Streaming Authenticated Data Structures: Abstraction and Implementation

Yi Qian
Computer Science Dept.
University of Maryland
yiqian@umd.edu

Yupeng Zhang
ECE Dept. & UMIACS
University of Maryland
zhangyp@umd.edu

Xi Chen
ECE Dept.
University of Maryland
xchen128@umd.edu

Charalampos Papamanthou
ECE Dept. & UMIACS
University of Maryland
cpap@umd.edu

ABSTRACT

In the setting of streaming verifiable computation, a verifier and a prover observe a stream of n elements $x_1, x_2, \ldots, x_n$ and later, the verifier can delegate a computation (e.g., a range search query) to the untrusted prover over the stream. The prover returns the result of the computation and a cryptographic proof for its correctness. To verify the prover's result efficiently, the verifier keeps small local (logarithmic) state, which he updates while observing the stream. The challenge is to enable the verifier to update his local state with no interaction with the prover, while ensuring the prover can compute proofs efficiently.

Papamanthou et al. (EUROCRYPT 2013) introduced *streaming authenticated data structures* (SADS) to address the above problem. Yet their scheme is complex to describe and impractical to implement, mainly due to the use of Ajtai's lattice-based hash function. In this work we present an *abstract* SADS construction that can use any hash function satisfying properties that we formally define. This leads to a *simpler* exposition of the fundamental ideas of Papamanthou et al.'s work and to a *practical* implementation of a streaming authenticated data structure that employs the efficient SWIFFT hash function, which we show to comply with our abstraction. We implement both the EUROCRYPT 2013 construction and our new scheme and report major savings in prover time and public key size.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

streaming authenticated data structures; abstract SADS; GCK hash function

CCSW'14, November 7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2664168.2664177 .

1. INTRODUCTION

We consider verifiable computation in a *streaming* setting, where a client (verifier) and a server (prover) observe a rapidly evolving data stream $x_1, x_2, \ldots, x_n$, which can be stored in its entirety only by the server. The client has *limited local space* (e.g., logarithmic) and can keep only a small and *easily updatable* summary of the data stream. In such a setting, we wish to construct protocols that enable verification of computation on the data stream (e.g., range search queries) via *efficiently computable* proofs provided by the server.

Many prior verifiable computation schemes are unsuitable in the streaming setting: For example, some schemes [7, 15] require the client have access to all the data in the stream ahead of time in order to perform some preprocessing, leading to linear client space; Or they allow the client update his summary through an interactive protocol between the client and the server (e.g., [14]). Due to the nature of the streaming setting, however, interactions can be very expensive. For example, consider a network traffic accounting application [6], where an ISP charges a customer based on the type and duration of its network flows. To enforce that the ISP is performing the accounting correctly, the ISP logs a customer's network flows such that the customer can later make queries to the logs to perform auditing (typically the customer does not have sufficient local storage to log all the flows). In such high link-speed settings, performing an interactive protocol with every packet or flow sent is very expensive.

Related Work. Existing streaming verifiable protocols [3, 4, 5] are efficient in terms of verifier complexity. However, their prover complexity is linear even for sublinear computations such as membership and range search queries on a stream of elements drawn from an ordered universe.

Schröder and Schröder [16] were among the first to provide a verifiable data streaming protocol with logarithmic prover complexity. Their construction assumes a sequential stream and therefore cannot efficiently support non-membership and range search queries. However, as opposed to this work, their scheme achieves the unique property of maintaining the same public key while the stream elements are observed.

Papamanthou, Shi, Tamassia and Yi [13] (referred to as PSTY paper in the following) proposed an efficient streaming verifiable scheme with logarithmic proof size that supports verifiable membership, successor, frequency and range query. In this scheme, the verifier and the prover make updates *independently* of each other and updates are *non-interactive*. For any range query (i.e., return

the elements in the range $[x, y]$), the prover computes a proof in $O(\log M \log n)$ time, where n is an upper bound on the size of the stream and M is the size of the universe.

Despite its prover efficiency and query expressiveness, the PSTY scheme [13] has two problems. First, the protocol is rather complicated, and does not provide a good intuition of the high level scheme design. Second, the scheme is not practical since it uses Ajtai's collision resistant hash function based on the *small integer problem* [11], which has a very large public key size.

Our contributions. Our contributions are as follows:

1. We propose an abstract construction of a streaming authenticated data structure that can be instantiated with a class of hash functions, whose properties we formally define. Our abstract construction has the same computational efficiency, as well as query expressiveness of the PSTY work, which we show to comply with our proposed abstraction.

2. We introduce a class of hash functions based on the *generalized compact knapsack problem* (GCK) that fits our abstract scheme. The GCK hash function is proven to be collision resistant based on the worst case complexity assumption on cyclic lattices by Lyubasevsky and Micciancio [8]. We study the efficiency and cryptanalysis of our new hash function in detail. We also show that our new hash function achieves the desired level of security against state-of-the-art attacks.

3. We implement our streaming verifiable scheme using both hash functions (one from PSTY and the new one) and we compare the two implementations. The implementation using the new hash function achieves 7-8 times faster verification and 2-3 times faster updating with significantly smaller key size. On a stream of length 2^{32} in a universe of 2^{32} unique elements, it takes 0.27 seconds for verifying membership and 0.98 seconds for each update on a desktop machine with 16GB RAM.

The paper below is organized as follows. In Section 2, we review the notations and basic definitions of the streaming authenticated data structures (SADS) scheme, as well as the primitive, the generalized hash trees, that PSTY [13] introduces. In Section 3, we present an abstract SADS construction which is efficient in terms of update, query and verification. The PSTY scheme [13] is shown to be a special instantiation of our abstract SADS. In section 4, we carefully parameterize a class of collision-resistant GCK hash functions that fits our abstract SADS. Section 5 studies the cryptanalysis of our GCK hash function against the *generalized birthday attacks* [17] and the *lattice attacks* [12]. Section 6 proposes a modified construction that reduces the space complexity by a factor of $\sqrt{n}$. Section 7 provides the experimental results based on the implementations of both the hash function in PSTY [13] and our new GCK hash function. Section 8 concludes this paper.

2. PRELIMINARIES

In this section, we give the definition of a streaming authenticated data structure (SADS), as introduced in the PSTY paper [13]. We denote with λ the security parameter and with $n = \mathsf{poly}(\lambda)$ an upper bound on the size of the stream. PPT stands for *probabilistic polynomial-time* and $\mathsf{neg}(\lambda)$ is a negligible function, i.e., a function less than $1/\mathsf{poly}(\lambda)$, for all polynomials $\mathsf{poly}(\lambda)$. We define $[n] = \{0, 1, \ldots, n\}$.

DEFINITION 1 (SADS SCHEME). *Let D be* any *data structure that supports queries q and updates* upd. *An SADS scheme* $\mathbf{A}$ *is a collection of the following six PPT algorithms:*

1. $\mathsf{pk} \leftarrow \mathsf{genkey}(1^\lambda, n)$*: On input the security parameter λ and an upper bound n on the size of the stream, it outputs a public key* pk*;*

2. $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{initialize}(D_0, \mathsf{pk})$*: On input an empty data structure D_0 and the public key* pk*, it computes the authenticated data structure* $\mathsf{auth}(D_0)$ *and the respective state d_0 of it;*

3. $d_{h+1} \leftarrow \mathsf{updateVerifier}(\mathsf{upd}, d_h, \mathsf{pk})$*: On input an update* upd *to data structure D_h, the current state d_h and the public key* pk*, it outputs the updated state d_{h+1};*

4. $\{D_{h+1}, \mathsf{auth}(D_{h+1})\} \leftarrow \mathsf{updateProver}(\mathsf{upd}, D_h, \mathsf{auth}(D_h), \mathsf{pk})$*: On input an update* upd *to data structure D_h, the authenticated data structure* $\mathsf{auth}(D_h)$ *and the public key* pk*, it outputs the updated data structure D_{h+1} along with the updated authenticated data structure* $\mathsf{auth}(D_{h+1})$*;*

5. $\{\alpha(q), \Pi(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$*: On input a query q on data structure D_h, the authenticated data structure* $\mathsf{auth}(D_h)$ *and the public key* pk*, it returns the answer $\alpha(q)$ to the query, along with a proof $\Pi(q)$;*

6. $\{\mathsf{1}, \mathsf{0}\} \leftarrow \mathsf{verify}(q, \alpha(q), \Pi(q), d_h, \mathsf{pk})$*: On input a query q, an answer $\alpha(q)$, a proof $\Pi(q)$ for query q, a digest d_h and the public key* pk*, it outputs either* $\mathsf{1}$ *(accepts) or* $\mathsf{0}$ *(rejects);*

There is *no secret key in our definition*, supporting in this way *public verifiability*. There are two properties that an SADS scheme should satisfy, namely *correctness* and *security* (as in signature schemes definitions).

DEFINITION 2 (CORRECTNESS). *Let $\mathbf{A}$ be an SADS scheme consisting of the set of algorithms. We say that the SADS scheme $\mathbf{A}$ is* correct *if, for all $\lambda \in \mathbb{N}$, for all* pk *output by algorithm* genkey*, for all D_h, $\mathsf{auth}(D_h)$, d_h output by one invocation of* $\mathsf{initialize}$ *followed by polynomially-many invocations of* $\mathsf{updateVerifier}$ *and* $\mathsf{updateProver}$*, where $h \geq 0$, for all queries q and for all $\Pi(q), \alpha(q)$ output by $\mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$, with all but negligible probability $\mathsf{neg}(\lambda)$, it holds that $\mathsf{1} \leftarrow \mathsf{verify}(q, \Pi(q), \alpha(q), d_h, \mathsf{pk})$.*

Apart from the 6 algorithms in Definition 1, we also define the algorithm $\{\mathsf{0}, \mathsf{1}\} \leftarrow \mathsf{check}(q, \alpha, D_h)$ such that it outputs $\mathsf{1}$ if and only if α is the correct answer to query q on data structure D_h (otherwise it outputs $\mathsf{0}$).

DEFINITION 3 (SECURITY). *Let $\mathbf{A}$ be an SADS scheme, λ be the security parameter, D_0 be the empty data structure and $\mathsf{pk} \leftarrow \mathsf{genkey}(1^\lambda)$. Let also* Adv *be a PPT adversary and let d_0 be the state output by* $\mathsf{initialize}(D_0, \mathsf{pk})$.

- *(Update) For $i = 0, \ldots, h-1 = \mathsf{poly}(k)$,* Adv *picks the update upd_i to data structure D_i. Let $d_{i+1} \leftarrow \mathsf{updateVerifier}(\mathsf{upd}_i, d_i, \mathsf{pk})$ be the new state corresponding to the updated data structure D_{i+1}.*
- *(Forge)* Adv *outputs a query q, an answer α and a proof Π.*
- *(Check)* Adv *outputs a query q, an answer α and a proof Π.*

We say that the SADS scheme $\mathbf{A}$ is secure *if for all $\lambda \in \mathbb{N}$, for all* pk *output by algorithm* genkey*, and for any PPT adversary* Adv *the following probability is negligible* $\mathsf{neg}(\lambda)$.

$$\Pr\left[\begin{array}{ll} \{q, \Pi, \alpha\} \leftarrow \mathsf{Adv}(1^\lambda, \mathsf{pk}); & \mathsf{1} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk}); \\ & \mathsf{0} \leftarrow \mathsf{check}(q, \alpha, D_h). \end{array}\right].$$

2.1 Generalized Hash Trees

Recall that a Merkle hash tree [10] is a labeled binary tree T where the label $\lambda(w)$ of every node w is the collision resistant hash (e.g., a SHA-2 hash) of the labels $\lambda(u)$ and $\lambda(v)$ and of its children u and v, i.e., $\lambda(w) = h(\lambda(u), \lambda(v))$. When function h is applied recursively on all the nodes of the tree, the label $\lambda(r)$ of the root r has the following property: A PPT adversary cannot find two different data sets at the leaves that produce the same label at the root of a Merkle tree.

However, certain hash functions have different domain and range. E.g., in SWIFFT [9], the input is a binary vector while the output value is in a finite filed. Obviously, we cannot employ such hash functions in a traditional Merkle hash tree. Generalized hash trees, introduced in the PSTY paper, provide a way to overcome this domain-range discrepancy problem.

Let $h : \mathcal{D} \times \mathcal{D} \to \mathcal{R}$ be a collision resistant hash function accepting two inputs that take values from domain $\mathcal{D}$ and outputting a value in a *different* range $\mathcal{R}$. Let u and v be the two children of w with labels $\lambda(u)$, $\lambda(v)$ and $\lambda(w) \in \mathcal{D}$. Instead of applying hash functions on node labels directly, generalized hash trees use a *deterministic* and *easily computable* projection function $\phi : \mathcal{D} \to \mathcal{R}$, such that $\phi(\lambda(w)) = h(\lambda(u), \lambda(v))$.

Clearly, the labeling of a generalized hash tree need *not* to be unique: In the above example, $\lambda(w)$ can be any ϕ-preimage of $h(\lambda(u), \lambda(v))$. However, the collision resistant property of Merkle trees is still true: Any two valid hash trees representing different data sets at the leaves but with the same root label yield a collision to the underlying hash function. We now formally define generalized hash trees.

DEFINITION 4 (FULL BINARY TREE). *A* full binary tree T *is a non-empty tree where every internal node has two children. It is represented with set of binary strings, where ϵ is the empty string representing the root of T and $w0$ and $w1$ are the string representations of the left and right children of a node having string representation w.*

For example, a full binary tree with five nodes is $T = \{\epsilon, 0, 1, 00, 01\}$. Note that full binary trees need not be complete, i.e., not all leaves must lie at the same level.

DEFINITION 5 (LABELED BINARY TREE). *A* labeled binary tree (T, λ) *is a full binary tree T along with labels $\lambda(w)$ for all $w \in T$.*

DEFINITION 6 (GENERALIZED HASH TREE). *Let $h : \mathcal{D} \times \mathcal{D} \to \mathcal{R}$ be the hash function, and $\phi : \mathcal{D} \to \mathcal{R}$ be the projection function. A* generalized hash tree (T, λ, ϕ, h) *is a labeled binary tree (T, λ) such that **(a)** for all $w \in T$, $\lambda(w) \in \mathcal{D}$; **(b)** for all internal nodes $w \in T$, $\phi(\lambda(w)) = h(\lambda(w0), \lambda(w1))$, where $w0$ and $w1$ are the left and right children of w respectively.*

DEFINITION 7 (TREE COLLISION). *A* tree collision *is a pair of two distinct generalized hash trees (T, λ, ϕ, h) and (T, l, ϕ, h) such that $\lambda(\epsilon) = l(\epsilon)$.*

The next main security theorem establishes the collision resistance for generalized hash trees. Please refer to the PSTY work [13] for a detailed proof.

THEOREM 1 (COLLISION RESISTANCE). *Let λ be the security parameter, T be a full binary tree of $\mathsf{poly}(\lambda)$ depth. If h is collision resistant, there is no PPT algorithm that can output a tree collision (T, λ, ϕ, h) and (T, l, ϕ, h), except with probability $\mathsf{neg}(\lambda)$.*

2.2 The generalized hash trees for SADS

For the SADS, we need an extension of the full binary tree from Definition 4 that can store values at its leaves.

DEFINITION 8 (STRUCTURED BINARY TREE). *Let M be a power of two. A* structured binary tree $T_{\mathcal{C}}$ *is a full binary tree T of $\log M$ levels where all the leaves lie at the last level of the tree, storing values $\mathcal{C} = [c_0, c_1, \ldots, c_{M-1}]$, where $c_i \in [n]$.*

In a structured binary tree, each leaf corresponds to an element in the universe of the stream. The sequence of elements (from leftmost to rightmost) form an ordered universe. E.g., let the universe of a structured binary tree with 8 leaves be $\{0, 1, \ldots, 7\}$. The leaves from left-to-right correspond to elements $0, 1, \ldots, 7$. Without loss of generality, we will assume the universe of size M is $\{0, 1, \ldots, M-1\}$. The value stored at each leaf indicates the frequency of the corresponding leaf element. E.g., in a structured binary tree of 8 leaves, c_6 indicates the frequency of element 6.

2.3 Groups and homomorphisms

We now give a brief review on groups and homomorphism that are going to be needed for defining our abstract construction. Please refer to [1] for a comprehensive study.

A *commutative group* is a set $\mathcal{G}$ with an operation $\odot$, such that (1) the operation $\odot$ is associative and commutative; (2) $\mathcal{G}$ has an identity; (3) Every element in $\mathcal{G}$ has an inverse. We use $0_{\mathcal{G}}$ to denote the identity and use $\sum \in \mathcal{G}$ to denote the summation over group $\mathcal{G}$. A subset $\mathcal{H}$ of $\mathcal{G}$ is a *subgroup* of $\mathcal{G}$ if $\mathcal{H}$ forms a group under the same operation $\odot$. A subgroup $\mathcal{N}$ of group $\mathcal{G}$ is a *normal subgroup* if it is invariant under conjugation; that is, $\forall n \in \mathcal{N}$, $\forall g \in \mathcal{G}$, $g \odot n \odot g^{-1} \in \mathcal{N}$.

In this paper, we focus on the two commutative groups with respect to the *domain* and the *range*. Namely, let $\mathcal{D}$ be the domain with an operation $\oplus$, and $\mathcal{R}$ be the range with an operation $\otimes$.

A *homomorphism* $\phi : \mathcal{D} \to \mathcal{R}$ is a map from $\mathcal{D}$ to $\mathcal{R}$ such that for all x, y in $\mathcal{D}$, $\phi(x \oplus y) = \phi(x) \otimes \phi(y)$. The *kernel* of the homomorphism ϕ is the set of elements in $\mathcal{D}$ that are mapped to the identity in $\mathcal{R}$: $ker(\phi) = \{x \in \mathcal{D} \,|\, \phi(x) = 0_{\mathcal{R}}\}$. Notice $ker(\phi)$ is a normal subgroup of $\mathcal{D}$ and always contains $0_{\mathcal{D}}$, i.e., $\phi(0_{\mathcal{D}}) = 0_{\mathcal{R}}$. The *image* of the homomorphism ϕ is $im(\phi) = \{\phi(x) \,|\, x \in \mathcal{D}\}$. The homomorphism ϕ is surjective if and only if $im(\phi) = \mathcal{R}$.

An *isomorphism* is a bijective group homomorphism. Two groups $\mathcal{D}$ and $\mathcal{R}$ are *isomorphic* if there exists an isomorphism from one to the other.

Let H be a subgroup of $\mathcal{D}$ and $a \in \mathcal{D}$. The subset $coset(a, H) = \{a \oplus h \,|\, h \in H\}$ is called a *coset* of H. For a normal subgroup $N \in \mathcal{D}$, the *quotient group* $\mathcal{D}/N$ is defined as the set of cosets of N in $\mathcal{D}$: $\mathcal{D}/N = \{coset(a, N) | a \in \mathcal{D}\}$.

Let $\phi : \mathcal{D} \to \mathcal{R}$ be a surjective group homomorphism. The first isomorphism theorem states that the quotient group $\mathcal{D}/ker(\phi)$ is isomorphic to $\mathcal{R}$ [1].

For example, let $\phi : \mathbb{C} \to \mathbb{R}$ be the mapping from every complex number to its absolute value. It is easy to check ϕ forms a homomorphism under multiplication. The kernel of ϕ is the unit circle U, as it maps to the multiplication identity 1 in $\mathbb{R}$. The quotient group $\mathbb{C}/ker(\phi)$ consists of all multiples of U. In other words, it is the collection of all circles centered at the origin in $\mathbb{C}$, each of which is a coset of U. Clearly, there is a bijection between each circle in $\mathbb{C}$ and its radius in $\mathbb{R}$.

3. ABSTRACT CONSTRUCTION

In this section, we define the hash function, the projection function and the labeling function for our abstract SADS construction. See Definitions 9, 10 and 15 respectively.

The class of hash functions. We characterize the class of hash functions that fit our abstract SADS. Let $\mathcal{D}$ and $\mathcal{R}$ be the domain and the range of interest, both of which are commutative groups.

DEFINITION 9 (HASH FUNCTION). *The class of hash functions of SADS, $h : \mathcal{D} \times \mathcal{D} \to \mathcal{R}$ is characterized as follows:*

1. $h(\mathbf{x}, \mathbf{y}) = \mathcal{H}_{\mathbf{L}}(\mathbf{x}) \otimes \mathcal{H}_{\mathbf{R}}(\mathbf{y})$.
2. *There is no PPT algorithm can find $\mathbf{x}_\delta, \mathbf{y}_\delta \neq \mathbf{0}_\mathcal{D}$ such that $\mathcal{H}_{\mathbf{L}}(\mathbf{x}_\delta) + \mathcal{H}_{\mathbf{R}}(\mathbf{y}_\delta) = \mathbf{0}_\mathcal{R}$ with non-negligible probability.*
3. *For $\mathbf{x}, \mathbf{y} \in \mathcal{D}$, $\mathcal{H}_{\mathbf{A}}(\mathbf{x} \oplus \mathbf{y}) = \mathcal{H}_{\mathbf{A}}(\mathbf{x}) \otimes \mathcal{H}_{\mathbf{A}}(\mathbf{y})$, where $\mathbf{A}$ is either $\mathbf{L}$ or $\mathbf{R}$.*

COROLLARY 1. *The hash function h in Definition 9 is collision resistant.*

PROOF. Assume there is a PPT algorithm A that outputs two distinct pairs of $(\mathbf{x}_1, \mathbf{y}_1)$, $(\mathbf{x}_2, \mathbf{y}_2) \in \mathcal{D}$, such that

$$\mathcal{H}_{\mathbf{L}}(\mathbf{x}_1) \otimes \mathcal{H}_{\mathbf{R}}(\mathbf{y}_1) = \mathcal{H}_{\mathbf{L}}(\mathbf{x}_2) \otimes \mathcal{H}_{\mathbf{R}}(\mathbf{y}_2)$$

with non-negligible probability. Then, the PPT algorithm can find $\mathbf{x}_\delta, \mathbf{y}_\delta \neq \mathbf{0}_\mathcal{D}$ s.t. $\mathcal{H}_{\mathbf{L}}(\mathbf{x}_\delta) \otimes \mathcal{H}_{\mathbf{R}}(\mathbf{y}_\delta) = \mathbf{0}_\mathcal{R}$ with non-negligible probability. Contradiction. □

We construct our collision resistant hash h as the sum of two hash functions over the group $\mathcal{R}$, where both component hash functions form homomorphisms from $\mathcal{D}$ to $\mathcal{R}$. Boneh and Boyen proved that there is no generic construction that combines two arbitrary collision resistant hash functions $\mathcal{H}_1$, $\mathcal{H}_2$ into one collision resistant hash $\mathcal{H}$, such that the output of $\mathcal{H}$ is any shorter than the the concatenation of the outputs of $\mathcal{H}_1$ and $\mathcal{H}_2$ [2]. Hence, our construction of h cannot guarantee collision resistance, and condition 2 in Definition 9 is necessary. In the later sections, we will show h is inherently collision resistant, given that $\mathcal{H}_{\mathbf{L}}$ and $\mathcal{H}_{\mathbf{R}}$ are collision resistant and have certain matrix structures.

The projection function. As we saw before, one important component of the generalized hash tree is the projection function. Specifically, we will need the projection function be homomorphic.

DEFINITION 10 (PROJECTION FUNCTION FOR SADS). *The projection function for SADS is a surjective homomorphism $\phi : \mathcal{D} \to \mathcal{R}$. That is, for all $x, y \in \mathcal{D}$, $\phi(x \oplus y) = \phi(x) \otimes \phi(y)$.*

By the first isomorphism theorem, $\mathcal{D}/ker(\phi)$ and $\mathcal{R}$ form an isomorphism [1]. Denote this canonical isomorphism as:

$$\pi : \mathcal{D}/ker(\phi) \to \mathcal{R}.$$

We define an inverse projection functions as follows.

DEFINITION 11 (INVERSE PROJECTION). *The inverse projection function ψ is a function from $\mathcal{R}$ to $\mathcal{D}$ such that*

1. *for each $y \in \mathcal{R}$, $\psi(y) = x$, where $x \in \pi^{-1}(y)$;*
2. *$\psi(0_\mathcal{R}) = 0_\mathcal{D}$.*

COROLLARY 2. *Let ϕ, ψ be the projection function and the inverse projection function by Definition 10 and Definition 11. For any $y \in \mathcal{R}$, we have $\phi(\psi(y)) = y$.*

In practice, the projection function and the inverse projection function should be efficiently computable.

The labeling function. Let h, ϕ be the hash function and the projection function respectively. We now continue with defining the labels of the generalized hash tree (see Definition 15). Before that, we give some necessary definitions:

DEFINITION 12 (RANGE OF A NODE). *Let w be a node of a structured binary tree $T_\mathcal{C}$. The set* range(w) *contains the leaves of the subtree of $T_\mathcal{C}$ rooted on w.*

DEFINITION 13. *Define the functions $g_0 : \mathcal{D} \to \mathcal{D}$ and $g_1 : \mathcal{D} \to \mathcal{D}$ such that $g_0(\mathbf{x}) = \psi(\mathcal{H}_{\mathbf{L}}(\mathbf{x}))$ and $g_1(\mathbf{x}) = \psi(\mathcal{H}_{\mathbf{R}}(\mathbf{x}))$. Also, for a bitstring $w = b_1 b_2 \dots b_e$, define the function $g_w : \mathcal{D} \to \mathcal{D}$ as the composition $g_w(\mathbf{x}) = g_{b_1} \circ g_{b_2} \circ \dots \circ g_{b_e}(\mathbf{x})$.*

To construct the labeling function, we start with defining a class of γ functions that maps frequency values stored at leaf nodes to labels. Namely, $\gamma : [n] \to \mathcal{D}$ satisfies the following equation

$$\text{Given } c_v + c_\delta \in [n], \gamma(c_v + c_\delta) = \gamma(c_v) \oplus \gamma(c_\delta). \quad (1)$$

DEFINITION 14 (PARTIAL LABELS). *Let $T_\mathcal{C}$ be a structured binary tree. The partial label of a node is defined recursively by the follows.*

1. *The* partial label *of a leaf node v with respect to itself is defined by $\mathcal{L}_v(v) = \gamma(c_v)$, where c_v is the frequency value stored at leaf v and γ is a function by Equation 1.*
2. *For every other node w of $T_\mathcal{C}$, and for every leaf $v \in$* range(w)*, the* partial label *$\mathcal{L}_w(v)$ of w with respect to v is defined as $\mathcal{L}_w(v) = g_{v-w}(\gamma(c_v))$, where $v - w$ is the result of removing prefix w from bitstring v.*

E.g., for a structured binary tree of 8 leaves, the partial label of the root wrt leaves 2 and 3 are $\mathcal{L}_\epsilon(2) = \psi(\mathcal{H}_{\mathcal{L}} \circ \psi(\mathcal{H}_{\mathbf{R}} \circ \psi(\mathcal{H}_{\mathbf{L}} \circ \gamma(c_2))))$ and $\mathcal{L}_\epsilon(3) = \psi(\mathcal{H}_{\mathbf{L}} \circ \psi(\mathcal{H}_{\mathbf{R}} \circ \psi(\mathcal{H}_{\mathbf{R}} \circ \gamma(c_3))))$ respectively.

COROLLARY 3. *Let $T_\mathcal{C}$ be a structured binary tree, with $w \in T_\mathcal{C}$ be any internal node. Given a leaf $v \in T_\mathcal{C}$ with its value $c_v = 0$, the* partial label *of w with respect to v, $\mathcal{L}_w(v) = 0_\mathcal{D}$.*

PROOF. By Definition 14, $\mathcal{L}_w(v) = g_{v-w}(\gamma(0))$. By Equation 1, it is easy to see $\gamma(0) = 0_\mathcal{D}$. By the homomorphism of $\mathcal{H}_{\mathbf{A}}$, we have $\mathcal{H}_{\mathbf{A}}(0_\mathcal{D}) = 0_\mathcal{R}$, where $\mathbf{A}$ is either $\mathbf{L}$ or $\mathbf{R}$. Since ψ is an isomorphism from $\mathcal{R}$ to $\mathcal{D}$, we have $\psi(0_\mathcal{R}) = 0_\mathcal{D}$. Consequently, the composed function $\psi(\mathcal{H}_{\mathbf{A}}(0_\mathcal{D})) = 0_\mathcal{D}$. Since g_{v-w} is a chain of such composed function, $g_{v-w}(\gamma(0)) = g_{v-w}(0_\mathcal{D}) = 0_\mathcal{D}$. □

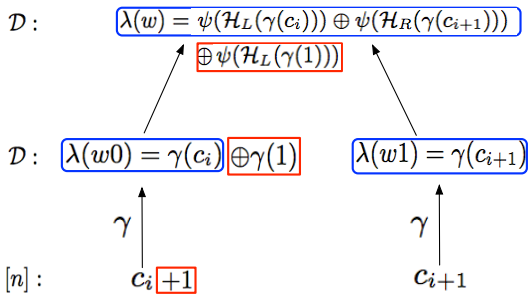

Figure 1: Update of labels. At the leaf nodes $w0$ and $w1$, frequency values c_i and c_{i+1} are stored respectively. The labels of $w0$ and $w1$ are therefore $\gamma(c_i)$ and $\gamma(c_{i+1})$. By Definition 15, the label of their parent node w is $\psi(\mathcal{H}_L(\gamma(c_i))) \oplus \psi(\mathcal{H}_R(\gamma(c_{i+1})))$. Now, assume frequency value stored at leaf $w0$ is updated to $c_i + 1$. The label of $w0$ can simply be updated by an $\oplus$ operation with $\gamma(1)$. Similarly, the label of w is updated by an $\oplus$ operation with $\psi(\mathcal{H}_L(\gamma(1)))$.

DEFINITION 15 (LABELING FUNCTION). *Let $T_{\mathcal{C}}$ be a structured binary tree, where $\mathcal{C} = [c_0, c_1, \ldots, c_{M-1}]$. For every node $w \in T_{\mathcal{C}}$, we define its labeling as $\lambda(w) = \sum_{v \in \mathsf{range}(w)} \mathcal{L}_w(v) \in \mathcal{D}$.*

E.g., Figure 1 illustrates a simple example of the labeling function. At the leaf level of a generalized hash tree, frequency values c_i and c_{i+1} are stored at leaf node $w0$ and $w1$. Hence, the labels of $w0$ and $w1$ are $\lambda(w0) = \gamma(c_i)$ and $\lambda(w1) = \gamma(c_{i+1})$. By Definition 15, the label of their parent node w should be the summation over $\mathcal{D}$ of the partial labels of w with respect to $w0$ and $w1$. Hence, $\lambda(w) = \psi(\mathcal{H}_L(\gamma(c_i))) \oplus \psi(\mathcal{H}_R(\gamma(c_{i+1})))$.

LEMMA 1. *Let $T_{\mathcal{C}}$ be a structured binary tree. Let h, ϕ, λ be the hash function, projection function and labeling function described above. Then, $\phi(\lambda(w)) = h(\lambda(w0), \lambda(w1))$, where w is any internal node of $T_{\mathcal{C}}$ and $w0, w1$ are two children of w.*

PROOF.

$$\phi(\lambda(w)) = \phi\left(\sum_{v \in \mathsf{range}(w)} \mathcal{L}_w(v) \in \mathcal{D}\right) \quad \text{(Def. 15)}$$

$$= \sum_{v \in \mathsf{range}(w0)} \phi\left(\mathcal{L}_w(v)\right) \otimes \sum_{v \in \mathsf{range}(w1)} \phi\left(\mathcal{L}_w(v)\right) \quad \text{(Def. 9)}$$

$$= \sum_{v \in \mathsf{range}(w0)} \phi\left(g_{w-v}(\gamma(c_v))\right) \otimes \sum_{v \in \mathsf{range}(w1)} \phi\left(g_{w-v}(\gamma(c_v))\right) \quad \text{(Def. 14)}$$

$$= \sum_{v \in \mathsf{range}(w0)} \phi\left(g_0(\mathcal{L}_{w0}(v))\right) \otimes \sum_{v \in \mathsf{range}(w1)} \phi\left(g_1(\mathcal{L}_{w1}(v))\right) \quad \text{(Def. 14)}$$

$$= \sum_{v \in \mathsf{range}(w0)} \phi\left(\psi(\mathcal{H}_{\mathbf{L}} \cdot \mathcal{L}_{w0}(v))\right) \otimes \sum_{v \in \mathsf{range}(w1)} \phi\left(\psi(\mathcal{H}_{\mathbf{R}} \cdot \mathcal{L}_{w1}(v))\right) \quad \text{(Def. 13)}$$

$$= \sum_{v \in \mathsf{range}(w0)} \mathcal{H}_{\mathbf{L}} \cdot \mathcal{L}_{w0}(v) \otimes \sum_{v \in \mathsf{range}(w1)} \mathcal{H}_{\mathbf{R}} \cdot \mathcal{L}_{w1}(v) \quad \text{(Cor. 2)}$$

$$= \mathcal{H}_{\mathbf{L}}(\lambda(w0)) \otimes \mathcal{H}_{\mathbf{R}}(\lambda(w1)) = h(\lambda(w0), \lambda(w1)). \quad \text{(Def. 15)}$$

□

THEOREM 2. *Let $T_{\mathcal{C}}$ be a structured binary tree. Then, $(T_{\mathcal{C}}, \lambda, f, h)$ is a generalized hash tree, where h, ϕ, λ are the hash function, projection function and labeling function described above. .*

PROOF. It follows from Lemma 1 and by Definition 6. □

3.1 Efficient updates of the labels

The output $\lambda(v)$ of the labeling function applied to node v can be updated very efficiently whenever the leaf changes. We first introduce the following definition.

DEFINITION 16 (UNIT UPDATE). *For any internal node $w \in T_{\mathcal{C}}$ and any leaf node $v \in T_{\mathcal{C}}$, let the unit update of w in terms of v be $\delta_w(v) = g_{\epsilon - v_i}(\gamma(1))$.*

Each occurrence of an element i contributes $\delta_w(i) = g_{w-v_i}(\gamma(1))$ on the internal node w. Adding (or removing) an element i is equivalent to adding (or subtracting) $\delta_w(i)$ to (from) $\lambda(w)$. It is important to note that computing a unit update, as well as a partial label, only requires $O(\log M)$ recursive calls of hash function h.

Figure 1 shows how to update the corresponding labels when the value stored at leaf node $w0$ increments by 1. For a more concrete example, let $\lambda(\epsilon)$ be the label of the root of a generalized hash tree $(T_{\mathcal{C}}, \lambda, \phi, h)$ with eight leaves $\{v_0, v_1, \ldots, v_7\}$ where $c_3 = 2$, $c_4 = c_6 = c_7 = 1$ and $c_0 = c_1 = c_2 = c_5 = 0$. By Corollary 3, the root label $\lambda(\epsilon) = \sum_{v \in \mathsf{range}(\epsilon)} \mathcal{L}_w(v) \in \mathcal{D}$ can be expressed as $\mathcal{L}_\epsilon(v_2) + \mathcal{L}_\epsilon(v_4) + \mathcal{L}_\epsilon(v_6) + \mathcal{L}_\epsilon(v_7)$. Adding (or removing) an element i is equivalent to adding (or subtracting) $\delta_\epsilon(i)$ to (from) $\lambda(\epsilon)$, which only takes $O(\log M)$ calls of h.

3.2 SADS construction

Let $T_{\mathcal{C}}$ be a structured binary tree with M leaves corresponding to the universe. Let $(T_{\mathcal{C}}, \lambda, \phi, h)$ denote the generalized hash tree of interest as described above. To store the generalized hash tree, we store only the labels that are defined on the paths from non-zero leaves to the root (all other labels are zero). This requires space proportional to $O(\nu \log M)$, where ν is the number of distinct element appearing in the stream. Figure 2 presents the 6 algorithms of our abstract SADS scheme.

Range search queries. The abstract SADS inherits the query expressiveness from the PSTY work [13]. It supports the range search queries by the same algorithm.

The proof for a range search query $[x, y]$ simply contains the two proofs $\Pi(x)$ and $\Pi(y)$ as output by algorithms $\mathsf{query}(x, D_h, \mathsf{auth}(D_h), \mathsf{pk})$ and $\mathsf{query}(y, D_h, \mathsf{auth}(D_h), \mathsf{pk})$ respectively from Figure 2. It also contains the frequencies $\mathcal{C}_{xy} = \{c_{a_1}, c_{a_2}, \ldots, c_{a_s}\}$ of the reported range as an answer. Let now $\mathcal{R}_{xy} = \{a_1, a_2, \ldots, a_s\}$ denote the respective reported range that corresponds to $\mathcal{C}_{xy}$.

For verification, the proofs $\Pi(x)$ and $\Pi(y)$ are verified first by using algorithm verify from Figure 2. If this verification is successful, perform the following test (else reject): If for all labels $\lambda(v) \in \Pi(x) \cup \Pi(y)$ such that $\mathsf{range}(v) \cap \mathcal{R}_{xy}$ is not empty, the following relation (as in Definition 15)

$$\lambda(v) = \sum_{i \in \mathsf{range}(v) \cap \mathcal{R}_{xy}} \mathcal{L}_v(i) \tag{3}$$

is true, output $\mathtt{1}$ (i.e., accept), else output $\mathtt{0}$ (i.e., reject). The above relation ensures that all the range (with the correct frequencies) has been reported, or otherwise, the adversary could find a collision. The above technique can be also used for verifying successor queries, where the reported range is empty.

3.3 PSTY instantiation of our abstract SADS

We show the PSTY scheme [13] is an instantiation of our abstract SADS. Recall our SADS is built upon a generalized hash tree $(T, \lambda, \phi, h_{old})$, where h_{old} is a collision resistant hash function, ϕ is a projection function and λ is a labeling function. In particular, the labeling function λ is determined by the hash function h_{old}, the γ function and the inverse projection function ψ. We next study all four components of the PSTY scheme [13].

Hash function. The hash function h_{old} uses $\mathcal{D} = \mathbb{Z}_n^t$ and $\mathcal{R} = \mathbb{Z}_q^\nu$ as its domain and range. The operation $\oplus$ of $\mathcal{D}$ is mod n addition, and the operation $\otimes$ of $\mathcal{R}$ is mod q addition. Specifically, h_{old} is defined as:

1. $h_{old}(\mathbf{x}, \mathbf{y}) = \mathcal{H}_{\mathbf{L}}(\mathbf{x}) + \mathcal{H}_{\mathbf{R}}(\mathbf{y}) \bmod \mathrm{q}$;
2. $\mathcal{H}_{\mathbf{A}}(\mathbf{x}) = \mathbf{A} \cdot \mathbf{x} \bmod \mathrm{q}$, where $\mathbf{A} = \mathbf{L}, \mathbf{R}$ is a randomly picked matrix from $\mathbb{Z}_q^{\nu \times t}$.

$\mathcal{H}_{\mathbf{A}}$ is collision resistant based on the hardness assumption of the small integer solution problem $\mathsf{SIS}_{q,t,\beta}$ [11]. Here, q is a large prime, $t = \nu\lceil \log q \rceil$, β is set to $n\sqrt{2t}$.

First, we show h_{old} is collision resistant. Assume there exist $\mathbf{x}_\delta, \mathbf{y}_\delta \neq \mathbf{0}_{\mathcal{D}}$ such that $\mathcal{H}_{\mathbf{L}}(\mathbf{x}_\delta) + \mathcal{H}_{\mathbf{R}}(\mathbf{y}_\delta) = \mathbf{0}_{\mathcal{R}}$ with non-negligible probability. This would give a solution to $\mathsf{SIS}_{q,2t,\beta}$ with the matrix be $[L,\ R]$ randomly chosen from $\mathbb{Z}_q^{\nu \times 2t}$.

Second, given $\mathbf{x}, \mathbf{y} \in \mathcal{D}$ such that $\mathbf{x} + \mathbf{y} \in \mathcal{D}$, it is obvious that

$$\mathcal{H}_{\mathbf{A}}(\mathbf{x} \oplus \mathbf{y}) = \mathcal{H}_{\mathbf{A}}(\mathbf{x} + \mathbf{y}) = \mathbf{A} \cdot \mathbf{x} + \mathbf{A} \cdot \mathbf{y} \bmod \mathrm{q}.$$

The labeling function below ensures that for any two labels $\mathbf{x}, \mathbf{y} \in \mathcal{D}$ in the generalized hash tree, $\mathbf{x} + \mathbf{y} \in \mathcal{D}$ holds.

From the above two arguments, h_{old} meets the characterization of Definition 9.

Algorithm $\mathsf{pk} \leftarrow \mathsf{genkey}(1^\lambda, n)$.
On input the security parameter λ and a bound n on the size of the stream, set $\mathsf{pk} = \{\mathbf{L}, \mathbf{R}, \mathcal{U}\}$, where $\mathcal{U}$ is a universe such that $|\mathcal{U}| = M$ and $\mathbf{L}, \mathbf{R}$ specify the hash function.

Algorithm $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{initialize}(D_0, \mathsf{pk})$.
Let D_0 be a structured binary tree $T_\mathcal{C}$ where $c_i = 0$ $(i = 0, \ldots, M-1)$. The algorithm outputs the generalized hash tree $(T_\mathcal{C}, \lambda, \phi, h)$ as $\mathsf{auth}(D_0)$, where $\lambda(v) = \mathbf{0}_\mathcal{D}$ for all nodes v in $T_\mathcal{C}$. Also it outputs $d_0 = \mathbf{0}_\mathcal{D}$.

Algorithm $d_{h+1} \leftarrow \mathsf{updateVerifier}(x, d_h, \mathsf{pk})$.
Let $x \in \mathcal{U}$ be the current element of the stream. The algorithm updates the local state by setting $d_{h+1} = d_h \oplus \delta_\epsilon(x)$, where ϵ is the root of $T_\mathcal{C}$ and $\delta_\epsilon(x)$ is defined in Definition 16.

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1})\} \leftarrow \mathsf{updateProver}(x, D_h, \mathsf{auth}(D_h), \mathsf{pk})$.
Let $x \in \mathcal{U}$ be the current element of the stream. The algorithm sets $c_x = c_x + 1$, outputting the updated tree $T_\mathcal{C}$. Let $v_\ell, \ldots, v_1$ be the path in $T_\mathcal{C}$ from node v_ℓ (v_ℓ stores c_x) to the child v_1 of the root ϵ of $T_\mathcal{C}$. Set

$$\lambda(v_i) = \lambda(v_i) \oplus \delta_{v_i}(x) \text{ for } i = \ell, \ell-1, \ldots, 1\,, \tag{2}$$

where $\delta_{v_i}(x)$ is defined in Definition 16. The new authenticated data structure $\mathsf{auth}(D_{h+1})$ is the new generalized hash tree with the updated labels as computed in Equation 2.

Algorithm $\{\alpha(q), \Pi(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$.
Let q be a frequency query for element $x \in \mathcal{U}$. Set $\alpha(q) = c_x$ (note that if $c_x = 0$, x is not contained in the collection). Let $v_\ell, \ldots, v_1$ be the path in the structured binary tree $T_\mathcal{C}$ from node v_ℓ (v_ℓ stores the value c_x) to the child v_1 of the root ϵ of $T_\mathcal{C}$. Let also $w_\ell, \ldots, w_1$ be the *sibling nodes* of $v_\ell, \ldots, v_1$. Proof $\Pi(q)$ contains the ordered sequence of the pairs of labels belonging to the tree path from leaf v_ℓ to the root ϵ of the tree, i.e., the pairs $\{(\lambda(v_\ell), \lambda(w_\ell)), (\lambda(v_{\ell-1}), \lambda(w_{\ell-1})), \ldots, (\lambda(v_1), \lambda(w_1))\}$.

Algorithm $\{1, 0\} \leftarrow \mathsf{verify}(q, \alpha(q), \Pi(q), d_h, \mathsf{pk})$.
Let q be a frequency query for element $x \in \mathcal{U}$. Parse $\Pi(q)$ as

$$\{(\lambda(v_\ell), \lambda(w_\ell)), \ldots, (\lambda(v_1), \lambda(w_1))\}$$

and $\alpha(q)$ as c_x.
If $\lambda(v_\ell) \neq \gamma(c_x)$ or $\lambda(v_\ell), \lambda(w_\ell) \notin \mathcal{D}$, output 0. Compute values $y_{\ell-1}, y_{\ell-2}, \ldots, y_0$ as $y_i = h(\lambda(v_{i+1}), \lambda(w_{i+1}))$ (if v_{i+1} is v_i's left child) or $y_i = h(\lambda(v_{i+1}), \lambda(w_{i+1}))$ (if v_{i+1} is v_i's right child). For $i = \ell - 1, \ldots, 1$, if $\phi(\lambda(v_i)) \neq y_i$ **or** $\lambda(v_i), \lambda(w_i) \notin \mathcal{D}$ output 0. If $\phi(d_h) \neq y_0$, output 0. Output 1.

Figure 2: Algorithms of the abstract SADS for verifying frequency queries.

Projection function. The projection function $\phi : \mathbb{Z}_n^t \to \mathbb{Z}_q^\nu$ *parses* the input vector $\mathbf{x}$ as a radix-2 representation (i.e., a base-2 representation but not necessarily of binary coefficients) and converts it to the respective vector in $\mathbb{Z}_q^\nu$.

On input a vector $\mathbf{x} \in \mathbb{Z}_n^t$, where $\tau = \lceil \log q \rceil$ and $t = \nu \cdot \tau$, output a vector $\mathbf{y} = \phi(\mathbf{x})$ of ν entries such that each $\mathbf{y}_i$ $(i = 0, \ldots, \nu-1)$ is the number in $\mathbb{Z}_q$ represented by the radix-2 representation $[\mathbf{x}_{i\tau}, \mathbf{x}_{i\tau+1}, \ldots, \mathbf{x}_{(i+1)\tau-1}]^\mathsf{T}$, namely

$$\mathbf{y}_i = \sum_{j=0}^{\tau-1} \mathbf{x}_{i\tau+j} 2^j \mod q\,, \text{ for } i = 0, \ldots, \nu-1\,.$$

It is easy to check that ϕ here forms a surjective homomorphism from $\mathbb{Z}_n^t$ to $\mathbb{Z}_q^\nu$.

Inverse projection function. The inverse projection function $\psi : \mathbb{Z}_q^\nu \to \mathbb{Z}_n^t$ is simply the binary representation of vectors. For example, if $\nu = 2$, $q = 8$ and $\mathbf{a} = [6, 3]^\mathsf{T} \in \mathbb{Z}_8^2$, then $\psi(\mathbf{a}) = [1, 1, 0, 0, 1, 1]^\mathsf{T}$, since $\psi(6) = [1, 1, 0]^\mathsf{T}$ and $\psi(3) = [0, 1, 1]^\mathsf{T}$. Obviously, ψ here satisfies Definition 11.

γ function. The gamma function $\gamma : [n] \to \mathbb{Z}_n^t$ is defined as:

$$\gamma(c_v) = c_v \cdot \mathbf{1}, \tag{4}$$

where $\boldsymbol{1} = [1, 1, \ldots, 1]^\mathsf{T} \in \mathbb{Z}_n^t$. Clearly, $\gamma(c_{v1} + c_{v2}) = \gamma(c_{v1}) + \gamma(c_{v2})$ mod n, given $c_{v1} + c_{v2} \in [n]$.

Let $T_C = (T, \lambda, \phi, h_{old})$ be the generalized hash tree as described above. For any node $w \in T_C$, by Definition 15 we have its label $\lambda(w) = \sum_{v \in \mathsf{range}(w)} \mathcal{L}_w(v) \in \mathcal{D}$. Since the entries of each partial label $\mathcal{L}_w(v) = c_v \cdot \{0, 1\}^t$ and $\sum_{i=0}^{M-1} c_i \leq n$, we know $\lambda(w) \in \mathbb{Z}_n^t$. It follows that for any two labels $\mathbf{x}, \mathbf{y} \in T_C$, $\mathbf{x} + \mathbf{y} \in \mathbb{Z}_n^t$ is also true. Therefore, T_C, as it is described in PSTY [13], is an instance of the abstract SADS.

In the following sections, we show how to build an improved instantiation of the abstract SADS by employing a better hash function. The choices of the *projection function*, the *inverse projection function* and γ *function* **remain the same**.

4. NEW HASH FUNCTION BASED ON GENERALIZED KNAPSACK PROBLEM

In the implementation of our SADS, we found the hash function h_{old} in PSTY [13] computationally costly both in terms of running time and key size. We study both the algebraic and the security properties of the Generalized Compact Knapsack problem (GCK), and take a subclass of GCK hash functions to instantiate our SADS.

Preliminaries. We give a brief preliminary review on rings and ideals. Please refer to [1] for a comprehensive study. Let $\mathbb{Z}[x]$ and $\mathbb{R}[x]$ be the set of polynomials with integers and real coefficients. A polynomial is *monic* if the coefficient of the highest power is 1. A polynomial is *irreducible* if it cannot be represented as a product of lower degree polynomials. Each polynomial corresponds to a vector of its coefficients. E.g., we can represent polynomial $a_0 + \ldots + a_{k-1}x^{k-1}$ simply as $(a_0, \ldots, a_{k-1})$. We define the l_p norm $|g(x)|_p$ of a polynomial $g(x)$ as the norm of the corresponding vector.

Let R be a ring. An *ideal* I of R is an additive subgroup of R closed under multiplication by arbitrary $g \in R$. For any ring $f \in R$, $\langle f \rangle$ denotes the set of all multiples of f. The quotient R/I is the set of all equivalence classes $(g + I)$ of R modulo I.

Let $\mathcal{R} = \mathbb{Z}[x]/\langle f \rangle$ where f is monic and irreducible. When f is monic and of degree k, every equivalence calss $(g + \langle f \rangle) \in \mathbb{Z}[x]/\langle f \rangle$ has a unique representative $g' \in (g + \langle f \rangle)$ of degree less than k. We define the norm $|(g + \langle f \rangle)|_f$ over ring $\mathbb{Z}[x]/\langle f \rangle$ as $|g \mod f|_\infty$. As short hand, we write $|g|_f$ instead of $|(g + \langle f \rangle)|_f$.

In this paper, we focus on the ring $\mathcal{R} = \mathbb{Z}_p[\alpha]/(\alpha^k + 1)$, where $k - 1$ is the highest degree of an arbitrary $g \in \mathcal{R}$ and p is a prime.

4.1 Generalized Compact Knapsack Problem

We begin with a brief review of the Generalized Compact Knapsack problem. Lyubasevsky et al propose an efficient SWIFFT hash that is provably collision-resistant [9]. Finding a collision on the average with any noticeable probability is at least as hard as solving worst case problems for cyclic lattices. The SWIFFT function has a simple algebraic expression over the ring $\mathcal{R} = \mathbb{Z}_p[\alpha]/(\alpha^k + 1)$. A particular function in the family is specified by m fixed $\mathbf{a}_1, \ldots, \mathbf{a}_m \in \mathcal{R}$. The function corresponds to the following expression over the ring $\mathcal{R}$:

$$\Sigma_{i=1}^{m}(\mathbf{a}_i \cdot \mathbf{x}_i) \ \in \mathcal{R},$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathcal{R}$ are polynomials with binary coefficients, and corresponding to the input of length $k \cdot m$.

The above formula is a special instance of generalized compact knapsack problem proposed in [8].

DEFINITION 17. **(GCK problem).** *Given m random elements $\mathbf{a}_1, \ldots, \mathbf{a}_m \in \mathcal{R}$ for some ring, and a target $\mathbf{t} \in \mathcal{R}$, find elements $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathcal{I}$ such that $\Sigma_{i=1}^{m}\mathbf{a}_i \cdot \mathbf{x}_i = \mathbf{t}$, where $\mathcal{I}$ is some given subset of $\mathcal{R}$.*

Lyubashevsky and Micciancio prove that for appropriate choices of $\mathcal{R}$ and $\mathcal{I}$, finding collisions in such hash functions is at least as hard as solving worst case hard problems on ideal lattices [8]. We present the constraints on $\mathcal{R}$ and $\mathcal{I}$ to make $\mathcal{H}$ collision-resistant [8].

1. Ring $\mathcal{R} = \mathbb{Z}[\alpha]_p/\langle f \rangle$, where $f \in \mathbb{Z}[\alpha]$ is irreducible, monic polynomial of degree n with expansion factor $EF(f, 3) \leq \varepsilon$. The definition of expansion factor is as the following.

$$EF(f, q) = \max_{g \in \mathbb{Z}[\alpha], deg(g) \leq q(deg(f)-1)} |g|_f / |g|_\infty$$

2. $\mathcal{I} = \{g \in \mathcal{R} : |g|_f \leq d\}$, where d is some positive integer.

3. It is required that $m > \log p / \log 2d$ and $p > 2\varepsilon dmk^{1.5} \log k$.

4.2 New Hash Function

We focus on the following GCK hash function family $\{\mathcal{H}_\mathbf{A}\} : \mathcal{D} \to \mathcal{R}$, where $\mathcal{D} = \mathcal{I}^m = \mathbb{Z}_n^{k \cdot m}$ and $\mathcal{R} = \mathbb{Z}[\alpha]_p/\langle \alpha^k + 1 \rangle$.

Given input $X = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m] \in \mathcal{D}$,

$$\mathcal{H}_\mathbf{A}(X) = \Sigma_{i=1}^{m}(\mathbf{a_i} \cdot \mathbf{x_i}) \ \in \mathcal{R}, \tag{5}$$

where $A = [\mathbf{a}_1, \ldots, \mathbf{a}_m]$ and each $\mathbf{a}_i \in \mathcal{R}$.

We construct a subclass of collision-resistant GCK hash functions from the above function family by careful parameter selection. First, since $EF(f, 3) \leq 3$ is proved for $f = \alpha^k + 1$ in [8], ε in the constraints above can be set to 3. By picking a large enough prime p such that $p/\log p > 6nk^{1.5} \log k$ and setting $m = \lceil \log p \rceil$, it is easy to check that conditions 1 2 3 for $\mathcal{H}_\mathbf{A}$ to be collision resistant are all met. Second, we make $\mathcal{H}_\mathbf{A}$ achieve the desired level of security against the best known attack algorithms by additional constraints. Figure 3 shows the algorithm of parameter configuration for our GCK hash function. In particular, constraints 1 and 4 make $\mathcal{H}_\mathbf{A}$ theoretically collision resistant; constraints 2 and 3 ensure $\mathcal{H}_\mathbf{A}$ reaches the level of security λ against the best known algorithms of the *generalized birthday attack* [17] and the *lattice attack* [12] respectively. We study the cryptanalysis of $\mathcal{H}_\mathbf{A}$ in section 5.

Algorithm $\{k, p, m\} \leftarrow (n, \lambda)$.
Let k be a power of 2 and p be a prime. Find the smallest k and then the smallest p such that

1. $p/\log(p) > 6nk^{1.5} \log k$;
2. $2^{\lfloor \log m \rfloor} p^{k/(1+\lfloor \log m \rfloor)} \geq 2^\lambda$;
3. $n\sqrt{km} < 2^{2\sqrt{k \log p \log \delta}}$;
4. $m = \lceil \log(p) \rceil$.

Figure 3: Parameter configuration of our GCK hash function $\mathcal{H}_\mathbf{A}$.

We construct a new hash function for our SADS based on $\mathcal{H}_\mathbf{A}$, $h_n : \mathbb{Z}_n^{k \cdot m} \times \mathbb{Z}_n^{k \cdot m} \to \mathbb{Z}[\alpha]_p/\langle \alpha^n + 1 \rangle$, as follows.

$$h_n(\mathbf{x}, \mathbf{y}) = \mathcal{H}_L(\mathbf{x}) + \mathcal{H}_\mathcal{R}(\mathbf{y}) \mod p, \tag{6}$$

where L, R are randomly chosen from $\mathbb{Z}[\alpha]_p^m/\langle \alpha^k + 1 \rangle$. By similar arguments in section 3.3, h_n belongs to the class of hash functions of the abstract SADS by Definition 9.

The bottleneck in terms of efficiency of the GCK hash function $\mathcal{H}_\mathbf{A}$ is computing the product of two polynomials in Equation 5. Similar as SWIFFT [9], we use FFT for computing polynomial products. Notice $\alpha^k + 1 = 0$, for $\alpha = e^{j\pi i/k}$ when j is odd. Hence, we only need to do k evaluations at ω^j, where $\omega = e^{\pi i/k}$ and $j = 1, 3, \ldots, 2k - 1$, to interpolate the product polynomial in $\mathcal{R} = \mathbb{Z}[\alpha]_p/\langle \alpha^k + 1 \rangle$.

Notice every polynomial has a unique vector representation. In our implementation, we use a vector in $[p]^k$ to represent a ring in $\mathcal{R} = \mathbb{Z}[\alpha]_p/\langle \alpha^k + 1 \rangle$. The algorithm of our GCK hash function is given in Figure 4. For each $\mathbf{x}_i$, we run twice FFT, which requires $O(k \log k)$ operations with small constant. Hence, the total running time for our algorithm is $O(km \log k)$ with a small constant.

Algorithm $\mathbf{z} \leftarrow (X, A)$.

1. Let input $X = [\mathbf{x}_1, \ldots, \mathbf{x}_m] \in [n]^{k \times m}$. Let ringe $\mathcal{R} = \mathbb{Z}[\alpha]_p / \langle \alpha^k + 1 \rangle$. $A = [\mathbf{a}_1, \ldots, \mathbf{a}_m]$, where each $\mathbf{a}_i \in \mathcal{R}$.
2. Precompute and store $\text{FFT}(\mathbf{a}_i, \omega)$ for each $\mathbf{a}_i$, where $\omega = e^{\pi i/k}$. Let $\hat{\mathbf{a}}_i = \text{FFT}(\mathbf{a}_i, \omega)$.
3. For $i = 1, \ldots, m$, do step (4) (5).
4. For each $\mathbf{x}_i$, compute $\hat{\mathbf{x}}_i = \text{FFT}(\mathbf{x}_i, \omega)$. Compute $\mathbf{y}_i = \hat{\mathbf{x}}_i \cdot \hat{\mathbf{a}}_i$, where $\cdot$ refers to the inner product of two vectors. Compute $\hat{\mathbf{y}}_i = \frac{1}{k}\text{FFT}(\mathbf{y}_i, w^{-1})$.
5. update $\mathbf{z} = \mathbf{z} + \hat{\mathbf{y}}_i \bmod p$.

Figure 4: Algorithms of the our GCK hash function $\mathcal{H}_{\mathbf{A}}$.

5. SECURITY ANALYSIS

We have shown that the GCK hash function chosen in this paper is collision-resistant. In this section, we study its cryptanalysis, and determine the concrete levels of security against the best known attacks. Specifically, we consider the *generalized birthday attacks* [17] and the *lattice attacks* [12].

As it is shown in [9], the product of two polynomials $\mathbf{a}, \mathbf{x} \in \mathcal{R}$ is equivalent to the matrix product of the skew-circulant matrix of $\mathbf{a}$ with $\mathbf{x}$ in the field $\mathbb{Z}_p[\alpha]$. Thus, equation 5 can be represented as the matrix product of $\mathcal{A} \in \mathbb{Z}_p[\alpha]^{k \times k \cdot m}$ with $\mathbf{x} \in [n]^{k \cdot m}$, where $\mathcal{A}$ consists of m skew-circulant matrices corresponding to each $\mathbf{a}_i \in \mathcal{R}$, as shown below by Equation 7. This formulation is the sum of a subset of the $k \cdot m$ columns of $\mathcal{A}$ over the field $\mathbb{Z}_p[\alpha]$. Relaxing the dependencies within each skew-circulant matrix, the best known algorithm for finding collision is the same one for solving the subset sum problem [17].

$$\Sigma_{i=1}^{m}(\mathbf{a}_i \cdot \mathbf{x}_i) \in R = \begin{bmatrix} \mathcal{A}_1 & \mathcal{A}_2 & \ldots & \mathcal{A}_m \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_{k \cdot m} \end{bmatrix} \quad (7)$$

5.1 Generalized Birthday Attacks

Finding a collision in our GCK hash function is equivalent to finding a nonzero $\mathbf{x} \in \{-n, \ldots, n\}^{k \cdot m}$ such that $\mathcal{A} \cdot \mathbf{x} = \mathbf{0} \bmod p$, where A is the $k \times k \cdot m$ matrix shown above. We provide the best known attack algorithm based on Wagner's work [17].

1. Divide the columns of $\mathcal{A}$ into m groups, each of which consists of k columns.
2. Create a list of $(2n+1)^k$ vectors where each vector is a different $\{-n, \ldots, n\}$ combination of k columns within the group.
3. Finding one vector from each list such that their sum is $\mathbf{0}$ in $\mathbb{Z}_p[\alpha]$ is the k-list problem studied by Wagner [17][1]. Wagner's algorithm has $O(2^{\lfloor \log m \rfloor} p^{k/(1+\lfloor \log m \rfloor)})$ time and space complexity. By constraint 2 in Algorithm 3, we achieve the required level of security against the best known generalized birthday attack algorithms.

[1]k-list problem consider the condition when vectors are random and independent. Relaxing the dependencies is a conservative assumption. Hence, the lower bound analysis still holds

We briefly illustrate how Wagner's algorithm works here. The first observation is solving m-sum problem is computational equivalent to solving a $m' < m$-sum problem. Pick one vector per list from lists $m'+1, ..., m$, and denote their sum as $\mathbf{c}$. Applying the m'-sum algorithm to find the solution for sum$= -\mathbf{c} \bmod p$ over the first m' lists will solve the corresponding m-sum problem. In our case, the m-sum problem is reduced to a k'-sum problem, where $k' = 2^{\lfloor \log m \rfloor}$ is the largest power of two less than m. The algorithm follows a recursive fashion on a complete binary tree of $\lfloor \log m \rfloor$ depth. It first extends each list to size $O(p^{k/(1+\lfloor \log m \rfloor)})$. Then, it constructs new lists with $h \cdot k/(1 + \lfloor \log m \rfloor)$ more zero entries at internal layer of height h. Finally, at the root we will find the vectors that gives us $\mathbf{0}-$sum solution. Notice that this algorithm easily applies to sum=$\mathbf{c}$ problem, where $\mathbf{c}$ is an arbitrary vector.

5.2 Lattice Attacks

According to the analysis in [12], collisions in our GCK hash functions are vectors in the $k \cdot m$-dimensional lattice with coordinates in $\{-n, \ldots, n\}$. The Euclidean length of such vectors can at most be $n\sqrt{k \cdot m}$. However, the state of art lattice reduction algorithm cannot find non-trivial lattice vectors of which the Euclidean length is less than $2^{2\sqrt{k \log p \log \delta}}$, where $\delta = 1.01$. By constraint 3 in Algorithm 3, the best known lattice attack algorithm cannot find collision in our GCK hash functions efficiently.

6. OPTIMIZATION

In this section, we introduce a modified labeling function for the PSTY scheme [13] that reduces the space complexity by a factor of $\sqrt{n}$, except with negligible probability. Since this labeling function does not rely on any properties of the hash function, the modification also applies to any instantiations with the same projection function, inverse projection function and the γ function introduced in section 3.3.

DEFINITION 18 (MODIFIED LABELING). *Let $T_{\mathcal{C}}$ be a structured binary tree, where $\mathcal{C} = [c_0, c_1, \ldots, c_{M-1}]$. For every node $w \in T_{\mathcal{C}}$ we define its label $\lambda(w) = \sum_{v \in \text{range}(w)} c_v \cdot \delta_w(v) \bullet \text{rand}()$, where $\delta_w(v)$ is the unit update defined by Definition 16,* rand() *outputs a random bit from $\{-1, 1\}$ and $\bullet$ is the scalar product.*

The random function rand() can be generated and retrieved by a pseudorandom generator with a public key. Given the γ function as Equation 4, it is obvious that $\mathcal{L}_w(v) = c_v \cdot \delta_w(v)$. Next, we prove the size of the labels will be reduced to $O(\sqrt{n})$ by this optimization.

LEMMA 2. *Assume elements in $\mathcal{C} = [c_0, c_1, \ldots, c_{M-1}]$ are independent and identically distributed (i.i.d.), with mean μ and variance σ^2. For a node w and a leaf $v \in \text{range}(w)$, Let $\mathcal{P}_w(v) = c_v \cdot \delta_w(v) \bullet \text{rand}() = [p_1, p_2, \ldots, p_{k \cdot t}]$. Let $\delta_w(v) = [\delta_1, \delta_2, \ldots, \delta_{k \cdot t}]$. For $1 \le j \le k \cdot t$, we have*

$$\begin{cases} p_j = 0 & \text{if } \delta_j = 0; \\ p_j \text{ is } i.i.d. \text{ with mean 0 and variance } \mu^2 + \sigma^2 & \text{if } \delta_j = 1. \end{cases}$$

PROOF. For $1 \le j \le k \cdot t$, δ_j is a deterministic binary value by Definition 16. In the first case, $p_j = 0$ is trivially true when $\delta_j = 0$. We consider the second case when $\delta_j = 1$. Since rand() returns $1(-1)$ with $1/2$ probability independently and all elements in $\mathcal{C}$ are independent and identically distributed, we have the probability mass function of p_j as the follows:

$$f_{p_j}(x) = \begin{cases} \frac{1}{2} f_C(x) & x > 0 \\ \frac{1}{2} f_C(-x) & x < 0 \end{cases},$$

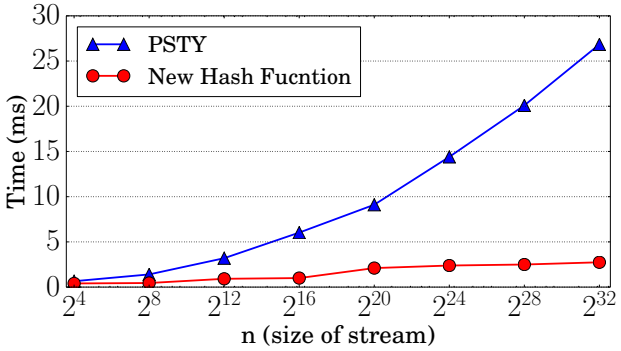

Figure 5: Running time of the hash functions. In the x axis, we present the size n of the stream in log scale. The new hash function h_{new} is 1.6× faster when $n = 2^4$, and 9.8× faster when $n = 2^{32}$, than h_{old}.

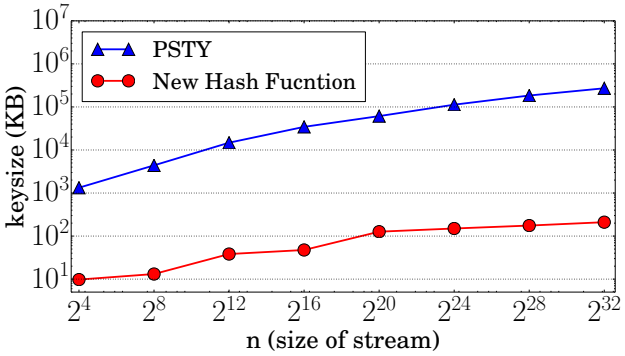

Figure 6: Key size of the hash functions. In the x axis, we present the size n of the stream and in the y axis we show the key size in KB. The figure is in log − log scale. The key size of h_{new} is only 0.73% when $n = 2^4$, and 0.077% when $n = 2^{32}$, of h_{old}.

where f_C is the probability mass function of elements in C. Since the distribution of C has mean μ and variance σ^2, we have $\mu_{p_j} = 0$ and $\sigma^2_{p_j} = \mu^2 + \sigma^2$. □

LEMMA 3. *Let the label of the root $\lambda(r) = [\lambda_1, \lambda_2, \ldots, \lambda_{k\cdot t}]$. For $1 \le j \le k \cdot t$, let n_j be the total number of leaves with nonzero p_j defined in Lemma 2. Then, $\frac{\lambda_j}{\sqrt{n_j}}$ follows Gaussian distribution $N(0, \mu^2 + \sigma^2)$ as $n_j \to \infty$.*

PROOF. Following directly from the Definition 18 and Lemma 2, λ_j is the summation of n_j i.i.d random variables. By the **Central Limit Theorem**, $\frac{\lambda_j}{\sqrt{n_j}} \sim N(0, \mu^2 + \sigma^2)$ as $n_j \to \infty$.

□

LEMMA 4. *Let $\mathbf{X}$ be a Gaussian distribution with mean 0 and variance σ^2, then $\Pr[\mathbf{X} > t] < \frac{1}{\sqrt{2\pi}\sigma t} e^{-\frac{t^2}{2\sigma^2}}$.*

PROOF. By definition, $\Pr[\mathbf{X} > t] = \int_t^\infty \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} dx$. We can bound the above expression by the following:

$$\int_t^\infty \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} dx < \frac{1}{\sqrt{2\pi}\sigma t} \int_t^\infty x e^{-\frac{x^2}{2\sigma^2}} dx = \frac{1}{\sqrt{2\pi}\sigma t} e^{-\frac{t^2}{2\sigma^2}}$$

. □

THEOREM 3. *Let w be any node in a generalized hash tree. Every element λ_j of its modified labeling $\lambda(w)$ by Definition 18 is in $[-t\sqrt{n}, t\sqrt{n}]$ for some constant t, except with negligible probability* neg(t).

PROOF. By Lemma 3, $\frac{\lambda_j}{\sqrt{n_j}} \sim N(0, \mu^2 + \sigma^2)$. Therefore,

$$\begin{aligned}
&\Pr[|\lambda_j| > t\sqrt{n}] < \Pr[|\lambda_j| > t\sqrt{n_j}] \quad \text{(Lemma 3)} \\
&= \Pr[|\frac{\lambda_j}{\sqrt{n_j}}| > t] \\
&= 2\Pr[\frac{\lambda_j}{\sqrt{n_j}} > t] \\
&< 2 \times \frac{1}{\sqrt{2\pi(\mu^2+\sigma^2)}t} e^{-\frac{t^2}{2(\mu^2+\sigma^2)}} \quad \text{(Lemma 4)},
\end{aligned}$$

which is negligible neg(t).

□

7. EXPERIMENTS AND COMPARISONS

We implement the instantiations of our abstract SADS with both the hash function h_{old} in PSTY [13] and our new GCK based hash function h_{new} introduced in Section 4.2. We conduct a series of comparing experiments of these two instantiations. We show major performance improvements due to the GCK hash function.

First, we highlight the asymptotic improvement of h_{new} compared to h_{old} in terms of the running time and key size. Next, we discuss the experimental results on various algorithms of Figure 2 for both instantiations. In particular, we present the empirical performance of *verification*, *update* and *range search query* by the two instantiations.

Experiment setup. The two instantiations are implemented in Mathlab 2014Ra and executed on a Windows 8.1 Desktop with 16GB of RAM. The same data samples randomly generated are used in all experiments. We collected 10 runs for each data point and report the average.

There are two critical parameters involved in the experiments:

1. The size of the stream n. This determines the parameter configuration of our GCK hash function h_{new} by Algorithm 3. In particular, the running time and the key size of both h_{new} and h_{old} increase with respect to n.

2. The size of the universe M. $|M|$ determines the size of the generalized hash tree (the generalized hash tree has $\log M$ layers), and consequently affects the running time of the query, update and verification of the SADS.

Table 1: Asymptotic comparison between h_{old} and h_{new}. Notice $t = \nu \log q$.

	running time	key size
PSTY	$O(\nu^2 \log q)$	$O(\nu^2 \log q)$
New Hash Function	$O(k \log k \log p)$	$O(k \log p)$

Asymptotic comparison. Table 1 shows the asymptotic complexity comparison of running time of hash functions and key size between h_{old} and h_{new}[2]. As it is shown in both Section 3.3 and PSTY [13], the hash function h_{old} is a matrix–vector multiplication, of which the running time is $O(\nu^2 \log q)$ and the key size is

[2]PSTY [13] has a set of different notations for parameters.

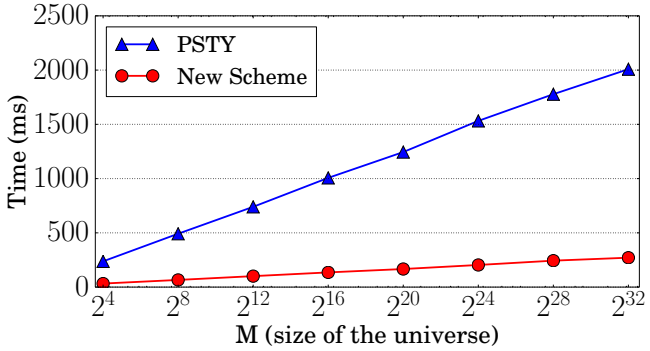

Figure 7: Verification time. The x axis is the size of universe $|M|$ in log scale. The size of stream is fixed to $n = 2^{32}$. The new scheme with h_{new} achieves $7.5\times$ speed up.

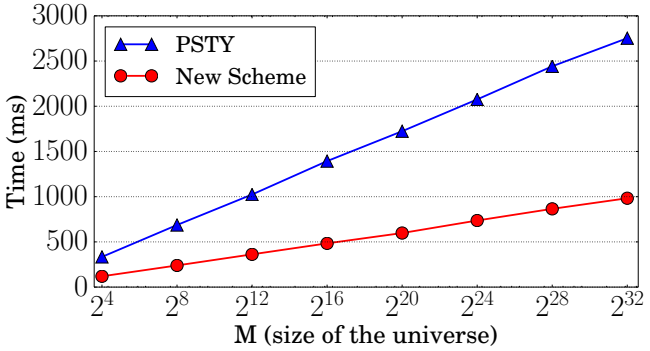

Figure 8: Update time (client side). The x axis is the size of universe $|M|$ in log scale. The size of stream is fixed to $n = 2^{32}$. The new scheme with h_{new} achieves $2.8\times$ speed up.

the size of the matrix, $O(\nu^2 \log q)$. The hash function h_{new} constructed in Section 4 is primarily based on polynomial multiplication, of which the running time using FFT is $O(k \log k \log p)$ and the key size is $O(k \log p)$.

We generate parameters (e.g. k, p and q) according to the size of the stream n to ensure approximately the same security parameter (100^+ in both cases). In practice, k is much smaller than ν and $\log p$ is roughly equal to $\log q$. Consequently, the improvement is rather significant.

Running time of the hash functions. Figure 5 shows the running time of hashing one message by both h_{old} and h_{new} with increasing n. Our new hash function h_{new} turns to outperform h_{old} by orders of magnitude. Specifically, h_{new} runs $1.6\times$ faster when $n = 2^4$, and $9.8\times$ faster when $n = 2^{32}$, than h_{old}. This matches the asymptotic complexity comparison mentioned above. More importantly, the time cost of h_{new} doesn't grow much as n increases while it grows quasi-linearly with n for h_{old}. As a result, the new hash function h_{new} supports much larger streaming volume than h_{old}.

Key size of hash functions. Figure 6 shows the key sizes of both h_{old} and h_{new} with increasing n, which is plotted in log-log scale. The key size of our new hash h_{new} is only 0.73% of the one of h_{old} when $n = 2^4$, and 0.077% of the ones of h_{old} when $n = 2^{32}$. This significant improvement is due to the advantages of circular lattices over regular lattices. Moreover, the key size of h_{new} grows roughly linearly with n while it grows quadratically with n for h_{old}. Notice the public key of the abstract SADS is the key of the two hash functions $\mathcal{H}_{\mathbf{L}}$, $\mathcal{H}_{\mathbf{R}}$. Hence, the improvement of hash function key size in Figure 6 applies directly to the public key size of the entire scheme.

Preprocessing. In algorithms initialize() of Figure 2, the digest and every node labels of the generalized hash tree are initialized to be **0**s. Hence, the preprocessing time is independent of the hash function choice. Notice that in our implementations, the generalized hash tree is dynamically allocated, and the storage cost is proportional to the nonzero values stored in the leaves. In this way, we can run the experiments on a large universe of size up to $|M| = 2^{32}$.

Proof Computation. The server needs to compute the result along with a proof, responding to a query from the verifier. We discuss two ways to compute the proof on the server side. (1) The server constructs and stores the whole generalized hash tree, and returns the corresponding labels of nodes required by the proof. (2) The server only stores the values in the leaves of the generalized hash tree, and computes labels of nodes for a proof each time. There is a time-space trade-off between these two approaches. We choose the former in our implementations, in which case the proof computation is just an index searching and returning procedure. Hence, the proof computation time is the same for both instantiations.

Verification. Figure 7 shows the comparison for verification time by the instantiation with h_{new} and the instantiation with h_{old}. The x axis is the size of the universe M. The upper bound of the streaming n is fixed to 2^{32}. As we can see, the instantiation with our new hash function h_{new} outperforms the instantiation with h_{old} dramatically. Specifically, the verification time of the new instantiation with h_{new} is 7.5 times faster than the one with h_{old}. Moreover, Figure 7 shows that the verification time grows on the order of $O(\log M)$, matching what the algorithm verify() indicates. Finally, the verification time is only $10 \sim 100$ milliseconds with h_{new}, which makes the new scheme practical.

Update. Figure 8 shows that the instantiation with h_{new} runs 2.8 times faster than the one with h_{old} for the client side update. Notice that the client side update and the server side update go through the same computations, except that the server also updates the labels of nodes along the verification path. Therefore, the time cost of the client side update and the server side update is roughly the same. We omit the comparison for the server update time here. Moreover, the update time grows logarithmically with M as desired in Figure 8.

In practice, the size of the universe $|M|$ is usually fixed for a certain streaming application. To illustrate such scenario, we show the update time as n increases, with $|M|$ fixed to 2^{32}. Figure 9 shows that the instantiation with h_{new} is $1.7 \sim 3.0$ times faster than the one with h_{old}. The larger streaming volume n results in the more significant improvement in terms of update time.

Range search. The range search functionality is implemented and the results are shown in Figure 10. The range search time cost of both instantiations is slightly larger than two times of their verification time cost, and hence grows logarithmically with M as desired. Moreover, the range search of the instantiation with h_{new} is 7.4 times faster than the one with h_{old}.

Finally, Table 2 shows the statistics for both instantiations when $n = 2^{32}$ and $|M| = 2^{32}$. We can see that hashing a message by h_{new} only takes 2.74 milliseconds while *update*, *verification* and *range search* of the new instantiation with h_{new} all takes less than one second. Meanwhile, the key size of the new scheme is only

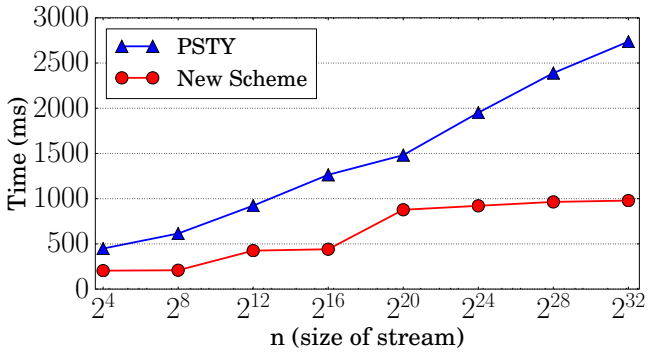

Figure 9: Update time (client side). The x axis is the size of stream n in log scale. The size of the universe is fixed to $|M| = 2^{32}$. The new scheme with h_{new} is $1.7\times$ faster when $n = 2^4$, and $3.0\times$ faster when $n = 2^{32}$, than the scheme with h_{old}.

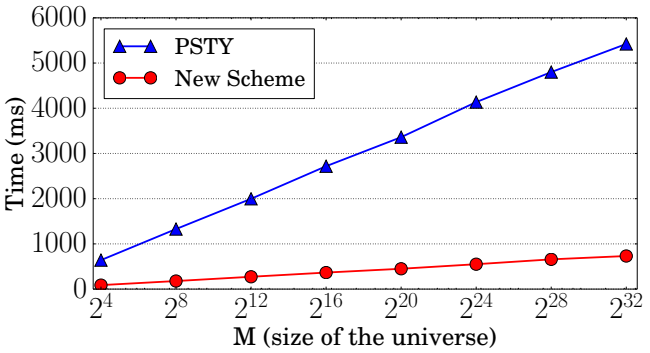

Figure 10: Range search time. The x axis is the size of universe $|M|$ in log scale. The size of stream is fixed to $n = 2^{32}$. The new scheme with h_{new} achieves $7.4\times$ speed up.

Table 2: Detail statistics for $n = 2^{32}$ and $|M| = 2^{32}$.

	running time (ms)	key size (MB)	verification (s)	update (s)	range search (s)
Instantiation with h_{new}	2.74	0.21	0.27	0.98	0.73
Instantiation with h_{old}	26.84	265.02	2.01	2.75	5.42

0.21MB, which is practical to store and manage, to the key of h_{old} which is 265MB.

8. CONCLUSION

This paper proposes an abstract construction of a streaming authenticated data structure, and presents two instantiations of the proposed abstraction. The first intantiation is the PSTY work [13]. The second instantiation is a scheme with a different collision-resistant hash function, which is based on the generalized compact knapsack (GCK) problem [8]. The new hash function is carefully parameterized, such that it is secure against state-of-the-art attacks [17, 12].

We implement both schemes. Our experiments highlight major savings in prover complexity and public key size of our second (new) instantiation over the PSTY work.

Acknowledgments

We thank Bobby Bhattacharjee, Youngsam Park, Elaine Shi and Emil Stefanov for many useful discussions.

This paper is dedicated to Emil's memory, who encouraged us to investigate the practicality of the PSTY paper.

9. REFERENCES

[1] M. Artin. *Algebra (Vol. 2.)*. Pearson, 2010.

[2] D. Boneh and X. Boyen. Memory Delegation. On the Impossibility of Efficiently Combining Collision Resistant Hash Functions. In *CRYPTO*, pp. 570–583, 2006.

[3] K.-M. Chung, Y. T. Kalai, F.-H. Liu, and R. Raz. Memory Delegation. In *CRYPTO*, pp. 151–168, 2011.

[4] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical Verified Computation with Streaming Interactive Proofs. In *ITCS*, pp. 90–112, 2012.

[5] G. Cormode, J. Thaler, and K. Yi. Verifying Computations with Streaming Interactive Proofs. In *PVLDB*, 5(1):25–36, 2011.

[6] C. Estan, G. V. C. Estan, and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. In *ACM TOCS*, 21(3):270–313, 2003.

[7] R. Gennaro, C. Gentry, and B. Parno. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *CRYPTO*, pp. 465–482, 2010.

[8] V. Lybuashevsky and D. Micciancio. Generalized Compact Knapsacks Are Collision Resistant. In *ICALP*, pp. 144–155, 2006.

[9] V. Lybuashevsky, D. Micciancio, C. Peikert, and A. Rosen. Swifft: A Modest Proposal for FFT Hashing. In *FSE*, pp. 54–72, 2008.

[10] R. C. Merkle. A Certified Digital Signature. In *CRYPTO*, pp. 218–238, 1989.

[11] D. Micciancio and O. Regev. Worst-case to Average-case Reductions Based on Gaussian Measures. In *SICOMP*, 37(1):267–302, 2007.

[12] D. Micciancio and O. Regev. Lattice-based Cryptography. In *PQCRPTO*, pp. 147–191, 2009.

[13] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi. Streaming Authenticated Data Structures. In *EUROCRYPT*, pp. 353–370, 2013.

[14] C. Papamanthou and R. Tamassia. Time and Space Efficient Algorithms for Two-party Authenticated Data Structures. In *ICICS*, pp. 1–15, 2007.

[15] B. Parno, M. Raykova, and V. Vaikuntanathan. How to Delegate and Verify in Public: Verifiable Computation from Attribute-based Encryption. In *TCC*, pp. 422–439, 2012.

[16] D. Shröder, and H. Shröder. Verifiable Data Streaming. In *CCS*, pp. 953–964, 2012.

[17] D. Wagner. A Generalized Birthday Problem. In *CRYPTO*, pp. 288–303, 2002.