Memory Optimizations of Wagner's Algorithm with Applications to Equihash

Lili Tang[1,2], Rui Ding[1,2], Yao Sun[1,2] and Xiaorui Gong[1,2](✉)

[1] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{tanglili,dingrui2024,sunyao,gongxiaorui}@iie.ac.cn

Abstract.
The Generalized Birthday Problem (GBP) serves as a cornerstone for a broad spectrum of cryptanalytic research. The classical solution, Wagner's k-tree algorithm (CRYPTO'02), is characterized by inherently high memory complexity. Subsequently, Biryukov and Khovratovich (NDSS'16) introduced Equihash, a memory-hard proof-of-work scheme constructed upon a single-list variant of Wagner's algorithm. Due to its robust resistance to ASIC mining, Equihash has emerged as one of the most widely adopted proof-of-work schemes in blockchain. Memory optimization for Wagner-type algorithms remains a persistent challenge in cryptographic research. Conventional approaches primarily focus on reducing list size to lower memory consumption, yet this strategy typically incurs a prohibitive time penalty. For instance, halving the memory usage of the mining algorithm for Equihash$(200, 9)$ increases its theoretical time complexity by a factor of $2^{24.6}$.
In this work, we explore a new optimization direction: List Item Reduction (LIR), which facilitates practical and fine-grained memory-time trade-offs. We systematize existing LIR techniques and propose novel optimization methods integrated into a new hybrid framework. While our approach does not improve asymptotic memory complexity, it achieves a near-linear trade-off in practice, offering substantial benefits for the concrete design of efficient Wagner-style algorithms. Specifically, our techniques reduce peak memory usage by more than 50% (from $> 2nN$ to nN bits) across all Equihash parameters, with only an approximate twofold time penalty. For Equihash$(144, 5)$, our optimized algorithm requires only 700 MB of memory, compared to approximately 2.5 GB in previous implementations.

Keywords: Wagner's Algorithm · Equihash · Generalized Birthday Problem

1 Introduction

Since Wagner's k-tree algorithm [Wag02] was first introduced at CRYPTO'02, it has been widely applied in various areas of cryptography including incremental hashing [BGG94, BM97, GSC01, MGS15], the blind signature scheme (ROS problem) [Sch01, BLL+21] and the subset sum problem [Sha08, DRX17]. The k-tree algorithm is designed to solve the Generalized Birthday Problem (GBP) and the k-SUM problem, which aims to find k elements from k lists L_i, whose XOR (or modular sum) equals zero, i.e., $x_1 \oplus \cdots \oplus x_k = 0$ where each $x_i \in L_i$. The single-list variant of Wagner's k-tree algorithm first appeared in the context of the LPN problem [LF06], which we refer to as the single-chain algorithm in this work. Unlike the k-tree algorithm, it operates on a single list and achieves better time and memory efficiency. The single-chain algorithm was later adopted in Equihash [BK16].

Equihash is currently one of the most widely deployed memory-hard proof-of-work schemes, used in blockchains such as Zcash [HBHW17] and Bitcoin Gold. As of December 2025, Zcash alone had a market capitalization exceeding $6 billion. Equihash's primary

advantage lies in its claimed ASIC-resistance owing to the memory-hardness of solving GBP. The influx of large numbers of ASIC miners can quickly erode the mining advantage of ordinary users (CPU/GPU), leading to mining centralization. This concentration of hash power substantially increases the risk of 51% attacks. However, the emergence of new memory optimization techniques [Equ16, Tro16] has significantly weakened the intended memory hardness of Equihash, which led to a 51% attack on Bitcoin Gold in 2018, resulting in losses of approximately $18 million. Thus, the memory complexity of Wagner's algorithms serves as a crucial security guarantee for Equihash-based blockchains.

The memory optimization within Wagner's algorithmic framework has long remained an open problem. A large body of previous work [Ber07, MS12, NS15, Din19] has focused on reducing the input list size of Wagner's k-tree algorithm to lower the memory requirements with significant time penalty. However, these studies concentrate on optimizing the trade-off of asymptotic complexity while neglecting substantial polynomial factors, thus failing to accurately capture the concrete trade-offs. The work of [BLN+09] studied more concrete trade-off techniques for attacks on FSB hashing [AFS03] based on Wagner's k-tree algorithm, and was the first to introduce techniques aimed at reducing the amortized storage cost per list item, which we refer to as list item reduction (LIR) techniques in this paper. These techniques were later further developed in the context of the single-chain setting of Equihash [BK16]. After the launch of Zcash [HBHW17], the first blockchain based on Equihash, many software implementations [Equ16, Tro16] of the single-chain algorithm utilize the index-pointer technique, which significantly reduces the amortized storage cost of index information and thereby greatly lowers the peak memory. Subsequently, Bai et al. [BGHZ19] proposed a multi-chip ASIC miner design for Equihash, in which the index-pointer technique plays a crucial role within the overall architecture.

In this work, we provide the first comprehensive treatment of list-item reduction techniques, presenting novel optimizations and a new algorithmic framework. On the one hand, memory hardness is a major practical bottleneck when applying Wagner's algorithms. For instance, Wagner-type algorithms are widely applicable to the ROS problem [BLL+21], the LWE problem [BKW03] and the LPN problem [LF06]. Consequently, the associated memory optimization techniques may also be integrated into efficient solvers for these problems. On the other hand, our memory optimization techniques substantially weaken the ASIC-resistance of Equihash. Blockchains that rely on Equihash should therefore reassess their choice of parameters regarding ASIC-resistance.

1.1 Wagner's Algorithmic Framework

The following definition of GBP follows that used in Equihash, whereas Wagner's original formulation corresponds to a structured variant of GBP with additional regularity constraints. We assume $K = 2^k$ and consider only hash-based GBP, unless stated otherwise.

Definition 1 ($\mathbf{GBP}(n, K = 2^k)$)**.** Given a list L containing n-bit random values, e.g., sampled from cryptographic hashing, find K distinct elements $x_1, \ldots, x_K \in L$ such that:

$$x_1 \oplus x_2 \oplus \cdots \oplus x_K = 0.$$

For the remainder of this paper, we define $\ell = \frac{n}{k+1}$ and $N = 2^{\ell+1}$ for $\mathbf{GBP}(n, 2^k)$.

Single-Chain and k-tree Algorithms. Wagner's k-tree algorithm [Wag02] is based on k lists which differ from the single-list setting in Definition 1. The essential distinction between the k-tree and single-chain algorithms lies in regularity [AFS03]: a blockwise structure first introduced in the code-based cryptography literature. The work in [TSG25] provides the first systematic study of the average-case and worst-case hardness between regular and non-regular GBP. Furthermore, it delineates the distinct application scenarios for k-tree and single-chain algorithms in cryptanalysis, specifically mapping them to

preimage and collision attacks on incremental hashing [MGS15], respectively. Hence, it is meaningful to distinguish the two algorithms strictly. Due to their algorithmic similarity, however, our work focuses mainly on the single-chain algorithm. Nevertheless, our new methods can be extended to the k-tree setting as well.

For clarity, we present the single-chain algorithm in Algorithm 1, which involves k merge operations. Each $L' = \mathsf{merge}_\ell(L)$ takes as input a list L of size N, where each item consists of an XOR value and its associated index vector, denoted (x, i). The merge operation searches for all distinct pairs $(x_1, i_1), (x_2, i_2) \in L$ such that the least significant ℓ bits of $x_1 \oplus x_2$ are zero. The output list L' then stores the new XOR value $(x_1 \oplus x_2) \gg \ell$ along with the concatenated index vector $i_1 \| i_2$. The correctness of Algorithm 1 follows from the fact that $\binom{N}{2}/2^\ell \approx N$, ensuring that the size of L' remains close to N. In Algorithm 1, the list sizes of $L^{(0)}, L^{(1)}, \ldots, L^{(k-1)}$ are all approximately N, while the expected size of the final list $L^{(k)}$ is $\binom{N}{2}/2^{2\ell} \approx 2$. The merge operation can be performed in time and memory $\mathcal{O}(N)$ [Wag02, Equ16]. Therefore, the single-chain algorithm is expected to generate two solutions in time $\mathcal{O}(kN)$. However, its memory complexity cannot be simply summarized as $\mathcal{O}(N)$, since the storage cost of the index vector in Algorithm 1 grows exponentially with the layer height j (of $L^{(j)}$). This observation lies at the core of list-item reduction techniques.

Algorithm 1 $\mathsf{CIV}(n, k)$: The Single-Chain Algorithm with Index Vector for $\mathbf{GBP}(n, 2^k)$

Input: A single list L of size $N = 2^{\ell+1}$ where $\ell = \frac{n}{k+1}$.
Output: A solution list to $\mathbf{GBP}(n, 2^k)$.

1: Initialize $L^{(0)} \leftarrow L$. ▷ List item in $L^{(0)}$: n-bit initial value.
2: **for** $j = 1$ to $k-1$ **do**
3: $L^{(j)} \leftarrow \mathsf{merge}_\ell(L^{(j-1)})$. ▷ List item in $L^{(j)}$: $\left(\underbrace{(n - j \cdot \ell)}_{\text{XOR value}}, \underbrace{2^j \times (\ell+1)}_{\text{Index Vector}}\right)$.
4: Remove $L^{(j-1)}$ from memory.
5: Return $L^{(k)} \leftarrow \mathsf{merge}_{2\ell}(L^{(k-1)})$. ▷ List item in $L^{(k)}$: $2^k \times (\ell+1)$-bit index vector.

Remark 1 (Wagner's k-tree algorithm [Wag02]). Given $\mathbf{GBP}(n, 2^k)$, Wagner's k-tree algorithm takes 2^k input lists, denoted as $L_1^{(0)}, L_2^{(0)}, \ldots, L_{2^k}^{(0)}$, each of size 2^ℓ. The k-tree algorithm proceeds as follows:

- **Round** 1 **to** $(k-1)$: For $i \in [1, 2^{k-1}]$, merge the two lists $L_{2i-1}^{(0)}$ and $L_{2i}^{(0)}$ by matching all pairs $x \in L_{2i-1}^{(0)}$, $y \in L_{2i}^{(0)}$ such that $x \oplus y \equiv 0 \pmod{2^\ell}$. The resulting lists are denoted as $L_1^{(1)}, L_2^{(1)}, \ldots, L_{2^{k-1}}^{(1)}$. For the next round, take the new lists as input.

- **Round** k: Merge $L_1^{(k-1)}$ and $L_2^{(k-1)}$ by matching 2ℓ bits. Recover the solution to GBP using the stored index information.

All intermediate lists $L_{1,2,\ldots,2^k}^{(0)}, L_{1,2,\ldots,2^{k-1}}^{(1)}, \ldots, L_{1,2}^{(k-1)}, L_1^{(k)}$ form a complete binary tree from bottom to top. If we traverse the entire binary tree in a post-order manner to generate the root list $L_1^{(k)}$ (solution list), then at any moment we need to keep at most k lists in memory. Similarly, the storage cost of index information also grows exponentially with the height of the tree. As a result, the memory complexity of the k-tree algorithm is typically substantially higher than that of the single-chain algorithm.

1.2 Contributions

Notations on Time Penalty. To quantify the efficiency loss caused by memory optimization techniques, we denote by T_0 the baseline runtime of the plain algorithm, and by T_1 the runtime of a memory-optimized variant. The time penalty is defined as the extra runtime: $\Delta = T_1 - T_0$ and the ratio $\gamma = T_1 / T_0 = 1 + \frac{\Delta}{T_0}$ represents the time penalty factor (greater than 1). Similarly, we define the memory reduction factor as $q = \frac{M_0}{M_1} > 1$. For example, $q = 2$ corresponds to a twofold reduction in memory usage.

Bottlenecks of State-of-the-Art. The known List Size Reduction (LSR) trade-off techniques typically ignore polynomial factors in n and K, focusing instead on the asymptotic complexity $\widetilde{\mathcal{O}}(N) = \widetilde{\mathcal{O}}(2^{\frac{n}{k+1}})$. Such trade-offs usually incur substantial time penalties and are unsuitable for fine-grained memory optimizations. In particular, within the Equihash setting, the algorithm-bound constraints render many LSR-based techniques inapplicable. Applying the state-of-the-art LSR trade-off [BK16] to halve the peak memory usage of Equihash(200, 9) results in a time penalty factor $\gamma = 2^{24.6}$. This high time-memory trade-off cost constitutes one of the key properties underpinning Equihash's ASIC resistance. Furthermore, existing merge implementations are not-in-place; that is, computing $L_2 = \mathsf{merge}(L_1)$ results in a peak memory consumption of $|L_1| + |L_2|$, rather than $\max(|L_1|, |L_2|)$. This imposes a strict lower bound on the peak memory for state-of-the-art implementations [Equ16, Tro16], which cannot be less than $2nN$ bits.

New Techniques for List Item Reduction. In this work, we focus on a new optimization direction: list-item reduction (LIR), which enables fine-grained memory-time trade-offs within a constrained range, significantly mitigating the bottlenecks of existing methods. We investigate two existing implementations of the single-chain algorithm. The first, referred to as the CIV implementation, employs the index-vector structure for index representation, while the second, the CIP implementation, utilizes the index-pointer structure.

1. For the CIV implementation, we significantly improve the index-trimming technique by introducing a constrained single-chain algorithm to reconstruct the full solution vector in the second run. The peak memory of our new algorithm is bounded by $\max(nN, (2^k + 2\ell)N)$, while incurring only a small time penalty factor ≈ 2.

2. For the CIP implementation, we introduce a new technique, termed post-retrieval, which reduces the peak memory usage from approximately $2nN$ to nN bits, while incurring only a modest time penalty factor of about $1 + k/8$. In particular, when external-memory caching is available, the CIP algorithm can be executed with nN bits of memory and a negligible time overhead in practice.

Finally, we introduce a hybrid algorithmic framework combining our new techniques, within which the optimal configuration can achieve a peak memory usage below nN bits. Theoretical estimates of our new techniques applied to standard Equihash parameter sets are summarized in Table 1. In the k-tree setting [Wag02], the involved lists form a complete binary tree structure, which makes LIR techniques for CIP implementation less effective. Nevertheless, trivially applying the index-trimming technique [BLN+09] can still achieve a significant reduction in peak memory at very low computational cost, as illustrated in Table 1. How to further optimize LIR techniques within the k-tree setting remains an open problem for future research.

Practical Implications on Equihash. Existing high-performance implementations [Tro16, Equ16] rely on a not-in-place merge operation, which leads to elevated peak memory usage. In contrast, our implementation provides the first concrete in-place merge and

Table 1: New Trade-offs within Wagner's Algorithmic Framework for $\mathbf{GBP}(n, 2^k)$. The single-chain (CIP) and k-tree (TIV) algorithms represent state-of-the-art.

(n,k)	Single-Chain							k-tree		
	CIP [Equ16]	**CIV-IT 3.1**		**CIP-PR 3.2**		**Hybrid 3.3**		**TIV** [Wag02]	**TIV-IT 3.1**	
	Mem.	Mem.	Time	Mem.	Time	Mem.	Time	Mem.	Mem.	Time.
(96, 5)	$2^{24.39}$	$2^{23.09}$	$2.3 \cdot T_0$	$2^{23.58}$	$2.2 \cdot T_0$	$2^{23.04}$	$3.8 \cdot T_0$	$2^{25.86}$	$2^{24.81}$	$1.3 \cdot T_1$
(128, 7)	$2^{24.88}$	$2^{23.58}$	$3.2 \cdot T_0$	$2^{24.00}$	$2.4 \cdot T_0$	$2^{23.64}$	$3.9 \cdot T_0$	$2^{27.42}$	$2^{25.67}$	$1.3 \cdot T_1$
(160, 9)	$2^{25.25}$	$2^{25.17}$	$1.4 \cdot T_0$	$2^{24.32}$	$2.7 \cdot T_0$	$2^{24.07}$	$4.0 \cdot T_0$	$2^{29.17}$	$2^{26.58}$	$1.3 \cdot T_1$
(144, 5)	$2^{32.95}$	$2^{31.64}$	$2.3 \cdot T_0$	$2^{32.17}$	$2.2 \cdot T_0$	$2^{31.61}$	$3.8 \cdot T_0$	$2^{34.44}$	$2^{33.36}$	$1.3 \cdot T_1$
(150, 5)	$2^{34.01}$	$2^{32.70}$	$2.3 \cdot T_0$	$2^{33.23}$	$2.2 \cdot T_0$	$2^{32.67}$	$3.8 \cdot T_0$	$2^{35.50}$	$2^{34.41}$	$1.3 \cdot T_1$
(192, 7)	$2^{33.44}$	$2^{32.00}$	$3.2 \cdot T_0$	$2^{32.58}$	$2.4 \cdot T_0$	$2^{32.21}$	$3.9 \cdot T_0$	$2^{36.00}$	$2^{34.18}$	$1.3 \cdot T_1$
(240, 9)	$2^{33.81}$	$2^{33.25}$	$1.4 \cdot T_0$	$2^{32.91}$	$2.7 \cdot T_0$	$2^{32.63}$	$4.0 \cdot T_0$	$2^{37.75}$	$2^{35.00}$	$1.3 \cdot T_1$
(288, 8)	$2^{42.04}$	$2^{40.64}$	$2.9 \cdot T_0$	$2^{41.17}$	$2.2 \cdot T_0$	$2^{41.00}$	$2.9 \cdot T_0$	$2^{45.27}$	$2^{42.93}$	$1.3 \cdot T_1$
(200, 9)	$2^{29.55}$	$2^{29.21}$	$1.4 \cdot T_0$	$2^{28.64}$	$2.7 \cdot T_0$	$2^{28.38}$	$4.0 \cdot T_0$	$2^{33.49}$	$2^{30.81}$	$1.3 \cdot T_1$

TIV: k-tree Algorithm with Index Vector CIP: Single-Chain Algorithm with Index Pointer
IT: Index Trimming PR: Post Retrieval T_0, T_1: plain runtime of CIP and TIV

integrates it with our proposed optimization techniques. These new techniques reduce the ASIC-resistance of Equihash as shown in Figure 1.

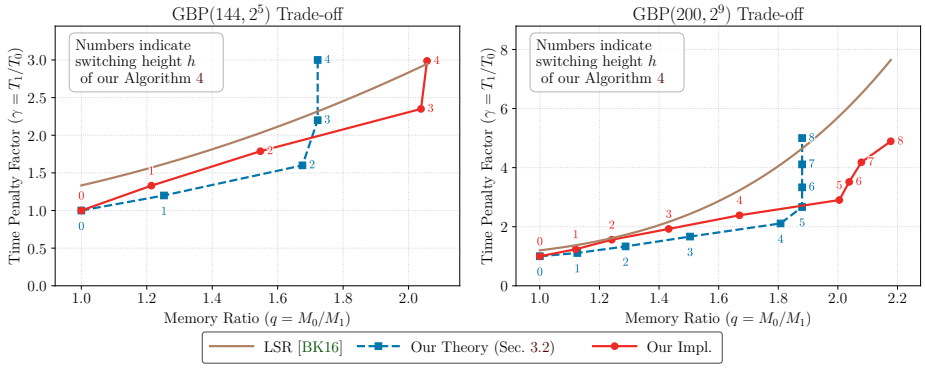


Figure 1: LSR and LIR Trade-off of GBP. The optimal switching height is $h^\star = (k+1)/2$. When $h > h^\star$, no further memory benefits can be obtained in theory.

Remark 2. At switching height $h = 0$, the baseline memory M_0 stores all index-pointer lists. Since practical implementations require byte alignment, the actual storage cost of index pointers in our implementation is higher than the theoretical estimate, e.g., 32 bits instead of 25 bits for Equihash(144,5). This gap gradually diminishes as the switching height increases and our implementation reduces more memory than predicted by theory when measured relative to M_0. In Figure 1, the slight memory variations after the optimal switching height are mainly due to lazy memory allocation mechanism in C++.

1. **Memory Hardness**. Compared to state-of-the-art software implementations in [Tro16], our baseline implementation for Equihash$(144, 5)$ achieves nearly the same runtime, while requiring only 1.45 GB of memory (0.57× of Tromp's 2.5 GB footprint). Moreover, by accepting roughly a 2× time penalty factor, the peak memory can be further reduced to 700 MB (0.28× of Tromp's). For hardware implementations, the combined use of our external-memory caching design and the in-place merge offers additional advantages for ASICs. Full details are provided in Table 5.

2. **Time-Memory Trade-off**. Our concrete trade-off results of the proposed post-

retrieval technique are presented in Figure 1. Compared with existing LSR trade-off techniques, our approach is both theoretically and practically superior in the small trade-off regime with memory fraction $q \in [1, 2]$. In particular, relative to algorithm-bound LSR trade-offs for Equihash, our method incurs a negligible time-penalty factor compared to 2^{24} for Equihash(200,9) and 2^{11} for Equihash(144, 5) when $q = 2$.

Roadmap. The remainder of this paper is organized as follows. Section 2 reviews state-of-the-art memory optimization techniques in Wagner's algorithms, including the LSR and LIR approaches. In Section 3, we introduce our two main optimization techniques along with the proposed hybrid algorithmic framework. Section 4 describes our proof-of-concept implementations, which demonstrate the effectiveness of the proposed methods. Finally, Section 5 discusses several open problems and outlines directions for future work. Our implementations and estimators are publicly available at:

https://github.com/tl2cents/Wagner-Algorithms

2 Related Works and Known Techniques

Over the past two decades, the memory optimization of Wagner's algorithms has been extensively studied. In this section, we discuss related works and present state-of-the-art trade-off techniques. To the best of our knowledge, we categorize existing memory optimizations of Wagner's algorithmic framework into two types:

1. **List Size Reduction (LSR)** [Ber07, BLN+09, MS12, NS15, BK16, Din19]. The size of list $L^{(0)}$ is reduced by partially sampling, which exponentially decreases the probability of obtaining a correct solution in a single run. However, this can be mitigated by introducing a *preparation phase* to preprocess the lists and by optimizing the *list-sum phase* (merge), thereby improving the overall time-memory trade-off.
2. **List Item Reduction (LIR)** [BLN+09, BK16, Equ16]. Instead of reducing the list size, this approach focuses on optimizing the storage of list items: hash value and its index vector in L_j for $j > 1$. **The goal is to minimize the amortized cost of storing indices and hash values, while running Wagner's algorithm multiple times to reconstruct the complete solution**.

It is worth noting that LSR techniques typically emphasize asymptotic memory complexity, allowing for a significant time penalty. The associated time-memory trade-off curves generally ignore large polynomial factors. In contrast, LIR techniques provide more practical memory optimizations without incurring a significant time penalty. Although the achievable memory savings are bounded (with asymptotic memory complexity remaining $\mathcal{O}(N)$), they are more common in practice—for instance, in optimizing Equihash.

2.1 State-of-the-Art: List Size Reduction

Our work primarily focuses on the LIR technique; therefore, we only provide a brief overview of the current state-of-the-art results on LSR, particularly under the algorithm-bound constraint. Denote the peak memory and time complexity for solving $\mathsf{GBP}(n, K = 2^k)$ as $M(n, k)$ and $T(n, k)$, respectively. Dinur's work [Din19] provides a comprehensive framework for the k-tree algorithm. For $k > 3$, its time-memory trade-off curve:

$$T(n,k)^{\lceil k/2 \rceil + 1} M(n,k)^{\lfloor k/2 \rfloor} = 2^n,$$

remains state-of-the-art, particularly by optimizing cases where K is not a power of two. It should be emphasized that the trade-off curves above reflect only asymptotic complexity, neglecting the constant factor k, and therefore do not serve as accurate trade-off formulas.

For the single-chain case, the best-known technique is given in Equihash [BK16]. Specifically, when the single-chain algorithm operates with a reduced list size by a factor of q (i.e., using only $M(n,k)/q$ memory), the corresponding time complexity becomes

$$T'_q(n,k) \approx \frac{3q^{\frac{k-1}{2}} + k}{k+1} \cdot T(n,k). \tag{LSR}$$

This type of trade-off usually incurs significant time penalties, particularly as k increases. For $\mathsf{GBP}(200, 2^9)$, applying LSR reduces the memory peak by half, while the new time complexity becomes $5.7 \cdot T(200, 9)$ in theory (no practical implementation available). However, the above trade-off modifies the standard single-chain procedure: in the first merge, it collides on $\ell + k \log q$ bits instead of ℓ bits, then reduces the collision bits by $\log q$ bits in each of the following $k-2$ merge operations, and by $2 \log q$ bits in the final merge. The first step with limited memory is more expensive and relies on parallel memory-efficient collision search techniques such as [vOW99, TID21]. It should be particularly noted that such trade-offs are not applicable to Equihash, as Equihash strictly binds the algorithmic binary-tree structure to Wagner's original algorithm.

Algorithm-Bound Constraints in Equihash. $\mathsf{Equihash}(n,k)$ is a tree-bound variant of $\mathbf{GBP}(n, 2^k)$. Let $x_1^{(0)}, x_2^{(0)}, \ldots, x_{2^k}^{(0)}$ be a solution to Equihash. Define $x_j^{(i)} := x_{2j-1}^{(i-1)} \oplus x_{2j}^{(i-1)}$ for all $i \in [1,k]$ with corresponding $j \in [1, 2^{k-i}]$. These nodes exactly form a complete binary tree. At height $h \in [1, k-1]$, $x_j^{(h)}$ has $h \cdot \ell$ leading zeros and the root $x^{(k)} = x_1 \oplus x_2 \oplus \cdots \oplus x_{2^k}$ is zero. These constraints impose much stricter limitations on the design of time-memory trade-offs, making it difficult to adapt existing LSR techniques [MS12, NS15, Din19] to Equihash. The state-of-the-art trade-off for Equihash is given in the following proposition.

Proposition 1 (LSR for Equihash [BK16])**.** *Given limited $M(n,k)/q$ memory for Equihash with $q > 1$, one can find 2 solutions to Equihash in time:*

$$T'_q(n,k) = 2^k q^{\frac{k}{2}} k^{\frac{k}{2}-1} \cdot T(n,k). \tag{LSR*}$$

The constant factor is substantially larger than that of the plain (LSR). For $\mathsf{Equihash}(200, 2^9)$, achieving a 50% reduction in peak memory requires about $2^{24.6}$ times the baseline running time $T(n,k)$. Hence, LSR is not suitable as a concrete trade-off technique for GBP, especially for its variant Equihash. Nevertheless, our work demonstrates that while Equihash exhibits inherent resistance to conventional LSR trade-offs, it can still be challenged in practice by our advanced LIR techniques.

2.2 State-of-the-Art: List Item Reduction

The LIR technique for the k-tree algorithm was initially explored in [BLN+09]. Recent studies are mostly centered around Equihash, where the methods are specifically designed for the single-chain setting. In Algorithm 1, to reconstruct the complete set of message indices corresponding to a solution, each list item at the i-th layer $L^{(i)}$ consists of:

1. **XOR Part**: the XOR value for colliding. We denote the XOR value list as X_i, where the minimal storage cost is $n - i \cdot \ell$ bits at layer $L^{(i)}$.
2. **Index Part**: the correspondence between the XOR value and the initial list $L^{(0)}$. Two storage strategies are possible:
   - **Index Vector** [BK16]: The index vector $(i_1, i_2, \ldots, i_{2^i})$ indicates that the current hash value is obtained as the XOR of 2^i elements from the initial list $L^{(0)}$. Under index vectors, the amortized storage cost for indices is $2^i \cdot (\ell + 1)$.

Algorithm 2 $\mathsf{CIP}(n,k)$: The Single-Chain Algorithm with Index Pointer for $\mathbf{GBP}(n,2^k)$

Input: A single list L of size $N = 2^{\ell+1}$ where $\ell = \frac{n}{k+1}$.
Output: A solution list to $\mathbf{GBP}(n,2^k)$.

1: Initialize $L^{(0)} \leftarrow L$. ▷ List item in $L^{(0)}$: n-bit XOR.
2: **for** $j = 1$ to $k-1$ **do**
3: $L^{(j)} \leftarrow \mathsf{merge}_\ell(L^{(j-1)})$. ▷ List item in $L^{(j)}$: $\left(\underbrace{(n - j\cdot\ell)}_{\text{XOR value}}, \underbrace{2\times(\ell+1)}_{\text{Index Pointer}}\right)$.
4: Remove XOR Value of $L^{(j-1)}$ from memory.
5: $L^{(k)} \leftarrow \mathsf{merge}_{2\ell}(L^{(k-1)})$. ▷ $L^{(k)}$ of expected size 2 storing index pointers
6: Backtrack index pointers in $L^{(k)}, L^{(k-1)}, \dots L^{(1)}$ and return solution indices.

- **Index Pointer** [Equ16]: The index pointer pair (l,r) represents the two indices in the previous layer $L^{(i-1)}$ that collide to form an element in the current layer $L^{(i)}$. The index pointer costs constant $2(\ell+1)$ bits at each layer.

The practical single-chain implementation [Equ16] for Equihash employs the index-pointer LIR technique. In Algorithm 2, we store index-pointer pair lists $\mathsf{IP}_1, \mathsf{IP}_2, \dots, \mathsf{IP}_k$ across all layers. The final list IP_k is expected to contain two index pointer pairs, each corresponding to one desired solution. Consequently, the complete message indices can be reconstructed by recursively resolving the index pointers from the top layer IP_k back to IP_1. In $L^{(i)}$, the amortized storage cost with index pointers is $2i\cdot(\ell+1)$.

Proposition 2 (Complexity of Single-Chain Algorithm)**.** *The $\mathsf{CIV}(n,k)$ algorithm runs in time $\mathcal{O}(kN)$ and requires $\mathcal{O}\left(\left(2^{k-1}\cdot(\ell+1)+2\ell\right)\cdot N\right)$ bits memory. The $\mathsf{CIP}(n,k)$ algorithm runs in time $\mathcal{O}(kN)$ and requires $\mathcal{O}\left(2\cdot(n+k-\ell-1)\cdot N\right)$ bits memory.*

Proof. The time complexity of $\mathsf{CIV}(n,k)$ algorithm 1 is $\mathcal{O}(kN)$ i.e. k times merge operations on a single list of size N. The peak memory consumption occurs in the final step, where it is necessary to store 2^{k-1} indices and 2ℓ bits for each list item. The resulting peak memory, measured in bits, is given by:

$$\mathcal{O}\left(\left(2^{k-1}\cdot(\ell+1)+2\ell\right)\cdot N\right).$$

In each merge of $\mathsf{CIP}(n,k)$, at least $2(\ell+1)$ bits are added in list item and the temporary XOR value decreases by ℓ bits. After one merge, the memory usage increases by $(2(\ell+1)-\ell)N = (\ell+2)N$ bits. The memory usage reaches its peak before the final merge:

$$\mathcal{O}\left((n+(\ell+2)\cdot(k-1))N\right) = \mathcal{O}\left(2\cdot(n+k-\ell-1)\cdot N\right).$$

□

Known Techniques for List Item Reduction. There are two types of low-time-penalty trade-offs for Wagner-type algorithms (the k-tree and single-chain variants); however, both apply only to the index-vector setting.

- **XOR Removal** [BLN+09]: Removing intermediate XOR values in Wagner's algorithm necessitates recomputing hash values in each layer. This technique can be applied in the low-height layers with small time penalty.
- **Index Trimming** [BLN+09, BK16]: In the first run of Wagner's algorithm, the index trimming technique stores only $t < \ell+1$ bits of the index. After the first run, the obtained t-bit partial solution partitions the single list into 2^k sub-lists, each of size $2^{\ell+1-t}$. Then retrieve the full solution in a subsequent run using the k-tree algorithm with the sub-lists.

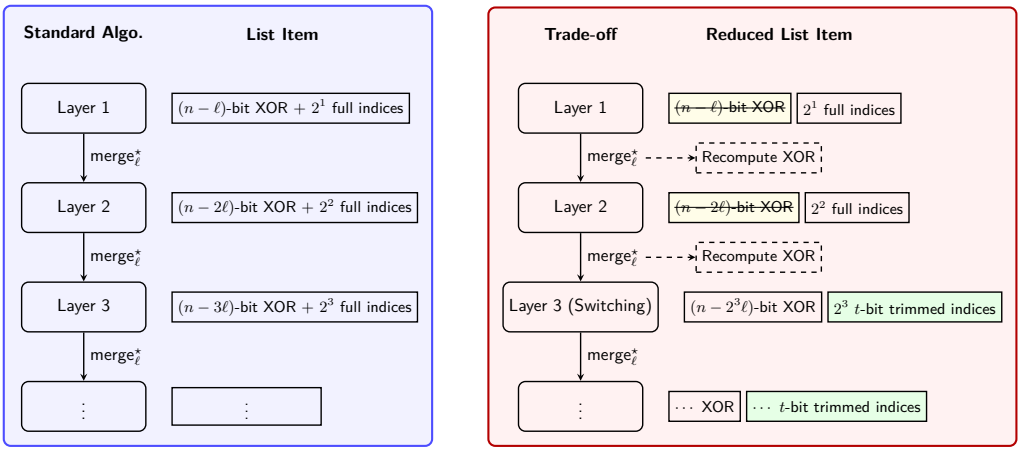


Figure 2: A combined view of the list item structures in single-chain algorithm with index vector. The trade-off strategy (right) utilizes a switching height of 3 (the second run of k-tree algorithm is omitted).

Single-Chain Algorithm Trade-off [BK16]. XOR-removal and index-trimming techniques can be jointly applied in single-chain algorithm with index vector. For example, as illustrated in Figure 2, when the layer height is below 3, XOR-removal technique reduces memory usage without significantly compromising time performance. At the switching height of 3, switching to index-trimming lowers peak memory consumption. Equihash utilizes this combined trade-off strategy with an index trimming length of $t = 8$ and the peak memory is given by $M = \left(2^{k+2} + 2\ell\right) \cdot N$ bits. However, under existing not–in-place implementations of merge, the practical peak memory requirement doubles.

3 New Techniques for List Item Reduction

In this section, our work focuses on the list item reduction technique for the single-chain algorithm. As previously discussed, the single-chain algorithm admits two implementation variants: one that directly stores index vectors, referred to as CIV, and another that stores index pointers, denoted as CIP. In the following discussion, the initial list at height 0 is denoted by L_0. The list generated after the i-th collision is denoted by L_i, where the XOR-value part of the list is denoted as X_i. For the index part, we denote it by IV_i and IP_i in the CIV and CIP algorithms, respectively. Most importantly, all the following results are evaluated under the in-place merge setting. Our proposed in-place merge framework is further discussed in Section 4.

3.1 New Trade-off for Index Vector

The trade-off strategy in [BK16] is suboptimal because it employs the k-tree algorithm in the second run. This necessitates a relatively large index trimming length ($t = 8$) to ensure that the time and memory complexity of the second run do not exceed those of the initial single-chain run. An alternative approach would be to reuse the single-chain algorithm for the second run. However, achieving both full index recovery and list size reduction in this setting is nontrivial.

Improving Index Trimming. Our key observation is that the partial solution obtained from the first run can be leveraged to reduce the size of layer L_i with height i greater than some $\hat{h}$ in the second run, thereby significantly lowering memory usage while still fully

recovering the index. Given a t-bit partial solution vector, in the second run we retain list items whose indices are consistent with the t-bit suffix (or prefix). For instance, consider a 1-bit partial solution vector $(0, 1, 1, 0, 1, 0, 1, 1)$. Here, the tuple represents the least significant bit of each index in the solution vector. During the first merge, any colliding pair whose indices have least significant bits equal to $(0, 0)$ is discarded, since the valid pair candidates induced by the partial solution vector $\{(0, 1), (1, 0), (1, 0), (1, 1)\}$ are restricted to $(0, 1)$, $(1, 0)$, and $(1, 1)$. Generally, at height h, enumerating all t-bit index combinations yields $(2^t)^{2^h}$ possible distinct index items. However, with the known t-bit solution vector, the number of valid index items is bounded by 2^{k-h}. Denote $\hat{h}(t, k)$ as:

$$\hat{h}(t, k) := \arg\min_h \left\{ 2^{t \cdot 2^h} > 2^{k-h} \right\}$$

The constraint is activated at height $\hat{h}(t, k)$ and beyond. Once activated, the combined effect of merge and the constraint leads to an exponential reduction in the size of subsequent lists. Given that $\hat{h}(t, k)$ is typically small (commonly 2 or 3 for $t = 1$), we can further employ the XOR-removal technique to reduce memory usage for layers with height $h \in [0, \hat{h}(t, k) - 1]$. Let t denote the index-trimming length and (h_1, h_2) the switching heights. Our trade-off strategy can be summarized in Algorithm 3.

Algorithm 3 $\mathsf{CIV\text{-}IT}(n, k, t, h_1, h_2)$: Index Vector with Index Trimming and XOR-Removal

Input: A list L of size $N = 2^{\frac{n}{k+1}+1}$.
Output: The full solution vector S_f.

1. **First run.** Run the single-chain algorithm as follows:
   (a) For $j \in [0, h_1 - 1]$, the layer $L^{(j)}$ stores only index vector list IV_j.
   (b) For $j \in [h_1, \quad k]$, the layer $L^{(j)}$ stores XOR list X_j and the trimmed index vector list $\mathsf{IV}_j^{(t)}$ with t-bit index.

   This yields a partial solution vector S_t of length $2^k \times t$ bits.

2. **Second run.** Run the single-chain algorithm with the constraint S_t as follows:
   (a) For $j \in [0, h_2 - 1]$, the layer $L^{(j)}$ stores only index vector list IV_i.[1]
   (b) For $j \in [h_2, \quad k]$, the layer $L^{(j)}$ stores XOR list X'_j and the full index vector list IV'_j where the corresponding bits of index match the partial solution vector S_t.

   This yields a full solution vector S_f of length $2^k \times (\ell + 1)$ bits.

We demonstrate that all parameters proposed in Equihash can be optimized to the case with index trimming length $t = 1$ and iterative depth 2. Without XOR-removal ($h_1 = 0$), the peak memory complexity occurs either at layer 0 or layer $k - 1$, which is given by

$$M = \max\left(n, 2^{k-1} + 2\ell\right) \cdot N$$

with a time complexity no more than $\mathcal{O}(2kN)$. For the case of $2^{k-1} + 2\ell < n$, the peak memory can be further reduced by allowing a slightly larger runtime overhead via XOR-removal. This refined trade-off achieves a lower peak memory usage than the single-chain algorithm with plain index-pointer technique (see Table 1). In the following, we provide a rigorous analysis of $\mathsf{CIV\text{-}IT}(n, k, t, h_1, h_2)$ algorithm along with a strategy for selecting the optimal trade-off parameters.

[1] The constraints are typically not activated at this stage. Consistency checks are not necessary here.

List Size of the Second Run. We denote the lists in the first run by $L_0, L_1, \ldots, L_k$ and in the second run by $L'_0, L'_1, \ldots, L'_k$. For lists L'_i with height $i < \hat{h}(t,k)$, their expected sizes remain close to N. Beyond and at height $\hat{h}(t,k)$, due to the constraints of partial solutions, the list size decreases exponentially under the effect of merge. Once the constraint is activated for $i > \hat{h}(t,k)$, the number of t-bit trimmed index in L'_i is at most 2^{k-i} and there are only $2^{(k-i)\cdot 2}$ possible indices in the following merge. Therefore, for next layer L'_{i+1}, the constraint scales the list size by a filtering factor of $\frac{2^{k-i}}{2^{(k-i+1)\cdot 2}}$. More precisely, the size $\mathcal{S}(i)$ of list L'_i follows recurrence:

$$\mathcal{S}(i) = \begin{cases} N, & i \in [0, \hat{h}(t,k) - 1] \\ \frac{2^{k-i}}{2^{t\cdot 2^i}} \cdot \frac{\mathcal{S}(i-1)\cdot(\mathcal{S}(i-1)-1)}{2\cdot 2^\ell}, & i = \hat{h}(t,k) \\ \frac{2^{k-i}}{2^{(k-i+1)\cdot 2}} \cdot \frac{\mathcal{S}(i-1)\cdot(\mathcal{S}(i-1)-1)}{2\cdot 2^\ell}, & i \in [\hat{h}(t,k)+1, k-1] \end{cases} \tag{S1}$$

The sequence $\mathcal{S}(i)$ closely matches the list sizes produced by our proof-of-concept implementation (see Table 4). The item size of list L_i and L'_i under different techniques is given by:

$$\begin{cases} \mathbf{IS}_f(n,k,i) = 2^i \cdot (\ell+1) + n - i \cdot \ell, & \text{full index vector + XOR part} \\ \mathbf{IS}_x(n,k,i) = 2^i \cdot (\ell+1), & \text{full index vector} \\ \mathbf{IS}_t(n,k,i,t) = 2^i \cdot t + n - i \cdot \ell, & \text{trimmed index vector + XOR part} \end{cases}$$

Peak Memory. The peak memory usage of the constrained second run of the single-chain algorithm can easily be lower than that of the first run. In most cases, it suffices to set the switching height h_2 to $\hat{h}(t,k)$ or $\hat{h}(t,k)+1$. Hence, we restrict our attention to the lower bound of the peak memory usage in the initial run. In particular, by setting trimming length $t = 1$, the peak memory required by each layer in $\mathsf{CIV\text{-}IT}(n,k,1,h_1,h_2)$ is given as follows:

$$M(i) = \begin{cases} \mathbf{IS}_x(n,k,i) \cdot N = 2^i \cdot (\ell+1) \cdot N, & i \in [0, h_1 - 1] \\ \mathbf{IS}_t(n,k,i,1) \cdot N = (2^i + n - i \cdot \ell) \cdot N, & i \in [h_1, k-1] \end{cases}$$

- For $i \in [0, h_1 - 1]$, $M(i) = 2^i \cdot (\ell+1) \cdot N$ strictly increases. Therefore, the peak memory usage in the former part is $M_1 = 2^{h_1 - 1} \cdot (\ell+1) \cdot N$.
- For $i \in [h_1, k-1]$, it is straightforward to observe that the minimum of $M(i) = (2^i + n - i \cdot \ell) \cdot N$ occurs at $\log_2(\ell / \ln 2)$. Therefore, the peak memory usage in the latter part is $M_2 = \max(M(h_1), M(k-1))$.

The lower bound of peak memory for algorithm $\mathsf{CIV}(n,k,1,h_1,h_2)$ is given by:

$$M = \max(M_1, M_2) \geq M(k-1) = (2^{k-1} + 2\ell) \cdot N.$$

By choosing an appropriate value of h_1, the theoretical peak memory can reach $\mathcal{M} = (2^{k-1} + 2\ell) \cdot N$. In particular, when $nN < \mathcal{M}$, the XOR-removal technique is unnecessary ($h_1 = 0$) in the first run, which typically applies when k is relatively large.

In Table 2, to mitigate the excessive runtime overhead introduced by the XOR-removal technique, we set the upper bounds of h_1 and h_2 to $\lceil \log_2 k \rceil$. Our search strategy first determines the optimal switching height h_1, and subsequently identifies the optimal switching height h_2 in the second run. Since the overall search space is relatively small, an exhaustive search remains computationally feasible. For all valid Equihash parameters, we report the optimal configurations that minimize peak memory usage toward $\mathcal{M}$, along with the corresponding runtime for each stage.

Table 2: Concrete Index Vector Trade-off for $\mathbf{GBP}(n, 2^k)$

$(n, 2^k)$	t	Mem	**Time**	First Run				Second Run				
				h_1	Mem_1	**Time**	ℓ_1	h_2	Mem_2	**Time**	ℓ_2	$\hat{h}(n,k)$
$(96, 2^5)$	1	$2^{23.09}$	$2.30 \cdot T_0$	2	$2^{23.09}$	$1.40 \cdot T_0$	2	2	$2^{23.04}$	$0.90 \cdot T_0$	2	2
$(128, 2^7)$	1	$2^{23.58}$	$3.15 \cdot T_0$	3	$2^{23.58}$	$1.86 \cdot T_0$	6	3	$2^{23.09}$	$1.29 \cdot T_0$	2	3
$(160, 2^9)$	1	$2^{25.17}$	$1.36 \cdot T_0$	0	$2^{25.17}$	$1.00 \cdot T_0$	8	0	$2^{24.61}$	$0.36 \cdot T_0$	2	3
$(96, 2^3)$	1	$2^{30.70}$	$3.04 \cdot T_0$	2	$2^{30.70}$	$1.67 \cdot T_0$	2	2	$2^{30.64}$	$1.37 \cdot T_0$	1	2
$(144, 2^5)$	1	$2^{31.64}$	$2.30 \cdot T_0$	2	$2^{31.64}$	$1.40 \cdot T_0$	2	2	$2^{31.61}$	$0.90 \cdot T_0$	2	2
$(150, 2^5)$	1	$2^{32.70}$	$2.30 \cdot T_0$	2	$2^{32.70}$	$1.40 \cdot T_0$	2	2	$2^{32.67}$	$0.90 \cdot T_0$	2	2
$(192, 2^7)$	1	$2^{32.00}$	$3.15 \cdot T_0$	3	$2^{32.00}$	$1.86 \cdot T_0$	3	3	$2^{31.64}$	$1.29 \cdot T_0$	2	3
$(240, 2^9)$	1	$2^{33.25}$	$1.36 \cdot T_0$	0	$2^{33.25}$	$1.00 \cdot T_0$	8	0	$2^{33.19}$	$0.36 \cdot T_0$	2	3
$(96, 2^2)$	1	$2^{39.04}$	$1.75 \cdot T_0$	1	$2^{39.04}$	$1.00 \cdot T_0$	1	1	$2^{39.02}$	$0.75 \cdot T_0$	1	1
$(288, 2^8)$	1	$2^{40.64}$	$2.89 \cdot T_0$	3	$2^{40.64}$	$1.75 \cdot T_0$	3	3	$2^{40.04}$	$1.14 \cdot T_0$	2	3
$(200, 2^9)$	1	$2^{29.21}$	$1.36 \cdot T_0$	0	$2^{29.21}$	$1.00 \cdot T_0$	8	0	$2^{28.93}$	$0.36 \cdot T_0$	2	3

t: trimmed length T_0: runtime of plain CIV $\hat{h}(n,k)$: activating height
h_1, h_2: switching height ℓ_1, ℓ_2: layer height of peak memory

Implications for k-Tree Algorithm. The work of [BLN+09] originally proposed the index-trimming technique for the k-tree algorithm, but did not provide a detailed analysis of its memory complexity. We address this gap in the present work. In the first run of k-tree algorithm, store only t-bit index. Then we recover the full index vector solution in the second run of the k-tree algorithm with the list size reduced to $2^{\ell-t}$. Typically, we can store only one bit of the index during the first run. In the second run, where the initial list size is reduced to $2^{\ell-1}$, the lists at height h will have a size of

$$S(h) = 2^{\ell - 2^h} \tag{S2}$$

as ℓ bits must collide in each merge (see Table 4 for a concrete example). As a result, the complexity of the second run will be significantly lower than that of the first run. The following proposition presents the optimized peak memory of k-tree algorithm.

Proposition 3. *The k-tree algorithm with index trimming length $t = 1$ achieves a peak memory of $M = \left(\frac{k^2+5k+2}{4}\ell + 2^{k-1}\right) \cdot N$ with a time complexity of at most $\mathcal{O}\left(2^{k+1}N\right)$.*

Proof. Since all the lists form a complete binary tree, the k-tree algorithm can be performed in post-order using stack to reduce peak memory [Wag02]. Therefore, the peak memory consumption occurs during the merging of the last two leaves, wherein exactly one list exists at each layer except for the root (0 list) and layer 0 (2 lists). In layer i, the list element consists of $2^i \times \ell$-bit (trimmed to 2^i-bit) indices and a temporary XOR-result of $n - i\ell = (k + 1 - i)\ell$ bit. The peak memory measured in bits is given by:

$$\begin{aligned} M &= \underbrace{((k+1)\cdot\ell + 1)\cdot\frac{N}{2}}_{\text{Layer } 0} + \underbrace{\sum_{i=0}^{k-1}\left(2^i + (k+1-i)\cdot\ell\right)\cdot\frac{N}{2}}_{\text{Layer } 0, 1, \ldots, k-1} \\ &= \left(\frac{(k+1)}{2}\cdot\ell + \frac{1}{2} + 2^{k-1} - \frac{1}{2} + \frac{k(k+3)}{4}\cdot\ell\right)\cdot N \\ &= \left(\frac{k^2+5k+2}{4}\ell + 2^{k-1}\right)\cdot N \end{aligned}$$

In principle, the initial list at Layer 0 does not need to store trimmed indices. Under this assumption, the peak memory usage can be further reduced by $\frac{N}{2} \cdot 2 = N$ bits. □

3.2 New Trade-off for Index Pointer

The index-pointer technique itself substantially reduces the peak memory of the single-chain algorithm. In this section, we discuss how our new trade-off technique can reduce the peak memory of CIP from approximately $2nN$ to nN.

Plain Post Retrieval. Under the index-pointer technique, the XOR-value list X_i and the index-pointer list IP_i both depend solely on the XOR-value list X_{i-1} from the previous layer. That is, the index pointer IP is neither a required input nor an essential output of the merge operation. This observation implies that the candidate index pointers in IP_k can be obtained without storing $\mathsf{IP}_1, \mathsf{IP}_2, \ldots, \mathsf{IP}_{k-1}$. Subsequently, by repeatedly executing a round-reduced single-chain algorithm, one can recursively reconstruct the index pointers IP_j for $j \in [1, k-1]$ layer by layer to recover the complete solution vector. Thus, the peak memory consumption is $n \cdot N$ bits (size of X_0) which achieves a nearly twofold reduction compared to the plain CIP Algorithm 2 requiring $2(n+k-\ell-1)N$ bits. The overall time complexity is dominated by the number of merge operations, namely $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$, leading to a time complexity of $T = \frac{k(k+1)}{2} \cdot N$. Nevertheless, the time penalty factor can be further reduced while keeping the peak memory at nN bits.

Advanced Post Retrieval. A more advanced trade-off strategy is to start storing index pointers from a certain layer h as the Step 2 of our Algorithm 4, which leads to a peak memory of

$$\begin{aligned} M_{ip}(h) &= (n - h \cdot \ell)\, N + (k - 1 - h)\,(\ell + 2)\, N \\ &= (n + (k - 1 - 2h)\ell + 2(k - 1 - h))\, N \end{aligned}$$

Let $\widetilde{h} = \arg\min_h \{M_{ip}(h) < nN\}$. Then CIP-PR$(n, k, h)$ for $h > \widetilde{h}$ attains a peak memory of roughly nN bits. The corresponding time penalty is formally analyzed in the following proposition.

Algorithm 4 CIP-PR(n, k, h): Index Pointer with Post Retrieval

Input: A list L of size N.
Output: The full solution vector IP^k.

1. Run an h-round single-chain algorithm without index part, the involved lists in each layer are $X_0, X_1, \ldots, X_h$. The peak memory occurs at X_0.
2. Run a $(k-h)$-round single-chain algorithm with index pointer and recover IP_k. The peak memory occurs in the $(k-1)$-th layer: $\mathsf{IP}_{h+1}, \mathsf{IP}_{h+2}, \ldots, \mathsf{IP}_{k-1}, X_{k-1}$.
3. Use $\mathsf{IP}_{h+1}, \mathsf{IP}_{h+2}, \ldots, \mathsf{IP}_k$ to backtrack index pointers and recover partial solution list containing $\mathsf{IP}^{(k-h)} = (i_1^{(k-h)}, i_2^{(k-h)}, \cdots, i_{2^{k-h}}^{(k-h)})$. Remove $\mathsf{IP}_{h+1}, \mathsf{IP}_{h+2}, \ldots, \mathsf{IP}_k$.
4. **For** $j = h, h-1, \ldots, 1$ **do** ▷ **Post Retrieval**: recover $\mathsf{IP}_h, \mathsf{IP}_{h-1} \cdots, \mathsf{IP}_1$
   (a) Run a j-round single-chain algorithm: $X_0 \implies \cdots \implies X_{j-1} \implies \mathsf{IP}_j$.
   (b) Expand $\mathsf{IP}^{(k-j)}$ into $\mathsf{IP}^{(k-j+1)} = (i_1^{(k-j+1)}, \ldots, i_{2^{k-j+1}}^{(k-j+1)})$ using IP_j.

Proposition 4. *Algorithm 4 runs in peak memory* $\max(M_{ip}(h), n \cdot N)$ *bits. By setting* $h \geq \widetilde{h} = \frac{k+1}{2}$*, it can achieve a peak memory of* $n \cdot N$ *bits in time complexity* $T = \Delta + T_0$ *where* $T_0 = kN$ *is the plain time complexity of CIP and the time penalty* Δ *is bounded by:*

$$\frac{(k-1)(k+1)}{8} N < \Delta < \frac{(k+1)(k+3)}{8} N.$$

Proof. To ensure that the single-chain algorithm yields a solution with constant probability, the parameter k must be smaller than ℓ (see Theorem 4 in [TSG25]). Then one can easily verify that $M_{ip}(\frac{k-1}{2}) > nN$ and $M_{ip}(\frac{k+1}{2}) < nN$. Consequently, $\widetilde{h}$ is bounded in $\left(\frac{k-1}{2}, \frac{k+1}{2}\right)$. The time penalty introduced by Step 4 of Algorithm 4 is given by

$$\Delta = \sum_{i=1}^{\widetilde{h}} iN = \frac{\widetilde{h}(\widetilde{h}+1)}{2} N \in \left(\frac{(k-1)(k+1)}{8} N,\ \frac{(k+1)(k+3)}{8} N \right).$$

□

By choosing $h < \widetilde{h}$, the peak memory consumption of Algorithm 4 becomes slightly higher than nN, yet it remains lower than that of the original CIP algorithm. The time-memory trade-off behaves almost linearly, but its theoretical minimum peak memory is bounded below by nN. In addition, Step 4 of Algorithm 4 can alternatively be carried out in the following ways:

- Directly run an h-round single-chain algorithm with index pointer. This makes the time penalty factor $\gamma < 2$ and the theoretical peak memory in $(nN, 2nN)$.
- Apply switching heights recursively to further reduce the time penalty. Find a new switching height $\widetilde{h}' \in [1, \widetilde{h}]$ and rerun a similar CIP-PR$(n, \widetilde{h}', \widetilde{h})$ algorithm.

Applying our algorithm to Equihash, we ensure that the peak memory reaches the theoretical minimum of nN. The peak memory and time penalty are summarized in Table 1.

Remark 3 (External Memory Design). In practice, the runtime overhead can be further alleviated (or even eliminated) by adopting I/O-efficient designs, such as external-memory caching. While running a k-round CIP algorithm, the index-pointer lists $\mathsf{IP}_1, \mathsf{IP}_2, \cdots, \mathsf{IP}_{k-1}$ can be cached in external memory. Subsequently, extracting the remaining $k-1$ layers of index pointers only requires $\mathcal{O}(2^k)$ random external-memory accesses, whose time cost is negligible in practice. **Therefore, the additional runtime overhead comes solely from $k-1$ sequential write operations of size N from memory to external storage.** Our proof-of-concept implementation of CIP with SSD-based external storage reduces the peak memory of popular Equihash$(144, 5)$ by half with negligible I/O penalty! Given sufficient memory bandwidth, this may enable more efficient ASIC implementations. Some detailed proof-of-concept experiments are provided in Table 5.

3.3 A Hybrid Algorithmic Framework

We observe that different optimization techniques are best suited to different layer heights. For instance, the XOR-removal technique achieves the most favorable time-memory trade-off when applied to low-height layers, whereas the index-trimming technique performs particularly well at higher layers. This motivates us to consider a **hybrid optimization framework** in Figure 3.

- **Implementation switching**: At any height, one can switch from index pointer implementation to index vector implementation, and vice versa.
- **Optimization switching**: At any height, one can switch from plain index vector (or index pointer) to index trimming (or post-retrieval). Moreover, the XOR-removal technique applies to both the naive index pointer and index vector implementations.

As a result, the k-chain algorithm admits a highly complex set of possible implementations (at least four distinct implementation strategies in Figure 3). The two trade-off techniques that do not involve implementation switching have already been discussed in the

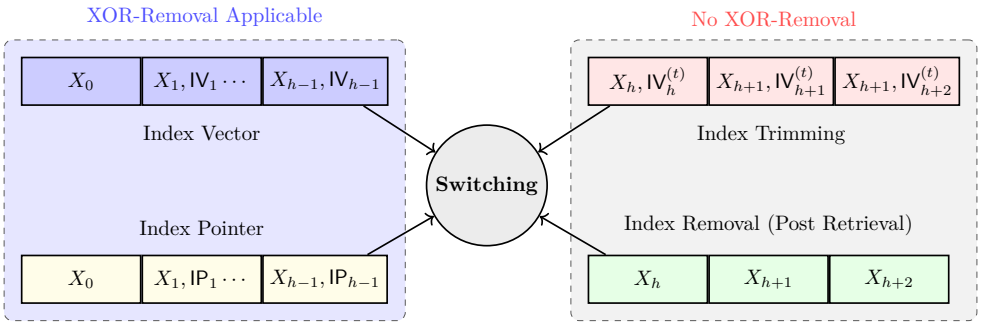


Figure 3: A Hybrid Algorithmic Framework for Wagner's Algorithms

previous sections. In Algorithm 5, we investigate how to combine different implementation strategies to achieve a lower peak memory usage. Before proceeding, we emphasize the following two facts:

- **XOR-Removal for Index Pointer**: In the context of (optimization) switching, the XOR-removal technique is not applicable to index pointer, as it does not lead to any reduction in peak memory usage. For example, when switching to post-retrieval at height h, one cannot free $\mathsf{IP}_1, \mathsf{IP}_2 \ldots, \mathsf{IP}_h$ while simultaneously generating X_h. As a result, the peak memory still includes $\mathsf{IP}_1, \mathsf{IP}_2, \ldots, \mathsf{IP}_h, X_h$.
- **Index Vector vs Index Pointer**: When the layer height does not exceed 2, the storage cost of IP and IV representations is identical. However, the IV representation is more favorable for applying optimization techniques such as XOR-Removal.

The difference between the hybrid algorithm and the previous CIP-PR algorithm 4 lies in the use of the index vector and XOR-removal techniques at the lower layers, which enables the peak memory to be reduced to below nN. The peak memory of Algorithm 5 is as follows:

$$M = \max\left(\underbrace{2^{h_1}\cdot(\ell+1)\cdot N}_{\text{Step 1}}, \underbrace{(n-h_1\cdot\ell)\cdot N}_{\text{Step 2}}, \underbrace{((k-1-h_2)\cdot 2\cdot(\ell+1)+2\ell)\cdot N}_{\text{Step 3}}\right)$$

since Step 4 and Step 5 have peak memory not greater than Step 3 and Step 1,2, respectively. In Table 1, to avoid the excessive time penalty introduced by XOR-removal, we choose $h_1 \leq 2$. In addition, to ensure that the peak memory remains below nN, we set $h_2 > (k-1)/2$. The best trade-off in Table 1 is estimated with time penalty factor $\gamma \leq 4$.

3.4 Summary of New Trade-offs.

The proposed trade-offs are all techniques for list item reduction, and thus the achievable memory reduction is inherently limited. Our techniques can be combined with list size reduction techniques, such as Dinur's framework [Din19], to provide a more advanced trade-off. We briefly summarize our techniques as follows:

1. **Single-Chain**: In the single-chain setting, the plain index-pointer implementation is simpler and more efficient. Since XOR-removal is incompatible with index pointer, the peak memory lower bound for index-pointer is exactly nN. Although the trade-off based on index vectors is more complex in terms of implementation and memory organization, its theoretical peak memory usage can be lower than nN when combined with XOR-removal technique. **In general, our new technique provides**

Algorithm 5 Hybrid-Single-Chain(n, k, h_1, h_2)

Input: A list L of size N.
Output: The full solution vector IP^k.

1. **Index Vector + XOR-Removal**: Run an h_1-round single-chain algorithm, the involved lists in each layer are $\mathsf{IV}_1, \mathsf{IV}_2, \ldots, \mathsf{IV}_{h_1}$. The peak memory occurs at IV_{h_1}.
2. **Index Removal**: Run an $(h_2 - h_1)$-round single-chain algorithm without index part, the involved lists in each layer are $X_{h_1+1}, \cdots, X_{h_2}$. The peak memory occurs in the (h_1+1)-th layer: X_{h_1+1}.
3. **Index Pointer**: Run a $(k - h_2)$-round single-chain algorithm with index pointer, the involved lists in each layer are $X_{h_2}, (X_{h_2+1}, \mathsf{IP}_{h_2+1}) \cdots, (X_{k-1}, \mathsf{IP}_{k-1}), \mathsf{IP}_k$. The peak memory occurs in the $(k-1)$-th layer: $\mathsf{IP}_{h_2+1}, \mathsf{IP}_{h_2+2}, \ldots, \mathsf{IP}_{k-1}, X_{k-1}$.
4. Use $\mathsf{IP}_{h_2+1}, \mathsf{IP}_{h_2+2}, \ldots, \mathsf{IP}_k$ to backtrack index pointers and recover partial solution list containing $\mathsf{IP}^{(k-h_2)} = (i_1^{(k-h_2)}, i_2^{(k-h_2)}, \cdots, i_{2^{k-h_2}}^{(k-h_2)})$. Remove $\mathsf{IP}_{h_2+1}, \ldots, \mathsf{IP}_k$.
5. **For** $j = h_2, h_2 - 1, \ldots, h_1 + 1$ **do** ▷ **Post Retrieval**: recover $\mathsf{IP}_{h_2}, \ldots, \mathsf{IP}_{h_1+1}$
   (a) Run a j-round single-chain algorithm to recover IP_j. The involved lists are:
   $$\underbrace{\mathsf{IV}_1 \Longrightarrow \mathsf{IV}_2 \Longrightarrow \cdots \Longrightarrow \mathsf{IV}_{h_1}}_{\text{Index Vector + XOR-Removal}} \Longrightarrow \underbrace{X_{h_1+1} \Longrightarrow \cdots \Longrightarrow X_{j-1}}_{\text{Index Removal}} \Longrightarrow \mathsf{IP}_j$$
   (b) Expand $\mathsf{IP}^{(k-j)}$ into $\mathsf{IP}^{(k-j+1)}$ using IP_j.
6. Run an h_1-round single-chain algorithm to recover IV_{h_1}. The involved lists are $\mathsf{IV}_1, \ldots, \mathsf{IV}_{h_1}$. Use IV_{h_1} to expand $\mathsf{IP}^{(k-h_1)}$ into full solution indices IP^k.

an efficient implementation framework for the single-chain algorithm under an $\mathcal{O}(nN)$ bits of memory.

2. **k-Tree**: In the context of the k-tree algorithm, the index-trimming technique is highly effective, whereas the index-pointer technique instead incurs a significant memory increase since it requires storing 2^k index-pointer lists simultaneously. Notably, while the index-trimming technique proves highly effective, the k-tree algorithm exhibits an asymptotic memory complexity of $\mathcal{O}\left((2^k + n)N\right)$. As k increases, its memory cost becomes substantially higher than that of the single-chain variant. For instance, when evaluating preimage attacks of incremental hashes [BGG94, BM97, GSC01, MGS15], the optimal k is $\sqrt{n}$. In this case, the asymptotic memory complexity of k-tree algorithm is greater than that of single-chain algorithm by a factor of $2^{\sqrt{n}}$. **Whether the list-item reduction technique can eliminate the 2^k factor remains a highly meaningful open problem.**

4 Experiments and Impacts on Equihash

We developed a fine-grained memory complexity evaluator for Wagner's algorithms, which incorporates all the trade-off frameworks proposed in this work. Furthermore, our proof-of-concept implementations confirm the practical feasibility of our approach. Our accompanying code is publicly available at:

https://github.com/tl2cents/Wagner-Algorithms

Remark 4 (Time Unit). In GBP or Equihash, the cost of generating the initial list (by a hash function or read directly from external memory) is considered equivalent to that of a single merge operation. For instance, the time cost of performing N hash computations is comparable to the $\mathcal{O}(N)$ cost of a merge operation. However, the actual time ratio between these operations depends on the specific implementation and context. This discrepancy is one of the main reasons why practical performance may deviate from theoretical estimates.

4.1 Practical In-place Merge

Not-in-place merging implementations may consume up to twice the theoretical peak memory predicted for Equihash [BK16]. If a not-in-place merge implementation is used, merging L_0 into L_1 requires approximately $2nN$ bits of memory, corresponding to the cost of $X_0+X_1+\mathsf{IP}_1$. Since the theoretical memory of CIP implementations [Equ16, Tro16] are close to $2nN$ bits, carefully designed memory layouts allow not-in-place merge implementations to avoid significantly increasing the peak memory. However, to reduce the peak memory footprint to nN bits, an in-place merge implementation becomes indispensable.

In-place Merge Framework. To the best of our knowledge, neither existing implementations nor prior academic studies have provided a detailed investigation of in-place merge. Our work presents the first in-place implementation of merge. Given a list L of size N, any in-place sorting algorithm can be utilized to construct an in-place $\mathsf{merge}_\ell(L)$ operation.

1. **In-place Sort**: Sort the list L in place according to ℓ bits of XOR value in list item.
2. **Linear Scan**: Initialize a temporary array T and sequentially scan the sorted list L:
   (a) Identify groups of elements sharing the same key value, and for each group, pair the elements to form collisions, and store the resulting pairs in T.
   (b) Reuse the freed memory space in L by writing back as many elements from T as possible into L.

Table 3: In-Place Merge for Equihash(200, 9) in Layer 0: $X_0 \implies X_1$.

Phase	Metric	Value	
		Avg	Max
In-place Sort [Vou]	Runtime	0.059 s	0.065 s
Linear Scan	Runtime	0.027 s	0.033 s
	Max Size of T	66.2	120
In-place Merge	Theo. Memory Peak	50.0 MB	
	Actual Memory Peak	51.6 MB	

To perform a merge: $L_0 \implies L_1$, our framework runs in memory of approximately $\max(|L_0|, |L_1|)$. We implemented the in-place merge in C++, where an in-place MSD radix sort is adopted. Assuming a uniform random distribution, we have $\mathbb{E}[\#\{x = k\}] = 1$ for collided key value $k \in [0, 2^\ell - 1]$. Thus, the max size of temporary list T is negligible compared to the total size N. In Table 3, the max size of T is 120 which can effectively be treated as constant-space overhead. Our implementation does not employ aggressive optimizations. Several more efficient and parallelizable in-place sorting algorithms [El-05, CBB+15, OKFS19] might further improve performance. Given the uniform distribution, it remains an open problem whether more efficient in-place collision search techniques (such as the radix tree structure in [TID21]) could offer additional improvements.

4.2 Proof-of-Concept Implementations

In this section, we primarily demonstrate the effectiveness of the improved index-trimming and post-retrieval techniques. We consider two Equihash parameter settings that are widely deployed in practice: Equihash$(200, 9)$, used by the Zcash blockchain, and Equihash$(144, 5)$, adopted by Bitcoin Gold. Notably, for Equihash$(200, 9)$, it required a peak memory usage of at least 522 MB in [BK16]. Tromp's implementation [Tro16] reduced this requirement to approximately 144 MB, whereas our optimized implementation further lowers it to only 58 MB. As a result, Equihash$(200, 9)$ has largely lost its intended ASIC-resistance.

Correctness of Index Trimming. Table 4 presents the concrete list sizes in the second run of our proof-of-concept implementation with index trimming length $t = 1$. For the single-chain algorithm, the peak memory in the first run is $2^{29.2}$ bits, while the second run requires $2^{28.9}$ bits at height 2. Thus, the overall peak memory is reduced to $2^{29.2}$ compared to $2^{29.55}$ in Proposition 2. Similarly, for the k-tree algorithm, this trade-off reduces the peak memory complexity from $2^{33.5}$ to $2^{30.8}$. This theoretically demonstrates the feasibility of the index-trimming technique.

Table 4: Concrete List Size during the Second Run of Wagner's Algorithms with Single-bit Index Constraints for GBP$(200, 2^9)$. When the estimate given by S1 and S2 is smaller than the list size required for a single solution to exist, use the latter as the estimate.

Height		0	1	2	3	4	5	6	7	8
Single-Chain	Eq. S1	2^{21}	2^{21}	2^{21}	$2^{19.00}$	$2^{10.00}$	2^4	2^3	2^2	2^1
	Act.	2^{21}	2^{21}	2^{21}	$2^{18.88}$	$2^{10.07}$	2^4	2^3	2^2	2^1
k**-Tree**	Eq. S2	2^{19}	2^{18}	2^{16}	2^{12}	2^4	1	1	1	1
	Act.	2^{19}	$2^{18.00}$	$2^{16.01}$	$2^{12.02}$	2^4	1	1	1	1

Advanced Post Retrieval. To evaluate the proposed CIP-PR framework, we implemented both the *in-memory* and *external-memory* variants for Equihash$(200, 9)$ and Equihash$(144, 5)$. For comparison, we select Tromp's C++ implementation [Tro16] as the SoTA baseline, which is the runner-up in the Zcash open miner competition[2]. Tromp's implementation is incorporated into the official Zcash miners as well as Bitcoin Gold miners. Thus, adopting Tromp's implementation as the baseline offers a practical point of comparison. As shown in Table 5, our techniques reduce the peak memory usage of Wagner's algorithm by more than half (from $> 2nN$ bits to $\approx nN$ bits), while incurring no more than a twofold time penalty. In the plain CIP implementation, the primary memory savings compared to the state of the art stem from our in-place merge implementation.

Memory Layout of CIP-APR. Figure 4 presents the ideal memory layout and our implemented memory layouts with switching height $h = 5$ on Equihash$(200, 9)$, corresponding to Step 1 and Step 2 of Algorithm 4. In our implementation, for every merge operation that requires generating index pointers, we store an additional array $I_i = \{1, 2, \ldots, 2^{\ell+1}\}$ of element indices before performing the sorting step. This enables correct reconstruction of the index pointers but introduces an extra storage cost of $(\ell + 1)N$ bits compared to the theoretical memory layout at each layer in Figure 4. For ease of implementation, the CIP-APR instance for Equihash$(200, 9)$ reported in Table 5 incurs some additional memory overhead. First, we store indices in a byte-aligned manner, meaning that each index occupies 24 bits rather than the ideal $\ell + 1 = 21$ bits. Second, a minor difference from Figure 4 is that, in our implementation, the memory buffer for IP_8 is pre-allocated during

[2] https://zcashminers.org/submissions. The winners rely on assembly-level optimizations or GPU.

Table 5: Implementations of Post Retrieval Algorithm 4 for Equihash.

Implementation	Equihash(144, 5)			Equihash(200, 9)		
	Sol/s	Mem.	Mem. Ratio	Sol/s	Mem.	Mem. Ratio
CIP	0.23	1454 MB	0.57×	2.12	124 MB	0.85×
CIP-PR	0.08	707 MB	0.28×	0.44	58 MB	0.40×
CIP-APR	0.10	713 MB	0.28×	0.74	63 MB	0.43×
CIP-EM	0.22	707 MB	0.28×	1.93	59 MB	0.40×
SoTA [Tro16]	0.20	2570 MB	1.00×	6.30	146 MB	1.00×

[1] CIP-APR uses an optimal switching height $\frac{k+1}{2}$: 3 for Equihash(144, 5) and 5 for Equihash(200, 9).
[2] CIP-PR uses a switching height of $k-1$ and plain CIP uses a switching height of 0.
[3] Experiments were conducted on an Intel i7-13700K CPU. The performance of CIP-EM relies on the bandwidth between main memory and external storage (SSD).

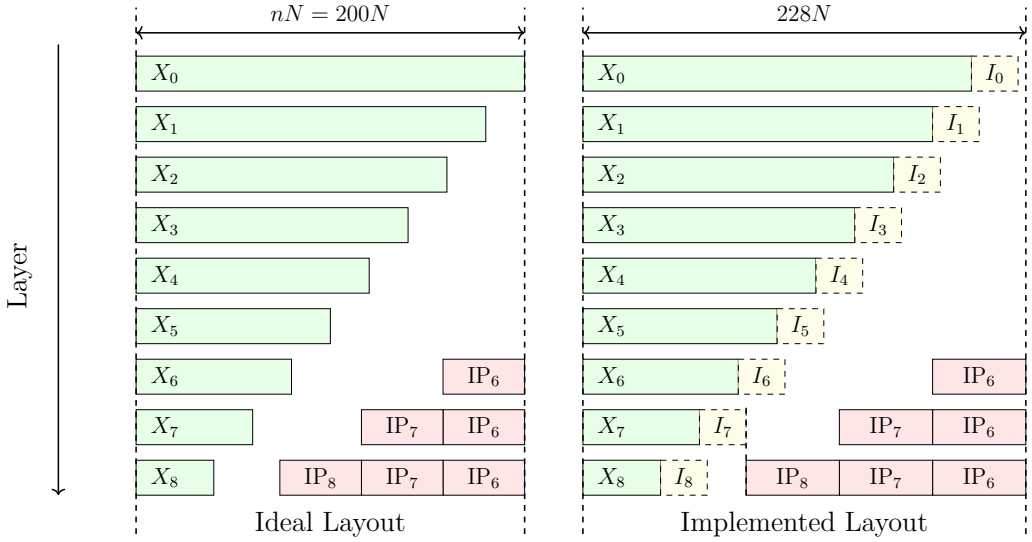


Figure 4: Memory layout of CIP-APR on Equihash$(200, 9)$ where I_i denotes the sequential index $\{1, 2, \ldots, 2^{\ell+1}\}$ for generating index pointers in the next layer.

the preparation phase of the merge from $X_7 \to X_8$. Consequently, the theoretical peak memory of CIP-APR in Table 5 is given by:

$$M = X_7 + I_7 + \mathsf{IP}_8 + \mathsf{IP}_7 + \mathsf{IP}_6.$$

For Equihash$(200, 9)$, the performance gap between our implementation and Tromp's [Tro16] is mainly attributed to the absence of low-level optimizations in our design. Tromp's solver applies aggressive optimizations such as hand-tuned memory layouts and optimized bucket sorting, whereas our PoC adopts the in-place sorting algorithm from [Vou] with fixed bucket size of 256 to ensure modularity.

Trade-off Curve of CIP-PR. As shown in Figure 5, the resulting time-memory trade-offs for all possible switching heights closely match our theoretical estimates. This empirical agreement validates the correctness of our proposed technique (Proposition 4) and demonstrates a concrete attack against the claimed steep time-memory trade-offs of Equihash, which are central to its ASIC-resistance.

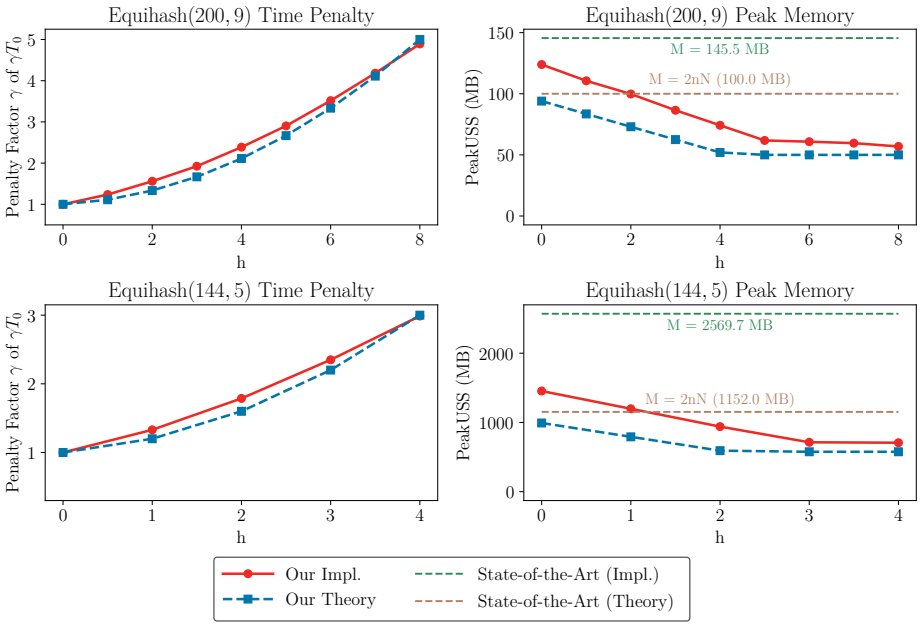


Figure 5: Concrete LIR Trade-offs for Equihash via Post-Retrieval

5 Conclusion and Future Work

Our work introduces a fine-grained memory optimization framework for Wagner's single-chain and k-tree algorithms, enabling nearly linear time-memory trade-offs within a constrained range. On one hand, this technique allows efficient implementations of Wagner's algorithms for cryptanalytic applications. On the other hand, it significantly weakens the memory hardness and ASIC resistance properties of Equihash. Several directions for future research and open problems are outlined below:

1. **Impact on k-tree algorithms.** Investigate how our proposed hybrid framework influences the k-tree algorithm, in particular whether the LIR technique can eliminate the 2^k memory factor inherent to the k-tree setting.

2. **Combining LIR and LSR techniques.** Explore the integration of our LIR approach with LSR-based trade-offs, which may theoretically yield a more efficient and flexible time-memory balance.

Acknowledgments.

We sincerely thank the reviewers of CHES 2026 for their insightful comments, which helped improve the quality of this work. This work was supported by the National Natural Science Foundation of China (Grant No. 12441106).

References

[AFS03] Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A fast provably secure cryptographic hash function. Cryptology ePrint Archive, Report 2003/230, 2003.

[Ber07] Daniel J Bernstein. Better price-performance ratios for generalized birthday attacks. In *Workshop Record of SHARCS*, volume 7, page 160, 2007.

[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.

[BGHZ19] Xiaofei Bai, Jian Gao, Chenglong Hu, and Liang Zhang. Constructing an adversary solver for Equihash. In *NDSS 2019*. The Internet Society, February 2019.

[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.

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, July 2003.

[BLL+21] Fabrice Benhamouda, Tancrède Lepoint, Julian Loss, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 33–53. Springer, Cham, October 2021.

[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.

[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.

[CBB+15] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent KulandaiSamy, and Ruchir Puri. PARADIS: an efficient parallel algorithm for in-place radix sort. *Proc. VLDB Endow.*, 8(12):1518–1529, 2015.

[Din19] Itai Dinur. An algorithmic framework for the generalized birthday problem. *DCC*, 87(8):1897–1926, 2019.

[DRX17] Srinivas Devadas, Ling Ren, and Hanshen Xiao. On iterative collision search for LPN and subset sum. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 729–746. Springer, Cham, November 2017.

[El-05] Fouad El-Aker. Fast in-place integer radix sorting. In Vaidy S. Sunderam, G. Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Computational Science - ICCS 2005, 5th International Conference, Atlanta, GA, USA, May 22-25, 2005, Proceedings, Part III*, volume 3516 of *Lecture Notes in Computer Science*, pages 788–791. Springer, 2005.

[Equ16] Improved Equihash solver using index pointer. https://github.com/xenoncat/equihash-xenon, 2016.

[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.

[HBHW17] Daira-Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, 2017.

[LF06] Éric Levieil and Pierre-Alain Fouque. An improved LPN algorithm. In Roberto De Prisco and Moti Yung, editors, *SCN 06*, volume 4116 of *LNCS*, pages 348–359. Springer, Berlin, Heidelberg, September 2006.

[MGS15] Hristina Mihajloska, Danilo Gligoroski, and Simona Samardjiska. Reviving the idea of incremental cryptography for the zettabyte era use case: Incremental hash functions based on SHA-3. Cryptology ePrint Archive, Report 2015/1028, 2015.

[MS12] Lorenz Minder and Alistair Sinclair. The extended k-tree algorithm. *Journal of Cryptology*, 25(2):349–382, April 2012.

[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.

[OKFS19] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. Theoretically-efficient and practical parallel in-place radix sorting. In Christian Scheideler and Petra Berenbrink, editors, *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 213–224. ACM, 2019.

[Sch01] Claus-Peter Schnorr. Security of blind discrete log signatures against interactive attacks. In Sihan Qing, Tatsuaki Okamoto, and Jianying Zhou, editors, *ICICS 01*, volume 2229 of *LNCS*, pages 1–12. Springer, Berlin, Heidelberg, November 2001.

[Sha08] Andrew Shallue. An improved multi-set algorithm for the dense subset sum problem. In *International Algorithmic Number Theory Symposium*, pages 416–429. Springer, 2008.

[TID21] Monika Trimoska, Sorina Ionica, and Gilles Dequen. Time-memory analysis of parallel collision search algorithms. *IACR TCHES*, 2021(2):254–274, 2021.

[Tro16] John Tromp. Multi-parameter Equihash proof-of-work with multi-threaded C solvers. https://github.com/tromp/equihash, 2016.

[TSG25] Lili Tang, Yao Sun, and Xiaorui Gong. Revisiting the generalized birthday problem and Equihash: single or K lists? Cryptology ePrint Archive, Paper 2025/1351, 2025.

[Vou] Voutcn. kxsort: fast in-place radix sort with STL-like API. https://github.com/voutcn/kxsort. Accessed: 2025-10-11.

[vOW99] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, January 1999.

[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.