.com/alireza-shirzad/irondict

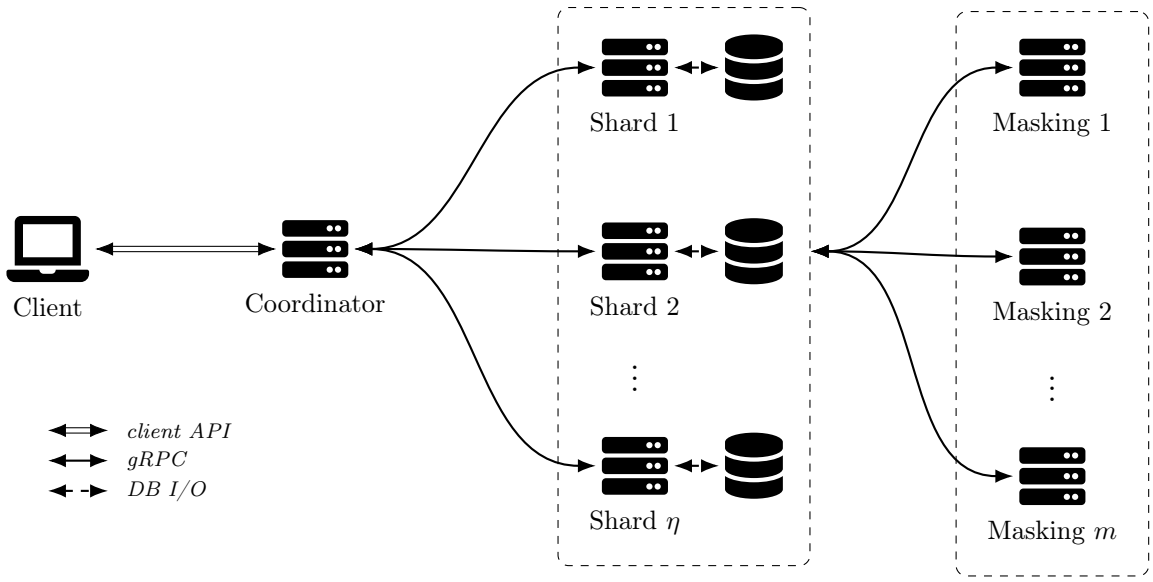

Figure 5: Aegon system architecture

Shard server. Each shard server owns four multilinear polynomials of capacity $2^c = \alpha N/\eta$ over a Boolean hypercube (the index and value polynomials, plus their two chain-blinded variants), where N is the true dictionary capacity, $\alpha = 4$ is the over-provisioning factor, and η is the number of shards. Alongside the polynomials, each shard owns a co-located label–value database and a gRPC interface the coordinator uses to commit batches, open the polynomials at any slot, and read the current per-shard commitment. Shards never talk to each other and never touch another shard's database.

Masking Server. The critical path of every lookup is the MSM work that masks the polynomial opening for zero knowledge, so we lift that work off the shards entirely: a pool of m *masking servers* pre-computes opening-point-agnostic masking packages and ships them to shards on demand via gRPC, keeping per-lookup shard work independent of the masking cost. Because the shards round-robin their fetches across the pool, the pool size is the primary throughput knob for lookups: operators scale m up under peak load and back down when load is small, without touching the shards or the coordinator. The masking servers are only required when the deployment opts into zero-knowledge openings, which is the configuration we report on throughout this section.

8.2 Optimisations

We implement several optimizations to improve efficiency compared to IronDict.

Tuning the KZH-k block parameter k. KZH-k exposes a block parameter k that controls the trade-off between the prover's and the verifier's workload. We optimize for the prover's opening time, since it sits on the critical path of every client lookup and thus governs both the masking server's throughput and the user-perceived latency.

At Boolean points, the only k-dependent server-side cost in Aegon is the zero-knowledge overhead that runs once per zero-knowledge opening by the masking server. We model the cost-relevant work as the cost of building the auxiliary table for the *sparse masking polynomial* used by the zero-knowledge compiler of Appendix E.1, since this table construction dominates the masking server's

work in a zero-knowledge opening. More specifically, for a multilinear polynomial of size $N = 2^\mu$, the zero-knowledge compiler samples a random masking polynomial $r(\vec{X})$ with $k \cdot 2^{\mu/k}$ non-zero evaluations. For every opening, the prover must build the KZH-k auxiliary table of r. The auxiliary table has $k-1$ levels (Figure 13, $j = 1, \ldots, k-1$). On the sparse path each level performs one G1 addition per non-zero coefficient of r, regardless of the level's index, because every non-zero coefficient contributes to exactly one prefix cell per level. The work is therefore

$$f(k) \;=\; \underbrace{(k-1)}_{\text{levels}} \cdot \underbrace{k \cdot 2^{\mu/k}}_{\text{non-zero evals per level}} \;=\; k\,(k-1) \cdot 2^{\mu/k}, \qquad 1 \le k \le \mu,$$

matching the leading term $O(k^2 \cdot N^{1/k})$ of the zero-knowledge overhead claimed in Section 4.1. We pick the integer $k^\star$ that minimises $f(k)$ at SRS-generation time in the following way: Treating k as continuous, the derivative is

$$f'(k) \;=\; (2k-1) \cdot 2^{\mu/k} \;-\; k(k-1) \cdot 2^{\mu/k} \cdot \tfrac{\mu \ln 2}{k^2} \;=\; 2^{\mu/k} \Big[(2k-1) \;-\; \tfrac{(k-1)\,\mu \ln 2}{k} \Big].$$

Setting $f'(k) = 0$ yields the implicit equation $k(2k-1) = (k-1)\,\mu \ln 2$, which is quadratic in k with positive root

$$k^\star \;=\; \frac{(1+\mu \ln 2) \;+\; \sqrt{(1+\mu \ln 2)^2 - 8\mu \ln 2}}{4} \;\approx\; \tfrac{\ln 2}{2}\,\mu \;\approx\; 0.347\,\mu,$$

i.e. $k^\star$ grows linearly in the number of variables $\mu = \log_2 N$ with constant $\frac{\ln 2}{2}$. Rounding to the nearest integer across the deployment range gives the values in Table 2. For our concrete deployment

$k^\star$	1	2	3	4	5	6	7	8	9	10	11	12
μ range	1	2–9	10–12	13–14	15–17	18–20	21–23	24–26	27–28	29–31	32–34	35

Table 2: Optimal KZH-k block parameter $k^\star$ as a function of the number of variables $\mu = \log_2 N$

regimes this yields $k = 7$ for the *small* regime ($\mu = 22$) and $k = 9$ for the *medium* and *large* regimes ($\mu = 27$).

Efficient Implementation of MSM. Our implementation relies on Arkworks for finite field and group operations. To further optimize performance, we introduce a wrapper for multi-scalar multiplication (MSM) that dynamically selects the most efficient algorithm based on the input size. In particular, we use Pippenger's bucket method for large MSM instances, while for smaller instances (such as those arising in zero-knowledge masking of polynomials), we instead use parallel group exponentiations, which are more efficient in this regime. Our implementation also adapts parallelism to the underlying hardware, such as the number of available CPU cores.

9 Evaluation

We evaluate the setup, publish, lookup, history, and audit paths of three systems: Aegon (this work); IronDict [Haf+26], the academic state of the art; and AKD, the industrial state of the art deployed by WhatsApp, based on the academic works SEEMless [Cha+19] and Parakeet [Mal+23]. All reported runtimes and sizes in this section are approximate and are rounded to two significant figures.

Dictionary Size. We fix the dictionary size to planetary scale: a true capacity of 2^{32} entries (over 4 billion) at an over-provisioning factor $\alpha = 4$, for a total directory capacity of 2^{34} entries.

Hardware. All measurements run on Google Cloud Platform in zone `us-central1-f` using general-purpose machines of the `n2-standard` family. We evaluate Aegon on a 128-shard cluster in which each shard provides 2^{27} polynomial slots. All shards run on identical commodity `n2-standard-16` instances (16 vCPU, 64 GB RAM); the coordinator runs on a smaller `n2-standard-4` instance (4 vCPU, 16 GB RAM); and the Redis label–value backend runs on an `n2-standard-2` instance (2 vCPU, 8 GB RAM). Every process communicates by gRPC over the cloud-provider network. Unless stated otherwise, we provision a single masking server running on the same commodity hardware as a shard. The deployment therefore comprises 131 machines in total (128 shards plus coordinator, Redis backend, and masking server). Aegon's client and auditor benchmarks run single-threaded on a dedicated `n2-standard-16` host in the same zone, so the wall-clock numbers we report for client verify time and auditor verify time reflect a commodity x86 core rather than a laptop.

Neither baseline can run on commodity hardware at planetary scale. IronDict is a single-machine system whose SRS and polynomial state at $\log_2 N = 34$ do not fit in commodity RAM, so we run it on a `c4-highmem-144-lssd` instance (144 vCPU, 1.1 TB RAM), the same machine used by the authors in the IronDict paper [Haf+26]; its clients and auditors run on the same host as the server, which is a strict advantage: there is no network round-trip in the reported latencies. AKD's publish path holds each batch's VRF-keyed Merkle updates and per-leaf state in memory over the course of the batch, and at planetary scale this working set also exceeds commodity RAM; we run its server on `n2-highmem-32` (32 vCPU, 256 GB RAM), which we picked so that its publish path does not run out of RAM. Its MySQL backend runs on a second `n2-highmem-32` in the same zone. As with IronDict, AKD's client and auditor benchmarks run in-process on the server host, which is again a strict advantage: there is no network round-trip in the reported latencies.

9.1 Setup Benchmark

We measure the one-time cost of the trusted setup, which produces the structured reference string (SRS) consumed by every shard. Aegon runs setup on a commodity `n2-standard-16` shard host (16 vCPU), while IronDict runs its setup on a much stronger `c4-highmem-144-lssd` instance (144 vCPU). Since SRS generation is highly parallelizable, IronDict's 9× core-count advantage translates almost directly into faster setup, so the comparison is intentionally biased in IronDict's favour. Figure 6 compares Aegon's setup cost to the equivalent single-shard configuration of IronDict and to AKD, whose only setup step is a VRF keypair generation.

Both Aegon and IronDict require a trusted setup to sample the KZH trapdoors. Even under IronDict's hardware advantage, Aegon is 11× faster in wall-clock time (324 s vs. 60 min) and produces a 41× smaller prover key per shard (9.14 GB vs. 350 GiB), because each Aegon shard commits to a 2^{27}-slot slice of the directory rather than to the full 2^{34}-slot polynomial. On this axis AKD is

strictly better than both: it requires no trusted setup at all, and its key material is a single VRF keypair on the order of 64 bytes that is generated in microseconds.

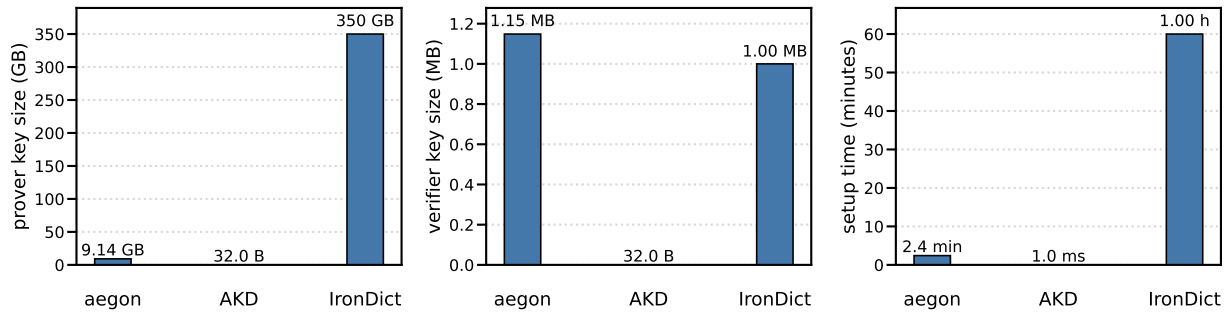

Figure 6: Setup cost across systems: prover-key size, verifier-key size, and wall-clock setup time for Aegon (per shard), IronDict (single shard at $\log_2 N = 34$), and AKD (VRF keypair only).

9.2 Publish & Migration Time

Publish time is the wall-clock latency of a single publish batch: the delay between a batch of user registrations reaching the server and those keys being committed to the bulletin board and made available for lookup. This is exactly the wait a newly-registered user experiences before their key becomes usable, so keeping it short is central to the user experience.

Figure 7's left panel reports per-publish latency as a function of batch size. At batch size $B = 65{,}536$, Aegon commits a batch in 3.9 s, versus 68 s for AKD at 1% fill (a 18× gap) and 900 s (15 min) for IronDict, a 230× gap. IronDict's publish is dominated by a sumcheck over the entire dictionary, whose per-batch cost grows with the full directory capacity rather than with the batch size; Aegon eliminates that sumcheck entirely, which is the primary source of the speedup.

Migration. A second dimension that matters for verifiable key directories is the time to build the directory from scratch: the one-time cost paid by a messenger or organization migrating an existing user base onto the system. Because the whole migration is a batch job with no live user waiting on any individual publish, batch size can be pushed as high as machine memory allows; the sweep in the right panel of Figure 7 does exactly this, enlarging the publish batch until each system runs out of memory (OOM). That sweep exposes the best-possible throughput each system can sustain, and dividing the target directory size by it gives the wall-clock time to fill a planetary-scale dictionary.

At its OOM ceiling Aegon sustains 18k entries/s cluster-wide. AKD's much larger `n2-highmem-32` accommodates much bigger batches, but its per-batch throughput remains flat at 0.95k entries/s regardless of batch size (because publish latency is proportional to batch size), so Aegon still enjoys a 19× throughput advantage. IronDict takes the opposite extreme: on its `c4-highmem-144-lssd` beast it can push batches to 70M entries before OOMing, at which point its throughput reaches 78k entries/s, roughly 4× higher than Aegon's plateau. Dividing the target directory size ($2^{32} \approx 4.3$B entries) by these ceilings, migrating a full planetary-scale dictionary takes 66 hours (2.8 days) on Aegon, 52 days on AKD, and 15 hours on IronDict.

The lookup-side serving-latency knee is reported alongside the value-lookup benchmark in Figure 9, since it measures lookup rather than publish behaviour.

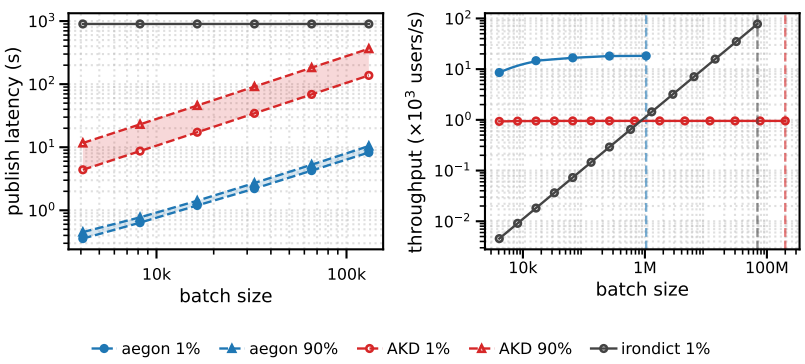

Figure 7: Publish-side benchmark summary. *Left:* publish latency vs. batch size (log y-axis) for Aegon at 1% and 90% fill, AKD at 1% and 90%, and IronDict at zero fill. *Right:* publish throughput vs. batch size (log-log axes). Aegon's end-to-end fill throughput plateaus above 18 k entries/s. AKD's per-batch throughput asymptotes to 950 entries/s at 1% fill (throughput $= B/\text{publish\_ms}(B)$, essentially flat because publish latency is proportional to B). IronDict's throughput scales linearly with B under the fixed-cost commit model. Each curve extends to the per-batch OOM ceiling implied by the target machine's RAM: AKD 200 M on `n2-highmem-32`, IronDict 70 M on `c4-highmem-144-lssd`. Terminal × marks the OOM point.

9.3 Lookup Benchmark

Aegon exposes two point-lookup endpoints (*label* and *value*), benchmarked separately so that downstream applications can pay only for the query they need; the two *history* variants are reported in Section 9.4. For each endpoint we report three quantities side by side in a single figure: server-direct latency (the polynomial-commitment opening work that runs inside Aegon, excluding network and serialization), end-to-end client verify latency (a full gRPC round trip, deserialization, and pairing-based verification of the returned openings against the bulletin-board commitment), and on-wire proof size. The point lookups walk the open-addressing trail and produce one PCS opening per probe, so cost is dominated by probe count.

Label lookup. AKD stores each label bundled with its value in a single leaf and cannot answer a label-only lookup separately from a full value lookup, so we compare Aegon's label lookup only against IronDict. At 1% fill Aegon serves a label lookup in 5.8 ms of server-side crypto work vs. 30 ms for IronDict, a 5× speedup; end-to-end client verify latency is 6.6 ms vs. 30 ms, a 4.5× speedup; and the on-wire proof is 5.1 KB vs. 5.6 KB, a 1.1× reduction. IronDict's client verify runs in the same process as its server (Section 9), so its 30 ms client number excludes any network cost, while Aegon's 6.6 ms includes a full gRPC round-trip to a remote coordinator.

Value lookup. AKD's `Directory::lookup` is its value-binding operation, so the value-lookup benchmark compares Aegon against both baselines. At 1% fill Aegon serves a value lookup in 5.9 ms server-side vs. 9.7 ms for AKD (1.6× faster) and 30 ms for IronDict (5× faster). Client verify latency is 6.6 ms vs. 9.7 ms for AKD (1.5× faster) and 30 ms for IronDict (4.5× faster). The on-wire proof is 5.1 KB vs. 18.3 KB for AKD (3.6× smaller) and 5.6 KB for IronDict (1.1× smaller). As with the label case, both baselines' client-verify numbers exclude network cost, while Aegon's includes a full gRPC round-trip.

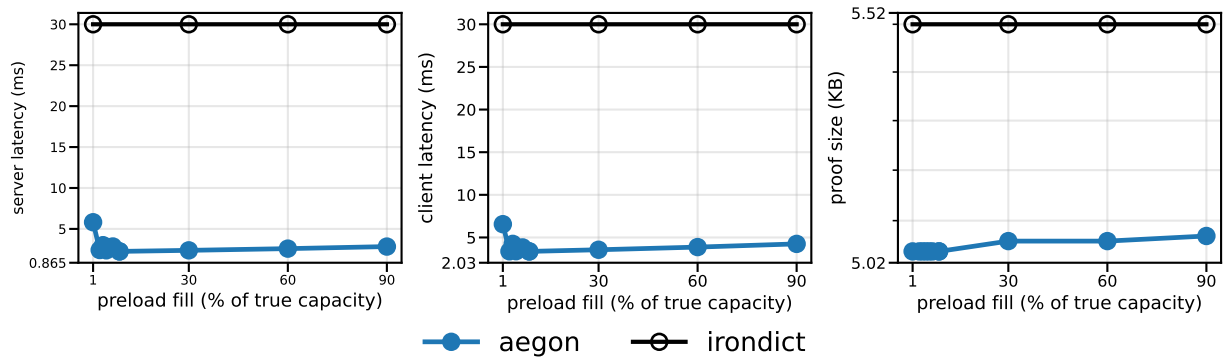

Figure 8: *Label* lookup: server-direct latency, client verify latency, and on-wire proof size vs. preload fill.

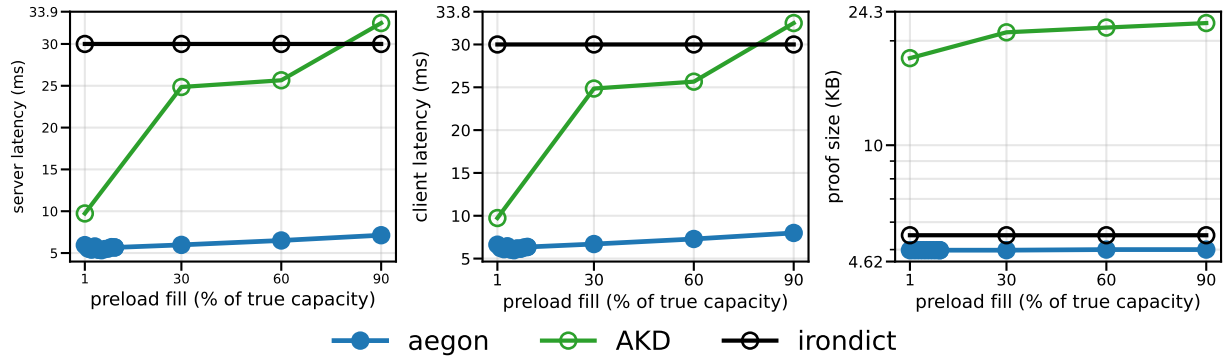

Figure 9: *Value* lookup, same 3-panel structure as Figure 8 (server latency, client latency, proof size vs. preload fill), but with AKD overlaid since AKD's `Directory::lookup` is the value-binding operation.

Serving throughput. Sustained lookup QPS is an important system-design knob: production lookup load on a planetary-scale KT deployment can reach up to 2k lookups/s at peak, and any serving stack must be able to absorb that. Table 3 reports the peak sustained lookup rate each system delivers on the hardware we benchmarked it on. The peak QPS figures for AKD and IronDict are also reported in the same table for reference; we do not directly compare the numbers because the underlying hardware differs.

Aegon is not throughput-bound at any one machine: each shard's zero-knowledge opening pulls a pre-computed *masking package* from a pool of masking servers over gRPC, so the serving ceiling scales linearly with the size of that pool. At our $\log_2 = 27$, $k = 9$ configuration each masking server sustains 120 packages/s, and hitting the 2k QPS peak load therefore requires 17 masking-server VMs sitting in front of the shard cluster (approximately what the reported number in Table 3 was measured with). Crucially, this pool is elastic: because the masking servers sit off the shard critical path and shards round-robin their fetches across whichever set of masking servers is registered, operators can grow or shrink the masking pool at runtime as load shifts, without touching the shards or the coordinator. Neither of the baselines exposes an analogous scale-out knob.

Table 3: Single-machine peak lookup throughput.

System	Machine	Cores	Peak QPS
Aegon	`n2-standard-16`	16	2 k
AKD	`n2-highmem-32`	32	0.98 k
IronDict	`c4-highmem-144-lssd`	144	4.8 k

9.4 History Lookup

Beyond the point lookups of Section 9.3, Aegon exposes two *history* endpoints, *label-history* and *value-history*, that return up to the last 5 changes on each axis (`HISTORY_WINDOW = 5` in the reference implementation), for 10 historical entries in total per call: one 5-entry chain of label placements and one 5-entry chain of value updates, plus a live freshness opening. These operations read a small number of pre-computed openings from the coordinator's key-value store and append the freshness opening, so their cost is dominated by KV-store I/O and history-window length rather than by probe count. Aegon splits history into two endpoints so that clients can pay only for the sequence they need. IronDict has no history operation at all: its `lookup_prove` returns only the current value of a label, i.e., one point, on the value side only, with no historical entries. AKD's `Directory::key_history` returns the complete history of a label in one bundled call.

At 1% fill, Aegon's combined label+value history call costs 11.8 ms server-side, 13.2 ms end-to-end client verify, and produces a 29.6 KB proof. Against IronDict's 30 ms server / 30 ms client / 5.6 KB single-point lookup, Aegon is 2.5× faster server-side and 2.3× faster client-side while returning 10× more information, at the cost of a 5× larger proof, since the extra bytes carry the two 5-entry chains and the freshness opening that IronDict's point lookup does not. Against AKD's `key_history` (3.3 ms / 3.3 ms / 24.8 KB at this fill), AKD's wall-time advantage is largely an artefact of the low-fill regime: each label has been updated at most once, so `key_history` collapses to a near-point lookup. As history depth grows, so does its cost: at 90% fill AKD's `key_history` rises to 59 ms and 43 KB per call, while Aegon's per-call cost is bounded by the 5-entry window.

Figure 10 plots the like-for-like comparison against the single-call systems: Aegon's combined label+value history against AKD's `key_history` and IronDict's `lookup_prove`. The IronDict curve is a reference: it shows what a single point lookup costs on that system, not a history operation, since IronDict cannot return one.

9.5 Auditor Benchmark

A third party can independently audit every epoch transition produced by the server using Aegon's commitment-homomorphism check. The check requires no PCS openings: given the previous and the new sharded epoch commitment, the auditor performs only $\mathcal{O}(\eta)$ group equations and updates its rolling Fiat–Shamir state. Figure 11 reports the per-epoch auditor latency together with the per-epoch audit bandwidth, which is exactly the size of one sharded epoch commitment, the only object the auditor must pull from the bulletin board per epoch. Both metrics are essentially independent of dictionary fill, which is the central efficiency claim of the audit construction.

At 1% fill, Aegon verifies one epoch transition in 64 ms and downloads a 30 KB commitment. Against AKD's per-epoch audit at the same fill, this is 7× faster in wall time (64 ms vs. 440 ms) and roughly 3,900× smaller on wire (30 KB vs. 115 MB). The gap widens dramatically as the directory fills, since AKD's per-epoch audit scales with cumulative updates while Aegon's does not: at 90% fill AKD's audit balloons to 24 s of verify work and 2.5 GB of downloaded audit proof, whereas Aegon's per-epoch cost is unchanged. Against IronDict's 35 ms verify and 8 KB audit proof, the

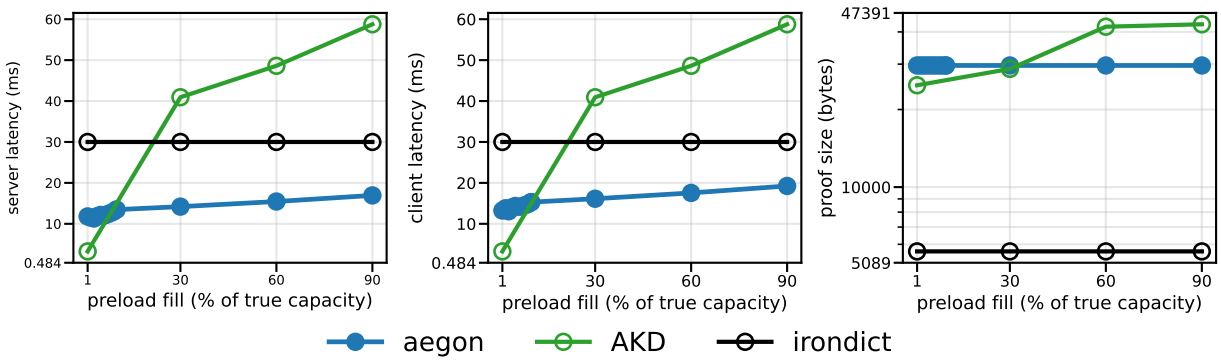

Figure 10: *Combined-history* lookup: Aegon's combined label+value history (elementwise sum of the two per-endpoint measurements), AKD's bundled `Directory::key_history`, and IronDict's single-lookup cost as a reference (it has no history operation). Server-direct latency, client verify latency, and on-wire proof size vs. preload fill. Proof size is reported in bytes because history-window proofs are an order of magnitude smaller than the point-lookup proofs.

trade goes the other way: Aegon is 1.8× slower and 4× larger, but 30 KB on the wire and tens of milliseconds of work per epoch remain well within any practical audit budget.

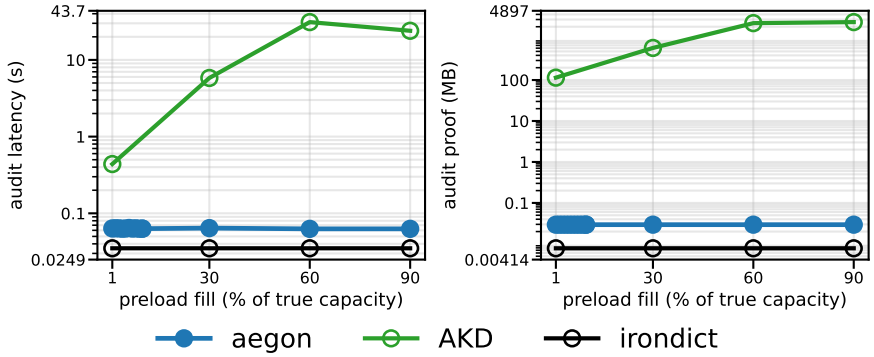

Figure 11: Auditor verify latency and per-epoch audit bandwidth vs. preload fill. Per-epoch audit cost is dominated by one group equation per shard, so latency and bandwidth both scale with $\eta = 128$ and are essentially flat in fill.

10 Extensions and Discussion

In this section, we discuss several extensions and design considerations that are necessary for the real-world, long-term deployment of Aegon. In Section 10.1, we focus on system reconfiguration. Any long-running deployment should anticipate the need for reconfiguration. For example, the number of users may exceed the initially provisioned dictionary capacity, necessitating an increase in the dictionary's capacity. Other reconfiguration scenarios include updating the PCS setup, migrating to a different PCS, or rotating the VRF secret key used for index assignment.

Finally, in Section 10.2, we address post-quantum considerations. In particular, we explain why initializing Aegon with KZH, which is not quantum-resistant, does not pose an immediate security risk. Since KZH commitments are statistically hiding, the privacy of historical data remains intact even in the presence of a future quantum adversary.

10.1 Dictionary Reconfiguration

A scheme intended for long-term deployment must support reconfiguration. In practice, cryptographic parameters cannot be assumed to remain fixed indefinitely, and the system should allow for updating critical components such as VRF keys, the PCS setup, or even migrating from one PCS construction to another.

This flexibility is essential in adversarial or long-lived settings. For instance, if the VRF keys are ever exposed, the system must be able to rotate them without requiring a complete redeployment. Similarly, Aegon is instantiated with KZH, which relies on a trusted but *updatable* SRS; best security practices dictate that it should be refreshed periodically to mitigate trust erosion and reduce the impact of potential compromise.

Beyond cryptographic reconfiguration, the system must also handle dictionary evolution. Aegon imposes an upper bound on the number of supported clients, but in many real-world deployments, the client base grows over time. As a result, the system should support dictionary expansion to accommodate new clients without disrupting the existing state.

We categorize reconfigurations into two types: *manual* and *automatic*. In a *manual reconfiguration*, each client must individually verify that their value remains consistent before and after the reconfiguration. In contrast, an *automatic reconfiguration* removes this burden from clients: the server instead produces a proof attesting to the correctness of the reconfiguration, which can be verified by the auditors. Crucially, manual reconfigurations are undesirable in practice, as they impose additional responsibility and trust assumptions on clients.

General Recipe For a Manual Reconfiguration. A general approach for handling reconfigurations to dictionary parameters—such as the setup, VRF keys, or the dictionary size—is for the server to construct a new empty dictionary with the updated parameters and selectively add clients to the new dictionary. For instance, in the case of an expansion, the server may add all existing clients. We then delegate the consistency check of the dictionary before and after the reconfiguration to the clients. When clients reconnect following the reconfiguration, they check that their values remain consistent across the dictionaries before and after the reconfiguration.

This process is reminiscent of the *resetting mechanism* in OPTIKS, where the server periodically rebuilds a new Merkle tree containing only the most recent label-value pairs to remove outdated values. In that context, clients are also responsible for verifying that their latest values are correctly included in the new tree. However, our approach differs from OPTIKS in two practical ways: (i) we do not compromise soundness—the server retains both the old and new versions of the dictionary. This allows clients to verify the consistency of their values at any time, or at least for a period

determined by server policies. In contrast, OPTIKS prioritizes storage and discards the old tree. (ii) Dictionary reconfigurations are expected to occur infrequently, whereas the resetting mechanism in OPTIKS is designed to operate periodically.

Finally, as a subtle detail, we highlight that when the dictionary reconfigures, the associated rand polynomials *reset* and are no longer applicable. However, we argue that this is not problematic for value consistency checks. Suppose the dictionary reconfigures at epoch n, and the server wants to prove that a value has remained unchanged from epoch s_0 to epoch s_1, such that $s_0 < n < s_1$. Since the dictionary reconfigures occurred at epoch n, there exist two commitments at epoch n—associated with the dictionary before and after the reconfiguration. The client can: (i) open their value in both commitments and ensure they match, and (ii) check value consistency separately for the two intervals: from s_0 to n (using the rand polynomials before the reconfiguration) and from n to s_1 (using the rand polynomials after the reconfiguration). We highlight that the index, value, and rand polynomials at epoch n (for both before and after the reconfiguration) must be stored, since some clients may reconnect later and need to verify consistency retrospectively.

Such a general configuration can be applied to *dictionary expansion* or *VRF key updates*. Next, we present an automatic reconfiguration mechanism for the PCS setup, i.e., updating the SRS for KZH or transitioning the system to a different PCS scheme without altering the dictionary size and the underlying field $\mathbb{F}$.

Remark 5. Our dictionary sharding technique enables the reuse of the existing polynomial setup even after the dictionary expands. This is advantageous because it avoids generating a new setup each time the dictionary grows. However, this approach comes with a trade-off in the auditor's cost. Specifically, if the current dictionary is represented using η shard polynomials, and the dictionary expands by a rate of r, then the new dictionary will be represented using $r \cdot \eta$ shard polynomials. Consequently, the auditor's communication cost increases proportionally by a factor of r.

Automatic Configuration to Update the PCS Setup. As mentioned in Section 4.1, KZH supports an updatable SRS that can be refreshed periodically. From a practical standpoint, updating the SRS periodically is not only desirable but often necessary. This consideration extends beyond KZH itself: whenever Aegon is instantiated with any PCS, it may be required to update or rerun the setup phase, or move to another PCS with a different setup. We describe an automatic reconfiguration for transitioning to a new PCS setup without requiring clients to perform manual consistency checks. In this approach, the server provides a succinct proof demonstrating that the dictionary commitment under the previous PCS setup (e.g., SRS) is consistent with the commitment generated under the new PCS setup. Such an automated consistency check for SRS rotation is beneficial because it may occur periodically for security reasons, whereas a dictionary expansion may occur less frequently.

At a high level, our solution relies on the Schwartz–Zippel lemma, i.e., if two committed polynomials f_1 and f_2 produce equal evaluations at a randomly chosen point r, then $f_1 = f_2$ with overwhelming probability. Hence, equality at a single random point suffices to certify consistency between two commitments. More concretely, the following steps take place:

1. *Setup distribution:* After the new PCS setup is generated, the new setup's prover key is sent to the server, while the verifier key is distributed among the clients and auditors. In practice, this can occur through a software update. Concretely, the key distribution remains efficient since the verifier's key is concretely small. For example, in KZH, the verifier key size stays below 1.2 MB for our concrete parameter choices to support a dictionary of size 4 billion.

2. *Re-commitment:* The server computes new polynomial commitments to the dictionary's state using the updated PCS setup.

3. *Consistency proof generation:* The server opens both the old and new polynomial commitments at a random point r, where r can be derived non-interactively using the Fiat–Shamir heuristic, akin to how the randomness required for the rand polynomials is generated.

4. *Verification by auditors:* Using both the old and new verifier keys, the auditors check that the polynomial state commitments with the old and new setup have consistent openings at the point r, based on the opening proofs provided by the server. Equality of evaluations at a random point confirms the correctness of the re-commitment.

After the four steps above, the system proceeds as usual. However, it may be necessary for clients to temporarily retain the old PCS verifier key, since it may be required to verify openings of historical values and previous rand polynomials. The precise key retention policy depends on system-level design choices rather than the transparent dictionary construction itself. For instance, if the system allows clients to query historical values only for the past year and the PCS setup is updated annually, then clients only need to store two verifier keys at any given time.

10.2 Quantum Security

One potential drawback of Aegon, when instantiated with KZH, lies in its reliance on the discrete logarithm assumption, which is not quantum-resistant. Nevertheless, we argue that this limitation does not pose an immediate concern. The main risk of using pre-quantum cryptography today is the so-called *harvest-now, decrypt-later* threat model, where an adversary stores public data—such as commitments or lookup proofs—with the intent to later exploit a quantum computer to extract secret information. However, this concern is largely irrelevant in the case of zk-KZH, which offers *statistical hiding*, rather than merely *computational hiding*. Thus, even if a future quantum adversary could compromise soundness, the privacy of past data would remain perfectly intact.

This property is particularly appealing because Aegon offers a lightweight and efficient solution to key transparency—much simpler than the currently deployed schemes based on Merkle trees. Thanks to the statistical hiding of zk-KZH, Aegon provides enduring privacy guarantees: even in a post-quantum world, historical data remains private, while today it enjoys the practical efficiency of existing cryptographic primitives.

We emphasize that Aegon is generic over multilinear PCS, with substantially improved efficiency when the PCS is homomorphic. This highlights a promising direction for future work: combining Aegon with a plausibly post-quantum homomorphic PCS, such as one based on lattice commitments [Ajt96], to obtain an efficient post-quantum instantiation.

Acknowledgment

Hossein Hafezi did this work partially whilst he was hosted by Martin Kleppmann at the University of Cambridge. We thank Kasra Abbaszadeh for his assistance in formalizing our notion of privacy in the UC model, and Michael Rosenberg and Nirvan Tyagi for valuable discussions and insightful comments.

References

[Mel+15] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. "CONIKS: Bringing Key Transparency to End Users". In: *24th USENIX Security Symposium*. USENIX Association, 2015, pp. 383–398. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara.

[Ung+15] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. "SoK: secure messaging". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 232–249.

[Cha+19] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. "SEEMless: Secure End-to-End Encrypted Messaging with less Trust". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2019, 1639–1656. ISBN: 9781450367479. DOI: 10.1145/3319535.3363202.

[Hu+21] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Yang, and Raluca Ada Popa. "Merkle2 : A Low-Latency Transparency Log System". In: May 2021, pp. 285–303. DOI: 10.1109/SP40001.2021.00088.

[Tya+22] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. "VeRSA: Verifiable Registries with Efficient Client Audits from RSA Authenticated Dictionaries". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 2793–2807. DOI: 10.1145/3548606.3560605.

[Tzi+22] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. "Transparency Dictionaries with Succinct Proofs of Correct Operation". In: *NDSS 2022*. July 2022. URL: https://www.microsoft.com/en-us/research/publication/transparency-dictionaries-with-succinct-proofs-of-correct-operation/.

[Len+24] Julia Len, Melissa Chase, Esha Ghosh, Kim Laine, and Radames Cruz Moreno. "OPTIKS: An Optimized Key Transparency System". In: *33rd USENIX Security Symposium*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4355–4372. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/len.

[Mal+23] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean Lawlor. "Parakeet: Practical Key Transparency for End-to-End Encrypted Messaging". In: Jan. 2023. DOI: 10.14722/ndss.2023.24545.

[Ros+24] Michael Rosenberg, Tushar Mopuri, Hossein Hafezi, Ian Miers, and Pratyush Mishra. "Hekaton: Horizontally-Scalable zkSNARKs Via Proof Aggregation". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2024, 929–940. ISBN: 9798400706363. DOI: 10.1145/3658644.3690282.

[Haf+26] Hossein Hafezi, Alireza Shirzad, Benedikt Bünz, and Joseph Bonneau. "IronDict: Transparent Dictionaries from Polynomial Commitments". In: *Proceedings of the 35th USENIX Security Symposium*. USENIX Association, 2026.

[Int25] Internet Engineering Task Force (IETF). *Charter: Key Transparency Working Group (KEYTRANS)*. IETF Datatracker. Charter approved, WG state: Active. 2025. URL: https://datatracker.ietf.org/doc/charter-ietf-keytrans/ (visited on 09/16/2025).

[LL23] Sean Lawlor and Kevin Lewi. *Deploying Key Transparency at WhatsApp*. Blog Post. Engineering at Meta Blog. Apr. 2023. URL: https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/.

[App23] Apple Security Engineering and Architecture. *Advancing iMessage security: iMessage Contact Key Verification*. Blog post, Apple Security Research. Accessed: 2025-09-16. 2023. URL: https://security.apple.com/blog/imessage-contact-key-verification/.

[Wha23] WhatsApp Messenger. *WhatsApp Key Transparency Overview: Technical White Paper*. Tech. rep. White paper, Version 1. Meta / WhatsApp, Aug. 2023. URL: https://www.whatsapp.com/security/WhatsApp-Key-Transparency-Whitepaper.pdf.

[LL25] Kevin Lewi and Sean Lawlor. *Auditing Key Transparency (RWC 2025)*. Talk at Real World Crypto (RWC) 2025. YouTube video, https://www.youtube.com/watch?v=JKkTZJPTh6o. 2025. URL: https://www.youtube.com/watch?v=JKkTZJPTh6o.

[Clo25] Cloudflare, Inc. *Key Transparency Dashboard*. Web service, Cloudflare Key Transparency Auditor. Dashboards showing the status of Key Transparency logs, including WhatsApp; Last updated: 2025-09-15. 2025. URL: https://dash.key-transparency.cloudflare.com/ (visited on 09/16/2025).

[CG24] Thibault Meunier Cloudflare and Mari Galicer. *Cloudflare helps verify the security of end-to-end encrypted messages by auditing Key Transparency for WhatsApp*. Blog post, Cloudflare. Accessed: 2025-09-16. 2024. URL: https://blog.cloudflare.com/key-transparency/.

[Fac25] Facebook. *facebook/akd: An implementation of an auditable key directory*. Version main. Accessed: 2025-12-04. 2025. URL: https://github.com/facebook/akd/tree/main/examples#whatsapp-key-transparency-auditor.

[Kad+25] George Kadianakis, Arantxa Zapico, Hossein Hafezi, and Benedikt Bünz. "KZH-Fold: Accountable Voting from Sublinear Accumulation". In: *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. Taipei, Taiwan: Association for Computing Machinery, 2025, 409–422. ISBN: 9798400715259. DOI: 10.1145/3719027.3744796.

[Kim+13] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. "Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure". In: *Proceedings of the 22nd International Conference on World Wide Web*. Association for Computing Machinery, 2013, 679–690. ISBN: 9781450320351. DOI: 10.1145/2488388.2488448.

[Bas+14] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. "ARPKI: Attack Resilient Public-Key Infrastructure". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2014, 382–393. ISBN: 9781450329576. DOI: 10.1145/2660267.2660298.

[Tom+19] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. "Transparency Logs via Append-Only Authenticated Dictionaries". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2019, 1299–1316. ISBN: 9781450367479. DOI: 10.1145/3319535.3345652.

[Lun+92] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. "Algebraic Methods for Interactive Proof Systems". In: *J. ACM* 39.4 (1992), pp. 859–868. DOI: 10.1145/146585.146605. URL: https://doi.org/10.1145/146585.146605.

[WCB25] Faxing Wang, Shaanan Cohney, and Joseph Bonneau. "SoK: Trusted setups for powers-of-tau strings". In: *FC '25: Proceedings of the 29th International Conference on Financial Cryptography and Data Security*. Miyakojima, Japan, 2025. URL: https://eprint.iacr.org/2025/064.pdf.

[Kap26] Gabriel Kaptchuk. *Forget-me-not Trees: Mass-scale Auditable Key Transparency from Hash Functions*. Cryptology ePrint Archive, Paper 2026/1276. 2026. URL: https://eprint.iacr.org/2026/1276.

[Syt+16] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. *Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning*. 2016. arXiv: 1503.08768 [cs.CR]. URL: https://arxiv.org/abs/1503.08768.

[Mei+20] Sarah Meiklejohn, Pavel Kalinnikov, Cindy S. Lin, Martin Hutchinson, Gary Belvin, Mariana Raykova, and Al Cutter. *Think Global, Act Local: Gossip and Client Audits in Verifiable Data Structures*. 2020. arXiv: 2011.04551 [cs.CR].

[FHP26] Edona Fasllija, Lena Heimberger, and Kevin Paul. *Signal and Ready to MINGLE: In-Band Gossip for Key Transparency Split-View Detection in E2EE Messengers*. Cryptology ePrint Archive, Paper 2026/1010. 2026. URL: https://eprint.iacr.org/2026/1010.

[Val08] Paul Valiant. "Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency". In: *Theory of Cryptography*. Ed. by Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–18. ISBN: 978-3-540-78524-8.

[Zip79] Richard Zippel. "Probabilistic algorithms for sparse polynomials". In: *EUROSAM 1979*. Springer. 1979, pp. 216–226.

[Sch80] Jack T. Schwartz. "Fast probabilistic algorithms for verification of polynomial identities". In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 701–717.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. "The Algebraic Group Model and its Applications". In: *38th Annual International Cryptology Conference*. Berlin, Heidelberg: Springer-Verlag, 2018, 33–62. ISBN: 978-3-319-96880-3. DOI: 10.1007/978-3-319-96881-0_2.

[Bün+21] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. "Proofs for Inner Pairing Products and Applications". In: 2021, pp. 65–97. DOI: 10.1007/978-3-030-92078-4_3.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: 2010, pp. 177–194. DOI: 10.1007/978-3-642-17373-8_11.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. Cryptology ePrint Archive, Paper 2017/1050. 2017. URL: https://eprint.iacr.org/2017/1050.

[Fou25] Ethereum Foundation. *KZG Ceremony*. https://ceremony.ethereum.org/. Accessed: 2025-11-27. 2025.

[con25] Wikipedia contributors. *Open addressing*. Wikipedia, The Free Encyclopedia. Accessed: 2025-11-25. 2025. URL: https://en.wikipedia.org/wiki/Open_addressing.

[FS87] Amos Fiat and Adi Shamir. "How to prove yourself: practical solutions to identification and signature problems". In: *Proceedings on Advances in Cryptology—CRYPTO '86*. Santa Barbara, California, USA: Springer-Verlag, 1987, 186–194. ISBN: 0387180478.

[MRV99] Silvio Micali, Michael Rabin, and Salil Vadhan. "Verifiable Random Functions". In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*. IEEE Computer Society, 1999, pp. 120–130. URL: https://dash.harvard.edu/handle/1/5028196.

[Zha+21] Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. "Doubly Efficient Interactive Proofs for General Arithmetic Circuits with Linear Prover Time". In: 2021, pp. 159–177. DOI: 10.1145/3460120.3484767.

[PPP25] Christodoulos Pappas, Dimitrios Papadopoulos, and Charalampos Papamanthou. "HydraProofs: Optimally Computing All Proofs in a Vector Commitment (With Applications to Efficient zkSNARKs Over Data from Multiple Users)". In: May 2025, pp. 3421–3439. DOI: 10.1109/SP61157.2025.00204.

[con22] arkworks contributors. ***arkworks*** *zkSNARK ecosystem*. 2022. URL: https://arkworks.rs.

[But14] Vitalik Buterin. "Ethereum: A next-generation smart contract and decentralized application platform". In: White Paper. 2014.

[Gol+23] Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Včelák. *Verifiable Random Functions (VRFs)*. Request for Comments RFC 9381. Internet Engineering Task Force, Aug. 2023. DOI: 10.17487/RFC9381. URL: https://www.rfc-editor.org/info/rfc9381.

[Sch90] C. P. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by Gilles Brassard. Springer New York, 1990, pp. 239–252. ISBN: 978-0-387-34805-6.

[Ajt96] M. Ajtai. "Generating hard instances of lattice problems (extended abstract)". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, 99–108. ISBN: 0897917855. DOI: 10.1145/237814.237838. URL: https://doi.org/10.1145/237814.237838.

[Chi+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". In: *Advances in Cryptology – EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I*. Zagreb, Croatia: Springer-Verlag, 2020, 738–768. ISBN: 978-3-030-45720-4. DOI: 10.1007/978-3-030-45721-1_26. URL: https://doi.org/10.1007/978-3-030-45721-1_26.

A Polynomial Commitment Scheme

Definition 2. *[Multilinear Polynomial Commitment Scheme] A* Multilinear Polynomial Commitment Scheme *is a tuple of algorithms* $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ *such that:*

- $(\mathsf{vk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(\lambda, k)$*: On input the security parameter* λ *and the number of variables* k*, this algorithm outputs a pair of verifier key (*vk*) and prover key (*pk*). This algorithm might require randomness as nondeterministic input too.*
- $(\mathsf{C}, \mathsf{aux}) \leftarrow \mathsf{Commit}(\mathsf{pk}, p(\vec{X}))$*: On input* pk *and a multilinear polynomial* $p(\vec{X})$*, it outputs a commitment* C *and auxiliary input* aux*. This algorithm might take randomness as non-deterministic input too, but for simplicity, we don't consider it here.*
- $\pi \leftarrow \mathsf{Open}(\mathsf{pk}, \mathsf{C}, \mathsf{aux}, p(\vec{X}), \vec{x})$*: On input* pk*, the commitment* C*, auxiliary input* aux*, the polynomial* $p(\vec{X})$*, and vector* $\vec{x} \in \mathbb{F}^k$*, outputs an evaluation proof* π *that* $y = p(\vec{x})$*.*
- $1/0 \leftarrow \mathsf{Verify}(\mathsf{vk}, \mathsf{C}, \vec{x}, \pi, y)$*: On input* vk*, the commitment* C*, vector of evaluations* $\vec{x}$*,* $y \in \mathbb{F}$*, and the proof of the correct evaluation, it outputs a bit indicating acceptance or rejection.*

A polynomial commitment scheme is said to be secure if it is complete and knowledge-sound as defined below:

Completeness. It captures the fact that an honest prover will always convince the verifier. Formally, for any efficient adversary $\mathcal{A}$, we have:

$$\Pr\left[\mathsf{Verify}_{\mathsf{PCS}}(\mathsf{vk}, \mathsf{C}, \vec{x}, \pi_{\mathsf{PCS}}, p(\vec{x})) = 1 \;\middle|\; \begin{array}{l} (\mathsf{vk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(\lambda, k), \\ p(\vec{X}) \leftarrow \mathcal{A}(\mathsf{srs}_{\mathsf{PCS}}), \\ (\mathsf{C}, \mathsf{aux}) \leftarrow \mathsf{Commit}(\mathsf{pk}, p(\vec{X})), \\ \pi_{\mathsf{PCS}} \leftarrow \mathsf{Open}(\mathsf{pk}, \mathsf{C}, \mathsf{aux}, p(\vec{X}), \vec{x}) \end{array}\right] = 1$$

Extractability. Captures the fact that whenever the prover provides a valid opening, it knows a valid pair $(p(\vec{X}), y)$, where $p(\vec{x}) = y$. Formally, for all PPT adversaries $\mathcal{A}$, there exists an efficient extractor $\mathcal{E}$ such that the probability of the following event is negligible:

$$\Pr\left[\begin{array}{c} \mathsf{Verify}_{\mathsf{PCS}}(\mathsf{vk}, \mathsf{C}, \vec{x}, \pi_{\mathsf{PCS}}, y) = 1 \\ \wedge\, p(\vec{x}) \neq y \end{array} \;\middle|\; \begin{array}{l} (\mathsf{vk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(\lambda, k), \\ \mathsf{C} \leftarrow \mathcal{A}(\mathsf{vk}, \mathsf{pk}), \\ p(\vec{X}) \leftarrow \mathcal{E}((\mathsf{vk}, \mathsf{pk}), \mathsf{C}, k), \\ \vec{x} \leftarrow \mathcal{A}((\mathsf{vk}, \mathsf{pk}), \mathsf{C}), \\ (y, \pi_{\mathsf{PCS}}) \leftarrow \mathcal{A}((\mathsf{vk}, \mathsf{pk}), p(\vec{X}), \vec{x}) \end{array}\right]$$

Zero-Knowledge. A PCS is said to be *zero-knowledge* if, informally, its commitment is hiding and reveals no information about the underlying polynomial, and its opening protocol constitutes a zero-knowledge argument of knowledge for the claimed polynomial evaluation. A formal definition can be found in [Chi+20].

B Transparent Security Definition

Given a dictionary state state consisting of two mappings, Index and Value, we define the following: For a label ℓ, its corresponding index is defined as $\mathsf{state.index}(\ell)$. If ℓ does not exist in the dictionary, the function returns $\perp$. Similarly, for an index i, its corresponding value is defined as $\mathsf{state.value}(i)$. If i is not a valid index, the function returns $\perp$.

B.1 Completeness

A transparent dictionary construction is said to be *complete* if *audit completeness*, *index consistency completeness* and *value consistency completeness* all hold. In the $\mathsf{Server.Update}$ algorithm, if the adversary outputs an invalid set of changes Δ_i (for example, inserting a label into the dictionary that already exists), we assume that the update algorithm treats Δ_i as empty. In this case, the update procedure neither aborts nor returns an error; instead, it simply returns the current state unchanged.

- **Audit Completeness.** For any $1 \leq m$ and any PPT adversary $\mathcal{A}$, the following holds:

$$\Pr\left[\mathsf{Auditor.VerifyInvariance}(\mathcal{C}_{m-1}, \mathcal{C}_m, \pi_m^{\mathsf{inv}}) = 1 \;\middle|\; \begin{array}{l} (\mathsf{key_{Server}}; \mathsf{key_{Auditor}}, \mathsf{key_{Client}}) \leftarrow \mathsf{Setup}(\lambda, \mu) \\ (\mathsf{state}_0, \mathcal{C}_0) \leftarrow \mathsf{Server.Init}(),\ (\Delta_1, \ldots, \Delta_m) \leftarrow \mathcal{A}(\mathsf{key_{Auditor}}, \mathsf{key_{Client}}), \\ \forall\, i \in [m-1] \,:\, (\mathsf{state}_{i+1}, \mathcal{C}_{i+1}, \pi_{i+1}^{\mathsf{inv}}) \leftarrow \mathsf{Server.Update}(\mathsf{state}_i, \Delta_{i+1}) \end{array}\right] = 1 - \mathrm{negl}$$

- **Index Consistency Completeness.** For any label ℓ and any two epochs $0 \leq s_0 < s_1$ we have that:

$$\Pr\left[\begin{array}{l} \mathsf{Client.VerifyIndexConsis}(\ell, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\mathrm{IndexConsis}}) = 1 \\ \wedge \bigwedge_{n=s_0}^{s_1} \left(\mathsf{Client.VerifyIndex}(\ell, \mathcal{C}_n, \mathsf{index}_n, \pi_{\mathrm{Index},n}) = 1 \;\wedge\; \mathsf{index}_n = \mathsf{index}_{s_0}\right) \end{array} \;\middle|\; \begin{array}{l} (\mathsf{key_{Server}}; \mathsf{key_{Auditor}}, \mathsf{key_{Client}}) \leftarrow \mathsf{Setup}(\lambda, \mu), \\ (\mathsf{state}_0, \mathcal{C}_0) \leftarrow \mathsf{Server.Init}(), \\ (\Delta_1, \ldots, \Delta_{s_1}) \leftarrow \mathcal{A}(\mathsf{key_{Auditor}}, \mathsf{key_{Client}}), \\ \forall\, i \in [1, s_1] : (\mathsf{state}_i, \mathcal{C}_i, \pi_i) \leftarrow \mathsf{Server.Update}(\mathsf{state}_{i-1}, \Delta_i), \\ \forall\, n \in [s_0, s_1] : (\mathsf{index}_n, \pi_{\mathrm{Index},n}) \leftarrow \mathsf{Server.LookupIndex}(\ell, \mathsf{state}_n) \\ \text{such that } \mathsf{state}_{s_0}.\mathsf{index}(\ell) = \cdots = \mathsf{state}_{s_1}.\mathsf{index}(\ell) \neq \perp, \\ \pi_{\mathrm{IndexConsis}} \leftarrow \mathsf{Server.IndexConsis}(\ell, \mathsf{state}_{s_0}, \mathsf{state}_{s_1}) \end{array}\right] = 1 - \mathrm{negl}$$

- **Value Consistency Completeness.** For any index index and any two epochs $0 \leq s_0 < s_1$ we have that:

$$\Pr\left[\begin{array}{l} \mathsf{Client.VerifyValueConsis}(\mathsf{index}, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\mathrm{ValueConsis}}) = 1 \\ \wedge \bigwedge_{n=s_0}^{s_1} \left(\mathsf{Client.VerifyValue}(\mathsf{index}, \mathcal{C}_n, v_n, \pi_{\mathrm{Value},n}) = 1 \;\wedge\; v_n = v_{s_0}\right) \end{array} \;\middle|\; \begin{array}{l} (\mathsf{key_{Server}}; \mathsf{key_{Auditor}}, \mathsf{key_{Client}}) \leftarrow \mathsf{Setup}(\lambda, \mu), \\ (\mathsf{state}_0, \mathcal{C}_0) \leftarrow \mathsf{Server.Init}(), \\ (\Delta_1, \ldots, \Delta_{s_1}) \leftarrow \mathcal{A}(\mathsf{key_{Auditor}}, \mathsf{key_{Client}}), \\ \forall\, i \in [1, s_1] : (\mathsf{state}_i, \mathcal{C}_i, \pi_i) \leftarrow \mathsf{Server.Update}(\mathsf{state}_{i-1}, \Delta_i), \\ \forall\, n \in [s_0, s_1] : (v_n, \pi_{\mathrm{Value},n}) \leftarrow \mathsf{Server.LookupValue}(\mathsf{index}, \mathsf{state}_n) \\ \text{such that } \mathsf{state}_{s_0}.\mathsf{value}(\mathsf{index}) = \cdots = \mathsf{state}_{s_1}.\mathsf{value}(\mathsf{index}) \neq \perp, \\ \pi_{\mathrm{ValueConsis}} \leftarrow \mathsf{Server.ValueConsis}(\mathsf{index}, \mathsf{state}_{s_0}, \mathsf{state}_{s_1}) \end{array}\right] = 1 - \mathrm{negl}$$

B.2 Soundness

A transparent dictionary is defined to be sound if it satisfies *lookup binding* and *consistency soundness*.

Lookup Binding. Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) adversary that is given the public parameters of the transparent dictionary scheme, denoted by pp. Then, the following probabilities are negligible.

- *Index Lookup Binding.*

$$\Pr\left[\begin{array}{l|l} \mathsf{Client.VerifyIndex}(\ell, \mathcal{C}, \mathsf{index}_0, \pi_0) = 1 \wedge & \mathsf{pp} := (\mathsf{key_{Server}}; \mathsf{key_{Auditor}}, \mathsf{key_{Client}}) \leftarrow \mathsf{Setup}(\lambda, \mu) \\ \mathsf{Client.VerifyIndex}(\ell, \mathcal{C}, \mathsf{index}_1, \pi_1) = 1 \wedge & (\mathcal{C}, \ell, (\mathsf{index}_0, \pi_0), (\mathsf{index}_1, \pi_1)) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \mathsf{index}_0 \neq \mathsf{index}_1 & \end{array}\right]$$

- *Value Lookup Binding.*

$$\Pr\left[\begin{array}{l|l} \mathsf{Client.VerifyValue}(\mathsf{index}, \mathcal{C}, \upsilon_0, \pi_0) = 1 \wedge & \mathsf{pp} := (\mathsf{key_{Server}}; \mathsf{key_{Auditor}}, \mathsf{key_{Client}}) \leftarrow \mathsf{Setup}(\lambda, \mu) \\ \mathsf{Client.VerifyValue}(\mathsf{index}, \mathcal{C}, \upsilon_1, \pi_1) = 1 \wedge & (\mathcal{C}, \mathsf{index}, (\upsilon_0, \pi_0), (\upsilon_1, \pi_1)) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \upsilon_0 \neq \upsilon_1 & \end{array}\right]$$

Consistency Soundness. Let $\mathcal{A}$ be a PPT adversary that is given the public parameters of the transparent dictionary scheme, denoted by pp, the following probability is negligible.

- *Index Consistency Soundness.*

$$\Pr\left[\begin{array}{l|l} \forall\, n \in [s_0, s_1 - 1] : \mathsf{Auditor.VerifyInvariance}\big(\mathcal{C}_n, \mathcal{C}_{n+1}, \pi^{\mathsf{inv}}_{n+1}\big) = 1 & \\ \mathsf{Client.VerifyIndexConsis}\big(\ell, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\mathsf{IndexConsis}}\big) = 1 & \mathsf{pp} := (\mathsf{key_{Server}}; \mathsf{key_{Auditor}}, \mathsf{key_{Client}}) \leftarrow \mathsf{Setup}(\lambda, \mu) \\ \mathsf{Client.VerifyIndex}\big(\ell, \mathcal{C}_{n_0}, \mathsf{index}_0, \pi^{\mathsf{idx}}_0\big) = 1 & \Big((s_0, s_1), \ell, \mathcal{C}_{s_0}, \{(\mathcal{C}_n, \pi^{\mathsf{inv}}_n)\}_{n \in [s_0+1, s_1]}, (n_0, n_1), \\ \mathsf{Client.VerifyIndex}\big(\ell, \mathcal{C}_{n_1}, \mathsf{index}_1, \pi^{\mathsf{idx}}_1\big) = 1 & (\mathsf{index}_0, \mathsf{index}_1), (\pi^{\mathsf{idx}}_0, \pi^{\mathsf{idx}}_1), \pi_{\mathsf{IndexConsis}}\Big) \leftarrow \mathcal{A}(\mathsf{pp}) \\ n_0, n_1 \in [s_0, s_1], \quad n_0 \neq n_1, \quad \mathsf{index}_0 \neq \mathsf{index}_1 & \end{array}\right]$$

- *Value Consistency Soundness.*

$$\Pr\left[\begin{array}{l|l} \forall\, n \in [s_0, s_1 - 1] : \mathsf{Auditor.VerifyInvariance}\big(\mathcal{C}_n, \mathcal{C}_{n+1}, \pi^{\mathsf{inv}}_{n+1}\big) = 1 & \\ \mathsf{Client.VerifyValueConsis}\big(\mathsf{index}, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\mathsf{ValueConsis}}\big) = 1 & \mathsf{pp} := (\mathsf{key_{Server}}; \mathsf{key_{Auditor}}, \mathsf{key_{Client}}) \leftarrow \mathsf{Setup}(\lambda, \mu) \\ \mathsf{Client.VerifyValue}\big(\mathsf{index}, \mathcal{C}_{n_0}, \upsilon_0, \pi^{\mathsf{val}}_0\big) = 1 & \Big((s_0, s_1), \mathsf{index}, \mathcal{C}_{s_0}, \{(\mathcal{C}_n, \pi^{\mathsf{inv}}_n)\}_{n \in [s_0+1, s_1]}, (n_0, n_1), \\ \mathsf{Client.VerifyValue}\big(\mathsf{index}, \mathcal{C}_{n_1}, \upsilon_1, \pi^{\mathsf{val}}_1\big) = 1 & (\upsilon_0, \upsilon_1), (\pi^{\mathsf{val}}_0, \pi^{\mathsf{val}}_1), \pi_{\mathsf{ValueConsis}}\Big) \leftarrow \mathcal{A}(\mathsf{pp}) \\ n_0, n_1 \in [s_0, s_1], \quad n_0 \neq n_1, \quad \upsilon_0 \neq \upsilon_1 & \end{array}\right]$$

C Formal Security Proofs

C.1 Completeness

Index Consistency Correctness. Let ℓ be an arbitrary label, and let its associated binary index at epoch s_0 be $\vec{x}_{\mathsf{ctr}_0} = \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}_0, \ell) \in \{0,1\}^{\log_2 N}$. For every $0 \le \mathsf{ctr} < \mathsf{ctr}_0$, define $\vec{x}_{\mathsf{ctr}} = \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}, \ell)$. As defined by the update algorithm of Figure 3,

$$(\mathsf{state}_i, \mathcal{C}_i, \pi_i) \leftarrow \mathsf{Server.Update}(\mathsf{state}_{i-1}, \Delta_i),$$

the server does not change the previously assigned indices, i.e., according to the index assignment algorithm (Figure 2), the evaluations $\mathsf{index}_n(\vec{x}_{\mathsf{ctr}})$ for all $\mathsf{ctr} \in [\mathsf{ctr}_0]$ remain fixed and consistent across all epochs $s_0 \le n \le s_1$. Let:

$$\mathsf{Server.IndexConsis}(\ell, \mathsf{state}_{s_0}, \mathsf{state}_{s_1}) \to \pi_{\mathrm{IndexConsis}}$$

$\pi_{\mathrm{IndexConsis}}$ denotes a proof comprising PCS evaluation proofs for the following polynomial evaluations: $\{\mathsf{rand}^{(\mathrm{I})}_{s_0}(\vec{x}_{\mathsf{ctr}})\}_{\mathsf{ctr}\in[\mathsf{ctr}_0]}$ and $\{\mathsf{rand}^{(\mathrm{I})}_{s_1}(\vec{x}_{\mathsf{ctr}})\}_{\mathsf{ctr}\in[\mathsf{ctr}_0]}$. The client verifies this proof by executing

$$\mathsf{Client.VerifyIndexConsis}(\ell, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\mathrm{IndexConsis}}),$$

which checks that, for every $\mathsf{ctr} \in [\mathsf{ctr}_0]$, the evaluations $\mathsf{rand}^{(\mathrm{I})}_{s_0}(\vec{x}_{\mathsf{ctr}})$ and $\mathsf{rand}^{(\mathrm{I})}_{s_1}(\vec{x}_{\mathsf{ctr}})$ are identical. We prove this inductively over the epochs from s_0 to s_1. Define the difference polynomial: $\Delta^{(\mathrm{I})}_n := \mathsf{index}_{n+1} - \mathsf{index}_n$. Since none of the previously assigned indices have changed, we have for all $s_0 \le n \le s_1 - 1$:

$$\mathsf{index}_{n+1}(\vec{x}_{\mathsf{ctr}}) = \mathsf{index}_n(\vec{x}_{\mathsf{ctr}}) \quad \Rightarrow \quad \Delta^{(\mathrm{I})}_n(\vec{x}_{\mathsf{ctr}}) = 0.$$

By the definition of the randomized polynomial, we have: $\mathsf{rand}^{(\mathrm{I})}_{n+1} = \mathsf{rand}^{(\mathrm{I})}_n + r^{(\mathrm{I})}_n \cdot \Delta^{(\mathrm{I})}_n$, where $r^{(\mathrm{I})}_n$ is the randomness, e.g., derived via the Fiat–Shamir heuristic. Since for $s_0 \le n \le s_1 - 1$ we have $\Delta^{(\mathrm{I})}_n(\vec{x}_{\mathsf{ctr}}) = 0$, it follows that: $\mathsf{rand}^{(\mathrm{I})}_{n+1}(\vec{x}_{\mathsf{ctr}}) = \mathsf{rand}^{(\mathrm{I})}_n(\vec{x}_{\mathsf{ctr}})$. Applying this inductively for each $s_0 \le n \le s_1 - 1$, we conclude:

$$\mathsf{rand}^{(\mathrm{I})}_{s_0}(\vec{x}_{\mathsf{ctr}}) = \mathsf{rand}^{(\mathrm{I})}_{s_0+1}(\vec{x}_{\mathsf{ctr}}) = \cdots = \mathsf{rand}^{(\mathrm{I})}_{s_1}(\vec{x}_{\mathsf{ctr}}) \quad \text{for all } \mathsf{ctr} \in [\mathsf{ctr}_0].$$

Finally, querying the index of ℓ from epoch s_0 to epoch s_1 always returns the same value $\vec{x}_{\mathsf{ctr}_0}$, since it was defined at epoch s_0 and remains unchanged until epoch s_1. Therefore, all lookups within this interval yield $\vec{x}_{\mathsf{ctr}_0}$, since the index is uniquely defined. Consequently, the check $\mathsf{Client.VerifyIndexConsis}$ succeeds.

Value Consistency Correctness. The proof here follows the same steps as the index consistency proof. Let $\vec{x}$ be an index and its associated value at epoch s_0 be v. Assume the value corresponding to $\vec{x}$ remains consistent across epochs $s_0 \le n \le s_1$. Let

$$\mathsf{Server.ValueConsis}(\vec{x}, \mathsf{state}_{s_0}, \mathsf{state}_{s_1}) \to \pi_{\mathrm{ValueConsis}}$$

$\pi_{\mathrm{ValueConsis}}$ denotes a proof comprising PCS openings for polynomial evaluations $\mathsf{rand}^{(\mathrm{V})}_{s_0}(\vec{x})$ and $\mathsf{rand}^{(\mathrm{V})}_{s_1}(\vec{x})$. The client verifies this proof by executing

$$\mathsf{Client.VerifyValueConsis}(\vec{x}, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\mathrm{ValueConsis}}),$$

which checks that the evaluations $\mathsf{rand}_{s_0}^{(\mathrm{V})}(\vec{x})$ and $\mathsf{rand}_{s_1}^{(\mathrm{V})}(\vec{x})$ are identical. By the definition of the randomized polynomial, we have: $\mathsf{rand}_{n+1}^{(\mathrm{V})} = \mathsf{rand}_{n}^{(\mathrm{V})} + r_n^{(\mathrm{V})} \cdot \Delta_n^{(\mathrm{V})}$, where $r_n^{(\mathrm{V})}$ is the randomness, e.g., derived via the Fiat–Shamir heuristic. Since for $s_0 \le n \le s_1 - 1$ we have $\Delta_n^{(\mathrm{V})}(\vec{x}) = 0$, it follows that: $\mathsf{rand}_{n+1}^{(\mathrm{V})}(\vec{x}) = \mathsf{rand}_{n}^{(\mathrm{V})}(\vec{x})$. Applying this inductively for each $s_0 \le n \le s_1 - 1$, we conclude:

$$\mathsf{rand}_{s_0}^{(\mathrm{V})}(\vec{x}) = \mathsf{rand}_{s_0+1}^{(\mathrm{V})}(\vec{x}) = \cdots = \mathsf{rand}_{s_1}^{(\mathrm{V})}(\vec{x}).$$

Finally, since the value has not changed throughout the epochs, Server.LookupValue returns the same value. Hence, the check Client.VerifyValueConsis passes.

Audit Correctness. The auditor executes $\mathsf{Auditor.VerifyInvariance}(\mathcal{C}_n, \mathcal{C}_{n+1}, \pi_{n+1})$, which verifies that the commitments to the randomized polynomials $\mathsf{rand}_{n+1}^{(\mathrm{I})}$ and $\mathsf{rand}_{n+1}^{(\mathrm{V})}$ have been computed correctly, i.e., that they satisfy the homomorphic recursion of Equation (1). Assuming the server runs $(\mathsf{state}_i, \mathcal{C}_i, \pi_i) \leftarrow \mathsf{Server.Update}(\mathsf{state}_{i-1}, \Delta_i)$, it is easy to verify that these checks pass, since the auditor simply recomputes the commitments.

C.2 Soundness

C.2.1 Binding Property.

Index Lookup Binding. According to the index verification algorithm, the client receives an identifier ctr_0 and the opening of the polynomial index_n at the points $\{\vec{x}_{\mathsf{ctr}}\}_{\mathsf{ctr} \in [\mathsf{ctr}_0]}$ such that $\vec{x}_{\mathsf{ctr}} := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}, \ell)$. The client then performs the following checks:

- For all $0 \le \mathsf{ctr} < \mathsf{ctr}_0$, it verifies that $\mathsf{index}_n(\vec{x}_{\mathsf{ctr}}) \neq \mathcal{H}^{\mathbb{F}}(\ell)$.
- It also checks that $\mathsf{index}_n(\vec{x}_{\mathsf{ctr}_0}) = \mathcal{H}^{\mathbb{F}}(\ell)$.

Assuming the PCS is sound—i.e., the adversarial server cannot open the same point to two different values—and given that the hash function $\mathcal{H}^{\mathrm{bits}}$ is deterministic, the client cannot accept two different indices $\vec{x}_1 := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}_1, \ell)$ and $\vec{x}_2 := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}_2, \ell)$ (with $\vec{x}_1 \neq \vec{x}_2$) for the same label ℓ. Without loss of generality, assume $\mathsf{ctr}_1 < \mathsf{ctr}_2$. For the client to accept index $\vec{x}_2$, it must verify that $\mathsf{index}_n(\vec{x}_1) \neq \mathcal{H}^{\mathbb{F}}(\ell)$; however, accepting index $\vec{x}_1$ requires verifying that $\mathsf{index}_n(\vec{x}_1) = \mathcal{H}^{\mathbb{F}}(\ell)$. This is a contradiction, violating the binding property of the underlying PCS.

Value Lookup Binding. Value lookup binding follows directly from the binding property of the underlying PCS. Let index be the index associated with a given label, and recall that Client.VerifyValue verifies a PCS opening of value_n at index to the hashed value $\mathcal{H}^{\mathbb{F}}(v)$. Suppose an adversary can open the dictionary commitment $\mathcal{C}_n$ at index to two distinct values, producing valid openings $(\mathcal{H}^{\mathbb{F}}(v_0), \pi_0)$ and $(\mathcal{H}^{\mathbb{F}}(v_1), \pi_1)$ with $v_0 \neq v_1$. This leads to a contradiction with either the binding property of the PCS or the collision resistance of the hash function $\mathcal{H}^{\mathbb{F}}$:

- If $\mathcal{H}^{\mathbb{F}}(v_0) = \mathcal{H}^{\mathbb{F}}(v_1)$ while $v_0 \neq v_1$, this violates the collision-resistance property of $\mathcal{H}^{\mathbb{F}}$.
- If $\mathcal{H}^{\mathbb{F}}(v_0) \neq \mathcal{H}^{\mathbb{F}}(v_1)$, then the binding property of the PCS is violated, since the commitment $\mathsf{com}(\mathsf{value}_n)$ is opened at the same evaluation point index to two distinct values.

C.2.2 Consistency Soundness

We restate Lemma 1 of IronDict in the notation used here and refer the reader to their work for the proof.

Lemma 2. *Fix an evaluation point $\vec{x}$ and two epochs $s_0 < s_1$. For each $n \in [s_0, s_1 - 1]$ let $a_n := \Delta_n(\vec{x}) \in \mathbb{F}$, and let the challenge $r_n \in \mathbb{F}$ be sampled uniformly and independently of a_n after a_n is fixed (in the random-oracle model, $r_n = \mathcal{O}(r_{n-1}, \mathrm{com}(\cdot))$ is uniform conditioned on the hashed commitments, which are fixed before r_n is revealed). If $a_n \neq 0$ for at least one $n \in [s_0, s_1 - 1]$, then*

$$\Pr\left[\sum_{n=s_0}^{s_1-1} a_n\, r_n = 0\right] \;\leq\; \frac{1}{|\mathbb{F}|}.$$

Applying Lemma 2 to the randomized polynomials, which are recursively defined as $\mathsf{rand}_{n+1} = \mathsf{rand}_n + r_n \cdot \Delta_n$, we obtain $\mathsf{rand}_{s_1}(\vec{x}) - \mathsf{rand}_{s_0}(\vec{x}) = \sum_{n=s_0}^{s_1-1} r_n \cdot \Delta_n(\vec{x})$. The server receives each challenge r_n from an auditor (in practice via Fiat–Shamir) only after publishing Δ_n; the auditor's recomputation of the recursion fixes the order of operations required by the lemma. Hence, if $\mathsf{rand}_{s_0}(\vec{x}) = \mathsf{rand}_{s_1}(\vec{x})$, then with overwhelming probability

$$\Delta_{s_0}(\vec{x}) = \Delta_{s_0+1}(\vec{x}) = \cdots = \Delta_{s_1-1}(\vec{x}) = 0.$$

Index Consistency Soundness. Assume an adversary $\mathcal{A}$ outputs

$$\Big((s_0, s_1), \ell, \mathcal{C}_{s_0}, \{(\mathcal{C}_n, \pi_n^{\mathsf{inv}})\}_{n\in[s_0+1,s_1]}, (n_0, n_1), (\mathsf{index}_0, \mathsf{index}_1), (\pi_0^{\mathsf{idx}}, \pi_1^{\mathsf{idx}}), \pi_{\mathsf{IndexConsis}}\Big)$$

such that $n_0, n_1 \in [s_0, s_1]$, $n_0 \neq n_1$, $\mathsf{index}_0 \neq \mathsf{index}_1$, and we have:

1. *Invariance verification (for all steps):* For every $n \in [s_0, s_1 - 1]$,

$$\mathsf{Auditor.VerifyInvariance}\big(\mathcal{C}_n,\ \mathcal{C}_{n+1},\ \pi_{n+1}^{\mathsf{inv}}\big) = 1$$

2. *Index consistency from epoch s_0 to s_1:*

$$\mathsf{Client.VerifyIndexConsis}\big(\ell,\ \mathcal{C}_{s_0},\ \mathcal{C}_{s_1},\ \pi_{\mathsf{IndexConsis}}\big) = 1$$

3. *Index verification at epochs n_0 and n_1:*

$$\mathsf{Client.VerifyIndex}\big(\ell,\ \mathcal{C}_{n_0},\ \mathsf{index}_0,\ \pi_0^{\mathsf{idx}}\big) = 1$$

$$\mathsf{Client.VerifyIndex}\big(\ell,\ \mathcal{C}_{n_1},\ \mathsf{index}_1,\ \pi_1^{\mathsf{idx}}\big) = 1$$

Let the index of ℓ at epoch s_0 be $\mathsf{index}_0 = \vec{x}_{\mathsf{ctr}_0} = \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}_0, \ell)$. For all $0 \leq \mathsf{ctr} < \mathsf{ctr}_0$, define $\vec{x}_{\mathsf{ctr}} = \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}, \ell)$. By definition of the index, ctr_0 is the smallest counter such that:

$$\mathsf{index}_{s_0}(\mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}_0, \ell)) = \mathcal{H}^{\mathbb{F}}(\ell),\ \forall\, \mathsf{ctr} < \mathsf{ctr}_0 \ :\ \mathsf{index}_{s_0}(\mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}, \ell)) \neq \mathcal{H}^{\mathbb{F}}(\ell).$$

By (2) and the binding property of the PCS, the openings underlying $\pi_{\mathsf{IndexConsis}}$ fix the evaluations $\mathsf{rand}_{s_0}^{(\mathrm{I})}(\vec{x}_{\mathsf{ctr}}) = \mathsf{rand}_{s_1}^{(\mathrm{I})}(\vec{x}_{\mathsf{ctr}})$ for all $\mathsf{ctr} \in [\mathsf{ctr}_0]$; by (1), the auditor checks enforce the recursion of Equation (1) on the commitments across every epoch in $[s_0, s_1]$. Hence Lemma 2 applies, and, except with negligible probability,

$$\mathsf{index}_{s_0}(\vec{x}_{\mathsf{ctr}}) = \mathsf{index}_{s_0+1}(\vec{x}_{\mathsf{ctr}}) = \cdots = \mathsf{index}_{s_1}(\vec{x}_{\mathsf{ctr}}) \quad \text{for all } \mathsf{ctr} \in [\mathsf{ctr}_0].$$

Now let $\mathsf{index}_1 = \vec{x}_{\mathsf{ctr}'_0} = \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}'_0, \ell)$. We argue that, except with negligible probability, the verification

$$\mathsf{Client.VerifyIndex}\big(\ell,\ \mathcal{C}_{n_1},\ \mathsf{index}_1,\ \pi_1^{\mathsf{idx}}\big) = 1$$

cannot hold. Since $\mathsf{index}_0 \neq \mathsf{index}_1$, we may assume without loss of generality that $\mathsf{ctr}_0 < \mathsf{ctr}'_0$, so ctr_0 lies in the range $\mathsf{ctr} < \mathsf{ctr}'_0$ that $\mathsf{Client.VerifyIndex}$ scans. The verification procedure checks that for all $\mathsf{ctr} < \mathsf{ctr}'_0$, $\mathsf{index}_{s_1}(\vec{x}_{\mathsf{ctr}}) \neq \mathcal{H}^{\mathbb{F}}(\ell)$. In particular, at $\mathsf{ctr} = \mathsf{ctr}_0$, we have $\mathsf{index}_{s_1}(\vec{x}_{\mathsf{ctr}_0}) = \mathcal{H}^{\mathbb{F}}(\ell)$ by the consistency established above, leading to a contradiction. Therefore, if the verification succeeds, it must be that either the soundness of the PCS is broken or the collision resistance of the hash function $\mathcal{H}^{\mathbb{F}}$ is violated, both of which occur only with negligible probability.

Value Consistency Soundness. The argument is analogous to index consistency soundness. Assume an adversary $\mathcal{A}$ outputs

$$\Big((s_0, s_1), \mathsf{index}, \mathcal{C}_{s_0}, \{(\mathcal{C}_n, \pi_n^{\mathsf{inv}})\}_{n\in[s_0+1,s_1]}, (n_0, n_1), (\upsilon_0, \upsilon_1), (\pi_0^{\mathsf{val}}, \pi_1^{\mathsf{val}}), \pi_{\mathsf{ValueConsis}}\Big)$$

such that $n_0, n_1 \in [s_0, s_1]$, $n_0 \neq n_1$, $\upsilon_0 \neq \upsilon_1$ and we have:

1. *Invariance verification (for all steps):* For every $n \in [s_0, s_1 - 1]$,

$$\mathsf{Auditor.VerifyInvariance}\big(\mathcal{C}_n, \mathcal{C}_{n+1}, \pi_{n+1}^{\mathsf{inv}}\big) = 1$$

2. *Value consistency from epoch s_0 to s_1:*

$$\mathsf{Client.VerifyValueConsis}\big(\mathsf{index}, \mathcal{C}_{s_0}, \mathcal{C}_{s_1}, \pi_{\mathsf{ValueConsis}}\big) = 1$$

3. *Value verification at epochs n_0 and n_1:*

$$\mathsf{Client.VerifyValue}\big(\mathsf{index}, \mathcal{C}_{n_0}, \upsilon_0, \pi_0^{\mathsf{val}}\big) = 1$$

$$\mathsf{Client.VerifyValue}\big(\mathsf{index}, \mathcal{C}_{n_1}, \upsilon_1, \pi_1^{\mathsf{val}}\big) = 1$$

By (1) and (2), Lemma 2 guarantees—except with negligible probability (i.e., unless the soundness of the PCS is violated)—that

$$\mathsf{value}_{s_0}(\mathsf{index}) = \mathsf{value}_{s_0+1}(\mathsf{index}) = \cdots = \mathsf{value}_{s_1}(\mathsf{index}). \quad (\blacksquare)$$

However, this contradicts (3), which states that the value polynomials open to different values at epochs n_0 and n_1. This directly violates ($\blacksquare$). Therefore, such an event can only occur if either the soundness of the PCS is broken or a collision in the hash function occurs, both of which happen only with negligible probability.

D Privacy

As noted earlier, IronDict offers no formal privacy definition. Instead, it argues for privacy informally by requiring its underlying cryptographic primitives—PCS openings and sumcheck proofs—to be zero-knowledge. This is not sufficient: zero-knowledge primitives alone do not imply that the surrounding protocol leaks no information beyond what is intended.

To see why, consider the index opening procedure. When the honest server opens the index $\vec{x}_{\mathsf{ctr}_0} := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}_0, \ell)$ associated with a label ℓ and counter ctr_0, the protocol forces it to additionally reveal every intermediate evaluation

$$\mathsf{index}(\vec{x}_{\mathsf{ctr}}) \quad \text{for all } \mathsf{ctr} \in [\mathsf{ctr}_0], \qquad \text{where } \vec{x}_{\mathsf{ctr}} := \mathcal{H}^{\mathrm{bits}}(\mathsf{ctr}, \ell).$$

These intermediate values constitute strictly more information than the client's own index $\vec{x}_{\mathsf{ctr}_0}$: they include VRF outputs at points the client never queried. This extra leakage is exploitable. Suppose a client knows some other label ℓ_0 together with its VRF evaluation $\mathcal{H}^{\mathbb{F}}(\ell_0)$. If, while observing an opening, the client notices that some intermediate value coincides with $\mathcal{H}^{\mathbb{F}}(\ell_0)$—that is, $\mathsf{index}(\vec{x}_i) = \mathcal{H}^{\mathbb{F}}(\ell_0)$ for some $i \in [\mathsf{ctr}_0]$ —then the client immediately learns the index associated with ℓ_0, information that should have remained hidden. Because the system contains only a polynomial number of labels, such a collision occurs with non-negligible probability. These observations show that zero-knowledge primitives are not a substitute for a privacy analysis of the protocol as a whole, and motivate the need for a rigorous, formal treatment of privacy.

Definition 3 (Privacy)**.** Let $\mathcal{F}$ be an ideal functionality. A transparent dictionary protocol π is *private with respect to* $\mathcal{F}$ if for every PPT adversary $\mathcal{A}$ corrupting a subset of clients, there exists a PPT simulator $\mathcal{S}$ such that for every PPT environment $\mathcal{Z}$,

$$\left\{ \mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda) \right\}_{\lambda\in\mathbb{N}} \approx_c \left\{ \mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) \right\}_{\lambda\in\mathbb{N}},$$

where each ensemble denotes the bit output by $\mathcal{Z}$ at the end of the corresponding execution, defined as follows.

- In $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$, the environment $\mathcal{Z}$ adaptively (i) instructs the honest server Server to initialize and to apply per-epoch updates Δ_i, and (ii) directs $\mathcal{A}$, which controls one or more clients. At each epoch Server runs the protocol π honestly, and $\mathcal{A}$ is given the public per-epoch output—the commitment $\mathcal{C}_i$ together with its invariance/update proof π_i—as well as every response returned to a corrupted client (index and value lookups and consistency proofs). The adversary $\mathcal{A}$ may deviate arbitrarily on behalf of corrupted clients. At the end $\mathcal{Z}$ outputs a bit.
- In $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$, the same $\mathcal{Z}$ drives an honest server Server that forwards its instructions to $\mathcal{F}$ in place of running π. The functionality $\mathcal{F}$ maintains the dictionary internally and, at each interface, computes only the information it designates as leakage. It hands this leakage to $\mathcal{S}$, receives back the corresponding wire objects (the per-epoch commitment and proof, or the client-facing response), and forwards them to $\mathcal{A}$. Thus every byte seen by $\mathcal{A}$ in the ideal world is produced by $\mathcal{S}$ from leakage alone. At the end $\mathcal{Z}$ outputs a bit.

Throughout, $\mathcal{F}$ never manufactures commitments or proofs itself; it is parameterised by the simulator $\mathcal{S}$ of Definition 3, exposing the interfaces $\mathcal{S}_{\mathsf{com}}, \mathcal{S}_{\mathsf{upd}}, \mathcal{S}_{\mathsf{lk}}, \mathcal{S}_{\mathsf{cp}}$ that produce, respectively, an epoch commitment, an update/invariance proof, a value-lookup proof, and a value-consistency proof. The simulator is stateful and, in the random-oracle model, may program the oracle. For the *definitions* below we merely posit such an $\mathcal{S}$, assuming it is *correct*—every object it returns is accepted by the

corresponding verification algorithm—and *consistent*—repeated queries are answered with identical bytes, and overlapping value-consistency queries on the same index are answered with agreeing evaluations. Section D.3 exhibits a concrete $\mathcal{S}$ for Aegon built from the zero-knowledge simulators of the underlying PCS and Σ-protocol.

D.1 An Idealized and Maximally Strong Definition of Privacy

We begin with an idealized, maximally strong privacy definition in the UC model: updates, lookup proofs, and consistency proofs must reveal *nothing whatsoever* beyond the value returned to the querying client and the consistency bit. We note that the bulletin board, which is load-bearing in the soundness model of Section 3 as the anti-split-view primitive, plays no role here: for privacy it is immaterial how the public per-epoch output is published, only what it contains, so we dispense with it and simply hand each public output directly to the adversary.

Definition 4 ($\mathcal{F}_{\mathsf{priv}}$)**.** The functionality interacts with a server Server, an adversarial client $\mathcal{A}$, and the simulator $\mathcal{S}$. It maintains an internal dictionary Dict_i and internal mappings CS, LK, and CP. It is parameterised by the number of epochs n and by the simulator $\mathcal{S}$.

Setup. On receiving Server.init from Server:

- Initialise $\mathrm{Dict}_0 := \emptyset$.
- Obtain $\mathsf{C}_0 \leftarrow \mathcal{S}_{\mathsf{com}}()$, set $\mathsf{CS}[0] := (\mathsf{C}_0, \bot)$, and output C_0 publicly.

Update. On receiving $(\mathsf{update}, i, \Delta_i)$ from Server:

- Compute Dict_i by applying Δ_i to Dict_{i-1}.
- Obtain $\mathsf{C}_i \leftarrow \mathcal{S}_{\mathsf{com}}()$ and $\pi_i \leftarrow \mathcal{S}_{\mathsf{upd}}()$ (no leakage about Δ_i), set $\mathsf{CS}[i] := (\mathsf{C}_i, \pi_i)$, and output (C_i, π_i) publicly.

Lookup. On input $(\mathsf{lookup}, \ell, i)$ from $\mathcal{A}$, where $0 \le i \le n$:

- Return $\bot$ if $\ell \notin \mathrm{Dict}_i$; otherwise, let $v := \mathrm{Dict}_i[\ell]$.
- If $\mathsf{LK}[(\ell, i)]$ is defined, return it to $\mathcal{A}$; otherwise obtain $\pi \leftarrow \mathcal{S}_{\mathsf{lk}}(\mathsf{C}_i, v)$, store $\mathsf{LK}[(\ell, i)] := (v, \pi)$, and return (v, π) to $\mathcal{A}$.

Proof of Consistency. On input $(\mathsf{consistency}, \ell, i, j)$ from $\mathcal{A}$, where $0 \le i < j \le n$:

- Return $\bot$ if $\ell \notin \mathrm{Dict}_i$; otherwise, set bit $b := 1$ if $\mathrm{Dict}_i[\ell] = \mathrm{Dict}_{i+1}[\ell] = \cdots = \mathrm{Dict}_j[\ell]$, and $b := 0$ otherwise.
- If $\mathsf{CP}[(\ell, i, j)]$ is defined, return it to $\mathcal{A}$. Otherwise, if $b = 1$, obtain $\pi \leftarrow \mathcal{S}_{\mathsf{cp}}(\mathsf{C}_i, \mathsf{C}_j)$, store $\mathsf{CP}[(\ell, i, j)] := (1, \pi)$, and return $(1, \pi)$ to $\mathcal{A}$; if $b = 0$, store $\mathsf{CP}[(\ell, i, j)] := (0, \bot)$ and return $(0, \bot)$ to $\mathcal{A}$.

D.2 A Relaxed Idealized Definition of Privacy

Definition 4 is too strong to be useful: no prior scheme satisfies it, and Aegon is no exception, as every existing construction leaks at least an upper bound N on the dictionary size. Before presenting a relaxed definition that Aegon provably satisfies, we first identify the unavoidable sources of leakage in our construction and describe how the functionality is structured to accommodate them. We emphasize, nevertheless, that the relaxed notion remains a form of *strong privacy* in the sense that matters: lookup proofs and consistency proofs for a given value reveal nothing about any other value.

Sources of necessary leakage. Three properties of Aegon force some information to be revealed:

- *An upper bound N on the dictionary size.* Concrete PCS constructions and the open-addressing layout both require a fixed capacity, so N is necessarily public.
- *A structured reference string (SRS).* The KZH instantiation of Aegon requires a trusted setup, and the resulting SRS is part of the public parameters.
- *Label-to-index mappings.* As discussed earlier, the privacy definition of IronDict fails precisely because open addressing can expose the index assigned to other labels. This leakage is inherent to open addressing. However, it is confined to the label-to-index mapping—an internal bookkeeping layer—and reveals nothing about the values themselves.

Resulting structure of the functionality. Guided by these observations, we parameterize the ideal functionality by the upper bound N and the SRS. To align with our notion of a transparent dictionary, we further split the dictionary into two sub-dictionaries: an *index dictionary* mapping each label to an internal index, and a *value dictionary* mapping each index to its value. The label-to-index mapping is treated as public, and our privacy guarantees concern only the value dictionary. This is sound because indices are static: they reveal neither the underlying values nor the pattern of value updates. In practice, and following prior work, labels are assigned indices via a VRF, and rate limiting is used to bound how much of the mapping an adversary can learn. Accordingly, the index layer is leaked to $\mathcal{S}$ in full (it is public and deterministically reproducible), while value-related objects are produced by $\mathcal{S}$ from minimal leakage.

Definition 5 ($\mathcal{F}_{\mathsf{Aegon}}$)**.** The functionality interacts with a server Server, an adversarial client $\mathcal{A}$, and the simulator $\mathcal{S}$. It maintains two internal dictionaries, $\mathsf{Dict}_{i,\mathsf{index}}$ and $\mathsf{Dict}_{i,\mathsf{value}}$, where $\mathsf{Dict}_{i,\mathsf{index}}$ is public. In addition, the functionality stores the internal mappings CS, LK, and CP. The functionality is parameterised by the number of epochs n, the dictionary capacity N, a structured reference string srs, an index-assignment rule AssignIdx that, given the current public index map and a new label, returns an index in $\{0, \ldots, N-1\}$ such that the resulting label-to-index map is injective and static (an assigned label is never reassigned), and the simulator $\mathcal{S}$.

Setup. On receiving Server.init from Server:

- Initialise $\mathrm{Dict}_0 := \emptyset$.
- Obtain $\mathsf{C}_0 \leftarrow \mathcal{S}_{\mathsf{com}}(\mathsf{Dict}_{0,\mathsf{index}})$, set $\mathsf{CS}[0] := (\mathsf{C}_0, \perp)$, and output C_0 publicly.

Update. On receiving $(\mathsf{update}, i, \Delta_i)$ from Server:

- Compute Dict_i by applying Δ_i to Dict_{i-1}. During this process, whenever a previously unseen label ℓ is encountered, assign it the index $\mathsf{index} \leftarrow \mathsf{AssignIdx}(\mathsf{Dict}_{i,\mathsf{index}}, \ell)$ and record $\mathsf{Dict}_{i,\mathsf{index}}[\ell] := \mathsf{index}$, where $\mathsf{Dict}_{i,\mathsf{index}}$ denotes the index map updated with all assignments made so far in this epoch.
- Obtain $\mathsf{C}_i \leftarrow \mathcal{S}_{\mathsf{com}}(\mathsf{C}_{i-1}, \mathsf{Dict}_{i,\mathsf{index}})$ and $\pi_i \leftarrow \mathcal{S}_{\mathsf{upd}}(\mathsf{C}_{i-1}, \mathsf{C}_i)$, set $\mathsf{CS}[i] := (\mathsf{C}_i, \pi_i)$, and output (C_i, π_i) publicly.

Lookup Index. On receiving $(\mathsf{lookup\ index}, \ell, i)$ from $\mathcal{A}$, where $0 \le i \le n$: returns $\mathsf{Dict}_{i,\mathsf{index}}$ to $\mathcal{A}$, who locally computes $\mathsf{Dict}_{i,\mathsf{index}}[\ell]$ and (since the index layer is public) reconstructs the corresponding index opening itself.

Lookup Value. On receiving $(\mathsf{lookup\ value}, \mathsf{index}, i)$ from $\mathcal{A}$, where $0 \le i \le n$:

- Return $\perp$ if $\mathsf{index} \notin \mathrm{Dict}_{i,\mathsf{value}}$; otherwise, let $v := \mathrm{Dict}_{i,\mathsf{value}}[\mathsf{index}]$.
- If $\mathsf{LK}[(\mathsf{index}, i)]$ is defined, return it to $\mathcal{A}$; otherwise obtain $\pi \leftarrow \mathcal{S}_{\mathsf{lk}}(\mathsf{C}_i, \mathsf{index}, i, v)$, store $\mathsf{LK}[(\mathsf{index}, i)] := (v, \pi)$, and return $(v,\ \pi)$ to $\mathcal{A}$.

Index Consistency. On receiving $(\mathsf{index\ consistency}, \ell, i, j)$ from $\mathcal{A}$, where $0 \le i < j \le n$: returns $\mathsf{Dict}_{i,\mathsf{index}}, \mathsf{Dict}_{i+1,\mathsf{index}}, \ldots, \mathsf{Dict}_{j,\mathsf{index}}$ to $\mathcal{A}$, who locally performs the consistency checks.

Value Consistency. On receiving $(\mathsf{value\ consistency}, \mathsf{index}, i, j)$ from $\mathcal{A}$, where $0 \le i < j \le n$:

- Return $\perp$ if $\mathsf{index} \notin \mathrm{Dict}_{i,\mathsf{value}}$; otherwise, set bit $b := 1$ if

$$\mathrm{Dict}_{i,\mathsf{value}}[\mathsf{index}] = \mathrm{Dict}_{i+1,\mathsf{value}}[\mathsf{index}] = \cdots = \mathrm{Dict}_{j,\mathsf{value}}[\mathsf{index}],$$

  and $b := 0$ otherwise.
- If $\mathsf{CP}[(\mathsf{index}, i, j)]$ is defined, return it to $\mathcal{A}$; otherwise if $b = 1$ obtain

$$\pi \leftarrow \mathcal{S}_{\mathsf{cp}}(\{\mathsf{C}_k\}_{i \le k \le j}, \mathsf{index}, i, j),$$

  store $\mathsf{CP}[(\mathsf{index}, i, j)] := (1, \pi)$, and return $(1,\ \pi)$ to $\mathcal{A}$; if $b = 0$, store $\mathsf{CP}[(\mathsf{index}, i, j)] := (0, \perp)$ and return $(0, \perp)$ to $\mathcal{A}$.

D.3 Privacy Proof for Aegon

We prove that the protocol π specified in Figure 3 and Figure 4 is private with respect to $\mathcal{F}_{\mathsf{Aegon}}$ of Definition 5, by exhibiting a concrete simulator $\mathcal{S}$ whose interfaces $\mathcal{S}_{\mathsf{com}}, \mathcal{S}_{\mathsf{upd}}, \mathcal{S}_{\mathsf{lk}}, \mathcal{S}_{\mathsf{cp}}$ are built from the zero-knowledge simulators $\mathsf{Sim}_{\mathsf{Commit}}, \mathsf{Sim}_{\mathsf{Open}}$ of the underlying PCS and the simulator Sim_{Σ} of the Σ-protocol.

Assumptions on the Construction. The proof relies on the following properties of π.

- On repeated queries, the server caches the proofs and returns identical bytes.
- The openings of value and $\mathsf{rand}^{(\mathrm{V})}$ are zero-knowledge, while the openings of index and $\mathsf{rand}^{(\mathrm{I})}$ are not; the latter are trivially reproducible from the public index map.
- The commitments to the value-related polynomials value and $\mathsf{rand}^{(\mathrm{V})}$ are (statistically) hiding.
- The challenges $r_i^{(\mathrm{I})}, r_i^{(\mathrm{V})}$ are derived by hashing commitments via a random oracle.
- *Update Proofs.* The update proof π_i is empty under a homomorphic PCS. For the zero-knowledge KZH commitments we use, π_i instead consists of the simple Σ-protocol of Section 6.4 that checks the homomorphic identities between blinded commitments. In the general non-homomorphic case, π_i is a fingerprinting proof. For simplicity, we only consider π_i to be a Σ-protocol proof for our concrete instantiation with KZH.

The Simulator. We construct a PPT simulator $\mathcal{S}$ that, invoked by $\mathcal{F}_{\mathsf{Aegon}}$ on the leakage of each interface, reproduces the view of $\mathcal{A}$ (Figure 12). Let $\mathsf{Sim}_{\mathsf{Commit}}$ and $\mathsf{Sim}_{\mathsf{Open}}$ be the commitment and opening simulators of the PCS, and Sim_{Σ} the simulator of the Σ-protocol that checks a homomorphic relation between two blinded zk-KZH commitments; all three exist by the zero-knowledge property of the underlying primitives. The value-related objects are produced through these simulators, whereas the index-related objects are reproduced verbatim from the public label–index map and require no simulation. The simulator is stateful: it stores the commitments it has produced and the evaluations it has assigned to each index, so that the openings it returns verify against the stored commitments and overlapping consistency queries on the same index are answered with a common evaluation.

Hybrids. Let $\mathsf{H}_0 := \mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda)$ and $\mathsf{H}_3 := \mathsf{IDEAL}_{\mathcal{F}_{\mathsf{Aegon}},\mathcal{S},\mathcal{Z}}(\lambda)$. We interpolate between them, changing only how bytes are produced at each step and never the dictionary logic. Throughout, q_{lk} and q_{cp} denote the number of value lookup and value consistency queries, and n the number of epochs.

H_0: **Real Execution.** The honest Server runs π: real commitments, real openings, and real Fiat–Shamir challenges.

H_1: **Route Bookkeeping Through $\mathcal{F}$.** Server still produces every byte itself, but the decisions of which value, which index, and whether a value is consistent are answered from $\mathcal{F}_{\mathsf{Aegon}}$'s tables, and indices are assigned by $\mathsf{AssignIdx}$, which in $\mathcal{F}$ is the same VRF open-addressing rule (Figure 2) used by the real Server. Thus the index map is identically distributed and the value and consistency answers are functions of the same dictionary contents.

$$\implies \mathsf{H}_0 \equiv \mathsf{H}_1$$

H_2: **Simulate Epoch Outputs.** We define n sub-hybrids $\mathsf{H}_1 = \mathsf{H}_2^{(0)}, \mathsf{H}_2^{(1)}, \ldots, \mathsf{H}_2^{(n)} = \mathsf{H}_2$, where $\mathsf{H}_2^{(m)}$ produces the per-epoch output of the first m epochs via $\mathcal{S}_{\mathsf{com}}/\mathcal{S}_{\mathsf{upd}}$ and runs the remaining $n-m$ honestly. In the step from $\mathsf{H}_2^{(m-1)}$ to $\mathsf{H}_2^{(m)}$, the output of epoch m is replaced as follows: the value-related commitments $\mathsf{C}_m^{\mathsf{value}}, \mathsf{C}_m^{\mathsf{rand}^{(\mathrm{V})}}$ are produced by $\mathsf{Sim}_{\mathsf{Commit}}$; the random oracle is reprogrammed so that $r_m^{(\mathrm{I})}, r_m^{(\mathrm{V})}$ are the Fiat–Shamir challenges consistent with C_m

Simulator interfaces. $\mathsf{Sim}_{\mathsf{Commit}}() \to \mathsf{C}$ (commitments are statistically hiding, so no input); $\mathsf{Sim}_{\mathsf{Open}}(\mathsf{C}, \vec{x}, y) \to \pi$ (commitment, evaluation point, claimed value); $\mathsf{Sim}_{\Sigma}() \to \pi_{\Sigma}$ (the Σ-protocol is information-theoretically zero-knowledge, so no input).

- $\mathcal{S}_{\mathsf{com}}/\mathcal{S}_{\mathsf{upd}}$ **(Update**, given C_{i-1} and the public index map $\mathsf{Dict}_{i,\mathsf{index}}$**)**:
  1. $\mathsf{C}_i^{\mathsf{index}} \leftarrow \mathsf{Commit}(\mathsf{index}_i)$ from the public index map (no zero-knowledge); $\mathsf{C}_i^{\mathsf{value}} \leftarrow \mathsf{Sim}_{\mathsf{Commit}}()$.
  2. $r_i^{(\mathrm{I})}, r_i^{(\mathrm{V})} \xleftarrow{\$} \mathbb{F}$, program $\mathcal{O}$ so that $r_i^{(\mathrm{I})} = \mathcal{O}(r_{i-1}^{(\mathrm{I})}, \mathsf{C}_i^{\mathsf{index}})$ and $r_i^{(\mathrm{V})} = \mathcal{O}(r_{i-1}^{(\mathrm{V})}, \mathsf{C}_i^{\mathsf{value}})$.
  3. $\mathsf{C}_i^{\mathsf{rand}^{(\mathrm{I})}} \leftarrow \mathsf{Commit}(\mathsf{rand}_i^{(\mathrm{I})})$ (no zero-knowledge); $\mathsf{C}_i^{\mathsf{rand}^{(\mathrm{V})}} \leftarrow \mathsf{Sim}_{\mathsf{Commit}}()$.
  4. $\pi_{0,i} \leftarrow \mathsf{Sim}_{\Sigma}()$ and $\pi_{1,i} \leftarrow \mathsf{Sim}_{\Sigma}()$, obtaining $\pi_i := (\pi_{0,i}, \pi_{1,i})$, attesting the homomorphic relations

$$\mathsf{C}(\mathsf{rand}_i^{(\mathrm{V})}) = \mathsf{C}(\mathsf{rand}_{i-1}^{(\mathrm{V})}) + r_i^{(\mathrm{V})} \times \big(\mathsf{C}(\mathsf{value}_i) - \mathsf{C}(\mathsf{value}_{i-1})\big),$$

$$\mathsf{C}(\mathsf{rand}_i^{(\mathrm{I})}) = \mathsf{C}(\mathsf{rand}_{i-1}^{(\mathrm{I})}) + r_i^{(\mathrm{I})} \times \big(\mathsf{C}(\mathsf{index}_i) - \mathsf{C}(\mathsf{index}_{i-1})\big).$$

  Set $\mathsf{C}_i := (\mathsf{C}_i^{\mathsf{index}}, \mathsf{C}_i^{\mathsf{value}}, \mathsf{C}_i^{\mathsf{rand}^{(\mathrm{I})}}, \mathsf{C}_i^{\mathsf{rand}^{(\mathrm{V})}})$ and output (C_i, π_i).

- $\mathcal{S}_{\mathsf{lk}}$ **(Lookup Value**, leakage $(\mathsf{C}_i, \mathsf{index}, i, \upsilon)$**)**: $\pi \leftarrow \mathsf{Sim}_{\mathsf{Open}}\big(\mathsf{C}_i^{\mathsf{value}}, \mathsf{index}, \upsilon\big)$, attesting $\mathsf{value}_i(\mathsf{index}) = \upsilon$.

- $\mathcal{S}_{\mathsf{cp}}$ **(Value Consistency**, leakage $(\{\mathsf{C}_k\}_{i \le k \le j}, \mathsf{index}, i, j)$**)**: draw a fresh $y_{\mathsf{index}} \xleftarrow{\$} \mathbb{F}$ once per index index (reused on all later consistency queries for index), and produce the two openings

$$\pi_i \leftarrow \mathsf{Sim}_{\mathsf{Open}}\big(\mathsf{C}_i^{\mathsf{rand}^{(\mathrm{V})}}, \mathsf{index}, y_{\mathsf{index}}\big), \quad \pi_j \leftarrow \mathsf{Sim}_{\mathsf{Open}}\big(\mathsf{C}_j^{\mathsf{rand}^{(\mathrm{V})}}, \mathsf{index}, y_{\mathsf{index}}\big),$$

  so that $\mathsf{rand}_i^{(\mathrm{V})}(\mathsf{index}) = \mathsf{rand}_j^{(\mathrm{V})}(\mathsf{index}) = y_{\mathsf{index}}$.

- **Lookup Index / Index Consistency:** no call to $\mathcal{S}$; the public label–index map suffices and $\mathcal{A}$ reconstructs any index opening itself.

Figure 12: The privacy simulator $\mathcal{S}$ for Aegon.

(aborting if a programmed point was previously queried); the update proof π_m is produced by Sim_Σ; the index-related commitments remain computed from the public index map. Since commitments are statistically hiding and the Σ-protocol is information-theoretically zero-knowledge, the simulated commitments and proof are identically distributed to the real ones; the only deviation is the Fiat–Shamir abort, which occurs with negligible probability. Hence each sub-step differs by at most $\mathrm{negl}(\lambda)$, and

$$\implies |\Pr[\mathsf{H}_1] - \Pr[\mathsf{H}_2]| \leq n \cdot \mathrm{negl}(\lambda).$$

H_3: **Simulate Value Lookups and Consistency Proofs.** We define sub-hybrids that, one query at a time, replace real value-side openings with the outputs of $\mathcal{S}_{\mathsf{lk}}$ and $\mathcal{S}_{\mathsf{cp}}$. For each $\mathsf{LookupValue}$ query, the opening of value_i at index is replaced by $\mathsf{Sim}_{\mathsf{Open}}$; for each value consistency query with $b = 1$, the two $\mathsf{rand}^{(\mathrm{V})}$ openings at index are replaced by $\mathsf{Sim}_{\mathsf{Open}}$ against the common evaluation y_{index} (with $b = 0$ returning $(0, \perp)$ as in both worlds). Repeats are served from the cache. Each replacement reduces to the zero-knowledge of the PCS opening, so

$$\implies |\Pr[\mathsf{H}_2] - \Pr[\mathsf{H}_3]| \leq (q_{\mathsf{lk}} + q_{\mathsf{cp}}) \cdot \mathsf{Adv}^{\mathsf{zk}}_{\mathsf{PCS}}.$$

In H_3 every byte seen by $\mathcal{A}$ is produced by $\mathcal{S}$ from the leakage of $\mathcal{F}_{\mathsf{Aegon}}$ alone: epoch commitments and update proofs from $\mathcal{S}_{\mathsf{com}}, \mathcal{S}_{\mathsf{upd}}$, value openings from $\mathcal{S}_{\mathsf{lk}}$, consistency proofs from $\mathcal{S}_{\mathsf{cp}}$, and the index layer from the public index map. Hence $\mathsf{H}_3 = \mathsf{IDEAL}_{\mathcal{F}_{\mathsf{Aegon}},\mathcal{S},\mathcal{Z}}$. Summing across the hybrids,

$$\left|\Pr[\mathsf{REAL} = 1] - \Pr[\mathsf{IDEAL} = 1]\right| \;\leq\; n \cdot \mathrm{negl}(\lambda) + (q_{\mathsf{lk}} + q_{\mathsf{cp}}) \cdot \mathsf{Adv}^{\mathsf{zk}}_{\mathsf{PCS}},$$

which is negligible for any PPT environment $\mathcal{Z}$, since the PCS opening is zero-knowledge ($\mathsf{Adv}^{\mathsf{zk}}_{\mathsf{PCS}} = \mathrm{negl}(\lambda)$) and $q_{\mathsf{lk}}, q_{\mathsf{cp}}, n = \mathrm{poly}(\lambda)$. Therefore π is private with respect to $\mathcal{F}_{\mathsf{Aegon}}$. □

E KZH-k description

Description of KZH-k can be seen in Figure 13. We define an asymmetric pairing group as a tuple $(p, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, where p denotes the order of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$, and e is an efficiently computable, non-degenerate bilinear map. Let $\mathbb{F}$ be the finite field such that the scalars of $\mathbb{G}_1$ lie in $\mathbb{F}$. We adopt additive notation: for any $a \in \mathbb{F}$ and $\mathsf{G} \in \mathbb{G}_1$, scalar multiplication is written as $a \times \mathsf{G}$. We assume G is a generator of $\mathbb{G}_1$, and V is a generator of $\mathbb{G}_2$. For a multilinear polynomial $f(\vec{X}_1, \vec{X}_2, \ldots, \vec{X}_k)$, we denote $f_{\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_i}$ as the partial evaluation

$$f(\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_i, \vec{X}_{i+1}, \ldots, \vec{X}_k).$$

Free Boolean opening. A subtle point in Figure 13 is that although the auxiliary inputs are computed during the commitment phase for all $j = 1$ to $k - 1$, only those with $j \leq \lfloor k/2 \rfloor$ are actually used in the opening phase. This is because the opening procedure in Figure 13 is designed for the general case of evaluating at a random point $(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_k)$. To allow for free Boolean opening, we compute the full auxiliary input. In the special case of Boolean points, the summation in Equation (2) reduces to selecting a single value $\text{aux}_{\vec{b}_1, \ldots, \vec{b}_{j-1}, \vec{b}}$. This is due to the fact that

$$\text{eq}((\vec{x}_1, \ldots, \vec{x}_{j-1}), (\vec{b}_1, \ldots, \vec{b}_{j-1}))$$

vanishes everywhere on the Boolean hypercube except at $(\vec{b}_1, \ldots, \vec{b}_{j-1}) = (\vec{x}_1, \ldots, \vec{x}_{j-1})$. Consequently, openings at Boolean points come essentially for free.

KZH-k.setup$(\lambda, (d_1, d_2, \ldots, d_k), k)$**:**

- Sample $\mathsf{G} \leftarrow \mathbb{G}_1, \mathsf{V} \leftarrow \mathbb{G}_2$ and $[[\mu_{\vec{b},j}]_{\vec{b}\in\{0,1\}^{d_j}}]_{j=1}^{k} \leftarrow \mathbb{F}$
- $\mathbf{H}_1 = \{\mathsf{H}_{\vec{b}_1,\ldots,\vec{b}_i} \leftarrow (\prod_{j=1}^{k} \mu_{\vec{b}_j,j}) \times \mathsf{G} : \forall\, \vec{b}_1 \in \{0,1\}^{d_1}, \ldots, \vec{b}_k \in \{0,1\}^{d_k}\}$
- $\mathbf{H}_2 = \{\mathsf{H}_{\vec{b}_2,\ldots,\vec{b}_k} \leftarrow (\prod_{j=2}^{k} \mu_{\vec{b}_j,j}) \times \mathsf{G} : \forall\, \vec{b}_2 \in \{0,1\}^{d_2}, \ldots, \vec{b}_k \in \{0,1\}^{d_k}\}$
- $\ldots$
- $\mathbf{H}_k = \{\mathsf{H}_{\vec{b}_k} \leftarrow \mu_{\vec{b}_k,k} \times \mathsf{G} : \forall\, \vec{b}_k \in \{0,1\}^{d_k}\}$
- $\mathbf{V} = \{\mathsf{V}_{\vec{b}_j,j} \leftarrow \mu_{\vec{b}_j,j} \times \mathsf{V} : \forall\, j \in [1,k],\, \vec{b}_j \in \{0,1\}^{d_j}\}$
- Output $(\mathsf{G}, \mathsf{V}, \mathbf{H}_1, \mathbf{H}_2, \ldots, \mathbf{H}_k, \mathbf{V})$

KZH-k.commit$(\mathbf{srs}, f)$**:** For $f(\vec{X}_1, \vec{X}_2, \ldots, \vec{X}_k)$, compute the commitment as follows:

- Output $\mathsf{C} \leftarrow \langle f, \mathbf{H}_1\rangle = \sum_{\vec{b}_1\in\{0,1\}^{d_1},\ldots,\vec{b}_k\in\{0,1\}^{d_k}} f_{\vec{b}_1,\vec{b}_2,\ldots,\vec{b}_k} \times \mathsf{H}_{\vec{b}_1,\vec{b}_2,\ldots,\vec{b}_k}$
- For $j = 1$, ..., $k-1$ compute $\mathrm{aux}_{\vec{b}_1,\ldots,\vec{b}_j} := \langle f(\vec{b}_1, \ldots, \vec{b}_j, \vec{X}_{j+1}, \ldots, \vec{X}_k), \mathbf{H}_{j+1}\rangle$, for each $\vec{b}_i \in \{0,1\}^{d_i}$. The open algorithm takes the auxiliary input implicitly.

KZH-k.open$(\mathbf{srs}, \mathsf{C}, f, \vec{x}_1, \ldots, \vec{x}_k)$**:** Given commitment $\mathsf{C} \in \mathbb{G}_1$, polynomial $f(\vec{X}_1, \vec{X}_2, \ldots, \vec{X}_k)$ and inputs $\vec{x}_j \in \mathbb{F}^{d_j}$, compute opening as follows:

- For $j = 1, \ldots, k-1$:
  - Compute $f_{\vec{x}_1,\vec{x}_2,\ldots,\vec{x}_{j-1}}$ efficiently from previous partial evaluation.
  - Compute vector $\vec{\mathsf{D}}_j := \{\mathsf{D}_{j,\vec{b}} := \langle f_{\vec{x}_1,\vec{x}_2,\ldots,\vec{x}_{j-1},\vec{b}}, \mathbf{H}_{j+1}\rangle : \vec{b} \in \{0,1\}^{d_j}\}$.

  We observe Equation 2. For $j \le \lfloor k/2 \rfloor$, computing $\vec{\mathsf{D}}_j$ via this identity requires $O(N^{1/2})$ operations, where N denotes the overall size of the polynomial. For $j > \lfloor k/2 \rfloor$, it is more efficient to compute directly, since the size of partially-evaluated polynomial $f_{\vec{x}_1,\vec{x}_2,\ldots,\vec{x}_j}$ is already smaller than $O(N^{1/2})$.

$$\langle f_{\vec{x}_1,\ldots,\vec{x}_{j-1},\vec{b}}, \mathbf{H}_{j+1}\rangle = \sum_{\substack{\vec{b}_1\in\{0,1\}^{d_1} \\ \cdots \\ \vec{b}_{j-1}\in\{0,1\}^{d_{j-1}}}} \mathrm{eq}\big((\vec{x}_1, \ldots, \vec{x}_{j-1}), (\vec{b}_1, \ldots, \vec{b}_{j-1})\big) \times \mathrm{aux}_{\vec{b}_1,\ldots,\vec{b}_{j-1},\vec{b}} \tag{2}$$

- Output $\pi \leftarrow \{[\vec{\mathsf{D}}_j]_{j\in[1,k-1]},\, f_{\vec{x}_1,\ldots,\vec{x}_{k-1}}\}$

KZH-k.verify$(\mathbf{srs}, \mathsf{C}, [\vec{x}_j]_{j\in[1,k]}, y, \pi)$**:** Given commitment $\mathsf{C} \in \mathbb{G}_1$, inputs $\vec{x}_j \in \mathbb{F}^d$, output $y \in \mathbb{F}$ and opening proof π, does as follows:

- Parse $\{[\vec{\mathsf{D}}_j]_{j\in[1,k-1]},\, f_{\vec{x}_1,\ldots,\vec{x}_{k-1}}\} \leftarrow \pi$
- Set $\mathsf{C}_0 = \mathsf{C}$ and for $j \in 1, \ldots, k-1$, do the following:
  1. Check that $e(\mathsf{C}_{j-1}, \mathsf{V}) = \sum_{\vec{b}\in\{0,1\}^{d_j}} e(\mathsf{D}_{j,\vec{b}}, \mathsf{V}_{i,j})$
  2. Compute $\mathsf{C}_j = \sum_{\vec{b}\in\{0,1\}^{d_j}} \mathrm{eq}(\vec{x}_j, \vec{b}) \times \mathsf{D}_{j,\vec{b}}$
- Check that $\mathsf{C}_{k-1} = \langle f_{\vec{x}_1,\vec{x}_2,\ldots,\vec{x}_{k-1}}, \mathbf{H}_k\rangle$
- Check that $f_{\vec{x}_1,\vec{x}_2,\ldots,\vec{x}_{k-1}}(\vec{x}_k) = y$

Figure 13: (non-zero-knowledge) KZH-k description to commit to a multilinear polynomial $f(\vec{X}_1, \vec{X}_2, \ldots, \vec{X}_k)$ such that $|\vec{X}_i| = d_i$.

E.1 Zero-Knowledge Compiler for KZH

A general technique for transforming a homomorphic PCS into a hiding one is to use a Σ-protocol (see [Bün+21]). The key idea is to blind the original commitment C to a polynomial $f(\vec{X})$ to achieve hiding. During evaluation, the prover sends the evaluation $f(\vec{x})$ along with a commitment R to a

fully random polynomial $r(\vec{X})$ of the same degree as $f(\vec{X})$, and the evaluation $r(\vec{x})$ to the verifier. The verifier then samples a random challenge $\alpha \in \mathbb{F}$, and the prover proves that the combined and unblinded commitment $\alpha \times C + R$ opens to $\alpha \cdot f(\vec{x}) + r(\vec{x})$ at $\vec{x}$. IronDict uses a similar technique to build a zero-knowledge version of KZH and shows that for KZH-k, it suffices for $r(\vec{X})$ to have only $k \cdot N^{1/k}$ non-zero terms. Let

$$(\text{Setup}_{\text{KZH-k}}, \text{Commit}_{\text{KZH-k}}, \text{Open}_{\text{KZH-k}}, \text{Verify}_{\text{KZH-k}})$$

be the PCS algorithms for KZH-k. They define the algorithms

$$(\text{zkSetup}_{\text{KZH-k}}, \text{zkCommit}_{\text{KZH-k}}, \text{zkOpen}_{\text{KZH-k}}, \text{zkVerify}_{\text{KZH-k}})$$

as the corresponding hiding (zero-knowledge) variant of KZH-k, which can be defined by using a sparse random polynomial $r(\vec{X})$. For completeness, we include this compiler in Figure 14. For more details, we refer the reader to Appendix D of IronDict.

- $\text{zkSetup}_{\text{KZH-k}}(\lambda, k)$: This algorithm runs $\text{Setup}_{\text{KZH-k}}(\lambda, k)$ and outputs verifier key vk and prover key pk. It also outputs a new generator $h \in \mathbb{G}_1$, and we assume its Dlog is not known with respect to other generators.
- $\text{zkCommit}_{\text{KZH-k}}(\mathsf{pk}, f(\vec{X}))$:
  1. The prover samples randomness $\tau \leftarrow \mathbb{F}$
  2. Then computes the commitment $C \leftarrow \mathsf{Commit}_{\text{KZH-k}}(\mathsf{pk}, f(\vec{X}))$, and outputs $C_{\text{hide}} := C + \tau \cdot h$.
- $\text{zkOpen}_{\text{KZH-k}}(\mathsf{pk}, f(\vec{X}), \vec{x}, (C_{\text{hide}}, \tau))$:
  1. The prover samples $r(\vec{X})$ which has the same size as $f(\vec{X})$ and samples randomness $\rho \leftarrow \mathbb{F}$.
  2. The prover runs $R_{\text{hide}} \leftarrow \text{zkCommit}_{\text{KZH-k}}(\mathsf{pk}, r(\vec{X}); \rho)$.
  3. The prover computes $y_R \leftarrow r(\vec{x})$ and then sends (R_{hide}, y_R) to the verifier.
  4. The prover receives challenge α and computes $\rho' \leftarrow \alpha \cdot \tau + \rho$ and sends it to the verifier.
  5. The prover computes $C_{\text{lin}} := \alpha \times C_{\text{hide}} + R_{\text{hide}} - \rho' \times h$.
  6. Finally the prover runs $\pi \leftarrow \text{Open}_{\text{KZH-k}}(\mathsf{pk}, \alpha \times f(\vec{X}) + r(\vec{X}), C_{\text{lin}}, \vec{x})$ and outputs proof π.
- $\text{zkVerify}_{\text{KZH-k}}(\mathsf{vk}, C_{\text{hide}}, \vec{x}, y, \pi)$:
  1. The verifier receives (R_{hide}, y_R) from the prover and sends challenge α and receives ρ'.
  2. The verifier computes $C_{\text{lin}} := \alpha \times C_{\text{hide}} + R_{\text{hide}} - \rho' \times h$.
  3. The verifier computes $y_{\text{lin}} := \alpha \times y + y_R$.
  4. The verifier runs $\text{Verify}_{\text{KZH-k}}(\mathsf{vk}, C_{\text{lin}}, \vec{x}, y_{\text{lin}}, \pi)$ and outputs its result.

Figure 14: zk-friendly variant of KZH-k commitment and opening scheme. To simplify the protocol, we assume the auxiliary input of KZH is part of the opening and is generated while opening, while in reality, the prover can compute it during commitment and cache it.

F Sharded Open Addressing

We now isolate and formally define the primitive underlying our parallel index assignment, which we call *sharded open addressing*. Classical open addressing operates over a single address space of size $\eta \cdot N$ and is inherently sequential: insertions are processed one at a time, and a single probe sequence may traverse the entire space. Sharded open addressing runs open addressing at two levels: a probe-free top level that pins each label to one shard, and an ordinary single-shard open-addressing routine confined to that shard's local address space. Because the top level is counter-independent, the entire probe sequence of any label stays inside one shard, so the η shards run fully in parallel with no cross-shard coordination. We give the formal procedure in Figure 15.

- **Parameters.** Shard count η (a power of two); shard map $H_\eta \colon \{0,1\}^\star \to \{0,\dots,\eta-1\}$. The array arr is sharded into $\mathsf{arr}[0], \mathsf{arr}[1], \dots, \mathsf{arr}[\eta-1]$. $\mathsf{OA.append}(\cdot,\cdot)$ and $\mathsf{OA.lookup}(\cdot,\cdot)$ denote the standard (single-array) open-addressing algorithms.
- $\mathsf{ShardedOA.append}(\mathsf{arr}, \ell)$:
  1. $j := H_\eta(\ell)$.
  2. **run** $\mathsf{OA.append}(\mathsf{arr}[j], \ell)$.
- $\mathsf{ShardedOA.lookup}(\mathsf{arr}, \ell)$:
  1. $j := H_\eta(\ell)$.
  2. **return** $\mathsf{OA.lookup}(\mathsf{arr}[j], \ell)$.

Figure 15: Sharded (two-level) open addressing. H_η selects a shard j and the unmodified single-array open-addressing routine OA runs on $\mathsf{arr}[j]$. Distinct-shard appends are independent, so the η shards run in parallel. A plain hash suffices for H_η; we instantiate it with a VRF to resist adversarial label grinding (Section 7.1).

Sharded open addressing inherits its correctness directly from the single-array scheme. The first level is collision-free: H_η deterministically maps each label ℓ to a single shard $j = H_\eta(\ell)$, so a label is never split across shards and every append and lookup for ℓ acts on the same array $\mathsf{arr}[j]$. The second level is just the standard open-addressing routine OA running on $\mathsf{arr}[j]$, so its behavior—in particular the uniqueness of the assigned slot—is unchanged. The two levels therefore compose without interaction, and the sharded scheme is correct whenever OA is.

Capacity is the only quantity affected by sharding, and it is governed by Lemma 1. Since H_η assigns labels to shards uniformly and independently, the per-shard loads are exactly the balls-into-bins variables of the lemma, whose maximum concentrates tightly around the mean in our heavily-loaded regime. Consequently the most-loaded shard exceeds the average by only a vanishing margin: as computed in Section 7.1, for $\eta = 128$ and $N = 2^{25}$ no shard overflows until the system holds more than a $(1-\gamma)$ fraction of the total capacity $\eta \cdot N = 2^{32}$, with $\gamma \approx 1.4 \times 10^{-3}$ at $\lambda = 128$. Sharded open addressing thus recovers over 99.8% of the capacity of an unpartitioned table while making index assignment embarrassingly parallel across the η shards.