New Realizations of Somewhere Statistically Binding Hashing and Positional Accumulators

Tatsuaki Okamoto [*] Krzysztof Pietrzak[†] Brent Waters[‡] Daniel Wichs[§]

September 7, 2015

Abstract

A *somewhere statistically binding (SSB) hash*, introduced by Hubáček and Wichs (ITCS '15), can be used to hash a long string x to a short digest $y = H_{\mathsf{hk}}(x)$ using a public hashing-key hk. Furthermore, there is a way to set up the hash key hk to make it statistically binding on some arbitrary hidden position i, meaning that: (1) the digest y completely determines the i'th bit (or symbol) of x so that all pre-images of y have the same value in the i'th position, (2) it is computationally infeasible to distinguish the position i on which hk is statistically binding from any other position i'. Lastly, the hash should have a *local opening* property analogous to Merkle-Tree hashing, meaning that given x and $y = H_{\mathsf{hk}}(x)$ it should be possible to create a short proof π that certifies the value of the i'th bit (or symbol) of x without having to provide the entire input x. A similar primitive called a *positional accumulator*, introduced by Koppula, Lewko and Waters (STOC '15) further supports dynamic updates of the hashed value. These tools, which are interesting in their own right, also serve as one of the main technical components in several recent works building advanced applications from indistinguishability obfuscation (iO).

The prior constructions of SSB hashing and positional accumulators required fully homomorphic encryption (FHE) and iO respectively. In this work, we give new constructions of these tools based on well studied number-theoretic assumptions such as DDH, Phi-Hiding and DCR, as well as a general construction from lossy/injective functions.

1 Introduction

SSB Hashing. A *somewhere statistically binding (SSB) hash*, introduced by Hubáček and Wichs [HW15], can be used to create a *short* digest $y = H_{\mathsf{hk}}(x)$ of some *long* input $x = (x[0], \ldots, x[L-1]) \in \Sigma^L$, where Σ is some alphabet. The *hashing key* $\mathsf{hk} \leftarrow \mathsf{Gen}(i)$ can be chosen by providing a special "binding index" i and this ensures that the hash $y = H_{\mathsf{hk}}(x)$ is statistically binding for the i'th symbol, meaning that it completely determines the value $x[i]$. In other words, even though y has many preimages x' such that $H_{\mathsf{hk}}(x') = y$, all of these preimages agree in the i'th symbol $x'[i] = x[i]$. The index i on which the hash is statistically binding should remain computationally hidden given the hashing key hk. This is formalized analogously to semantic security so that for any indices i, i' the hashing keys $\mathsf{hk} \leftarrow \mathsf{Gen}(i)$ and $\mathsf{hk}' \leftarrow \mathsf{Gen}(i')$ should be computationally indistinguishable. Moreover, we will be interested in SSB hash functions with a "local opening" property that allows us to prove that j'th symbol of x takes on some particular value $x[j] = u$ by providing a *short* opening π. This is analogous to Merkle-Tree hashing, where it is possible to open the j'th symbol of x by providing a proof π that consists of the hash values associated with all the sibling nodes along the path from the root of the tree to the j'th leaf. In the case of SSB hashing, when $j = i$ is the "binding

[*]NTT Laboratories. `okamoto.tatsuaki@lab.ntt.co.jp`.

[†]IST Austria. `pietrzak@ist.ac.at`. Research supported by ERC starting grant (259668-PSPC)

[‡]UT Austin. `bwaters@cs.utexas.edu`. Supported by NSF CNS-1228599 and CNS-1414082. DARPA SafeWare, Google Faculty Research award, the Alfred P. Sloan Fellowship, Microsoft Faculty Fellowship, and Packard Foundation Fellowship.

[§]Northeastern University. `wichs@ccs.neu.edu`. Research supported by NSF grants CNS-1347350, CNS-1314722, CNS-1413964.

index", there should (statistically) exist only one possible value that we can open $x[j]$ to by providing a corresponding proof.

Positional Accumulators. A related primitive called a *positional accumulator*, was introduced at the same time as SSB hashing by Koppula, Lewko and Waters [KLW15]. Roughly speaking, it includes the functionality of SSB hashing along with the ability to perform "local updates" where one can very efficiently update the hash $y = H_{\mathsf{hk}}(x)$ if a particular position $x[j]$ is updated. Again, this is analogous to Merkle-Tree hashing, where it is possible to update the j'th symbol of x by only updating the hash values along the path from the root of the tree to the j'th leaf.[1]

Applications of SSB Hashing and Positional Accumulators. The above tools, which are interesting in their own right, turn out to be extremely useful in several applications when combines with indistinguishability obfuscation (iO) [BGI$^+$12, GGH$^+$13]. An iO scheme can be used to obfuscate a program (given by a circuit) so that the obfuscations of any two functionally equivalent programs are indistinguishable. Although this notion of obfuscation might a-priori seem too week to be useful, recent work has shown it to be surprisingly powerful (see e.g., [SW14]). Very recently, several results showed how to use iO in conjunction with SSB hashing and positional accumulators to achieve various advanced applications. The work of [HW15] uses SSB hashing and iO to construct the first general Multi-Party Computation (MPC) protocols in the semi-honest model where the communication complexity essentially matches that of the best insecure protocol for the same task. The work of [KLW15] uses positional accumulators and iO to construct succinct garbling for Turing Machines, and recent work extends this approach to RAM programs [CH15, CCC$^+$15]. Lastly, the work of [Zha14] uses SSB hashing and iO to construct the first adaptively secure broadcast encryption with short system parameters.

Example: the power of iO + SSB. To see the usefulness of combining iO and SSB hashing (or positional accumulators), let's take a simple illustrative example, adapted from [HW15].[2] Imagine that Alice has a (small) secret circuit C, and both Alice and Bob know a public value $x \in \Sigma^L$. Alice wishes to communicate the values $\{C(x[i])\}_{i \in [L]}$ to Bob while hiding some information about C. In particular, Bob shouldn't learn whether Alice has the circuit C or some other C' that satisfies $C(x[i]) = C'(x[i])$ for each $i \in [L]$. Note that C and C' may not be functionally equivalent and they only agree on the inputs $\{x[i]\}_{i \in [L]}$ but might disagree on other inputs. A naive secure solution would be for Alice to simply send the outputs $\{C(x[i])\}_{i \in [L]}$ to Bob, but this incurs communication proportional to L. An insecure but communication-efficient solution would be for Alice to just send the small circuit C to Bob. Can we get a secure solution with comparable communication independent of L? Simply sending an obfuscated copy of C is not sufficient since the circuits C, C' are not functionally equivalent and therefore their obfuscations might be easily distinguishable. However it is possible to achieve this with iO and SSB hashing. Alice can send an obfuscation of a circuit that has the hash $y = H_{\mathsf{hk}}(x)$ hard-coded and takes as input a tuple (j, u, π): it checks that $j \in [L]$ and that π is a valid opening to $x[j] = u$ and if so outputs $C(u)$. Bob can evaluate this circuit on the values $\{x[j]\}_{j \in [L]}$ by providing the appropriate openings. It is possible to show that the above hides whether Alice started with C or C'. The proof proceeds in a sequence of L hybrids where in the i'th hybrid we obfuscate a circuit C_i that runs C' instead of C when $j \le i$ and otherwise runs C. To go from hybrid i to $i + 1$ we first switch the SSB hash key hk to be binding in position $i + 1$ and then we can switch from obfuscating C_i to C_{i+1} by arguing that these are functionally equivalent; they only differ in the code they execute for inputs of the form $(j = i + 1, u, \pi)$ where π is a valid proof but in this case, by the statistical binding property, the only

[1] The formal definitions of SSB hashing and positional accumulators as given in [HW15, KLW15] are technically incomparable. On a high level, the latter notion requires additional functionality in the form of updates but only insists on a weaker notion of security which essentially corresponds to "target collision resistance" where the target hash value is computed honestly. In this work, we construct schemes that achieve the best of both worlds, having the additional functionality and the stronger security.

[2] The contents of this paragraph and the notion of iO are not essential to understand the results of the paper, but we provide it to give some intuition for how SSB hashing and positional accumulators are used in conjunction with iO in prior works to get the various applications described above.

possible value u for which a valid proof π exists is the unique value $u = x[j]$ for which both circuits produce the same output $C(x[j]) = C'(x[j])$.

Prior Constructions of SSB and Positional Accumulators. The work of [HW15] constructed a SSB hash by relying on fully homomorphic encryption (FHE). Roughly speaking the construction combines FHE with Merkle Hash Trees. To hash some value $x = (x[0], \ldots, x[L-1])$ the construction creates a full binary tree of height $\log L$ (for simplicity, assume L is a power of 2) and deterministically associates a ciphertext with each node of the tree. The L leaf nodes will be associated with some deterministically created encryptions of the values $x[0], \ldots, x[L-1]$, say by using all 0s for the random coins of the encryption procedure. The hash key hk consists of an encryption of a path from the root of the tree to the i'th leaf where i is the binding index; concretely it contains $\log L$ FHE ciphertexts $(ct_1, \ldots, ct_{\log L})$ which encrypt bits $\beta_1, \ldots, \beta_{\log L}$ corresponding to the binary representation of the binding index i so that $\beta_i = 0$ denotes "left" and $\beta_i = 1$ denotes "right". The ciphertext associated with each non-leaf node are computed homomorphically to ensure that the value $x[i]$ is contained in each ciphertext along the path from the root to the i'th leaf. Concretely, the ciphertext associated with some node at level j is is determined by a homomorphic computation which takes the two child ciphertexts c_0 (left) and c_1 (right) encrypting some values m_0, m_1 and the ciphertext ct_i contained in hk which encrypts β_i and homomorphically produces a ciphertext encrypting m_{β_i}. (For technical reasons, the actual construction is a bit more complicated and needs to use a different FHE key at each level of the tree – see [HW15] for full details.) This ensures that the binding index i is hidden by the semantic security of FHE and the statistically binding property follows by the correctness of FHE.

The work of [KLW15] constructs positional accumulators by also relying on a variant of Merkle Trees. However, instead of FHE, it relies on standard public-key encryption and iO. (It is relatively easy to see that the scheme of [HW15] would also yield an alternate construction of a positional accumulator).

1.1 Our Results

In this work we give new constructions of SSB hashing and positional accumulators from a wide variety of well studied number theoretic assumptions such as DDH, DCR (decisional composite residuocity), ϕ-hiding, LWE and others.

Two-to-One SSB. We first abstract out the common Merkle-tree style approach that is common to both SSB hashes and positional accumulators, and identify a basic underlying primitive that we call a *two-to-one* SSB hash, which can be used to instantiate this approach. Intuitively a two-to-one SSB hash takes as input $x = (x[0], x[1]) \in \Sigma^2$ consisting of just two alphabet symbols and outputs a value $y = H_{\mathsf{hk}}(x)$ which is not much larger than than a single alphabet symbol. The key hk can be set up to be statistically binding on either position 0 or 1.

Instantiations of Two-to-One SSB. We show how to instantiate a two-to-one SSB hash from the DDH assumption and the decisional composite residuocity (DCR) assumption. More generally, we show how to instantiate a (slight variant of) two-to-one SSB hash from any lossy/injective function. This is a family of functions $f_{\mathsf{pk}}(x)$ where the public key pk can be picked in one of two indistinguishable modes: in injective mode, the function $f_{\mathsf{pk}}(x)$ is an injective function and in lossy mode $f_{\mathsf{pk}}(x)$ it is a many-to-one function. To construct a two-to-one SSB hash from injective/lossy function we pick two public keys $\mathsf{hk} = (\mathsf{pk}_0, \mathsf{pk}_1)$ and define $H_{\mathsf{hk}}(x[0], x[1]) = h(f_{\mathsf{pk}_0}(x[0]), f_{\mathsf{pk}_1}(x[1]))$ where h is a universal hash function. To make the hk binding on index 0 we choose pk_0 to be injective and pk_1 to be lossy and to make is binding on index 1 we do the reverse. With appropriate parameters, we can ensure that the statistically binding property holds with overwhelming probability over the choice of h.

From Two-to-One SSB to Full SSB and Positional Accumulators. We can instantiate a (full) SSB hash with arbitrary input size Σ^L by combining two-to-one SSB hashes in a Merkle Tree, with a different key at each level. To make the full SSB binding at some location i, we choose the hash keys at each level to be binding on either the left or right child in such a way that they are binding along the path from the root of the tree to the leaf at position i. This allows us to "locally open" the j'th position of the input in the usual way, by giving the hash values of all the siblings along the path from the root to the j'th leaf. If $j = i$ is the binding index, then there is a unique value $x[j] = u$ for which there is a valid opening. To get positional accumulators, we use the fact that we can also locally update the hashed value by modifying one location $x[j]$ and only updating the hashes along the path from the root to the j'th leaf.

A Flatter Approach. We also explore a different approach for achieving SSB hashing from the ϕ-hiding assumption, which does not go through a Merkle-Tree type construction. Roughly our approach uses a construction is structurally similar to standard constructions RSA accumulators [BdM93]. However, we construct a modus N to be such that for some given prime exponent e we have that e divides $\phi(N)$. This means that if $y \in \mathbb{Z}_N$ is not an e-th residue $\mod N$, then there exists no value $\pi \in \mathbb{Z}_N$ where $\pi^e = y$. This will lead to our statistical binding property as we will leverage this fact to make the value e related to an index we wish to be binding on. Index hiding will follow from the ϕ-hiding assumption.

2 Preliminaries

SSB Hash (with Local Opening). Our definition follows that of [HW15], but whereas that work only defined SSB hash which included the local opening requirement by default, it will be convenient for us to also separately define a weaker variant which does not require the local opening property.

Definition 2.1 (SSB Hash)**.** A somewhere statistically binding (SSB) hash consists of PPT algorithms $\mathcal{H} = (\mathsf{Gen}, H)$ and a polynomial $\ell(\cdot,\cdot)$ denoting the output length.

- $\mathsf{hk} \leftarrow \mathsf{Gen}(1^\lambda, 1^s, L, i)$: Takes as input a security parameter λ a block-length s an input-length $L \leq 2^\lambda$ and an index $i \in \{0, \ldots, L-1\}$ (in binary) and outputs a public hashing key hk. We let $\Sigma = \{0,1\}^s$ denote the block alphabet. The output size is $\ell = \ell(\lambda, s)$ and is independent of the input-length L.
- $H_{\mathsf{hk}} : \Sigma^L \to \{0,1\}^\ell$: A deterministic poly-time algorithm that takes as input $x = (x[0], \ldots, x[L-1]) \in \Sigma^L$ and outputs $H_{\mathsf{hk}}(x) \in \{0,1\}^\ell$.

We require the following properties:

Index Hiding: We consider the following game between an attacker $\mathcal{A}$ and a challenger:

- The attacker $\mathcal{A}(1^\lambda)$ chooses parameters $1^s, L$ and two indices $i_0, i_1 \in \{0, \ldots, L-1\}$.
- The challenger chooses a bit $b \leftarrow \{0,1\}$ and sets $\mathsf{hk} \leftarrow \mathsf{Gen}(1^\lambda, 1^s, L, i_b)$.
- The attacker $\mathcal{A}$ gets hk and outputs a bit b'.

We require that for any PPT attacker $\mathcal{A}$ we have $|\Pr[b = b'] - \frac{1}{2}| \leq \text{negl}(\lambda)$ in the above game.

Somewhere Statistically Binding: We say that hk is *statistically binding for an* index $i \in [L]$ if there do not exist any values $x, x' \in \Sigma^L$ with $x[i] \neq x'[i]$ such that $H_{\mathsf{hk}}(x) = H_{\mathsf{hk}}(x')$. We require that for any parameters s, L and any integer $i \in \{0, \ldots, L-1\}$ we have:

$$\Pr[\mathsf{hk} \text{ is statistically binding for index } i \ : \ \mathsf{hk} \leftarrow \mathsf{Gen}(1^\lambda, 1^s, L, i)] \geq 1 - \text{negl}(\lambda).$$

We say that the hash is *perfectly binding* if the above probability is 1.

Definition 2.2 (SSB Hash with Local Opening)**.** An *SSB Hash with local opening* $\mathcal{H} = (\mathsf{Gen}, H, \mathsf{Open}, \mathsf{Verify})$ consists of an SSB hash (Gen, H) with output size $\ell(\cdot,\cdot)$ along with two additional algorithms $\mathsf{Open}, \mathsf{Verify}$ and an opening size $p(\cdot,\cdot)$. The additional algorithms have the following syntax:

- $\pi \leftarrow \mathsf{Open}(\mathsf{hk}, x, j)$: Given the hash key hk, $x \in \Sigma^L$ and an index $j \in \{0, \ldots, L-1\}$, creates an opening $\pi \in \{0,1\}^p$. The opening size $p = p(\lambda, s)$ is a polynomial which is independent of the input-length L.

- $\mathsf{Verify}(\mathsf{hk}, y, j, u, \pi)$: Given a hash key hk a hash output $y \in \{0,1\}^\ell$, an integer index $j \in \{0, \dots, L-1\}$, a value $u \in \Sigma$ and an opening $\pi \in \{0,1\}^p$, outputs a decision $\in \{\mathsf{accept}, \mathsf{reject}\}$. This is intended to verify that a pre-image x of $y = H_{\mathsf{hk}}(x)$ has $x[j] = u$.

We require the following two additional properties.

Correctness of Opening: For any parameters s, L and any indices $i, j \in \{0, \dots, L-1\}$, any $\mathsf{hk} \leftarrow \mathsf{Gen}(1^\lambda, 1^s, L, i)$, $x \in \Sigma^L$, $\pi \leftarrow \mathsf{Open}(\mathsf{hk}, x, j)$: we have $\mathsf{Verify}(\mathsf{hk}, H_{\mathsf{hk}}(x), j, x[j], \pi) = \mathsf{accept}$.

Somewhere Statistically Binding w.r.t. Opening: [3] We say that hk is *statistically binding w.r.t opening (abbreviated SBO) for an* index i if there do not exist any values $y, u \neq u', \pi, \pi'$ s.t.

$$\mathsf{Verify}(\mathsf{hk}, y, i, u, \pi) = \mathsf{Verify}(\mathsf{hk}, y, i, u', \pi') = \mathsf{accept}.$$

We require that for any parameters s, L and any index $i \in \{0, \dots, L-1\}$

$$\Pr[\mathsf{hk} \text{ is SBO for index } i \ : \ \mathsf{hk} \leftarrow \mathsf{Gen}(1^\lambda, 1^s, L, i)] \geq 1 - \mathrm{negl}(\lambda).$$

We say that the hash is *perfectly binding w.r.t. opening* if the above probability is 1.

Fixed-Parameter Variants. The above definitions allow for a flexible input-length L and block-length s specified by the user as inputs to the Gen algorithm. This will be the default throughout the paper, but we also consider variants of the above definition with a *fixed-input-length* L and/or *fixed-block-length* s where these values cannot be specified by the user as inputs to the Gen algorithm but are instead set to some fixed value (a constant or polynomial in the security parameter λ) determined by the scheme. In the case of a fixed-input-length variant, the definitions are non-trivial if the output-length ℓ and opening-size p satisfy $\ell, p < L \cdot s$.

Discussion. There are several constructions of SSB hash that do not provide local opening. For example, any PIR scheme can be used to realize an SSB hash without local opening. The hash key hk consists of a PIR query for index i and the hash $H_{\mathsf{hk}}(x)$ simply computes the PIR response using database x. Unfortunately, we do not know how to generically add a local opening capability to such SSB hash constructions.

3 Two-to-One SSB Hash

As our main building block, we rely on a notion of a "two-to-one SSB hash". Informally, this is a fixed-input-length and flexible-block-size SSB hash (we do not require local opening) that maps two input blocks ($L = 2$) to an output which is roughly the size of one block (up to some small multiplicative and additive factors).

Definition 3.1 (Two-to-One SSB Hash)**.** A *two-to-one* SSB hash is an SSB hash with a fixed input-length $L = 2$ and flexible block-length s. The output-length is $\ell(\lambda, s) = s \cdot (1 + 1/\Omega(\lambda)) + \textsc{poly}(\lambda)$.

We give three constructions of a Two-to-One SSB Hash systems. Our first construction is built from the DDH-hard groups with compact representation. This construction achieves perfect binding. Our next construction is built from the DCR assumption. Lastly, we generalize our approach by showing a (variant of) Two-to-One SSB hashing that can work from *any* lossy function. We note that lossy functions can be built from a variety of number theoretic primitives including DDH (without compact representation), Learning with Errors, and the ϕ-hiding assumption.

[3]Note that the "somewhere stat. binding w.r.t. opening" property implies the basic "somewhere stat. binding" property of SSB hash.

Remark: Impossibility without Overhead. We note that the need for some "slack" is inherent in the above definition and we cannot get a two-to-one SSB hash where the output is exactly $\ell(\lambda, s) = s$ matching the size of one of the inputs. This is because in that case, if we choose $\mathsf{hk} \leftarrow \mathsf{Gen}(1^\lambda, 1^s, i = 0)$ then for each $x_0 \in \{0,1\}^s$ there is a unique choice of $y \in \{0,1\}^s$ such that $H_{\mathsf{hk}}(x_0, x_1) = y$ no matter what x_1 is. In other words, the function $H_{\mathsf{hk}}(x_0, x_1)$ does not depend on the argument x_1. Symmetrically, if $\mathsf{hk} \leftarrow \mathsf{Gen}(1^\lambda, 1^s, i = 1)$ then the function $H_{\mathsf{hk}}(x_0, x_1)$ does not depend on the argument x_0. These two cases are easy to distinguish.

3.1 Two-to-One SSB Hash from DDH

3.1.1 DDH Hard Groups and Representation Overhead

Let $\mathcal{G}$ be a PPT group generator algorithm that takes as input the security parameter 1^λ and outputs a pair $\mathbb{G}, p$ where $\mathbb{G}$ is a group description of prime order p for $p \in \Theta(2^\lambda)$.

Assumption 1 (decision Diffie-Hellman Assumption)**.** Let $(\mathbb{G}, p) \leftarrow \mathcal{G}(1^\lambda)$ and $b \leftarrow \{0,1\}$. Choose a random generator $g \in G$ and random $x, y \in \mathbb{Z}_p$ Let $T \leftarrow \mathbb{G}$ if $b = 0$, else $T \leftarrow g^{xy}$. The advantage of algorithm $\mathcal{A}$ in solving the decision Diffie-Hellman problem is defined as

$$\mathsf{Adv}_{\mathcal{A}} = \left| Pr[b \leftarrow \mathcal{A}(\mathbb{G}, p, g, g^x, g^y, T)] - \frac{1}{2} \right|.$$

We say that the Decision-Diffie Hallman assumption holds if for all PPT $\mathcal{A}$, $\mathsf{Adv}_{\mathcal{A}}$ is negligible in λ.

Representation overhead of group elements In this work we will be concerned with how efficiently (prime order) group elements are represented. We are interested in the difference between the number of bits to represent a group element and $\lfloor \lg(p) \rfloor$. In our definition we consider the bit representation of a group to be intrinsic to a particular group description.

Definition 3.2 (Representational Overhead)**.** Consider a family of prime order groups output from some group generation algorithm $\mathcal{G}(1^\lambda)$ that outputs a group of prime order p for $2^\lambda < p < 2^{\lambda+1}$. Notice that for a generator g in such a group that $g^i \neq g^j$ for $i, j \in [0, 2^\lambda]$ and $i \neq j$. (I.e. no "wraparound" happens.)

We define the representational overhead $\delta(\lambda)$ to be the function which expresses maximum difference between the number of bits used to represent a group element of $\mathbb{G}$ and λ, where $\mathbb{G}, p \leftarrow \mathcal{G}(1^\lambda)$.

For this work we are interested in families of groups who representational overhead $\delta(\lambda)$ is some constant c. Examples of these include groups generated from strong primes and certain elliptic curve groups.

3.1.2 Construction of Two-to-One SSB

We now describe our Two-To-One SSB Hash. We will use a group generation algorithm $\mathcal{G}$ that has constant representational overhead c as defined in Definition 3.2. Consider a matrix $\mathbf{M}$ over $\mathbb{Z}_p$ and group generator g of order p we will use the notation $g^{\mathbf{M}}$ as short hand for giving out g raised to each element of $\mathbf{M}$.

The construction sets up a hash function key hk for a function that takes two s bit inputs x_A and x_B. If the index bit $\beta = 0$ it will be statistically binding on x_A; otherwise it is statistically binding on x_B. At a high level the construction setup is intuitively similar to the Lossy trapdoor function algorithms of Peikert and Waters [PW08] where the setup creates two functions — one injective and the other lossy and assigns whether the lossy function corresponds to the A or B input according to the index bit β.

There are two important differences from the basic PW construction. First the PW construction encrypted the input bit by bit. This low rate of encoding was needed in order to recover the input from a trapdoor in [PW08], but a trapdoor is not required for our hash function. Here we cram in as many bits into a group element as possible. This is necessary to satisfy the SSB output size properties. We note [BHK11] pack bits in a similar manner. The second property we have is that the randomness used to generate both the injective and lossy function is correlated such that we can intuitively combine the outputs of each into one output where the output length is both small and maintains the committing property of the injective function. We note that our description describes the procedures directly and the connection to injective and lossy functions is given for intuition, but not made formal.

$\mathsf{Gen}_{\text{Two-to-One}}(1^\lambda, 1^s, \beta \in \{0,1\})$
The generation algorithm first sets $t = \max(\lambda, \lfloor \sqrt{s \cdot c} \rfloor)$. (The variable t will be the number of bit each group element can uniquely represent.) It then calls $\mathcal{G}(1^t) \to (\mathbb{G}, p)$ with $2^t < p < 2^{t+1}$ and chooses a random generator $g \in \mathbb{G}$.

Next, it lets $d = \lceil \frac{s}{t} \rceil$. It then chooses random $w_1, \ldots, w_d \in \mathbb{Z}_p$, two random column vectors $\mathbf{a} = (a_1, \ldots, a_d) \in \mathbb{Z}_p^d$ and $\mathbf{b} = (b_1, \ldots, b_d) \in \mathbb{Z}_p^d$. We let $\tilde{\mathbf{A}}$ be the $d \times d$ matrix over $\mathbb{Z}_p$ where the (i,j)-th entry is $a_i \cdot w_j$ and $\tilde{\mathbf{B}}$ be the $d \times d$ matrix over $\mathbb{Z}_p$ where the (i,j)-th entry is $b_i \cdot w_j$. Finally, let $\mathbf{A}$ be $\tilde{\mathbf{A}} + (1-\beta) \cdot \mathbf{I}$ and $\mathbf{B}$ be $\tilde{\mathbf{B}} + \beta \cdot \mathbf{I}$ where $\mathbf{I}$ is the identity matrix. (I.e. we add in the identity matrix to $\tilde{\mathbf{A}}$ to get the $\mathbf{A}$ matrix if the selection bit $\beta = 0$; otherwise, if $\beta = 1$ add in the identity matrix to $\tilde{\mathbf{B}}$ to get $\mathbf{B}$.)

The hash key is $\mathsf{hk} = (g^{\mathbf{a}}, g^{\mathbf{b}}, g^{\mathbf{A}}, g^{\mathbf{B}})$.

$H_{\mathsf{hk}} : \{0,1\}^s \times \{0,1\}^s \to \mathbb{G}^{d+1}$
The hash function algorithm takes in two inputs $x_A \in \{0,1\}^s$ and $x_B \in \{0,1\}^s$. We can view the bitstrings x_A and x_B each as consisting of d blocks each of t bits (except the last block which may be less). The function first parses these each as row vectors $\mathbf{x}_A = (x_{A,1}, \ldots, x_{A,d})$ and $\mathbf{x}_B = (x_{B,1}, \ldots, x_{B,d})$. These have the property that for $j \in [d]$ we have $x_{A,j}$ is an integer $< 2^t \leq p$ representing the $j - th$ block of bits as an integer.

Next, it computes

$$V = g^{\mathbf{x}_A \mathbf{a} + \mathbf{x}_B \mathbf{b}}, \; Y = g^{\mathbf{x}_A \mathbf{A} + \mathbf{x}_B \mathbf{B}}.$$

We observe that V is one group element in $\mathbb{G}$ and Y is a vector of d group elements. Thus the output size of the hash is $(d+1) \cdot (t+c)$ bits.

3.1.3 Analysis

We now analyze the size overhead, index hiding and binding properties of the hash function.

Overhead The output of the hash function is $d+1$ group elements each of which takes $t+c$ bits to represent for a total output size of $(d+1)(t+c)$ bits. In the case where $\lfloor \sqrt{s \cdot c} \rfloor \geq \lambda$, we can plug in our parameter choices for t, d and see that the outputsize $\ell(\lambda, s) = s + \mathcal{O}(\sqrt{s})$, thus matching the requirements of Definition 3.1. In the case where $\lfloor \sqrt{s \cdot c} \rfloor < \lambda$ we have that $\ell(\lambda, s) = s + \mathcal{O}(\lambda)$ thus also matching our definition.

Somewhere Statistically Binding We show that the hash function above is selectively binding respective to the bit β. We demonstrate this for the $\beta = 0$ case. The $\beta = 1$ case follows analogously.

Suppose a hash key hk were setup according to the process $\mathsf{Gen}_{\text{Two-to-One}}$ as above with the input $\beta = 0$. Now consider the evaluation $H_{\mathsf{hk}}(x_A, x_B) = (V, Y = (Y_1, \ldots, Y_d))$. We have that for all $j \in [1, d]$ that $Y_j / V^{w_j} = g^{x_{A,j}}$. Let's verify this claim. First from the hash definition we can work out that

$$V = g^{\Sigma_{i \in [d]} x_{A,i} a_i + x_{B,i} b_i}$$

and

$$Y_j = g^{x_{A,j} + \Sigma_{i \in [d]} x_{A,i}(a_i w_j) + x_{B,i}(b_i w_j)} = g^{x_{A,j}} g^{w_j (\Sigma_{i \in [d]} x_{A,i} a_i + x_{B,i} b_i)}.$$

The claim that $Y_j / V^{w_j} = g^{x_{A,j}}$ follows immediately from these equations.

Now suppose that we are given two inputs (x_A, x_B) and (x'_A, x'_B) such that $x_A \neq x'_A$ There must then exist some j such that $x_{A,j} \neq x'_{A,j}$. Let $H_{\mathsf{hk}}(x_A, x_B) = (V, Y = (Y_1, \ldots, Y_d))$ and $H_{\mathsf{hk}}(x'_A, x'_B) = (V', Y' = (Y'_1, \ldots, Y'_d))$. From the above claim it follows that $Y_j / V^{w_j} = g^{x_{A,j}}$ and $Y_j / V^{w_j} = g^{x'_{A,j}}$. Therefore $(V, Y_j) \neq (V', Y'_j)$ and the outputs of the hashes are distinct.

Index Hiding We now prove index hiding. To do this we define $\mathsf{Game}_{\text{normal}}$ to be the normal index hiding game on the two-to-one construction and $\mathsf{Game}_{\text{random}}$ to be the index hiding game, but where the matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$ are chosen randomly when constructing the hash function hk.

We first argue that if the decision Diffie-Hellman assumption holds, then the advantage of any PPT attacker $\mathcal{A}$ in $\mathsf{Game}_{\text{normal}}$ must be negligibly close to its advantage in $\mathsf{Game}_{\text{random}}$. To show this we apply a particular case of the decision matrix linear assumption family introduced by Naor and Segev [NS12]. They show (as part of a more general theorem) that if the decision Diffie-Hellman assumption holds that a PPT attacker cannot distinguish if a $2d \times (d+1)$ matrix $\mathbf{M}$ over $\mathbb{Z}_p$ was sampled randomly from the set of rank 1 matrices or rank $d+1$ matrices given $g^{\mathbf{M}}$.

Suppose that the difference of advantage for some attacker in $\mathsf{Game}_{\text{normal}}$ and $\mathsf{Game}_{\text{random}}$ is some non-negligible function of λ. Then we construct an algorithm $\mathcal{B}$ on the above decision matrix linear assumption. $\mathcal{B}$ receives a challenge $g^{\mathbf{M}}$ and breaks this into $g^{\mathbf{M}_A}$ and $g^{\mathbf{M}_B}$ where $\mathbf{M}_A$ is the top half of the matrix $\mathbf{M}$ and $\mathbf{M}_B$ is the bottom half. It then takes $g^{\mathbf{a}}$ from the first column of $g^{\mathbf{M}_A}$ and $g^{\tilde{\mathbf{A}}}$ as the remaining d columns. Similarly, $\mathcal{B}$ takes $g^{\mathbf{b}}$ from the first column of $g^{\mathbf{M}_B}$ and $g^{\tilde{\mathbf{B}}}$ as the remaining d columns. It then samples a random index $\beta \in \{0,1\}$ and continues to use these values in executing $\mathsf{Gen}_{\text{Two-to-One}}$, giving the hash key hk to the attack algorithm.

If $g^{\mathbf{M}}$ were sampled as a rank 1 matrix, then the view of the attacker is the same as executing $\mathsf{Game}_{\text{normal}}$. Otherwise, if $g^{\mathbf{M}}$ were sampled as a rank $d+1$ matrix the attacker's view is statistically close to $\mathsf{Game}_{\text{random}}$ (as choosing a random rank $d+1$ matrix is statistically close to choosing a random matrix). If the attacker $\mathcal{A}$ correctly guesses $\beta' = \beta$, then $\mathcal{B}$ guesses the matrix was rank 1, else it guesses it was rank $d+1$. If the difference in advantage of $\mathcal{A}$ in the two games is non-neglgibile, then $\mathcal{B}$ has a non-negligible advantage in the decision matrix game.

Finally, we see that in $\mathsf{Game}_{\text{random}}$ any attacker's advantage must be 0 as the distributions of the outputs are independent of β.

3.2 Two-to-One SSB Hash from DCR

We can also construct a two-to-one hash with perfect binding from the decisional composite residuocity (DCR) assumption. We do so by relying on the Damgård-Jurik cryptosystem [DJ01] which is itself a generalization of the Pallier cryptosystem based on the DCR assumption [Pai99]. We rely on the fact that this cryptosystem is additively homomorphic and "length flexible", meaning that it has a small ciphertext expansion. When we plug this construction of a two-to-one SSB hash into our full construction of SSB hash with local opening, we essentially get the private-information retrieval (PIR) scheme of Lipmaa [Lip05]. Note that, in general, PIR implies SSB hash but only without local opening. However, the particular PIR construction of [Lip05] already has a tree-like structure which enables efficient local opening.

Damgård-Jurik. The Damgård-Jurik cryptosystem consists of algorithms (KeyGen, Enc, Dec). The key generation $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ generates a public key $\mathsf{pk} = n = pq$ which is a product of two primes p, q and $\mathsf{sk} = (p, q)$. For any (polynomial) w the scheme can be instantiated to have plaintext space $\mathbb{Z}_{n^w}$ and ciphertext space $\mathbb{Z}^*_{n^{w+1}}$. The encryption/decryption procedures $c = \mathsf{Enc}_{\mathsf{pk}}(m; r)$ and $\mathsf{Dec}_{sk}(c)$ satisfy perfect correctness so that for all $m \in \mathbb{Z}_{n^w}$ and all possible choices of the randomness r we have $\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(m; r)) = m$. Moreover the scheme is additively homomorphic, meaning that there is an operation $\oplus$ such that $\mathsf{Enc}_{\mathsf{pk}}(m; r) \oplus \mathsf{Enc}_{\mathsf{pk}}(m'; r') = \mathsf{Enc}_{\mathsf{pk}}(m + m'; r'')$ for some r'' (the operation $+$ is in the ring $\mathbb{Z}_{n^w}$). Similarly, we can define homomorphic subtraction $\ominus$. Furthermore, by performing repeated addition we can also implement an operation $\otimes$ that allows for multiplication by a plaitnext element $\mathsf{Enc}_{\mathsf{pk}}(m; r) \otimes m' = \mathsf{Enc}_{\mathsf{pk}}(m \cdot m'; r')$ for some r' (the operation $\cdot$ is in the ring $\mathbb{Z}_{n^w}$). The semantic security of the cryptosystem holds under the DCR assumption.

Construction of Two-to-One SSB. We use the Damgård-Jurik cryptosystem to construct an SSB hash as follows.

$\mathsf{hk} \leftarrow \mathsf{Gen}(1^\lambda, 1^s, \beta \in \{0,1\})$ Choose $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ to be a Damgård-Jurik public/secret key. We assume (without loss of generality) that the modulus n satisfies $n > 2^\lambda$. Set the parameter w which determines the plaintext space $\mathbb{Z}_{n^w}$ and the ciphertext space $\mathbb{Z}^*_{n^{w+1}}$ to be $w = \lceil s/\log n \rceil$ so that we can interpret $\{0,1\}^s$ as a subset of $\mathbb{Z}_{n^w}$. Choose $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\beta)$ and output $\mathsf{hk} = (\mathsf{pk}, c)$.

$H_{\mathsf{hk}}(x_0, x_1)$**:** Parse $\mathsf{hk} = (\mathsf{pk}, c)$ and interpret the values $x_0, x_1 \in \{0,1\}^s$ as ring elements $x_0, x_1 \in \mathbb{Z}_{n^w}$. Define the value $\mathbf{1}_{ct} = \mathsf{Enc}_{\mathsf{pk}}(1; r_0)$ to be a fixed encryption of 1 using some fixed randomness r_0 (say, all 0s). Compute $c^* := (x_1 \otimes c) \oplus (x_0 \otimes (\mathbf{1}_{ct} \ominus c))$. By the homomorphic properties of encryption, c^* is an encryption of x_β.

Theorem 3.1. The above construction is a two-to-one SSB hash with perfect binding under the DCR assumption.

Proof. The index hiding property follows directly from the semantic security of the Damgård-Jurik cryptosystem, which in turn follows from the DCR assumption.

The perfect binding property follows from the perfect correctness of the cryptosystem. In particular, if $\mathsf{hk} \leftarrow \mathsf{Gen}(1^\lambda, 1^s, \beta)$ then $y = H_{\mathsf{hk}}(x_0, x_1)$ satisfies $y = \mathsf{Enc}_{\mathsf{pk}}(x_\beta; r)$ for some r which perfectly determines x_β.

Lastly, the output size of the hash function is

$$\begin{aligned} \ell(s, \lambda) &= (w+1)\lceil \log n \rceil = (\lceil s/\log n \rceil + 1)\lceil \log n \rceil \\ &\leq (1 + 1/\log n)s + O(\log n) = (1 + 1/\Omega(\lambda))s + \mathsf{poly}(\lambda). \end{aligned}$$

■

3.3 SSB with Local Opening from Two-to-One SSB

We now show how to construct a SSB hash with local opening from a two-to-one SSB hash via the "Merkle Tree" construction. Assume that $\mathcal{H} = (\mathsf{Gen}, H)$ is a two-to-one SSB hash family with output length give by $\ell(s, \lambda)$. We use this hash function in a Merkle-Tree to construct an SSB hash with local opening $\mathcal{H}^* = (\mathsf{Gen}^*, H^*, \mathsf{Open}, \mathsf{Verify})$ as follows.

- $\mathsf{hk} \leftarrow \mathsf{Gen}^*(1^\lambda, 1^s, L, i)$: Let $(b_q, \ldots, b_1)$ be the binary representation of i (with b_1 being the least significant bit) where $q = \lceil \log L \rceil$. For $j \in [q]$ define the block-lengths $s_1, \ldots, s_q$ where $s_1 = s$ and $s_{j+1} = \ell(s_j, \lambda)$. Choose $\mathsf{hk}_j \leftarrow \mathsf{Gen}(1^\lambda, 1^{s_j}, b_j)$ and output $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_q)$.
- $y = H^*_{\mathsf{hk}}(x)$: For $x = (x[0], \ldots, x[L-1]) \in \Sigma^L$, $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_q)$ proceed as follows. Define T to be a complete binary tree of height q where level 0 of the tree denotes the leaves and level q denotes the root. We will assign a label to each vertex in the tree. The L leaf vertices are assigned the labels $x[0], \ldots, x[L-1]$. The rest of the labels are assigned inductively where each non-leaf vertex v at level j of the tree with children that have labels x'_0, x'_1 gets assigned the label $H_{\mathsf{hk}_j}(x'_0, x'_1)$. The output of the hash is the label y assigned to the root of the tree.
- $\pi = \mathsf{Open}(\mathsf{hk}, x, j)$: Compute the labeled tree T as above. Output the labels of all the sibling nodes along the path from the root to the j'th leaf.
- $\mathsf{Verify}(\mathsf{hk}, y, j, u, \pi)$: Recompute all of the labels of the nodes in the tree T that lie on the path from the root to the j'th leaf by using the value u for that leaf and the values given by π as the labels of all the sibling nodes along the path. Check that the recomputed label on the root of the tree is indeed y.

Theorem 3.2. If $\mathcal{H}$ is a two-to-one SSB hash then $\mathcal{H}^*$ is a SSB hash with local opening.

Proof. Firstly, the *index hiding* property of $\mathcal{H}^*$ follows directly from that of $\mathcal{H}$ via q hybrid arguments. In particular, if $i_0 = (b^0_q, \ldots, b^0_1)$ and $i_1 = (b^1_q, \ldots, b^1_1)$ are the two indices chosen by the attacker during the security game for index hiding, then we can prove the indistinguishability of $\mathsf{hk}^0 \leftarrow \mathsf{Gen}^*(1^\lambda, L, s, i_0)$ and

$\mathsf{hk}^1 \leftarrow \mathsf{Gen}^*(1^\lambda, L, s, i_1)$ via q hybrid games where we switch the component keys $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_q)$ from being chosen as $\mathsf{hk}_j \leftarrow \mathsf{Gen}(1^\lambda, 1^s, b_j^0)$ to $\mathsf{hk}_j \leftarrow \mathsf{Gen}(1^\lambda, 1^s, b_j^1)$.

Secondly, to show that $\mathcal{H}^*$ is *somewhere statistically binding w.r.t. opening*, assume that there exist some $y, u \neq u', \pi, \pi'$ s.t. $\mathsf{Verify}(\mathsf{hk}, y, i, u, \pi) = \mathsf{Verify}(\mathsf{hk}, y, i, u', \pi') = \mathsf{accept}$. Recall that the verification procedure assigns labels to all the nodes along the path from the root to the i'th leaf. During the two runs of the verification procedure with the above inputs, let $0 < j \leq q$ be the lowest level at which both runs assign the same label w to the node at level j (this must exist since the root at level q is assigned the same label y in both runs and the leafs at level 0 are assigned different values u, u' in the two runs). Let v, v' be the two different labels assigned to the node at level $j-1$ by the two runs. Then $w = H_{\mathsf{hk}_j}(x) = H_{\mathsf{hk}_j}(x')$ for some $x, x' \in \Sigma^2$ such that $x[b_j] = v \neq x'[b_j] = v'$. This means that hk_j is not statistically binding on the index b_j, but this can only happen with negligible probability by the somewhere statistically binding property of the 2-to-1 SSB hash $\mathcal{H}$. Therefore $\mathcal{H}^*$ is somewhere statistically binding w.r.t. opening.

Lastly, the output length of $\mathcal{H}^*$ is given by $\ell^*(s, \lambda) = s_{q+1}$ where $s_1 = s$ and for each other $j \in [q]$, $s_{j+1} = \ell(s_j, \lambda)$. The output length of a SSB hash guarantees that $\ell(s_j, \lambda) = s_j(1 + 1/\Omega(\lambda)) + a(\lambda)$ where $a(\cdot)$ is some fixed polynomial. This ensures that

$$\ell^*(s, \lambda) = s(1 + 1/\Omega(\lambda))^q + a(\lambda) \sum_{j=0}^{q-1} (1 + 1/\Omega(\lambda))^j = O(s) + a(\lambda) O(\lambda)$$

is polynomial in s, λ. We rely on the fact that $q \leq \lambda$ to argue that $(1 + 1/\Omega(\lambda))^q \leq (1 + 1/\Omega(\lambda))^\lambda = O(1)$. ■

4 SSB Hash from Lossy Functions

In this section we describe a simple construction of an SSB Hash with local opening, the main tool we'll use are lossy functions, introduced by Peikert and Waters [PW08]. They actually introduced the stronger notion of lossy *trapdoor* functions, where a trapdoor allowed to invert functions with injective keys, we only need the lossiness property, but no trapdoors.

Definition 4.1. An (m, Λ)-lossy function is given by a tuple of PPT algorithms

- For $m, \Lambda \in \mathbb{N}$ and $\mathsf{mode} \in \{\mathsf{injective} = 1, \mathsf{lossy} = 0\}$, $\mathsf{Gen}_{\mathsf{LF}}(m, \Lambda, \mathsf{mode})$ outputs a key hk.
- Every such key hk defines a function $\mathsf{hk}(.) : \{0,1\}^m \to \{0,1\}^{m'}$ (for some $m' \geq m$).

We have the following three properties:

injective: If $\mathsf{hk} \leftarrow \mathsf{Gen}_{\mathsf{LF}}(m, \Lambda, \mathsf{injective})$, then $\mathsf{hk}(.)$ is injective.

lossy: If $\mathsf{hk} \leftarrow \mathsf{Gen}_{\mathsf{LF}}(m, \Lambda, \mathsf{lossy})$, then $\mathsf{hk}(.)$'s output domain has size $\leq 2^\Lambda$, i.e.

$$|\{y \ : \ \exists x \in \{0,1\}^m, \mathsf{hk}(x) = y\}| \leq 2^\Lambda$$

indistinguishable: Lossy and injective keys are computationally indistinguishable. More concretely, think of Λ as a security parameter and let $m = \mathsf{poly}(\Lambda)$, then the advantage of any PPT adversary in distinguishing
$\mathsf{Gen}_{\mathsf{LF}}(m, \Lambda, \mathsf{injective})$ from $\mathsf{Gen}_{\mathsf{LF}}(m, \Lambda, \mathsf{lossy})$ is negligible in Λ.

The Construction. Our construction $(\mathsf{Gen}^*, H^*, \mathsf{Open}, \mathsf{Verify})$ is illustrated in Figure 1, we define it formally below.

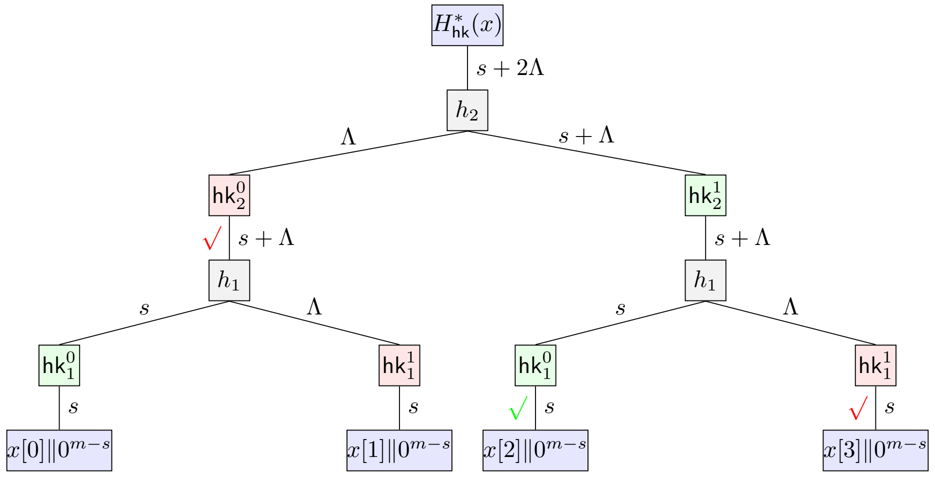

Figure 1: Illustration of the SSB hash from a lossy function with key $\mathsf{hk} \leftarrow \mathsf{Gen}^*(1^\lambda, 1^s, L = 2^q, i, \Lambda)$, i.e, $H_{\mathsf{hk}}(x)$ perfectly binds $x[i = 2]$. For every level $j \in \{1, \ldots, q\}$ we sample a pairwise independent function $h_j : \{0,1\}^{2m'} \to \{0,1\}^m$, where $m = 2(s + q\Lambda) + \lambda$ for a statistical security parameter λ, and two functions $\mathsf{hk}_j^0, \mathsf{hk}_j^1 : \{0,1\}^m \to \{0,1\}^{m'}$ from an (m, Λ)-lossy family of functions, one lossy and one injective (we decide which one of the two is the injective one such that the path from the perfectly binded value – here $x[2]$ – to the root only contains injective functions). The injective and lossy functions are shown in green and red, respectively. The SBB hash is now a Merkle-hash with the hash function $H_j(a, b) = h_j(\mathsf{hk}_j^0(a), \mathsf{hk}_j^1(b))$ used in level j. An edge label t in the figure means that there are at most 2^t possible values at this point, e.g., there are 2^s values of the form $x[0]\|0^{m-s}$ and the output of a lossy function like hk_1^1 has at most 2^Λ values. To locally open a value, say $x[2]$, we reveal $x[2]$ all the siblings of the nodes on the path from $x[2]$ to the root, those are marked with $\surd$ in the figure.

- $\mathsf{hk} \leftarrow \mathsf{Gen}^*(1^\lambda, 1^s, L = 2^q, i, \Lambda)$: Set $m = 2(s + q\Lambda) + \lambda$. For $i \in \{0, \ldots, 2^q - 1\}$, let $(b_q, \ldots, b_1)$ be the binary representation of i (with b_1 being the least significant bit).

  For every j: Choose $\mathsf{hk}_i^0 \leftarrow \mathsf{Gen}_{\mathsf{LF}}(m, \Lambda, 1 - b_j)$ and $\mathsf{hk}_i^1 \leftarrow \mathsf{Gen}_{\mathsf{LF}}(m, \Lambda, b_j)$. Sample a pairwise independent hash function $h_j : \{0,1\}^{2m'} \to \{0,1\}^m$ and let $\mathsf{hk}_j = (\mathsf{hk}_j^0, \mathsf{hk}_j^1, h_j)$. Each hk_j defines a mapping $H_j : \{0,1\}^{2m} \to \{0,1\}^m$ defined as

  $$H_j(a, b) = h_j(\mathsf{hk}_j^0(a), \mathsf{hk}_j^1(b))$$

  Output $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_q)$.

- $H^*_{\mathsf{hk}}(x)$: For $x = (x[0], \ldots, x[2^q - 1]) \in \{0,1\}^{s \cdot 2^q}$, $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_q)$ proceed as follows. Define T to be a complete binary tree of height q where level 0 of the tree denotes the leaves and level q denotes the root. We will assign a label to each vertex in the tree. The 2^q leaf vertices are assigned the labels $x[0]\|0^{m-s}, \ldots, x[2^q - 1]\|0^{m-s}$ (i.e., the input blocks padded to length m). The rest of the labels are assigned inductively where each non-leaf vertex v at level j of the tree with children that have labels x_0', x_1' gets assigned the label $y = H_j(x_0', x_1')$. The output $H^*_{\mathsf{hk}}(x)$ is the root of the tree.

- $\pi = \mathsf{Open}(\mathsf{hk}, x, j)$: Compute the labeled tree T as above. Output the labels of all the sibling nodes along the path from the root to the j'th leaf. Figure 1 the values to be opened to reveal $x[2]$ are marked with $\surd$.

- $\mathsf{Verify}(\mathsf{hk}, y, j, u, \pi)$: Recompute all of the labels of the nodes in the tree T that lie on the path from the root to the j'th leaf by using the value u for that leaf and the values given by π as the labels of all the sibling nodes along the path. Check that the recomputed label on the root of the tree is indeed y.

Theorem 4.1. The construction of a SSB Hash (with local opening) described below, which maps $L = 2^q$ blocks of length s bits to a hash of size $m = 2(s + q\Lambda) + \lambda$ bits where λ is a statistical security parameter and we assume (m, Λ)-lossy functions, is secure. More concretely, the somewhere statistically binding property holds with probability

$$1 - q/2^\lambda$$

over the choice of the hash key, and the index hiding property can be reduced to the indistinguishability property of the lossy function losing a factor q.

Proof. The index hiding property follows immediately from the indsitinguishability of injective and lossy modes.

To show that the hash is somewhere statistically binding, consider a key $\mathsf{hk} \leftarrow \mathsf{Gen}^*(1^\lambda, 1^s, L = 2^q, i, \Lambda)$. We must prove that with overwhelming probability no hash $y \in \{0,1\}^{2(s+q\cdot\Lambda)+\lambda}$ exists where $\mathsf{Verify}(\mathsf{hk}, y, i, u, \pi) = \mathsf{Verify}(\mathsf{hk}, y, i, u', \pi') = \mathsf{accept}$ for some $u \neq u'$, that is, $x[i]$ can be opened to u and u'.

In a nutshell, the reason why with high probability (over the choice of hk) the hash H_{hk} is perfectly binding on its ith coordinate is that the value $x[i]$ at the leaf of the tree only passes through two kinds of functions on its way to the root: injective functions and pairwise independent hashes. Clearly, no information can be lost when passing through an injective function. And every time the value passes through some hash h_j, the other half of the input is the output of a lossy function, and thus can take at most 2^Λ possible values. Thus even as we arrive at the root, there are only $2^{s+q\cdot\Lambda}$ possible values. We now set the output length $m = 2(s + q \cdot \Lambda) + \lambda$ of the h_j's so that 2^m is a larger – by a factor 2^λ – than the square of the possible values. This then suffices to argue that every h_j will be injective on its possible inputs (recall that there are at most $2^{s+q\cdot\Lambda}$ of them) with probability $\geq 1 - 2^{-\lambda}$.

For the formal proof it's convenient to consider the case $i = 0$ (i.e., the leftmost value should be perfectly binding). Let $\pi = (w_1, \ldots, w_q)$ and $\pi' = (w'_1, \ldots, w'_q)$ be two openings for values $x[0] \neq x'[0]$, we'll prove that with probability $q/2^\lambda$ (over the choice of hk) the verification procedure will compute different hashes corresponding to any two such openings (i.e., for every opening $(\pi, x[0])$, there's at most one y which makes $\mathsf{Verify}(\mathsf{hk}, y, i = 0, x[0], \pi)$ accept), and thus the hash is perfectly binding on index 0.

Let $v_0 = x[i]\|0^{m-s}$ and for $j = 1, \ldots, q$ define $v_j = h_j(\mathsf{hk}^0_j(v_{j-1}), \mathsf{hk}^1_j(w_j))$, the v'_j's are defined analogously for the other opening. Note that v_q is the final hash value, so we have to show that $v_q \neq v'_q$.

We will do so by induction, first, we claim that (for any hk) there are at most $2^{s+j\cdot\Lambda}$ possible values v_j can take. This is true for $j = 0$ as $v_0 = x[0]\|0^{m-s}$ can take exactly 2^s values by definition. Assume it holds for $j - 1$ and let $S_{j-1}, |S_{j-1}| \leq 2^{s+(j-1)\Lambda}$ denote the set of values v_{j-1} can take, then

$$\begin{aligned}
|S_j| &= \left|\left\{h_j(\mathsf{hk}^0_j(v_{j-1}), \mathsf{hk}^1_j(z)) \ : \ v_{j-1} \in S_{j-1}, z \in \{0,1\}^m)\right\}\right| && (1)\\
&\leq |\{(v_{j-1}, \mathsf{hk}^1_j(z)) \ : \ v_{j-1} \in S_{j-1}, z \in \{0,1\}^m))\}| && (2)\\
&\leq |S_{j-1}| \cdot 2^\Lambda && (3)\\
&\leq 2^{s+j\Lambda} && (4)
\end{aligned}$$

where the first step follows by definition of the set S_j, the second step follows as applying deterministic functions cannot increase the number of possible values, the third step follows as $\mathsf{hk}^1_j(.)$ is lossy and thus can take at most 2^Λ possible values. The last step follows by the induction hypothesis for $j - 1$.

For the proof we will think of the hash key $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_q)$, where $\mathsf{hk}_j = (\mathsf{hk}^0_j, \mathsf{hk}^1_j, h_j)$, as being lazy sampled. Initially, we sample all the $\mathsf{hk}^0_j, \mathsf{hk}^1_j$ keys. Let $L_j \subset \{0,1\}^m$ denote the range of the (lossy) $\mathsf{hk}^1_j(.)$ functions, note that $|L_j| \leq 2^\Lambda$ for all j. The h_j's will be sampled one by one in each induction step below.

Assume so far he have sampled $h_1, \ldots, h_{j-1}$, and so far for any openings where $x[0] \neq x'[0]$ we had $v_j \neq v'_j$. For $j = 0$ this holds as $x[0] \neq x'[0]$ implies $v_0 = x[0]\|0^{m-s} \neq x'[0]\|0^{m-s}$.

The inputs to the function h_j (which is still to be sampled) are from $I_{j-1} = h^0_j(S_{j-1}) \times L_{j-1}$, which (as shown above) contains at most $|S_{j-1}| \cdot |L_{j-1}| \leq 2^{s+(j-1)\Lambda}2^\Lambda = 2^{s+j\cdot\Lambda}$ elements.

We now sample the pairwise independent hash h_j, as it has range 2^m the probability that any two elements $(v,l) \neq (v',l') \in I_{j-1}$ collide[4] is 2^{-m}, taking the union bound over all pairs of elements we get

$$2^{2(s+j\cdot\Lambda)}/2^m \leq 2^{-\lambda}$$

Taking the union bound, we get that the probability that the induction fails for any of the q steps is $q/2^\lambda$ as claimed. ■

5 SSB from ϕ-hiding

We now move on to building SSB from the ϕ-hiding assumption [CMS99]. This construction will be qualitatively different from the prior ones in that we will not employ a Merkle tree type structure for proving and verifying opens. In contrast a hash output will consist of two elements $\mathbb{Z}^*_{N_0}$ and $\mathbb{Z}^*_{N_1}$ for RSA primes N_0, N_1. An opening will consist of a single element of either $\mathbb{Z}^*_{N_0}$ or $\mathbb{Z}^*_{N_1}$.

Our construction is structurally similar to standard constructions RSA accumulators [BdM93]. Intuitively, the initial hash key will consist of two RSA moduli N_0, N_1 as well as two group elements h_0, h_1 and keys K_0, K_1 which hash to prime exponents. To compute the hash on input $x \in \{0,1\}^L$ let $S_0 = \{i : x[i] = 0\}$ be the set of all indices where the i-th bit is 0 and $S_1 = \{i : x[i] = 1\}$ be the set of indices where the i-th bit is 1. The function computes the output

$$y_0 = h_0^{\prod_{i\in S_0} F_{K_0}(i)} \mod N_0, \quad y_1 = h_1^{\prod_{i\in S_1} F_{K_1}(i)} \mod N_1.$$

To prove that the j-th bit was 0 the open algorithm will give the $F_{K_0}(j)$-th root of y_0. It computes this by letting $S_0 = \{i : x[i] = 0\}$ and setting $\pi = h_0^{\prod_{i\neq j\in S_0} F_{K_0}(i)} \mod N_0$. A proof can be checked by simply checking if $y_0 \stackrel{?}{=} \pi^{F_{K_0}(j)} \mod N_0$. (Proving an opening of 1 follows analogously.)

The algorithms as described above very closely match a traditional RSA accumulator. The key distinction is that we can achieve statistical binding on index j by setting N_0 such that $K_0(j)$ divides $\phi(N_0)$ (and similarly for N_1). The idea is that in this setting if y_0 is not an $K_0(j)$-th residue then there will not exist a value π such that $y_0 \stackrel{?}{=} \pi^{F_{K_0}(j)} \mod N_0$. The index-hiding property will follow from the ϕ-hiding assumption.

5.1 RSA and ϕ-hiding Preliminaries

We begin by developing our notation and statement of the ϕ-hiding assumption both of which follow closely to Kiltz, O'Neill, and Smith [KOS10]. We let $\mathcal{P}_k$ denote the set of odd primes that are less than 2^k. In addition, we let $(N,p,q) \stackrel{\$}{\leftarrow} \mathcal{RSA}_k$ be the process of choosing two primes p,q uniformly from $\mathcal{P}_k$ and letting $N = pq$. Further we let $(N,p,q) \stackrel{\$}{\leftarrow} \mathcal{RSA}_k[p = 1 \mod e]$ be the be the process of choosing two primes p,q uniformly from $\mathcal{P}_k$ with the constraint that $p = 1 \mod e$, then letting $N = pq$.

We can now state the ϕ-hiding assumption relative to some constant $0 < c < .5$. Consider the following distributions relative to a security parameter λ.

$$\mathcal{R} = \{(e,N) : e,e' \stackrel{\$}{\leftarrow} \mathcal{P}_{c\lambda}; (N,p,q) \stackrel{\$}{\leftarrow} \mathcal{RSA}_\lambda[p = 1 \mod e']\}$$
$$\mathcal{L} = \{(e,N) : e \stackrel{\$}{\leftarrow} \mathcal{P}_{c\lambda}; (N,p,q) \stackrel{\$}{\leftarrow} \mathcal{RSA}_\lambda[p = 1 \mod e]\}$$

Cachin, Micali and Stadler [CMS99] show that the two distributions can be efficiently sampled if the Extended Riemann Hypothesis holds. The ϕ-hiding assumption states that for all $c \in (0,.5)$ no PPT attacker can distinguish between the two distributions with better than negligible in λ probability.

[4]Note that we prove something slightly stronger than required as we only need to consider pairs where $v \neq v'$.

5.2 Conforming Function

Before we give our construction we need one further abstraction. For any integer L we require the ability to sample a keyed hash function $F(K,\cdot)$ that hashes from an integer $i \in [0, L-1]$ to a random prime in $\mathcal{P}_{c\lambda}$. Furthermore, the function should have the property that it is possible to sample the key K in such a way that for a single pair $i^* \in [0, L-1]$ and $e^* \in \mathcal{P}_{c\lambda}$ $F(K, i^*) = e^*$. Moreover such programming should be undetectable if e^* is chosen at random from $\mathcal{P}_{c\lambda}$.

We give the definitions of such a function system here and show how to construct one in Appendix A. A conforming function system is parameterized by a constant $c \in (0, .5)$ and has three algorithms.

Sample-Normal$(1^\lambda, L) \to K$
Takes in a security parameter λ and a length L (in binary) and outputs a function key K.

Sample-Program$(1^\lambda, L, i^*, e^*) \to K$
Takes in a security parameter λ and a length L (in binary) as well as a program index $i^* \in [0, L-1]$ and $e^* \in \mathcal{P}_{c\lambda}$. It outputs a function key K.

$F_K : i \to \mathcal{P}_{c\lambda}$
If $\textsc{Sample-Normal}(1^\lambda, L) \to K$, then F_K takes in an index $i \in [0, L-1]$ and outputs a prime from $\mathcal{P}_{c\lambda}$.

5.2.1 Properties

Such a system will have four properties:

Efficiency The programs $\textsc{Sample-Normal}$ and $\textsc{Sample-Program}$ run in time polynomial in λ and L. Let $\textsc{Sample-Normal}(1^\lambda, L) \to K$, then F_K runs in time polynomial in λ and $\lg(L)$.

Programming at i^* For some λ, L, i^*, e^* let $\textsc{Sample-Program}(1^\lambda, L, i^*, e^*) \to K$. Then $F_K(i^*) = e^*$ with all but negligible probability in λ.

Non colliding at i^* For some λ, L, i^*, e^* let $\textsc{Sample-Program}(1^\lambda, L, i^*, e^*) \to K$. Then for any $i \neq i^*$ the probability that $F_K(i^*) = F_K(i)$ is negligible in λ.

Indistinguishability of Setup For any L, i^* consider the following two distributions:

$$\mathcal{R}_{L,i^*} = \{K : e^* \xleftarrow{\$} \mathcal{P}_{c\lambda}; \textsc{Sample-Normal}(1^\lambda, L) \to K\}$$
$$\mathcal{L}_{L,i^*} = \{K : e^* \xleftarrow{\$} \mathcal{P}_{c\lambda}; \textsc{Sample-Program}(1^\lambda, L, i^*, e^*) \to K\}$$

The indistinguishability of setup property states that all PPT adversaries have a most a negligible advantage in distinguishing between the two distributions for all L, i^*.

5.3 Our ϕ-hiding SSB construction

We now present our ϕ-hiding based SSB construction. Our construction is for an alphabet of a single bit, thus s is implicitly 1 and omitted from our notation. In addition, the construction is parameterized relative to some constant $c \in (0, .5)$.

$\mathsf{Gen}(1^\lambda, L, i^*)$
The generation algorithm first samples two random primes $e_0, e_1 \xleftarrow{\$} \mathcal{P}_{c\lambda}$. Next, it sets up two conforming functions as $\textsc{Sample-Program}$ $(1^\lambda, L, i^*, e_0) \to K_0$ and $\textsc{Sample-Program}$ $(1^\lambda, L, i^*, e_1) \to K_1$. Then it samples $(N_0, p_0, q_0) \xleftarrow{\$} \mathcal{RSA}_k[p_0 = 1 \mod e_0]$ and $(N_1, p_1, q_1) \xleftarrow{\$} \mathcal{RSA}_k[p_1 = 1 \mod e_1]$. Finally, it

chooses $h_0 \in Z^*_{N_0}$ randomly with the constraint that $h_0^{(p_0-1)/e_0} \neq 1 \mod p_0$ and $h_1 \in Z^*_{N_1}$ randomly with the constraint that $h_1^{(p_1-1)/e_1} \neq 1 \mod p_1$.

It outputs the hash key as $\mathsf{hk} = \{L, (N_0, N_1), (K_0, K_1), (h_0, h_1)\}$.

$H_{\mathsf{hk}} : \{0,1\}^L \to \mathbb{Z}^*_{N_0}, \mathbb{Z}^*_{N_1}$**:**
On input $x \in \{0,1\}^L$ let $S_0 = \{i : x[i] = 0\}$ be the set of all indices where the i-th bit is 0 and $S_1 = \{i : x[i] = 1\}$ be the set of indices where the i-th bit is 1. The function computes

$$y_0 = h_0^{\prod_{i \in S_0} F_{K_0}(i)} \mod N_0, \quad y_1 = h_1^{\prod_{i \in S_1} F_{K_1}(i)} \mod N_1.$$

The hash output is $y = (y_0, y_1)$.

We note that the computation in practice will be done by iteratively with repeated exponentiation as opposed to computing the large integer $\prod_{i \in S_0} F_{K_0}(i)$ up front.

$\mathsf{Open}(\mathsf{hk}, x, j)$**:**
If $x_j = 0$ it first lets $S_0 = \{i : x[i] = 0\}$. Then it computes

$$\pi = h_0^{\prod_{\mathbf{i} \neq \mathbf{j} \in S_0} F_{K_0}(i)} \mod N_0.$$

Otherwise, if $x_j = 1$ it first lets $S_1 = \{i : x[i] = 1\}$. Then it computes

$$\pi = h_1^{\prod_{\mathbf{i} \neq \mathbf{j} \in S_1} F_{K_1}(i)} \mod N_1.$$

$\mathsf{Verify}(\mathsf{hk}, y = (y_0, y_1), j, b \in \{0,1\}, \pi)$**:**

The verify algorithm checks
$$y_b \stackrel{?}{=} \pi^{F_{K_b}(j)} \mod N_b.$$

5.3.1 Properties

We now show that the above construction meets the required properties for SSB with local opening. One minor difference from the original definition is that we weaken the statistically binding requirement. Previously, we wanted the binding property to hold for any hash digest y, even one which does not correspond to a correctly generated hash output. In the version we achieve here, we require that $y = H_{\mathsf{hk}}(x)$ for some x. We define the property formally below.

Weak Somewhere Statistically Binding w.r.t. Opening: We say that hk is *weak statistically binding w.r.t opening (abbreviates wSBO) for an* index i if there do not exist any values $x \in \Sigma^L, u' \neq x[i], \pi'$ s.t. $\mathsf{Verify}(\mathsf{hk}, H_{\mathsf{hk}}(x), i, u', \pi) = \mathsf{accept}$. We require that for any parameters s, L and any index $i \in \{0, \ldots, L-1\}$

$$\Pr[\mathsf{hk} \text{ is wSBO for index } i \ : \ \mathsf{hk} \leftarrow \mathsf{Gen}(1^\lambda, 1^s, L, i)] \geq 1 - \text{negl}(\lambda).$$

We say that the hash is *perfectly binding w.r.t. opening* if the above probability is 1.

Correctness of Opening Consider any hk generated from the setup algorithm and let π be the output from a call to $\mathsf{Open}(\mathsf{hk}, x, j)$ for some x, j where that $x[j] = b \in \{0,1\}$. Then $y_b = h_b^{\prod_{i \in S_b} F_{K_b}(i)} \mod N_b$ and $\pi = h_b^{\prod_{\mathbf{i} \neq \mathbf{j} \in S_b} F_{K_b}(i)} \mod N_b$. It follows that $\pi^{F_{K_b}(j)} = y_b \mod N_b$.

Weak Somewhere Statistically Binding with Respect to Opening Suppose that $\mathsf{Gen}(1^\lambda, L, i^*) \to \mathsf{hk}$. We argue that with all but negligible probability for all inputs $x \in \{0,1\}^L$ that the function is statistically binding with respect to opening.

Consider a particular input x where $x[i^*] = 1 - b$ and $H_{\mathsf{hk}}(x) = y = (y_0, y_1)$. We want to show that there does not exist a value π such that $\mathsf{Verify}(\mathsf{hk}, y = (y_0, y_1), i^*, b \in \{0,1\}, \pi) = 1$. Let $e_b \in \mathcal{P}_{c\lambda}$ be the prime value chosen at hash function setup. By the setup process we have that $e_b | p_b - 1$ and that of $e_b = F_{K_b}(i^*)$. The latter follows from the Programming at i^* property of the conforming hash function. Therefore we have that $(\pi^{e_b})^{(p_b-1)/e_b} = 1 \mod p_1$ (i.e. π^{e_b} is an e_b-th residue mod p_b).

Recall that $y_b = h_b^{\prod_{i \in S_b} F_{K_b}(i)} \mod N_b$. Let $\alpha = \prod_{i \in S_b} F_{K_b}(i)$. By the non-colliding property of F coupled with the fact that $x[i^*] \neq b$ and thus $i^* \notin S_b$ with all but negligible probability for all $i \in S_b$ we have that $F_{K_b}(i)$ is a prime $\neq e_b$. Therefore α is relatively prime to e_b. Since h_b was chosen to not be a e_b-th residue mod p_b and α is relatively prime to e_b it follows that $y_b = h_b^\alpha$ is also not an e_b-th residue mod p_b. However, since π^{e_b} is an e_b-th residue mod p_b, it cannot be equal to y_b and the verification test will fail.

Index Hiding We sketch a simple proof of index hiding via a sequence of games. We begin by defining the sequence.

- Game_0: The Index Hiding game on our construction.
- Game_1: Same as Game_0 except that an additional prime $e'_0 \xleftarrow{\$} \mathcal{P}_{c\lambda}$ is sampled and $(N_0, p_0, q_0) \xleftarrow{\$} \mathcal{RSA}_\lambda[p = 1 \mod e'_0]$. Note that we still sample $(1^\lambda, L, i_b, e_0) \to K_0$ where w.h.p $e_0 \neq e'_0$.
- Game_2: Same as Game_1 except that an additional prime $e'_1 \xleftarrow{\$} \mathcal{P}_{c\lambda}$ is sampled and $(N_1, p_1, q_1) \xleftarrow{\$} \mathcal{RSA}_\lambda[p = 1 \mod e'_1]$. Note that we still sample $(1^\lambda, L, i_b, e_1) \to K_1$ where w.h.p $e_1 \neq e'_1$.
- Game_3 Same as Game_2 except that K_0 is sampled as $\textsc{Sample-Normal}(1^\lambda, L) \to K_0$.
- Game_4 Same as Game_3 except that K_1 is sampled as $\textsc{Sample-Normal}(1^\lambda, L) \to K_1$.

It follows directly from the ϕ-hiding assumption that no PPT attacker can distinguish between Game_1 and Game_1 and that no attacker can distinguish between Game_1 and Game_2. At this point the primes e_0 and e_1 are used only in the programming of the hash function and are not reflected in the choice of the RSA moduli. For this reason we can now use the Indistinguishability of Setup property of the conforming has to show that no PPT attack can distinguish between Game_2 and Game_3 and Game_3 and Game_4. Finally, we observe that the index i_b is not used at Game_4 and thus the bit b is hidden from the attacker's view.

6 Positional Accumulators

In Appendix B, we also discuss how to extend some the above results to positional accumulators. In particular, we show how to construct positional accumulators from a (perfectly binding) two-to-one SSB hash. The construction can also be naturally extended to one based on lossy functions.

References

[BdM93] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*, pages 274–285, 1993.

[BGI+12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[BHK11] Mark Braverman, Avinatan Hassidim, and Yael Tauman Kalai. Leaky pseudo-entropy functions. In *Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 7-9, 2011. Proceedings*, pages 353–366, 2011.

[CCC$^+$15] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Computation-trace indistinguishability obfuscation and its applications. Cryptology ePrint Archive, Report 2015/406, 2015. `http://eprint.iacr.org/`.

[CH15] Ran Canetti and Justin Holmgren. Fully succinct garbled ram. Cryptology ePrint Archive, Report 2015/388, 2015. `http://eprint.iacr.org/`.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 402–414, 1999.

[DJ01] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*, pages 119–136, 2001.

[GGH$^+$13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, 2013.

[HW15] Pavel Hubacek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 163–172, 2015.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *STOC*, 2015.

[KOS10] Eike Kiltz, Adam O'Neill, and Adam Smith. Instantiability of RSA-OAEP under chosen-plaintext attack. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 295–313, 2010.

[Lip05] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao, editors, *Information Security, 8th International Conference, ISC 2005, Singapore, September 20-23, 2005, Proceedings*, volume 3650 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2005.

[NS12] Moni Naor and Gil Segev. Public-key cryptosystems resilient to key leakage. *SIAM J. Comput.*, 41(4):772–814, 2012.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 223–238, 1999.

[PW08] Chris Peikert and Brent Waters. Lossy trapdoor functions and their applications. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 187–196, 2008.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC*, pages 475–484, 2014.

[Zha14] Mark Zhandry. Adaptively secure broadcast encryption with small system parameters. *IACR Cryptology ePrint Archive*, 2014:757, 2014.

A Constructing a Conforming Function

We now give our construction of a conforming hash function per the definition given in Section 5.2.

Recall, our goal is to construct a keyed hash function $F(K, \cdot)$ that hashes from an integer $i \in [0, L-1]$ to a prime in $\mathcal{P}_{c\lambda}$. Furthermore, the function should have the property that it is possible to sample the key K in such a way that for a single pair $i^* \in [0, L-1]$ and $e^* \in \mathcal{P}_{c\lambda}$ we have $F(K, i^*) = e^*$. (The constant $c \in (0, .5)$ is considered a parameter of the system.) Moreover, such programming should be undetectable if e^* is sampled at random from $\mathcal{P}_{c\lambda}$.

Our construction below is a simple implementation of this abstraction and all properties are statistically guaranteed (i.e. we do not require any computational assumptions).

Sample-Normal$(1^\lambda, L) \to K$
We first let $B = 2^{\lfloor c\lambda \rfloor}$ and let $T = \lambda^2$. The algorithm chooses random $w_1, \ldots, w_T \in [0, B-1]$. The key K is set as $K = (\lambda, w_1, \ldots, w_T)$.

Sample-Program$(1^\lambda, L, i^*, e^*) \to K$:
We first let $B = 2^{\lfloor c\lambda \rfloor}$ and let $T = \lambda^2$. Initialize a bit (local to this computation) PROGRAMMED to be 0. Then proceed in the following manner:

For $j = 1$ to T if PROGRAMMED $\overset{?}{=} 1$ choose v_j randomly in $[0, B-1]$ and set $w_j = v_j - i^* \mod B$. This corresponds to the case where the value e^* was "already programmed". Else, if PROGRAMMED $\overset{?}{=} 0$, it first chooses v_j randomly in $[0, B-1]$. If v_j is not prime it simply sets $w_j = v_j - i^* \mod B$. Otherwise, it sets $w_j = e^* - i^* \mod B$ and flips the bit PROGRAMMED to 1 so that e^* will not be programmed in again.

The key K is output as $K = (\lambda, w_1, \ldots, w_T)$.

$F_K : i \to \mathcal{P}_{c\lambda}$
The function proceeds as follows. Starting at $j = 1$ to T the function tests if $w_j + i$ is a prime (i.e. is in $\mathcal{P}_{c\lambda}$). If so it outputs $w_j + i$ and halts. Otherwise, it increments j and tests again. If j goes past T and no primes have been found, the algorithm outputs a default prime $3 \in \mathcal{P}_{c\lambda}$. [5]

A.0.2 Properties

We now confirm that our function meets all the required properties.

Efficiency The programs SAMPLE-NORMAL chooses T random values where T is polynomial in λ and SAMPLE-PROGRAM also chooses T random values as well as performing up to T primality tests. The keysizes of both are T integers in $[0, B]$. Thus the running times and keysizes are polynomial in λ and $\lg(L)$.

Programming at i^* Consider a call to SAMPLE-PROGRAM$(1^\lambda, L, i^*, e^*) \to K$. The function $F_K(i^*)$ will resolve to the smallest j such that $w_j + i^*$ is a prime (if any of these are a prime). By the design of SAMPLE-PROGRAM this will be e^* since it puts in $w_j = e^* - i^* \mod B$ the first time a prime is sampled. In constructing the function if all v_j sampled were composite then $F_K(i^*) \neq e^*$, however, this will only occur with negligible probability since the probability of choosing T random integers in $2^{\lfloor c\lambda \rfloor}$ and none of them being prime is negligible.

Non colliding at i^* For some λ, L, i^*, e^* let SAMPLE-PROGRAM$(1^\lambda, L, i^*, e^*) \to K$. Let's assume that $F_K(i^*) = e^*$. We first observe that the chances that there exist any pairs $(i_0, j_0) \neq (i_1, j_1)$ such that $w_{j_0} + i_0 = w_{j_1} + i_1$ is negligible. We consider the probability of this happening on an arbitrary pair and them apply the union bound.

[5] Note there is nothing special about choosing 3. Any default prime would suffice.

Consider a pair $(i_0, j_0) \neq (i_1, j_1)$ If $j_0 = j_1$ this cannot happen since the two terms differ by $i_1 - i_0$. Otherwise, we notice that the probability of a particular pair colliding is at most $1/B$ (which is negligible) since v_{j_0} and v_{j_1} are chosen independently at random. Since there are at most a polynomial $\binom{T \cdot L}{2}$ such pairs the chances that any collide is negligible.

It follows that the chances of $F_K(i^*) = F_K(i)$ for $i = i^*$ is negligible since the above condition would be necessary for this to occur.

Indistinguishability of Setup For any L, i^* consider the following two distributions:

$$\mathcal{R}_{L,i^*} = \{K : e^* \xleftarrow{\$} \mathcal{P}_{c\lambda}; \textsc{Sample-Normal}(1^\lambda, L) \to K\}$$
$$\mathcal{L}_{L,i^*} = \{K : e^* \xleftarrow{\$} \mathcal{P}_{c\lambda}; \textsc{Sample-Program}(1^\lambda, L, i^*, e^*) \to K\}$$

We argue that these two distributions are identical for all L, i^*. We show this by also considering an intermediate distribution $\mathcal{I}_{L,i^*}$. This distribution is generated by randomly sampling v_j in $[0, B-1]$ and setting $w_j = v_j - i^* \mod B$. This distribution is clearly equivalent to the Sample-Normal distribution as for all j selecting w_j randomly and selecting v_j randomly and setting $w_j = v_j - i^* \mod B$ both result in w_j being chosen uniformly at random.

We now argue that this intermediate distribution is equivalent to the $\mathcal{L}_{L,i^*}$ distribution which is equivalent to calling Sample-Program with sampling e^* randomly from $\mathcal{P}_{c\lambda}$. We will step through an execution of Sample-Program and argue that at each step j from $j = 1, \ldots, T$ v_j is chosen randomly from $[0, B-1]$ independently of all other $v_{j'}$ for $j' < j$.

Consider an execution starting with $j = 1$ and programmed $= 0$ and for our exposition let's consider that $e^* \in \mathcal{P}_{c\lambda}$ has not been sampled yet. While programmed $\stackrel{?}{=} 0$ the algorithm samples v_j is sampled at random. If v_j is composite it is kept and put in the key, otherwise if it is prime in $\mathcal{P}_{c\lambda}$, v_j is replaced with e^* as another randomly sampled prime. Thus, for any composite value x the probability that $w_j + i^* = x$ is $1/B$ and for any prime value x the probability that $w_j + i^* = x$ is also $1/B$. The reason is that replacing any sampled prime with a different randomly sampled prime does not change the distribution.

After programmed is set to 1 all further v_j values are chosen uniformly at random.

Remark A.1. We note that the Indistinguishability of Setup property holds perfectly while the programmability property holds statistically. One way to flip this is to always program $v_T = e^*$ at the end if if e^* has not been programmed in already.

B Positional Accumulators

We provide the definition of *positional accumulators* which is taken from [KLW15].

Intuitively, a positional accumulator will be a cryptographic data structure that maintains two values: a storage value and an accumulator value. The storage value will be allowed to grow comparatively large, while the accumulator value will be constrained to be short. Messages can be written to various positions in the the underlying storage, and new accumulated values can be computed as a stream, knowing only the previous accumulator value and the newly written message and its position in the data structure. Since the accumulator values are small, one cannot hope to recover everything written in the storage from the accumulator value alone. However, we define "helper" algorithms that essentially allow a party who is maintaining the full storage to help a more restricted party who is only maintaining the accumulator values recover the data currently written at an arbitrary location. The helper is not necessarily trusted, so the party maintaining the accumulator values performs a verification procedure in order to be convinced that they are indeed reading the correct messages.

In a positional accumulator as defined in [KLW15] the memory will be perfectly binding in one position. The actual definition in [KLW15] gives this property in a somewhat weak sense the accumulator is only required to be binding at INDEX^* in the case where the accumulator value is computed by inserting a particular set of messages at a particular set of indices $(m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k), \text{INDEX}^*)$. This sequence of messages is given as input to the setup algorithms. The rationale of [KLW15] for this weaker definition of binding is that it possibly opens the door for constructions that would not meet stronger definitions where the sequence of insertion was not given to the setup algorithm. A drawback of the weaker definition is that it is notationally somewhat more complicated.

As it turns out, our construction given in Appendix meets a stronger definition of security so the setup algorithms ignore the insertion pattern given and focus only on the index needing for binding.

A positional accumulator for message space $\mathcal{M}_\lambda$ consists of the following algorithms.

$\mathsf{Setup\text{-}Acc}(1^\lambda, T) \to \text{PP}, w_0, \mathbf{store}_0$ The setup algorithm takes as input a security parameter λ in unary and an integer T in binary representing the maximum number of values that can stored. It outputs public parameters PP, an initial accumulator value w_0, and an initial storage value STORE_0.

$\mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Read}(1^\lambda, T, (m_1, \mathbf{index}_1), \ldots, (m_k, \mathbf{index}_k), \mathbf{index}^*) \to \text{PP}, w_0, \mathbf{store}_0$ The setup enforce read algorithm takes as input a security parameter λ in unary, an integer T in binary representing the maximum number of values that can be stored, and a sequence of symbol, index pairs, where each index is between 0 and $T-1$, and an additional INDEX^* also between 0 and $T-1$. It outputs public parameters PP, an initial accumulator value w_0, and an initial storage value STORE_0.

$\mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Write}(1^\lambda, T, (m_1, \mathbf{index}_1), \ldots, (m_k, \mathbf{index}_k)) \to \text{PP}, w_0, \mathbf{store}_0$ The setup enforce write algorithm takes as input a security parameter λ in unary, an integer T in binary representing the maximum number of values that can be stored, and a sequence of symbol, index pairs, where each index is between 0 and $T-1$. It outputs public parameters PP, an initial accumulator value w_0, and an initial storage value STORE_0.

$\mathsf{Prep\text{-}Read}(\text{PP}, \mathbf{store}_{in}, \mathbf{index}) \to m, \pi$ The prep-read algorithm takes as input the public parameters PP, a storage value STORE_{in}, and an index between 0 and $T-1$. It outputs a symbol m (that can be $\perp$) and a value π.

$\mathsf{Prep\text{-}Write}(\text{PP}, \mathbf{store}_{in}, \mathbf{index}) \to aux$ The prep-write algorithm takes as input the public parameters PP, a storage value STORE_{in}, and an index between 0 and $T-1$. It outputs an auxiliary value aux.

$\mathsf{Verify\text{-}Read}(\text{PP}, w_{in}, m_{read}, \mathbf{index}, \pi) \to \{True, False\}$ The verify-read algorithm takes as input the public parameters PP, an accumulator value w_{in}, a symbol, m_{read}, an index between 0 and $T-1$, and a value π. It outputs $True$ or $False$.

$\mathsf{Write\text{-}Store}(\text{PP}, \mathbf{store}_{in}, \mathbf{index}, m) \to \mathbf{store}_{out}$ The write-store algorithm takes in the public parameters, a storage value STORE_{in}, an index between 0 and $T-1$, and a symbol m. It outputs a storage value STORE_{out}.

$\mathsf{Update}(\text{PP}, w_{in}, m_{write}, \mathbf{index}, aux) \to w_{out}$ **or** $Reject$ The update algorithm takes in the public parameters PP, an accumulator value w_{in}, a symbol m_{write}, and index between 0 and $T-1$, and an auxiliary value aux. It outputs an accumulator value w_{out} or $Reject$.

In general we will think of the $\mathsf{Setup\text{-}Acc}$ algorithm as being randomized and the other algorithms as being deterministic. However, one could consider non-deterministic variants.

Correctness We consider any sequence $(m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k)$ of symbols $m_1, \ldots, m_k$ and indices $\text{INDEX}_1, \ldots, \text{INDEX}_k$ each between 0 and $T-1$. We fix any $\text{PP}, w_0, \text{STORE}_0 \leftarrow \mathsf{Setup\text{-}Acc}(1^\lambda, T)$. For j from 1 to k, we define STORE_j iteratively as $\text{STORE}_j := \mathsf{Write\text{-}Store}(\text{PP}, \text{STORE}_{j-1}, \text{INDEX}_j, m_j)$. We similarly define aux_j and w_j iteratively as $aux_j := \mathsf{Prep\text{-}Write}(\text{PP}, \text{STORE}_{j-1}, \text{INDEX}_j)$ and $w_j := Update(\text{PP}, w_{j-1}, m_j, \text{INDEX}_j, aux_j)$. Note that the algorithms other than $\mathsf{Setup\text{-}Acc}$ are deterministic, so these definitions fix precise values, not random values (conditioned on the fixed starting values $\text{PP}, w_0, \text{STORE}_0$).

We require the following correctness properties:

1. For every INDEX between 0 and $T-1$, $\mathsf{Prep\text{-}Read}(\mathrm{PP}, \textsc{store}_k, \textsc{index})$ returns m_i, π, where i is the largest value in $[k]$ such that $\textsc{index}_i = \textsc{index}$. If no such value exists, then $m_i = \perp$.

2. For any INDEX, let $(m, \pi) \leftarrow \mathsf{Prep\text{-}Read}(\mathrm{PP}, \textsc{store}_k, \textsc{index})$. Then $\mathsf{Verify\text{-}Read}(\mathrm{PP}, w_k, m, \textsc{index}, \pi) = True$.

Remarks on Efficiency In our construction, all algorithms will run in time polynomial in their input sizes. More precisely, Setup-Acc will be polynomial in λ and $\log(T)$. Also, accumulator and π values should have size polynomial in λ and $\log(T)$, so Verify-Read and Update will also run in time polynomial in λ and $\log(T)$. Storage values will have size polynomial in the number of values stored so far. Write-Store, Prep-Read, and Prep-Write will run in time polynomial in λ and $\log(T)$.

Security Let Acc = (Setup-Acc, Setup-Acc-Enforce-Read, Setup-Acc-Enforce-Write, Prep-Read, Prep-Write, Verify-Read, Write-Store, Update) be a positional accumulator for symbol set $\mathcal{M}$. We require Acc to satisfy the following notions of security.

Definition B.1 (Indistinguishability of Read Setup)**.** A positional accumulator Acc is said to satisfy indistinguishability of read setup if any PPT adversary $\mathcal{A}$'s advantage in the security game $\mathsf{Exp\text{-}Setup\text{-}Acc}(1^\lambda, \mathsf{Acc}, \mathcal{A})$ is at most negligible in λ, where Exp-Setup-Acc is defined as follows.

$\mathsf{Exp\text{-}Setup\text{-}Acc}(1^\lambda, \mathsf{Acc}, \mathcal{A})$

noitemsep Adversary chooses a bound $T \in \Theta(2^\lambda)$ and sends it to challenger.

noiitemsep $\mathcal{A}$ sends k messages $m_1, \ldots, m_k \in \mathcal{M}$ and k indices $\textsc{index}_1, \ldots,$ $indexA_k \in \{0, \ldots, T-1\}$ to the challenger.

noiiitemsep The challenger chooses a bit b. If $b = 0$, the challenger outputs $(\mathrm{PP}, w_0, \textsc{store}_0) \leftarrow \mathsf{Setup\text{-}Acc}(1^\lambda, T)$. Else, it outputs $(\mathrm{PP}, w_0, \textsc{store}_0) \leftarrow \mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Read}(1^\lambda, T, (m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k))$.

noivtemsep $\mathcal{A}$ sends a bit b'.

$\mathcal{A}$ wins the security game if $b = b'$.

Definition B.2 (Indistinguishability of Write Setup)**.** A positional accumulator Acc is said to satisfy indistinguishability of write setup if any PPT adversary $\mathcal{A}$'s advantage in the security game $\mathsf{Exp\text{-}Setup\text{-}Acc}(1^\lambda, \mathsf{Acc}, \mathcal{A})$ is at most negligible in λ, where Exp-Setup-Acc is defined as follows.

$\mathsf{Exp\text{-}Setup\text{-}Acc}(1^\lambda, \mathsf{Acc}, \mathcal{A})$

noitemsep Adversary chooses a bound $T \in \Theta(2^\lambda)$ and sends it to challenger.

noiitemsep $\mathcal{A}$ sends k messages $m_1, \ldots, m_k \in \mathcal{M}$ and k indices $\textsc{index}_1, \ldots, \textsc{index}_k \in \{0, \ldots, T-1\}$ to the challenger.

noiiitemsep The challenger chooses a bit b. If $b = 0$, the challenger outputs $(\mathrm{PP}, w_0, \textsc{store}_0) \leftarrow \mathsf{Setup\text{-}Acc}(1^\lambda, T)$. Else, it outputs $(\mathrm{PP}, w_0, \textsc{store}_0) \leftarrow \mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Write}(1^\lambda, T, (m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k))$.

noivtemsep $\mathcal{A}$ sends a bit b'.

$\mathcal{A}$ wins the security game if $b = b'$.

Definition B.3 (Read Enforcing)**.** Consider any $\lambda \in \mathbb{N}$, $T \in \Theta(2^\lambda)$, $m_1, \ldots, m_k \in \mathcal{M}$, $\textsc{index}_1, \ldots, \textsc{index}_k \in \{0, \ldots, T-1\}$ and any $\textsc{index}^* \in \{0, \ldots, T-1\}$.

Let $(\mathrm{PP}, w_0, \mathsf{st}_0) \leftarrow \mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Read}(1^\lambda, T, (m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k), \textsc{index}^*)$. For j from 1 to k, we define $\textsc{store}_j$ iteratively as $\textsc{store}_j := \mathsf{Write\text{-}Store}(\mathrm{PP}, \textsc{store}_{j-1}, \textsc{index}_j, m_j)$. We similarly define

aux_j and w_j iteratively as $aux_j := \mathsf{Prep\text{-}Write}(\mathrm{PP}, \textsc{store}_{j-1}, \textsc{index}_j)$ and $w_j := \mathit{Update}(\mathrm{PP}, w_{j-1}, m_j, \textsc{index}_j, aux_j)$. Acc is said to be *read enforcing* if $\mathsf{Verify\text{-}Read}(\mathrm{PP}, w_k, m, \textsc{index}^*, \pi) = \mathit{True}$, then either $\textsc{index}^* \notin \{\textsc{index}_1, \ldots, \textsc{index}_k\}$ and $m = \perp$, or $m = m_i$ for the largest $i \in [k]$ such that $\textsc{index}_i = \textsc{index}^*$. Note that this is an information-theoretic property: we are requiring that for all other symbols m, values of π that would cause $\mathsf{Verify\text{-}Read}$ to output True at $\textsc{index}^*$ do no exist.

Definition B.4 (Write Enforcing)**.** Consider any $\lambda \in \mathbb{N}, T \in \Theta(2^\lambda)$, $m_1, \ldots, m_k \in \mathcal{M}$, $\textsc{index}_1, \ldots, \textsc{index}_k \in \{0, \ldots, T-1\}$. Let
$(\mathrm{PP}, w_0, \mathsf{st}_0) \leftarrow \mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Write}(1^\lambda, T, (m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k))$. For j from 1 to k, we define $\textsc{store}_j$ iteratively as $\textsc{store}_j := \mathsf{Write\text{-}Store}(\mathrm{PP}, \textsc{store}_{j-1}, \textsc{index}_j, m_j)$. We similarly define aux_j and w_j iteratively as $aux_j := \mathsf{Prep\text{-}Write}(\mathrm{PP}, \textsc{store}_{j-1}, \textsc{index}_j)$ and $w_j := \mathit{Update}(\mathrm{PP}, w_{j-1}, m_j, \textsc{index}_j, aux_j)$. Acc is said to be *write enforcing* if $\mathsf{Update}(\mathrm{PP}, w_{k-1}, m_k, \textsc{index}_k, aux) = w_{out} \neq \mathit{Reject}$, for any aux, then $w_{out} = w_k$. Note that this is an information-theoretic property: we are requiring that an aux value producing an accumulated value other than w_k or Reject deos not exist.

B.1 Constructing Positional Accumulators

We now show how to construct positional accumulators from a (perfectly binding) two-to-one SSB hash. The construction can also be naturally extended to one based on lossy functions.[6] Intuitively, the construction is similar to that of SSB hashing from two-to-one SSB and follows the Merkle tree approach. The public parameters PP will be like the hashing key hk of the SSB hash and the accumulator value w will be the output of a SSB hash. The $\textsc{store}$ component consists of all the labels in the tree. This allows us to update the output of the SSB hash (the label associated with the root of the tree) after modifying a single data position by updating the labels on the path from the root to that position. One other difference is that now the tree has T leaves where T is exponential, but only a small polynomial number of them contain any actual data (and the rest are empty). To handle this, we prune out empty subtrees; any node in the tree that does not have any data in the subtree under it gets assigned a special label $\perp$.

Construction. Assume that $\mathcal{H} = (\mathsf{Gen}, H)$ is a two-to-one SSB hash family with output length give by $\ell(s, \lambda)$. We construct a positional accumulator scheme as follows.

$\mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Read}(1^\lambda, T, (m_1, \mathbf{index}_1), \ldots, (m_k, \mathbf{index}_k), \mathbf{index}^*) \to \mathrm{PP}, w_0, \mathbf{store}_0$**.** Let $(b_q, \ldots, b_1)$ be the binary representation of $\textsc{index}^*$ (with b_1 being the least significant bit) where $q = \lceil \log T \rceil$. For $j \in [q]$ define the block-lengths $s_1, \ldots, s_{q+1}$ where $s_1 = s + 1$ and $s_{j+1} = \ell(s_j, \lambda) + 1$. Choose $\mathsf{hk}_j \leftarrow \mathsf{Gen}(1^\lambda, 1^{s_j}, b_j)$ and output $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_q)$.

Define $\textsc{store}_0$ to be an complete binary tree of height q (where level 0 of the tree denotes the leaves and level q denotes the root) where each node in the tree at level j is associated with some label $\mathsf{lbl} \in \{0,1\}^{s_{j+1}}$. At each level j, we denote the special value $\mathsf{lbl} = 0^{s_{j+1}}$ by $\perp$. Although the tree has 2^q nodes, which may be exponential, there will only be a polynomial number of nodes whose label is not $\perp$ and therefore, by only storing these nodes, we get a polynomial sized representation of $\textsc{store}_0$.

We define w_0 to be the label associated with the root of the tree and set it to initially be $\perp$ to indicate that the memory store starts out empty.

We note that the values $(m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k)$ are not used by our setup algorithm since our setup algorithm is able to provide a stronger property than required of being binding on $\textsc{index}^*$ no matter what sequence of messages are inserted into the memory.

$\mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Write}(1^\lambda, T, (m_1, \mathbf{index}_1), \ldots, (m_k, \mathbf{index}_k))$**.** Run
$\mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Read}(1^\lambda, T, 0)$ with $k = 0$ and $\textsc{index}^* = \textsc{index}_k$. Again, we note that the values $(m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k)$ are not used by our setup algorithm for the same reason as above.

[6] We note that the security definition of positional accumulators given in [KLW15] requires the enforcing algorithms to be perfectly binding, however, we could also consider a statistically binding variant which would be sufficient for the applications of [KLW15]. The lossy construction would be sufficient for this.

$\mathsf{Setup\text{-}Acc}(1^\lambda, T)$**.** Run $\mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Read}(1^\lambda, T, 0)$ with $k = 0$ and $\text{INDEX}^* = 0$.

$\mathsf{Prep\text{-}Read}(\mathrm{PP}, \mathbf{store}_{in}, \mathbf{index}) \to m, \pi$**.** Parse STORE_{in} as a representation of a binary tree of height q as described above. Let m be the label of the leaf at position INDEX (possibly $\perp$). Let π be the labels associates with all the sibling nodes along the path from the root to the leaf at position INDEX (some of these might be $\perp$). Output m, π.

$\mathsf{Prep\text{-}Write}(\mathrm{PP}, \mathbf{store}_{in}, \mathbf{index}) \to aux$**.** Compute
$aux = (m, \pi) \leftarrow \mathsf{Prep\text{-}Read}(\mathrm{PP}, \text{STORE}_{in}, \text{INDEX})$.

$\mathsf{Verify\text{-}Read}(\mathrm{PP}, w_{in}, m_{read}, \mathbf{index}, \pi) \to \{True, False\}$**.** Given $\mathrm{PP}, m_{read}, \text{INDEX}, \pi$ it is possible to recompute the label associated with the root of the tree as described below, by computing all the labels along the path from the root to the leaf in position INDEX. Then Check that this label matches w_{in} and if so output True else False.

We inductively define the labels at levels $j = 1, 2, \ldots, q$ of the tree as follows: for any node at level j whose left/right children are labeled with $\mathsf{lbl}_l, \mathsf{lbl}_r$ respectively, where at least one of $\mathsf{lbl}_r, \mathsf{lbl}_l$ is not $\perp$, we define the label of that node as $1||H_{\mathsf{hk}_j}(\mathsf{lbl}_l, \mathsf{lbl}_r)$ (and all other nodes are implicitly labeled with $\perp$). For level 0, the label at the INDEX-th leaf is $\perp$ if $m_{read} = \perp$ and it is $1||m_{read}$ otherwise.

The computation is first performed using m_{read} and the given sibling values in π. If the computation matches w_{in} output $True$; otherwise, outputs $False$.

$\mathsf{Write\text{-}Store}(\mathrm{PP}, \mathbf{store}_{in}, \mathbf{index}, m) \to \mathbf{store}_{out}$**.** Set the label of the leaf in position INDEX to m. Update the labels of all the nodes along the path from the root to the leaf in position INDEX (previously some of these may have been $\perp$) inductively for levels $j = 1, 2, \ldots, q$: for any node at level j whose left/right children are labeled with $\mathsf{lbl}_l, \mathsf{lbl}_r$ define the label of that node as $1||H_{\mathsf{hk}_j}(\mathsf{lbl}_l, \mathsf{lbl}_r)$.

$\mathsf{Update}(\mathrm{PP}, w_{in}, m_{write}, \mathbf{index}, aux) \to w_{out}$ **or** $Reject$**.** Parse $aux = (m, \pi)$ and run $\mathsf{Verify\text{-}Read}(\mathrm{PP}, w_{in}, m, \text{INDEX}, \pi)$. If the output if False then Reject and stop. Note that here m is the "old" (possibly $\perp$) value at INDEX that is to be overwritten by m_{write}.

Next, emulate the execution of $\mathsf{Write\text{-}Store}(\mathrm{PP}, \text{STORE}_{in}, \text{INDEX}, m_{write})$ by using the labels contained in π instead of the data contained in STORE_{in}. Output the updated label w_{out} associated with the root of the tree.

B.2 Security Analysis

Theorem B.1. Assuming $\mathcal{H}$ is a perfectly binding two-to-one SSB hash, the above construction gives a positional accumulator.

Proof. We sketch the proof by checking that it meets all of the security properties required from Appendix B. The proof follows in spirit very closely to that given for SSB security in Theorem 3.2.

We first see that the properties of Indistinguishability of Read Setup () and indistinguishability of Write Setup hold. The only difference in the output between $\mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Read}(1^\lambda, T, (m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k), \text{INDEX}^*)$ $\mathrm{PP}, w_0, \text{STORE}_0$ and $\mathsf{Setup\text{-}Acc}(1^\lambda, T)$ is that the $\mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Read}$ algorithm generates $\mathsf{hk}_j \leftarrow \mathsf{Gen}(1^\lambda, 1^{s_j}, b_j)$ where $(b_q, \ldots, b_1)$ be the binary representation of INDEX^*. Whereas the $\mathsf{Setup\text{-}Acc}$ does the same thing except setting $(b_q, \ldots, b_1) = 0^q$. An identical hybrid argument to that given in Theorem 3.2 shows that these two distributions are computationally indistinguishable if the two-to-one index hiding property holds.

We next argue that our construction satisfies read enforcing according to Definition . First let

$$(\mathrm{PP}, w_0, \mathsf{st}_0) \leftarrow \mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Read}(1^\lambda, T, (m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k), \text{INDEX}^*).$$

For j from 1 to k, we define STORE_j iteratively as $\text{STORE}_j := \mathsf{Write\text{-}Store}(\mathrm{PP}, \text{STORE}_{j-1}, \text{INDEX}_j, m_j)$. We similarly define aux_j and w_j iteratively as $aux_j := \mathsf{Prep\text{-}Write}(\mathrm{PP}, \text{STORE}_{j-1}, \text{INDEX}_j)$ and $w_j := Update(\mathrm{PP}, w_{j-1}, m_j, \text{INDEX}_j, aux_j)$. If $\text{INDEX}^* \notin \{\text{INDEX}_1, \ldots, \text{INDEX}_k\}$ then let $m = \perp$; otherwise let $m = m_i$ for the largest $i \in [k]$ such that $\text{INDEX}_i = \text{INDEX}^*$.

Now suppose that $\mathsf{Verify\text{-}Read}(\text{PP}, w_k, m', \text{INDEX}^*, \pi) = True$ for some $m' \neq m$. Then in the hash tree computation of Verify-Read there must be a smallest j such that in the verify computation $H_{\mathsf{hk}_j}(\mathsf{lbl}_l, \mathsf{lbl}_r)$ was equal to the stored value along the path in STORE_k, but the child label along the path to INDEX^* was different. (Otherwise, if there was not such a value the root computations would be different and thing would not verify.) However, this violates the (perfect) binding property of the 2-to-1 hash so it cannot happen.

Finally, we show that the write enforcing property of Definition B.4 the Setup-Acc-Enforce-Write holds. Consider the security game where we let $(\text{PP}, w_0, \mathsf{st}_0) \leftarrow \mathsf{Setup\text{-}Acc\text{-}Enforce\text{-}Write}(1^\lambda, T, (m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k))$. For j from 1 to k, we define STORE_j iteratively as $\text{STORE}_j := \mathsf{Write\text{-}Store}(\text{PP}, \text{STORE}_{j-1}, \text{INDEX}_j, m_j)$. We similarly define aux_j and w_j iteratively as $aux_j := \mathsf{Prep\text{-}Write}(\text{PP}, \text{STORE}_{j-1}, \text{INDEX}_j)$ and $w_j := Update(\text{PP}, w_{j-1}, m_j, \text{INDEX}_j, aux_j)$.

Now consider a call to $\mathsf{Update}(\text{PP}, w_{k-1}, m_k, \text{INDEX}_k, aux) = w_{out} \neq Reject$ where $aux = (m, \pi)$. In the first step the Update algorithm recomputes the hash values and checks it against w_{k-1}. All of these values computed along the path INDEX must match those in STORE_{k-1} or else the algorithm will reject. Otherwise, if the values did not match it would violate the binding properties of the two-to-one scheme. If the values it computes do match those in STORE_{k-1}, then $w_{out} = w_k$ since the algorithm Update's next step is defined to do the same computation as Write-Store.

■