Designs, Codes and Cryptography (2019) 87:1897–1926
https://doi.org/10.1007/s10623-018-00594-6

An algorithmic framework for the generalized birthday problem

Itai Dinur[1]

Received: 2 July 2018 / Revised: 6 December 2018 / Accepted: 7 December 2018 /
Published online: 17 December 2018

Abstract
The generalized birthday problem (GBP) was introduced by Wagner in 2002 and has shown to have many applications in cryptanalysis. In its typical variant, we are given access to a function $H : \{0, 1\}^\ell \to \{0, 1\}^n$ (whose specification depends on the underlying problem) and an integer $K > 0$. The goal is to find K distinct inputs to H (denoted by $\{x_i\}_{i=1}^K$) such that $\sum_{i=1}^K H(x_i) = 0$. Wagner's K-tree algorithm solves the problem in time and memory complexities of about $N^{1/(\lfloor \log K \rfloor + 1)}$ (where $N = 2^n$). In this paper, we improve the best known GBP time-memory tradeoff curve (published independently by Nikolić and Sasaki and also by Biryukov and Khovratovich) for all $K \geq 8$ from $T^2 M^{\lfloor \log K \rfloor - 1} = N$ to $T^{\lceil (\log K)/2 \rceil + 1} M^{\lfloor (\log K)/2 \rfloor} = N$, applicable for a large range of parameters. We further consider values of K which are not powers of 2 and show that in many cases even more efficient time-memory tradeoff curves can be obtained. Finally, we optimize our techniques for several concrete GBP instances and show how to solve some of them with improved time and memory complexities compared to the state-of-the-art. Our results are obtained using a framework that combines several algorithmic techniques such as variants of the Schroeppel–Shamir algorithm for solving knapsack problems (devised in works by Howgrave-Graham and Joux and by Becker, Coron and Joux) and dissection algorithms (published by Dinur, Dunkelman, Keller and Shamir).

Keywords Cryptanalysis · Time-memory tradeoff · Generalized birthday problem · K-tree algorithm

Mathematics Subject Classification 94A60

Communicated by A. Winterhof.

✉ Itai Dinur
dinuri@cs.bgu.ac.il

1 Department of Computer Science, Ben-Gurion University, Beersheba, Israel

1 Introduction

The generalized birthday problem (GBP) is a generalization of the classical birthday problem of finding a collision between two elements in two lists, introduced by Wagner in 2002 [15]. Since its introduction, Wagner's K-tree algorithm for GBP has become a widely applicable tool used in cryptanalysis of code-based cryptosystems [2] (that are important designs in post-quantum cryptography), hash functions (such as FSB [4]) and stream ciphers, where it is used as a procedure in fast correlation attacks [6,10]. Furthermore, it is an important component in improved algorithms for hard instances of the knapsack problem [1,9].

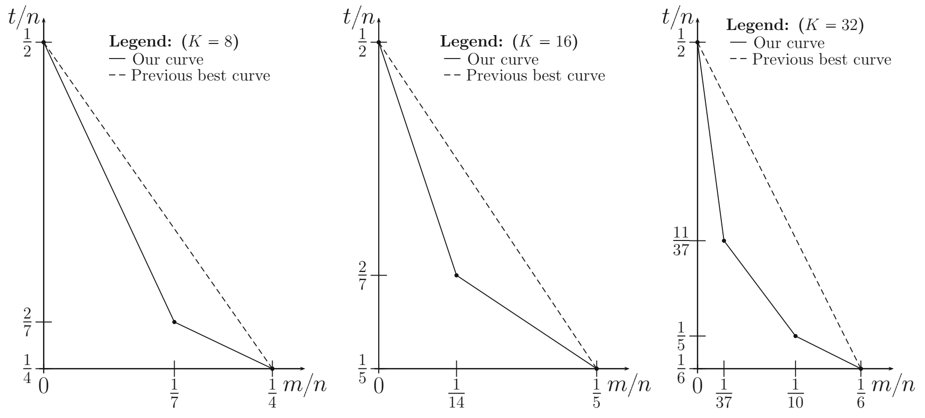
We consider the most relevant GBP variant in cryptanalysis. For integer parameters $K > 0$ and $0 \leq \ell \leq n$, we are given access to a function $H : \{0,1\}^{\ell} \rightarrow \{0,1\}^n$, and the goal is to find K distinct inputs to H, $\{x_i\}_{i=1}^{K}$, such that $\sum_{i=1}^{K} H(x_i) = 0$. For simplicity, we assume that addition is performed bitwise over $GF(2)$, but our algorithms easily extend to work with addition over $GF(2^n)$. We view H as a random oracle whose outputs are selected independently and uniformly at random from $\{0,1\}^n$. The number of K-tuples over ℓ-bit words is about $2^{K\ell}$, and as the problem places an n-bit constraint on the solution, the expected number of solutions is $2^{K\ell-n}$. In particular, we expect a solution only if $K\ell \geq n$. For $K \geq 2$, the problem can be solved in time $2^{n/2}$ using a simple collision search. Wagner's observation was that for values of $K \geq 4$, the problem can be solved much more efficiently assuming that the number of expected solutions is sufficiently large.

As a specific application of GBP to cryptanalysis, we consider the problem of breaking a code-based cryptosystem which can typically be reduced to solving a syndrome decoding (SD) problem. The input to the SD problem consists of a matrix $B \in \{0,1\}^{n \times m}$, a word $s \in \{0,1\}^n$ and an integer $w > 0$. The goal is to find a word $e \in \{0,1\}^m$ of Hamming weight (at most) w such that $Be = s$. In other words, we are looking for w columns of B that sum to s over $GF(2^n)$. We can reduce this problem to GBP with $K = w$ by defining $H(i)$ to output the i'th column of the matrix (for $i \in [m]$).[1] We note that instances of SD which originate from cryptography typically have many solutions, and hence GBP algorithms are relevant. For more details about the application of GBP to cryptanalysis of code-based cryptosystems, we refer the reader to [8].

Wagner's K-tree algorithm for GBP with $K = 2^k$ (where k is a positive integer) receives as input K lists $\{L_i\}_{i=1}^{K}$, each containing about $2^{n/(k+1)}$ strings of n bits, which are assumed to be uniform in $\{0,1\}^n$. The algorithm returns a K-tuple $\{y_i\}_{i=1}^{K}$, where $y_i \in L_i$ such that $\sum_{i=1}^{K} y_i = 0$. The algorithm can be used to solve GBP assuming that $\ell \geq n/(k+1)$ by initializing the lists $\{L_i\}_{i=1}^{K}$ with elements of the form $y = H(x)$ for arbitrary values of $x \in \{0,1\}^{\ell}$.

At a high level, the K-tree algorithm merges its 2^k inputs lists in a full binary tree structure with k layers. In each layer, the lists are merged in pairs, where each merged pair gives a new list that is input to the next layer and contains words with a larger zero prefix. Finally, the last layer yields a zero word which can be traced back to a K-tuple $\{x_i\}_{i=1}^{K}$ such that $\sum_{i=1}^{K} H(x_i) = 0$ as required. The time and memory complexities of the K-tree algorithm are about $2^{n/(k+1)} = N^{1/(\log K+1)}$ (up to constants and small multiplicative factors in n, K), as detailed in Sect. 3. Since any GBP algorithm for a certain value of K can be extended with the same complexity to any $K' > K$, the time and memory complexities of the K-tree algorithm for general K are $N^{1/(\log K+1)}$, where $\log K$ is rounded down to the nearest integer.

[1] GBP formally requires that $\sum_{i=1}^{K} H(x_i) = 0$, but it is typically easy to tweak GBP algorithms to output $\{x_i\}_{i=1}^{K}$, such that $\sum_{i=1}^{K} H(x_i) = s$ for any fixed $s \in \{0,1\}^n$.



If the point $(m',t') = (m/n,t/n)$ is on the curve, then given $2^{m'n} = 2^m$ memory, the algorithm solves GBP in time $2^{t'n} = 2^t$ (e.g., for $K = 8$, the point $(1/7, 2/7)$ is on the curve of our algorithm, namely, it solves GBP in memory and time complexities of $M = 2^{n/7} = N^{1/7}$ and $T = 2^{2n/7} = N^{2/7}$, respectively).

Fig. 1 GBP time-memory tradeoff curves for $K \in \{8, 16, 32\}$

Due to the high memory consumption of the K-tree algorithm, an important challenge (already pointed out by Wagner) is to investigate time-memory tradeoff algorithms for GBP, which optimize the time complexity T given only $2^m = M < 2^{n/(k+1)}$ memory. The trivial time-memory tradeoff algorithm repeatedly builds K lists of size M from arbitrary inputs to H and executes the K-tree algorithm until a solution is found. Simple analysis shows that this algorithm achieves a tradeoff of $TM^{\log K} = N$, which is very inefficient even when K is moderately large, as the time complexity T increases sharply for memory $M < N^{1/(\log K+1)}$.

Improved tradeoffs were first published in [3,4] (by Bernstein and Bernstein et al., respectively), where the main idea is to execute (what we call) a *preparation phase* before running the K-tree algorithm. This phase iterates over a portion of the domain space and builds lists that are input to the K-tree algorithm (rather than building them arbitrarily), thus increasing its probability to find a solution. A different approach to the preparation phase based on partial collisions in H was published independently by Nikolić and Sasaki [12] and by Biryukov and Khovratovich [5]. This technique gives the currently best known GBP time-memory tradeoff of $T^2 M^{\log K-1} = N$.

In this paper, we devise a sub-linear[2] time-memory tradeoff $T^{\lceil(\log K)/2\rceil+1} M^{\lfloor(\log K)/2\rfloor} = N$ for GBP with $K \geq 8$. This improves upon the currently best known tradeoff $T^2 M^{\log K-1} = N$ for all $K \geq 8$. Our tradeoff is applicable whenever $T^{1/2} \leq M \leq T$. For the range $1 < M \leq T^{1/2}$, we also improve upon the best known tradeoff for $K \geq 8$, but our curve formula becomes more complex as K grows. In general, given an instance of GBP, one selects a GBP algorithm that is applicable for the available amount of memory M.

We plot our GBP tradeoff curves for $K \in \{8, 16, 32\}$ in Fig. 1, while comparing them to the best known ones [5,12]. Note that the tradeoff $T^{\lceil(\log K)/2\rceil+1} M^{\lfloor(\log K)/2\rfloor} = N$ for the range $T^{1/2} \leq M \leq T$ corresponds to the rightmost linear piece of our curve (for $K = 16$ it extends beyond this range), and its improvement compared to the previously best known curve grows with K (visually, the angle between the curves grows with K).

[2] In a sub-linear time-memory tradeoff, the exponent of T is larger than the exponent of M.

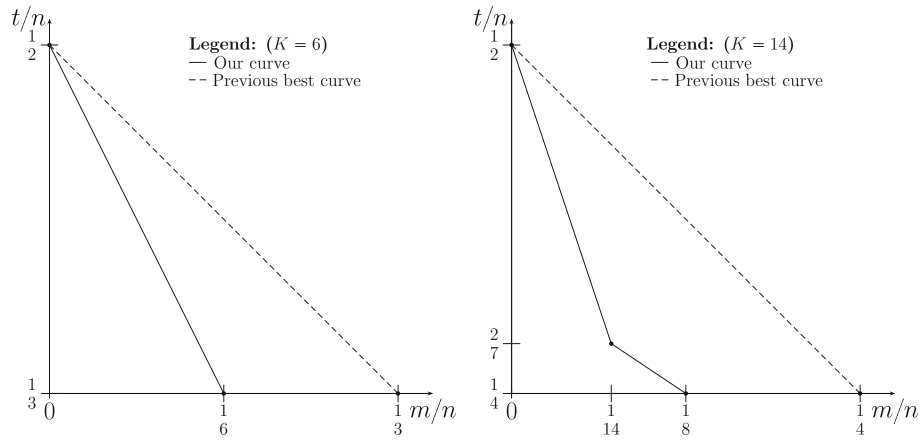


Fig. 2 GBP time-memory tradeoff curves for $K \in \{6, 14\}$

We further improve the best known tradeoffs for values of K that are not powers of 2. Our improved tradeoffs for $K \in \{6, 14\}$ are plotted in Fig. 2.

Finally, we consider practical settings in which the domain size $L = 2^\ell$ of H is limited and the K-tree algorithm cannot be directly applied. The best known algorithm for such cases is an extension of the K-tree algorithm, published by Minder and Sinclair [11]. In this restricted setting, we show how to solve many GBP instances more efficiently than the extended K-tree algorithm, improving both its time and memory complexities.

As noted above, previous time-memory tradeoffs applied a *preparation phase* to initialize several lists and search for a solution among them in (what we call) a *list sum phase*. In these works the focus was placed on the preparation phase, while the list sum phase applied the K-tree algorithm. In contrast, we focus on the list sum phase and develop algorithms that are superior to the straightforward application of the K-tree algorithm when the available memory is limited. We then carefully combine these algorithms with previous preparation phase techniques to obtain improved time-memory tradeoffs for GBP.

We begin by considering a *list sum problem* whose input consists of K sorted lists $\{L_i\}_{i=1}^K$ of n-bit words and the goal is to find a certain number of K-tuples $\{y_i\}_{i=1}^K$, where $y_i \in L_i$ such that $\sum_{i=1}^K y_i = 0$. There are several *exhaustive* list sum memory-efficient algorithms known for this problem that output all solutions. These algorithms are the starting points of the framework we develop in this paper. Our framework transforms such an exhaustive list sum algorithm into an efficient GBP algorithm for a given amount of memory. Obviously, an exhaustive list sum algorithm can be directly applied to solve GBP (after initializing $\{L_i\}_{i=1}^K$ accordingly), as the goal is to find only one out of all solutions. However, this trivial application is inefficient since it does not exploit the fact that we only search for a single solution and moreover, it does not use a preparation phase.

Our framework consists of three main parts. First, we transform a given exhaustive list sum algorithm to efficiently output a limited number of solutions, obtaining a *basic* list sum algorithm, optimized for a specific value of $K = P$. The second part of the framework composes basic algorithms for $K = P$ in a *layered* tree structure which resembles the K-tree algorithm (but with arity P instead of 2). This gives optimized list sum algorithms for values of $K = P^k$ (and additional values) where k is a positive integer. Finally, after optimizing the list sum phase, we combine it with a preparation phase to obtain a memory-efficient GBP algorithm.

Arguably, the most interesting part of our framework is the first part in which we analyze exhaustive list sum algorithms and transform them to efficiently output a limited number of solutions. There are two classes of exhaustive memory-efficient list sum algorithms relevant to this work: the first class consists of variants of the Schroeppel–Shamir algorithm for solving knapsacks, devised by Howgrave-Graham and Joux [9] (which focused on $K = 4$) and by Becker et al. [1] (which applied a recursive variant for $K = 16$). The second class consists of dissection algorithms [7], published by Dinur et al. and used to efficiently solve certain search problems with limited amount of memory.

In general, both classes of exhaustive list sum algorithms partition the problem on K lists into smaller subproblems, solve these subproblems and merge the solutions to solve the original problem. The difference between the classes is in the way the problem is partitioned: while Schroeppel–Shamir variants partition problem symmetrically into subproblems of equal sizes, dissection algorithms partition the problem asymmetrically into smaller and larger subproblems. Thus, Schroeppel–Shamir variants work best on values of K which are powers of 2, while dissection works best on "magic numbers" of K (which are not necessarily powers of 2) such as 7 and 11 that exhibit ideal asymmetric partitions.

Even though the focus of this work is on memory-efficient GBP algorithms, for many GBP instances with restricted domains our techniques yield improvements in both time and memory complexities compared to the extended K-tree algorithm [11] (which is the current state-of-the-art). This occurs for values of K which are not powers of 2 (such as $K = 7$), where our new algorithms extend the dissection framework. The time complexity improvement is due to the fact that symmetric algorithms round the value of K down to the nearest power of 2 and ignore many of the possible solutions. On the other hand, the efficiency of GBP algorithms depends on their ability to find one out of many solutions, and ignoring a large fraction of them in advance is a suboptimal approach.

The rest of the paper is organized as follows. In Sect. 2 we introduce our notations and conventions, and describe preliminaries and previous work in Sect. 3. The first part of our framework that transforms exhaustive list sum algorithms to basic ones is introduced in Sect. 4, while the second part that constructs new layered algorithms is described in Sect. 5. In Sect. 6 we focus on the third part of the framework that combines preparation and list sum phase algorithms to solve GBP. Finally, in Sect. 7 we apply our new algorithms to GBP instances with a restricted domain and conclude the paper in Sect. 8.

2 Notations and conventions

Given an n-bit string x, we label its bits as $x[1], x[2], \ldots, x[n]$ (where $x[1]$ is the least significant bit or LSB). Given integers $1 \leq a \leq b \leq n$, we denote by $x[a\text{–}b]$ the $(b - a + 1)$-bit string $x[a], x[a + 1], \ldots, x[b]$.

Let $F : \{0, 1\}^{\ell} \to \{0, 1\}^{n}$, be a function for $\ell \leq n$. Given parameters $\ell' \leq \ell$ and $n' \leq n$, define the *truncated function* $F_{|\ell',n'} : \{0, 1\}^{\ell'} \to \{0, 1\}^{n'}$ as $F_{|\ell',n'}(x') = F(x)[1\text{–}n']$ (where the ℓ-bit string x is constructed by appending $\ell - \ell'$ zero most significant bits to the ℓ'-bit string x').

The generalized birthday problem (GBP) GBP with parameter K is given oracle access to a function $H : \{0, 1\}^{\ell} \to \{0, 1\}^{n}$ for $\ell \leq n$, and the goal is to find a K-tuple $\{x_i\}_{i=1}^{K}$, where $x_i \in \{0, 1\}^{\ell}$ are distinct (i.e., $x_i \neq x_j$ for $i \neq j$) such that $\sum_{i=1}^{K} H(x_i) = 0$. The addition is performed bitwise over $GF(2)$.

We assume in this paper that H is a pseudo-random function. Our goal is to optimize the time complexity T of solving GBP with parameters K and ℓ, given $M = 2^m$ words of memory, each of length n bits. In some settings (such as in [4]) each x_i in the output K-tuple needs to come from a different domain. This can be modeled using K functions $\{H_i\}_{i=1}^K$. The adaptation of the algorithms we consider to this setting is mostly straightforward.

Typically, GBP algorithms evaluate the function H in a preparation phase in order to set up an instance of the list sum problem.

The list sum problem Given K sorted lists $\{L_i\}_{i=1}^K$, each of $M = 2^m$ words (chosen uniformly at random) of length at least n, the goal is to find S (one, several, or all) K-tuples $\{y_i\}_{i=1}^K$, where $y_i \in L_i$ such that $(\sum_{i=1}^K y_i)[1\text{–}n] = 0$. The number of required solutions S is a parameter to the problem. We note that in our framework list sum algorithms use about $M = 2^m$ memory (which is the input size, up to the constant K).[3]

The list sum problem is related to the well-known K-SUM problem, which searches for a single solution $\sum_{i=1}^K y_i = 0$ in one input set, as opposed to several lists. Moreover, typically the distribution of words in the input set of the K-SUM problem is arbitrary and it is a worst case problem (whereas we are interested in average case complexity).

2.1 Naming conventions and notations for list sum algorithms

An algorithm that finds all solutions to the list sum problem is called an *exhaustive* list sum algorithm. Otherwise, we name algorithms that output a limited number of solutions (with a bound on S) according to their internal structure: in general we have *basic* algorithms and *layered* algorithms that compose basic algorithms in a tree structure (similarly to the K-tree algorithm).

We refer to a list sum algorithm that solves the list sum problem for a specific value of K as a K-way list sum algorithm. However, when referring to specific list sum algorithms we mostly need more refined notation that distinguishes them according to the values of K and S (the number of required solutions) and also the type of basic algorithms composed in layered algorithms (which determine the arity of the tree). These parameters are sufficient to uniquely identify each list sum algorithm considered in this paper.

The (unique) non-layered list sum algorithm with parameters K, S is denoted by $A_{K,S}$, where S is the number of solutions it produces. In case the algorithm is exhaustive (outputs all solutions), we simply write A_K. For example, A_4 is an exhaustive 4-way list sum algorithm, while $A_{4,1}$ produces only a single solution (and hence can be naturally used to solve GBP with $K = 4$).[4] In general, we will be interested in exhaustive A_K algorithms, basic $A_{K,1}$ algorithms that output a single solution and basic $A_{K,2^m}$ algorithms that produce $S = 2^m$ solutions, allowing to compose basic algorithms and form layered ones.

The (unique) layered list sum algorithm with $S = 1$ and arity P is denoted by A_K^P (we do not consider layered algorithms with $S > 1$ in this paper). For example, the K-tree algorithm is denoted by A_K^2, as it merges its input lists in pairs. We note that layered algorithms can be uniquely distinguished by their arity P, while K is left in symbolic form (unlike basic

[3] This definition does not capture algorithms (such as Minder and Sinclair's algorithm [11]) that merge the initial lists into larger ones. However, the restriction typically does not result in loss of generality, as an initial merge can be considered as a preparation phase algorithm by defining the function H appropriately.

[4] When the number of expected solutions to the list sum problem is S, then $A_{K,S}$ typically coincides with A_K. The more interesting case is when the number of expected solutions is greater than S and $A_{K,S}$ could potentially be more efficient than A_K.

algorithms). We also remark that for any specific value of K, the algorithm A_K^K has a single layer and is actually the basic algorithm $A_{K,1}$ (which is our preferred notation).

When composing basic list sum algorithms in layers, the LSBs of the words in the input lists to the algorithms may already be zeroed by a previously applied basic algorithm and the task of the succeeding basic algorithm is to output solutions where the next sequence of bits is nullified. In such cases, we can ignore the zero LSBs in the input lists, resulting in a problem that complies with the above definition of list merge problem (which requires nullifying LSBs).

2.2 Complexity evaluation

The time and memory complexities of our algorithms are functions of the parameters n and K (and also $L = 2^\ell$ for some GBP instances). In the complexity analysis, we ignore multiplicative polynomial (and constant) factors in n and K, which is common practice in exponential-time algorithms. Nevertheless, we note that these factors are relatively small for the algorithms considered.

3 Preliminaries and previous work

The literature relevant to this work is vast and consists of [1,3–5,7,9,12,14,15]. In this section we summarize it constructively so we can build upon it in the rest of this paper. We first describe general properties of list sum algorithms in Sect. 3.1. Then, we describe previous exhaustive list sum algorithms A_K: in Sect. 3.2 we focus on exhaustive symmetric Schroeppel–Shamir variants for several values of K which are powers of 2 ($K \in \{4, 16\}$ and the basic $K = 2$), while in Sect. 3.3 we deal with asymmetric exhaustive dissection algorithms. In Sect. 3.4, we show how exhaustive algorithms for $K = 2$ and $K = 4$ were efficiently adapted to basic algorithms that output a limited number of solutions. In Sect. 3.5, we describe the (layered) K-tree algorithm. Next, in Sect. 3.6 we focus on the preparation phase algorithms parallel collision search (PCS) and clamping through precomputation (CTP). We end this section by summarizing the currently best known time-memory tradeoff for GBP in Sect. 3.7.

3.1 General properties of list sum algorithms

From a combinatorial viewpoint, the number of K-tuples $\{y_i\}_{i=1}^K$ in the K lists input to the list sum problem is 2^{Km}. Since the problem imposes an n-bit restriction on them, the number of expected solutions is 2^{Km-n}. Hence the list sum problem is interesting only if $m \geq n/K$. If we impose an additional b-bit constraint on the tuples (e.g., by requiring that $(y_1 + y_2)[1\text{–}b] = c$ for an arbitrary b-bit value c), then the number of expected solutions drops[5] to $S = 2^s = 2^{Km-b-n}$.

Nullifying bits In this paper, we mostly use an equivalent statement, where we view n as a parameter: if we search for $S = 2^s$ solutions to the problem and set an additional b-bit constraint on the solutions, then the number of bits we can *nullify* is

$$n = Km - b - s. \tag{1}$$

[5] Our algorithms will also assume that the number of solutions has low variance, hence such constraints have to be set carefully for this to hold.

By default, a list sum algorithm is given the n-bit *target value* 0, but it can easily be adapted and applied with similar complexity to an arbitrary n-bit target value w, such that it outputs K-tuples $\{y_i\}_{i=1}^{K}$ where $\sum_{i=1}^{K} y_i[1\text{–}n] = w$. This can be done by XORing w to the all the words in L_1, resorting it and solving the problem with a target of 0.

A basic list sum algorithm that searches for a single solution $S = 1$ for a certain K value using M memory (such that $m \geq n/K$) can be applied with the same complexity to any $K' > K$. This is done by choosing an arbitrary $(K' - K)$-tuple $\{y_i\}_{i=K+1}^{K'}$ from lists $\{L_i\}_{i=K+1}^{K'}$, and applying the given algorithm with input lists $\{L_i\}_{i=1}^{K}$ and target value $\sum_{i=K+1}^{K'} y_i[1\text{–}n]$. Hence the list sum problem for $S = 1$ does not become harder as K grows, in contrast to exhaustive list sum problems that require all solutions.

We now describe exhaustive list sum algorithms of type A_K for several values of K. We denote by $T = 2^{\tau_K m}$ (for a parameter τ_K) the time complexity of A_K.

3.2 Exhaustive symmetric list sum algorithms A_K for $K = 2^k$

3.2.1 A_2

The standard list sum algorithm A_2 looks for all matches on n bits between two sorted lists of size 2^m. There are 2^{2m-n} possible solutions to the problem and A_2 finds them in time $T = 2^m$ ($\tau_2 = 1$) assuming that their number is at most 2^m, namely, $2^{2m-n} \leq 2^m$ or $m \leq n$.

3.2.2 A_4 [9]

This algorithm was devised in [9] by Howgrave-Graham and Joux as a practical variant of the Schroeppel–Shamir's algorithm [13] (see Fig. 3).

1. For all 2^m possible values of the m-bit word c:
   (a) Apply A_2 to the sorted lists L_1, L_2 with the m-bit word c as the target value. Namely, look for pairs $(y_1, y_2) \in L_1 \times L_2$ such that $(y_1 + y_2)[1\text{–}m] = c$. Store the expected number of $2^{2m-m} = 2^m$ output sums $y_1 + y_2$ in a new sorted list L'_1, along with the corresponding $(y_1, y_2) \in L_1 \times L_2$.
   (b) Apply A_2 to the sorted lists L_3, L_4 with c as the target value and build the sorted list L'_2.
   (c) Apply A_2 to the sorted lists L'_1, L'_2 with target value 0.[a] Trace the output pairs (y'_1, y'_2) back to solutions to the list sum problem: $(y_1, y_2, y_3, y_4) \in L_1 \times L_2 \times L_3 \times L_4$ where $(y_1 + y_2 + y_3 + y_4)[1\text{–}n] = 0$.

[a] Note that for any $y'_1 \in L'_1$ and $y'_2 \in L'_2$ we have $(y'_1 + y'_2)[1\text{–}m] = 0$, hence it remains to nullify bits $[m + 1\text{–}n]$.

The A_4 algorithm enumerates all possible solutions, as any specific solution $y_1 + y_2 + y_3 + y_4 = 0$ is output when the value of c is set to $(y_1 + y_2)[1\text{–}m]$. The expected number of solutions is 2^{4m-n}, and assuming that their number is not larger than 2^{2m} (i.e., $2^{4m-n} \leq 2^{2m}$ or $m \leq n/2$), then its time complexity is $T = 2^{2m}$, namely $\tau_4 = 2$ (since the time complexity of the loop for each value of c is 2^m).

The algorithm may produce up to 2^{2m} solutions, but its memory complexity is only 2^m. This is possible since we not required to store the solutions, but may *stream* them to another

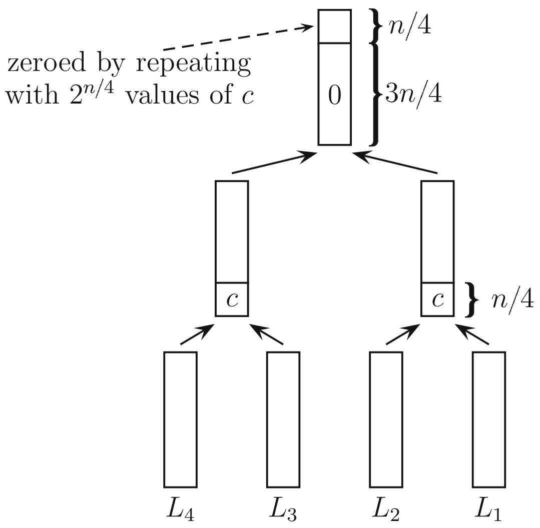


Fig. 3 A_4 with $M = 2^{n/4}$, $T = 2^{n/2}$

algorithm that will process them on-the-fly. This is an important property that holds for all list sum algorithms described in this paper.

3.2.3 A_{16} [1]

An extension of the A_4 algorithm will allow us to construct algorithms that find a single solution ($S = 1$) for a limited range of the memory parameter M. An extension of the A_{16} algorithm below will yield algorithms that find a single solution for a broader range of memory complexities. The exhaustive A_{16} algorithm is a recursive variant of the previous A_4 algorithm, published by Becker et al. [1] (see Fig. 4).

1. For all 2^{9m} possible values of the four $3m$-bit words c_1, c_2, c_3, c_4 that satisfy $c_1 + c_2 + c_3 + c_4 = 0$:
   (a) Apply the 4-way list sum algorithm A_4 four times to lists $\{L_i\}_{i=1}^{4}$, $\{L_i\}_{i=5}^{8}$, $\{L_i\}_{i=9}^{12}$ and $\{L_i\}_{i=13}^{16}$, with c_1, c_2, c_3, c_4 as the $3m$-bit target values, respectively. Store the outcomes of these algorithms in four sorted lists L'_1, L'_2, L'_3, L'_4, each of expected size $2^{4m-3m} = 2^m$.
   (b) Apply the 4-way list sum algorithm A_4 to L'_1, L'_2, L'_3, L'_4 (nullifying bits $[3m + 1\text{–}n]$) and from each output 4-tuple, derive a corresponding 16-tuple as a solution to the problem.

We iterate over all possible solutions in time $T = 2^{9m+2m} = 2^{11m}$ ($\tau_{16} = 11$),[6] assuming the number of solutions satisfies $2^{16m-n} \leq 2^{11m}$ or $m \leq n/5$.

[6] We note that [1] extended this algorithm to a time-memory tradeoff. However, as it uses memory larger than 2^m (the size of the input lists), we do not consider it in this paper.

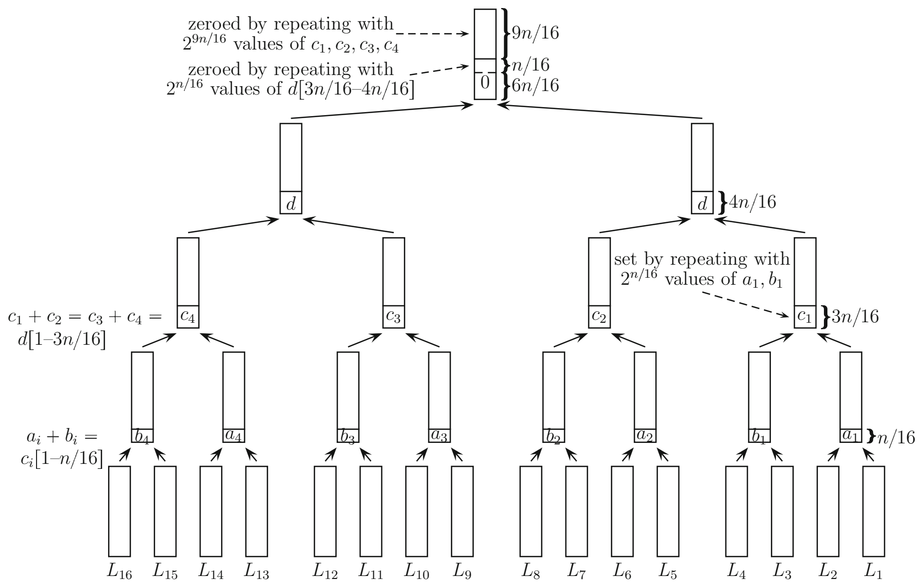

zeroed by repeating with $2^{9n/16}$ values of c_1, c_2, c_3, c_4
$9n/16$
zeroed by repeating with $2^{n/16}$ values of $d[3n/16–4n/16]$
$n/16$
0
$6n/16$
d
d
$4n/16$
set by repeating with $2^{n/16}$ values of a_1, b_1
$c_1 + c_2 = c_3 + c_4 = d[1–3n/16]$
c_4
c_3
c_2
c_1
$3n/16$
$a_i + b_i = c_i[1–n/16]$
b_4
a_4
b_3
a_3
b_2
a_2
b_1
a_1
$n/16$
L_{16} L_{15} L_{14} L_{13} L_{12} L_{11} L_{10} L_9 L_8 L_7 L_6 L_5 L_4 L_3 L_2 L_1

Fig. 4 A_{16} with $M = 2^{n/16}$, $T = 2^{11n/16}$

3.3 Exhaustive asymmetric list sum algorithms A_K [7]

Dissection is an algorithmic framework for solving certain combinatorial search problems with optimized time-memory tradeoffs. It was introduced by Dinur et al. at CRYPTO 2012 [7] and used in order to break multiple encryption (i.e., an iterated construction of a block cipher with independent keys) and to solve the knapsack and Rubic's cube problems with improved combinations of time and space complexities. In our context, we view dissection algorithms as memory-efficient asymmetric list sum algorithms of class A_K.

Given a $A_{K'}$ algorithm, it can be trivially utilized as a A_K algorithm for $K > K'$ (with no additional memory) by enumerating all the $2^{m(K-K')}$ possible tuples in the first $K - K'$ lists, and applying $A_{K'}$ on the remaining K' lists (with the target sum set to be the sum of the current $(K - K')$-tuple). However, for certain values of K we can do better than this trivial algorithm, and dissection algorithms define a sequence of values of K for which this efficiency gain occurs.

The first dissection algorithm is defined for $K = 2$ (namely A_2), and it looks for matches in its 2 sorted input lists. The next number in the sequence is $K = 4$ and this dissection algorithm essentially coincides with A_4 described above. Next is the 7-way dissection algorithm, which utilizes the 3-way list sum algorithm A_3 described below.

3.3.1 A_3

For each pair $(y_1, y_2) \in L_1 \times L_2$, compute $y_1 + y_2$, and search for a match $y_3 \in L_3$ such that $y_3[1–n] = (y_1 + y_2)[1–n]$. For each match found, output the triplet (y_1, y_2, y_3).

The algorithm enumerates over all 2^{3m-n} solutions in time $T = 2^{2m}$ ($\tau_3 = 2$) assuming $m \leq n$.

3.3.2 A_7

For $K \geq 7$, the asymmetry in dissection algorithms becomes more apparent, as they partition the problem of size K into two *subproblems* of different sizes. We begin by describing the $K = 7$ algorithm.

```
1. For each possible value of the $2m$-bit word $c$:
    (a) Apply the $A_3$ algorithm to $L_1, L_2, L_3$ with the $2m$-bit target value $c$, and store
        the expected number of $2^{3m-2m} = 2^m$ outputs (whose $2m$ LSBs equal to $c$) in
        a new sorted list $L'$. Each word is stored along with the corresponding triplet
        of indexes in $L_1 \times L_2 \times L_3$.
    (b) Apply the $A_4$ algorithm of Sect. 3.2 to $L_4, L_5, L_6, L_7$ with the $2m$-bit target $c$.
        For each obtained solution quartet, $(y_4, y_5, y_6, y_7) \in L_4 \times L_5 \times L_6 \times L_7$ (such
        that $(y_4 + y_5 + y_6 + y_7)[1–2m] = c$), search $L'$ for matches on $(y_4 + y_5 + y_6 +
        y_7)[2m + 1–n]$ and output the corresponding 7-tuples.
```

The algorithm enumerates all possible solutions to the problem, since each solution can be decomposed as above. The time complexity of each 3-way and 4-way list sum steps in the loop is 2^{2m}, while we iterate over 2^{2m} possible values of c. Hence, the expected time complexity is $T = 2^{4m}$ ($\tau_7 = 4$) as long as the number of solutions is at most 2^{4m}. Since the expected number of solution is 2^{7m-n}, we require $7m - n \leq 4m$ or $m \leq n/3$.

We also note that the algorithm splits the problem on 7 lists into 2 subproblems of respective sizes 3, 4, while the size 4 problem itself is internally split into two subproblems of sizes 2, 2 by the A_4 algorithm. Altogether, the problem of size 7 is split into 3 subproblems of respective sizes 3, 2, 2.

3.3.3 General dissection

The now give the details and analysis of general dissection algorithms. For a value of $i = 0, 1, 2, \ldots$, we construct a K_i-way list sum algorithm that runs in time τ_{K_i} such that $K_i = 1 + i(i+1)/2$ and $\tau_{K_i} = 1 + i(i-1)/2$. Note that, as always, we require that $m \geq n/K$ so at least one solution is expected. Moreover, the expected number of solutions cannot be larger than the runtime, i.e., $2^{K_i m - n} \leq 2^{\tau_{K_i} m}$ or $(K_i - \tau_{K_i})m \leq n$, giving the requirement $m \leq n/i$ (for a positive i).

In particular, after A_7 which corresponds to $K_3 = 7, \tau_7 = 4$, we have $K_4 = 11$ and $\tau_{11} = 7$, i.e. the A_{11} dissection algorithm has time complexity 2^{7m} (assuming $m \leq n/4$). Internally, A_{K_i} recursively splits the problem of size $K_i = 1 + i(i+1)/2$ into i subproblems of sizes $i, i-1, i-2, \ldots, 3, 2, 2$ and applies the algorithms $A_i, A_{i-1}, \ldots, A_3, A_2, A_2$, respectively, for various choices of *intermediate target values* (such as the $2m$-bit word c in A_7).

Assume that we have a $A_{K'}$ algorithm that given $M = 2^m$ memory runs in time complexity $T' = 2^{\tau_{K'} m} < 2^n$ (for an integer $\tau_{K'} \geq 1$). Moreover, the algorithm can be applied in streaming mode (as all list sum algorithms described in this paper). Namely, given a target of $n' = m(K' - \tau_{K'})$ bits with arbitrary value, it finds all the $2^{n-m(K'-\tau_{K'})}$ expected solutions in time $T' = 2^{\tau_{K'} m}$ (assuming $m \geq n/K'$). Then, we can construct a A_K algorithm for $K = 2K' - \tau_{K'} + 1$ with time complexity $T = 2^{K'm}$ ($\tau_K = K'$) and memory complexity $M = 2^m$ as follows.

```
1. For each $m(K' - \tau_{K'})$-bit value $c$:
    (a) Apply a $A_{K'-\tau_{K'}+1}$ algorithm to $\{L_i\}_{i=1}^{K'-\tau_{K'}+1}$, with $c$ as the target, and store the
        expected number of $2^{m(K'-\tau_{K'}+1-(K'-\tau_{K'}))} = 2^m$ outputs (whose $m(K'-\tau_{K'})$
        LSBs equal to $c$) in a new sorted list $L'$. Each word is stored along with the
        corresponding $(K'-\tau_{K'}+1)$-tuple of indexes in $\{L_i\}_{i=1}^{K'-\tau_{K'}+1}$.
    (b) Apply the given $A_{K'}$ algorithm to $\{L_i\}_{i=K'-\tau_{K'}+2}^{2K'-\tau_{K'}+1}$ (the remaining $K'$ lists) with
        $c$ as the target. For each obtained $K'$-tuple, search for a match in $L'$ on the
        remaining $n - m(K'-\tau_{K'})$ bits and output the corresponding $K$-tuples.
```

The memory complexity of the algorithm is indeed $M = 2^m$. In terms of time complexity, Since $\tau_{K'} \geq 1$, then the $A_{K'}$ algorithm dominates the $A_{K'-\tau_{K'}+1}$ algorithm in the inner loop (it is applied to a problem of size at least as large). Overall, the time complexity is $T = 2^{m(K'-\tau_{K'})+\tau_{K'}m} = 2^{K'm}$ as claimed.

The algorithm can be applied in streaming mode and therefore it can be applied recursively, giving a sequence of dissection algorithms. The dissection sequence starts with $K_0 = 1$ and $\tau_{K_0} = \tau_1 = 1$ (which is a trivial 1-way list sum algorithm). Next, we obtain $K_1 = 2K_0 - \tau_{K_0} + 1 = 2$ and $\tau_{K_1} = \tau_2 = K_0 = 1$, $K_2 = 2K_1 - \tau_{K_1} + 1 = 4$ and $\tau_{K_2} = \tau_4 = K_1 = 2$. In general, it is easy to verify that $K_i = 1 + i(i+1)/2$ and $\tau_{K_i} = 1 + i(i-1)/2$. In particular, after $K_3 = 7$, we have $K_4 = 11$ and $\tau_{11} = 7$, i.e. the A_{11} dissection algorithm has time complexity 2^{7m}. As mentioned above, observe that A_{K_i} recursively splits the problem of size $K_i = 1 + i(i+1)/2$ into i subproblems of respective sizes $i, i-1, i-2, \ldots, 3, 2, 2$.

3.4 Basic list sum algorithms

3.4.1 $A_{2,1}$ and $A_{2,2^m}$

The $A_{2,1}$ algorithm is an extension of the A_2 algorithm of Sect. 3.2. It searches for a single solution ($S = 2^s = 1$) to the problem by looking for an n-bit match in the two lists. According to (1), $A_{2,1}$ can nullify $n = 2m$ bits in time $T = 2^m$.

For $A_{2,2^m}$, instead of matching and nullifying $2m$ bits using $A_{2,1}$ with $S = 1$, we require $S = 2^m$ solutions, or $s = m$. Plugging $K = 2, b = 0, s = m$ into (1), we conclude that we match and nullify $n = m$ bits (in time complexity $T = 2^m$).

3.4.2 $A_{3,1}$

This algorithm extends the A_3 algorithm of Sect. 3.3. It searches for a single solution to the problem and hence can nullify $n = 3m$ bits in time 2^{2m}.

3.4.3 $A_{4,1}$ and $A_{4,2^m}$ [9]

The $A_{4,1}$ algorithm is an extension of the A_4 algorithm of Sect. 3.2, also described in [9]. When we search for a limited number of solutions to the list sum problem, we can enumerate over fewer values of the m-bit *intermediate target value* c in the A_4 algorithm. This is equivalent to placing another constraint on the 4-tuple solutions: for $0 \leq v \leq m$, we place a $(m - v)$-bit constraint by only enumerating over 2^v values of c. Setting $v = 0$ places an m-bit constraint and reduces the time complexity of the algorithm to 2^m, while nullifying $n = 4m - m = 3m$ bits (according to (1)). In general, we can nullify $n = 4m - (m - v) = 3m + v$ bits in time $T = 2^{m+v}$, giving the tradeoff

$$TM^2 = 2^{m+v} \cdot 2^{2m} = 2^{3m+v} = 2^n = N.$$

The tradeoff is only applicable for $0 \leq v \leq m$ or $T^{1/2} \leq M \leq T$.

For $A_{4,2^m}$ we apply a similar algorithm, but (according to (1)) since s is increased by m then n is reduced by m. Denote by n' the parameter of $A_{4,1}$, then $n = n' - m$. Using the tradeoff formula above, we obtain $TM^2 = 2^{n'} = 2^{n+m} = NM$ or $TM = N$, applicable once again for $T^{1/2} \leq M \leq T$.

3.5 The K-tree algorithm: A_K^2 [15]

We begin by describing the K-tree algorithm for $K = 4$ (namely, A_4^2) below. This algorithm is also shown in Fig. 5.

1. Apply $A_{2,2^m}$ to the sorted lists L_1, L_2. Namely, look for pairs $(y_1, y_2) \in L_1 \times L_2$ such that $(y_1 + y_2)[1\text{–}m] = 0$. Store the expected number of $2^{2m-m} = 2^m$ output sums $y_1 + y_2$ in a new sorted list L'_1, along with the corresponding $(y_1, y_2) \in L_1 \times L_2$.
2. Apply $A_{2,2^m}$ to the sorted lists L_3, L_4 and build the sorted list L'_2.
3. Apply $A_{2,1}$ to the sorted lists L'_1, L'_2. Trace an output pair (y'_1, y'_2) back to a solution to the list sum problem: $(y_1, y_2, y_3, y_4) \in L_1 \times L_2 \times L_3 \times L_4$ where $(y_1 + y_2 + y_3 + y_4)[1\text{–}n] = 0$.

The algorithm is composed to 2 layers. The input to the first layer consists of 4 input lists and it merges the lists in pairs by applying $A_{2,2^m}$ and outputting the lists L'_1, L'_2, each containing 2^m words whose m LSBs are zero (since $A_{2,2^m}$ nullifies m bits). These lists are input to layer 2 which applies $A_{2,1}$ to obtain the final solution, nullifying additional $2m$ bits. Altogether, $n = 3m$ bits are nullified in $T = 2^m = 2^{n/3} = N^{1/3}$ time and $M = 2^m = N^{1/3}$ memory.

Note that $A_{4,1}$ above for $M = T$ is, in fact, the A_4^2 algorithm. Indeed, when $M = T$, $A_{4,1}$ is composed of 2 layers of 2-way list sum algorithms.

The general algorithm. The K-tree algorithm for general $K = 2^k$ is a A_K^2 algorithm that works in k layers as summarized next. For more details, refer to [15].

For any integer $0 \leq j \leq k - 1$, the input to layer j consists of $K/2^j$ sorted input lists, where for each word in each list the $j \cdot m$ LSBs are zero. At layer $0 \leq j < k-1$ the algorithm merges the lists in pairs by applying $A_{2,2^m}$ and outputting $K/2^{j+1}$ new lists, each containing 2^m words whose $(j + 1) \cdot m$ LSBs are zero (as $A_{2,2^m}$ nullifies m bits). These lists are input to the next layer $j + 1$.

Finally, the input to layer $j = k - 1$ consists of $K/2^{k-1} = 2$ lists of expected size 2^m containing words whose $(k - 1) \cdot m$ LSBs are zero. The K-tree algorithm then applies $A_{2,1}$ to obtain the final solution, nullifying additional $2m$ bits. Altogether, $n = (k + 1) \cdot m$ bits are nullified in $T = 2^m = 2^{n/(k+1)} = N^{1/(\log K+1)}$ time and $M = 2^m = N^{1/(\log K+1)}$ memory.

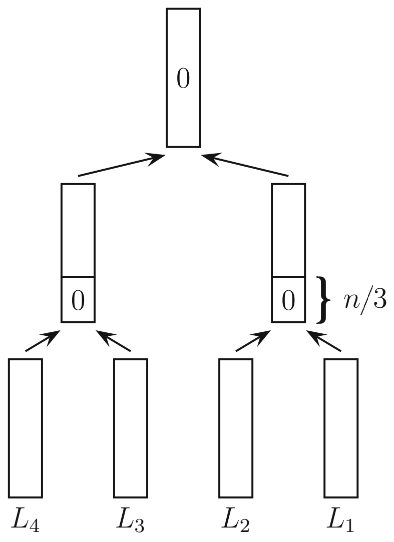


Fig. 5 A_4^2 with $M = 2^{n/3}, T = 2^{n/3}$

3.6 Preparation phase algorithms

3.6.1 Parallel collision search [14]

The parallel collision search (PCS) algorithm was published by van Oorschot and Wiener [14] as a memory-efficient technique for finding collisions in an r-bit function $F : \{0, 1\}^r \rightarrow \{0, 1\}^r$. Given $2^m \leq 2^r$ words of memory, the algorithm builds a chain structure containing 2^m chains, where a chain starts at an arbitrary point and computed by iteratively applying F. Each chain is terminated after about $2^{(r-m)/2}$ evaluations, hence the structure contains a total of about $2^m \cdot 2^{(r-m)/2} = 2^{(r+m)/2}$ distinct points. The fact that the chains are of length $2^{(r-m)/2}$ ensures that each chain collides with a different chain in the structure with high probability according to the birthday paradox, as the number of relevant pairs of points is $2^{(r-m)/2} \cdot 2^{(r+m)/2} = 2^r$. Therefore, the structure contains about 2^m collisions.

The collisions can be recovered efficiently by defining a set of $2^{(r+m)/2}$ distinguished points according to an easily verifiable condition (e.g., the $(r - m)/2$ LSBs of the r-bit word are 0). Each chain in the structure is terminated at a distinguished point (and hence its expected length is $2^r/2^{(r+m)/2} = 2^{(r-m)/2}$ as required). The PCS algorithm stores the distinguished points sorted in memory and collisions between chains are detected at their distinguished points. The actual collisions are obtained by recomputing the colliding chains.

In total, PCS finds 2^m collisions in an r-bit function in time complexity

$$T = 2^{(r+m)/2}.$$

3.6.2 Parallel collision search in expanding functions [14]

Assume that our goal is to find 2^m collisions using 2^m memory in the expanding function $F : \{0, 1\}^\ell \rightarrow \{0, 1\}^r$, where $(r + m)/2 \leq \ell \leq r$ (ensuring that 2^m collisions indeed exist in F). We apply PCS to the truncated function $F_{|\ell,\ell}$ to find 2^m collisions on ℓ bits of F in $2^{(\ell+m)/2}$ time. Each such collision extends to a full r-bit collision with probability $2^{\ell-r}$. In other words, the PCS execution gives an expected number of $2^{m+\ell-r}$ full collisions in F. To find the required 2^m collisions, we repeat the process $2^{r-\ell}$ times, giving time complexity of

$$T = 2^{r-\ell} \cdot 2^{(\ell+m)/2} = 2^{r+(m-\ell)/2}.$$

We note that in order to make the different PCS executions essentially independent, we have to use a different *flavor* of $F_{|\ell,\ell}$ in each execution (as done in [14]). For example, we can define the i'th flavor as $F^i_{|\ell,\ell}(x) = F_{|\ell,\ell}(x) + i$.

3.6.3 Clamping through precomputation [3,4]

The goal here is to find 2^m values x_i such that $F(x_i)[1\text{–}r] = 0$ for a parameter r, given a function $F : \{0,1\}^\ell \rightarrow \{0,1\}^n$ (for $m + r \leq \ell \leq n$). This can be done by using clamping through precomputation (CTP) [3,4], as detailed below.

```
1. For $2^{m+r}$ arbitrary strings of $\ell$ bits $x_0, x_1, \ldots, x_{2^{m+r}}$:
    (a) Compute $F(x_i)$. If $F(x_i)[1\text{–}r] = 0$, then add $x_i$ to a list $L$.
```

After exhausting $T = 2^{m+r}$ values of x_i, we expect L to contain $2^{m+r-r} = 2^m$ strings that satisfy the r-bit condition $F(x_i)[1\text{–}r] = 0$, as required.

3.7 Previous GBP tradeoff algorithms for $K = 2^k$ [5,12]

The best known time-memory tradeoff algorithm for GBP was published independently by Nikolić and Sasaki [5,12] and also by Biryukov and Khovratovich [5]. We describe this algorithm for GBP with $K = 2^k$ and $M = 2^m$ words of memory, such that $m \leq n/(k+1)$ (see Fig. 6).

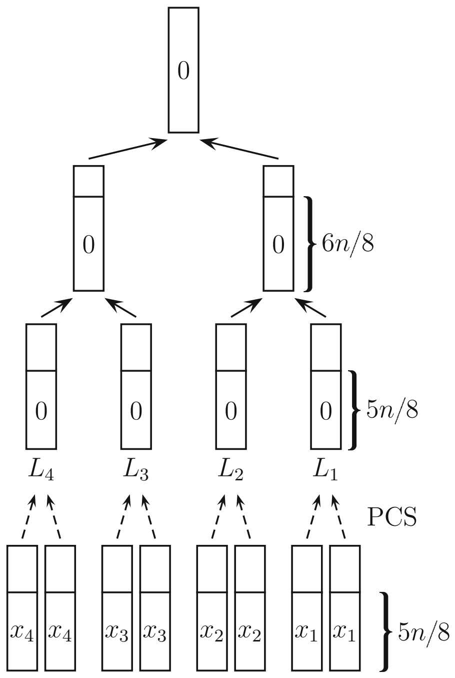
Given an integer parameter $r \geq m$ to be determined later, the algorithm uses the truncated function $H_{|r,r}$ (as defined in Sect. 2).

```
1. Run PCS on the function $H_{|r,r}$ and look for $2^m$ collisions. For each collision
   $H(x)[1\text{–}r] = H(x')[1\text{–}r]$, compute $y = H(x) + H(x')$ and store all these words
   in $K/2$ lists $\{L_i\}_{i=1}^{K/2}$ of size about $2^m$ (along with the corresponding $x$ values).
2. Apply the K-tree algorithm to the $K/2$ lists (nullifying bits $[r+1\text{–}n]$). Obtain $K/2$
   words $y_i \in L_i$ such that $\sum_{i=1}^{K/2} y_i = 0$ and use them (based on the first step) to
   construct a solution to GBP by recovering the $K$ words $x_i$ such that $\sum_{i=1}^{K} H(x_i) =$
   $\sum_{i=1}^{K/2} y_i = 0$ as required.
```

We now calculate the value of r that ensures that the algorithm succeeds to find a solution (with high probability). The K-tree algorithm is run using $2^{k'} = 2^{k-1}$ lists, each of size 2^m, and it has to zero the remaining $n' = n - r$ bits after the PCS is executed. Therefore, according to the analysis of the K-tree algorithm in Sect. 3.5 we require $m = n'/(k'+1) = (n-r)/k$. This gives $r = n - mk$.

The time complexity of the PCS algorithm is $2^{(r+m)/2} = 2^{(n-m(k-1))/2}$, while the time complexity of the K-tree algorithm is $2^m \leq 2^{(r+m)/2}$ (and can be neglected).

Fig. 6 Previous GBP tradeoff algorithm for $K = 8$ with $M = 2^{n/8}$, $T = 2^{3n/8}$



PCS colliding values (x_i's) are arbitrary.

Therefore, the total time complexity is $T = 2^{(n-m(\log K-1))/2}$. This gives a time-memory tradeoff of

$$T^2 M^{\log K-1} = N,$$

assuming that $M \leq N^{1/(\log K+1)}$. Note that when $M = N^{1/(\log K+1)}$, the PCS is similar to the first layer of the K-tree algorithm, as the function $H_{|r,r}$ is iterated only once. In addition, observe that the two steps of the algorithm are not balanced since the time complexity of PCS is larger than the time complexity of the K-tree algorithm.

4 Construction of new basic list sum algorithms

The first part of our framework transforms exhaustive list sum algorithms (of type A_K) into basic ones of types $A_{K,1}$ and $A_{K,2^m}$ (that are useful for devising layered algorithms). In this section we transform both exhaustive symmetric and asymmetric algorithms described in Sects. 3.2 and 3.3, respectively. The most relevant basic list sum algorithms obtained in this section and in [9] are summarized in the first four entries of Table 1.

4.1 Preliminary construction and analysis of basic list sum algorithms

Recall that we denote the time complexity of A_K by $2^{\tau_K m}$ for a parameter τ_K. As we show next, the time-memory tradeoff for $A_{K,1}$ is of the form $TM^{\alpha_K} = N$ for $\alpha_K = K - \tau_K$.

Table 1 Basic list sum algorithms

Algorithm	Time-memory tradeoff	Range (M vs. T)	Range (M vs. N)	Reference
$A_{4,1}$	$TM^2 = N$	$T^{1/2} \leq M \leq T$	$N^{1/4} \leq M \leq N^{1/3}$	[9]
$A_{7,1}$	$TM^3 = N$	$T^{1/4} \leq M \leq T^{1/2}$	$N^{1/7} \leq M \leq N^{1/5}$	New
$A_{11,1}$	$TM^4 = N$	$T^{1/7} \leq M \leq T^{1/2}$	$N^{1/11} \leq M \leq N^{1/6}$	New
$A_{16,1}$	$TM^5 = N$	$T^{1/11} \leq M \leq T^{1/2}$	$N^{1/16} \leq M \leq N^{1/7}$	New
$A_{K_i,1}$	$TM^i = N$	$T^{1/\tau_{K_i}} \leq M \leq T^{1/\tau_i}$	$N^{1/(\tau_{K_i}+i)} \leq M \leq N^{1/(\tau_i+i)}$	New
$K_i = 1 + i(i+1)/2)$		$(\tau_{K_i} = 1 + i(i-1)/2)$		

The basic idea generalizes the one used to construct $A_{4,1}$ in Sect. 3.4. We deal with an algorithm A_K that partitions the problem of size K into several smaller subproblems, solves each one for various choices of intermediate target values and for each such choice, merges the outputs, aiming to obtain a final solution. When we iterate over a subset that contains a 2^{-b} fraction of the possible intermediate target values we essentially set an additional b-bit constraint on the returned solutions. Ideally, this allows to reduce the time complexity of the algorithm by a factor of 2^b to $2^{\tau_K m - b}$ at the expense of nullifying less bits: recall from (1) that by setting a b-bit constraint on the solutions, we can nullify $n = Km - b - s = Km - b$ bits (as $s = 0$ for $A_{K,1}$). Therefore, we obtain a tradeoff of

$$TM^{K-\tau_K} = N. \tag{2}$$

Indeed, after setting the b-bit constraint, we hope to reduce the time complexity to $T = 2^{\tau_K m - b}$ and obtain $TM^{K-\tau_K} = 2^{\tau_K m - b + m(K-\tau_K)} = 2^{Km-b} = 2^n = N$. We stress that this is an ideal formula which cannot always be achieved using a concrete algorithm. We carefully design such algorithms below, aiming to apply the ideal formula to the widest range of parameters possible. This will be relatively simply for symmetric list sum algorithms, but requires deeper insight for asymmetric algorithms.

As deduced above in (2), ideally the tradeoff curve of a basic list sum algorithm of type $A_{K,1}$ is of the form

$$TM^{\alpha_K} = N \tag{3}$$

for a constant $\alpha_K = K - \tau_K$. When considering the variant $A_{K,2^m}$, we require 2^m solutions and the number of bits that can be nullified is reduced from n to $n - m$. Consequently, the tradeoff becomes $TM^{\alpha_K} = N'$ for $N' = 2^{n-m}$, giving

$$TM^{\alpha_K - 1} = N. \tag{4}$$

As a result, we do not need to analyze $A_{K,2^m}$ separately.

4.2 Basic symmetric list sum algorithms

The algorithms $A_{4,1}$ and $A_{4,2^m}$ for $K = 4$ (that are applicable in the range $T^{1/2} \leq M \leq T$) were already constructed in Sect. 3.4, where we showed that indeed $\alpha_4 = 4 - \tau_4 = 2$. We continue with $K = 16$.

4.2.1 $A_{16,1}$ and $A_{16,2^m}$

Following the general approach above, we extend the 16-way list sum algorithm A_{16} of Sect. 3.2. Since the time complexity of A_{16} is 2^{11m}, we have $\tau_{16} = 11$ and $\alpha_{16} = 16 - \tau_{16} = 5$ as given by (2) and (3). Below, we perform this computation in more detail and calculate the range for which this tradeoff is applicable.

If we fix all the words c_1, c_2, c_3, c_4 in A_{16} (such that their sum is 0), we cast a constraint of $9m$ bits on the 16-tuples and can nullify $n = 16m - 9m = 7m$ bits in time complexity 2^{2m} (which is the time complexity of the A_4 algorithms).

More generally, when we vary 2^v times the value of c_1, c_2, c_3, c_4, we cast a $(9m - v)$-bit constraint on the 16-tuples and nullify $n = 16m - (9m - v) = 7m + v$ bits in time complexity $T = 2^{2m+v}$, giving a tradeoff of

$$TM^5 = N,$$

namely $\alpha_{16} = 5$ as expected. Since we can choose any $0 \leq v \leq 9m$, the tradeoff is applicable for $T^{1/11} \leq M \leq T^{1/2}$.

4.2.2 Beyond 16-way list sum algorithms

In order to extend the tradeoff curve of $A_{4,1}$ to smaller memory ranges of $M \leq T^{1/2}$ we squared K. We can continue to extend the curve to very small memory values in a similar way by defining A_K for $K = 16^2 = 256$ and transforming it to $A_{K,1}$. For even smaller memory ranges, we use $K = 256^2 = 2^{16}$ and so forth.

4.3 Basic asymmetric list sum algorithms

4.3.1 $A_{7,1}$ and $A_{7,2^m}$

We extend the 7-way dissection A_7 of Sect. 3.3, whose time complexity is $2^{\tau_7 m} = 2^{4m}$ to $A_{7,1}$. According to the preliminary analysis above, we have $\alpha_7 = 7 - \tau_7 = 3$, as obtained in more detail below.

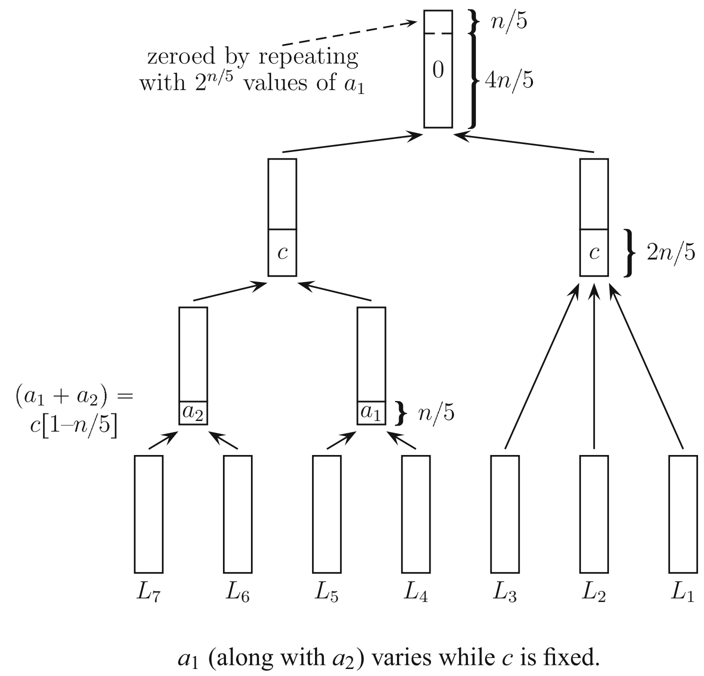
If we fix the $2m$-bit value c in the loop of A_7, we set a $2m$-bit constraint and can nullify $7m - 2m = 5m$ bits in time $2^{4m-2m} = 2^{2m}$. In general, when we vary 2^v times the value of c, we nullify $n = 5m + v$ bits in time $T = 2^{2m+v}$, giving

$$TM^3 = N,$$

namely $\alpha_7 = 3$ as obtained above. Since we can choose any $0 \leq v \leq 2m$, the tradeoff is applicable for $T^{1/4} \leq M \leq T^{1/2}$. The algorithm for the specific parameters $M = 2^{n/5}$, $T = 2^{2n/5}$ is sketched in Fig. 7.

4.3.2 $A_{11,1}$ and $A_{11,2^m}$

We consider the next value of $K = 11$ in the dissection sequence described in Sect. 3.3, which has time complexity of $T = 2^{7m}$ (nullifying $11m$ bits), i.e., $\tau_{11} = 7$. The main loop of A_{11} splits the problem into subproblem of sizes 4 and 7, while iterating over $3m$ intermediate target values. To construct $A_{11,1}$, we can easily fix these $3m$ values which reduces the time complexity to 2^{4m} (nullifying only $8m$ bits). In general, we obtain the tradeoff $TM^4 = N$



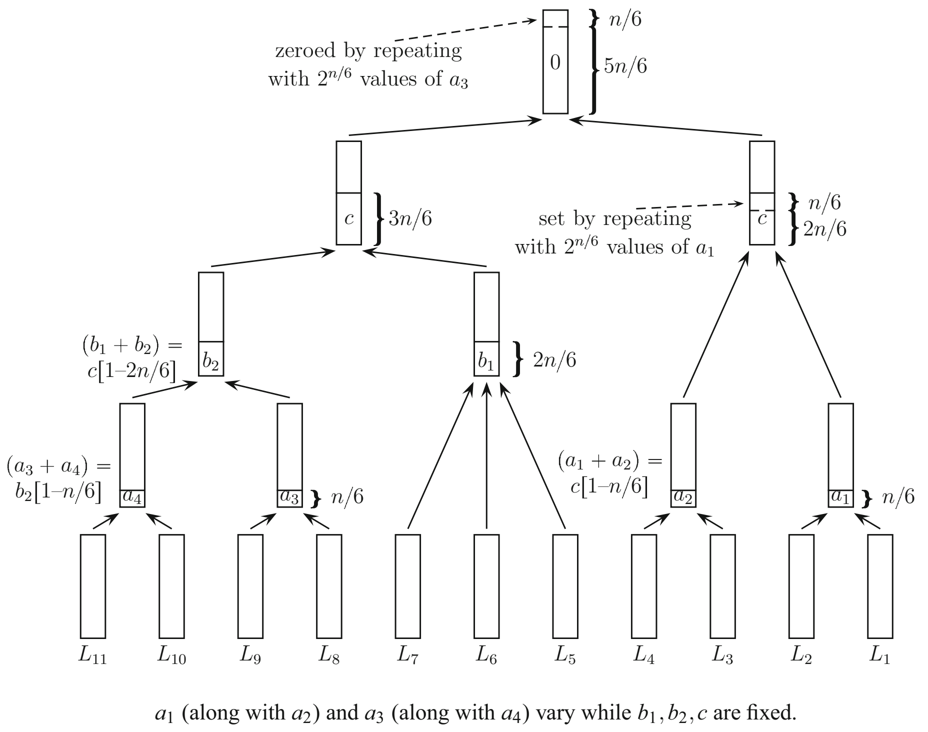
zeroed by repeating with $2^{n/5}$ values of a_1
$n/5$
0
$4n/5$
c
c
$2n/5$
$(a_1 + a_2) = c[1\text{–}n/5]$
a_2
a_1
$n/5$
L_7
L_6
L_5
L_4
L_3
L_2
L_1
a_1 (along with a_2) varies while c is fixed.

Fig. 7 $A_{7,1}$ with $M = 2^{n/5}, T = 2^{2n/5}$

$(\alpha_{11} = 11 - \tau_{11} = 4)$ for $T^{1/7} \leq M \leq T^{1/4}$. Interestingly, we can recursively fix more values and extend this tradeoff to $T^{1/7} \leq M \leq T^{1/2}$, which is crucial when the number of solutions is large. Below, we describe the algorithm for $M = T^{1/2}$ (i.e., $M = 2^{n/6}$, $T = 2^{2n/6}$). This algorithm is sketched in Fig. 8.

1. For a fixed $3m$-bit word c, apply the $A_{4,2^m}$ algorithm to $\{L_i\}_{i=1}^{4}$ with the target value c, and store the expected number of $2^{4m-3m} = 2^m$ outputs in a new sorted list L'. Each word is stored along with the corresponding 4-tuple of indexes from $\{L_i\}_{i=1}^{4}$.
2. Apply a $A_{7,2^{2m}}$ algorithm to $\{L_i\}_{i=5}^{11}$ with the $3m$-bit target value c by recursively fixing (additional) $2m$ bits (the expected number of solutions is indeed $2^{7m-3m-2m} = 2^{2m}$). For each returned solution $\{y_i\}_{i=5}^{11}$, look for matches with $\{y_i\}_{i=1}^{4}$ in L' and obtain an 11-tuple $\{y_i\}_{i=1}^{11}$ such that $\sum_{i=1}^{11} y_i = 0$ as required.

The complexity of both steps is 2^{2m}, hence $T = 2^{2m}$. Altogether, $3m + 2m = 5m$ bits are fixed and $n = 11m - 5m = 6m$ bits are nullified. Therefore, $M = 2^{n/6}$, $T = 2^{2n/6}$ as claimed. The ability to recursively fix target values (while maintaining the tradeoff of (3)) is a distinct feature of asymmetric algorithms. Next, we elaborate on which and how many values can be fixed this way for general K.



zeroed by repeating with $2^{n/6}$ values of a_3
$n/6$
$5n/6$
0
c
$3n/6$
set by repeating with $2^{n/6}$ values of a_1
c
$n/6$
$2n/6$
$(b_1 + b_2) = c[1\text{–}2n/6]$
b_2
b_1
$2n/6$
$(a_3 + a_4) = b_2[1\text{–}n/6]$
a_4
a_3
$n/6$
$(a_1 + a_2) = c[1\text{–}n/6]$
a_2
a_1
$n/6$
L_{11} L_{10} L_9 L_8 L_7 L_6 L_5 L_4 L_3 L_2 L_1
a_1 (along with a_2) and a_3 (along with a_4) vary while b_1, b_2, c are fixed.

Fig. 8 $A_{11,1}$ with $M = 2^{n/6}$, $T = 2^{2n/6}$

4.3.3 Generic analysis of basic asymmetric list sum algorithms

We analyze the transformation of the K_i-way dissection algorithm A_{K_i} (mentioned in Sect. 3.3) to $A_{K_i,1}$. As described in Sect. 3.3, A_{K_i} for $K_i = 1 + i(i+1)/2$ runs in time $2^{\tau_{K_i} m}$ for $\tau_{K_i} = 1 + i(i-1)/2$. Therefore $\alpha_{K_i} = K_i - \tau_{K_i} = i$, ideally giving the tradeoff

$$TM^i = N.$$

Determining the range of parameters for which this tradeoff applies is more subtle, as demonstrated for $A_{11,1}$ above. Recall from Sect. 3.3 that A_{K_i} splits the problem into subproblems of sizes $i, i-1, i-2, \ldots, 3, 2, 2$, and applies the algorithms $A_i, A_{i-1}, \ldots, A_3, A_2, A_2$, respectively. Hence, the time complexity of the algorithm cannot be reduced below the time complexity of A_i,[7] which is $2^{\tau_i m}$. In conclusion, the tradeoff is applicable in the range $T^{1/\tau_{K_i}} \leq M \leq T^{1/\tau_i}$.

For example, for $i = 3$, we have $K_3 = 7$, $\tau_7 = 4$ (as the time complexity of $A_{K_3} = A_7$ is 2^{4m}) and $\tau_3 = 2$ (as the time complexity of $A_i = A_3$ is 2^{2m}). Therefore, we obtain the tradeoff $TM^3 = N$, applicable for $T^{1/4} \leq M \leq T^{1/2}$, which indeed coincides with the $A_{7,1}$ tradeoff obtained above. For $i = 4$, we have $K_4 = 11$ and obtain the tradeoff $TM^4 = N$, applicable for $T^{1/7} \leq M \leq T^{1/2}$.

[7] We could try to fix additional intermediate target values that the algorithm A_i iterates over internally. However, this generally results in a less efficient tradeoff compared to $TM^i = N$.

Table 2 Multiple-layer list sum algorithms

Algorithm	Time-memory tradeoff	Range (M vs. T)	Reference
A_K^2	$T = M = N^{1/(\log K+1)}$	$T = M$	[15]
A_K^4	$T^{\lfloor(\log K)/2\rfloor} M^{\lceil(\log K)/2\rceil+1} = N$	$T^{1/2} \leq M \leq T$	New
A_K^7	$T^{\log_7 K} M^{2\log_7 K+1} = N$	$T^{1/4} \leq M \leq T^{1/2}$	New
A_K^{11}	$T^{\log_{11} K} M^{3\log_{11} K+1} = N$	$T^{1/7} \leq M \leq T^{1/2}$	New
A_K^{16}	$T^{\log_{16} K} M^{4\log_{16} K+1} = N$	$T^{1/11} \leq M \leq T^{1/2}$	New

5 Construction of new multiple-layer list sum algorithms

The second part of our framework uses the basic $A_{P,1}$ and $A_{P,2^m}$ algorithms developed in the previous sections in order to construct multi-layered[8] algorithms A_K^P. We recall the P is the arity of the tree of the multi-layered list sum algorithm (e.g., the K-tree algorithm uses $P = 2$, as it merges the lists in pairs). In this section we construct the algorithm A_K^P for any given choice of K, P, M. However, recall from Sect. 2 that P is not a formal parameter of the list sum problem, hence it is a free parameter. Therefore, given the parameters K, M, one should apply the algorithm A_K^P for a value of P that minimizes the time complexity.

The most relevant multi-layer list sum algorithms obtained in this section and in [15] are summarized in Table 2.

The analysis for A_K^P will use the parameter α_P established for $A_{P,1}$ according to its tradeoff curve (3).

5.1 Generic construction and analysis of multiple-layer algorithms

Given K, P, we write $K = P^k \cdot Q$, where $1 \leq Q < P$ and P, k, Q are integers (if K is not of the required form we round it down to K' of this form and apply the algorithm for K'). For example, if $K = 32$ and $P = 4$ then $Q = 2$ since $32 = 4^2 \cdot 2$. In this decomposition, we have $k = \log_P K$ (the logarithm is rounded down to the nearest integer). We now construct the algorithm A_K^P.

If $Q = 1$, A_K^P has $k - 1$ layers of $A_{P,2^m}$ (merging the lists in groups of P, where each merge outputs a list of 2^m inputs into the next layer) and a final layer of $A_{P,1}$. If $Q > 1$, A_K^P has k layers of $A_{P,2^m}$ and a final layer of $A_{Q,1}$. For example, A_{16}^4 has $\log_P K - 1 = 2 - 1 = 1$ layer of $A_{4,2^m}$ and one layer of $A_{4,1}$, while A_{32}^4 has $\log_P K = 2$ layers of $A_{4,2^m}$ and one layer of $A_{2,1}$.

First, we analyze the case of $K = P^k$ (namely, $Q = 1$), based on the tradeoff parameter α_P for $A_{P,1}$ (as specified in Eq. (3)). Fix a parameter n' such that each of the $k - 1$ layers of $A_{P,2^m}$ nullifies n' bits in time complexity $2^{n'-(\alpha_P-1)m}$, according to the tradeoff curve (4) for $A_{P,2^m}$. Altogether, $n'(k-1)$ bits are nullified in these layers and $n - n'(k-1)$ remain to be nullified by the final layer $A_{P,1}$ in time $2^{n-n'(k-1)-\alpha_P m}$. In order to balance the algorithm, we equate the time complexities of the layers by setting $n' - (\alpha_P - 1)m = n - n'(k-1) - \alpha_P m$ or $n' = (n-m)/k$. Consequently, the time complexity of the layered algorithm is $T = 2^{n'-(\alpha_P-1)m} = 2^{(n-m)/k-(\alpha_P-1)m} = 2^{(n-m(k(\alpha_P-1)+1))/k}$. This gives a time-memory tradeoff of $T^k M^{k(\alpha_P-1)+1} = N$ or

[8] Note that in this section we rename the parameter of basic algorithms from K to P.

$$T^{\log_P K} M^{\log_P K \cdot (\alpha_P - 1)+1} = N. \tag{5}$$

It is applicable for the same time-memory range as $A_{P,1}$ (and $A_{P,2^m}$).

For example, in the K-tree algorithm, we have $P = 2$ and $\alpha_2 = 2 - \tau_2 = 1$. Hence, the tradeoff of $T^{\log_P K} M^{\log_P K \cdot (\alpha_2 - 1)+1}$ collapses to $T^{\log K} M = N$. Setting $T = M$ (which is the only point in which the K-tree algorithm is directly applicable) gives the standard formula of $T^{\log K + 1} = N$ or $T = N^{1/(\log K+1)}$.

In case $K = P^k \cdot Q$ for $1 < Q < P$, the generic analysis becomes more involved and depends on the time-memory tradeoff curve of $A_{Q,1}$.

In this paper we focus on $Q = 2$, where the final merge is a basic $A_{2,1}$ algorithm that runs in fixed time complexity of 2^m and nullifies $2m$ bits. Fix a parameter n' such that each of the k layers of $A_{P,2^m}$ nullifies n' bits in time complexity $2^{n' - (\alpha_P - 1)m}$. Altogether, $n'k$ bits are nullified in these layers and $n - n'k$ remain to be nullified by the final layer $A_{2,1}$ in time 2^m. Since the final 2-way list sum algorithm nullifies $2m$ bits, we have $n - n'k = 2m$ or $n' = (n - 2m)/k$.

The algorithm's time complexity is dominated by the first k layers and is therefore $T = 2^{n' - (\alpha_P - 1)m} = 2^{(n-2m)/k - (\alpha_P - 1)m} = 2^{(n - m(k(\alpha_P - 1)+2))/k}$. This gives a time-memory tradeoff of

$$T^{\log_P K} M^{\log_P K \cdot (\alpha_P - 1)+2} = N, \tag{6}$$

applicable for the same time-memory parameter range as $A_{P,1}$ (and $A_{P,2^m}$).

5.2 Analysis of specific multiple-layer list sum algorithms

We analyze the A_K^4 algorithm according to the generic approach above. The analysis of the rest of the layered algorithms A_K^P summarized in Table 2 is obtained in a similar manner by plugging the relevant α_P parameter into (5).

5.2.1 A_K^4

For arity $P = 4$, we only analyze values of K which are powers of 2 (as this allows direct comparison to previous tradeoffs for GBP). For $A_{4,1}$, we have $\alpha_4 = 2$, established in Sect. 4.2.

In case $K = 4^k$, we plug $\log_P K = \log_4 K = \log K/2$ and $\alpha_P = \alpha_4 = 2$ into (5), obtaining

$$T^{(\log K)/2} M^{(\log K)/2+1} = N,$$

applicable for $T^{1/2} \leq M \leq T$ (as the $A_{4,1}$ and $A_{4,2^m}$ algorithms).

We demonstrate the A_{16}^4 algorithm (that works in two layers with $Q = 1$) below.

1. Apply the $A_{4,2^m}$ algorithm of Sect. 3.4 four times to lists $\{L_i\}_{i=1}^4$, $\{L_i\}_{i=5}^8$, $\{L_i\}_{i=9}^{12}$ and $\{L_i\}_{i=13}^{16}$, nullifying $n' = (n-m)/2$ bits. Obtain four sorted lists L_1', L_2', L_3', L_4', each of expected size 2^m.
2. Apply the $A_{4,1}$ algorithm of Sect. 3.4 to L_1', L_2', L_3', L_4', nullifying the remaining $n - n' = (n + m)/2$ bits. Derive a single solution to the list sum problem $\sum_{i=1}^{16} y_i[1\text{–}n] = 0$.

Table 3 Time-memory tradeoffs of GBP algorithms

K	Time-memory tradeoff	Range (M vs. T)	Building blocks
≥ 8	$T^{\lceil(\log K)/2\rceil+1}M^{\lfloor(\log K)/2\rfloor} = N$	$T^{1/2} \leq M \leq T$	PCS + $A^4_{K/2}$
≥ 8	$T^2 M^{3\lceil(\log K)/2\rceil-2-(\log K) \bmod 2} = N$	$1 \leq M \leq T^{1/2}$	PCS + $A^4_{K/2}$
8	$T^3 M = N$	$T^{1/2} \leq M \leq T$	PCS + $A_{4,1}$
8	$T^2 M^3 = N$	$1 \leq M \leq T^{1/2}$	PCS + $A_{4,1}$
16	$T^3 M^2 = N$	$T^{1/2} \leq M \leq T$	PCS + A^4_8
16	$T^3 M^2 = N$	$T^{1/4} \leq M \leq T^{1/2}$	PCS + $A_{7,1}$
16	$T^2 M^6 = N$	$1 \leq M \leq T^{1/4}$	PCS + $A_{7,1}$
32	$T^4 M^2 = N$	$T^{1/2} \leq M \leq T$	PCS + A^4_{16}
32	$T^3 M^4 = N$	$T^{1/11} \leq M \leq T^{1/2}$	PCS + $A_{16,1}$
32	$T^2 M^{15} = N$	$1 \leq M \leq T^{1/11}$	PCS + $A_{16,1}$
6	$T^2 M^2 = N$	$1 \leq M \leq T^{1/2}$	PCS + $A_{3,1}$
7	$T^2 M^2 = N$	$T^{1/4} \leq M \leq T^{1/2}$	CTP + $A_{7,1}$
14	$T^3 M^2 = N$	$T^{1/4} \leq M \leq T^{1/2}$	PCS + $A_{7,1}$
14	$T^2 M^6 = N$	$1 \leq M \leq T^{1/4}$	PCS + $A_{7,1}$

In case $K = 2 \cdot 4^k$, we have $P = 4$, $Q = 2$ and $\log_4 K = (\log K - 1)/2$. Plugging these values into (6) we obtain

$$T^{(\log K-1)/2} M^{(\log K-1)/2+2} = N,$$

applicable for $T^{1/2} \leq M \leq T$.

Unified formula for powers of 2 Unifying the tradeoff formulas obtained above to any $K = 2^k$, we obtain

$$T^{\lfloor(\log K)/2\rfloor} M^{\lceil(\log K)/2\rceil+1} = N,$$

applicable for $T^{1/2} \leq M \leq T$.

6 Construction of new algorithms for the generalized birthday problem

The third part of our framework combines preparation phase and list sum algorithms (for $S = 1$) to obtain new GBP algorithms. It combines either PCS (memory-consuming parallel collision search) or CTP (clamping through precomputation) with a given list sum algorithm. The generic formulas are then used to obtain improved tradeoffs for various specific K values (some of which are specified in Table 3). We focus on values of K that allow direct comparison to previous tradeoff curves in addition to values that are relevant to Sect. 7, where we analyze concrete GBP instances.

For a parameter K, we construct a GBP algorithm assuming we have a list sum algorithm A (which can be a basic or layered algorithm) with $S = 1$ for a parameter K' (whose value will be either $K' = K$ or $K' = K/2$). We assume that the time-memory tradeoff curve of A for K' is $T^\beta M^\gamma = N$ for parameters β, γ. Our analysis also uses a parameter δ, which specifies the lower range limit for the tradeoff algorithm of A as $M \geq T^{1/\delta}$. For example, we established the tradeoff $TM^3 = N$ for $A_{7,1}$ in the range $T^{1/4} \leq M \leq T^{1/2}$, hence we have $\beta = 1, \gamma = 3, \delta = 4$. We will derive a basic tradeoff for $M \geq T^{1/\delta}$ and then extend it to $M < T^{1/\delta}$. Furthermore, the tradeoffs depend on the input range size $L = 2^{\ell}$ since the complexity of the preparation phase algorithm varies according to whether or not ℓ is below some value. For example, when applying PCS with a small value of ℓ, we have to apply it to an expanding function and adapt the complexity as specified on Sect. 3.6.

Altogether, a GBP tradeoff formula depends on a triplet of parameters (preparation phase algorithm, memory range, input size range) and there are $2^3 = 8$ such possible triplets. However, only 6 are relevant in this paper and they are summarized in Table 4. Deriving these formulas is a mechanical procedure of parameter optimization. Below, we explicitly derive the first 4 tradeoffs for the specific case of $K = 14$ with the PCS preparation phase.

6.1 Tradeoff algorithm for $K = 14$

For $K = 14$ with the PCS preparation phase, we use an algorithm A for $K' = K/2 = 7$, namely $A_{7,1}$. The tradeoff formula for $A_{7,1}$ is $TM^3 = N$ (i.e., its time complexity is $T = 2^{n-3m}$) in the range $T^{1/4} \leq M \leq T^{1/2}$. The combination algorithm uses the truncated function $H_{|r,r}$ (as defined in Sect. 2) for a parameter $r \geq m$, set below to optimize the algorithm.

1. Run PCS on the function $H_{|r,r}$ and look for 2^m collisions. For each collision $H(x)[1–r] = H(x')[1–r]$, compute $y = H(x) + H(x')$ and store all these words in 7 lists $\{L_i\}_{i=1}^{7}$ of size about 2^m (along with the corresponding x values).
2. Run A on $\{L_i\}_{i=1}^{7}$ (nullifying bits $[r + 1–n]$). Obtain 7 words $y_i \in L_i$ such that $\sum_{i=1}^{7} y_i = 0$ and use them (based on the first step) to construct a solution to GBP by recovering the K words x_i such that $\sum_{i=1}^{14} H(x_i) = \sum_{i=1}^{7} y_i = 0$ as required.

In the first step, we execute PCS with parameter r in time $2^{(r+m)/2}$ (according to Sect. 3.6). In the second step we nullify the remaining $n' = n - r$ bits by using $A_{7,1}$ in time complexity $2^{n'-3m} = 2^{n-r-3m}$. Then, to balance the two steps we require $(r + m)/2 = n - r - 3m$ or $r = (2n - 7m)/3$, giving time complexity of $T = 2^{(n-2m)/3}$ and a tradeoff of

$$T^3 M^2 = N.$$

This matches Tradeoff 1 for $K = 14$ in Table 4 (recall that for $A_{7,1}$, $\beta = 1, \gamma = 3, \delta = 4$). This tradeoff is valid for $T^{1/4} \leq M \leq T^{1/2}$ (as $A_{7,1}$). The algorithm for parameters $T = N^{1/4}$, $M = N^{1/8}$ is sketched in Fig. 9.

When $M < T^{1/4}$, we can extend the tradeoff by applying $A_{7,1}$ to nullify less bits (i.e., with a smaller value of n'), implying that PCS will nullify more bits and dominate the complexity of the algorithm which becomes $T = 2^{(r+m)/2} = 2^{(n-n'+m)/2}$. In order to calculate n', we use the tradeoff curve $\hat{T}M^3 = N'$ of $A_{7,1}$ at its lower range $M = \hat{T}^{1/4}$ or $\hat{T} = M^4$ (here, $\hat{T}$ denotes the time complexity of $A_{7,1}$) and obtain $N' = M^7$, namely $n' = 7m$. Therefore, $T = 2^{(n-n'+m)/2} = (n - 6m)/2$, giving the tradeoff

$$T^2 M^6 = N.$$

This matches Tradeoff 2 in Table 4.

Table 4 Generic GBP time-memory tradeoffs formulas

Tradeoff	Time-memory tradeoff	K'	Range (M vs. T)	Range (L)	Building blocks
1	$T^{\beta+2}M^{\gamma-1} = N$	$K/2$	$M \geq T^{1/\delta}$ (same as A)	$\ell \geq \frac{2n-2m\gamma-m\beta}{\beta+2}$	PCS + A
2	$T^2 M^{\delta\beta+\gamma-1} = N$	$K/2$	$M \leq T^{1/\delta}$	$\ell \geq n - m(\delta\beta + \gamma)$	PCS + A
3	$T^{\beta+1}M^{\gamma-1/2}L^{1/2} = N$	$K/2$	$M \geq T^{1/\delta}$ (same as A)	$\ell \leq \frac{2n-2m\gamma-m\beta}{\beta+2}$	PCS + A
4	$T M^{\delta\beta+\gamma-1/2}L^{1/2} = N$	$K/2$	$M \leq T^{1/\delta}$	$\ell \leq n - m(\delta\beta + \gamma)$	PCS + A
5	$T^{\beta+1}M^{\gamma-1} = N$	K	$M \geq T^{1/\delta}$ (same as A)	$\ell \geq \frac{n-m\gamma+m}{\beta+1}$	CTP + A
6	$T^{\beta}M^{\gamma-1}L = N$	K	$M \geq T^{1/\delta}$ (same as A)	$\ell \leq \frac{n-m\gamma+m}{\beta+1}$	CTP + A

6.1.1 Restricted domain

Recall from the first tradeoff that $r = (2n - 7m)/3$. In case $\ell < r = (2n - 7m)/3$ (the domain size of H is $2^{\ell} < 2^{r}$) we are forced to use $H_{|\ell,r}$ which is an expanding function. The time complexity of PCS for the expanding function $H_{|\ell,r}$ is $2^{r+(m-\ell)/2}$ (as specified in Sect. 3.6), while the time complexity of $A_{7,1}$ remains $T = 2^{n-r-3m}$. Balancing the steps in this case gives $r + (m - \ell)/2 = n - r - 3m$ or $r = n/2 - 7m/4 + \ell/4$ and time complexity of $T = r + (m - \ell)/2 = n/2 - 5m/4 - \ell/4$. This gives the formula

$$T^2 M^{5/2} L^{1/2} = N,$$

matching Tradeoff 3 in Table 4.

When $M < T^{1/4}$, the steps cannot be balanced as above and the time complexity becomes $2^{r+(m-\ell)/2}$ (dominated by PCS). Here, $r = n - n' = n - 7m$ ($n' = 7m$ as in the case where the domain is not restricted). We obtain $T = 2^{n-13m/2-\ell/2}$, giving the formula

$$T M^{13/2} L^{1/2} = N$$

and matching Tradeoff 4 in Table 4.

6.2 Tradeoff formulas for $K = 2^k$ (and $K \geq 8$)

We now derive improved GBP tradeoffs for $K = 2^k$ assuming that $K \geq 8$. These tradeoffs are obtained by combining PCS with multiple layers of the 4-way list sum algorithm $A^4_{K/2}$ (described in Sect. 5) as the generic algorithm A. The formula is calculated according to Tradeoff 1 in Table 4.

We recall the formula of $A^4_{K/2}$ from Table 2, which[9] is $T^{\lfloor(\log K-1)/2\rfloor} M^{\lceil(\log K-1)/2\rceil+1} = N$, namely $\beta = \lfloor(\log K - 1)/2\rfloor = \lceil(\log K)/2\rceil - 1$ and $\gamma = \lceil(\log K - 1)/2\rceil + 1 = \lfloor(\log K)/2\rfloor + 1$. Adding 2 to β and reducing γ by 1 (as in Tradeoff 1 in Table 4), we obtain

$$T^{\lceil(\log K)/2\rceil+1} M^{\lfloor(\log K)/2\rfloor} = N,$$

applicable for $T^{1/2} \leq M \leq T$.

6.2.1 Extending the tradeoffs to $M < T^{1/2}$

In case $M < T^{1/2}$, PCS dominates the algorithm's time complexity and the formula is given in Tradeoff 2 in Table 4 as $T^2 M^{\delta\beta+\gamma-1}$, where $\delta = 2$. We obtain $T^2 M^{2\lceil(\log K)/2\rceil-2+\lfloor(\log K)/2\rfloor} = N$, or

$$T^2 M^{3\lceil(\log K)/2\rceil-2-(\log K) \bmod 2} = N.$$

The tradeoff is applicable for $M \leq T^{1/2}$.

6.2.2 Tradeoff formulas for $K = 16$, $K = 32$ and beyond

The tradeoff curve for $T^{1/2} \leq M \leq T$ above is the best we can obtain for $K = 2^k$. On the other hand, the curve for $M < T^{1/2}$ is only optimal for $K = 8$ as for larger K values it is possible to apply more complex list sum algorithms as described in Sect. 5.

9 Note that the evaluation is performed at $K/2$ rather than K.

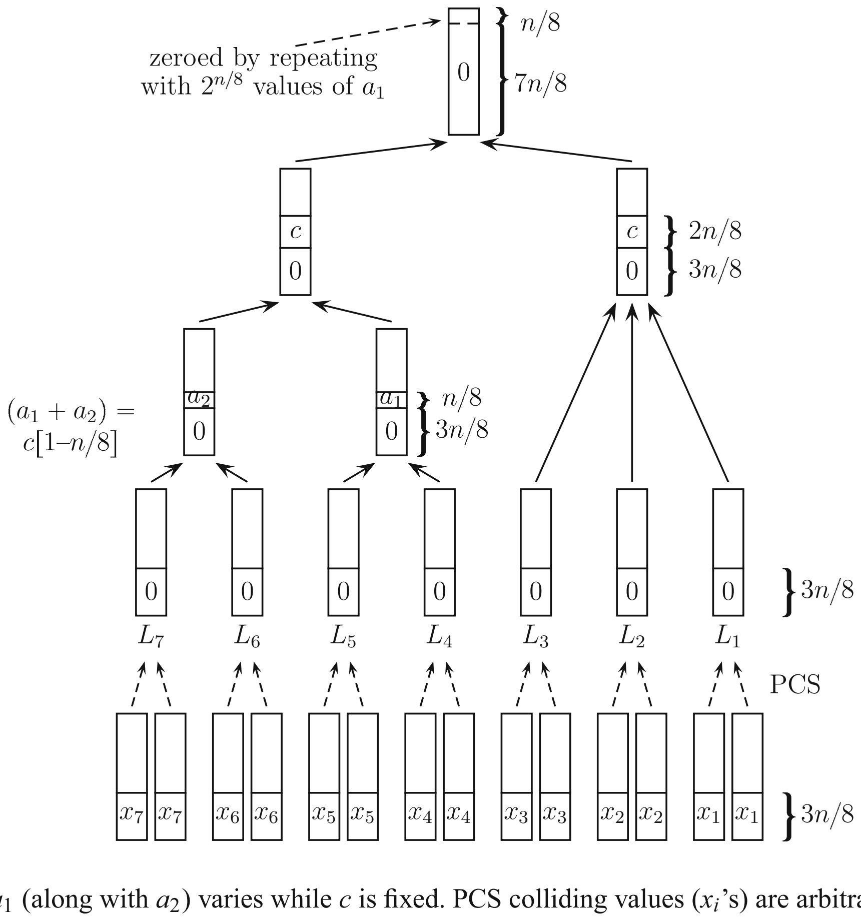


Fig. 9 GBP algorithm for $K = 14$ with $M = 2^{n/8}$, $T = 2^{n/4}$

While we cannot obtain generic optimal formulas that are applicable to all values of $K = 2^k$ and $M < T^{1/2}$, we further extend the tradeoffs for $K = 16$ and $K = 32$ in Table 3. The results were obtained using the formulas of the combination algorithms (Tradeoffs 1,2 in Table 4) after plugging in the parameters of the relevant list sum algorithms (specified as building blocks in Table 3). The methods to obtain tradeoffs for $K = 64$ and beyond are similar.

6.3 Tradeoff formulas for $K \in \{6, 7, 14\}$

As for values of $K = 2^k$ analyzed above, the results for $K \in \{6, 7, 14, 15\}$ in Table 3 were obtained using the formulas of the combination algorithms with the list sum algorithms specified as building blocks in the table.

Table 5 Some GBP algorithms for restricted domains

K	Time-memory tradeoff	Range (M vs. T)	Domain range	Building blocks
6	$TM^{5/2}L^{1/2} = N$	$1 \leq M \leq T^{1/2}$	$L \leq NM^{-3}$	PCS + $A_{3,1}$
7	$TM^{2}L = N$	$T^{1/4} \leq M \leq T^{1/2}$	$L \leq N^{1/2}M^{-1}$	CTP + $A_{7,1}$
14	$T^{2}M^{5/2}L^{1/2} = N$	$T^{1/4} \leq M \leq T^{1/2}$	$L \leq N^{2/3}M^{-7/3}$	$hbox PCS + A_{7,1}$

Table 6 Complexities for concrete GBP instances

K	n	ℓ	Time/memory [11] (Extended K-tree)	Our time/memory	Our tradeoff
6	120	37	$T = M = 2^{48}$	$T = 2^{48}/M = 2^{25}$	$TM^{5/2}L^{1/2} = N$
6	120	32	$T = M = 2^{58}$	$T = 2^{49}/M = 2^{24}$	$TM^{5/2}L^{1/2} = N$
7	120	36	$T = M = 2^{50}$	$T = 2^{45}/M = 2^{24}$	$TM^{2}L = N$
7	120	28	$T = M = 2^{66}$	$T = 2^{49}/M = 2^{26}$	$TM^{2}L = N$
14	120	14	$T = M = 2^{67}$	$T = 2^{39}/M = 2^{20}$	$T^{2}M^{5/2}L^{1/2} = N$

6.3.1 Restricted domains

Using combination algorithms for restricted domains $L = 2^{\ell} < N$, we derive additional tradeoffs in Table 5.[10] These tradeoffs are applicable to the concrete GBP instances analyzed in the next section.

7 Restricted domain instances of the generalized birthday problem

The best known algorithm for GBP instances with restricted domain is the extended K-tree algorithm [11], which can be directly applied to values of $K = 2^k$ for any $N^{1/K} \leq L \leq N^{1/(\log K+1)}$. Here, we use our GBP tradeoffs derived in Sect. 6.3 (specified in Table 5) and compare the results for some GBP instances in Table 6.

For fair comparison, we multiply the time and memory complexities obtained by our general formulas above by K. For example, for $K = 6, n = 120, \ell = 37$ the relevant curve in Table 5 is $TM^{5/2}L^{1/2} = N$ for $M \leq T^{1/2}$. To optimize the time complexity we set $M = T^{1/2}$, giving $T = N^{4/9}L^{-2/9} \approx 2^{45}$. After multiplication by $K = 6$, we obtain $M \approx 2^{25}, T \approx 2^{48}$.

Generally, our tradeoffs give better results for values of K that are far away from their nearest smaller power of 2. However, we remark that our algorithms also give interesting tradeoffs for some instances with values of K that are powers of 2. For example, for $K = 16, n = 120, \ell = 15$, the extended K-tree algorithm gives $T = M = 2^{34}$. Assume we want to keep the memory close to 2^{15}, then we can directly apply the 16-way list sum algorithm A_{16}^{4} of Sect. 5 (with the tradeoff of $T^2M^3 = N$) and obtain (after multiplication with $K = 16$),

[10] These algorithms are not optimal for a very small domain size close to $2^{n/K}$, which is the minimal value required for a solution to exist with high probability. In such cases it is more efficient to apply a final layer of random-walk collision search (as done in [4,7]). However, the practical relevance of these algorithms is relatively limited and they are beyond the scope of this paper (as we focus on practical tradeoffs for GBP instances with limited domain sizes).

$M = 2^{19}$ and $T \approx 2^{41.5}$. Hence we reduce the memory complexity by a factor of 2^{15} and increase the time complexity by $2^{7.5}$, which may be preferable is practical settings.

8 Conclusions

In this paper, we devised new GBP time-memory tradeoff algorithms, improving the state-of-the-art for a large range of parameters. The improvement is mainly due to a careful transformation of exhaustive list sum memory-efficient algorithms (i.e., variants of the Schroeppel–Shamir algorithm and dissection algorithms) to algorithms that efficiently find a single solution to the list sum problem. It is thus plausible that future improvements to exhaustive list sum algorithms will also result in improved time-memory tradeoffs for GBP using our transformation.

Acknowledgements The author was supported in part by the Israeli Science Foundation through Grant No. 573/16.

References

1. Becker A., Coron J., Joux A.: Improved generic algorithms for hard knapsacks. In: Paterson K.G. (ed) Advances in Cryptology—EUROCRYPT 2011—30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15–19, 2011. Lecture Notes in Computer Science, vol. 6632, pp. 364–385. Springer, New York (2011).
2. Becker A., Joux A., May A., Meurer A.: Decoding random binary linear codes in $2^{n/20}$: how $1 + 1 = 0$ improves information set decoding. In: Pointcheval D., Johansson T. (eds.) Advances in Cryptology—EUROCRYPT 2012—31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Lecture Notes in Computer Science, vol. 7237, pp. 520–536. Springer, New York (2012).
3. Bernstein D.J.: Better price-performance ratios for generalized birthday attacks. In: SHARCS07: Special-Purpose Hardware for Attacking Cryptographic Systems (2007).
4. Bernstein D.J., Lange T., Niederhagen R., Peters C., Schwabe P.: FSBday. In: Roy B.K., Sendrier N. (eds.) Progress in Cryptology—INDOCRYPT 2009, 10th International Conference on Cryptology in India, New Delhi, India, December 13–16, 2009. Lecture Notes in Computer Science, vol. 5922, pp. 18–38. Springer, New York (2009).
5. Biryukov A., Khovratovich D.: Equihash: asymmetric proof-of-work based on the generalized birthday problem. In: 23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21–24, 2016. The Internet Society (2016).
6. Canteaut A., Trabbia M.: Improved fast correlation attacks using parity-check equations of weight 4 and 5. In: Preneel B. (ed.) Advances in Cryptology—EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14–18, 2000. Lecture Notes in Computer Science, vol. 1807, pp. 573–588. Springer, New York (2000).
7. Dinur I., Dunkelman O., Keller N., Shamir A.: Efficient dissection of composite problems, with applications to cryptanalysis, knapsacks, and combinatorial search problems. In: Safavi-Naini R., Canetti R. (eds.) Advances in Cryptology—CRYPTO 2012—32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2012. Lecture Notes in Computer Science, vol. 7417, pp. 719–740. Springer, New York (2012).
8. Finiasz M., Sendrier N.: Security bounds for the design of code-based cryptosystems. In: Matsui M. (ed.) Advances in Cryptology—ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6–10, 2009. Lecture Notes in Computer Science, vol. 5912, pp. 88–105. Springer, New York (2009).
9. Howgrave-Graham N., Joux A.: New generic algorithms for hard knapsacks. In: Gilbert H. (ed.) Advances in Cryptology—EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Lecture Notes in Computer Science, vol. 6110, pp. 235–256. Springer, New York(2010).

10. Meier W., Staffelbach O.: Fast correlation attacks on certain stream ciphers. J. Cryptol. **1**(3), 159–176 (1989).
11. Minder L., Sinclair A.: The extended k-tree algorithm. J. Cryptol. **25**(2), 349–382 (2012).
12. Nikolic, I., Sasaki, Y.: Refinements of the k-tree Algorithm for the Generalized Birthday Problem. In: T. Iwata and J. H. Cheon, editors, Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II, volume 9453 of Lecture Notes in Computer Science, pages 683–703. Springer, (2015)
13. Schroeppel R., Shamir A.: AT $= O(2^{n/2})$, S $= O(2^{n/4})$ algorithm for certain NP-complete problems. SIAM J. Comput. **10**(3), 456–464 (1981).
14. van Oorschot P.C., Wiener M.J.: Parallel collision search with cryptanalytic applications. J. Cryptol. **12**(1), 1–28 (1999).
15. Wagner D.A.: A generalized birthday problem. In: Yung M. (ed.) Advances in Cryptology—CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18–22, 2002. Lecture Notes in Computer Science, vol. 2442, pp. 288–303. Springer, New York(2002).