

Wagner’s k -Tree Revisited: Memory-Time Costs of Preimage Attacks on Incremental Hashes

Rui Ding^{a,1,2} , Lili Tang^{a,1,2} , Shaomin Chen^{1,2}  and Xiaorui Gong^{1,2} 

¹ Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China

² School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

Abstract

Wagner’s k -tree algorithm solves the generalized birthday problem and underlies preimage attacks on randomize-then-combine incremental hashes (EUROCRYPT ’97) such as iSHAKE, LtHash, and AdHash. In its full-index execution, the 2^{k-1} index entries dominate peak memory at large k , and index trimming narrows each entry but leaves their number intact. Tang et al. (TCHES ’26) introduced post-retrieval for the single-chain algorithm and left the 2^k exponent of the k -tree setting as an open problem. We resolve it by extending post-retrieval to the k -tree algorithm. This cuts the peak working memory to $P_k \ell N = O(k^2 \ell N)$, removing the 2^{k-1} index entries from the forward pass at the cost of a $\Theta(k)$ time overhead. Free δ -level caching shrinks this time factor at no memory cost. The trade-off is starkly asymmetric: it removes an exponential number of index entries for only a linear recovery-time overhead.

As a practical application, we revisit the list-item-reduction landscape for the k -tree algorithm under the memory-time product metric, $MT = M \cdot T$. The gain is regime-dependent: for small k the index entries do not yet dominate, so the recovery overhead outweighs the saving. For large k the saving dominates, lowering the optimized $\log_2 MT$ from $4\sqrt{n}$ to $2\sqrt{2n}$ at leading order. For fixed-size iSHAKE preimage attacks, we save approximately 51 and 122 bits over state-of-the-art index trimming for iSHAKE-128 and iSHAKE-256, respectively, in the unlimited-block setting, narrowing to roughly 6 and 8 bits under block-count caps.

Keywords: Wagner algorithm, generalized birthday problem, incremental hashing, memory-time trade-off

1 Introduction

Wagner’s k -tree algorithm [Wag02] solves the generalized birthday problem: given $K=2^k$ independent lists of n -bit values, the goal is to find one element from each list whose XOR is zero. When each list has size 2^ℓ , where $\ell = n/(k+1)$ and $N = 2^{\ell+1}$, the algorithm runs in time $O(2^{k+\ell})$ and remains the fastest known algorithm for hash-generated instances in the equal-list-size setting [MGS16].

In symmetric cryptography, the generalized birthday problem arises naturally in the context of incremental hashing [BGG94]. Incremental hash functions allow a digest to be updated after local modifications without recomputing it from scratch. This appeals to applications that frequently modify large data objects, such as file-system integrity checking, database auditing, and blockchain account-state management. In the randomize-then-combine paradigm, each block is hashed to a group element and the results are

Emails: dingrui2024@iie.ac.cn (Rui Ding), tanglili@iie.ac.cn (Lili Tang), chenshaomin@iie.ac.cn (Shaomin Chen), gongxiaorui@iie.ac.cn (Xiaorui Gong)

^aThe first two authors contributed equally to this work.

combined in a commutative group. Representative constructions include AdHash and LtHash [BM97], multiset hashing [CDvD⁺03], and iSHAKE [MGS16], with notable deployments at Meta [LKMW19] and Solana [PCSK24]. Fixed-position preimage attacks (the message blocks occupy fixed, known positions) against iSHAKE and additive variants of AdHash/LtHash reduce to instances of the Regular Generalized Birthday Problem (RGBP) with independent hash-generated leaf lists. For iSHAKE, this typically places the attack in the large- k regime.

A standard k -tree execution stores, in each list item, a running hash value together with an index vector tracing it back to the leaves. In this large- k regime the peak memory is dominated by the index vectors rather than the hash values. The baseline *full-index* execution, which keeps these index vectors in full, is denoted TIV (k -Tree with Index Vector). In the post-order stack execution (the traversal that keeps only $O(k)$ lists in memory at once), its peak memory splits into a hash-value term with a coefficient polynomial in k and an exponential index-vector term $2^{k-1}\ell N$. The authors of [TDSG26] analyzed an index-trimming variant, denoted TIV-IT (TIV with Index Trimming), which retains one bit of each index entry during the forward pass and reconstructs the full indices through an additional constrained run. Thus TIV and TIV-IT differ only in the per-entry storage cost, ℓ bits versus one bit, while both keep the same 2^{k-1} index entries (the entries held in memory at once). At the time-optimal parameters for iSHAKE-128, namely $k=64$ and $\ell=64$, this index-vector term is 2^{134} bits for TIV and 2^{128} bits for TIV-IT, against roughly 2^{81} bits for hash values. Since memory is often more expensive than computation in k -tree algorithms, this exponential term is the dominant cost, yet existing list-item trade-offs do not remove it. The other main memory technique, list-size reduction [Ber07, MS12, NS15, Din19], only targets the regime $2^k \ll n$ and thus treats 2^k as a constant, which is not viable in the incremental-hashing setting considered here.

Beyond Time Complexity Bernstein [Ber07] observed that operation counts alone do not capture machine cost: a first-order price-performance metric is the memory-time product $MT = M \cdot T$; see Section 2.5. For both TIV and TIV-IT, the 2^{k-1} index entries force $\log_2 M$ to share the leading term $k+\ell$ with $\log_2 T$; hence $\log_2 MT$ carries leading term $2k+2\ell$, minimized at the same scale $k \approx \sqrt{n}$ as $\log_2 T$ alone [Wag02]. Trimming the per-entry width leaves this leading term unchanged. Only removing the entries themselves can alter the MT optimization.

Our Approach Removing these entries calls for *post-retrieval*: keep only hash values in the forward pass and recover the indices afterward. Tang et al. [TDSG26] introduced post-retrieval for the practical single-chain algorithm [LF06, BK16], cutting its peak memory from about $2nN$ to nN . For the k -tree the index term is instead exponential in k , and [TDSG26] leave post-retrieval open. Because k -tree merging depends only on hash values, we recover a match by rebuilding sibling subtrees and locating each child by hash, removing the exponential term at a $\Theta(k)$ time cost. Building on this, we adapt post-retrieval to the k -tree as PR-TIV (Post-Retrieval TIV), use δ for the number of cached levels and $\delta_{\text{free}}(k)$ for the largest cache depth that does not increase the modeled peak list-payload memory, and revisit the list-item-reduction landscape for the k -tree under the MT metric.

1.1 Contributions

Algorithm Post-retrieval was known only for the single-chain algorithm. We establish its k -tree counterpart, completing the memory-time picture for both variants of the generalized birthday problem. In the recovery, a fixed parent value at each level renders the two child subtrees independent, letting the depth-first search (DFS) reuse memory across siblings.

This reduces peak working memory to $P_k \ell N = O(k^2 \cdot \ell N)$, where $P_k = (k^2 + 5k + 2)/4$ is the hash-only memory coefficient; pure PR-TIV incurs a $\Theta(k)$ time overhead, while δ -level caching interpolates between pure PR-TIV and full-index TIV. See Section 3. Table 1 compares the relevant k -tree list-item reduction (LIR) methods and indicates which still carry an exponential index-vector term. The artifact recomputes the closed-form tables and figure coordinates, simulates the exact memory schedule, and validates small instances with a seeded BLAKE2b oracle.

Table 1: k -tree list-item reductions. Here $T_0 = 2^{k+\ell}$ is one forward-pass time unit and $\gamma > 1$ is the index-trimming rerun factor of Section 2.4. For PR-TIV with free caching ($\delta \leq \delta_{\text{free}}$), the index term $2^{\delta-1} \ell N$ is absorbed into the hash-value term $P_k \ell N$, so peak working memory remains $P_k \ell N$. Derivations appear in Section 3.

Algorithm	Peak working memory	Time	Index term
<i>Baselines</i>			
TIV [Wag02]	$(P_k + 2^{k-1}) \ell N$	T_0	$2^{k-1} \ell N$
TIV-IT [TDSG26]	$(P_k \ell + 2^{k-1}) N$	γT_0	$2^{k-1} N$
<i>Proposed</i>			
Pure PR-TIV	$P_k \ell N$	$(k+1)T_0$	0
PR-TIV with free caching	$P_k \ell N$	$(k-\delta+1)T_0$	$2^{\delta-1} \ell N$

MT Consequences Removing the index-vector term lowers the leading term of $\log_2$ MT from $2k + 2\ell$ to $k + 2\ell$. This shifts the MT-optimal k from $\approx \sqrt{n}$ to $\approx \sqrt{2n}$ and the optimized $\log_2$ MT from $4\sqrt{n}$ to $2\sqrt{2n}$. The gain is regime-dependent: at small k the recovery overhead fails to pay for itself, so index trimming remains preferable, and Section 4 locates the crossover. The gain shrinks under the output-inclusive convention but remains $\sqrt{n} + O(\log n)$. We further prove that list-size reduction (LSR) in its most direct form, uniformly shrinking all lists and compensating by repetition, never lowers the MT product. Joint optimization therefore selects no reduction (Proposition 3).

Concrete iSHAKE Impact For fixed-size iSHAKE instances [MGS16], PR-TIV reduces the optimized $\log_2$ MT relative to TIV-IT by approximately 51 bits for iSHAKE-128 and 122 bits for iSHAKE-256 in the unlimited-block regime. Under the corresponding block-count caps, the reductions are approximately 6 and 8 bits, respectively.

Figure 1 illustrates the resulting $\log_2$ MT landscape for iSHAKE-256, with output-inclusive accounting reported in Section 5.

1.2 Related Work

The k -Tree Algorithm Wagner [Wag02] introduced the k -tree algorithm for the generalized birthday problem; Dinur [Din19] later recast it as a broad algorithmic framework. Beyond incremental hashing, Wagner-type algorithms have been applied to blind-signature forgery via the ROS problem [BLL⁺22], memory-hard proof-of-work [BK16], learning parity with noise [BKW03], syndrome decoding [ES24], and the SIS_∞ lattice problem [DEL25]. Prior work [TSG26] formalized the regular and loose variants, provided an exact memory analysis of the k -tree, and showed that incremental-hashing preimage attacks naturally give rise to RGBP instances.

Memory-Reduction Landscape Memory reductions for Wagner-type algorithms target either the list size or the per-item representation. List-size reduction [Ber07, MS12, NS15, Din19] keeps less of each intermediate list and pays for the lower success probability by

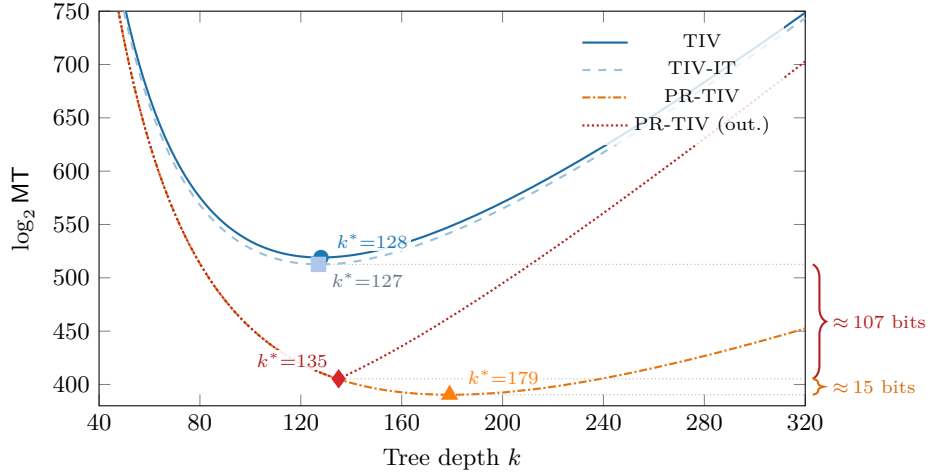


Figure 1: MT landscape for iSHAKE-256 in the unlimited-block setting ($n = 16512$, $\gamma = 3/2$). PR-TIV uses $\delta = \delta_{\text{free}}(k)$, while PR-TIV (out.) follows the output-inclusive convention of Section 4.4 with $r=1$.

repeating the attack. For the loose generalized birthday problem, this leads to Dinur’s trade-off curve $T^{\lceil k/2 \rceil + 1} M^{\lfloor k/2 \rfloor} = 2^n$. List-item reduction instead keeps the same list sizes but stores less information per item. Index trimming and XOR removal appeared in the FSB attack of Bernstein et al. [BLN⁺09], and index trimming was later used in Equihash [BK16]. For the k -tree, this stack-memory accounting provides one baseline, while Tang et al. [TDSG26] systematized the LIR variants, analyzed the TIV-IT baseline, and introduced post-retrieval for the single-chain algorithm. These works provide the baselines for our comparison, but the remaining distinction is structural: existing k -tree LIR methods compress the index entries, whereas PR-TIV removes the exponentially many index entries from the forward pass and reconstructs the leaf indices of a solution (its *witness*) only after a root match.

1.3 Roadmap

Section 2 reviews the RGBP model, the k -tree algorithm, the full-index, index-trimming, and post-retrieval baselines, and the $M \cdot T$ cost model. Section 3 presents PR-TIV and δ -level caching. Section 4 analyzes MT-optimal parameters, pointwise crossovers, and the interaction with list-size reduction. Section 5 instantiates the comparison for iSHAKE and discusses its design-level implications. Section 6 concludes with open problems.

Code availability. The source code used to produce the figures and tables in this paper is available at <https://github.com/Threonine/wagner-ktree-revisited>.

2 Preliminaries

This section introduces the k -tree algorithm, its memory accounting, and the $M \cdot T$ cost metric. We follow the terminology of [TSG26] and the memory-optimization taxonomy of [TDSG26].

2.1 The Regular Generalized Birthday Problem

Unless stated otherwise, list elements are modeled as uniformly distributed elements of $\{0, 1\}^n$, with addition written as bitwise XOR.

Definition 1 (Regular generalized birthday problem [Wag02]). An instance of RGBP(n, K) consists of K independent lists $L_1, \dots, L_K$ of n -bit values. The goal is to find $x_i \in L_i$, one from each list, with $x_1 \oplus \dots \oplus x_K = 0$.

Recent work [TSG26] formalized the regular/loose distinction. In incremental hashing, fixed-size preimage attacks give rise to RGBP instances with one list for each block position, whereas collision attacks lead to the loose variant (LGBP). The two variants differ in structure: the regular variant keeps the K lists $L_1, \dots, L_K$ separate and is solved by Wagner’s k -tree, whereas the loose variant collapses them to a single list, solved by the *single-chain* algorithm [LF06, BK16]. This paper focuses on the RGBP setting.

We consider RGBP instances whose leaf lists are generated on demand by a public oracle $\mathcal{H}$, where $\mathcal{H}(s, j)$ denotes the j th element of the s th list. Thus, each list can be regenerated without persistent storage. This regenerability is the key property that enables post-retrieval.

Throughout Sections 2–4 we set

$$K = 2^k, \quad N = 2^{\ell+1}, \quad |L_i| = N/2 = 2^\ell, \quad \ell = \frac{n}{k+1}.$$

This is the standard continuous-exponent convention used in asymptotic analyses of Wagner’s algorithm [Wag02, TSG26]. Equivalently, one may assume $(k+1) \mid n$ in the exact formulas.³ All memory bounds in this paper refer to expected peak working memory, where the expectation is taken over the uniformly distributed list elements.

2.2 Wagner’s k -Tree

Wagner’s k -tree algorithm [Wag02] solves RGBP($n, 2^k$) by arranging the 2^k input lists as the leaves of a complete binary tree of depth k and merging sibling lists bottom-up through partial collisions. In the full-index execution, a list item at level i is a pair (h, I) . Here h is a current hash value of $(k+1-i)\ell$ bits, since $i\ell$ bits have been eliminated by previous collisions, and I is an index vector of 2^i entries of ℓ bits each, recording which leaf-list entries produced h .

For two lists A and B at the same level and a collision width b , define

$$\text{merge}_b(A, B) = \left\{ ((h_a \oplus h_b) \gg b, I_a \| I_b) \mid \begin{array}{l} (h_a, I_a) \in A, (h_b, I_b) \in B, \\ \text{LSB}_b(h_a \oplus h_b) = 0 \end{array} \right\}.$$

Here $\text{LSB}_b(x)$ denotes the b least significant bits of x , $x \gg b$ denotes the value obtained by dropping these b bits, and $I_a \| I_b$ denotes concatenation of index vectors. If both input lists have expected size 2^ℓ and the hash values are uniform, then merge_ℓ also has expected size 2^ℓ .

The algorithm applies merge_ℓ at the first $k-1$ internal levels. At the root, the two input lists have expected size 2^ℓ and contain values of length 2ℓ . Applying $\text{merge}_{2\ell}$ therefore produces a constant expected number of solutions. Thus the base running time is $\Theta(2^{k+\ell})$; we take

$$T_0 = 2^k \cdot \frac{N}{2} = 2^{k+\ell}$$

as the unit of time and report all costs as multiples of T_0 .

³A bit-level implementation with $(k+1) \nmid n$ should choose integer collision widths $b_1, \dots, b_k$ whose total removed width is n ; uniformly replacing ℓ by $\lfloor n/(k+1) \rfloor$ would reduce the success probability.

Remark 1 (Group generality). The k -tree algorithm and its preimage attacks on iSHAKE, LtHash, and AdHash are established for their respective groups in [MGS16, BM97, TSG26]. For clarity, we state our definitions and theorems over $\{0, 1\}^n$ under XOR. They extend without change to 2-power component groups such as $(\mathbb{Z}/2^w\mathbb{Z})^m$, where quotient evaluation, partial-collision testing, and unique complementation are performed coordinate-wise. For $\mathbb{Z}_M$ with M not a power of 2, as in AdHash, the merge operation uses interval-based partial collisions [TSG26], and hash-directed recovery reduces to an interval range-search with the same expected cost.

Wagner’s k -tree admits a post-order stack execution. Leaves are loaded from left to right. Whenever the top two stack lists have the same level, they are merged and the resulting parent list is pushed back onto the stack. This execution stores only $O(k)$ lists simultaneously. Its peak occurs when the last two leaves are being merged: two level-0 lists are held, and one at each level $1, \dots, k-1$. Figure 2 illustrates this configuration for $k = 4$.

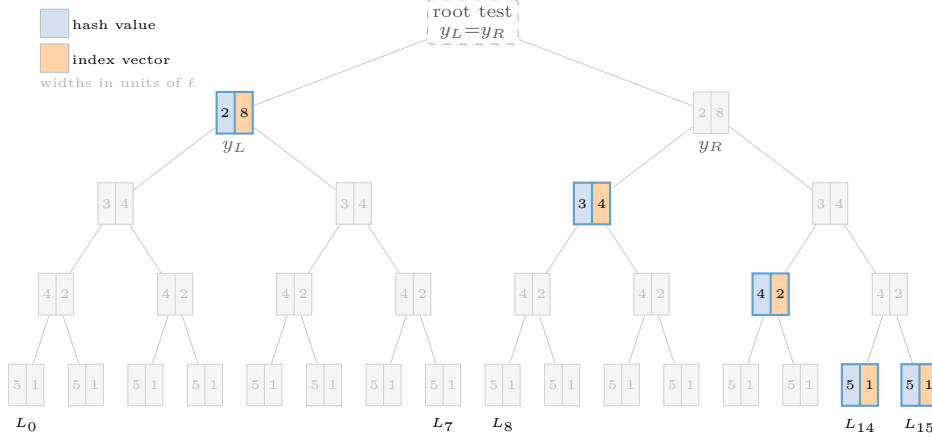


Figure 2: Full-index k -tree execution for $k = 4$ at the peak stack configuration. Split nodes show the hash-value and index-vector widths in units of ℓ ; highlighted nodes denote the five held lists. The dashed root represents the final $\text{merge}_{2\ell}$ test and is not stored as an intermediate list.

2.3 Full-Index Memory

Peak working memory denotes the maximum number of list-item payload bits held in intermediate lists at any one instant. Lookup-structure overhead contributes only a standard constant factor and is therefore omitted. This convention does not include storage for the final solution after it has been output separately. The following proposition is the exact stack-memory calculation of prior work [TSG26, App. D.1, Prop. 5], written in the notation of this paper.

Proposition 1 (Full-index k -tree memory). *For $\text{RGBP}(n, 2^k)$, the full-index k -tree execution has running time $O(2^k N)$ and peak working memory*

$$M_{\text{TIV}} = (P_k + 2^{k-1}) \ell N \quad \text{bits}, \quad P_k = \frac{k^2 + 5k + 2}{4}.$$

The calculation relies on the standard invariant that, at each non-root merge level, the output list remains uniform in its remaining hash bits and has expected size $N/2$. For large k , the $2^{k-1} \ell N$ index-vector term dominates the memory bound.

2.4 Index Trimming and Post-Retrieval

The index-vector term dominating this bound is the target of prior memory-reduction work. Two techniques address it. *Index trimming* [BLN⁺09, BK16, TDSG26] shrinks each entry without removing it: during the forward pass it keeps a single bit per index entry in place of its ℓ -bit value, recording only which child each partial collision came from. Once a root match is found, a constrained second pass re-expands these markers into the full leaf indices, at a constant time overhead γ . The hash-value term is thus unchanged while the index-vector term shrinks from $2^{k-1}\ell N$ to $2^{k-1}N$ bits, giving the TIV-IT baseline with peak working memory

$$M_{\text{TIV-IT}} = (P_k \ell + 2^{k-1})N$$

bits. The 2^{k-1} entries themselves remain. *Post-retrieval* [TDSG26] instead removes the entries. In the single-chain algorithm [LF06, BK16] that solves the loose variant LGBP, rather than storing index pointers throughout the forward pass, it discards them and reconstructs the witness after a match by re-running the chain, cutting peak memory from about $2nN$ to nN . As the single chain keeps one list, this re-run is cheap and saves a constant factor. The idea is conceptually transferable to the k -tree, but making it useful there requires a $\Theta(k)$ -factor recovery schedule and exact memory accounting; the gain is worthwhile only at large k , which [TDSG26] do not target. Consequently, no prior work removes the 2^{k-1} index entries in the k -tree setting.

2.5 Cost Model

Bernstein [Ber07] observes that operation counts alone can be a misleading primary metric for generalized-birthday attacks: Wagner-type algorithms are memory-intensive, so their apparent speed may disappear once machine cost is taken into account. The metric he advocates, price-performance (machine cost times running time), is the area-time (AT) product of a circuit, with the area dominated by storage and the time equal to the running time. We adopt a list-level proxy,

$$\text{MT} = M_{\text{work}} \cdot T,$$

in which M_{work} is the peak working memory of the k -tree execution (the storage that dominates the area) and T is the running time, in units of T_0 . An output-inclusive variant is considered in Section 4.4.

Under this metric, TIV and TIV-IT have similar operation counts but both maintain exponentially many index entries, whereas PR-TIV removes these entries from the forward pass, recovering witnesses by hash-directed search rather than stored pointers. Beyond a memory-time trade-off, PR-TIV changes the price-performance optimization problem for the k -tree.

Remark 2 (Scope of the $M \cdot T$ model). A fully layout-aware AT analysis would additionally model sorting networks, communication, routing, and memory-access locality [Ber07]. We do not attempt such an implementation-specific analysis here. Our results are price-performance statements in the standard list-payload model used for Wagner-type attacks. They show that, at this level of abstraction, removing the index vectors changes the optimized MT exponent.

3 Our Technique: Hash-Directed Post-Retrieval

By Proposition 1, the index-vector term $2^{k-1}\ell N$ dominates peak memory for large k . Yet the merge procedure never reads index vectors. They are needed only to trace a root-level match back to its leaf entries. Extending single-chain post-retrieval (Section 2.4) to the

k -tree therefore reduces to one problem: tracing a match without materializing (building and holding in memory, rather than regenerating on demand) index vectors across a 2^k -scale family of tree nodes.

PR-TIV addresses this difficulty by making recovery *hash-directed*.⁴ Given a fixed parent value, it rebuilds the two child subtrees, finds one child pair that produces the target, releases the rebuilt lists, and then recurses on the two child targets. The decomposition relies on two properties of the k -tree.

1. *Index-oblivious merging.* The merge predicate and output hash value depend only on hash values. Therefore, omitting index vectors does not change the hash-value multiset at any node, counted with multiplicities.
2. *Target-directed subtree reuse.* Once a parent target value is fixed, the two child subtrees need only be built long enough to identify one matching child pair. Once such a pair is selected, the child lists are released, and the two recursive recovery calls carry only target hash values and subtree ranges. Thus the same list memory can be reused across sibling subtrees.

Together, these properties yield a two-phase algorithm: a hash-only forward pass followed by a recursive DFS that recovers leaf indices through hash-directed search. In the single-chain algorithm this recovery is cheap: one list is kept throughout, and re-running the deterministic chain recovers the index pointers directly, so discarding their storage saves a constant factor ($2nN \rightarrow nN$). In the k -tree the index vectors are instead exponentially many; PR-TIV removes them by rebuilding sibling subtrees one at a time and locating each child pair by hash, at a $\Theta(k)$ time cost. We continue the notation of Section 2. Recovered indices are streamed in DFS order.

3.1 Algorithm

Let $\text{BuildSubtree}^*(s, d)$ denote the post-order k -tree computation of Section 2, rooted at the subtree spanning the 2^d leaf lists $L_s, \dots, L_{s+2^d-1}$, with index vectors carried alongside hash values. The hash-only variant $\text{BuildSubtree}(s, d)$ omits all index vectors. Both procedures perform d levels of merge_ℓ and return a list at global level d . Algorithm 1 gives the pure post-retrieval procedure. Throughout the algorithm, lists are multisets, and a pair set counts each occurrence of a value separately.

List representation Each list produced by BuildSubtree is a flat array of its $N/2$ hash values and stores no index vectors, so its footprint is exactly the output-list payload; the peak of the call is $F(d)$. The forward merges, root-set construction, and constrained search each use a hash join, indexing one operand and streaming the other in expected $O(N)$ time. Following the convention of Section 2, the lookup table is a constant-factor overhead and is not counted in the peak-memory bounds.

Hash-Directed Search The subroutine $\text{RecoverSubtree}(y, s, d)$ takes a target hash value y , a starting leaf index s , and a subtree depth d , and outputs leaf indices whose merged hash value is y . Its search step is a target-constrained merge: given target y and child lists $\hat{L}, \hat{R}$, find a pair (y_L, y_R) such that

$$\text{LSB}_\ell(y_L \oplus y_R) = 0 \quad \text{and} \quad (y_L \oplus y_R) \gg \ell = y.$$

In the XOR instantiation, each $y_L \in \hat{L}$ determines a unique complement

$$y_R = y_L \oplus (y \ll \ell).$$

⁴A pointer-rederivation implementation is possible, but our hash-directed recovery reuses the hashing already performed in the forward pass and avoids separate pointer machinery.

Algorithm 1: Pure PR-TIV, denoted PR-TIV₀

Input: A hash-generated RGBP instance with $K = 2^k$ leaf lists.
Output: Streamed leaf indices of one solution, or failure.

```

1  $L \leftarrow \text{BuildSubtree}(0, k-1);$ 
2  $R \leftarrow \text{BuildSubtree}(2^{k-1}, k-1);$ 
3  $C \leftarrow \{(y_L, y_R) \in L \times R \mid y_L = y_R\};$  //  $L$  and  $R$  are released
4 if  $C = \emptyset$  then return failure;
5 foreach  $(y_L, y_R) \in C$  do
6    $\text{RecoverSubtree}(y_L, 0, k-1);$ 
7    $\text{RecoverSubtree}(y_R, 2^{k-1}, k-1);$ 
8   return solution list;

9 Subroutine  $\text{RecoverSubtree}(y, s, d)$ 
10   if  $d = 0$  then output the index  $j$  such that  $\mathcal{H}(s, j) = y$ ; return;
11    $\hat{L} \leftarrow \text{BuildSubtree}(s, d-1);$ 
12    $\hat{R} \leftarrow \text{BuildSubtree}(s + 2^{d-1}, d-1);$ 
13   Find  $(y_L, y_R) \in \hat{L} \times \hat{R}$  such that  $\text{LSB}_\ell(y_L \oplus y_R) = 0$  and  $(y_L \oplus y_R) \gg \ell = y$ ;
14   Release  $\hat{L}$  and  $\hat{R}$ ;
15    $\text{RecoverSubtree}(y_L, s, d-1);$ 
16    $\text{RecoverSubtree}(y_R, s + 2^{d-1}, d-1);$ 

```

A hash table on $\hat{R}$ resolves each lookup in expected $O(1)$ time, so the search costs $O(N)$ expected time. For interval merges, as in Remark 1, each y_L constrains y_R to a contiguous range. Scanning a sorted range representation of $\hat{R}$ gives the same expected $O(N)$ cost. If multiple matching pairs exist, the procedure chooses one arbitrarily. Correctness is unaffected by this choice, as shown in Lemma 2. The root condition $y_L = y_R$ is the $\text{merge}_{2\ell}$ zero test. If a value has a occurrences in L and b occurrences in R , it yields ab root pairs. The bounds below are stated per selected root pair.

Worked Example Figures 3 and 4 trace PR-TIV on a $k=3$ instance ($K=8$). The forward pass in Figure 3(a) builds the two depth-2 hash-only child lists L, R and releases them once a root pair $y_L = y_R$ is found. Recovery in Figure 3(b) then descends each half in the depth-first order of the numbered markers, rebuilding and releasing the two child lists of each node one at a time, so the same workspace is reused across siblings (shaded regions). As the memory timeline in Figure 4 shows, recovery peaks at $F(k-1)$ (defined in the proof of Theorem 1) and returns to zero between sibling subtrees, so it never exceeds the forward-pass peak.

3.2 Correctness and Complexity

Lemma 1. *For all s and d , the hash-value multiset produced by $\text{BuildSubtree}(s, d)$ is equal to the projection of the output of $\text{BuildSubtree}^*(s, d)$ onto its hash-value component.*

Proof. By induction on d . For $d = 0$, both procedures scan the same leaf list, differing only in whether the index component is retained. For $d > 0$, the merge predicate and output hash value depend only on the two child hash values, which the hypothesis matches. The full-index procedure merely appends the child index vectors, leaving the hash-value multiset unchanged. Hence the parent multiset agrees, with multiplicities. $\square$

Lemma 2. *If y occurs in the output multiset of $\text{BuildSubtree}(s, d)$, then $\text{RecoverSubtree}(y, s, d)$ outputs leaf indices that produce it.*

Proof. By induction on d . For $d = 0$, membership of y means that some leaf entry $\mathcal{H}(s, j)$ equals y , so the scan in RecoverSubtree can output such a j . For $d > 0$, membership of y in $\text{BuildSubtree}(s, d)$ means, by the definition of merge_ℓ , that some pair

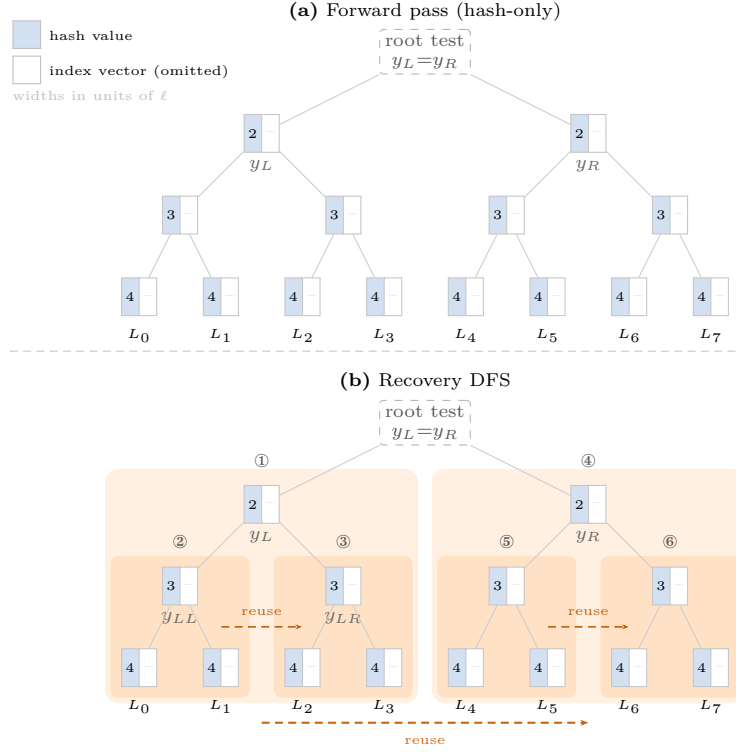


Figure 3: Worked example of PR-TIV on a $k=3$ tree. Shaded regions and dashed arrows indicate reuse of working memory across sibling subtrees.

$y_L \in \text{BuildSubtree}(s, d-1)$ and $y_R \in \text{BuildSubtree}(s + 2^{d-1}, d-1)$ merges to y . Recovery rebuilds exactly these two child lists, so its search can choose such a pair. The induction hypothesis recovers witnesses for y_L and y_R in the two disjoint child subtrees, and their concatenation gives leaf indices whose merged hash value is y . $\square$

Theorem 1 (Pure PR-TIV). *The forward pass of PR-TIV₀ induces the same multiset of root matches as the full-index k -tree after projecting away index vectors, so the two have identical success probability. Conditioned on a root match, PR-TIV₀ recovers valid leaf indices with peak working memory*

$$M_0 = P_k \ell N = \frac{k^2 + 5k + 2}{4} \ell N$$

in the expected-list-size accounting of Section 2, and time $\Theta((k+1)T_0)$, where $T_0 = 2^k N/2$.

Proof. Correctness. By Lemma 1, the two depth- $(k-1)$ lists produced in the forward pass have exactly the same hash-value multisets as the corresponding full-index lists after projection. The root test $y_L = y_R$ is the $\text{merge}_{2\ell}$ zero condition, so the root-pair multiset, and hence the success probability, is the same as in the full-index execution. Conditioned on a selected root pair (y_L, y_R) , Lemma 2 applied to the left and right depth- $(k-1)$ subtrees recovers witnesses for both child values. Their concatenation is a full RGBP solution because $y_L \oplus y_R = 0$ at the root.

Memory. Let $F(d)$ be the peak working memory of a $\text{BuildSubtree}(\cdot, d)$ call, and let $c(e) = \frac{N}{2}(k+1-e)\ell$ be the payload of one level- e list ($N/2$ items of $(k+1-e)\ell$ hash bits each).⁵ Building a depth- d subtree ($d \geq 1$) proceeds in three stages: (i) build the left child,

⁵Here $F(0) = c(0) = \frac{N}{2}(k+1)\ell$ is a single hash-only leaf list. The full-index k -tree of Proposition 1

peaking at $F(d-1)$; (ii) build the right child while the completed left output $c(d-1)$ stays in memory, peaking at $c(d-1) + F(d-1)$; (iii) the merge, which under the stack model of Section 2 replaces the two child lists by the single depth- d output $c(d)$ and introduces no new maximum. Since $c(d-1) > 0$, stage (ii) dominates, so the peak obeys the recurrence

$$F(0) = c(0) = \frac{N}{2}(k+1)\ell, \quad F(d) = F(d-1) + \frac{N}{2}(k+2-d)\ell \quad (d \geq 1),$$

whose increment $\frac{N}{2}(k+2-d)\ell = c(d-1)$ is the held left child. Every increment is positive for $d \leq k$, so F is strictly increasing, and telescoping gives the closed form

$$F(d) = \frac{N}{2}\ell \left[(k+1) + \sum_{i=1}^d (k+2-i) \right] = \frac{N}{2}\ell \left[(k+1) + d(k+2) - \frac{d(d+1)}{2} \right];$$

in particular $F(k) = \frac{k^2 + 5k + 2}{4} \ell N = P_k \ell N$. Equivalently, $F(d) = H_{k,d} \ell N$ with $H_{k,d} = (k+1) + \frac{(2k+2-d)(d-1)}{4}$ (the coefficient reused in Theorem 2), so that $P_k = H_{k,k}$. The forward pass is stage (ii) at the root: it holds the completed depth- $(k-1)$ list L (item width 2ℓ) while building R , so its peak is $F(k-1) + \frac{N}{2} \cdot 2\ell = F(k)$.

For recovery at depth d , rebuilding the two child subtrees peaks at $F(d)$; the matching pair is then selected and both children released before the two sequential depth- $(d-1)$ calls, whose peak is at most $F(d-1) < F(d)$. The overall peak is therefore $\max\{F(k), F(k-1)\} = F(k) = P_k \ell N$.

In the random-list model of [MS12], each intermediate list concentrates around its expected size 2^ℓ . In the asymptotic regimes considered here, standard Chernoff and union-bound concentration over the $O(k2^k)$ intermediate lists bounds the realized peak by $(1+o(1))P_k \ell N$ with high probability.

Time. Let $T(d)$ be the running time of a depth- d recovery call. Rebuilding the two depth- $(d-1)$ child subtrees costs $2 \cdot 2^{d-1}N/2 = 2^d N/2$; two recursive recovery calls add $2T(d-1)$. Thus $T(0) = N/2$ and $T(d) = 2^d N/2 + 2T(d-1)$. Setting $T(d) = f(d) \cdot 2^d N/2$ gives $f(0) = 1$, $f(d) = 1 + f(d-1)$, so $f(d) = d + 1$. The two root children cost $2T(k-1) = kT_0$; adding the forward pass gives $(k+1)T_0$. $\square$

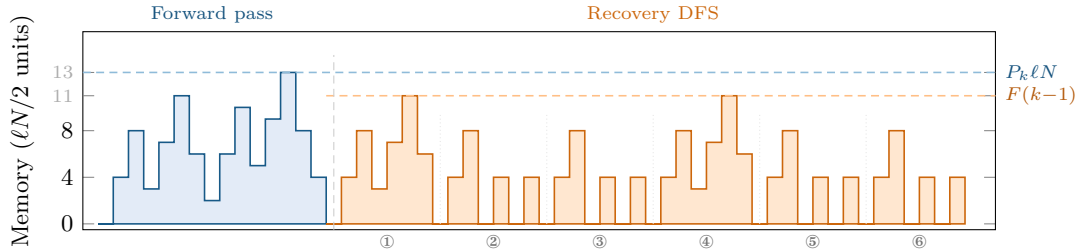


Figure 4: Memory timeline for the $k=3$ worked example of Figure 3, measured in units of $\ell N/2$. The forward pass peaks at $P_k \ell N = 13 \cdot \ell N/2$, while the recovery DFS peaks at $F(k-1) = 11 \cdot \ell N/2$ and returns to zero between sibling subtrees. Here $F(d)$ denotes the peak working memory of a $\text{BuildSubtree}(\cdot, d)$ call, as defined in the proof of Theorem 1.

Remark 3 (Multiple root matches). Each root match is already a full solution, so the preimage attacks of this paper recover one tuple and halt at the first match. More generally, if the forward pass yields $r = O(1)$ root matches, as in expectation for the root test, all

additionally stores a $2^0 \ell$ -bit index per item, so its level-0 list has payload $\frac{N}{2}(k+2)\ell$; the difference is exactly the index vector that PR-TIV omits. At the stack peak, two level-0 lists are held simultaneously, and the first increment $c(0)$ in the recurrence accounts for the second one.

r recoveries can share a single rebuilding pass by carrying the target values through the DFS. The subtree rebuilds dominate the r -dependent bookkeeping and output cost in this constant-root-match regime, so the total time remains $(k+1)T_0$.

Remark 4 (Parallel recovery). The sequential recovery reuses memory across siblings. When memory is not the binding constraint, sibling subtrees can instead be recovered in parallel. With p processors, peak memory scales by p and recovery time by $1/p$, leaving the MT product invariant.

3.3 δ -Level Caching

Pure PR-TIV pays a $(k+1) \times$ time factor. Caching index vectors near the leaves trades some of this factor for memory. For $\delta \in \{0, 1, \dots, k-1\}$, define PR-TIV $_\delta$ to switch to full-index computation at depth δ : when $d \leq \delta$, the recovery procedure computes BuildSubtree $^*(s, d)$ and directly outputs the index vector of the matching occurrence. At $\delta = 0$ this is Algorithm 1 (pure post-retrieval). By convention, PR-TIV $_k$ denotes the direct full-index execution TIV, for which the formulas below reduce to Proposition 1. Write $\alpha_\delta = k - \delta + 1$ for the resulting time factor, so that the total time is $\alpha_\delta T_0$; in particular $\alpha_0 = k + 1$.

Figure 5 illustrates the depth- δ boundary for $k = 4$ and $\delta = 2$.

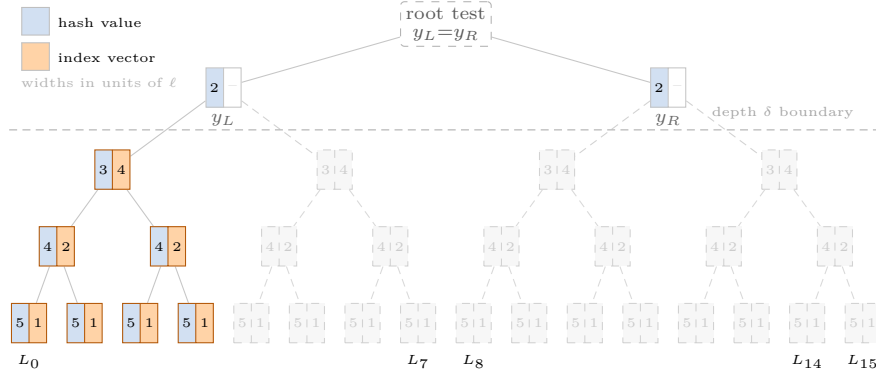


Figure 5: Depth- δ boundary for PR-TIV $_\delta$ with $k = 4$ and $\delta = 2$. The upper recovery is hash-only. The highlighted depth- δ subtree is rebuilt with full index vectors.

Theorem 2 (δ -level caching). *For $1 \leq \delta \leq k$, PR-TIV $_\delta$ recovers one root solution in expected time $\Theta((k-\delta+1)T_0)$ and expected peak working memory*

$$M_\delta = \max\{P_k \ell N, (H_{k,\delta} + 2^{\delta-1}) \ell N\}, \quad H_{k,\delta} = (k+1) + \frac{(2k+2-\delta)(\delta-1)}{4}.$$

Proof. The endpoint $\delta = k$ is exactly the full-index execution of Proposition 1, for which the displayed formulas give time T_0 and memory $(P_k + 2^{k-1}) \ell N$. Assume below that $1 \leq \delta < k$.

Time. The recurrence of Theorem 1 applies with base case $f(\delta) = 1$ instead of $f(0) = 1$, giving recovery cost $(k-\delta)T_0$ plus one forward pass, for a total of $(k-\delta+1)T_0$.

Memory. The forward-pass peak remains $F(k) = P_k \ell N$. The only new configuration arises when recovery reaches depth δ and runs BuildSubtree $^*(\cdot, \delta)$. The peak level structure is the same as in the hash-only case, so the hash contribution equals $F(\delta)$, which by the closed form for F in the proof of Theorem 1 equals $H_{k,\delta} \ell N$. The index-vector contribution at the same peak configuration is

$$\frac{N}{2} \left(2\ell + \sum_{i=1}^{\delta-1} 2^i \ell \right) = 2^{\delta-1} \ell N.$$

Cached subtrees are processed one at a time and released before recursing, so the peak is $\max\{P_k \ell N, (H_{k,\delta} + 2^{\delta-1}) \ell N\}$. $\square$

Free Caching Range Under the list-payload peak-memory model of Section 2, caching is *free* in the precise sense of adding zero additional list-payload bits: it lowers the time factor without increasing the modeled peak when $H_{k,\delta} + 2^{\delta-1} \leq P_k$, i.e.,

$$2^{\delta-1} \leq \frac{(k-\delta)(k-\delta+3)}{4}.$$

For small δ/k , this is roughly $\delta \leq 2 \log_2 k - 1$. More precisely, for each k we define $\delta_{\text{free}}(k)$ as the largest δ satisfying the above inequality. Thus free caching saves $\Theta(\log k)$ levels of the $(k+1)$ time factor at zero modeled list-payload cost.

Table 2 lists δ_{free} for representative k values.

Table 2: Free caching parameters for selected k values. $\delta_f = \delta_{\text{free}}(k)$; $\alpha_0 = k+1$ is the pure PR-TIV time factor; $\alpha_{\delta_f} = k - \delta_f + 1$ is the time factor after free caching; “time saved” is $1 - \alpha_{\delta_f} / \alpha_0$. The cap rows use the limited-block depth bounds $k = 25, 28$ from Section 5.

Role	k	δ_f	α_0	α_{δ_f}	time saved
Low-depth reference	15	5	16	11	31%
iSHAKE-128 cap	25	7	26	19	27%
iSHAKE-256 cap	28	7	29	22	24%
iSHAKE-128 TIV opt.	64	10	65	55	15%
iSHAKE-128 PR-TIV opt.	89	11	90	79	12%
iSHAKE-256 TIV opt.	128	12	129	117	9%
iSHAKE-256 PR-TIV opt.	179	13	180	167	7%

MT-Optimal Caching Depth Within the free range, larger δ is strictly better because memory is constant and α_δ decreases. Beyond that range, Lemma 3 (Appendix A) shows that the non-free optimum is the first non-free depth, so the MT-optimal depth is either δ_f or $\delta_f + 1$.

Proposition 2 (MT-optimal δ). *For fixed $k \geq 8$, put $\delta_f = \delta_{\text{free}}(k)$ and $m = k - \delta_f$. An MT-minimizing caching depth may be chosen as*

$$\delta^* = \begin{cases} \delta_f & \text{if } \left(2^{\delta_f} - \frac{(m-1)(m+2)}{4}\right) m \geq P_k, \\ \delta_f + 1 & \text{if } \left(2^{\delta_f} - \frac{(m-1)(m+2)}{4}\right) m < P_k. \end{cases}$$

Proof. Among free depths ($\delta \leq \delta_f$), $M_\delta = P_k \ell N$ is constant and $\alpha_\delta = k - \delta + 1$ decreases, so δ_f is optimal. Among non-free depths ($\delta_f + 1 \leq \delta \leq k - 1$), Lemma 3 shows that $\text{MT}(\delta)$ is minimized at $\delta_f + 1$. (Full caching $\delta = k$ is the TIV baseline of Proposition 1, outside the PR-TIV family.) Hence $\delta^* \in \{\delta_f, \delta_f + 1\}$. Comparing these two depths, $\delta_f + 1$ is better iff $\text{MT}(\delta_f + 1) < \text{MT}(\delta_f)$, i.e.,

$$(H_{k,\delta_f+1} + 2^{\delta_f})(k - \delta_f) < P_k(k - \delta_f + 1).$$

Substituting $H_{k,\delta_f+1} = P_k - (m-1)(m+2)/4$ gives the displayed dichotomy. In the equality case we choose the free depth δ_f . $\square$

Remark 5 (iSHAKE parameters). For the iSHAKE working-memory parameters evaluated in Section 5, the criterion in Proposition 2 always selects the first branch. Thus the reported working-memory PR-TIV optima use the free depth $\delta^* = \delta_f$.

3.4 Summary

The central mechanism of PR-TIV is that recovery carries only scalar hash targets down the recursion, never index vectors. Sibling subtrees are rebuilt and released one at a time, so the same working memory is reused across the DFS. Table 1 summarizes the resulting list-item reductions. Which trade-off dominates depends on the MT product analyzed in Section 4.

4 Memory-Time Analysis

Using the cost model of Section 2.5, this section compares the MT product across the three k -tree variants. We continue the notation of Sections 2 and 3. In particular, $\ell = n/(k+1)$, $N = 2^{\ell+1}$, $T_0 = 2^{k+\ell}$, and $P_k = (k^2+5k+2)/4$. For PR-TIV, $\alpha_\delta = k-\delta+1$ is the time factor (Section 3.3). For pure post-retrieval, $\alpha_0 = k+1$.

4.1 Global MT-Optimal Parameters

Table 1 summarizes the working-memory and time expressions. The TIV-IT overhead γ (Section 2.4) satisfies $1 < \gamma \leq 2$, as a full reconstruction is one extra pass ($\gamma \approx 2$ [TDSG26]). Our conservative $\gamma = 3/2$ only understates the baseline time γT_0 and cannot inflate PR-TIV's advantage. Over this range it moves the TIV-IT curves by $\log_2 \gamma \leq 1$ bit, and PR-TIV is γ -independent, so the asymptotic gap $(4-2\sqrt{2})\sqrt{n}$ is unaffected.

For TIV and TIV-IT, the 2^{k-1} index entries couple the memory and time exponents, giving $\log_2 \text{MT} = 2k + 2\ell + O(\log n)$ with $2\ell = 2n/(k+1)$, minimized at $k^* = \sqrt{n} + O(1)$. For PR-TIV, removing them reduces the leading exponent to $k + 2\ell$, minimized at $k_{\text{PR-TIV}}^* = \sqrt{2n} + O(1)$.

Theorem 3 (Asymptotic MT optimum). *In the continuous-exponent convention:*

(a) Working memory.

$$\log_2 \text{MT}^* = \begin{cases} 4\sqrt{n} + O(\log n) & \text{TIV/TIV-IT,} \\ 2\sqrt{2n} + O(\log n) & \text{PR-TIV.} \end{cases}$$

(b) Output-inclusive ($r = O(1)$).

$$\log_2 \text{MT}_{\text{out}}^* = \begin{cases} 4\sqrt{n} + O(\log n) & \text{TIV/TIV-IT,} \\ 3\sqrt{n} + O(\log n) & \text{PR-TIV.} \end{cases}$$

Proof. Part (a). For TIV and TIV-IT, the leading exponent $2k + 2n/(k+1)$ is minimized at $k+1 = \sqrt{n}$, giving $4\sqrt{n} + O(1)$; the factors P_k , ℓ , and γ contribute $O(\log n)$. For PR-TIV, the leading exponent $k + 2n/(k+1)$ is minimized at $k+1 = \sqrt{2n}$, giving $2\sqrt{2n} + O(1)$; the factors P_k , ℓ , and α_δ contribute $O(\log n)$.

Part (b). For TIV and TIV-IT, the witness size $2^k \ell$ is dominated by the $2^{k-1} \ell N$ index-vector term in peak memory, so output inclusion does not change the leading exponent. For PR-TIV with $r = O(1)$, peak memory is $\max\{P_k \ell N, r 2^k \ell\}$; the leading exponents are ℓ and k respectively. The working-memory term dominates when $\ell > k$ and the witness dominates when $k > \ell$. The MT exponent is $k + \ell + \max\{k, \ell\}$, minimized at $k = \ell$, i.e., $k+1 = \sqrt{n}$; this gives $\log_2 \text{MT} = 3\sqrt{n} + O(\log n)$. $\square$

The gain has two sources: at a fixed large k , PR-TIV removes the 2^{k-1} index entries from peak working memory; this removal then shifts the MT-optimal k from the time-optimal scale $\sqrt{n}$ up to $\sqrt{2n}$. The leading asymptotic gaps are

$$\log_2 \text{MT}_{\text{TIV/TIV-IT}}^* - \log_2 \text{MT}_{\text{PR-TIV}}^* = \begin{cases} (4 - 2\sqrt{2})\sqrt{n} + O(\log n) & (\text{working memory}), \\ \sqrt{n} + O(\log n) & (\text{output-inclusive}). \end{cases}$$

4.2 Pointwise Crossover

Global optimality does not entail pointwise dominance: PR-TIV need not win at every fixed k . Write $C_X(k) = \text{MT}_X/(NT_0)$ for the fixed- k cost after removing the common factor NT_0 . Figure 6 plots this normalized comparison, including both pure PR-TIV ($\delta = 0$) and the best free-caching choice $\delta_{\text{free}}(k)$. Free δ -level caching lowers the PR-TIV curve by replacing $k+1$ with $k - \delta_{\text{free}}(k) + 1$.

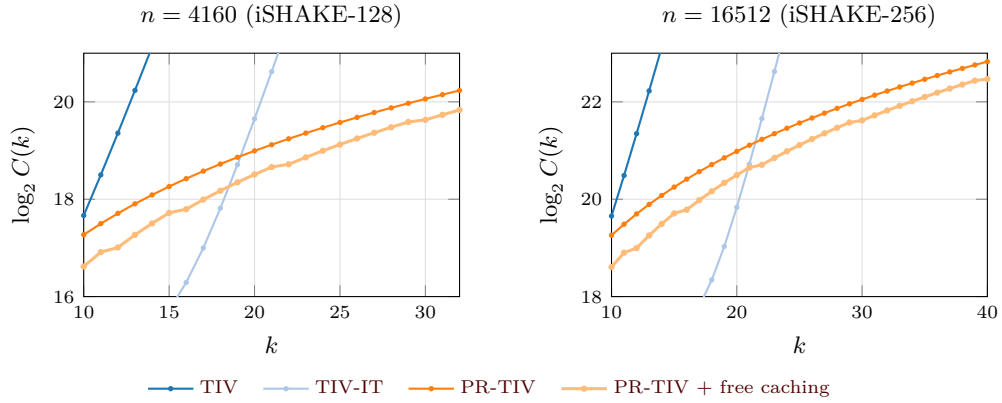


Figure 6: Pointwise MT comparison at fixed k ($\gamma = 3/2$; common factor NT_0 omitted). The free-caching PR-TIV curve uses the largest free-caching depth $\delta_{\text{free}}(k)$. Free caching shifts the first integer k where PR-TIV beats both baselines from 20 to 19 for $n = 4160$, and from 22 to 21 for $n = 16512$.

Algebraically, the normalized costs in the figure are

$$\begin{aligned} C_{\text{TIV}}(k) &= (P_k + 2^{k-1})\ell, \\ C_{\text{TIV-IT}}(k) &= \gamma(P_k\ell + 2^{k-1}), \\ C_{\text{PR-TIV}}(k) &= \alpha_\delta P_k\ell. \end{aligned}$$

PR-TIV pays a time factor α_δ , so the exponential number of index entries must be large enough to compensate. At a fixed k , PR-TIV improves over TIV when $2^{k-1} > (\alpha_\delta - 1)P_k$, and over TIV-IT when $2^{k-1} > (\alpha_\delta/\gamma - 1)P_k\ell$ (nonpositive when $\alpha_\delta \leq \gamma$). Thus PR-TIV simultaneously improves over both baselines whenever

$$2^{k-1} > \max \left\{ (\alpha_\delta - 1)P_k, \left(\frac{\alpha_\delta}{\gamma} - 1 \right) P_k\ell \right\}.$$

For the parameter ranges used in our evaluations, the second term dominates the maximum.

4.3 Combination with List-Size Reduction

List-size reduction (LSR) [Ber07, MS12, NS15, Din19] is the other major memory-optimization direction for the k -tree: shrink the leaf lists and offset the resulting drop in success probability by repeating independent runs. Tang et al. [TDSG26] pose the combination of LIR and LSR as an open problem.

Setup For an integer reduction parameter $s \geq 0$ with $s < \ell$, reduce every leaf list from size 2^ℓ to size $2^{\ell-s}$ and set each non-root collision width to $\ell' = \ell - s$, so that intermediate lists maintain expected size $2^{\ell-s}$ after every merge. A single run then succeeds with probability $2^{-(k+1)s}$, so the repeated forward cost is $T_{\text{fwd}}(s) = 2^{(k+1)s} \cdot 2^{-s} T_0 = 2^{ks} T_0$. Post-retrieval or index reconstruction, when present, is charged only on the successful repetition.

Under LSR, hash widths at level i satisfy $n - i\ell' = (k+1-i)\ell + is \geq (k+1-i)\ell$, while list sizes scale by 2^{-s} . Hence every variant satisfies $M(s) \geq 2^{-s} P_k \ell N$.

Proposition 3 (LSR does not improve MT). *For the k -tree with $k \geq 3$ and an integer reduction parameter $s \in \mathbb{Z}_{\geq 1}$ satisfying $s < \ell$, equal list-size reduction at every level gives a strictly larger MT product than $s = 0$ for TIV, for TIV-IT with $\gamma = 3/2$, and for PR-TIV $_\delta$ with $\delta \leq \delta_{\text{free}}(k)$. Consequently, joint optimization over the range $k \geq 3$ with direct LSR selects $s^* = 0$.*

Proof. We use the hash-value stack as a common memory lower bound. Any remaining index-vector memory only strengthens the inequality. In all three cases $T(s) \geq 2^{ks} T_0$ and $M(s) \geq 2^{-s} P_k \ell N$.

TIV. $M_{\text{TIV}}(0) = (P_k + 2^{k-1})\ell N$ and $T_{\text{TIV}}(0) = T_0$, so

$$\frac{MT_{\text{TIV}}(s)}{MT_{\text{TIV}}(0)} \geq \frac{2^{(k-1)s} P_k}{P_k + 2^{k-1}} \geq \frac{2^{k-1} P_k}{P_k + 2^{k-1}} > 1 \quad (k \geq 3).$$

TIV-IT. $M_{\text{TIV-IT}}(0) = (P_k \ell + 2^{k-1})N$ and $T_{\text{TIV-IT}}(0) = \gamma T_0$ with $\gamma = 3/2$, so

$$\frac{MT_{\text{TIV-IT}}(s)}{MT_{\text{TIV-IT}}(0)} \geq \frac{2^{(k-1)s} P_k \ell}{\gamma(P_k \ell + 2^{k-1})} \geq \frac{2^{k-1} P_k}{\gamma(P_k + 2^{k-1})} > 1 \quad (k \geq 3).$$

PR-TIV $_\delta$. With $\alpha_\delta = k - \delta + 1$, recovery is charged only on the successful repetition, giving $T_{\text{PR-TIV}_\delta}(s) = (2^{ks} + (\alpha_\delta - 1)2^{-s})T_0$. Since $\delta \leq \delta_{\text{free}}(k)$, $M_{\text{PR-TIV}_\delta}(0) = P_k \ell N$, so

$$\frac{MT_{\text{PR-TIV}_\delta}(s)}{MT_{\text{PR-TIV}_\delta}(0)} \geq \frac{2^{(k-1)s} + (\alpha_\delta - 1)2^{-2s}}{\alpha_\delta} \geq \frac{2^{k-1} + k/4}{k+1} > 1 \quad (k \geq 3).$$

The middle inequality first minimizes the ratio over $\alpha_\delta \leq k+1$: since $2^{(k-1)s} > 2^{-2s}$, the ratio decreases in α_δ , so the minimum over α_δ is at $\alpha_\delta = k+1$. For integer $s \geq 1$ and $k \geq 3$, the resulting expression is increasing in s , so setting $s = 1$ gives the final bound. $\square$

4.4 Output-Inclusive Accounting

So far M has counted working memory only. Under the *output-inclusive* convention (abbreviated “out.” in the tables), if r complete witnesses must be held simultaneously, the output itself can dominate the peak footprint. Each witness contains 2^k leaf indices of ℓ bits each, so the output-inclusive peak is $\max\{M_\delta, r2^k \ell\}$.

Remark 6 (Output-inclusive witness). A solution names 2^k leaf indices in both the k -tree and the single-chain algorithm, so its exponential output size is intrinsic to the problem, not to PR-TIV. The single-chain post-retrieval of [TDSG26] counts working memory only, streaming the solution rather than holding it, and so omits this term. We track it because we aim at a concrete price-performance (MT) accounting: once PR-TIV drives working memory down to $P_k \ell N$, the output $r2^k \ell$ can become the binding cost, which a complete model must include.

Expanding M_δ (Theorem 2),

$$M_{\text{PR-TIV}_\delta}^{\text{out}}(r) = \max\{P_k \ell N, (H_{k,\delta} + 2^{\delta-1})\ell N, r2^k \ell\},$$

which for pure PR-TIV ($\delta = 0$) specializes to $\max\{P_k \ell N, r 2^k \ell\}$. For $1 \leq \delta \leq k$, the list-memory part dominates the recovered witnesses exactly when

$$k + \log_2 r \leq \ell + 1 + \log_2 \max\{P_k, H_{k,\delta} + 2^{\delta-1}\}.$$

When this fails, the MT comparison must be recomputed with $M_{\text{PR-TIV}_\delta}^{\text{out}}(r)$.

For $r = O(1)$ and pure PR-TIV, the witness exponent exceeds the working-memory exponent at $k^* \approx \sqrt{2n}$; the output-inclusive convention shifts the optimum to $k \approx \sqrt{n}$ and the gap narrows from $(4-2\sqrt{2})\sqrt{n}$ to $\sqrt{n}$ (Theorem 3(b)).

In the output-inclusive regime, the witness term may also extend the free caching range: caching levels $\delta > \delta_{\text{free}}$ that would raise working memory above $P_k \ell N$ remain free as long as $(H_{k,\delta} + 2^{\delta-1})\ell N \leq r 2^k \ell$. Section 5 applies both conventions to iSHAKE.

5 Application to Incremental Hashing

This section instantiates the MT comparison for fixed-size iSHAKE preimage attacks, the large- k benchmark of prior analyses [MGS16, TSG26, TDSG26].

5.1 Setup

iSHAKE is an incremental hash based on SHAKE in which randomized block hashes are combined in $(\mathbb{Z}/2^{64}\mathbb{Z})^{n/64}$ [MGS16]. The k -tree merge extends unchanged to this group (Remark 1). In the fixed-size setting with blocks $M_1, \dots, M_K$ at prescribed positions, a preimage attack chooses one candidate per position, giving a hash-generated RGBP instance with leaf generator $\mathcal{H}(i, j) = H(M_{i,j} \| \langle i \rangle)$ [TSG26]. Since the attacker can define each candidate block $M_{i,j}$ by an efficient function of the index j (e.g., $M_{i,j} = j$), the leaf index alone recovers the message block, so PR-TIV applies without persistent leaf storage.

The *unlimited-block* mode ($K_{\max} = \infty$ in prior work) uses the maximum output lengths: $n = 4160$ for iSHAKE-128 and $n = 16512$ for iSHAKE-256. The *limited-block* mode uses the minimum output lengths: $n = 2688$ with $K_{\max} = 2^{25}$ for iSHAKE-128 and $n = 6528$ with $K_{\max} = 2^{28}$ for iSHAKE-256, capping the tree depth at $k \leq 25$ and $k \leq 28$, respectively.

5.2 Results

Table 3 gives the MT optima obtained by enumerating integer k .

PR-TIV reduces $\log_2$ MT by 51–122 bits over TIV-IT with unlimited blocks and by 6–8 bits under the block caps (12–15 bits over TIV).

Figure 1 plots the full $\log_2$ MT landscape for the iSHAKE-256 unlimited-block instance.

Output-Inclusive Materializing the witness lowers the optimal k (from 89 to 70 for iSHAKE-128 and from 179 to 135 for iSHAKE-256), adds 5–15 bits to $\log_2$ MT, and leaves the limited-block optima unchanged because the witness is at most 2^{36} bits.

5.3 Design Implications

PR-TIV is an *attack-instance* property: it applies when the concrete attack is a hash-generated RGBP instance whose leaves can be regenerated and whose group supports the partial-collision merge of Remark 1.

Concretely, PR-TIV needs all of the following.

1. *Publicly regenerable leaves.* For every leaf list and index j , the attacker can recompute the corresponding group element from a public oracle. Keyed multiset hashes such as MSet-XOR-Hash and MSet-Add-Hash [CDvD⁺03] fail this condition for ordinary external attackers.

Table 3: MT estimates for fixed-size iSHAKE preimage attacks. TIV-IT uses $\gamma = 3/2$; PR-TIV uses $\delta = \delta_{\text{free}}(k)$ for working-memory rows; “out.” rows use the output-inclusive convention of Section 4.4 with $r = 1$ and may use $\delta > \delta_{\text{free}}$ when the witness term dominates peak memory.

Instance	Algorithm	k	ℓ	δ	$\log_2 T$	$\log_2 M$	$\log_2 \text{MT}$	Δ_{TIV}	$\Delta_{\text{TIV-IT}}$
iSHAKE-128, $n = 4160$ unlimited blocks	TIV [Wag02]	64	64.0	—	128.0	134.0	262.0	—	—
	TIV-IT [TDSG26]	63	65.0	—	128.6	128.0	256.6	−5.4	—
	PR-TIV	89	46.2	11	141.5	63.8	205.3	−56.7	−51.3
	PR-TIV (out.)	70	58.6	10	134.5	75.9	210.4	−51.6	−46.2
iSHAKE-256, $n = 16512$ unlimited blocks	TIV	128	128.0	—	256.0	263.0	519.0	—	—
	TIV-IT	127	129.0	—	256.6	256.0	512.6	−6.4	—
	PR-TIV	179	91.7	13	278.1	112.3	390.4	−128.6	−122.2
	PR-TIV (out.)	135	121.4	13	263.4	141.9	405.3	−113.7	−107.3
iSHAKE-128, $n = 2688$ $k \leq 25$	TIV	25	103.4	—	128.4	135.1	263.5	—	—
	TIV-IT	25	103.4	—	129.0	128.4	257.4	−6.1	—
	PR-TIV	25	103.4	7	132.6	118.6	251.3	−12.2	−6.1
iSHAKE-256, $n = 6528$ $k \leq 28$	TIV	28	225.1	—	253.1	260.9	514.0	—	—
	TIV-IT	28	225.1	—	253.7	253.1	506.8	−7.2	—
	PR-TIV	28	225.1	7	257.6	241.8	499.3	−14.7	−7.5

2. *Independent RGBP lists.* The attack must select one candidate from each fixed block position. Fixed-size iSHAKE has this form [MGS16], as do fixed-position additive variants of AdHash and LtHash [BM97]. Set, multiset, variable-size collision, and pair-chaining settings usually give an LGBP or another dependent subset-sum instance instead [CDvD⁺03, GSC01, PW06].
3. *Wagner-compatible group representation.* The group operation must admit the quotient or partial-collision tests and hash-directed child search used by the k -tree. This holds directly for XOR and for vector groups such as $(\mathbb{Z}/2^w\mathbb{Z})^m$; for scalar modular addition it requires the interval-based treatment of [TSG26]. Multiplicative groups and prime-order elliptic-curve groups are useful for homomorphic hashing [BM97, Bro08, MSTA17], but homomorphism alone does not provide the quotient chain needed by the merge.

The applicability conditions are preserved for common public position-dependent variants, such as $\sum f(i) \cdot H(M_i \| \langle i \rangle)$ with public units $f(i)$ or $\sum H(M_i) \cdot x^i$ with public unit x : the scalar attached to each position is public and invertible, so the leaf oracle remains independently queryable and the Wagner merge is preserved.

Scope Table 5 (Appendix B) records representative cases, classified not by whether the construction is incremental but by which of the three conditions is absent. For the positive cases, the group-element size n spans an order of magnitude: AdHash operates in a scalar group ($n = \lceil \log_2 M \rceil$, with $n \geq 1600$ needed for 80-bit collision resistance [LKMW19]), while LtHash and iSHAKE use high-dimensional vector groups ($n = 10^3$ – 10^4 bits).

In its deployed form, LtHash hashes a *dynamic set* rather than a fixed-position sequence. Both Meta’s Ledger Account Database and Solana’s Accounts Lattice Hash maintain a digest $\sum_{r \in S} h(r)$ over a mutable record set S , updated incrementally via subtract-old/add-new. A preimage attacker chooses which records to include and how many. Neither the number nor the identity of summands is fixed, so the resulting problem is a variable-size subset-sum instance rather than an independent-list RGBP.

Crossover and Selection For positive-case constructions, the practical impact depends on the maximum tree depth k_{max} relative to two thresholds: the pointwise crossover depth $k_{\text{cross}}(n)$ from Section 4.2 and the MT-optimal depth $k_{\text{PR-TIV}}^* \approx \sqrt{2n}$ from Theorem 3. When $k_{\text{max}} < k_{\text{cross}}$, the number of index entries is too small to compensate for the recovery

overhead and TIV-IT gives lower MT. When $k_{\text{cross}} \leq k_{\text{max}} < k_{\text{PR-TIV}}^*$, PR-TIV improves over both baselines with gains growing toward the full gap. When $k_{\text{max}} \geq k_{\text{PR-TIV}}^*$, the asymptotic $(4-2\sqrt{2})\sqrt{n}$ -bit gap is realized. Table 4 lists the thresholds; k_{cross} grows slowly across an order-of-magnitude range of n .

Table 4: Crossover and MT-optimal tree depths for representative group sizes. k_{cross} : smallest integer k where PR-TIV with free caching ($\gamma=3/2$) beats both baselines.

n	Construction	k_{cross}	$\lfloor \sqrt{2n} \rfloor$
1 600	AdHash [BM97]	17	56
2 688	iSHAKE-128, $k \leq 25$ [MGS16]	18	73
4 160	iSHAKE-128 [MGS16]	19	91
6 528	iSHAKE-256, $k \leq 28$ [MGS16]	20	114
16 384	LtHash [BM97, LKMW19]	21	181
16 512	iSHAKE-256 [MGS16]	21	181

Minimum Group Size Meta’s LtHash [LKMW19] and Solana’s SIMD-0215 [PCSK24] specify $n = 16384$; if a fixed-position RGBP attack were instantiated at this size, the PR-TIV-optimal tree depth would be $k^* \approx \sqrt{2 \cdot 16384} \approx 181$, giving $\log_2 \text{MT}^* \approx 2\sqrt{2 \cdot 16384} \approx 362$ bits, compared with a baseline near 512 bits. For iSHAKE-128 in the unlimited-block setting ($n = 4160$), the drop is sharper: from 262 to 205 bits (Table 3). Restoring the original margin would require $n \geq 8580$ at leading order.

More generally, by Theorem 3, an S -bit MT security target (i.e., $\log_2 \text{MT}^* \geq S$) requires

$$n \geq \begin{cases} S^2/16 & \text{under TIV/TIV-IT,} \\ S^2/8 & \text{under PR-TIV (working memory),} \\ S^2/9 & \text{under PR-TIV (output-inclusive, } r=O(1)). \end{cases}$$

PR-TIV raises the minimum group size by a factor of 2 (working memory) or 16/9 (output-inclusive) at leading order. For example, a 128-bit MT target requires $n \geq 1024$ under TIV/TIV-IT, 2048 under PR-TIV, and 1821 output-inclusive. At 256 bits the three values are 4096, 8192, and 7282.

Mitigations Within the randomize-then-combine paradigm, the available knobs are n and the block-count cap K_{max} . *Increasing* n restores the MT margin directly, at the cost of larger hash outputs and higher per-block computation. *Capping* K_{max} limits the tree depth k and confines PR-TIV to the small- k regime where the index vectors do not yet dominate. The limited-block results in Section 5 (6–15 bits of MT reduction over the two baselines) illustrate this effect. In all cases, the absolute MT exponents remain far beyond practical search budgets [Ber07].

6 Conclusion

The key structural observation is that k -tree merging depends only on hash values, so for regenerable-leaf RGBP instances a hash-only forward pass suffices, with witnesses rebuilt only after a root match. Removing these 2^{k-1} index entries reshapes the MT optimum: the optimized $\log_2 \text{MT}$ drops from $4\sqrt{n}$ to $2\sqrt{2n}$ at large k . The fixed-size iSHAKE results illustrate this shift, with reductions relative to TIV-IT of about 51 and 122 bits in the unlimited-block regime, and smaller gains under block-count caps. A direct LSR analysis rules out the obvious alternative: uniformly shrinking every list and compensating by repetition only inflates the list-payload MT product. This leaves two natural directions open. *Asymmetric cost metrics*: in the loose setting, Dinur’s framework [Din19] gives

the trade-off curve $T^{\lceil k/2 \rceil + 1} M^{\lfloor k/2 \rfloor} = 2^n$; the analogous characterization for the RGBP is open, since its regular structure of independent fixed-position lists differs from the single list of the LGBP. *MT lower bounds:* no non-trivial lower bound is known for the RGBP MT cost, and whether $\log_2 \text{MT} = \Omega(\sqrt{n})$ is tight would determine whether PR-TIV's improvement is near-optimal.

Disclosure of AI Assistance

We used Kimi K2.6 to polish and translate selected paragraphs and OpenAI Codex to implement the paper's code. We reviewed and verified all such assistance.

References

- [Ber07] Daniel J. Bernstein. Better price-performance ratios for generalized birthday attacks. In *Workshop Record of SHARCS '07*, 2007. URL: <https://cr.yp.to/rumba20/genbday-20070904.pdf>.
- [BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 216–233. Springer, Berlin, Heidelberg, August 1994. doi:10.1007/3-540-48658-5_22.
- [BK16] Alex Biryukov and Dmitry Khovratovich. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. In *NDSS 2016*. The Internet Society, February 2016. doi:10.14722/ndss.2016.23108.
- [BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *Journal of the ACM*, 50(4):506–519, 2003. doi:10.1145/792538.792543.
- [BLL⁺22] Fabrice Benhamouda, Tancrède Lepoint, Julian Loss, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. *Journal of Cryptology*, 35(4):25, October 2022. doi:10.1007/s00145-022-09436-0.
- [BLN⁺09] Daniel J. Bernstein, Tanja Lange, Ruben Niederhagen, Christiane Peters, and Peter Schwabe. FSBday. In Bimal K. Roy and Nicolas Sendrier, editors, *INDOCRYPT 2009*, volume 5922 of *LNCS*, pages 18–38. Springer, Berlin, Heidelberg, December 2009. doi:10.1007/978-3-642-10628-6_2.
- [BM97] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 163–192. Springer, Berlin, Heidelberg, May 1997. doi:10.1007/3-540-69053-0_13.
- [Bro08] Daniel R. L. Brown. The encrypted elliptic curve hash. Cryptology ePrint Archive, Report 2008/012, 2008. URL: <https://eprint.iacr.org/2008/012>.
- [CDvD⁺03] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In Chi-Sung Lai, editor, *ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 188–207. Springer, Berlin, Heidelberg, November / December 2003. doi:10.1007/978-3-540-40061-5_12.
- [DEL25] Léo Ducas, Lynn Engelberts, and Johanna Loyer. Wagner’s algorithm provably runs in subexponential time for rmSIS^∞ . In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part I*, volume 16000 of *LNCS*, pages 353–384. Springer, Cham, August 2025. doi:10.1007/978-3-032-01855-7_12.
- [Din19] Itai Dinur. An algorithmic framework for the generalized birthday problem. *Designs, Codes and Cryptography*, 87(8):1897–1926, 2019. doi:10.1007/s10623-018-00594-6.
- [ES24] Andre Esser and Paolo Santini. Not just regular decoding: Asymptotics and improvements of regular syndrome decoding attacks. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VI*, volume 14925 of *LNCS*, pages 183–217. Springer, Cham, August 2024. doi:10.1007/978-3-031-68391-6_6.

- [GSC01] Bok-Min Goi, M. U. Siddiqi, and Hean-Teik Chuah. Incremental hash function based on pair chaining & modular arithmetic combining. In C. Pandu Rangan and Cunsheng Ding, editors, *INDOCRYPT 2001*, volume 2247 of *LNCS*, pages 50–61. Springer, Berlin, Heidelberg, December 2001. doi:[10.1007/3-540-45311-3_5](https://doi.org/10.1007/3-540-45311-3_5).
- [LF06] Éric Levieil and Pierre-Alain Fouque. An improved LPN algorithm. In Roberto De Prisco and Moti Yung, editors, *SCN 2006*, volume 4116 of *LNCS*, pages 348–359. Springer, Berlin, Heidelberg, September 2006. doi:[10.1007/11832072_24](https://doi.org/10.1007/11832072_24).
- [LKMW19] Kevin Lewi, Wonho Kim, Ilya Maykov, and Stephen Weis. Securing update propagation with homomorphic hashing. Cryptology ePrint Archive, Report 2019/227, 2019. URL: <https://eprint.iacr.org/2019/227>.
- [MGS16] Hristina Mihajloska, Danilo Gligoroski, and Simona Samardjiska. Reviving the idea of incremental cryptography for the zettabyte era use case: Incremental hash functions based on SHA-3. In Jan Camenisch and Dogan Kesdogan, editors, *Open Problems in Network Security – iNetSec 2015*, volume 9591 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2016. doi:[10.1007/978-3-319-39028-4_8](https://doi.org/10.1007/978-3-319-39028-4_8).
- [MS12] Lorenz Minder and Alistair Sinclair. The extended k -tree algorithm. *Journal of Cryptology*, 25(2):349–382, April 2012. doi:[10.1007/s00145-011-9097-y](https://doi.org/10.1007/s00145-011-9097-y).
- [MSTA17] Jeremy Maitin-Shepard, Mehdi Tibouchi, and Diego F. Aranha. Elliptic curve multiset hash. *The Computer Journal*, 60(4):476–490, 2017. doi:[10.1093/comjnl/bxw053](https://doi.org/10.1093/comjnl/bxw053).
- [NS15] Ivica Nikolic and Yu Sasaki. Refinements of the k -tree algorithm for the generalized birthday problem. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 683–703. Springer, Berlin, Heidelberg, November / December 2015. doi:[10.1007/978-3-662-48800-3_28](https://doi.org/10.1007/978-3-662-48800-3_28).
- [PCSK24] Brooks Prumo, Emanuele Cesena, Josh Siegel, and Sam Kim. Homomorphic hashing of account state (SIMD-0215). Solana Improvement Document, 2024. URL: <https://github.com/solana-foundation/solana-improvement-documents/blob/main/proposals/0215-accounts-lattice-hash.md>.
- [PW06] Raphael C.-W. Phan and David Wagner. Security considerations for incremental hash functions based on pair block chaining. *Computers & Security*, 25(2):131–136, 2006. doi:[10.1016/j.cose.2005.08.002](https://doi.org/10.1016/j.cose.2005.08.002).
- [TDSG26] Lili Tang, Rui Ding, Yao Sun, and Xiaorui Gong. Memory optimizations of Wagner’s algorithm with applications to Equihash. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):218–239, 2026. doi:[10.46586/tches.v2026.i2.218-239](https://doi.org/10.46586/tches.v2026.i2.218-239).
- [TSG26] Lili Tang, Yao Sun, and Xiaorui Gong. On the regularity of the generalized birthday problem. In Nadia Heninger and Mike Rosulek, editors, *Advances in Cryptology – CRYPTO 2026*, Lecture Notes in Computer Science. Springer, 2026. IACR ePrint 2025/1351. URL: <https://eprint.iacr.org/2025/1351>.
- [Wag02] David Wagner. A generalized birthday problem. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 288–303. Springer, Berlin, Heidelberg, August 2002. doi:[10.1007/3-540-45708-9_19](https://doi.org/10.1007/3-540-45708-9_19).

A First Non-Free Depth for MT

Lemma 3 (First non-free depth). *Let $k \geq 8$, let $\delta_f = \delta_{\text{free}}(k)$, and put $d_0 = \delta_f + 1$. Among the non-free caching depths $d_0 \leq \delta \leq k - 1$, the product $\text{MT}(\delta) = M_\delta \cdot \alpha_\delta T_0$ is minimized at $\delta = d_0$. (Caching all k levels, $\delta = k$, is the full-index TIV baseline of Proposition 1.)*

Proof. Drop the common factor $\ell N T_0$ and write $t = k - \delta$ and $u = k - d_0$. Then

$$H_{k,\delta} = P_k - \frac{t(t+3)}{4}, \quad f(\delta) = \frac{M_\delta \alpha_\delta}{\ell N T_0} = \left(P_k - \frac{t(t+3)}{4} + 2^{\delta-1} \right) (t+1),$$

where $\alpha_\delta = t + 1$, and

$$f(\delta + 1) - f(\delta) = (t - 1)2^{\delta-1} + \frac{t(3t+5)}{4} - P_k. \quad (1)$$

It suffices to show that (1) is positive for every non-free depth with $2 \leq t \leq u$. For $k = 8, 9$ this is a direct check. For $k \geq 10$, $q = \lfloor k/2 \rfloor$ is non-free because $2^{q-1} > \frac{1}{4}(q+1)(q+4) \geq \frac{1}{4}(k-q)(k-q+3)$. Thus $d_0 \leq q$, $u \geq \lceil k/2 \rceil \geq 5$, and $k \leq 2u$. With $B = u(u+3)/4$, this gives $P_k = \frac{1}{4}(k^2 + 5k + 2) \leq u^2 + \frac{5}{2}u + \frac{1}{2} < u(u+3) = 4B$. Finally, put $r = u - t$. Since $2^{d_0-1} > B$ and $(u - r - 1)2^r \geq u - 1 \geq 4$,

$$(t - 1)2^{\delta-1} = (u - r - 1)2^r 2^{d_0-1} > 4B > P_k,$$

so (1) is positive. Hence f increases on $[d_0, k - 1]$ and is minimized at $d_0 = \delta_f + 1$. $\square$

B Applicability by Construction

Table 5: PR-TIV applicability by attack-instance structure. A single $\times$ in the public-leaves, RGBP-lists, or Wagner-merge columns blocks PR-TIV.

Construction	Variant (source)	Public leaves	RGBP lists	Wagner merge	PR-TIV
<i>Fixed-position fixed-size preimage instances</i>					
iSHAKE	benchmark [MGS16]	✓	✓	✓	✓
LtHash	analysis variant [BM97, LKMW19]	✓	✓	✓	✓
AdHash	analysis variant [BM97]	✓	✓	✓	✓
<i>Set/update or non-RGBP settings</i>					
LtHash set/update	Meta LAD [LKMW19]; Solana ALH [PCSK24]	✓	✗	✓	✗
MSet-XOR-Hash	MSet design [CDvD ⁺ 03]	✗	✗	✓	✗
MSet-Add-Hash	MSet design [CDvD ⁺ 03]	✗	✗	✓	✗
MSet-Mu-Hash	MSet design [CDvD ⁺ 03]	✓	✗	✗	✗
MSet-VAdd-Hash	MSet design [CDvD ⁺ 03]	✓	✗	✓	✗
MuHASH fixed-position	analysis variant [BM97]	✓	✓	✗	✗
EECH fixed-position	analysis variant [Bro08]	✓	✓	✗	✗
ECMH	ECMH [MSTA17]	✓	✗	✗	✗
PCIHF	PCIHF [GSC01, PW06]	✓	✗	✓	✗