Quantum Cryptanalysis of Subset-Sum Hashing

Xiong Fan and Chris Peikert

Algorand, Inc.

August 13, 2021

1 Concrete Cryptanalysis of Subset-Sum Hash

compression	property	tree depth	time exp. $\geq$	mem exp. $\geq$	time*mem exp. $\geq$
2.0	TCR	6	0.250	0.100	0.350
	CR	8	0.211	0.078	0.289
3.0	TCR	7	0.213	0.082	0.295
	CR	9	0.184	0.070	0.254
4.0	TCR	8	0.190	0.077	0.268
	CR	10	0.167	0.067	0.234
6.0	TCR	10	0.167	0.071	0.238
	CR	12	0.154	0.058	0.212
8.0	TCR	11	0.159	0.062	0.221
	CR	13	0.144	0.055	0.199
10.0	TCR	12	0.151	0.058	0.209
	CR	14	0.138	0.052	0.190
12.0	TCR	13	0.147	0.054	0.201
	CR	14	0.130	0.053	0.183

Figure 1: Bounds for quantum merging-tree attacks against Ajtai's subset-sum hash function, optimized to minimize the time*memory metric. "Compression" c is the compression factor of the function, i.e., it maps cg input bits to g output bits, for some g. "Property" refers to the security property being attacked: (T)CR is (target) collision resistance. "Tree depth" is the smallest depth of the (complete) merging tree that optimizes the attack. "Time exponent" $\geq \mathcal{T}$ means the attack uses at least $2^{\mathcal{T}g}$ (quantum) operations on g-bit words. "Memory exponent" $\geq \mathcal{M}$ means the attack uses at least $2^{\mathcal{M}g}$ words of (classical) memory. "Time*memory exponent" is the sum of the time and memory exponent bounds. **In the Algorand `sumhash-512` implementation, $g = 512$ and $c = 2$, so the quantum attack on TCR (respectively, CR) uses at least 2^{128} (resp., 2^{108}) operations on 512-bit words, and at least 2^{51} (resp., 2^{40}) words of memory, for a time*memory cost of at least 2^{179} (resp., 2^{148}). Therefore, we believe that these parameters safely exceed the desired 128 bits of security in this cost metric.**

1.1 Subset-Sum Hash

Let G be a finite additive abelian group of order $|G| = 2^g$.[1] Let $m = c \cdot g$ denote the input length, for some $c > 1$ called the *compression factor*. For a given $\vec{a} = (a_1, \ldots, a_m) \in G^m$, the *subset-sum* hash function (also known as Ajtai's function) $H_{\vec{a}} \colon \{0,1\}^m \to G$ is defined as

$$H_{\vec{a}}(\mathbf{x}) := \langle \vec{a}, \mathbf{x} \rangle = \sum_{i=1}^{m} a_i x_i \in G.$$

In words, the function simply sums the subset of elements in $\vec{a}$ indicated by the bits of $\mathbf{x}$.[2]

The security notions we consider for hash functions are *(target) collision resistance*. Informally, collision resistance says that it is infeasible to find a collision, i.e., two distinct inputs $\mathbf{x}, \mathbf{x}' \in \{0,1\}^m$ that hash to the same output. Target collision resistance says that it is infeasible to find a collision with a specific given input. For the subset-sum hash, these notions can be formalized as follows:

- CR: given uniformly random $\vec{a} \in G^m$, the adversary is asked to output distinct $\mathbf{x}, \mathbf{x}' \in \{0,1\}^m$ such that $H_{\vec{a}}(\mathbf{x}) = H_{\vec{a}}(\mathbf{x}')$. By definition and rearranging, this is equivalent to finding some nonzero $\mathbf{z} \in \{0, \pm 1\}^m$ such that $\langle \vec{a}, \mathbf{z} \rangle = 0 \in G$, i.e., a nontrivial "zero preimage" $\mathbf{z}$ from the relaxed domain $\{0, \pm 1\}^m$.
- TCR: the adversary initially chooses some $\mathbf{x} \in \{0,1\}^m$, and then, given uniformly random $\vec{a} \in G^m$, is asked to output some $\mathbf{x}' \in \{0,1\}^m$ such that $\mathbf{x}' \neq \mathbf{x}$ and $H_{\vec{a}}(\mathbf{x}') = y := H_{\vec{a}}(\mathbf{x})$. By definition and rearranging, this is equivalent to $\langle (\vec{a}, -y), (\mathbf{x}', 1) \rangle = 0$. This is essentially a more demanding form of the above CR experiment, where the adversary must produce a nontrivial *binary* (not ternary) zero preimage relative to $(\vec{a}, -y)$, whose final entry is 1.[3]

Therefore, the goals in the (T)CR attacks can be unified as follows:

Definition 1.1 (Unified collision finding). Given uniformly random $\vec{a} \in G^m$, find a nonzero $\mathbf{x} \in B^{cg}$ such that $H_{\vec{a}}(\mathbf{x}) = 0 \in G$, for a specified entry set $B \subset \mathbb{Z}$. For TCR and CR, the entry sets are respectively $B = \{0,1\}$ and $B = \{0, \pm 1\}$.

For the class of attacks we analyze, we need to recast the collision-finding problem in the following way. For some positive integer k (chosen by the attacker), factor the hash function's domain $D = B^{cg}$ as a Cartesian product $D = D_1 \times \cdots \times D_k$, where $D_i = B^{c_i g}$ for some desired non-negative reals $c_i \geq 0$, and $c = \sum_i c_i$.[4] Note that each subdomain D_i has cardinality $|D_i| = 2^{d_i g}$, where $d_i := c_i \log |B|$; we call these d_i the *subdomain exponents* and $d = \sum_i d_i$ the *domain exponent* (relative to the codomain size 2^g).

Correspondingly, express

$$H_{\vec{a}}(\mathbf{x}) = \sum_i H_i(\mathbf{x}_i),$$

[1] A common choice in lattice cryptography, which we also adopt in this work, is $G = \mathbb{Z}_q^n$ for some positive integers q, n. However, for cryptanalysis the group order is the primary parameter governing security.

[2] One can generalize the input domain $\{0,1\}^m$ to allow larger integers or negative ones, with a corresponding degradation in concrete security. For our purposes, bits suffice.

[3] We do note a subtle difference with the CR experiment: $(\vec{a}, -y)$ is not uniformly random given the adversary's entire view, because $y = \langle \vec{a}, \mathbf{x} \rangle$ and the adversary knows $\mathbf{x}$. However, the best known attacks do not appear to benefit from any potential non-uniformity, and in fact do not use $\mathbf{x}$ in any way at all. Moreover, if $\mathbf{x}$ has sufficient min-entropy (which is often the case in applications of interest), then the leftover hash lemma implies that $(\vec{a}, -y)$ is indeed uniformly random to any attack that ignores $\mathbf{x}$.

[4] Note that this abstraction favors the adversary, since this exact factorization may not be possible for the precise desired reals c_i, if some $c_i g$ is not integral. However, this has very little effect on our ultimate security bounds for parameters of interest.

where each subfunction $H_i \colon D_i \to G$ and $\mathbf{x}_i \in D_i$ is respectively defined by the corresponding part of $\vec{a}$ and $\mathbf{x}$. Then the collision-finding problem of Definition 1.1 can be restated as:

Definition 1.2 (k-sum problem). Given the subfunctions $H_i \colon D_i \to G$, find nontrivial $x = (x_1, \ldots, x_k) \in D_1 \times \cdots \times D_k = D$ such that $H(x) := \sum_i H_i(x_i) = 0$.

The above k-sum problem is closely related to the k-xor problem studied in a long series of prior works: given oracle access to a random function $H \colon \{0,1\}^d \to \{0,1\}^g$, the k-xor problem asks the adversary to output distinct $x_1, \ldots, x_k \in \{0,1\}^d$ for which $H(x_1) \oplus \cdots \oplus H(x_k) = 0$. There are a few main differences between this k-xor problem and the collision-finding problem we consider in Definitions 1.1 and 1.2.

- In the k-xor problem, the same hash function H is applied to all the x_i and the results are XORed (so the x_i must be distinct in order for the problem to be nontrivial), while in k-sum a different function is applied to each x_i (hence there is no distinctness condition) and the results are summed under the group operation. These do not appear to be major differences, since the best known algorithms for k-xor are easily adaptable to our setting.
- In the k-xor problem, the parameter k and hash domain $\{0,1\}^d$ are fixed and given, but in our problem the adversary can choose k and the subdomain sizes $|D_i|$ however it likes, subject to the overall domain size. This is a significant difference between the problems: in k-xor the hardness decreases as k and d increase (up to a point), whereas in our problem the adversary can optimize the tradeoff between them.

Despite these differences, prior work on k-xor is very useful for analyzing our problem. In fact, the freedom to choose k subject to a constraint on the total domain size actually makes it easier to optimize the known quantum attacks, as we shall see below.

1.2 Merging Trees

We analyze quantum algorithms for the k-sum problem using the quantum *merging trees* framework of Schrottenloher [Sch21], which generalizes and improves upon a long series of (both classical and quantum) algorithms going back to the generalized-birthday attacks of Blum, Kalai, and Wasserman [BKW00] and Wagner [Wag02].

1.2.1 Quantum Merging Trees

Definition 1.3 (Merging tree). A *k-merging tree* T is a (possibly incomplete) binary tree having exactly k leaves, where:

- each node is either a *list* node (L-node) or a *sample* node (S-node), and
- every non-leaf node has exactly one L-node child and one S-node child.

In addition, the root of an entire merging tree (that is not a subtree of some larger tree) must be an L-node.[5]

Each of the k leaves of a k-merging tree corresponds to a different subfunction $H_i \colon D_i \to G$ from the k-sum problem (Definition 1.2). More generally, each subtree S of T corresponds to the function $H_S \colon D_S \to G$, where D_S and H_S are respectively defined as the Cartesian product of the subdomains, and

[5]This is simply because the root node represents the corresponding algorithm's output, which must be written down explicitly.

the sum of the subfunctions, associated with the leafs of S. In particular, the domain and function associated with the entire tree T are $D_T = D$ and $H_T = H$, respectively.

Looking ahead a bit, any merging tree has a corresponding quantum algorithm. For each S-node at the root of a subtree S, it *samples* an input-output pair $(x \in D_S, y = H_S(x))$ satisfying a certain condition, and for each L-node it *constructs a list* of such pairs by repeatedly sampling. The main difference between the two types of nodes is that the set associated with an S-node is not stored in memory (it is only sampled), whereas the list associated with an L-node is stored in memory and searched (possibly repeatedly).

We associate the following parameters to each node v of a merging tree.

- The *zero-prefix fraction* z_v is a lower bound on the fraction of the "initial bits" of the y elements (from the pairs (x, y) sampled at v) that is guaranteed to be zero. For example, if $G = \{0,1\}^g$ (with bitwise exclusive-or as the group operation), then the first $z_v g$ bits of each y are zero, and hence the sum of any such elements also has the same property.

  More generally, we can imagine that G has a fixed tower of subgroups, one of which has order $2^{(1-z_v)g}$ and contains all the y elements (and hence any sum of such elements). However, the algorithm does not actually require such subgroups, and can be made to work just as well with "near subgroups" (e.g., sets of suitably bounded integers).

  In any case, observe that this abstraction favors the adversary, since the exact desired fraction z_v of "bits" may not be possible, if $(1 - z_v)g$ is not integral.

- The *size exponent* ℓ_v denotes the cardinality $2^{\ell_v g}$ of the constructed list corresponding to the (L-)node, or of the sample space corresponding to the (S-)node.

- The *sample-time exponent* t_v of a node v denotes the time $2^{t_v g}$ needed to sample a single element associated with the node, *after* any of the needed lists at L-nodes have already been constructed. It is defined as a certain function of the other variables based on the operation of the merging algorithm, as shown below.

Constraints. In order for a merging tree to represent a valid algorithm, the node parameters z_v and ℓ_v are constrained in the following ways.

1. *Root:* the root node r has the desired number of all-zero elements: $z_r = 1$ and $\ell_r = 0$. (Or, if we seek many collisions, we can let $\ell_r > 0$.)
2. *Zeros:* each non-leaf node v represents the merging of its two children s, l, so the children must have the same bound on their fraction of zeros (without loss of generality), and the merge does not decrease this fraction:
$$z_v \geq z_l = z_s. \tag{1.1}$$
3. *Size:* the size at a non-leaf node v is limited by the sizes at its two children l, s, and the fraction of additional bits being zeroed out:
$$\ell_v = \ell_l + \ell_s - (z_v - z_l). \tag{1.2}$$
This is because the exponent for the number of combined list-sample child pairs is $\ell_l + \ell_s$, and the merging process selects only those having an additional $z_v - z_l$ fraction of zero bits (see the algorithm description below for details).

4. *Domain size:* For each subdomain D_i, which is associated with the ith leaf node v, the subdomain exponent d_i must be at least $z_v + \ell_v$, due to the leaf's combined size and zero-prefix requirements. Recall that the attacker may factor the overall domain D into subdomains however it likes, so by considering all leaf nodes, we have the constraint

$$d \geq \sum_{\text{leaves } v} (z_v + \ell_v). \tag{1.3}$$

Algorithm. We now describe the quantum merging algorithm A_T associated with a merging tree T, which is defined recursively as follows. If T's root node v is an L-node, then A_T runs the following sampling procedure $2^{\ell_v g}$ times, storing the results in a list in memory. Otherwise (i.e., v is an S-node), A_T samples an input-output pair $(x \in D_T, y = H_T(x) \in G)$ where y has a zero-prefix fraction at least z_v, as follows:

- If T is merely a leaf node, let $H_i \colon D_i \to G$ be the corresponding subfunction from the k-sum problem. Quantumly sample an input-output pair $(x \in D_i, y = H_i(x) \in G)$, where x is uniformly random conditioned on y having zero-prefix fraction at least z_v.
- Otherwise, T is not a leaf; let L, S be the subtrees rooted at T's L- and S-node children, respectively.
  1. Run A_L to build a list of input-output pairs $(x_i^L \in D_L, y_i^L = H_L(x_i^L) \in G)$, where each y_i^L has the appropriate zero-prefix fraction.
  2. Run the sampling algorithm A_S to obtain an input-output pair $(x^S, y^S = H_S(x^S) \in G)$, and search the list (in quasi-linear quantum time) for a pair (x_i^L, y_i^L) such that $y_i^L + y^S \in G$ has zero-prefix fraction at least z_v.

     Repeat (more accurately, amplify) this entire sample-and-search process until it succeeds, and output the resulting sample $(x = (x_i^L, x^S) \in D_T, y = y_i^L + y^S = H_T(x) \in G)$.

Observe that by taking $z_v = 1$, the algorithm produces at least one pair $(x, y = H(x) = 0 \in G)$, as desired.

The algorithm can be implemented advantageously using classical memory with quantum random access (QRACM), i.e., the memory holds classical data, but superposition access is allowed. The algorithm uses (a generalization of) Grover's search as a key subroutine, as detailed in [Sch21]. In summary, the sample-time exponent of a node v is as follows:

- if v is a leaf, then we just sample an input-output pair where the output has the desired zero-prefix fraction, so $t_v := z_v/2$;
- otherwise, letting s, l respectively be the S- and L-node children of v, [Sch21, Lemma 2] shows that

$$t_v := t_s + \tfrac{1}{2} \max(0, z_v - z_l - \ell_l). \tag{1.4}$$

The (exponential) running time and memory usage of the algorithm associated with a merging tree are then given by the following quantities.

Definition 1.4. Let T be a k-merging tree (and recall that its root is an L-node). The *time exponent* $\mathcal{T}(T)$ and *memory exponent* $\mathcal{M}(T)$ of the associated algorithm A_T are

$$\mathcal{T}(T) := \max_{\text{L-nodes } l} (t_l + \ell_l), \quad \text{and} \quad \mathcal{M}(T) := \max_{\text{L-nodes } l} \ell_l.$$

The correctness of these quantities can be seen as follows. In the algorithm, the lists at the L-nodes need to be constructed and stored in memory. For each L-node l, we call the sampling subroutine $2^{\ell_l g}$ times, where each sample takes time $2^{t_l g}$ by definition, for an overall time exponent of $t_l + \ell_l$ and memory exponent of ℓ_l. Thus, the time and memory exponents of the entire algorithm are respectively the maxima of these, taken over all L-nodes in the tree. (Note that since the algorithm can be executed in a depth-first manner, the memory can be reused for different L-nodes, but this affects only the lower-order factors, not the exponent.)

1.2.2 Optimization

We now consider the optimization of merging trees for our specific problem of interest (Definitions 1.1 and 1.2). That is, for a given domain size, we wish to find a value of k, a topology for a k-merging tree T, and valid node parameters z_v, ℓ_v that minimize a desired metric. We mainly focus on the time-memory metric, given by the total exponent $\mathcal{T}(T) + \mathcal{M}(T)$, but what follows applies equally well when optimizing the time exponent $\mathcal{T}(T)$ alone.

Given some moderate value of k (up to several thousands or more) and a k-merging tree topology T, one can quickly find optimal node parameters using an off-the-shelf optimization library.[6] However, as k grows beyond a dozen or so, the number of binary trees having k leaves grows very rapidly, and the cost of trying all of them quickly becomes prohibitive.

To address this problem, here we show some novel structural properties of merging trees that allow us to restrict our attention to *complete* binary trees, of small or moderate height for hash parameters of interest. Our main result is that any topological "extension" of a valid merging tree is able to "simulate" the original one, obtaining the same or better time and memory exponents. Therefore, when optimizing over all merging trees, it suffices to consider only complete trees, which dramatically shrinks the search space. For the same reason, increasing the height of the complete tree cannot increase the tree's optimal time and memory exponents, so it suffices to consider a suitably tall complete tree.

We first show that in any nontrivial merging tree (whether optimal or not), the zero-prefix fraction of every leaf can be made zero without any loss in efficiency. So, we impose this constraint in all that follows and in our optimization scripts.

Lemma 1.5. *For any $k > 1$, let T be a k-merging tree with valid node parameters. Then for every leaf node v of T, the zero-prefix fraction can be made $z'_v = 0$ while preserving validity, without changing $\mathcal{M}(T)$, and without increasing $\mathcal{T}(T)$.*

Proof. Leaving all non-leaf nodes unchanged, consider the following alterative parameters for each leaf node:

- if the leaf is an L-node l, then set $z'_l = 0$ and keep $\ell'_l = \ell_l$ unchanged;
- otherwise it is an S-node s, so set $z'_s = 0$ and $\ell'_s = z_s + \ell_s$.

Also, for each node v in T, let t'_v denote its sample-time exponent according to these parameters.

We now show that the alternative parameters still satisfy all the constraints. The root constraint (Item 1) is satisfied because the root's values are unchanged, since $k > 1$. The zeros constraint (Item 2) is satisfied because all leaves v have $z'_v = 0$. The list-size constraint (Item 3) is satisfied because $z_l = z_s$ for every pair of sibling leafs l, s with parent node v, so

$$\ell_v = \ell_s + \ell_l - (z_v - z_l) = \ell_s + z_s + \ell_l - z_v = \ell'_s + \ell'_l - (z_v - z'_l).$$

[6] While all the constraints are linear, the objective function is not quite so due to the appearance of $\max$, so one needs to go slightly beyond linear programming.

Finally, the domain-size constraint (Item 4) is satisfied because for every leaf node v we have $z_v + \ell_v \geq z'_v + \ell'_v$.

The memory exponent is unchanged simply because the size exponent ℓ_v of every L-node v is unchanged. For the time exponent, let l, s be any pair of sibling leaves with parent node v; then $t'_l + \ell'_l = \ell_l \leq t_l + \ell_l$, and because $t_s = z_l/2$, we have

$$t'_v = t'_s + \tfrac{1}{2}\max(0, z_v - z'_l - \ell'_l) = \tfrac{1}{2}(\max(0, z_v - \ell_l)) \leq t_s + \tfrac{1}{2}\max(0, z_v - z_l - \ell_l) = t_v.$$

It follows by induction that $t'_v + \ell'_v \leq t_v + \ell_v$ for all L-nodes v, so the time exponent is not increased, as claimed. □

We next show that extending the topology of a merging tree (by iteratively adding children to leaves) cannot worsen the optimal time or memory exponents for the tree.

Definition 1.6. Let T and T' be binary trees. We say that T' *extends* T (or *is an extension of* T) if it can be obtained from T by a (finite) sequence of "leaf extension" steps, each of which adds two children to some leaf of the current tree.

Lemma 1.7. *Let T, T' be merging trees where T' extends T. Any valid node parameters for T (having zero-prefix fraction $z_v = 0$ for every leaf v) extend to ones for T', with $\mathcal{T}(T') = \mathcal{T}(T)$ and $\mathcal{M}(T') = \mathcal{M}(T)$.*

Proof. By induction, it suffices to show how to extend an arbitrary leaf v of T with two children l, s (having zero-prefix fractions $z_l = z_s = 0$), while satisfying all the constraints and preserving the time and memory exponents. To do this, we simply define $z_l = 0, \ell_l = 0$ and $z_s = 0, \ell_s = \ell_v$.

With this extension, the root constraint (Item 1) remains satisfied; the zeros constraint (Item 2) is trivially satisfied because $z_v = z_l = z_s = 0$; the size constraint (Item 3) is satisfied because $\ell_v = \ell_s + \ell_l$ and $z_v = z_l = 0$; and the domain-size constraint (Item 4) is satisfied because the contribution $z_v + \ell_v = \ell_v$ of the original leaf v is replaced by the equal contribution $(z_l + \ell_l) + (z_s + \ell_s) = \ell_v$ of new leaves l, s.

Lastly, the extension leaves the time and memory exponents unchanged, because all the L-nodes in the original tree are unchanged, and the only new L-node l has $t_l = z_l/2 = 0$ and $\ell_l = 0$. □

Lemmas 1.5 and 1.7 immediately imply the following.

Corollary 1.8. *The optimal time and memory exponents of a* complete *binary tree are no larger than for any tree of equal or lesser height.*

References

[BKW00] A. Blum, A. Kalai, and H. Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, 2003. Preliminary version in STOC 2000.

[Sch21] A. Schrottenloher. Improved quantum algorithms for the k-XOR problem. Cryptology ePrint Archive, Report 2021/407, 2021. `https://ia.cr/2021/407`.

[Wag02] D. Wagner. A generalized birthday problem. In *CRYPTO*, pages 288–303. 2002.