Securing Update Propagation with Homomorphic Hashing

Kevin Lewi, Wonho Kim, Ilya Maykov, Stephen Weis

Facebook

Abstract

In database replication, ensuring consistency when propagating updates is a challenging and extensively studied problem. However, the problem of *securing* update propagation against malicious adversaries has received less attention in the literature. This consideration becomes especially relevant when sending updates across a large network of untrusted peers.

In this paper we formalize the problem of secure update propagation and propose a system that allows a centralized distributor to propagate signed updates across a network while adding minimal overhead to each transaction. We show that our system is secure (in the random oracle model) against an attacker who can maliciously modify any update and its signature. Our approach relies on the use of a cryptographic primitive known as *homomorphic hashing*, introduced by Bellare, Goldreich, and Goldwasser.

We make our study of secure update propagation concrete with an instantiation of the lattice-based homomorphic hash LtHash of Bellare and Miccancio. We provide a detailed security analysis of the collision resistance of LtHash, and we implement LtHash using a selection of parameters that gives at least 200 bits of security. Our implementation has been deployed to secure update propagation in production at Facebook, and is included in the Folly open-source library.

1 Introduction

In the context of distributed systems, *database replication* refers to the process of copying and synchronizing data across multiple locations to improve availability and read latency. Database modifications must be propagated to each location in order to maintain the consistency of the replicated objects with the master database.

Propagating updates efficiently. In this work, we consider the model in which database writes are published by a central *distributor* that manages the master database. This distributor is responsible for propagating updates to a set of *subscribers* in a reliable and efficient manner. The simplest method for publishing updates is for the distributor to be in charge of directly sending the updates to each subscribed client. In practice, this method does not scale well as the number of subscribed clients and the rate of updates increases. Handling frequent updates with a centralized distributor can saturate the distributor's network interface controller, leaving it unable to fully distribute an update before a subsequent update is ready to be published.

To efficiently offload the responsibilities of propagating the changes from the distributor to every subscriber, the system can delegate the update propagation through its clients, so that subscribers can also participate in forwarding the distributor's original updates to other subscribers. This

approach effectively reduces the number of connections the distributor must manage, without incurring an unnecessary bandwidth overhead. But now, to ensure consistency, every downstream subscriber needs to trust a set of intermediate subscribers to have correctly propagated the original updates from the distributor. The challenge of maintaining the integrity of the distributor's updates across a network of untrusted subscribers is what we refer to as the *secure update propagation problem*.

1.1 Secure Update Propagation

To address the secure update propagation problem, the distributor can use digital signatures to assert the authenticity and integrity of the messages it distributes. Concretely, the distributor generates a public and private key pair, publishes the public key to every subscriber upon joining the network, and keeps the private key secret. The per-update signatures can then be constructed over either the contents of the update or the contents of the updated database. These two approaches offer various efficiency tradeoffs, outlined below.

Approach 1: Signing each update. The most straightforward approach to securely handling update propagation is for the distributor to directly sign the contents of each update, and send the update along with its signature to its subscribers. Subscribers can then use the signature to verify the new content before applying the update to their database replica. While this approach successfully prevents an attacker from modifying updates maliciously, it also adds complications to the handling of batch updates and offline database validation:

- **Batch updates.** When a subscriber needs to update another subscriber with a sequence of updates, the signatures for each of these updates must be sent and verified individually by the recipient. This situation can arise after a subscriber has been disconnected from the network for some period of time and then attempts to rejoin and receive the latest updates. Having signatures on each update means that if the subscriber missed m updates, it must now verify m signatures in order to catch up.
- **Offline database validation.** In offline validation, a subscriber wishes to validate its database replica against the original copy held by the distributor, without having to reach out to the distributor. The subscriber performs this validation by scanning the entire database to perform an integrity check. With an approach in which each update is signed, the subscriber must validate a signature for every update it received since the original version of the database.

To illustrate the wastefulness of this approach, consider the case in which the first row of the database holds an integer counter, and each update simply increments this integer by 1. If there are m such updates, then the batch update and offline database validation operations could involve m signature validations, even though the complete sequence of transformations trivially updates a single row of the database.

Typically in these scenarios, it may be more efficient for the subscriber to directly ask the distributor to re-sign and send an updated database. But in a network where failures are the norm, this approach quickly devolves to a setup where a large number of subscribers are frequently reaching out to the distributor for updates, defeating the original purpose of delegating the propagation responsibilities to subscribers.

Approach 2: Signing the database. Instead of relying on a signature which covers the update contents, an alternative approach is to rely on database signatures. Here, the signature algorithm is computed over the *database contents* after each update, rather than on the update itself. When each subscriber receives the update along with the database signature, they can first apply the update to the database, and then use the database signature to verify that the updated database is still valid.

Now, when handling batch updates and offline database validation, the subscribers can apply updates to their database replicas without having to worry about integrity. Once they reach the final result, they can then perform a single integrity check to verify that their end state matches the distributor's end state. In this scenario, each batch operation only requires a single signature validation as opposed to one for each update. However, the main drawback to directly signing the database in this manner is that for each update published, the distributor must iterate over the entire database to produce the signature. Similarly, each subscriber must load the entire database into memory in order to verify the database signature.

Approach 3: Efficiently updatable hashing. Ideally, we want to take an approach that provides integrity of updates using a computation that does not depend on the size of the database or the total number of updates. To address these issues, the distributor can use an *efficiently updatable collision resistant hash function* to hash the entire database into a small digest. The resulting digest would be directly signed, as opposed to having the distributor sign the database itself. On each update, the "efficiently updatable" property of the hash function guarantees that the distributor can compute the hash of the updated database using only the hash of the previous database version and the update. The "collision resistant" property of the hash function, along with the unforgeability of the signature algorithm, ensures integrity through the sequence of updates.

Merkle trees [Mer87] provide a partial solution to this problem. By associating each database row with a leaf node of the tree and signing the root hash using the distributor's signing key, the distributor can update the database signature in time proportional to the depth of the Merkle tree (i.e., logarithmic in the database size). However, this improved update and verification performance can only be attained by subscribers who keep a representation of the entire Merkle tree in memory—thus requiring memory overhead linear in the number of database rows.

A truly efficient solution would allow the distributor and its subscribers to update the database hash entirely independently of the size of the database. These requirements are satisfiable through the use of a cryptographic primitive called *homomorphic hashing*.

Homomorphic hashing. Bellare, Goldreich, and Goldwasser [BGG94] introduced the concept of an "incremental" (homomorphic[1]) hash function to solve the following problem: Given the hash of an input, along with a small update to the input, how can we compute the hash of the new input with its update applied, without having to recompute the entire hash from scratch? In a follow-up work, the authors proposed an application of homomorphic hashing to the use of authentication tags in virus protection [BGG95]. Subsequently, Bellare and Micciancio introduced a general paradigm for designing homomorphic hash functions whose collision resistance is based on the hardness of a certain computational problem in groups [BM97]. Under this paradigm, they instantiated three constructions: AdHash, MuHash, and LtHash.

[1]The original name for the property is "incremental" [BGG94, BM97], but subsequent works adopted the term "homomorphic" [KFM04, MTA17].

AdHash initially received the most attention by several works which aimed to implement the construction [SY98, CL99, GSC01], each using a 128-bit or 256-bit modulus. However, Wagner [Wag02] later showed an attack on the generalized birthday problem which could be used to find collisions for AdHash on an n-bit modulus in time $O(2^{2\sqrt{n}})$, and that the AdHash modulus needs to be greater than 1600 bits long to provide 80-bit security. Lyubashevsky [Lyu05] and Shallue [Sha08] showed how to solve the Random Modular Subset Sum problem (essentially equivalent to finding collisions in AdHash) in time $O(2^{n^\epsilon})$ for any $\epsilon < 1$, which indicates that AdHash requires several more orders of magnitude larger of a modulus just to provide 80-bit security.

MuHash was also implemented in later works [Bro08, MTA17] using binary elliptic curves. Although MuHash on binary elliptic curves is quite performant, it requires the use of a special encoding function and hence the implementation is not straightforward.

1.2 Our Contributions

In this work, we conduct an end-to-end study of the secure update propagation problem, from theory to practice. In doing so, we make the following contributions:

- We solidify the theoretical foundations of Bellare and Micciancio's lattice-based homomorphic hash known as LtHash, giving us stronger confidence in its security, especially in a concrete setting.
- We define a security model for update propagation and provide a construction based on homomorphic hash functions.
- We implement LtHash, benchmark it on test hardware, and discuss lessons learned when using it to secure update propagation in production at Facebook.

Revisiting lattice hashing. Bellare and Micciancio originally defined LtHash on inputs that could be split up into individually indexed blocks, so that the underlying hash takes as input the unique index along with the block contents to produce the hash output. This definition was later generalized to one that takes sets and multisets of bitstrings as input to LtHash [CDvD$^+$03].

In Section 2, we define the set homomorphism and collision resistance properties of a homomorphic hash function, and in Appendix A we show that LtHash is collision resistant in the random oracle model through a *direct* reduction to the hardness of the Short Integer Solutions (SIS) problem. Prior works have either presented this proof on a specialization of LtHash with indexed blocks as inputs [BM97], or have indirectly proved collision resistance to a variant of a weighted knapsack problem with polynomially-large set multiplicities [CDvD$^+$03]. Given the history of AdHash and the various implementations that instantiated it without fully understanding the security implications, we believe it is valuable to robustly prove the collision resistance property of LtHash with well-defined security models. Having these concrete models also helps to facilitate future works that aim to improve or apply homomorphic hashing to other protocols. For example, in our work, we use homomorphic hashing to achieve secure update propagation, and the direct proof enables us to concretely analyze its security.

Update propagation. In Section 3, we introduce a definition and security model for update propagation, represented by a tuple of five algorithms: Setup, Publish, GetUpdates, ApplyUpdates,

and Validate. We show how to apply homomorphic hashing to secure update propagation efficiently, reducing the total number of signatures which need to be verified in the ApplyUpdates and Validate procedures without incurring an extra memory overhead (as is the case with Merkle trees). As an example, consider a database $\mathcal{D}$ consisting of three indexed rows:

Row	Data
1	"apple"
2	"orange"
3	"banana"

Using $||$ as the string concatenation operation, and $+$ as a special combining operation specific to LtHash[2] (with $-$ being its inverse), we define the hash of $\mathcal{D}$ as

$$\mathsf{LtHash}(\mathcal{D}) \coloneqq \mathsf{LtHash}(1 \,||\, \text{“apple”}) + \mathsf{LtHash}(2 \,||\, \text{“orange”}) + \mathsf{LtHash}(3 \,||\, \text{“banana”}).$$

Now, suppose an update comes along which changes row 2 from "orange" to "peach", and let $\mathcal{D}^*$ represent the updated database. Instead of having to recompute $\mathsf{LtHash}(\mathcal{D}^*)$ from scratch, we can use the existing hash $\mathsf{LtHash}(\mathcal{D})$ to get

$$\mathsf{LtHash}(\mathcal{D}^*) = \mathsf{LtHash}(\mathcal{D}) + \mathsf{LtHash}(2 \,||\, \text{“peach”}) - \mathsf{LtHash}(2 \,||\, \text{“orange”}).$$

By the set homomorphism property of LtHash, the output is guaranteed to be consistent with the result of directly computing LtHash on D^*. This technique allows us to update the database hash in time linear in the size of the update, as opposed to linear in the size of the database. Our construction is also not specific to LtHash, and we expect that further developments or optimizations to homomorphic hashing will translate to performance improvements for update propagation. In Section 3.2, we formally define and prove the security of this homomorphic hashing construction for update propagation.

Our framework is closely modeled after a real-world distributed system at Facebook known as Location Aware Distribution [Zav18]. Given the ubiquity of database replication in modern distributed systems, we anticipate that our formal treatment of update propagation will benefit future implementations that have similar system requirements.

Concrete instantiation. Finally, we instantiated LtHash with a concrete parameter setting (16-bit modulus with 1024 vector components), which we refer to as `lthash16`. Borrowing an analysis of the concrete hardness of the Short Integer Solutions problem [BGLS19], we estimate that `lthash16` provides at least 200 bits of security for collision resistance.

We have also included an implementation of LtHash in the Folly open-source library.[3] In Section 4, we discuss several optimizations that help to speed up the modular vector addition operations over 16-bit elements, and show our benchmarks for its performance in terms of wall time.

We deployed `lthash16` to secure update propagation within production at Facebook. Although we have yet to encounter instances of malicious actors attempting to compromise subscribers to forge updates, the homomorphic hash checks have so far been used to detect bugs, unconsidered edge cases, and logical inconsistencies which were promptly addressed and fixed. We found that a slow rollout of `lthash16` in a fail-open mode with logging acted as a safe way to prototype these integrity checks before switching over to a fail-closed and production-ready mode of operation.

[2]This combining operation is simply component-wise modular vector addition, and is defined formally in Section 2.3.
[3]This library can be found at https://github.com/facebook/folly

1.3 Related Work

The concept of homomorphic hashing originated from the seminal works that founded incremental cryptography [BGG94, BGG95, BM97] and the constructions known as AdHash, MuHash, and LtHash. Since then, there have been numerous follow-up works that propose alternative homomorphic hash functions.

Homomorphic hashing for multiset inputs. Clarke et al. [CDvD+03] extended homomorphic hashing to handle multiset inputs. They proposed several constructions, some of which require a secret key to evaluate the hash (and are not relevant in our model or its application to update propagation). They define the hash function MSet-VAdd-Hash to be a multiset homomorphic collision resistant hash function. Their construction is the natural extension of LtHash to handling multiset inputs (by introducing the multiplicity as a linear component to the hash of each unique set element), and collision resistance is also based on the hardness of the worst-case shortest vector problem.

However, Clarke et al. implicitly require that the modulus of their vector components is of the same order as the security parameter. In particular, if the multiplicity of any set elements were allowed to exceed the modulus, this would immediately result in easy-to-find collisions. In our work, we are mainly focused on set homomorphic hashing as it applies to update propagation (which does not use multiset inputs), and so we set our modulus for `lthash16` to be 2^{16} for efficiency reasons despite not being able to handle multisets with multiplicities that exceed the modulus. It remains an open problem to support multiset inputs for small moduli implementations of LtHash.

Elliptic curve multiset hash. Recently, Maitin-Shepard, Tibouchi, and Aranha [MTA17] demonstrated how to construct a homomorphic (multiset) hash known as Elliptic Curve Multiset Hash (ECMH). This hash is an instantiation of MuHash with a binary elliptic curve as the output group, and the collision resistance of ECMH is dependent on the computational complexity of solving the discrete logarithm problem for binary elliptic curves. The reliance on characteristic 2 in their construction eliminates the need for expensive field operations that would otherwise dominate computation time for the hash.

The instantiation of ECMH uses a binary curve variant of Shallue and van de Woestijne's encoding function [SvdW06], which involves implementation techniques that are significantly more sophisticated than those involved in the implementation of LtHash, and seemingly less standardized. Also, the use of characteristic 2 comes with a potential downside—somewhat recently, new techniques for computing discrete logarithms in binary elliptic curves have been proposed [Sem15, Kar15, KY15]. Although the attacks remain mostly theoretical, they could potentially affect the parameter settings and output size of ECMH. Nevertheless, we believe that ECMH could be an attractive option for other applications that use homomorphic hashing.

$\mathbf{SL_2}$ homomorphic hashing. Mullan and Tsaban [MT16] describe a homomorphic hash function (originally proposed by Tillich and Zémore [TZ94]) with outputs in $\mathrm{SL}_2(q)$, the group of 2×2 matrices of determinant 1 with entries in the finite field $\mathbb{F}_q$. Their construction is a departure from the randomize-then-combine paradigm, and unlike previous works, the input homomorphism is over the concatenation of bitstrings in $\{0,1\}^*$, rather than sets of bitstrings. They analyze the collision resistance property for sufficiently large q through a worst case to average case reduction for their hash construction, but since their cryptographic assumption remains nonstandard (especially given

the context of [CP94, PQTZ09]), the construction is incomparable to the hash algorithms described above.

LtHash with small parameters. Mihajloska et al. [MGS15] also revisit the use of LtHash for incremental hashing. However, their recommendations are to use a 64-bit modulus and a total of 2500 to 16000 bits of output from their underlying hash function, with benchmarks for settings of 2688 and 6528 bits of output from the hash. Under a 64-bit modulus, these constraints imply at most 102 vector components, which we believe is unlikely to yield the purported 128 or 256 bits of security against collision attacks. Although these numbers may have been suitable if Wagner's generalized birthday attack [Wag02] had remained the best attack on LtHash, advances in lattice reduction techniques (including the BKZ [SE94, CN11] algorithm) suggest that solving the Short Integer Solutions problem in a dimension-102 lattice may now be well within the range of feasibility.

Other applications. Homomorphic hashing has found applications in several other fields: most prominently, in efficient file distribution [KFM04, GR06], practical Byzantine fault tolerance [CL99], and memory integrity checking [CDvD+03]. These applications typically have a space requirement that renders the use of Merkle trees as a suboptimal solution. However, when space is less of a concern, using Merkle trees can be advantageous because they offer inclusion and exclusion proofs.

Specifically, it is possible to offer a short witness of inclusion (or exclusion) of an element as being part of (or not part of) a Merkle tree hash, to which an owner of the root tree hash can efficiently verify. This property is not present in any of the existing homomorphic hash constructions, and makes Merkle trees an attractive option for blockchains, verifiable logs like Certificate Transparency [LLK13, Lau14], and Apache Cassandra's incremental repair feature [Dej].

1.4 Notation

For $n \in \mathbb{N}$, we write $[n]$ to represent the set $\{1, \ldots, n\}$. We use $\mathcal{P}(S)$ to represent the powerset of a set S. We use $\{0,1\}^*$ to represent the set of all bitstrings of arbitrary length. For a set S, we write $x \xleftarrow{R} S$ to represent a uniformly random sampling of an element x in S. We use λ to denote the security parameter. We say that a function is *negligible* in λ if $f = o(1/\lambda^c)$ for all $c \in \mathbb{N}$. An algorithm or adversary is *efficient* if it runs in time polynomial in the security parameter λ.

2 Set Homomorphic Hashing

A *hash function* produces a fixed-length output from an arbitrary-length input, and is said to be *collision resistant* if it is computationally infeasible to find two distinct inputs which hash to the same output. Collision resistant hash functions have served as not only a fundamental building block of cryptographic primitives, but are also ubiquitous among applications in the handling of large amounts of data efficiently. In this section, we provide the formal definition of a set homomorphic collision resistant hash function, which is a generalization of incremental hashing as defined by Bellare, Goldreich, and Goldwasser [BGG94]. We then discuss the collision resistance of LtHash and analyze its concrete security for a specific set of parameters.

2.1 Definition

A set homomorphic hash function $\mathsf{H} : \mathcal{P}(\{0,1\}^*) \to \mathsf{G}$ is defined as transforming a set of input bit-strings in $\{0,1\}^*$ to elements of a commutative group $(\mathsf{G}, \circ)$ with two properties: set homomorphism and collision resistance.

Set Homomorphism. We say that the function H is *set homomorphic* for a commutative group $(\mathsf{G}, \circ)$ if for any two disjoint sets $S, T \in \mathcal{P}(\{0,1\}^*)$, we have that

$$\mathsf{H}(S \cup T) = \mathsf{H}(S) \circ \mathsf{H}(T).$$

Collision Resistance. We say that H is *collision resistant* if a computationally bounded adversary $\mathcal{A}$ cannot produce two input sets $S, T \in \mathcal{P}(\{0,1\}^*)$ such that $S \neq T$ and $\mathsf{H}(S) = \mathsf{H}(T)$ with non-negligible probability. Note that this is simply the standard definition of collision resistance applied to the function H.

We also define the collision resistance property within the random oracle model, as we will be modeling our underlying hash function as a random oracle. We write $\mathsf{H}^{(\mathcal{O})}$ to represent the algorithm H having access to the random oracle $\mathcal{O}$. This notation will only appear when formally defining and proving the collision resistance property within the random oracle model.

Definition 2.1 ($\mathsf{Expt}^{\mathsf{RO}}_{\mathsf{cr}}$)**.** For a security parameter λ and an efficient adversary $\mathcal{A}$, we define the experiment $\mathsf{Expt}^{\mathsf{RO}}_{\mathsf{cr}}(\mathsf{H}, \mathcal{A})$ for H, in the presence of a random oracle $\mathcal{O}$ controlled by the challenger, as follows:

1. The adversary $\mathcal{A}$ first submits to the challenger an integer m representing the maximum number of unique random oracle queries that $\mathcal{A}$ will make.

2. Then, $\mathcal{A}$ can adaptively make up to m unique random oracle queries to the challenger. For each $i \in [m]$, when the adversary submits the input $x_i \in \{0,1\}^*$, the challenger responds with $\mathcal{O}(x_i)$.

3. Eventually, $\mathcal{A}$ must output two distinct sets $S, T \in \mathcal{P}(\{0,1\}^*)$. The output of the experiment is 1 if $\mathsf{H}^{(\mathcal{O})}(S) = \mathsf{H}^{(\mathcal{O})}(T)$. Otherwise, the output of the experiment is 0.

The advantage of an adversary $\mathcal{A}$ for H in $\mathsf{Expt}^{\mathsf{RO}}_{\mathsf{cr}}$ is defined as

$$\mathsf{Adv}^{\mathsf{RO}}_{\mathsf{cr}}(\mathsf{H}, \mathcal{A}) \coloneqq \Pr[\mathsf{Expt}^{\mathsf{RO}}_{\mathsf{cr}}(\mathcal{A}) = 1].$$

We say that H is *collision resistant in the random oracle model* if for all efficient adversaries $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{RO}}_{\mathsf{cr}}(\mathsf{H}, \mathcal{A})$ is negligible in λ.

Multisets. In this work, we define homomorphic hashing with respect to sets of unique elements and not multisets, since the application to update propagation does not need to support multisets. Applications which rely on supporting collision resistance for multiset inputs can attempt to pre-index the input elements with distinct integers in order to ensure uniqueness. However, in general, collision resistance for multiset inputs is a strictly stronger requirement than collision resistance for set inputs.

2.2 Generalizing Set Homomorphic Hash Constructions

Bellare and Micciancio [BM97] introduced the "randomize-then-combine" paradigm for constructing a set homomorphic (a.k.a. "incremental") hash function. For an input set $S = \{x_1, \ldots, x_n\} \in \mathcal{P}(\{0,1\}^*)$, a commutative group G with operation $\circ$, and an underlying hash function $h : \{0,1\}^* \to \mathsf{G}$ modeled as a random oracle, the set homomorphic hash H is defined as

$$\mathsf{H}(S) = h(x_1) \circ h(x_2) \circ \cdots \circ h(x_n).$$

Set homomorphism of H follows immediately from this construction, and all known set homomorphic hash functions in previous works follow this paradigm. Bellare and Micciancio show that the collision resistance of H follows from the hardness of a computational problem in G known as the *balance problem*: given a sequence of n group elements $a_1, \ldots, a_n \in \mathsf{G}$, to find weights $w_1, \ldots, w_n \in \{-1, 0, 1\}$ such that

$$a_1^{w_1} \circ \cdots \circ a_n^{w_n} = e,$$

where e represents the identity element of G. Using this paradigm, they also describe three instantiations of H: AdHash, MuHash, and LtHash.

In AdHash, the group $(\mathsf{G}, \circ)$ is addition over $\mathbb{Z}_q$ for some sufficiently large modulus q. The collision resistance of AdHash is based on the difficulty of solving the Random Modular Subset Sum problem (which reduces to the aforementioned balance problem). MuHash is an instantiation of the randomize-then-combine paradigm for any group $(\mathsf{G}, \circ)$ for which the discrete logarithm problem is conjectured to be computationally hard (say, $\mathbb{Z}_q^*$ for sufficiently large and prime q). Finally, LtHash is instantiated over the group $(\mathbb{Z}_q^n, +)$, where collision resistance relies on the hardness of approximating the shortest vector in a lattice. Note that AdHash is equivalent to LtHash with $n = 1$.

We will restrict our attention to LtHash for the remainder of this section, and we refer the reader to Section 1.3 for further background on set homomorphic hash constructions explored in prior works.

2.3 Set Homomorphic Hashing from Lattices

Let $n, d > 0$ be positive integers, and fix an extensible output function (XOF) [Nat15] $\mathsf{h} : \{0,1\}^* \to \{0,1\}^{nd}$, where the output can be represented as a vector of n components each consisting of d bits, written as $\vec{h}(x) = \langle [\mathsf{h}(x)]_1, \ldots, [\mathsf{h}(x)]_n \rangle$ for an input $x \in \{0,1\}^*$. Using $q = 2^d$, we define $\mathsf{LtHash}_{n,d} : \mathcal{P}(\{0,1\}^*) \to \mathbb{Z}_q^n$ as follows:

$$\mathsf{LtHash}_{n,d}\left(\{x_1, \cdots, x_k\}\right) = \sum_{i=1}^{k} \vec{h}(x_i) \pmod q,$$

where the summation is taken by applying component-wise vector addition mod q.

Set homomorphism. As is the case with all constructions which follow the randomize-then-combine paradigm, the set homomorphism of LtHash follows directly from the associativity and commutativity of the group $(\mathbb{Z}_q^n, +)$.

Constructing collisions with multiset inputs. We note that LtHash is *not* collision resistant among multiset inputs: for any two elements $x, y \in \{0,1\}^*$, we have $\mathsf{LtHash}_{n,d}(\{x\}) = \mathsf{LtHash}_{n,d}(\{x, y^{(q)}\})$, where $q = 2^d$ represents both the multiplicity of y and the modulus used in the component-wise vector operation. Clarke et al. [CDvD+03] handle this issue in their hash construction that supports multiset inputs (called $\mathsf{MSet\text{-}VAdd\text{-}Hash}$) by using a sufficiently large modulus q so that constructing these multiset inputs is infeasible (with respect to the security parameter λ), under the implicit assumption that an input multiset with super-polynomial multiplicities cannot be represented succinctly. For our purposes, this solution is problematic because the required increase in the modulus q to avoid this type of collision results in an order of magnitude increase in the length of the LtHash output, which is unnecessary given that our main application to update propagation does not benefit at all from support for multiset inputs.

Collision resistance. Bellare and Micciancio showed how to reduce the security of LtHash to the matrix kernel problem, which is also known as the Short Integer Solutions (SIS) problem, defined as follows.

Definition 2.2 (Short integer solutions (SIS))**.** Let n, m, q, B be positive integers with q prime. For an adversary $\mathcal{A}$, consider the following experiment $\mathsf{Expt}_{\mathsf{SIS}}(\mathcal{A})$:

1. The challenger samples $\mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n \times m}$ and sends $\mathbf{A}$ to the adversary $\mathcal{A}$.
2. The adversary outputs a non-zero vector $\mathbf{x} \in \mathbb{Z}^m$ with $\|\mathbf{x}\|_\infty \leq B$, and the output of the experiment is 1 if and only if $\mathbf{A} \cdot \mathbf{x} = \mathbf{0} \pmod{q}$.

We define the advantage of $\mathcal{A}$ in $\mathsf{Expt}_{\mathsf{SIS}}$ as

$$\mathsf{Adv}_{\mathsf{SIS}}^{(n,m,q,B)}(\mathcal{A}) \coloneqq \Pr[\mathsf{Expt}_{\mathsf{SIS}}(\mathcal{A}) = 1].$$

We say that $\mathsf{SIS}(n, m, q, B)$ is hard if for every adversary $\mathcal{A}$, it is the case that $\mathsf{Adv}_{\mathsf{SIS}}^{(n,m,q,B)}(\mathcal{A})$ is negligible in the security parameter λ.

Bellare and Micciancio proved the following theorem which links the collision resistance of LtHash with the computational complexity of the SIS problem:

Theorem 2.3 ([BM97], Theorem 6.1)**.** *For integers $n, d, Q > 0$, if $\mathsf{SIS}(n, Q, 2^d, 1)$ is hard, then the function $\mathsf{LtHash}_{n,d}$ is collision resistant for Q queries in the random oracle model.*

However, their reduction did not explicitly cover the generalization of LtHash to using sets as inputs, since they defined the input of LtHash to be a data string which could be indexed into blocks. Later works considered the generalization of LtHash to input sets, but did not provide a proof of security for the generalization.[4] For completeness, we reproduce the proof of collision resistance of LtHash in Appendix A, with minor alterations to encapsulate the generalization and ensure that it still applies in our setting.

By demonstrating a direct reduction from the collision resistance of LtHash to SIS, we can then establish a bound on the concrete security for a specific setting of LtHash parameters by analyzing and bounding the performance of the best attacks on SIS.

[4] [CDvD+03] attempts to reprove the collision resistance of LtHash, but do so with multiset inputs, and reduce from weighted knapsack as opposed to SIS, which is incompatible with, our setting.

2.4 Concrete Security of Short Integer Solutions

The SIS problem for a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ can also be formulated as the problem of finding a sufficiently short vector in the lattice $\Lambda_q^\perp(\mathbf{A})$ defined as

$$\Lambda_q^\perp(\mathbf{A}) \coloneqq \{\mathbf{x} \in \mathbb{Z}^m : \mathbf{A}\mathbf{x} = \mathbf{0} \in \mathbb{Z}_q^n\}.$$

There are two lines of work that can be applied to solve SIS: lattice reduction algorithms and combinatorial algorithms. Following the lead of Micciancio [Mic11], we find that the combinatorial algorithms are more relevant for the cryptanalysis of SIS in our setting, since they are able to fully take advantage of the large dimension m. Indeed, in the context of LtHash, m can be adversarially selected to be as large as possible, so we must analyze settings with $m \gg n$.

Lattice reduction algorithms. The Blockwise Korkine-Zolotarev (BKZ) [SE94, CN11] algorithm is a lattice reduction technique that aims to find a short basis for an input lattice, and hence, a solution to SVP. However, applying BKZ to SIS directly is somewhat problematic, especially for our setting of parameters. First, we note that the lattice $\Lambda_q^\perp(\mathbf{A})$ has dimension m, and the number of elements that belong to $\Lambda_q^\perp(\mathbf{A})$ is approximately $q^{n/m}$. Experiments have shown that BKZ does not perform very well when the dimension of the lattice is $m \geq n \geq 1000$ [CN11]. Even if BKZ were to be feasible for such high dimension lattices, there is another problem: the solutions to SIS that we require must be in $\{-1, 0, 1\}^m$, whereas existing BKZ experiments attain vectors with short ℓ_2-norm. In other words, the short vectors output by BKZ could be unsuitable as solutions to SIS.

In general, it appears that the lattice reduction techniques are not optimal for solving SIS with our setting, especially since they are unable to take advantage of settings where $m \gg n$.

Combinatorial algorithms. There is a wealth of literature on the study of solving hard instances of knapsack problems and the application of these combinatorial techniques to solving SIS [SS79, HJ10, BCJ11, CP91, Wag02, MS09, Lyu05, Sha08, LMPR08, BGLS19] (see [BGLS19] for an in-depth survey).

Since $m \gg n$, we are interested in algorithms which solve high-density SIS over $\mathbb{Z}_q^n$. We briefly describe the Camion-Patarin-Wagner (CPW) algorithm [CP91, Wag02] below. Our analysis is guided by the work of Bai, Galbraith, Li, and Sheffield [BGLS19] (BGLS), which gives a complete description and also performs a more rigorous survey of the CPW algorithm and its subsequent improvements.

Camion-Patarin-Wagner algorithm. For some number t, the algorithm proceeds in t rounds, where by the end of the last round, it is expected to output a linear combination of the m vectors that sums to $\mathbf{0}$.

The first round begins by splitting the m vectors into $k = 2^t$ lists, each containing m/k vectors. For each list, the algorithm builds a list of all linear combinations of the vectors with coefficients in $\{-1, 0, 1\}$. This results in $3^{m/k}$ elements per list. These lists are then paired up, and for each pair of lists L_1 and L_2, for each $(\mathbf{z}_1, \mathbf{z}_2) \in L_1 \times L_2$, the algorithm constructs a new list L_3 containing the sum $\mathbf{z}_1 + \mathbf{z}_2$ if the first ℓ coordinates are all zeros (for some appropriately chosen integer ℓ). This concludes the first round.

At this point, we have $k/2$ lists of elements, each containing m/k vectors on expectation. These lists are input to the second round, which repeats the same process of the previous round (with the

			CPW			MS		
	n	q	m	t	λ	m	t	λ
`lthash16`	1024	2^{16}	2^{167}	162	263	2^{133}	127	255
`lthash20`	1008	2^{20}	2^{204}	199	300	2^{147}	141	282
`lthash32`	1024	2^{32}	2^{168}	162	364	2^{186}	180	361

Table 1: Camion-Patarin-Wagner (CPW) and Minder-Sinclair (MS) algorithm statistics for selected LtHash parameters. We use n for the length of vector outputs of LtHash, q for the modulus, m for the dimension of SIS, t for number of rounds of the combinatorial attacks, and λ for the security parameter, equal to the base-2 log of the expected runtime of each algorithm.

second batch of ℓ coordinates matching all zeros). After t rounds, the CPW algorithm is able to find a solution with reasonable probability for any setting of t that satisfies:

$$\frac{2^{t-1}}{t} < \frac{m \log_2(3)}{n \log_2(q)} < \frac{2^t}{t+1}.$$

The complexity of the CPW algorithm can be measured by the size of the lists in the first round, multiplied by the total number of such lists. BGLS estimates that the complexity is

$$\mathsf{CPW}(m) = 2^t \cdot 3^{m/2^{t-1}},$$

but also note that this can improved to $2^t \cdot q^{n/t}$.

Minder-Sinclair algorithm. BGLS note that the Minder-Sinclair (MS) algorithm [MS09] yields a slightly better bound by taking advantage of the fact that the number of zeroed coordinates ℓ need not be the same on every level, and use a sequence of numbers $\ell_1, \ldots, \ell_t$ to express this extra flexibilty. As analyzed by BGLS, the MS algorithm achieves a complexity of $O(2^t \cdot q^{n-\ell_1} \cdot t^{-1})$, with $0 < \ell_1 < n/(t+1)$. We define

$$\mathsf{MS}(m) \coloneqq 2^t \cdot q^{n-n/(t+1)} \cdot t^{-1}$$

to represent a lower bound on the complexity of the MS algorithm.

The BGLS analysis provides a method to analyze the complexity of SIS for a fixed setting of m. We are interested in understanding the complexity of these combinatorial algorithms for extremely large m, which means that we can bound the security parameter λ with:

$$\lambda = \min_{m>0} \left(\max\{m, \mathsf{MS}(m)\} \right).$$

Instantiation. We define `lthash16` $= \mathsf{LtHash}_{1024,16}$, `lthash20` $= \mathsf{LtHash}_{1008,20}$, and `lthash32` $= \mathsf{LtHash}_{1024,32}$, and our results are summarized in Table 1. For each setting, we computed the minimum m and corresponding t that results in the smallest complexity for the CPW and MS algorithms. These values are set as the security parameter λ, with MS being more efficient.

Conservative estimation. Since we aim to obtain a comfortable lower bound on the security parameter, we chose to not factor in the success probability for each algorithm in our analysis, and to just assume that each run of the algorithm always successfully solves the SIS problem.

Also, we note that Bai et al. also present an improvement to the MS algorithm using the Hermite normal form (HNF) of the matrix $\mathbf{A}$. This variant results in a $n \times (m-n)$ matrix of random q-ary elements as opposed to $n \times m$, and they show how to take advantage of this to further improve upon the best attacks for SIS. We note however that since $m \gg n$ in our setting, it seems unlikely that the BGLS technique will result in a substantially improved attack on LtHash.

Nevertheless, we caution the reader to note that the BGLS refinement (and further improvements) upon the MS algorithm could result in having to lower the security estimates for LtHash. For `lthash16` in particular, although we calculated that the MS algorithm takes time 2^{255}, we conservatively estimate that it yields at least 200 bits of security.

3 Update Propagation

The update propagation problem focuses on how to handle distributed database replication in an efficient and secure manner. The central challenge we aim to address in this section is on how to maintain a large number of replicas of a database $\mathcal{D}$ yet still preserve performance and security while updating these replicas. Every entity that holds a replica of the database $\mathcal{D}$ is either a *distributor* or *subscriber*. For the sake of simplicity, we can assume that there only exists a single distributor,[5] and all other entities are subscribers. Typically, the distributor would be responsible for handling writes to the database, whereas each of the subscribers would provide a read interface to their clients. The writes to the database can be modeled as a constantly streaming log that the distributor has access to, and it is the sole job of the distributor to propagate these writes (database updates) to each of its subscribers.

So far, what we have described is encapsulated by the leader-follower framework that is popular among distributed database architectures, and is well-studied. However, in our model, we are interested in what happens when the number of subscribers is very large (say, millions), and are all *untrusted*.

We first note that, for supporting a small set of subscribers, the distributor can usually afford to maintain the connections to each of its subscribers directly. The per-subscriber upkeep for the distributor typically includes: storing internal subscriber state, pushing missing updates to each subscriber, and managing retries. As the number of subscribers grows, this upkeep becomes more challenging to maintain.

Scaling issues with Apache Zookeeper. Zookeeper is an example of a coordination service that enables its client nodes (subscribers) to connect directly to members of a centralized ensemble (the distributor). Since these ensembles consist of only a very limited set of servers, Zookeeper has trouble with scaling to a large number of client nodes.

To resolve this issue, Zookeeper introduced a new type of node, called an "observer", which is similar to a participant of the ensemble, with the exception that it does not participate in establishing consensus, and is used to scale out read traffic. In the context of update propagation, these observers can be used as proxies between clients and ensemble nodes to reduce the number of clients that

[5]The coordinating of multiple distributors can be delegated to a consensus mechanism which equally trusts every leader.

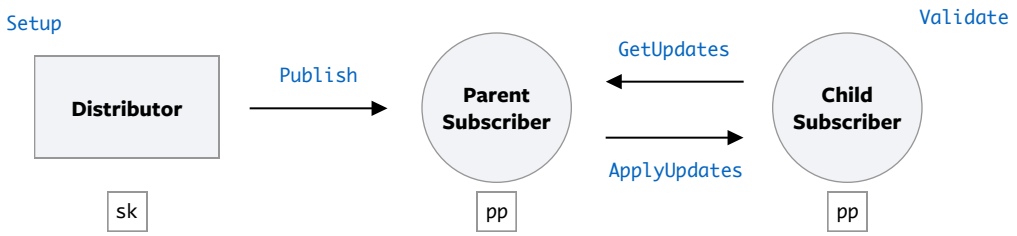

Figure 1: Organization of the update propagation algorithms. Note that the child subscriber may itself have other child subscribers, so that in general, subscribers may be multiple hops away from the distributor.

ensemble nodes must manage connections to. However, this is a highly bandwidth-inefficient approach to scaling writes to a large set of subscriber nodes.

In general, to avoid the inherent scaling and reliability issues associated with having a central set of machines which must directly publish updates to a large number of clients, the central distributor can instead defer the publishing of updates to its clients, who are then responsible for propagating updates to one another in the fashion of a peer-to-peer network.

Untrusted subscribers. In our model, the distributor aims to maintain the integrity of its updates, even if the subscribers that propagate its updates are untrustworthy. If each subscriber were to directly receive its updates from the distributor, then it would suffice for the distributor to use a secure and authenticated communication protocol (such as TLS with X.509 certificates) to transmit updates its subscribers, so that the subscribers could verify that all updates are coming directly from the distributor before applying them. However, when subscribers are receiving updates from other subscribers, we need to consider alternative approaches to ensuring update integrity.

Update propagation setting. To formally describe the various approaches we consider in this section, we first define the properties of an update propagation scheme, including correctness and security. To motivate our definition, we consider the roles of three entities in the system: a distributor, a parent subscriber, and a child subscriber, organized in Figure 1. The distributor is responsible for initializing the system and publishing updates as they are encountered (Setup). The published updates are received by the parent subscriber, who validates the updates before sending them to the child subscriber (Publish).

In a separate setting, the child subscriber may ask for a sequence of updates from the parent subscriber (GetUpdates), which the parent subscriber executes to produce a response to the child subscriber. Then, the child subscriber performs a validation on the response (ApplyUpdates) to verify that the updates it is receiving match those of the distributor. In an offline phase, any subscriber (parent or child) can periodically run a procedure that verifies the consistency and integrity of their replica database against the distributor's master database (Validate). These procedures are defined formally in Section 3.1.

Constructions. After presenting the definitions, we consider three different instantiations of a secure update propagation scheme. The first construction (Π_{update}, described in Appendix B.3) essentially amounts to having the distributor directly sign the contents of each update it publishes

to its subscriber. In the second construction (Π_{db}, described in Appendix B.4), the distributor produces a signature over the database after each update, as opposed to signing the contents of the update itself. And finally, in our third construction (Π_{hash}, presented in Section 3.2) the distributor first generates a hash of the database before signing it, and keeps this database hash around to produce signatures for future updates. We show that if the hash is set homomorphic, then this process is efficient. In Section 3.3 we present a full comparision of the tradeoffs between the three approaches.

3.1 Definitions

An update propagation scheme $\mathsf{UP} = (\mathsf{Setup}, \mathsf{Publish}, \mathsf{GetUpdates}, \mathsf{ApplyUpdates}, \mathsf{Validate})$ consists of a tuple of algorithms defined as follows.

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{sk})$. The Setup algorithm takes as input the security parameter λ and outputs public parameters pp and a secret key sk. The parameters pp are initially distributed to every subscriber in the network, and are also given to subscribers who later join the network. The secret key sk is given to the distributor. The internal state of all entities is set to $\perp$ (empty when initialized).

- $\mathsf{Publish}(\mathsf{sk}, \mathbf{u}, \mathsf{st}^*_{\mathsf{in}}) \to (\sigma, \mathsf{st}^*_{\mathsf{out}})$. The $\mathsf{Publish}$ algorithm takes as parameters a secret key sk, an update $\mathbf{u}$, and the input state $\mathsf{st}^*_{\mathsf{in}}$, and outputs a digest σ, and an updated state $\mathsf{st}^*_{\mathsf{out}}$. The distributor runs the publish algorithm to distribute an update $\mathbf{u}$ to the database, with the output σ being sent to its neighboring subscribers alongside the update contents $\mathbf{u}$. The distributor also updates its internal state from $\mathsf{st}^*_{\mathsf{in}}$ to $\mathsf{st}^*_{\mathsf{out}}$.[6]

- $\mathsf{GetUpdates}(\mathsf{pp}, \mathsf{v}, \mathsf{w}, \mathsf{st}) \to (\vec{\mathbf{u}}, \mu)$. The $\mathsf{GetUpdates}$ algorithm takes as input the public parameters pp, two sequence numbers $\mathsf{v}, \mathsf{w} \in \mathbb{Z}$, and an internal state st, and outputs a list of updates $\vec{\mathbf{u}}$ along with digest μ. A client subscriber that wishes to receive updates will send the sequence number v to a neighboring peer, who then computes the difference in updates associated with the most recent sequence number w and the client's sequence number v. This difference is represented by $\mathbf{u}$, which is returned to the client subscriber along with the digest μ.

- $\mathsf{ApplyUpdates}(\mathsf{pp}, \mathsf{v}, \mathsf{w}, \vec{\mathbf{u}}, \mu, \mathsf{st}_{\mathsf{in}}) \to (b, \mathsf{st}_{\mathsf{out}})$. The algorithm $\mathsf{ApplyUpdates}$ takes as input the public parameters, two sequence numbers $\mathsf{v}, \mathsf{w} \in \mathbb{Z}$, a list of updates $\vec{\mathbf{u}}$, a digest μ, and the input state $\mathsf{st}_{\mathsf{in}}$, to output a bit $b \in \{0, 1\}$ along with an updated internal state $\mathsf{st}_{\mathsf{out}}$. This is run by a client subscriber upon receiving a series of updates $\vec{\mathbf{u}}$ from a peer, and used to verify the authenticity of these updates before applying them to its replica of the database. Afterwards, the client subscriber also updates its internal state from $\mathsf{st}_{\mathsf{in}}$ to $\mathsf{st}_{\mathsf{out}}$.

- $\mathsf{Validate}(\mathsf{pp}, \mathcal{D}, \mathsf{v}, \mathsf{st}) \to b$. The $\mathsf{Validate}$ algorithm takes as input the public parameters, the database replica $\mathcal{D}$, a sequence number $\mathsf{v} \in \mathbb{Z}$, and an internal state st, and outputs a bit $b \in \{0, 1\}$. This is used by the subscriber as an integrity check of its own replica of the database against the distributor's copy.

[6] For ease of exposition, we use the asterisk for st^* to denote the distributor's state, as opposed to the subscribers' states which will be denoted without the asterisk.

Note that the distributor and each subscriber are stateful entities, with $\mathsf{st}^*_{\mathsf{in}}$ representing the distributor's state, and $\mathsf{st}_{\mathsf{in}}$ representing a subscriber state. The distributor state is updated on calls to Publish, whereas each subscriber's state is only updated on calls to ApplyUpdates. Furthermore, we note that only the distributor has access to sk, and hence is the only entity that can run Publish. The remaining algorithms only rely on the public parameters pp, and are run by the subscribers.

Multiple distributors. We present the update propagation model in the presence of a single distributor. In practice, there can be multiple distributors, each of whom hold the same secret signing key sk, and our formulation of update propagation extends naturally to the multiple-distributor scenario.

Resolving missing updates. When a client subscriber reaches out to a peer to obtain updates from a version v to a target version w, the peer is expected to be able to respond by using its internal state (which would store the sequence of updates necessary to respond correctly). However, if the peer does not have the necessary state to form the response, in practice, this peer will reach out with another GetUpdates call to a different peer in order to receive the necessary updates. This process could theoretically continue until the chain of requests reaches the distributor (who will have the appropriate update information to respond). In our model, we assume that this network behavior has a resolution, and that the peer *eventually* obtains the necessary state to respond to GetUpdates.[7]

Semantics. We use $\mathcal{D}' = \mathcal{D} + \mathbf{u}$ to denote the operation of "applying" a set of updates $\mathbf{u}$ to a database $\mathcal{D}$, to result in an updated database $\mathcal{D}'$. Similarly, from the associativity of the + operator, we write $\mathbf{u} = \mathbf{u}_1 + \mathbf{u}_2$ to denote the operation of combining two updates $\mathbf{u}_1$ and $\mathbf{u}_2$ to result in a batched update $\mathbf{u}$. For a list of updates $\vec{\mathbf{u}} = \langle \mathbf{u}_1, \ldots, \mathbf{u}_\ell \rangle$, we write $\mathtt{Sum}(\vec{\mathbf{u}}) = \mathbf{u}_1 + \cdots + \mathbf{u}_\ell$. In the definitions below, we fix a sequence of $\mathtt{n}$ updates $\mathbf{u}_1, \ldots, \mathbf{u}_\mathtt{n}$, and for each $i \in [1, \mathtt{n}]$, we define $\mathcal{D}_i = \mathcal{D}_{i-1} + \mathbf{u}_i$ with $\mathcal{D}_0 = \perp$ (the empty database).

A delta to a database can be either an addition or removal of a row in the database[8], and an update is simply a set of deltas. We refer to a *valid state* as a state which would arise from a normal (non-adversarial) execution of the Publish, GetUpdates, and ApplyUpdates algorithms. We formally define these terms, along with the notion of a *valid update*, in Appendix B.1.

Correctness. Fix a sequence of updates $\mathbf{u}_1, \ldots, \mathbf{u}_\mathtt{n}$, and let $\mathcal{D}_i = \mathbf{u}_1 + \cdots + \mathbf{u}_i$ for each $i \in [1, \mathtt{n}]$. Let $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, and for each $i \in [1, \mathtt{n}]$, let $(\sigma_i, \mathsf{st}^*_i) \leftarrow \mathsf{Publish}(\mathsf{sk}, \mathbf{u}_{i-1}, \mathsf{st}^*_{i-1})$. An update propagation scheme UP is *correct* if for every $i \in [1, \mathtt{n}]$, we have that:

1. With st_i as a valid state, and $(\vec{\mathbf{u}}, \mu) \leftarrow \mathsf{GetUpdates}(\mathsf{pp}, i, \mathtt{n}, \mathsf{st}_i)$, it is the case that $\mathcal{D}_\mathtt{n} = \mathcal{D}_i + \mathtt{Sum}(\vec{\mathbf{u}})$,
2. With st'_i as a valid state, $\mathsf{ApplyUpdates}(\mathsf{pp}, i, \mathtt{n}, \vec{\mathbf{u}}, \mu, \mathsf{st}'_i) \to (b, \mathsf{st}'_\mathtt{n})$, we have that $b = 1$, and
3. $\mathsf{Validate}(\mathsf{pp}, \mathcal{D}_i, i, \mathsf{st}_i) = 1$.

[7] The mechanism with which the system guarantees this eventuality is certainly an important consideration, but it is not relevant to the correctness or security of the update propagation problem, and hence we do not specify this in our model.

[8] In the context of defining deltas, a mutation to a database row can be modeled as a removal followed by an addition, executed atomically.

In other words, the update propagation scheme is correct if a client executing the ApplyUpdates algorithm on the output of a GetUpdates call from a peer in the network is successful whenever the received updates can be applied to the client's replica to match the expected database version originally reported by the distributor, and the Validate algorithm on the client's replica of the database also outputs success, so long as the replica matches the distributor's version.

Security. We define the notion of security for an update propagation scheme through an experiment between a challenger and an adversary $\mathcal{A}$ that can choose the database updates and can make Publish queries. We use λ as the security parameter.

Definition 3.1 ($\mathsf{Expt}^{\mathsf{up}}(\mathcal{A})$)**.** The experiment $\mathsf{Expt}^{\mathsf{up}}$ takes as input an adversary $\mathcal{A}$. The challenger starts the experiment by computing $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$ and sending pp to the adversary $\mathcal{A}$. The challenger maintains an index $\mathtt{n} \leftarrow 0$ which will represent the number of times the Publish oracle has been called. It keeps track of the sequence $\mathbf{u}_1^*, \ldots, \mathbf{u}_{\mathtt{n}}^*$ of updates from calls to the Publish oracle, with $\mathbf{u}_i^*$ representing the i^{th} update to the database. It also keeps track of each database version by setting $\mathcal{D}_0^* = \perp$ and defines $\mathcal{D}_i^* = \mathcal{D}_{i-1}^* + \mathbf{u}_i^*$.

The challenger also maintains two global states: st^* for the Publish oracle, and $\hat{\mathsf{st}}$ for the ApplyUpdates and Validate oracles, both initialized to $\perp$. Then, the challenger responds to each oracle query type made by $\mathcal{A}$ in the following manner:

- **Publish oracle.** This oracle is a stateful oracle that maintains the database with updates applied, along with an internal state. On input an update $\mathbf{u}$, the challenger updates $\mathtt{n}^* \leftarrow \mathtt{n}^* + 1$, computes $(\sigma, \mathsf{st}_{\mathsf{out}}^*) \leftarrow \mathsf{Publish}(\mathsf{sk}, \mathbf{u}, \mathsf{st}^*)$, storing the updated database with $T^*[\mathtt{n}] \leftarrow \mathbf{u}$, and updating the state $\mathsf{st}^* \leftarrow \mathsf{st}_{\mathsf{out}}^*$. The oracle responds with the output (σ, st^*).
- **ApplyUpdates oracle.** This oracle is a stateful oracle that takes as input v, w, $\vec{\mathbf{u}}$, and μ. Then, the challenger computes $(b, \mathsf{st}_{\mathsf{out}}) \leftarrow \mathsf{ApplyUpdates}(\mathsf{pp}, \mathsf{v}, \mathsf{w}, \vec{\mathbf{u}}, \mu, \hat{\mathsf{st}})$. If $b = 1$, then the challenger first stores $\hat{\mathsf{st}} \leftarrow \mathsf{st}_{\mathsf{out}}$, and then checks if $\mathtt{Sum}(\vec{\mathbf{u}}) \neq \mathbf{u}_{\mathsf{v}+1}^* + \cdots + \mathbf{u}_{\mathsf{w}}^*$, outputting Success if so. Otherwise, the challenger returns (b, st).
- **Validate oracle.** This oracle takes as input a database $\mathcal{D}$ and state $\mathsf{st}_{\mathsf{in}}$, which the challenger uses to compute $b \leftarrow \mathsf{Validate}(\mathsf{pp}, \mathcal{D}, \mathtt{n}, \mathsf{st}_{\mathsf{in}})$. If $b = 1$, the challenger checks if $\mathcal{D} \neq \mathcal{D}_{\mathtt{n}}^*$, outputting Success if so. Otherwise, the challenger returns b.

Note that the experiment either outputs nothing, or outputs Success from an ApplyUpdates or Validate oracle query. We define the advantage of $\mathcal{A}$ in $\mathsf{Expt}^{\mathsf{up}}$ for an update propagation scheme Π as

$$\mathsf{Adv}_{\mathsf{up}}(\Pi, \mathcal{A}) \coloneqq \Pr[\mathsf{Expt}^{\mathsf{up}}(\mathcal{A}) = \mathsf{Success}].$$

We say that Π is *secure* if for all efficient adversaries $\mathcal{A}$, $\mathsf{Adv}_{\mathsf{up}}(\Pi, \mathcal{A})$ is negligible in the security parameter λ.

Intuitively, the experiment $\mathsf{Expt}^{\mathsf{up}}$ models an adversary that can choose any sequence of updates, maliciously simulate the GetUpdates algorithm, and alter the database or state input to the Validate algorithm. The adversary's queries to the Publish oracle are used along with the challenger's secret key to mark the database updates submitted by the adversary as valid. The aim of this adversary is to force the challenger (acting as an honest subscriber) into accepting a list of updates which is actually invalid, or validating a database state which is actually invalid. When the adversary is successful in doing so, the challenger outputs Success, which is also the output of the experiment.

In other words, our model accounts for an adversary that unable to access the secret key sk and unable to modify the distributor's internal state st^*, but is able to choose the updates, modify the internal state of any subscriber and the communication between peers, with the goal of fooling an uncompromised subscriber into applying an update which is invalid, or validating a database which is invalid. Note that our security model also includes adversaries which can corrupt multiple subscribers—still, the adversary wins the experiment if it is able to produce invalid behavior on a subscriber that the adversary does not control.

3.2 Signing with Homomorphic Hashing

In this section we describe our construction of secure update propagation based on homomorphic hashing, which we denote as Π_{hash}.

Recall from Section 2 that a set homomorphic hash $\mathsf{H} : \{0,1\}^* \to \mathsf{G}$ for a commutative group $(\mathsf{G}, \circ)$ is collision resistant and satisfies the property that for any two disjoint sets $\mathcal{S}, \mathcal{T} \in \mathcal{P}(\{0,1\}^*)$, it is the case that $\mathsf{H}(\mathcal{S} \cup \mathcal{T}) = \mathsf{H}(\mathcal{S}) \circ \mathsf{H}(\mathcal{T})$.

Homomorphically hashing updates. We define H acting on a delta δ as follows:

- **Row addition:** If δ corresponds to a row addition $(i, \perp \to x)$, we define $\mathsf{H}(\delta)$ as $\mathsf{H}(\{(i,x)\})$.
- **Row deletion:** If δ corresponds to a row deletion $(i, x \to \perp)$ we define $\mathsf{H}(\delta)$ as $\mathsf{H}(\{(i,x)\})^{-1}$.

(Further details can be found in Appendix B.1.) For an update $\mathbf{u}$ represented by a sequence of deltas $\delta_1, \ldots, \delta_m$, we define $\mathsf{H}(\mathbf{u}) = \mathsf{H}(\delta_1) \circ \cdots \circ \mathsf{H}(\delta_m)$, and $\mathsf{Set}(\mathbf{u}) = \{\delta_1, \ldots, \delta_m\}$. For a database $\mathcal{D}$ that can be expressed as a sequence of row addition deltas, we define $\mathsf{H}(\mathcal{D})$ in the same manner, and $\mathsf{Set}(\mathcal{D})$ to be the set of row addition deltas.

Description of Π_{hash}. The distributor maintains an internal state $\mathsf{st}^* = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{hash}}, \mathsf{T}_{\mathsf{sig}}, \mathtt{n}^*)$, corresponding to: a table $\mathsf{T}_{\mathsf{update}}$ of updates, a table $\mathsf{T}_{\mathsf{hash}}$ where each index i consists of a hash of a sum of updates, a table $\mathsf{T}_{\mathsf{sig}}$ where each index i will be the signature of $\mathsf{T}_{\mathsf{hash}}[i]$, and a sequence number $\mathtt{n}^*$. The subscribers each maintain an internal state $\mathsf{st} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{hash}}, \mathsf{T}_{\mathsf{sig}})$ consisting of a list of sums of updates $\mathsf{T}_{\mathsf{update}}$, a list of hashes $\mathsf{T}_{\mathsf{hash}}$, and a list of signatures $\mathsf{T}_{\mathsf{sig}}$ (defined similar to as in st^*), all initialized to $\perp$. We define $\Pi_{\mathsf{hash}} = (\mathsf{Setup}, \mathsf{Publish}, \mathsf{GetUpdates}, \mathsf{ApplyUpdates}, \mathsf{Validate})$ as follows:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{sk})$. The Setup algorithm sets $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{Sig.Setup}(1^\lambda)$ and sets $\mathtt{n}^* \leftarrow 0$.
- $\mathsf{Publish}(\mathsf{sk}, \mathbf{u}, \mathsf{st}^*_{\mathsf{in}} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{hash}}, \mathsf{T}_{\mathsf{sig}}, \mathtt{n}^*)) \to (\sigma, \mathsf{st}^*_{\mathsf{out}})$. The $\mathsf{Publish}$ algorithm sets

$$\mathtt{n}^* \leftarrow \mathtt{n}^* + 1, \quad h \leftarrow \mathsf{T}_{\mathsf{hash}}[\mathtt{n}^* - 1] \circ \mathsf{H}(\mathbf{u}), \quad \sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}, (h, \mathtt{n}^*)),$$
$$\mathsf{T}_{\mathsf{hash}}[\mathtt{n}^*] \leftarrow h, \quad \mathsf{T}_{\mathsf{sig}}[\mathtt{n}^*] \leftarrow \sigma, \quad \mathsf{st}^*_{\mathsf{out}} \leftarrow (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{hash}}, \mathsf{T}_{\mathsf{sig}}, \mathtt{n}^*),$$

  and outputs $(\sigma, \mathsf{st}^*_{\mathsf{out}})$.
- $\mathsf{GetUpdates}(\mathsf{pp}, \mathsf{v}, \mathsf{w}, \mathsf{st} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{hash}}, \mathsf{T}_{\mathsf{sig}})) \to (\vec{\mathbf{u}}, \mu)$. The $\mathsf{GetUpdates}$ algorithm outputs

$$(\vec{\mathbf{u}}, \mu) \leftarrow (\langle \mathsf{T}_{\mathsf{update}}[\mathsf{v}], \ldots, \mathsf{T}_{\mathsf{update}}[\mathsf{w}] \rangle, \mathsf{T}_{\mathsf{sig}}[\mathsf{w}]).$$

- $\mathsf{ApplyUpdates}(\mathsf{pp}, \mathsf{v}, \mathsf{w}, \vec{\mathbf{u}} = \langle \mathbf{u}_{\mathsf{v}+1}, \ldots, \mathbf{u}_{\mathsf{w}} \rangle, \mu, \mathsf{st}_{\mathsf{in}} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{hash}}, \mathsf{T}_{\mathsf{sig}})) \rightarrow (b, \mathsf{st}_{\mathsf{out}})$. The $\mathsf{ApplyUpdates}$ algorithm checks that $\mathsf{v} < \mathsf{w}$, sets $s = \mathsf{T}_{\mathsf{hash}}[\mathsf{v}] \circ (\mathsf{H}(\mathbf{u}_{\mathsf{v}+1}) \circ \cdots \circ \mathsf{H}(\mathbf{u}_{\mathsf{w}}))$, and checks

$$\mathsf{Sig.Verify}(\mathsf{pp}, (s, \mathsf{w}), \mu) = 1.$$

  If so, it sets $b = 1$, for each $j \in [\mathsf{v}+1, \mathsf{w}]$ it sets $\mathsf{T}_{\mathsf{update}}[j] \leftarrow \mathbf{u}_j$, sets $(\mathsf{T}_{\mathsf{hash}}[\mathsf{w}], \mathsf{T}_{\mathsf{sig}}[\mathsf{w}]) \leftarrow (s, \mu)$, and updates $\mathsf{st}_{\mathsf{out}} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{hash}}, \mathsf{T}_{\mathsf{sig}})$. Otherwise, it sets $b = 0$ (without updating $\mathsf{st}_{\mathsf{out}}$). In both cases, it outputs $(b, \mathsf{st}_{\mathsf{out}})$.

- $\mathsf{Validate}(\mathsf{pp}, \mathcal{D}, \mathsf{v}, \mathsf{st} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{hash}}, \mathsf{T}_{\mathsf{sig}})) \rightarrow b$. The $\mathsf{Validate}$ algorithm checks that $\mathsf{T}_{\mathsf{hash}}[\mathsf{v}] = \mathsf{H}(\mathcal{D})$, and then checks that $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathsf{T}_{\mathsf{hash}}[\mathsf{v}], \mathsf{v}), \mathsf{T}_{\mathsf{sig}}[\mathsf{v}]) = 1$. If so, it outputs $b = 1$, and outputs $b = 0$ otherwise.

Correctness. To prove correctness of Π_{hash}, we note that the internal state tables $\mathsf{T}_{\mathsf{update}}$, $\mathsf{T}_{\mathsf{hash}}$ and $\mathsf{T}_{\mathsf{sig}}$ kept by each subscriber maintain the invariant that every entry in $\mathsf{T}_{\mathsf{hash}}$ has a corresponding signature over its contents in $\mathsf{T}_{\mathsf{sig}}$, and for each $i \in [1, \mathtt{n}]$, $\mathsf{T}_{\mathsf{hash}}[i] = \mathsf{H}(\mathsf{T}_{\mathsf{update}}[1]) \circ \cdots \circ \mathsf{H}(\mathsf{T}_{\mathsf{update}}[i])$. Fix a sequence of updates $\mathbf{u}_1, \ldots, \mathbf{u}_{\mathtt{n}}$, and let $\mathcal{D}_i = \mathbf{u}_1 + \cdots + \mathbf{u}_i$ for each $i \in [1, \mathtt{n}]$. Hence, for $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, with $(\sigma_i, \mathsf{st}_i^*) \leftarrow \mathsf{Publish}(\mathsf{sk}, \mathbf{u}_i, \mathsf{st}_{i-1}^*)$, for each $i \in [1, \mathtt{n}]$, we note that:

1. Using $(\vec{\mathbf{u}}, \mu) \leftarrow \mathsf{GetUpdates}(\mathsf{pp}, i, \mathtt{n}, \mathsf{st}_i)$ we have that

$$\vec{\mathbf{u}} = \langle \mathsf{T}_{\mathsf{update}}[i], \ldots, \mathsf{T}_{\mathsf{update}}[\mathtt{n}] \rangle,$$

   which by the definition of $\mathsf{Publish}$ means that $\vec{\mathbf{u}} = \langle \mathbf{u}_i, \ldots, \mathbf{u}_{\mathtt{n}} \rangle$, which by definition are exactly the updates for which $\mathcal{D}_i + \mathtt{Sum}(\vec{\mathbf{u}}) = \mathcal{D}_{\mathtt{n}}$.

2. We have that $\mathsf{T}_{\mathsf{hash}}[\mathsf{v}] \circ \mathsf{H}(\mathbf{u}_{\mathsf{v}+1}) \circ \cdots \circ \mathsf{H}(\mathbf{u}_{\mathtt{n}}) = \mathsf{T}_{\mathsf{hash}}[\mathtt{n}]$ from the definition of the $\mathsf{Publish}$ algorithm and the set homomorphism of H. Therefore, ApplyUpdates runs $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathsf{T}_{\mathsf{hash}}[\mathtt{n}], \mathtt{n}), \mathsf{T}_{\mathsf{sig}}[\mathtt{n}]) = 1$, by the correctness of the signature scheme. Thus, the $\mathsf{ApplyUpdates}$ call outputs with $b = 1$ as desired.

3. By the set homomorphism of H, $\mathsf{T}_{\mathsf{hash}}[i] = \mathsf{H}(\mathbf{u}_1) \circ \cdots \circ \mathsf{H}(\mathbf{u}_i) = \mathsf{H}(\mathcal{D}_i)$, and we established that $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathsf{T}_{\mathsf{hash}}[i], i), \mathsf{T}_{\mathsf{sig}}[i]) = 1$ by the correctness of the signature scheme. Therefore, we conclude that $\mathsf{Validate}(\mathsf{pp}, \mathcal{D}_i, i, \mathsf{st}_i) = 1$.

Security. We show that Π_{hash} is a secure update propagation scheme by defining an intermediate experiment and constructing two simulators which can act as adversaries to find collisions in H or forge signatures for Π_{Sig}, given an adversary for breaking the security of Π_{hash}.

Theorem 3.2. *If Π_{Sig} is a signature scheme which is existentially unforgeable under a chosen message attack, and H is a homomorphic hash function, then Π_{hash} is a secure update propagation scheme.*

Proof. To prove security, we define a new experiment $\widetilde{\mathsf{Expt}}$ which is very similar to $\mathsf{Expt}^{\mathsf{up}}$, except that it can output the event Bad if a collision is found as the first step of the $\mathsf{ApplyUpdates}$ and $\mathsf{Validate}$ oracle.

The experiment $\widetilde{\mathsf{Expt}}$ takes as input an adversary $\mathcal{A}$. The challenger starts the experiment identically to $\mathsf{Expt}^{\mathsf{up}}(\mathcal{A})$, by computing $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$ and sending pp to the adversary $\mathcal{A}$.

The challenger maintains an index $\mathtt{n} \leftarrow 0$ which will represent the number of times the Publish oracle has been called. It keeps track of the sequence $\mathbf{u}_1^*, \ldots, \mathbf{u}_{\mathtt{n}}^*$ of updates from calls to the Publish oracle, with $\mathbf{u}_i^*$ representing the i^{th} update to the database. It also keeps track of each database version by setting $\mathcal{D}_0^* = \perp$ and defines $\mathcal{D}_i^* = \mathcal{D}_{i-1}^* + \mathbf{u}_i^*$.

The challenger also maintains two global states: st^* for the Publish oracle, and $\hat{\mathsf{st}}$ for the ApplyUpdates and Validate oracles, both initialized to $\perp$. Then, the challenger responds to each oracle query type made by $\mathcal{A}$ in the following manner:

- **Publish oracle.** This oracle behaves identically as the Publish oracle of $\mathsf{Expt}^{\mathsf{up}}(\mathcal{A})$.
- **ApplyUpdates oracle.** This oracle is a stateful oracle that takes as input v, w, $\vec{\mathbf{u}}$, and μ. It checks if $\mathtt{Sum}(\vec{\mathbf{u}}) \neq \mathbf{u}_{\mathsf{v}+1}^* + \cdots + \mathbf{u}_{\mathsf{w}}^*$ and $\mathsf{H}(\mathtt{Sum}(\vec{\mathbf{u}})) = \mathsf{H}(\mathbf{u}_{\mathsf{v}+1}^*) \circ \cdots \circ \mathsf{H}(\mathbf{u}_{\mathsf{w}}^*)$, outputting Bad if so. Otherwise, the challenger simulates the ApplyUpdates oracle of $\mathsf{Expt}^{\mathsf{up}}(\mathcal{A})$.
- **Validate oracle.** This oracle takes as input a database $\mathcal{D}$ and state $\mathsf{st}_{\mathsf{in}}$. It checks if $\mathcal{D} \neq \mathcal{D}_{\mathtt{n}}^*$ and $\mathsf{H}(\mathcal{D}) = \mathsf{H}(\mathcal{D}_{\mathtt{n}}^*)$, outputting Bad if so. Otherwise, the challenger simulates the Validate oracle of $\mathsf{Expt}^{\mathsf{up}}(\mathcal{A})$.

For an efficient adversary $\mathcal{A}$, we define the advantage of $\mathcal{A}$ in $\widetilde{\mathsf{Expt}}$ for an update propagation scheme Π as

$$\mathsf{Adv}_{\mathsf{Sim}}(\Pi, \mathcal{A}) \coloneqq \Pr[\widetilde{\mathsf{Expt}}(\mathcal{A}) = \mathsf{Success}].$$

We define a simulator Sim_1 that can participate as a challenger in $\widetilde{\mathsf{Expt}}$ and an adversary in $\mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}$. Intuitively, Sim_1 behaves identically to $\widetilde{\mathsf{Expt}}$, with two differences:

- Instead of computing $\mathsf{H}(\cdot)$ on its own, Sim_1 submits oracle queries to the challenger of $\mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}$.
- Instead of outputting the Bad event in $\widetilde{\mathsf{Expt}}$, Sim_1 submits the sets $\mathcal{S} = \mathsf{Set}(\mathtt{Sum}(\mathbf{u}))$ and $\mathcal{T} = \mathbf{u}_{\mathsf{v}+1}^* + \cdots + \mathbf{u}_{\mathsf{w}}^*$ (in the case of the ApplyUpdates oracle), or $\mathcal{S} = \mathsf{Set}(\mathcal{D})$ and $\mathcal{T} = \mathsf{Set}(\mathcal{D}_{\mathtt{n}}^*)$ (in the case of the Validate oracle).

Lemma 3.3. *For all efficient adversaries $\mathcal{A}$, we have that*

$$|\mathsf{Adv}_{\mathsf{up}}(\Pi_{\mathsf{hash}}, \mathcal{A}) - \mathsf{Adv}_{\mathsf{Sim}}(\Pi_{\mathsf{hash}}, \mathcal{A})| \leq \mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}(\mathsf{H}, \mathsf{Sim}_1).$$

Proof. By construction, note that $\widetilde{\mathsf{Sim}}_1$ perfectly simulates $\mathsf{Expt}^{\mathsf{up}}$ when $\mathcal{A}$ never triggers the Bad event, and it perfectly simulates $\widetilde{\mathsf{Expt}}$ when $\mathcal{A}$ is able to trigger the Bad event. When the Bad event is triggered, the pair of input sets submitted to the challenger for $\mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}$ result in collisions, by definition. Hence, we have that the probability that Bad is triggered is precisely equal to $\mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}(\mathsf{H}, \mathsf{Sim})$, and the former quantity is bounded by the difference in advantage between $\mathsf{Expt}^{\mathsf{up}}$ and $\widetilde{\mathsf{Expt}}$. □

For the next step of the proof, we define a simulator Sim_2 which acts as a challenger in $\widetilde{\mathsf{Expt}}$ and an adversary in $\mathsf{Expt}_{\mathsf{Sig}}$. For a sequence of $\mathtt{n}$ updates, the simulator Sim_2 first receives the verification key vk from the challenger of $\mathsf{Expt}_{\mathsf{Sig}}$, which it forwards to $\mathcal{A}$ as the public parameters pp. Sim_2 also initializes an empty database $\mathcal{D}^* = \perp$. Then, for each type of oracle query that the adversary makes, Sim_2 responds as follows:

- **Publish oracle.** The simulator Sim_2, on input an update $\mathbf{u}$, first checks if the update $\mathbf{u}$ has been submitted before, returning the same signature response if it is a repeat. Otherwise, it simply forwards $\mathsf{T}_{\mathsf{hash}}[\mathtt{n}-1] \circ \mathsf{H}(\mathbf{u})$ to the challenger of $\mathsf{Expt}_{\mathsf{Sig}}$ to receive a signature σ. This is returned as the digest σ, and the internal state st^* is updated appropriately. For the i^{th} call to the Publish oracle, the input $\mathbf{u}$ is labeled as $\mathbf{u}_i^*$, and we define $\mathcal{D}_i^* = \mathcal{D}_{i-1}^* + \mathbf{u}_i^*$.
- **ApplyUpdates oracle.** This oracle can be completely simulated by Sim_2 since it does not require access to sk. If Sim_2 outputs $\mathsf{Success}$, then Sim_2 sets $\mathsf{m} = (\mathsf{T}_{\mathsf{hash}}[\mathsf{v}] \circ \mathsf{H}(\mathtt{Sum}(\vec{\mathbf{u}})), \mathsf{w})$ and submits the message-signature pair (m, μ) to the challenger for $\mathsf{Expt}_{\mathsf{Sig}}$, ending the experiment.
- **Validate oracle.** Again, this oracle can be simulated by Sim_2 using pp. If Sim_2 outputs success, then the simulator submits the message-signature pair $((\mathsf{H}(\mathcal{D}), \mathtt{n}), \mathsf{T}_{\mathsf{sig}}[\mathtt{n}])$ to the challenger for $\mathsf{Expt}_{\mathsf{Sig}}$, ending the experiment.

Lemma 3.4. *For all efficient adversaries $\mathcal{A}$, $\mathsf{Adv}_{\mathsf{Sim}}(\Pi_{\mathsf{hash}}, \mathcal{A}) = \mathsf{Adv}_{\mathsf{Sig}}(\Pi_{\mathsf{Sig}}, \mathsf{Sim}_2)$.*

Proof. Note that the only signature oracle queries that Sim_2 makes to its challenger are on the hashes $\mathsf{H}(\cdot)$ of the databases $\mathcal{D}_1^*, \ldots, \mathcal{D}_{\mathtt{n}}^*$ that the Publish oracle computes. There are two cases in which the experiment outputs $\mathsf{Success}$:

- **ApplyUpdates oracle.** In the event of $\mathsf{Success}$ from ApplyUpdates, Sim submits the message-signature pair (m, μ) to the challenger. Since the Bad event did not occur, it must be the case that $\mathsf{H}(\mathcal{D}_{\mathsf{v}}^*) \circ \mathsf{H}(\mathtt{Sum}(\mathbf{u})) \neq \mathsf{H}(\mathcal{D}_{\mathsf{v}}^*) \circ \mathsf{H}(\mathbf{u}_{\mathsf{v}+1}^* + \cdots + \mathbf{u}_{\mathsf{w}}^*) = \mathsf{H}(\mathcal{D}_{\mathsf{w}}^*)$, for every $\mathsf{v}, \mathsf{w} \in [1, \mathtt{n}]$. Therefore, we can conclude that m is distinct from all previous signature oracle queries sent to the challenger. Also, Sim outputting $\mathsf{Success}$ means that $\mathsf{Sig.Verify}(\mathsf{vk}, \mathsf{m}, \mu) = 1$ by definition.
- **Validate oracle.** In the event of $\mathsf{Success}$ from Validate, Sim submits the message-signature pair $((\mathsf{H}(\mathcal{D}), \mathtt{n}), \mathsf{T}_{\mathsf{sig}}[\mathtt{n}])$ to the challenger. Again, since the Bad event did not occur, then we have that $\mathsf{H}(\mathcal{D}) \neq \mathsf{H}(\mathbf{u}_1^*) \circ \cdots \circ \mathsf{H}(\mathbf{u}_{\mathtt{n}}^*) = \mathsf{H}(\mathcal{D}_{\mathtt{n}}^*)$ is distinct from previous signature oracle queries sent to the challenger, and since Sim is outputting $\mathsf{Success}$, this already means that $\mathsf{Sig.Verify}(\mathsf{vk}, (\mathsf{H}(\mathcal{D}), \mathtt{n}), \mathsf{T}_{\mathsf{sig}}[\mathtt{n}]) = 1$ by definition.

In both cases, the pair submitted by Sim to the challenger matches the criteria for which $\mathsf{Expt}_{\mathsf{Sig}}(\mathsf{Sim})$ outputs 1. □

Putting Lemmas 3.3 and 3.4 together, we see that

$$\mathsf{Adv}_{\mathsf{up}}(\Pi_{\mathsf{hash}}, \mathcal{A}) \leq \mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}(\mathsf{H}, \mathsf{Sim}_1) + \mathsf{Expt}_{\mathsf{Sig}}(\Pi_{\mathsf{Sig}}, \mathsf{Sim}_2),$$

which concludes the proof of Theorem 3.2. □

3.3 Performance Comparison

In this section, we compare the efficiency of four different secure update propagation schemes:

- Π_{update}, signing each update directly (Appendix B.3),
- Π_{db}, signing the whole database (Appendix B.4),
- Π_{hash}, using homomorphic hashing (Section 3.2), and
- Π_{merkle}, based on Merkle trees and described below.

Limiting subscriber updates. In our definition of an update propagation scheme, we represented the state held by subscribers which is passed as input to GetUpdates to generate a list of updates, and also passed into ApplyUpdates, which mutates the state and allows the subscriber to retain the updates passed into it. Similarly, the internal subscriber state is an input to the Validate algorithm, allowing the subscriber to verify the integrity of its database replica.

In order to answer arbitrary GetUpdates requests, a subscriber must keep the entire list of all updates $\mathbf{u}_1, \ldots, \mathbf{u}_n$, by the correctness requirements of the scheme. However, in practice subscribers rarely get requests for updates far in the past. Indeed, it is much more efficient for each subscriber to manage a fixed-size queue of the most recent updates it has received. We can therefore enforce an integer cap $\mathbf{c}$ on the number of updates held by each subscriber, and when a subscriber is unable to answer a GetUpdates query with its update queue, it can simply forward the client to the distributor to obtain the necessary updates.

The appropriate setting of $\mathbf{c}$ depends on network behavior. If $\mathbf{c}$ is too small, then arbitrary network outages could cause a "thundering herd" effect where many subscribers are frequently needing to connect to the distributor to receive updates, which puts stress on the distributor's bandwidth. If $\mathbf{c}$ is too large, then each subscriber ends up having a longer-than-necessary list of updates in memory.

Table 2 shows a comparison of these constructions, considering three variables: $\mathbf{n}$ to represent the size of the database (total number of database rows), $\mathbf{m}$ to represent the total number of updates to the database, and $\mathbf{c}$ to represent the maximum number of updates that each subscriber can hold without overwhelming the distributor with failover traffic.

Discussion. We measure the overall performance of Publish and ApplyUpdates, the space requirements on the distributor and subscriber, and the number of signature validations performed in ApplyUpdates and Validate (since the CPU overhead of public-key cryptographic operations tends to be orders of magnitude more than the other operations, including hashing). We find that Π_{update} is unsurprisingly efficient for publishing updates, but is inefficient in terms of space overhead and executing the Validate procedure. The Π_{db} construction does not require storing per-update signatures and hence is more space efficient and involves fewer signature validations, but since the signatures involve iterating over the entire database just to sign or verify, it is quite suboptimal in performance.

In this comparison, we also describe a scheme Π_{merkle} which we do not define formally, but we can think of as being similar to Π_{db}, except that instead of recomputing the database signature from scratch on each update, we keep an in-memory Merkle tree representation of hashes, where each database row corresponds to a leaf node in this tree. The main advantage to this tree construction is that now both the distributor and the subscriber can use the intermediate hashes to modify the root hash for an update in time logarithmic in the size of the database. The drawback is that both parties must now keep these intermediate hashes in memory, which takes space linear in the total number of database rows.

Finally, we note that Π_{hash} is able to avoid all of the aforementioned pitfalls, with only a dependence on $\mathbf{c}$ for the space overhead on each subscriber (necessary as the subscriber must keep $\mathbf{c}$ updates in memory to respond to GetUpdates queries). The main benefit of using homomorphic hashing is that we can obtain all of the performance benefits of Π_{db} without having to scan the entire database to compute or validate signatures, thereby removing the dependence on the database

Parameter	Π_{update}	Π_{db}	Π_{merkle}	Π_{hash}
Publish Time	$O(1)$	$O(\mathbf{n})$	$O(\log \mathbf{n})$	$O(1)$
ApplyUpdates Time	$O(\mathbf{c})$	$O(\mathbf{n})$	$O(\mathbf{c} \log \mathbf{n})$	$O(1)$
Distributor Space	$O(\mathbf{m})$	$O(1)$	$O(\mathbf{n})$	$O(1)$
Subscriber Space	$O(\mathbf{m})$	$O(\mathbf{c})$	$O(\mathbf{n})$	$O(\mathbf{c})$
# Sigs in ApplyUpdates	$\mathbf{c}$	1	1	1
# Sigs in Validate	$\mathbf{m}$	1	1	1

Table 2: Comparison of the performance of our update propagation schemes. We use $\mathbf{n}$ for the number of database rows, $\mathbf{m}$ for the total number of updates, and $\mathbf{c}$ for the total number of "recent" updates each subscriber keeps in memory.

size for the Publish and ApplyUpdates running times.

4 Implementation and Performance

In this section, we discuss our open-source implementation of LtHash including our performance optimizations, benchmarking results, and deployment considerations.

Implementation. To implement LtHash, we instantiate the underlying hash function with Blake2xb [ANWOW16], which is an extendable-output function (XOF) [Nat15]. Our Blake2xb implementation is built on top of the Blake2b primitives provided by Libsodium [BLS12] and is available in the Folly open-source library. Note that Libsodium is an optional dependency for Folly, and our implementations of Blake2xb and LtHash will be compiled only if the configure script finds a Libsodium installation at compile time. We also use the `IOBuf` and `ByteRange` classes from Folly to simplify the passing of input parameters and the memory management of buffers.

Recall that in our description of LtHash from Section 2, we use two parameters d and n to represent the base-2 log (number of bits) of the modulus and the number of vector components per hash output. An LtHash object contains an `IOBuf` of sufficient length to store n components, each consisting of d bits, as a contiguous buffer. We refer to this buffer as the checksum. LtHash defines the following methods:

- `hashObject` (private): This method takes a `ByteRange` input and a `MutableByteRange` output. It evaluates Blake2xb on the input and writes the resulting hash to the output.
- `addObject`: Calls `hashObject` on the input, then interprets the result as n components, each consisting of d bits, performs a component-wise vector addition of the result and checksum, and writes the result to checksum.
- `removeObject`: Calls `hashObject` on the input, then interprets the result as n components, each consisting of d bits, performs a component-wise vector subtraction of the result from the checksum, and writes the result to checksum.

We also define the plus, minus, assignment, and equality operators over LtHash objects, which function exactly as one would expect. Plus and minus perform component-wise vector addition

or subtraction of the checksum values of the two LtHash operands. Assignment sets the checksum value of the destination LtHash to the checksum value of the source LtHash. Equality tests if the checksum values of the two LtHash operands are equal in a data-independent way to avoid leaking information through timing side channels.

Parameterizing word sizes. Our main implementation uses $(d, n) = (16, 1024)$, and our concrete security analysis from Section 2.4 shows that this parameter choice yields at least 200 bits of security. As a precaution, we also implemented support for larger settings of d for applications that prefer to use an even more conservative setting of parameters. Recall that in Section 2.4, we use `lthash16` to refer to the setting of $(d, n) = (16, 1024)$, `lthash20` for $(d, n) = (20, 1008)$, and `lthash32` for $(d, n) = (32, 1024)$.

SIMD implementations. We implemented three different execution engines to perform the vector additions and subractions. The `SIMPLE` engine uses standard C++ code and is the least efficient, but is portable and should work on any CPU which supports 64-bit integer math (however, we have only tested it on Intel x86-64 and 64-bit ARM CPUs). The `SSE2` engine uses Intel's SSE2 instructions to add or subtract 128 bits at a time, and the `AVX2` engine uses Intel's AVX2 instructions to add or subtract 256 bits at a time. Both `SSE2` and `AVX2` are types of SIMD ("Single Instruction Multiple Data") instruction sets and are commonly used to accelerate algorithms which need to perform the same operation repeatedly on many independent data elements. While we did not implement any SIMD engines for non-Intel CPUs, the code is structured in a way that makes it easy to add additional execution engines in the future. Adding SIMD support for ARM or other non-Intel CPUs is a possible future area of work.

Optimizations. We found that aligning data buffers along cache line boundaries improves the performance of the vector operations used in `addObject` and `removeObject` by approximately 10%, and hence we make a best effort to allocate the buffers which store the Blake2xb hash outputs on a cache line boundary. In our implementation, we have hard-coded the cache line size to be 64 bytes, the correct value for modern Intel CPUs. Dynamically detecting the cache line size could be implemented in the future to better support other architectures.

To further optimize the hashing computation, we aimed to minimize the length of the Blake2xb output while still supporting enough bits for each of `lthash16`, `lthash20`, and `lthash32`. Our parameter choices were selected to work well with processors that support a 64-bit word size, in a way that allows us to pack as many vector components into each word as possible. For `lthash16` this was simple, since we could pack 4 components into each 64-bit word, relying on mod 16 addition and subtraction SIMD instructions to perform the component-wise arithmetic operations efficiently. The case for `lthash32` is analogous, except that we only pack 2 components into each 64-bit word and use SIMD instructions that perform mod 32 addition/subtraction. The implementation of `lthash20` is somewhat different because 64 does not evenly divide by 20. We implement `lthash20` efficiently by adding padding bits between the vector elements so for any 64-bit word in the checksum, there are three 20-bit data elements, separated by padding bits which are always set to 0. A 64-bit word is laid out as follows: `00[20 bits of data]0[20 bits of data]0[20 bits of data]`. This allows us to perform the addition and subtraction operations on 64 / 128 / 256 bits of data at a time and then perform another quick operation to zero out the padding bits in case any of the 20-bit elements overflowed and resulted in a carry that set the adjacent padding bit to 1.

	Simple	SSE*	AVX2
`lthash16.hashObject`	7214	6786	6187
`lthash16.add / remove`	452	125	81
`lthash20.hashObject`	9382	8260	7949
`lthash20.add / remove`	372	177	91
`lthash32.hashObject`	14145	13253	11053
`lthash32.add / remove`	906	230	137

Table 3: Our experimental benchmarks (measured in nanoseconds of wall time) for the performance of our LtHash implementations. For SSE*, the `hashObject` implementation used the SSSE3 instruction set for computing Blake2xb, while the vector addition and subtraction operations were done with SSE2.

4.1 Benchmarking Results

We ran our benchmarks on a 2.4GHz Intel Skylake CPU with 16MiB L3 cache. The test CPU had hardware and OS support for all three of our execution engines, so we used compile-time flags to disable SSE2 and AVX2 support, which allowed us to force a particular execution engine to be used. We also forced Libsodium to disable AVX2 or SSSE3 support for its Blake2b implementation to accurately simulate the performance on a CPU which does not support those SIMD instructions. Our benchmarks are presented in Table 3. We measured the performance of `lthash16`, `lthash20`, and `lthash32`, in nanoseconds of wall time, for each of the operations `hashObject`, `add`, and `remove`. Note that when measuring `add` and `remove`, we isolated the running time of the vector addition and subtraction operations, so it does not include the time of running `hashObject` (unlike the `addObject` and `removeObject` operations defined above). All benchmarks used a random 150-byte array as the hash input. Folly's high-quality benchmarking functions were used to reduce noise and provide more accurate performance numbers. However, a more controlled benchmark that turns off CPU frequency scaling and measures CPU cycles rather than wall time could provide even more precise results.

Vector operations. The running time of adding two hashes (corresponding to a set union of the hash inputs) was an order of magnitude less than the running time of computing the Blake2xb hashes. We found that for a given combination of (d, n) parameters, AVX2 was always faster than SSE2 which was faster than `SIMPLE`, as expected. We also found that the running times of `hashObject`, `add`, and `remove` scaled approximately linearly with the length of the Blake2xb hash, which is also expected. We observed that subtraction (corresponding to a set difference of the hash inputs) exhibited similar behavior with essentially the same performance as addition.

4.2 Deployment

Our implementation of `lthash16` has been deployed in production at Facebook to secure update propagation across the network.

Handling multiple execution engines. As we rolled out the implementation, we had to account for a diverse set of newer and older CPUs across the fleet. In particular, our initial implementations

triggered illegal instruction crashes that were a result of the implementation attempting to perform vector operations on the hash values using AVX2 instructions that were unavailable on certain older CPUs. We addressed this issue by adding support for an array of execution engines and auto-detecting the fastest supported engine at runtime when the LtHash library was initialized.

Fail-open rollout. As the homomorphic hash construction was introduced into the update propagation mechanism, the rollout was initially left in a "fail-open" mode, which meant that the propagation of an update would still occur despite a hash discrepancy or signature validation failure. This not only gave us the opportunity to increase confidence in the correctness of the implementation of `lthash16`, but it also helped to detect and fix logical inconsistencies within the update propagation mechanism itself.

After addressing these system bugs and monitoring the rate of these inconsistencies for a sufficiently long period of time, the integrity checks were then switched to a "fail-closed" mode, preventing the propagation of updates and triggering system alarms if the hashes or signatures were invalid. We believe such a two-phase approach would lead to a safe and incident-free rollout of secure update propagation in other systems as well.

5 Conclusion

We presented a formal definition of the update propagation problem and showed how homomorphic hashing can be applied to achieve a secure, efficient update propagation scheme. We then focused on the security of LtHash and its collision resistance in the random oracle model, based on the hardness of the Short Integer Solutions problem in lattice-based cryptography. Along with a concrete security analysis, we instantiated LtHash with parameters that produce a 2KB output hash, and produced benchmarks measuring its performance. Our implementation has been deployed in production at Facebook, and is also available in the Folly open-source library. We conclude by offering the following questions for future study:

- Can we construct a lattice-based homomorphic hash function which has shorter outputs (significantly less than 2KB) without compromising on security?
- Is it possible to extend LtHash to support multiset inputs for arbitrarily large multiplicities, without increasing the total output size?

Acknowledgments

We thank David Freeman for the technical discussions and helpful edits that went into this work. We thank Soner Terek and Ali Zaveri in helping to formalize the problem and reviewing our candidate constructions using homomorphic hashing.

References

[ABSS93] Sanjeev Arora, László Babai, Jacques Stern, and Z. Sweedyk. The hardness of approximate optima in lattices, codes, and systems of linear equations. In *34th Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA, 3-5 November 1993*, pages 724–733, 1993.

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 99–108, 1996.

[Ajt98] Miklós Ajtai. The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 10–19, 1998.

[ANWOW16] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2X. `https://blake2.net/blake2x.pdf`, 2016.

[BCJ11] Anja Becker, Jean-Sébastien Coron, and Antoine Joux. Improved generic algorithms for hard knapsacks. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, pages 364–385, 2011.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, pages 216–233, 1994.

[BGG95] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography and application to virus protection. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 45–56, 1995.

[BGLS19] Shi Bai, Steven D. Galbraith, Liangze Li, and Daniel Sheffield. Improved combinatorial algorithms for the inhomogeneous short integer solution problem. *J. Cryptology*, 32(1):35–83, 2019.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, pages 159–176, 2012.

[BM97] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, pages 163–192, 1997.

[Bro08] Daniel R. L. Brown. The encrypted elliptic curve hash. *IACR Cryptology ePrint Archive*, 2008:12, 2008.

[CDvD+03] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *Advances in Cryptology - ASIACRYPT 2003, 9th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, November 30 - December 4, 2003, Proceedings*, pages 188–207, 2003.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186, 1999.

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, pages 1–20, 2011.

[CP91] Paul Camion and Jacques Patarin. The knapsack hash function proposed at crypto'89 can be broken. In *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings*, pages 39–53, 1991.

[CP94] Chris Charnes and Josef Pieprzyk. Attacking the SL_2 hashing scheme. In *Advances in Cryptology - ASIACRYPT '94, 4th International Conference on the Theory and Applications of Cryptology, Wollongong, Australia, November 28 - December 1, 1994, Proceedings*, pages 322–330, 1994.

[Dej] Alex Dejanovski. Should you use incremental repair? `http://thelastpickle.com/blog/2017/12/14/should-you-use-incremental-repair.html`. Accessed: 2019-01-15.

[GGH96] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Collision-free hashing from lattice problems. *Electronic Colloquium on Computational Complexity (ECCC)*, 3(42), 1996.

[GR06] Christos Gkantsidis and Pablo Rodriguez. Cooperative security for network coding file distribution. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 23-29 April 2006, Barcelona, Catalunya, Spain*, 2006.

[GSC01] Bok-Min Goi, M. U. Siddiqi, and Hean-Teik Chuah. Incremental hash function based on pair chaining & modular arithmetic combining. In *Progress in Cryptology - INDOCRYPT 2001, Second International Conference on Cryptology in India, Chennai, India, December 16-20, 2001, Proceedings*, pages 50–61, 2001.

[HJ10] Nick Howgrave-Graham and Antoine Joux. New generic algorithms for hard knapsacks. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, pages 235–256, 2010.

[Kar15] Koray Karabina. Point decomposition problem in binary elliptic curves. In *Information Security and Cryptology - ICISC 2015 - 18th International Conference, Seoul, South Korea, November 25-27, 2015, Revised Selected Papers*, pages 155–168, 2015.

[KFM04] Maxwell N. Krohn, Michael J. Freedman, and David Mazières. On-the-fly verification of rateless erasure codes for efficient content distribution. In *2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA*, pages 226–240, 2004.

[Kho04] Subhash Khot. Hardness of approximating the shortest vector problem in lattices. In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 126–135, 2004.

[KY15] Michiel Kosters and Sze Ling Yeo. Notes on summation polynomials. *arXiv e-prints*, page arXiv:1503.08001, March 2015.

[Lau14] Ben Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.

[LLK13] Ben Laurie, Adam Langley, and Emilia Käsper. Certificate transparency. *RFC*, 6962:1–27, 2013.

[LMPR08] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFT: A modest proposal for FFT hashing. In *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, pages 54–72, 2008.

[Lyu05] Vadim Lyubashevsky. On random high density subset sums. *Electronic Colloquium on Computational Complexity (ECCC)*, (007), 2005.

[Mer87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, pages 369–378, 1987.

[MGS15] Hristina Mihajloska, Danilo Gligoroski, and Simona Samardjiska. Reviving the idea of incremental cryptography for the zettabyte era use case: Incremental hash functions based on SHA-3. In *Open Problems in Network Security - IFIP WG*, pages 97–111, 2015.

[Mic98a] Daniele Micciancio. *On the hardness of the shortest vector problem.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1998.

[Mic98b] Daniele Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 92–98, 1998.

[Mic11] Daniele Micciancio. Lattice-based cryptography. In *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 713–715. 2011.

[MP13] Daniele Micciancio and Chris Peikert. Hardness of SIS and LWE with small parameters. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 21–39, 2013.

[MS09] Lorenz Minder and Alistair Sinclair. The extended k-tree algorithm. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 586–595, 2009.

[MT16] Ciaran Mullan and Boaz Tsaban. Sl_2 homomorphic hash functions: worst case to average case reduction and short collision search. *Des. Codes Cryptography*, 81(1):83–107, 2016.

[MTA17] Jeremy Maitin-Shepard, Mehdi Tibouchi, and Diego F. Aranha. Elliptic curve multiset hash. *Comput. J.*, 60(4):476–490, 2017.

[Nat15] National Institute of Standards and Technology. *FIPS 202: SHA-3 standard: Permutation-based hash and extendable-output functions*. 2015.

[PQTZ09] Christophe Petit, Jean-Jacques Quisquater, Jean-Pierre Tillich, and Gilles Zémor. Hard and easy components of collision search in the zémor-tillich hash function: New attacks and reduced variants with equivalent security. In *Topics in Cryptology - CT-RSA 2009, The Cryptographers' Track at the RSA Conference 2009, San Francisco, CA, USA, April 20-24, 2009. Proceedings*, pages 182–194, 2009.

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994.

[Sem15] Igor A. Semaev. New algorithm for the discrete logarithm problem on elliptic curves. *IACR Cryptology ePrint Archive*, 2015:310, 2015.

[Sha08] Andrew Shallue. An improved multi-set algorithm for the dense subset sum problem. In *Algorithmic Number Theory, 8th International Symposium, ANTS-VIII, Banff, Canada, May 17-22, 2008, Proceedings*, pages 416–429, 2008.

[SS79] Richard Schroeppel and Adi Shamir. A T s^2 = o(2^n) time/space tradeoff for certain np-complete problems. In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*, pages 328–336, 1979.

[SvdW06] Andrew Shallue and Christiaan E. van de Woestijne. Construction of rational points on elliptic curves over finite fields. In *Algorithmic Number Theory, 7th International Symposium, ANTS-VII, Berlin, Germany, July 23-28, 2006, Proceedings*, pages 510–524, 2006.

[SY98] Liuba Shrira and Ben Yoder. Trust but check: Mutable objects in untrusted cooperative caches. In *Advances in Persistent Object Systems, Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3), Tiburon, California, USA, 1998*, pages 29–36, 1998.

[TZ94] Jean-Pierre Tillich and Gilles Zémor. Hashing with SL_2. In *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, pages 40–49, 1994.

[van81] Peter van Emde Boas. Another NP-complete problem and the complexity of computing short vectors in a lattice. Technical Report 81-04, athematische Instituut, Universiry of Amsterdam, 1981.

[Wag02] David A. Wagner. A generalized birthday problem. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, pages 288–303, 2002.

[Zav18] Ali Haider Zaveri. Location-Aware Distribution: Configuring servers at scale. `https://code.fb.com/data-infrastructure/location-aware-distribution-configuring-servers-at-scale/`, 2018.

A Proof of Collision Resistance of LtHash

In this section, we first review some background behind lattice-based assumptions in cryptography, and then we provide a proof of Theorem 2.3.

Lattice-based cryptography. Lattices are a well-studied source of hardness conjectures with useful cryptographic applications. Some of the earliest applications of lattices in cryptography were in the design of one-way, collision resistant hash functions [Ajt96, GGH96]. Both the Closest Vector Problem [van81, ABSS93] and the Shortest Vector Problem [Ajt98, Mic98a, Mic98b, Kho04] are NP-hard to approximate. The SIS problem was introduced by Ajtai [Ajt96] and shown to be hard in the average case if the SVP is hard in the worst case.

Matrix kernel problem. Note that in Section 6.1 of [BM97], the matrix kernel problem with respect to parameters k, n, s is defined as equivalent to our formulation of the short integer solutions problem, with

$$(k, n, s)\text{-}\mathsf{matrix\text{-}kernel} = \mathsf{SIS}(k, n, \log_2(q), 1).$$

Ajtai [Ajt96] showed that the an efficient algorithm to solve random instances of $\mathsf{SIS}(n, m, q, B)$ can be used to approximate the shortest vector of a lattice $\mathbf{A}_q^{n\times m}$ to within a factor of $B\sqrt{n}$ in the worst case. [MP13] showed an amplification result with the modulus q, yielding that SIS can still retain its hardness for much smaller moduli. In particular, so long as $q \geq \beta \cdot n^\delta$ for any constant $\delta > 0$, where β is defined as the Euclidean norm of the integer solution $\mathbf{x}$. In fact, q can be composite, so long as its prime factorization consists exclusively of small primes (for our application we use q as a power of 2).[9]

Proof of collision resistance. The remainder of this section covers the proof of Theorem 2.3.

Proof. For an adversary $\mathcal{A}$, we consider the experiment $\mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}$ for $\mathsf{LtHash}_{n,d}$, modeling its underlying hash function after a random oracle. In order to prove the collision resistance property within the random oracle model from SIS, we describe an efficient simulator Sim which acts as a challenger in $\mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}$ and an adversary in $\mathsf{Expt}_{\mathsf{SIS}}$. In this simulation, Sim will respond to random oracle queries made by $\mathcal{A}$, while keeping track of its responses with a key-value store $\mathcal{M}$ containing mappings (initially empty) from inputs to random oracle outputs.

[9] [MP13] also state that $q \leq \beta$ is trivial to solve—however this is not known to be true with the extra restriction that the ℓ_∞-norm must be at most 1, since this requirement rules out the trivial solution.

Description of simulator Sim**.** To begin, $\mathcal{A}$ submits an integer Q to Sim which represents the maximum number of random oracle queries it will make. Then, Sim begins the experiment $\mathsf{Expt}_{\mathsf{SIS}}$ with parameters $(n, Q, 2^d, 1)$, so that the $\mathcal{C}$ samples $\mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n \times Q}$ and sends $\mathbf{A}$ to Sim. The simulator Sim maintains an index for $\mathbf{A}$ by its columns $\mathbf{a}_1, \ldots, \mathbf{a}_Q$. When $\mathcal{A}$ submits the i^{th} unique random oracle query $y_i \in \{0,1\}^*$, Sim responds with $\mathbf{a}_i$ and adds the mapping $(y_i \mapsto \mathbf{a}_i)$ to its key-value store $\mathcal{M}$. Eventually, $\mathcal{A}$ outputs two distinct sets $S = \{s_1, \ldots, s_J\}$ and $T = \{t_1, \ldots, t_K\}$ of sizes $J, K > 0$, respectively. Next, the simulator Sim checks if any elements of $S \cup T$ have no matching key entries in $\mathcal{M}$. If this is the case, then Sim immediately aborts the experiment by outputting the all-ones vector $\mathbf{x} = \mathbf{1}^Q \in \mathbb{Z}^Q$ to $\mathcal{C}$. Otherwise, Sim constructs the vector $\mathbf{x} \in \mathbb{Z}^Q$ as follows, for each $i \in [Q]$:

$$\mathbf{x}_i = \begin{cases} 1, & \text{if } \exists y \in S \setminus T \text{ such that } (y \mapsto \mathbf{a}_i) \text{ is stored in } \mathcal{M} \\ -1, & \text{if } \exists y \in T \setminus S \text{ such that } (y \mapsto \mathbf{a}_i) \text{ is stored in } \mathcal{M} \\ 0, & \text{otherwise} \end{cases}$$

This vector $\mathbf{x}$ is then sent to the challenger $\mathcal{C}$, concluding the simulation.

We first show that Sim is indeed a faithful simulation of the challenger in collision resistance experiment and also a faithful simulation of the adversary in the SIS experiment.

Lemma A.1. *The simulation* Sim *is correct.*

Proof. We first show that Sim is correct as a challenger for $\mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}$. Note that the random oracle responses that Sim sends to $\mathcal{A}$ match the columns of the matrix $\mathbf{A}$ obtained from the challenger of $\mathsf{Expt}_{\mathsf{SIS}}$, which by definition is selected uniformly at random. Hence, the distribution of random oracle query responses made by Sim identically match that of a uniformly random output distribution to unique inputs.

To see that the simulation Sim is correct as an adversary for $\mathsf{Expt}_{\mathsf{SIS}}$, we show that the output vector $\mathbf{x}$ is non-zero (since by definition it clearly satisfies $\|\mathbf{x}\|_\infty \leq 1$). When Sim aborts the experiment, it outputs the all-ones vector, which is indeed non-zero. If Sim did not abort the experiment, then this means that there is a mapping in $\mathcal{M}$ associated with every element in $S \cup T$, and that $S \neq T$. The first condition implies that the entry $\mathbf{x}_i$ is 0 only if there exists a $y \in S \mathbf{c} T$ with $(y \mapsto \mathbf{a}_i)$ in $\mathcal{M}$. However, we know that this cannot be true for every $i \in [Q]$, for otherwise $S = T$, contradicting the second condition. This means that the vector $\mathbf{x}$ is always non-zero, which concludes the proof of correctness for the simulation. □

Lemma A.2. *For all efficient adversaries* $\mathcal{A}$ *that make* Q *oracle queries, we have that*

$$\mathsf{Adv}_{\mathsf{cr}}^{\mathsf{RO}}(\mathsf{LtHash}_{n,d}, \mathcal{A}) \leq \mathsf{Adv}_{\mathsf{SIS}}^{(n,Q,2^d,1)}(\mathsf{Sim}) + \frac{1}{2^{nd}}.$$

Proof. We first examine the abort condition for Sim. The simulator aborts if $\mathcal{A}$ outputs two distinct sets S and T where at least one element of $S \cup T$ has not been submitted as an input to a random oracle query. Hence, the output is independent and uniformly random from the input, and any fixed target sum that would be required to satisfy $\mathsf{LtHash}_{n,d}(S) = \mathsf{LtHash}_{n,d}(T)$. Since the output space consists of nd bits, the probability of the simulator aborting when $\mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}(\mathcal{A}) = 1$ is exactly $1/2^{nd}$.

Now, assuming that the abort event does not happen, we note that each element in $S \cup T$ corresponds to an entry stored in $\mathcal{M}$. If $\mathsf{Expt}_{\mathsf{cr}}^{\mathsf{RO}}(\mathcal{A}) = 1$, then $\mathsf{LtHash}_{n,d}(S) = \mathsf{LtHash}_{n,d}(T)$, which

means that

$$\sum_{i=1}^{J} \mathcal{M}(s_i) - \sum_{j=1}^{K} \mathcal{M}(t_j) = \mathbf{0} \pmod{2^d}$$

Note that this is simply a rewriting of the equality $\mathbf{Ax} = \mathbf{0} \pmod{2^d}$, with $\mathbf{x}$ as defined in Sim. Thus, we have shown that $\mathsf{Expt}_{\mathsf{SIS}}^{(n,Q,2^d,1)}(\mathsf{Sim}) = 1$, which concludes the proof. □

Putting Lemmas A.1 and A.2 together, we have shown that breaking the collision resistance of LtHash is at least as hard as solving the SIS problem for matching parameters, concluding the proof of the theorem. □

B Update Propagation Constructions

B.1 Valid States and Update Semantics

In this section, we provide a formal definition of a valid state, and also the semantics of an "update" as used when discussing update propagation schemes.

Definition B.1 (Valid State)**.** Both the distributor and each subscriber hold internal state values denoted by st_i (for subscribers) and st_i^* (for the distributor), for some integer $i \in [0, \mathtt{n}]$. We (recursively) define a state st_i or st_i^* to be *valid* if any of the following hold:

- The state is $\perp$ (as when initialized).
- The state st^* belongs to the output of $\mathsf{Publish}(\mathsf{sk}, \mathbf{u}_i, \mathsf{st}_{i-1}^*)$ for some $i \in [1, \mathtt{n}]$, and st_{i-1}^* is also valid.
- The state st belongs to the output of $\mathsf{ApplyUpdates}(\mathsf{pp}, i, j, \vec{\mathbf{u}}, \mu, \mathsf{st}_{j-1})$, where $i \in [0, j-1]$, and $(\vec{\mathbf{u}}, \mu) \leftarrow \mathsf{GetUpdates}(\mathsf{pp}, i, j, \mathsf{st}')$, with both st' and st_{j-1} also being valid.

Intuitively, we can think of the set of all valid states as those which would arise from a normal execution of the $\mathsf{Publish}$, $\mathsf{GetUpdates}$, and $\mathsf{ApplyUpdates}$ algorithms between the distributor and the subscribers. In addition to the validity of states, we also define the validity of updates.

Definition B.2 (Valid Update)**.** We can think of a database $\mathcal{D}$ as consisting of a set of indexed rows, where the index is an integer $i \in \mathbb{Z}$ and a row's contents is an arbitrary bitstring in $\{0,1\}^*$. A *delta* to a database can be either an addition or a removal of a row in the database. The two types of deltas are denoted as:

- **Row addition.** For an index $i \in \mathbb{Z}$ and bitstring $x \in \{0,1\}^*$, we write $(i, \perp \to x)$ to represent adding the value x to the i^{th} row of the database.
- **Row deletion.** For an index $i \in \mathbb{Z}$ and bitstring $x \in \{0,1\}^*$, we write $(i, x \to \perp)$ to represent the deletion of the i^{th} row from the database.

A delta is *valid* for a database $\mathcal{D}$ if the delta does not add a row which already exists in $\mathcal{D}$, and does not remove a row which does not already exist in $\mathcal{D}$. An update $\mathbf{u}$ is an ordered sequence of deltas, and the update is *valid* for a database $\mathcal{D}$ if each delta is valid upon being applied to $\mathcal{D}$.[10]

[10] Note that while we do not define a row mutation delta, one can be modeled as an update consisting of a valid row removal followed by a valid row addition.

Note that we can express a database $\mathcal{D}$ as a single update, consisting of the deltas which are just the row additions of each row of the database.

B.2 Secure Signatures

We review the definition of a secure signature scheme. A signature scheme consists of three algorithms $\Pi_{\mathsf{Sig}} = (\mathsf{Sig.Setup}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$, where $\mathsf{Sig.Setup}(1^\lambda) \to (\mathsf{vk}, \mathsf{sk})$ outputs a verification key vk and a signing key sk, $\mathsf{Sig.Sign}(\mathsf{sk}, m) \to \sigma$ takes as input the signing key sk and input message m to produce a signature σ, and $\mathsf{Sig.Verify}(\mathsf{vk}, m, \sigma) \to b \in \{0, 1\}$ takes as input the verification key vk, a message m, a signature σ, and outputs a bit. The signature scheme Π_{Sig} is correct if for $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Sig.Setup}(1^\lambda)$, for every message m, $\mathsf{Sig.Verify}(\mathsf{vk}, m, \mathsf{Sig.Sign}(\mathsf{sk}, m)) = 1$.

To define security, we define an experiment $\mathsf{Expt}_{\mathsf{Sig}}(1^\lambda, \mathcal{A})$ between an adversary $\mathcal{A}$ and a challenger as follows:

1. The challenger runs $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Sig.Setup}(1^\lambda)$ and returns vk to $\mathcal{A}$.
2. $\mathcal{A}$ can adaptively query a signature oracle managed by the challenger. $\mathcal{A}$ can submit an input message m, and the signature oracle returns $\sigma = \mathsf{Sig.Sign}(\mathsf{sk}, m)$.
3. Eventually, $\mathcal{A}$ outputs a final message-signature pair (m^*, σ^*). The output of the experiment is 1 if $\mathsf{Sig.Verify}(\mathsf{vk}, m^*, \sigma^*) = 1$ and m^* was not previously submitted as input to the signature oracle.

We define the advantage of $\mathcal{A}$ in $\mathsf{Expt}_{\mathsf{Sig}}$ for a scheme Π_{Sig} as

$$\mathsf{Adv}_{\mathsf{Sig}}(\Pi_{\mathsf{Sig}}, \mathcal{A}) \coloneqq \Pr[\mathsf{Expt}_{\mathsf{Sig}}(\mathcal{A}) = 1].$$

We say that Π_{Sig} is *existentially unforgeable under a chosen message attack* if for all efficient adversaries $\mathcal{A}$, $\mathsf{Adv}_{\mathsf{Sig}}(\Pi_{\mathsf{Sig}}, \mathcal{A})$ is negligible in the security parameter λ.

B.3 Signing Each Update Directly

The following construction involves creating a signature over the update contents on each update. This is efficient from the distributor's perspective when publishing updates, but the handling of batch update operations in $\mathsf{ApplyUpdates}$ and calls to $\mathsf{Validate}$ require verifying lists of signatures.

Description of Π_{update}. The distributor maintains an internal state $\mathsf{st}^* = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{sig}}, \mathtt{n}^*)$, corresponding to table $\mathsf{T}_{\mathsf{update}}$ of all updates submitted to the Publish algorithm, a table $\mathsf{T}_{\mathsf{sig}}$ of all signature digests output by it, and a sequence number $\mathtt{n}^* \leftarrow 0$. The subscribers each maintain an internal state $\mathsf{st} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{sig}})$ consisting of the two tables $\mathsf{T}_{\mathsf{update}}$ and $\mathsf{T}_{\mathsf{sig}}$ (analogous to the tables in the distributor's state), with $\mathsf{T}_{\mathsf{update}}$ consisting of every update $\mathbf{u}$ that it has received as input to $\mathsf{ApplyUpdates}$, and $\mathsf{T}_{\mathsf{sig}}$ consisting of every signature corresponding to each update in $\mathsf{T}_{\mathsf{update}}$ from the inputs to $\mathsf{ApplyUpdates}$. We define $\Pi_{\mathsf{update}} = (\mathsf{Setup}, \mathsf{Publish}, \mathsf{GetUpdates}, \mathsf{ApplyUpdates}, \mathsf{Validate})$ as follows:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{sk})$. The setup algorithm sets $(\mathsf{pp}, \mathsf{sk}) = (\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Sig.Setup}(1^\lambda)$.
- $\mathsf{Publish}(\mathsf{sk}, \mathbf{u}, \mathsf{st}^*_{\mathsf{in}} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{sig}}, \mathtt{n}^*)) \to (\sigma, \mathsf{st}^*_{\mathsf{out}})$. The publish algorithm sets $\mathtt{n}^* \leftarrow \mathtt{n}^* + 1$, computes $\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}, (\mathbf{u}, \mathtt{n}^*))$, setting $\mathsf{T}_{\mathsf{update}}[\mathtt{n}^*] \leftarrow \mathbf{u}$, $\mathsf{T}_{\mathsf{sig}}[\mathtt{n}^*] \leftarrow \sigma$ and updating $\mathsf{st}^*_{\mathsf{out}} \leftarrow (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{sig}}, \mathtt{n}^*)$ with these new parameters, outputting $(\sigma, \mathsf{st}^*_{\mathsf{out}})$.

- $\mathsf{GetUpdates}(\mathsf{pp}, \mathsf{v}, \mathsf{w}, \mathsf{st} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{sig}})) \rightarrow (\vec{\mathbf{u}}, \mu)$. The GetUpdates algorithm constructs $\vec{\mathbf{u}} = \langle \mathsf{T}_{\mathsf{update}}[\mathsf{v}+1], \ldots, \mathsf{T}_{\mathsf{update}}[\mathsf{w}] \rangle$, and then sets $\mu = \langle \mathsf{T}_{\mathsf{sig}}[\mathsf{v}+1], \ldots, \mathsf{T}_{\mathsf{sig}}[\mathsf{w}] \rangle$, outputting $(\vec{\mathbf{u}}, \mu)$..
- $\mathsf{ApplyUpdates}(\mathsf{pp}, \mathsf{v}, \mathsf{w}, \vec{\mathbf{u}} = \langle \mathbf{u}_{\mathsf{v}+1}, \ldots, \mathbf{u}_{\mathsf{w}} \rangle, \vec{\mu} = \langle \mu_{\mathsf{v}+1}, \ldots, \mu_{\mathsf{w}} \rangle, \mathsf{st}_{\mathsf{in}} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{sig}})) \rightarrow (b, \mathsf{st}_{\mathsf{out}})$. The ApplyUpdates algorithm checks that $\mathsf{v} < \mathsf{w}$ and $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathbf{u}_i, i), \mu_i) = 1$ for all $i \in [\mathsf{v}+1, \mathsf{w}]$. If so, it sets $b = 1$, and for each $j \in [\mathsf{v}+1, \mathsf{w}]$, adds the entries $(\mathsf{T}_{\mathsf{update}}[j], \mathsf{T}_{\mathsf{sig}}[j]) \leftarrow (\mathbf{u}_j, \mu_j)$ and sets $\mathsf{st}_{\mathsf{out}} \leftarrow (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{sig}})$. Otherwise, it sets $b = 0$ and sets $\mathsf{st}_{\mathsf{out}} \leftarrow \mathsf{st}_{\mathsf{in}}$. In both cases, it outputs $(b, \mathsf{st}_{\mathsf{out}})$.
- $\mathsf{Validate}(\mathsf{pp}, \mathcal{D}, \mathsf{v}, \mathsf{st} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{sig}})) \rightarrow b$. The Validate algorithm first checks that $\mathsf{T}_{\mathsf{update}}[0] + \cdots + \mathsf{T}_{\mathsf{update}}[\mathsf{v}] = \mathcal{D}$, and then checks that $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathsf{T}_{\mathsf{update}}[i], i), \mathsf{T}_{\mathsf{sig}}[i]) = 1$ for all $i \in [0, \mathsf{v}]$. If so, it outputs $b = 1$, and outputs $b = 0$ otherwise.

Correctness. To prove correctness of Π_{update}, we note that the internal states $\mathsf{T}_{\mathsf{update}}$ and $\mathsf{T}_{\mathsf{sig}}$ kept by each subscriber maintain the invariant that every update in $\mathsf{T}_{\mathsf{update}}$ has a corresponding signature over its contents in $\mathsf{T}_{\mathsf{sig}}$. Fix a sequence of updates $\mathbf{u}_1, \ldots, \mathbf{u}_{\mathtt{n}}$, and let $\mathcal{D}_i = \mathbf{u}_1 + \cdots + \mathbf{u}_i$ for each $i \in [1, \mathtt{n}]$. Hence, for $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, with $(\sigma_i, \mathsf{st}_i^*) \leftarrow \mathsf{Publish}(\mathsf{sk}, \mathbf{u}_i, \mathsf{st}_{i-1}^*)$, for each $i \in [1, \mathtt{n}]$, we note that:

1. Using $(\vec{\mathbf{u}}, \mu) \leftarrow \mathsf{GetUpdates}(\mathsf{pp}, i, \mathtt{n}, \mathsf{st}_i)$ we have that
$$\vec{\mathbf{u}} = \langle \mathsf{T}_{\mathsf{update}}[i], \ldots, \mathsf{T}_{\mathsf{update}}[\mathtt{n}] \rangle.$$
By the definition of $\mathsf{Publish}$, we have that $\vec{\mathbf{u}} = \langle \mathbf{u}_i, \ldots, \mathbf{u}_{\mathtt{n}} \rangle$, which by definition are exactly the updates for which $\mathcal{D}_i + \mathtt{Sum}(\vec{\mathbf{u}}) = \mathcal{D}_{\mathtt{n}}$.
2. Note that, for each $j \in [i, \mathtt{n}]$, $\mathsf{T}_{\mathsf{update}}[j]$ and $\mathsf{T}_{\mathsf{sig}}[j]$ are constructed from $\mathsf{Publish}$ so that $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathsf{T}_{\mathsf{update}}[j], j), \mathsf{T}_{\mathsf{sig}}[j]) = 1$, by the correctness of the signature scheme. Thus, the $\mathsf{ApplyUpdates}$ call outputs with $b = 1$ as desired.
3. By definition, $\mathsf{T}_{\mathsf{update}}[1] + \cdots + \mathsf{T}_{\mathsf{update}}[i] = \mathcal{D}_i$, and we established that $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathsf{T}_{\mathsf{update}}[i], i), \mathsf{T}_{\mathsf{sig}}[i]) = 1$ by the correctness of the signature scheme. Therefore, we conclude that $\mathsf{Validate}(\mathsf{pp}, \mathcal{D}_i, i, \mathsf{st}_i) = 1$.

Security. To prove security, we construct a simulator Sim which acts as a challenger in $\mathsf{Expt}^{\mathsf{up}}$ and an adversary in $\mathsf{Expt}_{\mathsf{Sig}}$. For a sequence of $\mathtt{n}$ updates, the simulator Sim first receives the verification key vk from the challenger of $\mathsf{Expt}_{\mathsf{Sig}}$, which it forwards to $\mathcal{A}$ as the public parameters pp. Then, for each type of oracle query that the adversary makes, Sim responds as follows:

- **Publish oracle.** The simulator Sim, on an input update $\mathbf{u}$, first checks if the update $\mathbf{u}$ has been submitted before, returning the same signature response if it is a repeat. Otherwise, it simply forwards $\mathbf{u}$ to the challenger of $\mathsf{Expt}_{\mathsf{Sig}}$ to receive a signature σ. This is returned as the digest σ, and the internal state st^* is updated appropriately. For the i^{th} call to the Publish oracle, the input $\mathbf{u}$ is labeled as $\mathbf{u}_i^*$.
- **ApplyUpdates oracle.** This oracle can be completely simulated by Sim since it does not require access to sk. If Sim outputs $\mathsf{Success}$, then let x be the first index in $[\mathsf{v}+1, \mathsf{w}]$ for which $\mathbf{u}_{\mathsf{x}} \neq \mathbf{u}_{\mathsf{x}}^*$. Sim submits the message-signature pair $(\mathbf{u}_{\mathsf{x}}, \mu_{\mathsf{x}})$ to the challenger for $\mathsf{Expt}_{\mathsf{Sig}}$, ending the experiment.

- **Validate oracle.** Again, this oracle can be simulated by Sim using pp. If Sim outputs success, then let k be the smallest index of a row in which $\mathcal{D}$ and $\mathbf{u}_1^* + \cdots + \mathbf{u}_{\mathtt{n}}^*$ differ. Let $\mathsf{x} \in [1, \mathtt{n}]$ be the largest index which adds the row k. The simulator submits the message-signature pair $(\mathsf{T}_{\mathsf{update}}[\mathsf{x}], \mathsf{T}_{\mathsf{sig}}[\mathsf{x}])$ to the challenger for $\mathsf{Expt}_{\mathsf{Sig}}$, ending the experiment.

Lemma B.3. *For all efficient adversaries $\mathcal{A}$, $\mathsf{Adv}_{\mathsf{up}}(\Pi_{\mathsf{update}}, \mathcal{A}) = \mathsf{Adv}_{\mathsf{Sig}}(\Pi_{\mathsf{Sig}}, \mathsf{Sim})$.*

Proof. Note that the only signature oracle queries that Sim makes to its challenger are on the (unique) input message updates $\mathbf{u}_1^*, \ldots, \mathbf{u}_{\mathtt{n}}^*$ that $\mathcal{A}$ submits to the Publish oracle. There are two cases in which the experiment outputs $\mathsf{Success}$:

- **ApplyUpdates oracle.** In the event of $\mathsf{Success}$ from ApplyUpdates, Sim submits the message-signature pair $(\mathbf{u}_{\mathsf{x}}, \mu_{\mathsf{x}})$ to the challenger, and we have by definition that $\mathbf{u}_{\mathsf{x}}$ is distinct from previous signature oracle queries sent to the challenger. Also, since Sim is outputting $\mathsf{Success}$, this means that $\mathsf{Sig.Verify}(\mathsf{vk}, (\mathbf{u}_{\mathsf{x}}, \mathsf{x}), \mu_{\mathsf{x}}) = 1$ again by definition.
- **Validate oracle.** In the event of $\mathsf{Success}$ from Validate, Sim submits the message-signature pair $(\mathbf{u}_{\mathsf{x}}, \mathsf{T}_{\mathsf{sig}}[\mathsf{x}])$ to the challenger. The index x was chosen carefully to ensure that $\mathsf{T}_{\mathsf{update}}[\mathsf{x}]$ is distinct from previous signature oracle queries sent to the challenger, based on the fact that the existing updates could not have introduced the row k which is mismatched in $\mathcal{D}$. Since Sim is outputting $\mathsf{Success}$, this already means that $\mathsf{Sig.Verify}(\mathsf{vk}, (\mathsf{T}_{\mathsf{update}}[\mathsf{x}], \mathsf{x}), \mathsf{T}_{\mathsf{sig}}[\mathsf{x}]) = 1$ again by definition.

In both cases, the pair submitted by Sim to the challenger matches the criteria for which $\mathsf{Expt}_{\mathsf{Sig}}(\mathsf{Sim})$ outputs 1, which concludes the proof. □

B.4 Signing the Database

The next construction creates a signature over the database contents instead of the update contents. This is less efficient from the distributor's perspective when publishing updates, but the handling of batch update operations in $\mathsf{ApplyUpdates}$ and calls to $\mathsf{Validate}$ is now much simpler, involving only a single signature operation in each procedure.

Description of Π_{db}. The distributor maintains an internal state $\mathsf{st}^* = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{db}}, \mathsf{T}_{\mathsf{sig}}, \mathtt{n}^*)$, corresponding to: a table $\mathsf{T}_{\mathsf{update}}$ of each update, a table T_{db}, where each index i will consist of an update constructed as $\mathtt{Sum}(\mathsf{T}_{\mathsf{update}}[1], \ldots, \mathsf{T}_{\mathsf{update}}[i])$, a table $\mathsf{T}_{\mathsf{sig}}$, where each index i will be the signature of $\mathsf{T}_{\mathsf{db}}[i]$, and a sequence number $\mathtt{n}^* \leftarrow 0$. The subscribers each maintain an internal state $\mathsf{st} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{db}}, \mathsf{T}_{\mathsf{sig}})$ consisting of a list of updates $\mathsf{T}_{\mathsf{update}}$, a list of sums of updates T_{db}, along with a list of signatures $\mathsf{T}_{\mathsf{sig}}$ (defined similar to st^*). We define $\Pi_{\mathsf{db}} = (\mathsf{Setup}, \mathsf{Publish}, \mathsf{GetUpdates}, \mathsf{ApplyUpdates}, \mathsf{Validate})$ as follows:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{sk})$. The setup algorithm sets $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{Sig.Setup}(1^\lambda)$.
- $\mathsf{Publish}(\mathsf{sk}, \mathbf{u}, \mathsf{st}_{\mathsf{in}}^* = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{db}}, \mathsf{T}_{\mathsf{sig}}, \mathtt{n}^*)) \to (\sigma, \mathsf{st}_{\mathsf{out}}^*)$. The publish algorithm sets $\mathtt{n}^* \leftarrow \mathtt{n}^* + 1$, $s = \mathsf{T}_{\mathsf{db}}[\mathtt{n}^* - 1] + \mathbf{u}$, $\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}, (s, \mathtt{n}^*))$, sets $\mathsf{T}_{\mathsf{db}}[\mathtt{n}^*] \leftarrow s$, $\mathsf{T}_{\mathsf{sig}}[\mathtt{n}^*] \leftarrow \sigma$, updates $\mathsf{st}_{\mathsf{out}} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{db}}, \mathsf{T}_{\mathsf{sig}}, \mathtt{n}^*)$, and outputs $(\sigma, \mathsf{st}_{\mathsf{out}}^*)$.
- $\mathsf{GetUpdates}(\mathsf{pp}, \mathsf{v}, \mathsf{w}, \mathsf{st} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{db}}, \mathsf{T}_{\mathsf{sig}})) \to (\vec{\mathbf{u}}, \mu)$. The GetUpdates algorithm outputs $(\vec{\mathbf{u}}, \mu) \leftarrow (\langle \mathsf{T}_{\mathsf{update}}[\mathsf{v}+1], \ldots, \mathsf{T}_{\mathsf{update}}[\mathsf{w}] \rangle, \mathsf{T}_{\mathsf{sig}}[\mathsf{w}])$.

- $\mathsf{ApplyUpdates}(\mathsf{pp}, \mathsf{v}, \mathsf{w}, \vec{\mathbf{u}} = \langle \mathbf{u}_{\mathsf{v}+1}, \ldots, \mathbf{u}_{\mathsf{w}} \rangle, \mu, \mathsf{st}_{\mathsf{in}} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{db}}, \mathsf{T}_{\mathsf{sig}})) \rightarrow (b, \mathsf{st}_{\mathsf{out}})$. The ApplyUpdates algorithm checks that $\mathsf{v} < \mathsf{w}$ and $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathsf{T}_{\mathsf{db}}[\mathsf{v}] + \mathtt{Sum}(\vec{\mathbf{u}}), \mathsf{w}), \mu) = 1$. If so, it sets $b = 1$, updates $\mathsf{T}_{\mathsf{update}}[j] \leftarrow \mathbf{u}_j$ for each $j \in [\mathsf{v}+1, \mathsf{w}]$, and $(\mathsf{T}_{\mathsf{db}}[\mathsf{w}], \mathsf{T}_{\mathsf{sig}}[\mathsf{w}]) \leftarrow (\mathsf{T}_{\mathsf{db}}[\mathsf{v}] + \mathtt{Sum}(\vec{\mathbf{u}}), \mu)$, and updates $\mathsf{st}_{\mathsf{out}} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{db}}, \mathsf{T}_{\mathsf{sig}})$. Otherwise, it sets $b = 0$ (without changing $\mathsf{st}_{\mathsf{out}}$). In both cases, it outputs $(b, \mathsf{st}_{\mathsf{out}})$.
- $\mathsf{Validate}(\mathsf{pp}, \mathcal{D}, \mathsf{v}, \mathsf{st} = (\mathsf{T}_{\mathsf{update}}, \mathsf{T}_{\mathsf{db}}, \mathsf{T}_{\mathsf{sig}})) \rightarrow b$. The Validate algorithm checks that $\mathsf{T}_{\mathsf{db}}[\mathsf{v}] = \mathcal{D}$, and then checks that $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathsf{T}_{\mathsf{db}}[\mathsf{v}], \mathsf{v}), \mathsf{T}_{\mathsf{sig}}[\mathsf{v}]) = 1$. If so, it outputs $b = 1$, and outputs $b = 0$ otherwise.

Correctness. To prove correctness of Π_{db}, we note that the internal state tables $\mathsf{T}_{\mathsf{update}}$, T_{db} and $\mathsf{T}_{\mathsf{sig}}$ kept by each subscriber maintain the invariant that every entry in T_{db} has a corresponding signature over its contents in $\mathsf{T}_{\mathsf{sig}}$, and for each $i \in [1, \mathtt{n}]$, $\mathsf{T}_{\mathsf{db}}[i] = \mathsf{T}_{\mathsf{update}}[1] + \cdots + \mathsf{T}_{\mathsf{update}}[i]$. Fix a sequence of updates $\mathbf{u}_1, \ldots, \mathbf{u}_{\mathtt{n}}$, and let $\mathcal{D}_i = \mathbf{u}_1 + \cdots + \mathbf{u}_i$ for each $i \in [1, \mathtt{n}]$. Hence, for $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, with $(\sigma_i, \mathsf{st}_i^*) \leftarrow \mathsf{Publish}(\mathsf{sk}, \mathbf{u}_i, \mathsf{st}_{i-1}^*)$, for each $i \in [1, \mathtt{n}]$, we note that:

1. Using $(\vec{\mathbf{u}}, \mu) \leftarrow \mathsf{GetUpdates}(\mathsf{pp}, i, \mathtt{n}, \mathsf{st}_i)$ we have that

   $$\vec{\mathbf{u}} = \langle \mathsf{T}_{\mathsf{update}}[i], \ldots, \mathsf{T}_{\mathsf{update}}[\mathtt{n}] \rangle,$$

   which by the definition of $\mathsf{Publish}$ means that $\vec{\mathbf{u}} = \langle \mathbf{u}_i, \ldots, \mathbf{u}_{\mathtt{n}} \rangle$, which by definition are exactly the updates for which $\mathcal{D}_i + \mathtt{Sum}(\vec{\mathbf{u}}) = \mathcal{D}_{\mathtt{n}}$.
2. We have that $\mathsf{T}_{\mathsf{db}}[\mathsf{v}] + \mathbf{u}_{\mathsf{v}+1} + \cdots + \mathbf{u}_{\mathtt{n}} = \mathsf{T}_{\mathsf{db}}[\mathtt{n}]$ from the definition of the $\mathsf{Publish}$ algorithm. Therefore, ApplyUpdates runs $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathsf{T}_{\mathsf{db}}[\mathtt{n}], \mathtt{n}), \mathsf{T}_{\mathsf{sig}}[\mathtt{n}]) = 1$, by the correctness of the signature scheme. Thus, the $\mathsf{ApplyUpdates}$ call outputs with $b = 1$ as desired.
3. By definition, $\mathsf{T}_{\mathsf{db}}[i] = \mathbf{u}_1 + \cdots + \mathbf{u}_i = \mathcal{D}_i$, and we established that $\mathsf{Sig.Verify}(\mathsf{pp}, (\mathsf{T}_{\mathsf{db}}[i], i), \mathsf{T}_{\mathsf{sig}}[i]) = 1$ by the correctness of the signature scheme. Therefore, we conclude that $\mathsf{Validate}(\mathsf{pp}, \mathcal{D}_i, i, \mathsf{st}_i) = 1$.

Security. To prove security, we construct a simulator Sim which acts as a challenger in $\mathsf{Expt}^{\mathsf{up}}$ and an adversary in $\mathsf{Expt}_{\mathsf{Sig}}$. For a sequence of $\mathtt{n}$ updates, the simulator Sim first receives the verification key vk from the challenger of $\mathsf{Expt}_{\mathsf{Sig}}$, which it forwards to $\mathcal{A}$ as the public parameters pp. Sim also initialized an empty database $\mathcal{D}^* = \perp$. Then, for each type of oracle query that the adversary makes, Sim responds as follows:

- **Publish oracle.** The simulator Sim, on an input update $\mathbf{u}$, first checks if the update $\mathbf{u}$ has been submitted before, returning the same signature response if it is a repeat. Otherwise, it simply forwards $\mathsf{T}_{\mathsf{db}}[\mathtt{n}-1] + \mathbf{u}$ to the challenger of $\mathsf{Expt}_{\mathsf{Sig}}$ to receive a signature σ. This is returned as the digest σ, and the internal state st^* is updated appropriately. For the i^{th} call to the Publish oracle, the input $\mathbf{u}$ is labeled as $\mathbf{u}_i^*$, and we define $\mathcal{D}_i^* = \mathcal{D}_{i-1}^* + \mathbf{u}$.
- **ApplyUpdates oracle.** This oracle can be completely simulated by Sim since it does not require access to sk. If Sim outputs $\mathsf{Success}$, then Sim sets $\mathbf{m} = (\mathsf{T}_{\mathsf{db}}[\mathsf{v}] + \mathtt{Sum}(\vec{\mathbf{u}}), \mathsf{w})$ and submits the message-signature pair $(\mathbf{m}, \mu)$ to the challenger for $\mathsf{Expt}_{\mathsf{Sig}}$, ending the experiment.

- **Validate oracle.** Again, this oracle can be simulated by Sim using pp. If Sim outputs success, then the simulator submits the message-signature pair $(\mathcal{D}, \mathsf{T}_{\mathsf{sig}}[\mathtt{n}])$ to the challenger for $\mathsf{Expt}_{\mathsf{Sig}}$, ending the experiment.

Lemma B.4. *For all efficient adversaries* $\mathcal{A}$, $\mathsf{Adv}_{\mathsf{up}}(\Pi_{\mathsf{db}}, \mathcal{A}) = \mathsf{Adv}_{\mathsf{Sig}}(\Pi_{\mathsf{Sig}}, \mathsf{Sim})$.

Proof. Note that the only signature oracle queries that Sim makes to its challenger are on the databases $\mathcal{D}_1^*, \ldots, \mathcal{D}_{\mathtt{n}}^*$ that the Publish oracle computes. There are two cases in which the experiment outputs $\mathsf{Success}$:

- **ApplyUpdates oracle.** In the event of $\mathsf{Success}$ from ApplyUpdates, Sim submits the message-signature pair $(\mathbf{m}, \mu)$ to the challenger. It must be the case that $\mathbf{m} = \mathcal{D}_{\mathsf{v}}^* + \mathtt{Sum}(\mathbf{u}) \neq \mathcal{D}_{\mathsf{v}}^* + \mathbf{u}_{\mathsf{v}+1}^* + \cdots + \mathbf{u}_{\mathsf{w}}^* = \mathcal{D}_{\mathsf{w}}^*$, for every $\mathsf{v}, \mathsf{w} \in [1, \mathtt{n}]$. Therefore, we can conclude that $\mathbf{m}$ is distinct from all previous signature oracle queries sent to the challenger. Also, since Sim is outputting $\mathsf{Success}$, this means that $\mathsf{Sig.Verify}(\mathsf{vk}, \mathbf{m}, \mu) = 1$ again by definition.

- **Validate oracle.** In the event of $\mathsf{Success}$ from Validate, Sim submits the message-signature pair $((\mathcal{D}, \mathtt{n}), \mathsf{T}_{\mathsf{sig}}[\mathtt{n}])$ to the challenger. Since $\mathcal{D} \neq \mathcal{D}_{\mathtt{n}}^*$ is distinct from previous signature oracle queries sent to the challenger, and since Sim is outputting $\mathsf{Success}$, this already means that $\mathsf{Sig.Verify}(\mathsf{vk}, (\mathcal{D}, \mathtt{n}), \mathsf{T}_{\mathsf{sig}}[\mathtt{n}]) = 1$ again by definition.

In both cases, the pair submitted by Sim to the challenger matches the criteria for which $\mathsf{Expt}_{\mathsf{Sig}}(\mathsf{Sim})$ outputs 1, which concludes the proof. □