

# Dissection-BKW

Andre Esser<sup>1</sup>, Felix Heuer<sup>1</sup>, Robert Kübler<sup>1</sup>,  
Alexander May<sup>1</sup>, and Christian Sohler<sup>2</sup>

<sup>1</sup> Horst Görtz Institute for IT Security  
Ruhr University Bochum, Germany  
`{andre.esser,felix.heuer,robert.kuebler,alexander.may}@rub.de`  
<sup>2</sup> Department of Computer Science  
TU Dortmund, Germany  
`christian.sohler@tu-dortmund.de`

**Abstract.** The slightly subexponential algorithm of Blum, Kalai and Wasserman (BKW) provides a basis for assessing LPN/LWE security. However, its huge memory consumption strongly limits its practical applicability, thereby preventing precise security estimates for cryptographic LPN/LWE instantiations.

We provide the first time-memory trade-offs for the BKW algorithm. For instance, we show how to solve LPN in dimension  $k$  in time  $2^{\frac{4}{3}\frac{k}{\log k}}$  and memory  $2^{\frac{2}{3}\frac{k}{\log k}}$ . Using the Dissection technique due to Dinur et al. (Crypto '12) and a novel, slight generalization thereof, we obtain fine-grained trade-offs for any available (subexponential) memory while the running time *remains subexponential*. Reducing the memory consumption of BKW below its running time also allows us to propose a first quantum version QBKW for the BKW algorithm.

## 1 Introduction

The Learning Parity with Noise (LPN) problem [5] and its generalization to arbitrary moduli, the Learning with Errors (LWE) problem [33], lie at the heart of our most promising coding-based and lattice-based post-quantum cryptographic constructions [3,32,29]. With the NIST standardization [1], we have the urgent pressure to identify LPN/LWE instantiations that allow for efficient constructions, but yet give us the desired level of classic and quantum security.

Hence, we have to run cryptanalytic algorithms on medium-scale parameter sets in order to properly extrapolate towards cryptographic instantiations.

In LPN of dimension  $k$  and error-rate  $0 \leq p < 1/2$ , one has to recover a secret  $\mathbf{s} \in \mathbb{F}_2^k$  from samples  $(\mathbf{a}_i, \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i)$  for uniformly random  $\mathbf{a}_i \in \mathbb{F}_2^k$  and inner products with Bernoulli distributed error  $e_i$ , i.e.,  $\Pr[e_i = 1] = p$ .

It is not hard to see that for constant error  $p$ , any candidate solution  $\mathbf{s}'$  can be checked for correctness in time polynomial in  $p$  and  $k$ . This gives a simple brute-force LPN algorithm with time complexity  $2^k$  and constant memory.

The algorithm of Blum, Kalai and Wassermann [6] solves the faulty system of linear equations from LPN/LWE by a block-wise Gaussian elimination. In a nutshell, BKW takes sums of two vectors whenever they cancel a block of  $\Theta(\frac{k}{\log k})$  bits to keep the accumulating error under control. In cryptographic constructions we usually have constant  $p$ , for which the BKW algorithm runs in time  $2^{\frac{k}{\log k}(1+o(1))}$ , albeit using the same amount of memory.

These two algorithms, brute-force and BKW, settle the initial endpoints for our time-memory trade-offs. Hence, for gaining a  $\log(k)$ -factor in the run time exponent one has to invest memory up to the point where memory equals run time. Interestingly, this behavior occurs in the second method of choice for measuring LWE security, lattice reduction, too.

Whereas on lattices of dimension  $n$ , lattice enumeration such as Kannan's algorithm [22] takes time  $2^{\mathcal{O}(n \log n)}$  with polynomial memory only, lattice sieving methods [26,13,25,27] require  $2^{\mathcal{O}(n)}$  time and memory. Due to their large memory requirements, in practice lattice sieving is currently outperformed by enumeration, and there is an increasing interest in constructing time-memory trade-offs, e.g., by lattice tuple sieving [4,18,19].

For BKW, the research so far mainly concentrated on run time, where many optimizations have been proposed in the cryptographic literature, e.g. [28,16,17,36,8]. While these improvements may significantly

Table 1:  $c$ -sum-algorithms: Given a list  $L$  and some target  $t$ , the algorithms output  $|L|$  sets each containing  $c$  entries from  $L$  adding to  $t$ . Memory consumption of the algorithms coincides with the size of list  $L$ . Let  $N_c := (M_{\text{BKW}})^{\frac{\log c}{c-1}} = 2^{\frac{\log c}{c-1} \frac{k}{\log k}}$ .

	$c$ -sum Algorithm	Memory	Time	for	
classic	sorting (BKW)	$N_2$	$N_2$	$c = 2$	[6]
	Naive	$N_c$	$N_c^{c-1}$	$c \geq 2$	Section 4.1
	Dissection	$N_c$	$N_c^{c-\sqrt{2c}}$	$c = 4, 7, 11, 16, \dots$	Section 5.2
	Tailored Dissection	$N_c^\alpha$	$N_c^{c-\alpha\sqrt{2c}}$	$c = 4, 7, 11, 16, \dots$ $\alpha \in [1, \frac{\sqrt{c}}{\sqrt{c-1}}]$	Section 5.3
quantum	Naive + Grover	$N_c$	$N_c^{c/2}$	$c \geq 2$	Section 4.2

reduce the running time for breaking concrete LPN instances such as  $(k, p) = (512, 1/4)$  or  $(512, 1/8)$ , to the best of our knowledge for the running time exponent  $\frac{k}{\log k}(1 + o(1))$  all improvements only affect the  $o(1)$ -term. Moreover, all proposals share the same huge memory requirements as original BKW, making it impossible to run them even in moderately large dimensions.

As a consequence state-of-the-art BKW implementations are currently only possible in dimension  $k$  up to around 100. For instance [14] reported a break of  $(k, p) = (135, 1/4)$ . However, of the total 6 days running time, the authors spent 2.5 days for an *exponential* preprocessing, followed by less than 2 hours BKW in dimension 99, and another 3.5 days of *exponential* decoding. The reason for this run-time imbalance is that BKW in dimension 99 had already consumed all available memory, namely  $2^{40}$  bits.

Hence, if we really want to break larger LPN instances in practice, we must study time-memory trade-offs that sacrifice a bit of running time, but stay in the *sub-exponential time* regime at the benefit of a manageable memory.

**Our contribution.** We provide the first time-memory trade-offs for the BKW algorithm. These trade-offs give us a smooth interpolation for the complexities of solving LPN between the known endpoints  $2^k$  time for brute-force and  $2^{\frac{k}{\log k}}$  for BKW.

Since our algorithms' running times remain subexponential even for given memory below the requirement  $2^{\frac{k}{\log k}}$  of classic BKW, we (asymptotically) outperform all previous algorithms (e.g. [14]) that solved LPN in exponential time when classic BKW was not applicable due to memory restrictions.

As a starting point, we consider—instead of 2-sums as in the original BKW algorithm— $c$ -sums for (constant)  $c > 2$  that cancel some blocks of bits. The use of sums of more than 2 vectors has already been discussed in the literature for improving the running time of BKW, e.g. by Zhang et al. [36] as an extension  $LF(k)$  of the BKW variants  $LF1$  and  $LF2$  by Leveil and Fouque [28], using Wagner's  $k$ -list algorithm [35].

Since the number of  $c$ -sums grows exponentially in  $c$ , so does the number of  $c$ -sums whose sum cancels some block of bits. In turn, we systematically use  $c$ -sums to significantly lower the number of samples  $N$  that need to be stored, at the slightly increased cost of finding such  $c$ -sums.

We show that the complexities of any  $c$ -sum BKW algorithm are dominated by the cost of computing  $c$ -sums. As a consequence we abstract the *c-sum-problem* from the BKW algorithm and study various memory-friendly algorithms to solve it. We ensure that our  $c$ -sum algorithms do not require more memory than already consumed by  $c$ -sum BKW for storing its samples. In fact, our BKW algorithms have sample and (thus) memory complexity as little as  $N_c := M_{\text{BKW}}^{\frac{\log c}{c-1}}$  for any constant  $c \in \mathbb{N}$ , where  $M_{\text{BKW}} := 2^{\frac{k}{\log k}}$  denotes the memory (and sample) requirement of classic BKW.

In Table 1, we give a brief overview of our  $c$ -sum algorithms complexities, and therefore also of our  $c$ -sum BKW complexities. We stress that all  $c$ -sum algorithms from Table 1, including those that use the Dissection technique [34,11], may be studied for arbitrary list sizes outside of the BKW context.

**NAIVE.** We first consider a naive approach that computes all  $(c-1)$ -sums of list entries and looks for some matching  $c^{\text{th}}$  vector. This naive approach already gives us a smooth first time-memory trade-off, informally captured in the following theorem.

**Theorem 1.1 (Naive BKW Trade-Off, informal).** *Let  $c \in \mathbb{N}$ . The LPN problem in dimension  $k$  can be solved in Time  $T$  and space  $M$  where*

$$\log T = \log c \cdot \frac{k}{\log k} \quad , \quad \log M = \frac{\log c}{c-1} \cdot \frac{k}{\log k} \quad .$$

Observe that the trade-off behaves quite nicely as a function of  $c$ . While we can reduce the memory consumption almost by a factor of  $\frac{1}{c}$  this comes at the cost of only a  $(\log c)$ -factor in the run time exponent.

Note that for  $c = 2$  Theorem 1.1 yields the well-known BKW complexities. While we consider constant  $c$  throughout this work, we point out that our results hold up to a choice of  $c(k) = k^{1 - \frac{\log \log k}{\log k}}$  for which the formulas in Theorem 1.1 (as well as for the upcoming trade-offs) result in exponential running time in  $k$  with polynomial memory, matching the endpoint of the LPN brute-force algorithm. See Figure 1 (stars) for an illustration of this time-memory trade-off.

**QBKW.** Using a standard Grover-search in our naive  $c$ -sum algorithm to identify  $(c-1)$ -sums for which there exists a matching  $c^{\text{th}}$  vector, we halve the running time complexity exponent from  $\log c \cdot \frac{k}{\log k}$  down to  $\frac{\log c}{2} \cdot \frac{k}{\log k}$ . See Figure 1 (green triangles) for the resulting trade-off curve.

**DISSECTION.** We replace our naive  $c$ -sum algorithm by more advanced time-memory techniques like Schroeppe-Shamir [34] and its generalization, Dissection [11], to reduce the classic running time. We call the resulting algorithm DISSECTION-BKW. To give some illustrative results, with the Schroeppe-Shamir technique DISSECTION-BKW achieves exponents

$$\log T = \frac{4}{3} \frac{k}{\log k} \quad , \quad \log M = \frac{2}{3} \frac{k}{\log k}$$

(see the blue diamond at  $\log M = \frac{2}{3} \frac{k}{\log k}$  in Figure 1). Using 7-Dissection, DISSECTION-BKW achieves exponents

$$\log T = 1.87 \frac{k}{\log k} \quad , \quad \log M = 0.47 \frac{k}{\log k}$$

(see the blue diamond at  $\log M \approx 0.47 \frac{k}{\log k}$  in Figure 1).

**Theorem 1.2 (Dissection BKW Trade-Off, informal).** *Let  $c \in \mathbb{N}$  be sufficiently large. The LPN problem in dimension  $k$  can be solved in Time  $T$  and space  $M$  where*

$$\log T = \left(1 - \sqrt{\frac{2}{c}}\right) \cdot \log c \cdot \frac{k}{\log k} \quad , \quad \log M = \frac{\log c}{c-1} \cdot \frac{k}{\log k} \quad .$$

Hence, in comparison to Theorem 1.1 Dissection mitigates the price we pay for saving a factor of  $\frac{\log c}{c-1}$  in memory from 1 down to  $\left(1 - \sqrt{2/c}\right)$ .

The trade-off is depicted by the blue diamonds in Figure 1. Interestingly, when classically employing Schroeppe-Shamir we are on par (see point  $(\frac{2}{3}, \frac{4}{3})$  in Figure 1) with the complexities from the quantum trade-off as Schroeppe-Shamir allows for a square-root gain in the running time; the same as using a Grover-search in a quantum algorithm.

**TAILORED DISSECTION.** Eventually, we introduce a new slight generalization of the Dissection technique that we call *tailored Dissection*. It allows us to achieve a piece-wise continuous trade-off (black line segments depicted in Figure 1) covering the sub-exponential memory regime entirely.

**OPEN PROBLEM.** There are quantum versions of Schroeppe-Shamir/Dissection [23] faster than their classic counterpart when finding a single solution is satisfactory. However, their advantage over Schroeppe-Shamir/Dissection seems to evaporate if one is required to find many solutions as in our BKW case.

## 2 Preliminaries

### 2.1 Notation

For  $a \leq b \in \mathbb{N}$  let  $[a, b] := \{a, a+1, \dots, b\}$  and  $[a] := [1, a]$ . For a set  $S$  and  $s \leq |S|$  let  $\binom{S}{s}$  denote the set of all size- $s$  subsets of  $S$ . A *list*  $L = (l_1, \dots, l_i)$  is an element  $L \in S^i$  and is of length  $|L| = i$ . We let

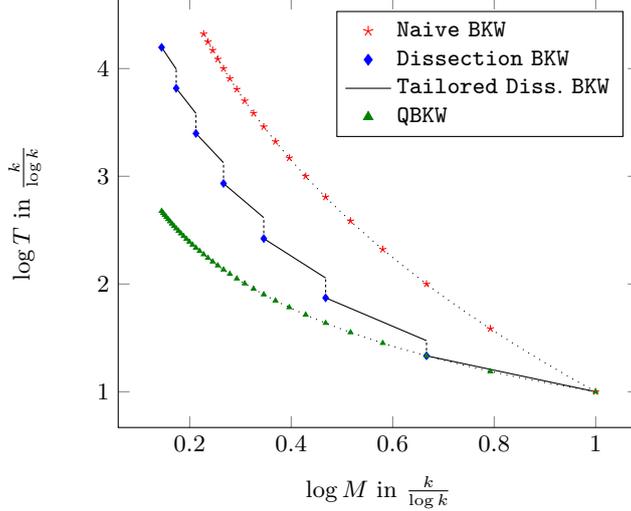


Fig. 1: Illustration of our BKW trade-offs. Instantiations exist exactly for marks as well as everywhere on solid lines. Naive BKW trade-off in red stars (see Theorem 1.1), QBKW trade-off in green triangles and Dissection-BKW trade-offs in blue diamonds and solid black line segments (see Theorem 1.2).

$\emptyset$  denote the empty list. For two lists  $L_1, L_2$  we write  $L_1 \subseteq L_2$  if all elements from  $L_1$  are contained in  $L_2$  at least as often as in  $L_1$ . We write shortly  $l \in L_2$  for  $(l) \subseteq L_2$ . For lists  $L_1 = (l_1, \dots, l_i)$  and  $L_2 = (l_{i+1}, \dots, l_j)$  we let  $L_1 \cup L_2 := (l_1, \dots, l_i, l_{i+1}, \dots, l_j)$ . Logarithms are always base 2.

For  $v \in \mathbb{F}_2^a$  and  $b \leq a$  we denote the last  $b$  coordinates of  $v$  by  $\text{low}_b(v)$ .  $\mathbf{u}_i$  denotes the  $i^{\text{th}}$  unit vector.  $0^b$  denotes the zero vector of dimension  $b$ .

By  $\mathcal{U}_M$  we denote the uniform distribution on a finite set  $M$ , by  $\text{Ber}_p$  we denote the Bernoulli distribution, i.e.,  $X \sim \text{Ber}_p$  means that  $\Pr[X = 1] = 1 - \Pr[X = 0] = p$ . The sum  $X$  of  $n$  independent random variables  $X_1, \dots, X_n \stackrel{iid}{\sim} \text{Ber}_p$  is binomial distributed with parameters  $n$  and  $p$ , denoted by  $X \sim \text{Bin}_{n,p}$ . A probability  $p(k)$  is called *overwhelming* in  $k$ , if  $1 - p(k)$  is negligible in  $k$ . We denote deterministic assignments in algorithms by  $\leftarrow$ .

**Theorem 2.1 (Chernoff Bound, [30]).** *Let  $X \sim \text{Bin}_{n,p}$ . Then*

$$\Pr[X \leq (1 - r)np] \leq \exp\left(-\frac{1}{2}r^2np\right) \quad \text{for any } r \in [0, 1]. \quad (1)$$

## 2.2 The LPN Problem

**Definition 2.1 ((Search) LPN Problem).** *Let  $k \in \mathbb{N}$ ,  $\mathbf{s} \in \mathbb{F}_2^k$  and  $p \in [0, \frac{1}{2})$  be a constant. Let **Sample** denote an oracle that, when queried, samples  $\mathbf{a} \sim \mathcal{U}_{\mathbb{F}_2^k}$ ,  $e \sim \text{Ber}_p$  and outputs a sample of the form  $(\mathbf{a}, b) := (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$ . The  $\text{LPN}_k$  problem consists of recovering  $\mathbf{s}$  given access to **Sample**. In the following we call  $k$  the dimension,  $\mathbf{s}$  the secret,  $p$  the error rate,  $b$  the label of  $\mathbf{a}$  and  $e$  the noise.*

**Brute-Force Sampling Algorithm.** A straight-forward way to recover the first bit  $s_1$  of the secret  $\mathbf{s}$  is to query **Sample** until we obtain a sample of the form  $(\mathbf{u}_1, b)$ . Then  $b = \langle \mathbf{u}_1, \mathbf{s} \rangle + e = s_1 + e$ . Hence,  $\Pr[s_1 = b] = 1 - p > \frac{1}{2}$ . However, as **Sample** draws  $\mathbf{a}$  uniformly from  $\mathbb{F}_2^k$ , we expect to have to query the oracle  $2^k$  times to have  $\mathbf{a} = \mathbf{u}_1$ .

Further, to boost the confidence in recovering  $s_1$  correctly from merely  $1 - p$  to being overwhelming (in  $k$ ) one may collect many samples  $(\mathbf{u}_1, b_i)$  and decide on  $s_1$  by majority vote, whose error is bounded in the following lemma.

**Lemma 2.1 (Majority Vote).** *Let  $q > \frac{1}{2}$  and  $X_1, \dots, X_n \sim \text{Ber}_q$  independently. Then*

$$\Pr\left[\sum_{i=1}^n X_i > \frac{n}{2}\right] \geq 1 - \exp\left(-\frac{n}{2q}\left(q - \frac{1}{2}\right)^2\right).$$

*Proof.* Since  $X := \sum_{i=1}^n X_i \sim \text{Bin}_{n,q}$ , using Theorem 2.1 with  $r = 1 - \frac{1}{2q}$  gives

$$\Pr \left[ X > \frac{n}{2} \right] \geq 1 - \exp \left( -\frac{1}{2} \left( 1 - \frac{1}{2q} \right)^2 nq \right) = 1 - \exp \left( -\frac{n}{2q} \left( q - \frac{1}{2} \right)^2 \right) .$$

□

**Corollary 2.1.** For  $n := \frac{2(1-p)k}{(\frac{1}{2}-p)^2}$  many samples, a majority vote recovers  $s_1$  correctly with probability at least  $1 - \exp(-k)$ .

*Proof.* Let us define  $X_i = 1$  iff  $b_i = s_1$ , which implies  $q = 1 - p$ . Lemma 2.1 yields the desired result. □

Therefore, for any constant error rate  $p$ , the majority vote requires only a linear number  $n = \mathcal{O}(k)$  of labels of the form  $(\mathbf{u}_1, b_i)$ . Clearly, we can recover the remaining bits  $s_j$ ,  $j = 2, \dots, k$  of the secret  $\mathbf{s}$  by querying `Sample` until we obtained sufficiently many samples with  $\mathbf{a} = \mathbf{u}_j$ . Overall, the probability that we recover all bits of  $\mathbf{s}$  correctly is at least  $(1 - \exp(-k))^k \geq 1 - k \cdot \exp(-k) = 1 - \text{negl}(k)$ .

### 2.3 Combining Samples

In [6], Blum, Kalai and Wasserman introduced the idea to construct  $\mathbf{a} = \mathbf{u}_1$  from a collection of  $N$  arbitrary samples rather than merely waiting for a sample where  $\mathbf{a} = \mathbf{u}_1$ . Their core idea is based on synthesizing a new sample from two existing ones  $(\mathbf{a}_1, b_1)$ ,  $(\mathbf{a}_2, b_2)$  via addition

$$(\mathbf{a}_1 \oplus \mathbf{a}_2, b_1 \oplus b_2) = (\mathbf{a}_1 \oplus \mathbf{a}_2, \langle \mathbf{a}_1 \oplus \mathbf{a}_2, \mathbf{s} \rangle \oplus e_1 \oplus e_2) .$$

In this synthesized sample, which we call a *2-sum of samples*, we have  $\mathbf{a}_1 \oplus \mathbf{a}_2 \sim \mathcal{U}_{\mathbb{F}_2^k}$  and  $e_1 \oplus e_2 \sim \text{Ber}_{p'}$  where  $p' = \frac{1}{2} - \frac{1}{2}(1-2p)^2 > p$  according to the following Piling-Up lemma.

**Lemma 2.2 (Piling-Up Lemma [14]).** Let  $p \in [0, 1]$  and  $e_i \sim \text{Ber}_p$ ,  $i \in [n]$  be identically, independently distributed. Then

$$\bigoplus_{i=1}^n e_i \sim \text{Ber}_{\frac{1}{2} - \frac{1}{2}(1-2p)^n} .$$

Summing up two or more samples enables us to synthesize samples at the expense of an error rate approaching  $\frac{1}{2}$  exponentially fast in the number of summands.

## 3 The $c$ -Sum-Problem and its Application to BKW

### 3.1 A Generalized BKW Algorithm

While BKW repeatedly adds *pairs* of samples to zero out chunks of coordinates, we add  $c > 2$  samples to accomplish the same. Beneficially, as the number of size- $c$  subsets grows exponentially in  $c$ , this allows for a drastic reduction in the number of initially required samples (thus, memory as well) while still finding sufficiently many sums of  $c$  vectors adding up to zero on a block of coordinates.

We give our  $c$ -sum-BKW in Algorithm 1. For a *block-size*  $b$  and  $j \in [a]$  we refer to the coordinates  $[k - jb + 1, k - (j-1)b]$  as the  $j^{\text{th}}$  *stripe*. Essentially,  $c$ -sum-BKW consists of a **for**-loop (line 4) that generates zeros (resp. the first unit vector) on the  $j^{\text{th}}$  stripe for  $j \in [a-1]$  (resp. the  $a^{\text{th}}$  stripe). This step is repeated multiple times (see line 2) to obtain sufficiently many labels of  $\mathbf{u}_1$  samples in order to let a majority vote determine the first bit  $s_1$  of the secret  $\mathbf{s}$  with overwhelming probability. This process is repeated  $k$  times to recover all bits of  $\mathbf{s}$ .

For a list  $L$  as constructed in line 3,  $j \in [a]$  and  $t \in \mathbb{F}_2^b$  we let  $c\text{-sum}(L, j, t)$  denote an algorithm that outputs a new list  $L$  where the coordinates of each entry match  $t$  on the  $j^{\text{th}}$  stripe (see lines 5, 6). If  $L$  should shrink to size 0 throughout an execution, we abort and return a failure symbol (see lines 7, 8).

We point out that (essentially) the original BKW algorithm may be obtained by letting  $c$ -sum add pairs of vectors whose sum matches  $t$  on the  $j^{\text{th}}$  stripe to  $L$ .

Let us introduce the *c-sum-problem* lying at the heart of any algorithm that shall be used to instantiate  $c$ -sum. In short, given a list of vectors  $L \in (\mathbb{F}_2^b)^*$ , i.e., a stripe from the  $c$ -sum-BKW point of view, the  $c$ -sum-problem asks to collect sums of  $c$  vectors that add up to some target  $t \in \mathbb{F}_2^b$ .

**Input:** dimension  $k$ , error rate  $p$ ,  $\epsilon_a > 0$ ,  $N \geq 2^{\frac{b+c \log c+1}{c-1}}$ , access to **Sample**

**Output:**  $\mathbf{s} \in \mathbb{F}_2^k$

1:  $a := \frac{\log k}{(1+\epsilon_a) \log c}$ ,  $b := \frac{k}{a}$ ,  $n := \frac{8(1-p)k}{(1-2p)^2 c^\alpha}$

2: **for**  $i \leftarrow 1, \dots, n$  **do**

3:     Query  $N$  samples from **Sample** and save them in  $L$ .

4:     **for**  $j \leftarrow 1, \dots, a-1$  **do**

5:          $L \leftarrow c\text{-sum}(L, j, 0^b)$

6:      $L \leftarrow c\text{-sum}(L, a, \mathbf{u}_1)$

7:     **if**  $L = \emptyset$  **then**

8:         **return**  $\perp$

9:     Pick  $(\mathbf{u}_1, b_i)$  uniformly from  $L$ .

10:  $s_1 \leftarrow \text{majorityvote}(b_1, \dots, b_n)$

11: Determine  $s_2, \dots, s_k$  the same way.

12: **return**  $\mathbf{s} = s_1 \dots s_k$

---

**Definition 3.1 (The  $c$ -Sum-Problem ( $c$ -SP)).** Let  $b, c, N \in \mathbb{N}$  with  $c \geq 2$ . Let  $L := (l_1, \dots, l_N)$  be a list where  $l_i \sim \mathcal{U}_{\mathbb{F}_2^b}$  for all  $i$  and let  $t \in \mathbb{F}_2^b$  be a target. A single-solution of the  $c$ -SP $_b$  is a size- $c$  set  $\mathcal{L} \subseteq \binom{[N]}{c}$  such that

$$\bigoplus_{j \in \mathcal{L}} l_j = t .$$

A solution is a set of at least  $N$  distinct single-solutions. The  $c$ -sum-problem  $c$ -SP $_b$  consists of finding a solution when given  $L, t$  while  $c$  is usually fixed in advance. We refer to  $(L, t, c)$  as an instance of the  $c$ -SP $_b$ , concisely  $(L, t)$  if  $c$  is clear from the context.

Note that by definition a solution to the  $c$ -sum-problem  $c$ -SP $_b$  consists of at least  $N$  single-solutions. These may again be combined into a new list of size (at least)  $N$ . Thus, we may apply a  $c$ -SP $_b$  solving algorithm on different  $b$ -bit stripes of a list. Further, the list does not shrink if a solution exists in each iteration.

Obviously, a solution should exist whenever the list size  $N$  is large enough, since then sufficiently many  $c$ -sums add up to some given target  $t$ . In the following, we show that the lower bound for the list-size  $N$  from Algorithm 1 guarantees the existence of such a solution with overwhelming probability under the following heuristic assumption that is regularly used in the analysis of BKW-type algorithms [36,8,7].

**Independence Heuristic.** Obviously,  $c$ -sums  $\bigoplus_{j \in \mathcal{L}} l_j$  are stochastically dependent for  $\mathcal{L} \subseteq [N]$  with  $|\mathcal{L}| = c$ . However, their dependency should only mildly affect the runtime of an iterative collision search as in BKW-type algorithms. For instance, it has been shown in [10] that the dependence between 2-sums  $\bigoplus_{j \in \mathcal{L}} l_j$  merely affects the overall runtime exponent by an  $o(1)$ -term. We heuristically assume that this also holds for  $c > 2$ , and therefore treat (iterative)  $c$ -sums as independent in our run time analyses.

We provide various experiments in Section 7.1 that support the Independence Heuristic.

For Algorithm 1, we need the following lemma only for the special case  $\alpha = 1$ . However, our Tailored Dissection approach in Section 5.3 also requires  $\alpha > 1$ .

**Lemma 3.1.** Let  $(L, t)$  be a  $c$ -SP $_b$  instance with

$$|L| = N^\alpha, \text{ where } N = 2^{\frac{b+c \log c+1}{c-1}} \text{ and } \alpha \geq 1 .$$

Under the Independence Heuristic,  $(L, t)$  has at least  $N^\alpha$  single-solutions with probability  $1 - \exp(-N/4)$ .

*Proof.* For every  $\mathcal{L} \subseteq [N]$  with  $|\mathcal{L}| = c$  define an indicator variable that takes value  $X_{\mathcal{L}} = 1$  iff  $\bigoplus_{j \in \mathcal{L}} l_j = t$ .

Let  $X = \sum_{\mathcal{L}} X_{\mathcal{L}}$  be the number of single-solutions to the  $c$ -SP $_b$ . Under the Independence Heuristic, the  $X_{\mathcal{L}}$  can be analyzed as if independent, thus  $X \sim \text{Bin}_{\binom{N^\alpha}{c}, 2^{-b}}$  is binomially distributed. Hence,

$$\mathbb{E}[X] = \binom{N^\alpha}{c} \cdot 2^{-b} \geq \left(\frac{N^\alpha}{c}\right)^c \cdot 2^{-b} . \quad (2)$$

Since  $\log N = \frac{b+c \log c+1}{c-1}$ , we obtain

$$\begin{aligned}
\log \mathbb{E}[X] &\geq c(\alpha \log N - \log c) - b \geq \alpha (c(\log N - \log c) - b) \\
&= \alpha \left( c \left( \frac{b + c \log c + 1 - (c-1) \log c}{c-1} \right) - b \right) \\
&= \alpha \cdot \frac{cb + c \log c + c - (c-1)b}{c-1} = \alpha \cdot \frac{b + c \log c + 1 + c - 1}{c-1} \\
&= \alpha(\log N + 1) \geq \alpha \log N + 1.
\end{aligned} \tag{3}$$

Thus, our choice of  $N$  guarantees  $\mathbb{E}[X] \geq 2N^\alpha$ . We can upper-bound the probability that the  $c$ -sum-problem has less than  $N^\alpha$  single-solutions solution using our Chernoff bound from Theorem 2.1 as

$$\Pr[X < N^\alpha] \leq \Pr\left[X < \frac{1}{2}\mathbb{E}[X]\right] \leq \exp\left(-\frac{\mathbb{E}[X]}{8}\right) \leq \exp\left(-\frac{N}{4}\right).$$

□

**Observation 3.1.** Any algorithm solving the  $c$ -sum-problem may be used to replace **c-sum** in lines 5 and 6 of **c-sum-BKW** (Algorithm 1) by

- 1) Projecting all entries of  $L$  onto their  $j^{\text{th}}$  stripe to obtain a list  $L^j$ .
- 2) Solving the  $c$ -sum-problem instance  $(L^j, t)$  to obtain a solution  $S$ .
- 3) Summing up the entries of  $L$  as specified in  $S$ .

Thereby step 2) is clearly dominating.

As a consequence of Observation 3.1 we do not distinguish between **c-sum** and an algorithm solving the  $c$ -sum-problem.

We now give Theorem 3.2 stating that **c-sum-BKW** inherits its complexities from **c-sum**. As a result, we may focus on solving the  $c$ -sum-problem in the remainder of this work.

**Theorem 3.2 (Correctness and Complexities of c-sum-BKW).** *Let **c-sum** denote an algorithm solving the  $c$ -SP $_b$  in expected time  $T_{c,N}$  and expected memory  $M_{c,N}$  with overwhelming success probability, where  $N \geq 2^{\frac{b+c \log c+1}{c-1}}$ . Under the Independence Heuristic **c-sum-BKW** solves the LPN $_k$  problem with overwhelming probability in time  $T$ , memory  $M$ , where*

$$T = (T_{c,N})^{1+o(1)}, \quad M = (M_{c,N})^{1+o(1)},$$

using  $N^{1+o(1)}$  samples.

*Proof.* Let  $a = \frac{\log k}{(1+\varepsilon_\alpha) \log c}$  and  $b = \frac{k}{a}$  as in Algorithm 1. Consider one iteration of the **for**-loop in line 2. As stated in Lemma 3.1 there exists a solution to the  $c$ -SP $_b$  instance (implicitly defined via the  $j^{\text{th}}$  stripe of  $L$  and target  $0^b$  (resp.  $\mathbf{u}_1$ )) with probability at least  $1 - \exp(-\frac{N}{4})$ . Hence, the probability that there is a solution to the instance in each iteration is greater than

$$\left(1 - \exp\left(-\frac{N}{4}\right)\right)^{an} \geq 1 - an \cdot \exp\left(-\frac{N}{4}\right).$$

We now analyze the probability that a single bit of the secret gets recovered correctly. Since we build  $c$ -sums iteratively  $a$  times in lines 4-6, eventually we obtain vectors being a sum of at most  $c^a$  samples, having an error of  $\frac{1}{2} - \frac{1}{2}(1-2p)^{c^a}$  according to Lemma 2.2.

Note that the labels  $b_1, \dots, b_n$  collected in line 10 are stochastically independent as each  $b_i$  is obtained from freshly drawn samples. Now Corollary 2.1 yields that  $n := \frac{8(1-p)k}{(1-2p)^{2c^a}}$  samples are sufficient for the majority vote to determine a bit of  $\mathbf{s}$  correctly with error probability at most  $\exp(-k)$ . Using the Union Bound, one bit of the secret  $\mathbf{s}$  gets successfully recovered with probability at least

$$1 - an \cdot \exp\left(-\frac{N}{4}\right) - \exp(-k).$$

As we have to correctly recover all, i.e.,  $k$  bits of  $\mathbf{s}$ , the overall success probability of **c-sum-BKW** is at least

$$\left(1 - 2an \cdot \exp\left(-\frac{N}{4}\right) - \exp(-k)\right)^k \geq 1 - 2ank \cdot \exp\left(-\frac{N}{4}\right) - k \exp(-k).$$

Let us look at the term  $2ank \cdot \exp\left(-\frac{N}{4}\right)$ . Since  $n = \tilde{O}(2^{\kappa_p e^a})$  for some constant  $\kappa_p$ , we have for constant  $c$

$$2ank = \tilde{O}(n) = 2^{\mathcal{O}(k^{\epsilon'})} \text{ with } \epsilon' < 1, \text{ whereas } N = 2^{\Theta(b)} = 2^{\Theta(\frac{k}{a})} = 2^{\Theta(\frac{k}{\log k})}. \quad (4)$$

Thus, the factor  $\exp\left(-\frac{N}{4}\right)$  clearly dominates and makes the overall success probability overwhelming in  $k$ .

Let us now analyze the time and memory complexity of **c-sum-BKW**. Since **c-sum** has only *expected* time/memory complexity, this expectation will be inherited to **c-sum-BKW**. We later show how to remove the expectation from **c-sum-BKW**'s complexities by a standard technique.

Let us start with the running time of **c-sum-BKW**, where we ignore the negligible overhead of iterations of line 2 caused by failures in the **c-sum**-algorithm. Clearly,  $T_{c,N} \geq N$ . Hence, one iteration of the **for**-loop can be carried out in time  $\tilde{O}(\max\{N, a \cdot T_{c,N}\}) = \tilde{O}(T_{c,N})$ . Thus, for recovering the whole secret we get a running time of  $\tilde{O}(n \cdot T_{c,N})$ . From Equation (4) we already know that  $N$  dominates  $n$ , which implies that  $T_{c,N}$  dominates  $n$ . More precisely, we have  $n \cdot T_{c,N} = (T_{c,N})^{1+o(1)}$ . Hence, we can also express the overall *expected* running time as  $(T_{c,N})^{1+o(1)}$ .

The memory consumption is dominated by **c-sum**, which gives us in total *expected*  $\tilde{O}(M_{c,N}) = (M_{c,N})^{1+o(1)}$ . The sample complexity of **c-sum-BKW** is  $\tilde{O}(knN) = \tilde{O}(nN) = N^{1+o(1)}$ .

It remains to remove the expectations from the complexity statements for time/memory, while keeping the success probability overwhelming. We run **c-sum-BKW**  $k$  times aborting each run if it exceeds its expected running time or memory by a factor of 4. A standard Markov bound then shows that this modified algorithm fails to provide a solution with probability at most  $2^{-k}$ . For details see Appendix A.  $\square$

In the following section, we discuss an algorithm to naively solve the  $c$ -sum-problem leading to a first trade-off in the subexponential time/memory regime that also allows for a quantum speedup.

## 4 First Time-Memory Trade-Offs for BKW

### 4.1 A Classic Time-Memory Trade-Off

A straight-forward algorithm to solve an instance of the  $c$ -sum-problem is to compute all sums of  $c$  elements from  $L$ . For  $N = |L|$  this approach takes time  $\mathcal{O}(N^c)$  while it requires memory  $\mathcal{O}(N)$ .

With little effort, we can do slightly better: Let  $L$  be sorted. Now let us brute-force all  $(c-1)$ -sums from  $L$ , add  $t$ , and check whether the result appears in  $L$ . This gives us by construction a  $c$ -sum that matches  $t$ .

The details of this **c-sum-naive** approach are given in Algorithm 2. Notice that whenever we call **c-sum-naive**, we should first sort the input list  $L$ , which can be done in additional time  $\tilde{O}(N)$ .

---

#### Algorithm 2 **c-sum-naive**( $L, t$ )

$\triangleright c \in \mathbb{N}$

**Input:** Sorted list  $L = (v_1, \dots, v_N) \in (\mathbb{F}_2^b)^N$ , target  $t \in \mathbb{F}_2^b$

**Output:**  $S \subseteq \binom{[N]}{c}$  or  $\perp$

```

1: for all  $\mathcal{V} = \{i_1, \dots, i_{c-1}\} \subseteq [N]$  do
2:   for all  $i_c \in [N] \setminus \mathcal{V}$  satisfying  $v_{i_c} = t \oplus (\oplus_{i \in \mathcal{V}} v_i)$  do
3:      $S \leftarrow S \cup \{\{i_1, \dots, i_c\}\}$ 
4:   if  $|S| = N$  then
5:     return  $S$ 
6: return  $\perp$ 

```

---

The following lemma shows correctness and the complexities of Algorithm 2.

**Lemma 4.1.** *c-sum-naive finds a solution of the c-SP<sub>b</sub> in time  $\tilde{\mathcal{O}}(N^{c-1})$  and memory  $\tilde{\mathcal{O}}(N)$ .*

*Proof.* Assume there exists a solution to instance  $(L, c)$ . The **for**-loop in line 1 iterates over each  $(c-1)$ -sum of  $L$  and checks, if there is another vector in  $L$ , completing the  $(c-1)$ -sum to a  $c$ -sum with value  $t$ . If there is such an element, the respective single-solution is added to  $S$ . By iterating over each  $(c-1)$ -sum, the algorithm is able to find all single-solutions of  $(L, t)$  as for every single-solution it holds that  $\bigoplus_{j=1}^c v_{i_j} = t$ , which is equivalent to  $v_{i_c} = t \oplus (\bigoplus_{j=1}^{c-1} v_{i_j})$  and thus the single-solution is recovered when  $\mathcal{V}$  takes the value  $\{i_1, \dots, i_{c-1}\}$ . Hence, if a solution to the c-SP<sub>b</sub> exists, then it is found by Algorithm 2, showing correctness.

For the run time statement, line 1 performs  $\binom{N}{c-1} \leq N^{c-1}$  iterations. Each solution can be found in the sorted list  $L$  in time  $\mathcal{O}(\log N) = \tilde{\mathcal{O}}(1)$ . Since Algorithm 2 halts once  $N$  single-solutions have been added to  $S$ , this leaves us with a total running time of  $\tilde{\mathcal{O}}(N^{c-1})$  and memory consumption of  $\tilde{\mathcal{O}}(\max(N, |S|)) = \tilde{\mathcal{O}}(N)$ .  $\square$

Let us now replace in the c-sum-BKW algorithm the c-sum subroutine with Algorithm 2 c-sum-naive, and call the resulting algorithm c-sum-naive-BKW.

**Theorem 4.1 (Naive Trade-Off).** *Let  $c \in \mathbb{N}$ . For all  $\varepsilon > 0$  and sufficiently large  $k$ , under the Independence Heuristic c-sum-naive-BKW solves the LPN<sub>k</sub> problem with overwhelming success probability in time  $T = 2^{\vartheta(1+\varepsilon)}$ , using  $M = 2^{\mu(1+\varepsilon)}$  memory and samples, where*

$$\vartheta = \log c \cdot \frac{k}{\log k} \quad , \quad \mu = \frac{\log c}{c-1} \cdot \frac{k}{\log k} \quad .$$

*Proof.* Let  $N := 2^{\frac{b+c \cdot \log c + 1}{c-1}}$ . According to Lemma 4.1 c-sum-naive is correct and we can apply Theorem 3.2 to conclude that c-sum-naive-BKW solves LPN with overwhelming success probability.

Further Lemma 4.1 shows that c-sum-naive runs in time

$$T_{c,N} = \tilde{\mathcal{O}}(N^{c-1}) = \tilde{\mathcal{O}}(2^{b+c \log c + 1}) = \tilde{\mathcal{O}}(2^b) \text{ for constant } c.$$

Thus, by Theorem 3.2 c-sum-naive-BKW runs in time  $T = 2^{b(1+o(1))}$ . Since c-sum-BKW operates on stripes of width  $b = \log c \cdot \frac{k(1+\varepsilon_a)}{\log k}$  (see the definition in Algorithm 1), we obtain the claimed complexity  $T = 2^{\vartheta(1+\varepsilon_a+o(1))} = 2^{\vartheta(1+\varepsilon)}$  for every  $\varepsilon > \varepsilon_a$  and sufficiently large  $k$ .

Since c-sum-naive requires memory  $M_{c,N} = \tilde{\mathcal{O}}(N)$ , by Theorem 3.2 the memory complexity of c-sum-BKW is (for constant  $c$ )

$$M = (M_{c,N})^{1+o(1)} = N^{1+o(1)} = (2^{\frac{b}{c-1}})^{1+o(1)} = (2^{\frac{\log c}{c-1} \cdot \frac{k}{\log k}})^{1+o(1)}.$$

The sample complexity of c-sum-BKW is  $N^{1+o(1)}$  (see Theorem 3.2) and therefore identical to  $M$ .  $\square$

Figure 2 depicts the time-memory trade-off obtained in Theorem 4.1. Thereby, for increasing  $c$  the memory consumption decreases by  $\frac{\log c}{c-1}$  whereas the running time increases only by  $\log c$ .

## 4.2 A Quantum Time-Memory Trade-Off

Grover's algorithm [15] identifies a marked element in an *unsorted* database  $D$  in time  $\mathcal{O}(\sqrt{|D|})$  with overwhelming probability. A matching lower bound  $\Omega(\sqrt{|D|})$  by Donataru and Høyer [12] shows that Grover's algorithm is optimal. We use a modification of Grover's algorithm due to [9], denoted **Grover**, that applies even in case the number of marked elements is unknown.

**Theorem 4.2 (Grover Algorithm [9]).** *Let  $f: D \rightarrow \{0, 1\}$  be a function with non-empty support. Then **Grover** outputs with overwhelming probability a uniformly random preimage of 1, making  $q$  queries to  $f$ , where*

$$q = \tilde{\mathcal{O}} \left( \sqrt{\frac{|D|}{|f^{-1}(1)|}} \right) \quad .$$

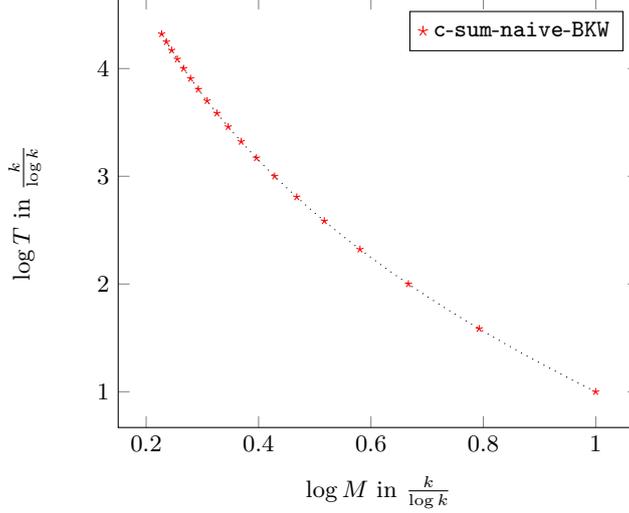


Fig. 2: Time-Memory trade-off as given in Theorem 4.1. Stars from right to left corresponding to  $c = 2, 3, \dots$

We use Grover's algorithm to speed up our naive approach to solving the  $c$ -sum-problem. While we previously brute-forced all  $(c-1)$ -sums in list  $L$  and checked if there is a fitting  $c^{\text{th}}$  entry in  $L$ , we now employ a Grover-search to immediately obtain  $(c-1)$ -sums for which there exists a suitable  $c^{\text{th}}$  element in  $L$ . Let us define the Grover function  $f_t$  as

$$f_t: \binom{[L]}{c-1} \rightarrow \{0, 1\}, \mathcal{V} \mapsto \begin{cases} 1 & \exists i_c \in [L] \setminus \mathcal{V}: \sum_{j=1}^{c-1} l_{ij} = l_{i_c} + t \\ 0 & \text{else} \end{cases} .$$

Given some  $\mathcal{V} \in f_t^{-1}(1)$  we can recover all  $i_c$  such that  $\mathcal{V} \cup \{i_c\}$  is a single-solution of instance  $(L, t)$  in time  $\tilde{\mathcal{O}}(\log |L|)$  if  $L$  is sorted.

---

**Algorithm 3** Q- $c$ -sum( $L, t$ )

$\triangleright c \in \mathbb{N}$

**Input:** Sorted list  $L = (v_1, \dots, v_N) \in (\mathbb{F}_2^b)^N$ , target  $t \in \mathbb{F}_2^b$

**Output:**  $S \subseteq \binom{[N]}{c}$  or  $\perp$

- 1: **repeat**  $\tilde{\mathcal{O}}(N)$  times
  - 2:      $\mathcal{V} = (i_1, \dots, i_{c-1}) \leftarrow \text{Grover}^{f_t}$
  - 3:     **for all**  $i_c \in [N] \setminus \mathcal{V}$  satisfying  $v_{i_c} = t \oplus (\oplus_{i \in \mathcal{V}} v_i)$  **do**
  - 4:          $S \leftarrow S \cup \{\{i_1, \dots, i_c\}\}$
  - 5:         **if**  $|S| = N$  **then**
  - 6:             **return**  $S$
  - 7: **return**  $\perp$
- 

Lines 3-7 of Q- $c$ -sum and c-sum-naive are identical. We merely replaced the brute-force search (line 1 in Algorithm 2) by a Grover-search (lines 1, 2 in Algorithm 3).

**Lemma 4.2.** Q- $c$ -sum solves the  $c$ -SP $_b$  with overwhelming probability in time  $\tilde{\mathcal{O}}(N^{c/2})$  and memory  $\tilde{\mathcal{O}}(N)$ .

*Proof.* Assume there exists a solution to instance  $(L, c)$ . Then  $f_t^{-1}(1) \geq |L| = N$ . As stated in Theorem 4.2 Grover queries  $f_t$

$$\tilde{\mathcal{O}}\left(\sqrt{\frac{|D|}{|f^{-1}(1)|}}\right) = \tilde{\mathcal{O}}\left(\sqrt{\frac{N^{c-1}}{N}}\right) = \tilde{\mathcal{O}}(N^{\frac{c}{2}-1}) .$$

times, since  $f_t$  has domain  $D = \binom{[L]}{c-1}$  of size  $|D| = \binom{[L]}{c-1} = \binom{N}{c-1}$ . Observe that  $f_t$  can be computed in time  $\tilde{O}(\log N) = \tilde{O}(1)$  if  $L$  is sorted. Hence, one iteration if the **for**-loop in line 1 requires time  $\tilde{O}(N^{\frac{c}{2}-1})$ .

It follows that **Q-c-sum** runs in time  $\tilde{O}(N^{\frac{c}{2}})$  requiring memory  $\tilde{O}(N)$ .

It remains to show that **Q-c-sum** recovers an existing solution with overwhelming probability. As **Grover** returns a uniform element from  $f_t^{-1}(1)$ , we are given an instance of the *coupon collectors problem* [31] where we have an overwhelming probability to have seen  $N$  distinct preimages after obtaining  $N \ln N = \tilde{O}(N)$  preimages via **Grover**.  $\square$

Let **QBKW** denote algorithm **c-sum-BKW** where **c-sum** is instantiated using **Q-c-sum**.

**Theorem 4.3.** *Let  $c \in \mathbb{N}$ . For all  $\varepsilon > 0$  and sufficiently large  $k$ , under the Independence Heuristic **QBKW** solves the  $\text{LPN}_k$  problem with overwhelming success probability in time  $T = 2^{\vartheta(1+\varepsilon)}$ , using  $M = 2^{\mu(1+\varepsilon)}$  memory and samples, where*

$$\vartheta = \frac{c}{2 \cdot (c-1)} \cdot \log c \cdot \frac{k}{\log k}, \quad \mu = \frac{\log c}{c-1} \cdot \frac{k}{\log k}.$$

*Proof.* The proof proceeds along the lines of the proof of Theorem 4.1.

The trade-off from Theorem 4.3 is depicted in Figure 3 on Page 4 by triangles.

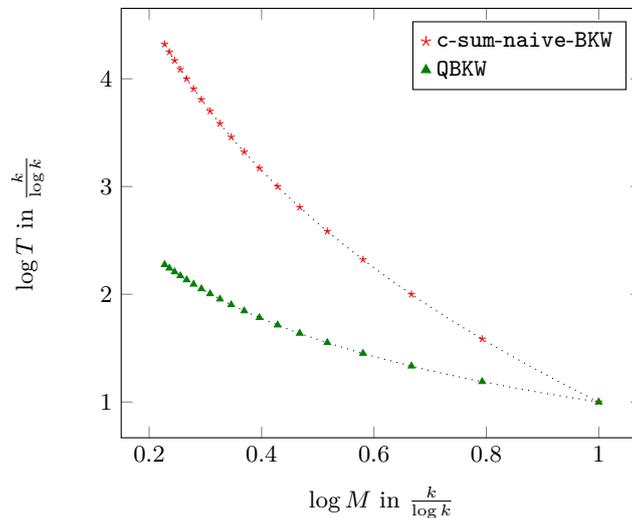


Fig. 3: Time-Memory trade-off of **QBKW** (see Theorem 4.3) halving the running time exponent compared to **c-sum-naive-BKW** (see Theorem 4.1). Marks from right to left corresponding to  $c = 2, 3, \dots$

## 5 Time-Memory Trade-Offs for BKW via Dissection

While a naive approach already led to a first time-memory trade-off, a meet-in-the-middle approach for solving the  $c$ -sum-problem prohibits a trade-off, as we explain in Section 5.1. As a consequence, we resort to the more advanced Dissection technique [11,34] in Section 5.2. However, as Dissection merely leads to instantiations for a rather sparse choice of available memory, we give a slight generalization of Dissection *tailored* to any given amount of memory. This allows for a trade-off covering the whole range of possible memory (see Section 5.3).

Let us define the specific structure of single-solutions that are recovered by all  $c$ -sum-problem algorithms in this section.

**Definition 5.1 (Equally-split (Single-)Solution).** *Let  $(L, t)$  be an  $c\text{-SP}_t$  instance. Partition  $L$  into lists  $L_1, \dots, L_c$  of size  $\frac{L}{c}$  each. A single-solution is equally-split (wrt.  $L_1, \dots, L_c$ ) if it corresponds to list elements  $l_1, \dots, l_c$  whereby  $l_i \in L_i$  for all  $i \in [c]$ . An equally-split solution is a collection of  $N$  equally-split single-solutions.*

**Lemma 5.1.** *Let  $(L, t)$  be a  $c$ -SP <sub>$b$</sub>  instance with*

$$|L| = N^\alpha, \text{ where } N = 2^{\frac{b+c \log c + 1}{c-1}} \text{ and } \alpha \geq 1 .$$

*Then, under the Independence Heuristic,  $(L, t)$  has at least  $N^\alpha$  equally-split single-solutions with probability  $1 - \exp(-N/4)$ .*

*Proof.* Let  $X$  be a random variable for the number of *equally-split* single-solutions. Then  $\mathbb{E}[X] = \left(\frac{N^\alpha}{c}\right)^c \cdot 2^{-b}$ . Hence, Equation (2) is satisfied, and the rest follows as in the proof of Lemma 3.1.  $\square$

### 5.1 Meet-in-the-Middle and beyond

Let  $(L, t)$  be an instance of the  $c$ -sum-problem. In a meet-in-the-middle approach one splits a  $c$ -sum  $t = v_{i_1} \oplus \dots \oplus v_{i_c}$  into two parts  $t = (v_{i_1} \oplus \dots \oplus v_{i_{\frac{c}{2}}}) \oplus (v_{i_{\frac{c}{2}+1}} \oplus \dots \oplus v_{i_c})$ . Let us define  $L_1 := (v_1, \dots, v_{\lfloor \frac{|L|}{2} \rfloor})$ ,  $L_2 := (v_{\lfloor \frac{|L|}{2} \rfloor + 1}, \dots, v_{|L|})$  and consider a single-solution corresponding to  $\frac{c}{2}$  elements from  $L_1$  and  $L_2$  each:

$$(v_{i_1}, \dots, v_{i_{\frac{c}{2}}}) \subseteq L_1 \text{ and } (v_{i_{\frac{c}{2}+1}}, \dots, v_{i_c}) \subseteq L_2 .$$

If we compute all  $\frac{c}{2}$ -sums of elements from  $L_1$  and save them in a new list  $L_1^{\frac{c}{2}}$ , then for each  $\frac{c}{2}$ -sum  $v$  of  $L_2$  we can check if  $w := v \oplus t \in L_1^{\frac{c}{2}}$ . If so,  $v$  and  $w$  form a single-solution since  $v \oplus w = t$ . Obviously, this approach has expected time and memory complexity  $\tilde{O}(\max(|L|, |L|^{\frac{c}{2}})) = \tilde{O}(|L|^{\frac{c}{2}})$  for  $c \geq 2$ . Yet, choosing  $c > 2$  only leads to worse complexities, time and memory-wise, while for  $c = 2$  the complexities for the meet-in-the-middle approach are as bad as for **c-sum-naive**.

**Schroeppe and Shamir's Meet-in-the-Middle.** We present a heuristic simplification of the Schroeppe-Shamir algorithm [34] due to Howgrave-Graham and Joux [21].

In a nutshell, the idea of Schroeppe and Shamir is to run a meet-in-the-middle attack but impose an artificial constraint  $\tau$  on the  $\frac{c}{2}$ -sums. This results in lists  $L_1^{\frac{c}{2}, \tau}$  that are significantly smaller than  $L_1^{\frac{c}{2}}$  in the original meet-in-the-middle approach. In order to find all single-solutions, one iterates  $\tau$  over its whole domain.  $L_1^{\frac{c}{2}, \tau}$  is in turn built from smaller lists as follows. Let  $t = v_{i_1} \oplus \dots \oplus v_{i_c}$  and write

$$t = \underbrace{v_{i_1} \oplus \dots \oplus v_{i_{\frac{c}{4}}}}_{\ell_{11}} \oplus \underbrace{v_{i_{\frac{c}{4}+1}} \oplus \dots \oplus v_{i_{\frac{c}{2}}}}_{\ell_{12}} \oplus \underbrace{v_{i_{\frac{c}{2}+1}} \oplus \dots \oplus v_{i_{\frac{3c}{4}}}}_{\ell_{21}} \oplus \underbrace{v_{i_{\frac{3c}{4}+1}} \oplus \dots \oplus v_{i_c}}_{\ell_{22}} .$$

Create four lists  $L_{11}^{\frac{c}{4}}, L_{12}^{\frac{c}{4}}, L_{21}^{\frac{c}{4}}, L_{22}^{\frac{c}{4}}$  containing all  $\frac{c}{4}$ -sums of elements from the first, second, third, and fourth quarter of  $L$ . Let  $\ell_{i,j}$  denote elements from  $L_{ij}^{\frac{c}{4}}$  for  $i, j = 1, 2$ . As a constraint we choose  $\text{low}_{\frac{c}{4} \log |L|}(\ell_{11} \oplus \ell_{12}) = \tau$  for some fixed  $\tau$ . Now we construct  $L_1^{\frac{c}{2}, \tau}$  from  $L_{11}^{\frac{c}{4}}$  and  $L_{12}^{\frac{c}{4}}$  using a meet-in-the-middle approach requiring expected time  $\tilde{O}(|L|^{\frac{c}{4}})$ . Similarly, we can compute all elements of list  $L_2^{\frac{c}{2}, t \oplus \tau}$  to obtain sums of  $\frac{c}{2}$  vectors ( $\frac{c}{4}$  from  $L_{21}^{\frac{c}{4}}$  and  $L_{22}^{\frac{c}{4}}$  each) matching  $t \oplus \tau$  on the last  $|\tau| = \frac{c}{4} \log |L|$  bits. Eventually, given the elements from  $L_2^{\frac{c}{2}, t \oplus \tau}$  and list  $L_1^{\frac{c}{2}, \tau}$  one can some recover single-solutions as before. Thereby list  $L_1^{\frac{c}{2}, \tau}$  is of expected size

$$\mathbb{E} \left[ \left| L_1^{\frac{c}{2}, \tau} \right| \right] = \left| L_{11}^{\frac{c}{4}} \right| \cdot \left| L_{12}^{\frac{c}{4}} \right| \cdot 2^{-|\tau|} = |L|^{2 \cdot \frac{c}{4} - \frac{c}{4}} = |L|^{\frac{c}{4}} .$$

We conclude that the expected memory consumption of the algorithm is given by  $\tilde{O}(\max\{|L|, |L|^{\frac{c}{4}}\})$ .

As we iterate over  $2^{|\tau|} = |L|^{\frac{c}{4}}$  choices of  $\tau$  requiring expected time  $\tilde{O}(|L|^{\frac{c}{4}})$  per iteration, the overall expected running time is given by  $\tilde{O}(|L|^{\frac{c}{2}})$  for all  $c \geq 4$ . Hence, for  $c = 4$  we obtain an algorithm as fast as the meet-in-the-middle approach, while consuming only expected memory  $\tilde{O}(|L|)$ .

Note that it is sufficient to store list  $L_1^{\frac{c}{2}, \tau}$  while all elements from  $L_2^{\frac{c}{2}, t \oplus \tau}$  may be computed and checked on-the-fly, i.e., without storing its elements.

The Schroeppe-Shamir algorithm is a special case of Dissection run on 4 lists in the subsequent section, which further exploits the asymmetry between storing lists and computing (memory-free) lists on the fly.

## 5.2 Dissection

Dissection can be considered as a memory-friendly member of the class of  $k$ -list algorithms [35]. A Dissection algorithm is given  $c$  lists  $L_c, \dots, L_1$  and a target  $t$ . For simplicity the algorithm merely sums list elements to obtain target  $t$  rather than outputting a single-solution explicitly. One could keep track of indices of those elements that sum to  $t$ , but for ease of notation we omit this. Instead, we employ some abstract “index recovering” procedure later.

**HIGH-LEVEL STRUCTURE OF DISSECTION.** A Dissection identifying sums of vectors adding up to some target  $t$  consists of the following steps

- 1) A loop iterates over an artificially introduced constraint  $\tau$ . For each  $\tau$ :
  - 1.1) A meet-in-the-middle approach is run to obtain a list  $L$  of sums from the first few lists that add up to  $\tau$  on a some bits. List  $L$  is kept in memory.
  - 1.2) Dissection is called recursively to find sums from the remaining lists that sum to  $t \oplus \tau$  on a some bits. These sums are passed to the parent call on-the-fly.
  - 1.3) For all sums passed from the recursive call in step 1.2) list  $L$  is searched to construct sums adding up to  $\tau$  on some more bits.

Before giving the fully-fledged Dissection algorithm, let us give an 11-Dissection example.

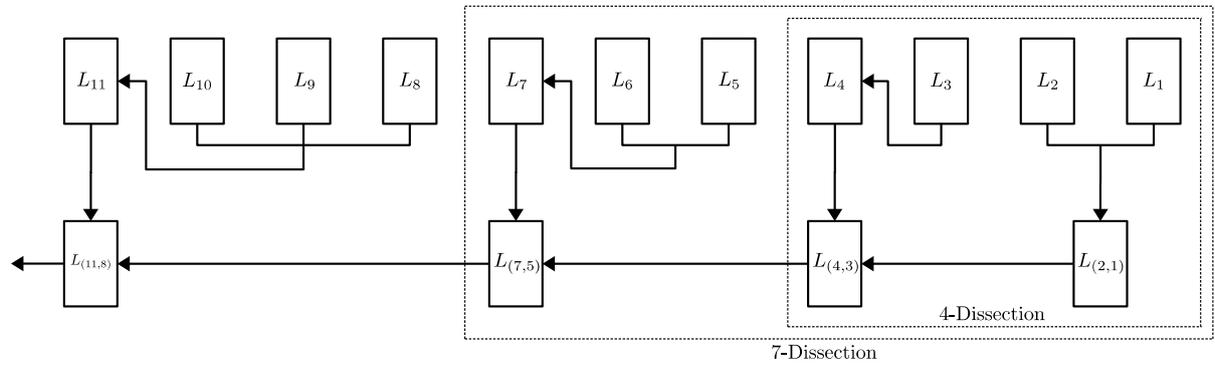


Fig. 4: Structure of an 11-Dissection on input lists  $L_{11}, \dots, L_1$ . Recursively called 7- and 4-Dissection enclosed in dashed boxes. Arrows entering a list from the right side indicate a check on-the-fly. Arrows leaving a list on the left side indicate that a found match is returned on-the-fly. Arrows entering a list from the top indicate that the list below is populated with entries from above and stored entirely.

*Example 5.1 (11-Dissection).* An 11-Dissection is run on lists  $L_{11}, \dots, L_1$  and some target  $t$ . It loops through an artificial constraint  $\tau_3$ . Within each iteration: A list  $L_{(11,8)}$  containing sums  $l_{11} \oplus \dots \oplus l_8$  consistent with constraint  $\tau_3$  where  $l_{11} \in L_{11}, \dots, l_8 \in L_8$  is computed and stored. Then, still within the first iteration of the loop, a 7-Dissection is run (see Figure 4) on lists  $L_7, \dots, L_1$  and a modified target  $t \oplus \tau_3$ . The 7-Dissection itself introduces a constraint  $\tau_2$  and stores a list  $L_{(7,5)}$  containing sums of elements from  $L_7, L_6, L_5$  fulfilling  $\tau_2$ . It recursively calls a 4-Dissection, i.e. Schroepel-Shamir (Section 5.1), on target  $t \oplus \tau_3 \oplus \tau_2$ . Internally, the 4-Dissection introduces another constraint  $\tau_1$ .

Whenever a partial sum is identified by the 4-Dissection, it is passed to the 7-Dissection on-the-fly while the 4-Dissection carries on. See the “chain” at the bottom in Figure 4 traversing lists from right to left. Once, the 7-Dissection receives a sum, it immediately checks for a match in list  $L_{(7,5)}$  and discards or returns it—in case of the latter—enriched by a matching element from  $L_{(7,5)}$  to the 11-Dissection and continues.

Whenever the 11-Dissection receives a sum from the 7-Dissection, it instantly checks for a match in list  $L_{(11,8)}$  to detect if a single-solution has been found.

**Definition 5.2 (Join Operator).** Let  $d \in \mathbb{N}$  and  $L_1, L_2 \in (\mathbb{F}_2^d)^*$  be lists. The join of  $L_1$  and  $L_2$  is defined as

$$L_1 \bowtie L_2 := (l_1 \oplus l_2 : l_1 \in L_1, l_2 \in L_2) .$$

For  $t \in \mathbb{F}_2^{\leq d}$  the join of  $L_1$  and  $L_2$  on  $t$  is defined as

$$L_1 \bowtie_t L_2 := (l_1 \oplus l_2 : l_1 \in L_1, l_2 \in L_2 \wedge \text{low}_{|t|}(l_1 \oplus l_2) = t) .$$

If  $L_2 = (l_2)$  we write  $L_1 \bowtie_t l_2$  instead of  $L_1 \bowtie_t (l_2)$ .

Clearly, computing elements contained in a sequence of joins can be implemented memory-friendly without having to explicitly compute intermediate lists.

**Definition 5.3 (The Magic Sequence [11]).** Let  $c_{-1} := 1$  and

$$c_i := c_{i-1} + i + 1 \tag{5}$$

for all  $i \in \mathbb{N} \cup \{0\}$ . The magic sequence is defined as  $\text{magic} := (c_i)_{i \in \mathbb{N} \geq 1}$ .

One easily verifies that, alternatively,

$$\text{magic} = \left( \frac{1}{2} \cdot (i^2 + 3i + 4) \right)_{i \in \mathbb{N} \geq 1} . \tag{6}$$

As Dissection formulated in the language of  $k$ -list algorithms might be of independent interest we deter from adding  $\mathbf{c}\text{-SP}_b$ -solving related details to the Dissection algorithm presented next, but rather introduce a simple wrapper algorithm (see Algorithm 5) to solve the  $c$ -sum-problem afterwards.

We define the class of Dissection algorithms recursively. Let  $L_2, L_1 \in (\mathbb{F}_2^d)^*$ ,  $d \in \mathbb{N}$  and  $t \in \mathbb{F}_2^{\leq d}$ . Then

$$\mathbf{c}_0\text{-Dissect}(L_2, L_1, t, \mathbf{inner}) := L_2 \bowtie_t L_1 . \tag{7}$$

Here, “**inner**” indicates that  $\mathbf{c}_0\text{-Dissect}$  was called recursively by another Dissection algorithm in contrast to an initial, explicit call to run Dissection that will be called with parameter “**outer**”.

We proceed to define  $\mathbf{c}_i\text{-Dissect}$  for  $i \geq 1$ . Lists are denoted by  $L$  and input-lists are numbered  $L_{c_i}$  down to  $L_1$  for  $c_i \in \text{magic}$ . As a reading aid, list and element indices keep track of input-lists they originated from: A list  $L_{(j,i)}$  is the output of a join of lists  $L_j \bowtie \dots \bowtie L_i$  for  $j > i$ . List elements are denoted  $l$ . We write  $l_{(j,i)}$  to indicate that  $l_{(j,i)}$  is a sum of elements  $l_\kappa \in L_\kappa$  for  $\kappa = j, \dots, i$ .

---

**Algorithm 4**  $\mathbf{c}_i\text{-Dissect}(L_{c_i}, \dots, L_1, t, \mathbf{pos})$  ▷  $\mathbf{c}_i \in \text{magic}$

---

**Input:** Lists  $L_{c_i}, \dots, L_1 \in (\mathbb{F}_2^b)^{2^\lambda}$  where  $\lambda \leq \frac{b}{i}$ , target  $t \in \mathbb{F}_2^b$ ,  $\mathbf{pos} \in \{\mathbf{outer}, \mathbf{inner}\}$

**Output:**  $S \subseteq \binom{[N]}{c_i}$  or  $\perp$

```

1: for all  $\tau_i \in \mathbb{F}_2^{i \cdot \lambda}$  do
2:    $L_{(c_i, c_{i-1}+1)} \leftarrow L_{c_i} \bowtie_{\tau_i} (L_{c_{i-1}} \bowtie \dots \bowtie L_{c_{i-1}+1})$ 
3:   for all  $l_{(c_{i-1}, 1)}$  passed from  $\mathbf{c}_{i-1}\text{-Dissect}(L_{c_{i-1}}, \dots, L_1, \text{low}_{i \cdot \lambda}(t) \oplus \tau_i, \mathbf{inner})$  do
4:     for all  $l_{(c_i, 1)} \in L_{(c_i, c_{i-1}+1)} \bowtie_t l_{(c_{i-1}, 1)}$  do
5:       if  $\mathbf{pos} = \mathbf{inner}$  then
6:         pass  $l_{(c_i, 1)}$  to  $\mathbf{c}_{i+1}\text{-Dissect}$ 
7:       else
8:          $S \leftarrow S \cup \{\mathbf{recover\ indices}(l_{(c_i, 1)})\}$ 

```

---

The optimality of this recursive structure is shown in [11]. We now establish the required properties of Algorithm 4 in a series of lemmata and provide detailed proofs that were merely sketched in [11].

**Lemma 5.2 (Correctness of  $\mathbf{c}_i\text{-Dissect}$ ).** For some fixed  $j_a$  let  $l_a := L_a(j_a)$  denote the  $j_a^{\text{th}}$  element of list  $L_a$ . When  $\mathbf{c}_i\text{-Dissect}(L_{c_i}, \dots, L_1, t, \mathbf{outer})$  halts, set  $S$  contains  $(j_{c_i}, \dots, j_1) \in [2^\lambda]^{c_i}$  if and only if  $\bigoplus_{a=1}^{c_i} l_a = t$ .

---

**Algorithm 5**  $c_i$ -sum-Dissect( $L, t$ ) $\triangleright c_i \in \text{magic}$ 

---

**Input:**  $L \in (\mathbb{F}_2^b)^N$ , target  $t \in \mathbb{F}_2^b$ **Output:**  $S \subseteq \binom{[N]}{c_i}$  or  $\perp$ 

- 1: Partition  $L$  into  $c_i$  lists  $L_{c_i}, \dots, L_1$  of size  $\frac{N}{c_i}$  each
  - 2:  $S \leftarrow c_i$ -Dissect( $L_{c_i}, \dots, L_1, t, \text{outer}$ )
  - 3: **if**  $|S| < N$  **then**
  - 4:     **return**  $\perp$
  - 5: **return**  $S$
- 

*Proof.* We prove the claim by induction over  $i \geq 1$ .

BASE CLAUSE  $i = 1$ . Assume  $(j_4, j_3, j_2, j_1) \in [2^\lambda]^4$  is a single-solution. Then for  $\tau := l_4 \oplus l_3$  we have

$$l_4 \oplus l_3 = \tau \quad \text{and} \quad l_2 \oplus l_1 = t \oplus \tau .$$

Thus, once the **for**-loop in line 1 reaches  $\tau_1 := \text{low}_\lambda(\tau)$ , sum  $l_4 \oplus l_3$  is added to  $L_{4,3}$  in line 2. Further, clearly  $l_2 \oplus l_1 \in L_2 \bowtie_{t_1 \oplus \tau_1} L_1$ . Hence, once the **for**-loop in line 3 considers  $l_2 \oplus l_1$ , the single-solution is recovered in line 4 and added to  $S$  in line 6.

Now, if  $(j_4, j_3, j_2, j_1)$  is not a single-solution we have

$$l_4 \oplus l_3 = \tau \Rightarrow l_2 \oplus l_1 \neq t \oplus \tau .$$

Thus, when the **for**-loop in line 1 considers  $\tau_1 = \text{low}_\lambda(l_4 \oplus l_3)$  partial single-solutions on  $\lambda$  bits may be found in lines 2 and 3, however, all of these are disregarded in line 6 when the join on (all)  $b$  bits is computed.

STEP  $i \rightarrow i + 1$ . Observe that  $(j_{c_{i+1}}, \dots, j_1) \in [2^\lambda]^{c_{i+1}}$  is a single-solution for target  $t$  if and only if for  $\tau := l_{c_{i+1}} \oplus \dots \oplus l_{c_i+1}$  we have

$$l_{c_i} \oplus \dots \oplus l_1 = t \oplus \tau .$$

As soon as the **for**-loop in line 1 reaches  $\tau_{i+1} = \text{low}_{(i+1)\lambda}(\tau)$ , sum  $l_{c_{i+1}} \oplus \dots \oplus l_{c_i+1}$  is added to list  $L_{(c_{i+1}, c_i+1)}$ . The claim now follows from observing that  $\text{low}_a(l_{c_i} \oplus \dots \oplus l_1) = \text{low}_a(t \oplus \tau)$  and the correctness of  $c_i$ -Dissect called in line 3. □

**Lemma 5.3 (Memory Consumption of  $c_i$ -Dissect).** *For all  $i \geq 1$  algorithm  $c_i$ -Dissect requires expected memory  $\tilde{O}(\max\{2^\lambda, \mathbb{E}[|S|]\})$ .*

*Proof.* Only lists  $L_{(c_i, c_{i-1}+1)}$  for all  $i$  from the join operation in line 2 are kept in memory. As the sum of uniform vectors is uniform, we expect an  $|\mathbb{F}_2^{i\lambda}|$ -fraction of  $|L_{c_i}| \cdot \dots \cdot |L_{c_{i-1}+1}|$  elements to be added to  $L_{(c_i, c_{i-1}+1)}$ . Hence, lists  $L_{(c_i, c_{i-1}+1)}$  are of expected size  $\tilde{O}(2^{\lambda \cdot (i+1) - i\lambda}) = \tilde{O}(2^\lambda)$ .

These lists themselves may be computed in memory  $\tilde{O}(2^\lambda)$  when the entries in  $L_{c_{i-1}} \bowtie \dots \bowtie L_{c_{i-1}+1}$  are computed on-the-fly and are disregarded or added to list  $L_{(c_i, c_{i-1}+1)}$ .

At the “bottom” of the recursion for  $i = 1$  in line 3, the join of lists  $L_2$  and  $L_1$  in line 3 (see Equation (7)) can be computed using memory  $\tilde{O}(2^\lambda)$ .

Further, all elements passed from a recursive call in line 3 are processed on-the-fly in line 4 and—still on-the-fly—passed to  $c_{i+1}$ -Dissect (line 5) requiring no additional memory, dropped or added to  $S$  (line 6) consuming memory  $\mathbb{E}[|S|]$  overall. The claim follows. □

**Lemma 5.4.** *Let  $i \geq 1$  and consider one iteration of the **for**-loop in line 1 within a call of  $c_i$ -Dissect( $\dots, \text{inner}$ ). Then, in total, expected  $\tilde{O}(2^{c_i - 2\lambda})$  elements are returned in line 5.*

*Proof.* We proceed by induction over  $i$ .

BASE CLAUSE  $i = 1$ . Fix any  $\tau_1 \in \mathbb{F}_2^\lambda$  in the **for**-loop in line 1. List  $L_{(4,3)}$  computed in line 2 is of expected size  $\tilde{O}(2^\lambda)$ . The **for**-loop in line 3 iterates over expected  $|L_2 \bowtie_{\text{low}_{i\lambda}(t) \oplus \tau_i} L_1| = \tilde{O}(2^\lambda)$  values of  $l_{(2,1)}$ . Note that, as we consider an **inner** call  $c_1$ -Dissect was given a target  $t$  of length  $2 \cdot \lambda$  and the join

computed in line 4 imposes an *additional* restriction on  $2 \cdot \lambda - \lambda = \lambda$  bits. Thus,  $L_{(4,3)} \bowtie_t l_{(2,1)}$  is of expected size  $\tilde{\mathcal{O}}(|L_{(4,3)}|)/2^\lambda = \tilde{\mathcal{O}}(2^0)$ . Iterating over all values of  $l_{(2,1)}$  we expect  $\tilde{\mathcal{O}}(2^{\lambda+0}) = \tilde{\mathcal{O}}(2^\lambda) = \tilde{\mathcal{O}}(2^{c-1 \cdot \lambda})$  elements to be returned in line 5.

STEP  $i \rightarrow i+1$ . We consider a call of  $\mathbf{c}_{i+1}$ -Dissect( $\dots$ , **inner**). Fix any  $\tau_{i+1} \in \mathbb{F}_2^{(i+1) \cdot \lambda}$  in the **for**-loop in line 1. List  $L_{(c_{i+1}, c_i+1)}$  is of expected size  $\tilde{\mathcal{O}}(2^\lambda)$ . Let  $l_{(c_i, 1)}$  be an element returned to the **for**-loop in line 3 from a call of  $\mathbf{c}_i$ -Dissect( $\dots$ , **inner**). Thereby, the sum of  $l_{(c_i, 1)}$  and any element in  $L_{(c_{i+1}, c_i+1)}$  already takes the desired value on the last  $(i+1) \cdot \lambda$  coordinates.<sup>3</sup> Since we are in an **inner** call,  $\mathbf{c}_{i+1}$ -Dissect was invoked with a target of length  $(i+2) \cdot \lambda$  and the join in line 4 constitutes an additional restriction on  $(i+2) \cdot \lambda - (i+1) \cdot \lambda = \lambda$  bits. Hence, we expect  $|L_{(c_{i+1}, c_i+1)} \bowtie_t l_{(c_i, 1)}| = (|L_{(c_{i+1}, c_i+1)}| \cdot 1)/2^\lambda = \tilde{\mathcal{O}}(2^0)$  elements to be returned for each fix  $l_{(c_i, 1)}$ .

Now, by induction hypothesis  $\mathbf{c}_i$ -Dissect( $\dots$ , **inner**) returns expected  $2^{c_i-2 \cdot \lambda}$  values  $l_{(c_i, 1)}$  per iteration of its own **for**-loop (line 1) over  $\tau_i \in \mathbb{F}_2^{i \cdot \lambda}$ . The claim follows from  $(c_i-2 + i) \cdot \lambda = c_{i-1} \cdot \lambda$ .  $\square$

**Lemma 5.5.** *Let  $i \geq 1$ . Algorithm  $\mathbf{c}_i$ -Dissect( $\dots$ , **inner**) runs in expected time  $\tilde{\mathcal{O}}(2^{c_{i-1} \cdot \lambda})$ .*

*Proof.* We prove the running time by induction over  $i$ . Let  $T_{c_i}$  denote the expected running time of  $\mathbf{c}_i$ -Dissect( $\dots$ , **inner**).

BASE CLAUSE  $i = 1$ . Follows from the proof of Lemma 5.4 for  $i = 1$  and observing that the **for**-loop in line 1 iterates over  $2^\lambda$  values.

STEP  $i \rightarrow i+1$ .  $T_{c_{i+1}}$  is given by the cost of one iteration of the **for**-loop in line 1 times the number of iterations. The **for**-loop iterates over  $2^{(i+1) \cdot \lambda}$  values. Within each iteration in

**line 2** an “unbalanced” join on  $i+2$  lists  $L_{c_{i+1}}, \dots, L_{c_i+1}$  is computed taking time  $\tilde{\mathcal{O}}(2^{(i+1) \cdot \lambda})$ ,  
**line 3**  $\mathbf{c}_i$ -Dissect( $\dots$ , **inner**) is called requiring expected time  $T_{c_i}$ ,  
**line 5** expected  $\mathcal{O}(2^{c_{i-1} \cdot \lambda})$  partial single-solutions are returned.

Note that the number of elements returned in line 5 was proven in Lemma 5.4. Disregarding Oh-notation for convenience we have:

$$\begin{aligned} \log T_{c_{i+1}} &= \log \left( 2^{(i+1) \cdot \lambda} \cdot \max \left\{ 2^{(i+1) \cdot \lambda}, T_{c_i}, 2^{c_{i-1} \cdot \lambda} \right\} \right) \\ &= (i+1) \cdot \lambda + \max \left\{ (i+1) \cdot \lambda, \log T_{c_i}, c_{i-1} \cdot \lambda \right\} \end{aligned}$$

Using the induction hypothesis we have  $\log T_{c_i} = c_{i-1} \cdot \lambda$ . Thus

$$\log T_{c_{i+1}} = (i+1 + \max\{i+1, c_{i-1}\}) \cdot \lambda .$$

For  $i \geq 1$  we have  $i+1 \leq i+1 + \frac{1}{2}(i^2 - i) = c_{i-1}$ . Hence

$$\begin{aligned} \log T_{c_{i+1}} &= (c_{i-1} + i+1) \cdot \lambda \\ &= c_i \cdot \lambda . \end{aligned}$$

$\square$

**Lemma 5.6 (Running Time of  $\mathbf{c}_i$ -Dissect).** *Let  $i \geq 1$ . A call of algorithm  $\mathbf{c}_i$ -Dissect( $\dots$ , **outer**) runs in expected time  $\tilde{\mathcal{O}}(\max\{2^{c_{i-1} \cdot \lambda}, \mathbb{E}[|S|]\})$ .*

*Proof.* The proof closely follows the proof of Lemma 5.5. Let  $T_{c_i}^{\text{outer}}$  (resp.  $T_{c_i}^{\text{inner}}$ ) denote the expected running time of a call of algorithm  $\mathbf{c}_i$ -Dissect( $\dots$ , **outer**) (resp.  $\mathbf{c}_i$ -Dissect( $\dots$ , **inner**)). Again, the running time is given by the cost per iteration times the number of iterations of the **for**-loop in line 1. One easily verifies the claim for  $i = 1$ .

Consider  $T_{c_{i+1}}^{\text{outer}}$ . The **for**-loop in line 1 iterates over  $2^{(i+1) \cdot \lambda}$  values. Within each iteration in

**line 2** an “unbalanced” join on  $i+2$  lists  $L_{c_{i+1}}, \dots, L_{c_i+1}$  is computed taking time  $\tilde{\mathcal{O}}(2^{(i+1) \cdot \lambda})$ ,

<sup>3</sup> Observe that in a call  $\mathbf{c}_i$ -Dissect( $\dots$ , **inner**) partial solutions returned in line 5 already match the target on  $(i+1) \cdot \lambda$  bits (see the target length in line 3).

**line 3**  $c_i$ -Dissect( $\dots$ , inner) is invoked requiring expected time  $T_{c_i}^{\text{inner}}$ ,  
**line 6** expected  $|S|/2^{(i+1)\cdot\lambda}$  single-solutions are returned.

Dropping Oh-notation again:

$$\log T_{c_{i+1}} = (i+1) \cdot \lambda + \max \{ (i+1) \cdot \lambda, \log T_{c_i}^{\text{inner}}, \log \mathbb{E}[|S|] - (i+1) \cdot \lambda \} .$$

Now, employing Lemma 5.5 we have  $T_{c_i}^{\text{inner}} = \tilde{\mathcal{O}}(2^{c_{i-1}\cdot\lambda})$ . Hence

$$\begin{aligned} \log T_{c_{i+1}} &= (i+1) \cdot \lambda + \max \{ (i+1) \cdot \lambda, c_{i-1} \cdot \lambda, \log \mathbb{E}[|S|] - (i+1) \cdot \lambda \} \\ &= \max \{ c_i \cdot \lambda, \log \mathbb{E}[|S|] \} , \end{aligned} \quad (8)$$

similarly to the proof of Lemma 5.5 and the claim follows.  $\square$

**Lemma 5.7.** *Let  $b \in \mathbb{N}$  and  $c_i \in \text{magic}$ . For  $t \in \mathbb{F}_2^b$  let  $(L, t)$  be an instance of the  $c_i$ -SP $_b$  where we have  $|L| = N := 2^{\frac{b+c_i \cdot \log c_i + 1}{c_i - 1}}$ . Under the Independence Heuristic  $c_i$ -sum-Dissect solves the  $c_i$ -SP $_b$  with overwhelming probability in expected time  $T = \tilde{\mathcal{O}}(N^{c_{i-1}})$  and expected memory  $M = \tilde{\mathcal{O}}(N)$ .*

*Proof.* From Lemma 5.1 we know that at least  $N$  equally-split single-solutions exist with overwhelming probability. Lemma 5.2 ensures that  $c_i$ -Dissect recovers all of them. Note that the lists defined in line 1 are of length

$$2^\lambda = \frac{N}{c_i} = 2^{\frac{b+c_i \cdot \log c_i + 1}{c_i - 1} - \log c_i} = 2^{\frac{b + \log c_i + 1}{c_i - 1}} .$$

One easily verifies that  $\lambda = \frac{b + \log c_i + 1}{c_i - 1} \leq \frac{b}{i}$  as syntactically required by  $c_i$ -Dissect. Hence, Algorithm 5 solves the  $c$ -SP $_b$  with overwhelming probability.

From Lemma 5.3 we have

$$M = \tilde{\mathcal{O}}(\max\{2^\lambda, \mathbb{E}[|S|]\}) = \tilde{\mathcal{O}}(\max\{N, \mathbb{E}[|S|]\}) .$$

Under the Independence Heuristic we obtain  $\mathbb{E}[|S|] = \left(\frac{N}{c_i}\right)^{c_i} \cdot 2^{-b} = 2N$ , where the last equality follows from Equation (3). Therefore,  $M = \tilde{\mathcal{O}}(N)$ .

From Lemma 5.6 we conclude

$$T = \tilde{\mathcal{O}}(\max\{2^{c_{i-1}\cdot\lambda}, \mathbb{E}[|S|]\}) = \tilde{\mathcal{O}}(N^{c_{i-1}}) .$$

$\square$

Since  $c_{i-1} = c_i - i - 1$  by Equation (5) and  $i \approx \sqrt{2c_i}$  by Equation (6), we see that  $c_i$ -sum-Dissect reduces the time complexity of solving the  $c$ -sum-problem from  $N^{c-1}$  ( $c$ -sum-naive) down to roughly  $N^{c-\sqrt{2c}}$  (see also Table 1). Let  $c_i$ -Dissect-BKW denote the variant of  $c$ -sum-BKW, where  $c$ -sum is instantiated using  $c_i$ -sum-Dissect.

**Theorem 5.1 (Dissection Trade-Off).** *Let  $c_i \in \text{magic}$ ,  $\varepsilon > 0$  and  $k \in \mathbb{N}$  sufficiently large. Under the Independence Heuristic  $c_i$ -Dissect-BKW solves the LPN $_k$  problem with overwhelming success probability in time  $T = 2^{\vartheta(1+\varepsilon)}$ , using  $M = 2^{\mu(1+\varepsilon)}$  memory and samples, where*

$$\vartheta = \left(1 - \frac{i}{c_i - 1}\right) \cdot \log c_i \cdot \frac{k}{\log k} , \quad \mu = \frac{\log c_i}{c_i - 1} \cdot \frac{k}{\log k} .$$

*Proof.* Let  $N := 2^{\frac{b+c_i \cdot \log c_i + 1}{c_i - 1}}$ . It follows from Lemma 5.7 and Theorem 3.2 that  $c_i$ -Dissect-BKW solves LPN with overwhelming probability. We now combine Lemma 5.7 and Theorem 3.2 to compute the running time  $T$  and memory complexity  $M$  of  $c_i$ -Dissect-BKW. We have

$$\log T = c_{i-1} \cdot \frac{b + c_i \cdot \log c_i + 1}{c_i - 1} \cdot (1 + o(1)) ,$$

whereby  $c$ -sum-BKW operates on stripes of size  $b = \log c_i \cdot \frac{k \cdot (1 + \varepsilon_a)}{\log k}$ . Hence

$$\begin{aligned} \log T &= \left( \frac{c_{i-1} \log c_i}{c_i - 1} \cdot \frac{k \cdot (1 + \varepsilon_a)}{\log k} + \frac{c_{i-1} \cdot (c_i \log c_i + 1)}{c_i - 1} \right) \cdot (1 + o(1)) \\ &= \frac{c_{i-1} \log c_i}{c_i - 1} \cdot \frac{k}{\log k} \cdot (1 + \varepsilon_a + o(1)) \\ &= \frac{c_{i-1}}{c_i - 1} \cdot \log c_i \cdot \frac{k}{\log k} \cdot (1 + \varepsilon) \end{aligned}$$

for every  $\varepsilon > \varepsilon_a$  and sufficiently large  $k$ . Finally

$$\log T \stackrel{(5)}{=} \left( 1 - \frac{i}{c_i - 1} \right) \cdot \log c_i \cdot \frac{k}{\log k} \cdot (1 + \varepsilon) .$$

Analogously we have for  $M$ :

$$\log M = \frac{\log c_i}{c_i - 1} \cdot \frac{k}{\log k} \cdot (1 + \varepsilon) ,$$

for every  $\varepsilon > \varepsilon_a$  and sufficiently large  $k$ . The sample complexity of  $c$ -sum-BKW is  $N^{1+o(1)} = M$ .  $\square$

The trade-off of Theorem 5.1 clearly improves over the naive trade-off initially obtained in Theorem 4.1. While the memory consumption of the Dissection approach remains the same as for the naive trade-off, we can reduce the price we have to pay in time from  $\log c_i$  down to  $(1 - \frac{i}{c_i - 1}) \cdot \log c_i$ .

All trade-offs obtained so far are depicted in Figure 5. Note that the dashed lines merely illustrate the behavior of the trade-offs while instantiations only exist for the marked points. Although it improves

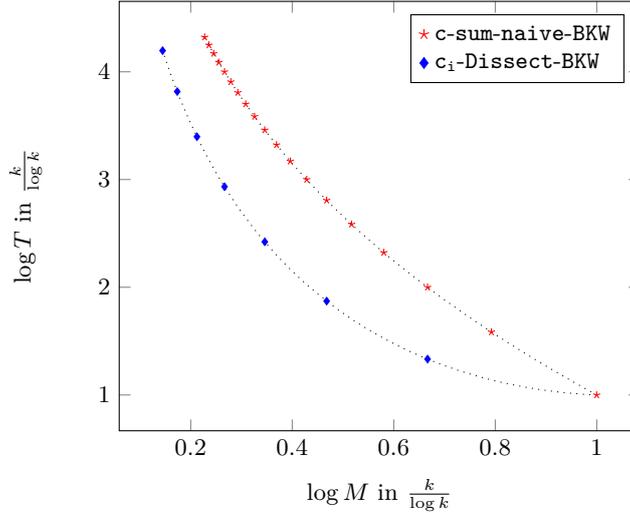


Fig. 5: New Dissection time-memory trade-off as given in Theorem 4.1 (blue diamonds from right to left for  $c_i = 4, 7, \dots \in \text{magic}$ ) compared to previous trade-off from 5.1 (red stars from right to left for  $c = 2, 3, 4, \dots$ ).

over the naive tradeoff from Section 4.1, this improvement comes at the price of covering only  $c_i$ -sums with  $c_i \in \text{magic}$ . Given some available memory  $M$ , one would choose the minimal  $c_i \in \text{magic}$  such that  $c_i$ -Dissect-BKW consumes at most memory  $M$ . However, such a choice of  $c_i$  is unlikely to fully use  $M$ . In the following section, we show how to further speed up the algorithm by using  $M$  entirely.

### 5.3 Tailored Dissection

Assume, we run  $c_i$ -sum-Dissect and our available memory is not fully used. Recall that  $c_i$ -sum-Dissect collects single-solutions while iterating an outmost loop over some constraint. In a nutshell, access to

additional memory allows us to increase the size of lists  $L_{c_i}, \dots, L_1$ , where  $N = |L_{c_i}| + \dots + |L_1|$ . Thereby, the number of equally-split single-solutions of a  $c$ -sum-problem instance increases significantly beyond  $N$ . As it suffices to identify (roughly)  $N$  single-solutions, we may prune the outmost loop of  $c_i$ -Dissect to recover  $N$  (rather than all) equally-split single-solutions.

Yet, examining a fraction of the solution space only leads to recovering a respective fraction of single-solutions if the latter are distributed sufficiently uniformly.<sup>4</sup>

Let us briefly recall that the total number of existing single-solutions taken over the initial choice of input lists is close to a binomial distribution under the Independence Heuristic. This allowed us to show that  $c_i$ -sum-Dissect succeeds with high probability as sufficiently many single-solutions exist with high probability.

Now, let us denote the random variable of the number of single-solutions gained in the  $j^{\text{th}}$  iteration of the outmost loop of  $c_i$ -Dissect by  $Z_j$ . In order to prove that a certain number of iterations is already sufficient to collect enough single-solutions with high probability we require information on the distribution of sums of  $Z_j$ . However, as we show shortly, already  $Z_j$  is distributed rather awkwardly and it seems to be a challenging task to obtain Chernoff-style results ensuring that the sum of  $Z_j$  does not fall too short from its expectation with high probability. In turn, we resort to the following heuristic.

**Tailoring Heuristic.** Let  $c_i \in \text{magic}$ . Let random variable  $Z_j$  denote the number of single-solutions gained in the  $j^{\text{th}}$  iteration of the outmost **for**-loop of  $c_i$ -Dissect taken over the initial choice of input lists. We heuristically assume that there exists a polynomial function  $\text{poly}(\lambda)$ , such that for all  $\mathcal{J} \subseteq \{1, \dots, 2^{i\lambda}\}$  we have

$$\Pr \left[ \sum_{j \in \mathcal{J}} Z_j < \frac{1}{\text{poly}(\lambda)} \cdot \mathbb{E} \left[ \sum_{j \in \mathcal{J}} Z_j \right] \right] \leq \text{negl}(\lambda) . \quad (9)$$

In particular, it follows from Equation (9) that for all  $\iota \leq 2^{i\lambda}$  we have

$$\Pr \left[ \sum_{j=1}^{\iota \cdot \text{poly}(\lambda)} Z_j \geq \mathbb{E} \left[ \sum_{j=1}^{\iota} Z_j \right] \right] \geq 1 - \text{negl}(\lambda) .$$

That is, we can compensate the deviation of the sums of  $Z_j$  below its expectation by iterating  $\text{poly}(\lambda)$  more often.<sup>5</sup>

As for the Independence Heuristic we ran experiments to verify the Tailoring Heuristic (Section 7.2).

---

**Algorithm 6** tailored- $c_i$ -sum-Dissect( $L, t$ )  $\triangleright c_i \in \text{magic}$

---

**Input:**  $L \in (\mathbb{F}_2^b)^{N^\alpha}$  where  $N := 2^{\frac{b+c_i \cdot \log c_i + 1}{c_i - 1}}$  and  $\alpha \geq 1$ , target  $t \in \mathbb{F}_2^b$

**Output:**  $S \subseteq \binom{[N^\alpha]}{c_i}$  or  $\perp$

1: Partition  $L$  into  $c_i$  lists  $L_{c_i}, \dots, L_1$  of size  $2^\lambda := \frac{N^\alpha}{c_i}$  each

2:  $S \leftarrow c_i$ -Dissect( $L_{c_i}, \dots, L_1, t, \text{outer}$ )

$\triangleright$  halt  $c_i$ -Dissect once  $|S| = N^\alpha$

3: **if**  $|S| < N^\alpha$  **then**

4:     **return**  $\perp$

5: **return**  $S$

---

We stress that the only syntactical difference of **tailored- $c_i$ -sum-Dissect** compared to  $c_i$ -sum-Dissect (Algorithm 5) is the increase of the size of list  $L$  from  $N$  to  $N^\alpha$  for  $\alpha \geq 1$  and the halt condition added as a comment in line 2.

**Lemma 5.8 (Tailored Dissection).** *Let  $b \in \mathbb{N}, c_i \in \text{magic}$  and  $\alpha \in [1, \frac{c_i-1}{c_i-1}]$ . For  $t \in \mathbb{F}_2^b$  let  $(L, t)$  be an instance of the  $c_i$ -SP $_b$  for  $|L| = N^\alpha$  whereby  $N := 2^{\frac{b+c_i \cdot \log c_i + 1}{c_i - 1}}$ . Under the Independence and Tailoring Heuristic tailored- $c_i$ -sum-Dissect solves the  $c_i$ -SP $_b$  with overwhelming probability in expected time  $T = \tilde{O}(N^{c_i-1-i \cdot (\alpha-1)})$  and expected memory  $M = \tilde{O}(N^\alpha)$ .*

<sup>4</sup> Imagine many single-solutions concentrated on few constraints  $\tau_i$  as a counterexample.

<sup>5</sup> Clearly, it does not make sense to iterate  $\iota \cdot \text{poly}(\lambda) > 2^{i \cdot \lambda}$  times. However, once we would have  $\iota \cdot \text{poly}(\lambda) > 2^{i \cdot \lambda}$  iterating  $2^{i \cdot \lambda}$  times is sufficient to collect enough single-solutions as shown in Lemma 3.1.

*Proof.* Let  $(L, t)$  be an  $c_i$ -SP $_b$  instance where  $|L| = N^\alpha$  for  $N$  and  $\alpha$  as in the statement. Lemma 5.2 implies that  $c_i$ -Dissect recovers only equally-split single-solutions. As  $N^\alpha \geq N$  we may use Lemma 5.1 stating that for a list size  $|L| = N^\alpha$  at least  $N^\alpha$  equally-split single-solutions exist with overwhelming probability. Let  $\lambda = \alpha \log N - \log c_i$  as defined in line 1 of Algorithm 6. One easily verifies that for  $\alpha \leq \frac{c_i-1}{c_{i-1}}$  we have  $i \cdot \lambda \leq b$  as syntactically required by  $c_i$ -Dissect.

We now argue that for

$$x := \max(0, b - c_{i-1} \cdot \lambda + \log c_i + 1)$$

iterating over roughly  $2^x$  constraints is sufficient to collect  $N^\alpha$  single-solutions with overwhelming probability.

First, we show that  $x \leq i \cdot \lambda$ , i.e., the number of constraints needed to find enough single-solutions, varies between one and the whole set of constraints, depending on the initial list size  $N^\alpha$ .

Since we know that  $\alpha \geq 1$ , which is equivalent to  $\lambda \geq \frac{b+\delta_i}{c_{i-1}} = \frac{b+\delta_i}{c_{i-1}+i}$ , it follows that

$$\begin{aligned} x &\leq b - c_{i-1} \frac{b + \log c_i + 1}{c_{i-1} + i} + \log c_i + 1 \\ &= \frac{i \cdot (b + \log c_i + 1)}{c_{i-1} + i} \\ &\leq i\lambda . \end{aligned}$$

Note that for  $\alpha = 1$  the algorithm collapses into  $c_i$ -sum-Dissect, since the lists are of size  $N$  and we have to iterate over the full set of constraints, which is  $x = i \cdot \lambda$ .

We now show that  $\text{poly}(\lambda) \cdot 2^x$  iterations are sufficient to find at least  $N^\alpha$  single-solutions with overwhelming probability. Define  $Z_j$  to be the number of single solutions gained in iteration  $j$  by solving a  $c_i$ -SP $_b$ . Since we just prune the outmost loop of the  $c_i$ -Dissect-call, the recursive call of  $c_{i-1}$ -Dissect is still an exhaustive search for all  $c_{i-1}$ -sums that match a given target on  $i \cdot \lambda$  bits, namely the constraint of the  $c_i$ -Dissect. So we know, that the number of returned solutions for target  $j$  of  $c_{i-1}$ -Dissect may be modeled as  $Y_j \sim \text{Bin}_{2^{\lambda c_{i-1}}, 2^{-i\lambda}}$  under the Independence Heuristic. Additionally, also using the Independence Heuristic, the size of list  $L_{c_i, c_{i-1}+1}$ , which is the join of  $i+1$  lists of size  $2^\lambda$  on  $i \cdot \lambda$  bits, is  $X_j := |L_{c_i, c_{i-1}+1}| \sim \text{Bin}_{2^{(i+1)\lambda}, 2^{-i\lambda}}$ . Note that the number of single solutions gained in iteration  $j$  is the join of the returned solutions from  $c_{i-1}$ -Dissect and the list  $L_{c_i, c_{i-1}+1}$  on the remaining  $b - i\lambda$  bits, therefore  $Z_j |_{X_j=x, Y_j=y} \sim \text{Bin}_{x \cdot y, 2^{-(b-i\lambda)}}$ .

Note that  $X_j$  and  $Y_j$  are independent, even without resorting to a heuristic as distinct lists were used to build the partial sums. This directly implies that

$$\mathbb{E}[Z_j] = \mathbb{E}[\mathbb{E}[Z_j | (X_j, Y_j)]] = \mathbb{E}[X_j \cdot Y_j \cdot 2^{-(b-i\lambda)}] = \mathbb{E}[X_j] \cdot \mathbb{E}[Y_j] \cdot 2^{-(b+i\lambda)} = 2^{(c_{i-1}+1)\lambda-b} . \quad (10)$$

Now, under the Tailoring Heuristic we have that with overwhelming probability

$$|S| = \sum_{j=1}^{\text{poly}(\lambda) \cdot 2^x} Z_j \geq \mathbb{E} \left[ \sum_{j=1}^{2^x} Z_j \right] .$$

Since all  $Z_j$  are identically distributed it follows

$$|S| \geq \sum_{j=1}^{2^x} \mathbb{E}[Z_j] = 2^x \cdot \mathbb{E}[Z_1] \stackrel{(10)}{=} 2^{(c_{i-1}+1)\lambda-b+x} = 2^{\lambda+\log c_i+1} = 2 \cdot N^\alpha ,$$

with overwhelming probability. Hence,  $\text{poly}(\lambda) \cdot 2^x = \tilde{\mathcal{O}}(2^x)$  iterations are sufficient to collect  $N^\alpha$  single-solutions with overwhelming probability.

Since **tailored- $c_i$ -sum-Dissect** only splits the input list and then invokes  $c_i$ -Dissect memory and time complexity remain as in  $c_i$ -Dissect. Hence, the expected memory complexity is given by Lemma 5.3 as  $\tilde{\mathcal{O}}(\max(2^\lambda, |S|)) = \tilde{\mathcal{O}}(N^\alpha)$  since the algorithm terminates as soon as  $N^\alpha$  single-solutions are found.

The expected running time of  $c_i$ -Dissect follows from Equation (8) in Lemma 5.6, by setting  $i$  to  $i-1$  (because of the induction) and by observing that now expected  $\mathbb{E}[Z_j] = 2^{(c_i-i)\lambda-b}$  single-solutions

are returned in each iteration. This leads to  $T = \tilde{O}\left(2^x \cdot \max(2^{c_{i-2}\lambda}, 2^{(c_i-i)\lambda-b})\right)$ . Before analyzing  $T$ , we show that  $x := \max(0, b - c_{i-1} \cdot \lambda + \log c_i + 1)$  and  $b - c_{i-1} \cdot \lambda + \log c_i + 1$  only differ by a constant. This follows from the fact, that  $\alpha \leq \frac{c_i-1}{c_{i-1}}$  and therefore  $\lambda \leq \frac{c_i-1}{c_{i-1}} \log N - \log c_i = \frac{b+c_i \cdot \log c_i+1}{c_{i-1}} - \log c_i$ , which gives us

$$\begin{aligned} x &= b - c_{i-1}\lambda + \log c_i + 1 \\ &\geq b - c_{i-1} \left( \frac{b + c_i \cdot \log c_i + 1}{c_{i-1}} - \log c_i \right) + \log c_i + 1 \\ &= c_{i-1} \log c_i - c_i \cdot \log c_i + \log c_i . \end{aligned}$$

Using  $c_{i-1} = c_i - i - 1$  we get

$$x \geq -i \cdot \log c_i + 1 .$$

Therefore the maximum just leads to the addition of a constant. For the running time  $T$  it follows, that

$$\begin{aligned} \log T &= \max(b - c_{i-1}\lambda + c_{i-2} \cdot \lambda, \alpha \log N) \\ &= \max(b - (c_{i-1} - c_{i-2}) \cdot \lambda, \alpha \log N) \\ &= \max(b - i \cdot \lambda, \alpha \log N) . \end{aligned}$$

Using  $b = \log N \cdot (c_i - 1) - c_i \cdot \log c_i - 1$  and  $\lambda = \alpha \log N - \log c_i$ , we get

$$\begin{aligned} \log T &= \max(\log N \cdot (c_i - 1) - c_i \cdot \log c_i - 1 - i \cdot (\alpha \log N - \log c_i), \alpha \log N) \\ &= \max(\log N \cdot (c_{i-1} - i \cdot (\alpha - 1)) - c_i \cdot \log c_i - 1 + i \cdot \log c_i, \alpha \log N) . \end{aligned}$$

Note that  $T = \tilde{O}(\max(N^{c_{i-1}-i \cdot (\alpha-1)}, N^\alpha))$  since  $c_i$  as well as  $i$  are constant.

It remains to show, that  $\tilde{O}(\max(N^{c_{i-1}-i \cdot (\alpha-1)}, N^\alpha)) = \tilde{O}(N^{c_{i-1}-i \cdot (\alpha-1)}) = T$ . To this end note that

$$\begin{aligned} N^{c_{i-1}-i \cdot (\alpha-1)} &\geq N^\alpha \\ \Leftrightarrow c_{i-1} - i \cdot (\alpha - 1) &\geq \alpha \\ \Leftrightarrow \frac{c_{i-1} + i}{i + 1} &\geq \alpha , \end{aligned}$$

since  $\alpha \leq \frac{c_i-1}{c_{i-1}}$  we get the more restrictive condition

$$\begin{aligned} \Leftarrow \frac{c_{i-1} + i}{i + 1} &\geq \frac{c_i - 1}{c_{i-1}} \\ \Leftrightarrow \frac{c_{i-1}}{i + 1} &\geq 1 , \end{aligned}$$

which holds for all  $i$ . □

The complexities of **tailored- $c_i$ -sum-Dissect** are given in Table 1 where we simplified  $\alpha$ 's upper bound using  $c_{i-1} \approx c_i - \sqrt{2c_i}$ . Let **tailored- $c_i$ -BKW** denote **c-sum-BKW**, where the  $c$ -sum-problem is solved via **tailored- $c_i$ -sum-Dissect**.

**Theorem 5.2 (Tailored-Dissection Trade-Off).** *Let  $c_i \in \text{magic}$  and further  $\alpha \in [1, \frac{c_i-1}{c_{i-1}}]$ ,  $\varepsilon > 0$  and  $k \in \mathbb{N}$  sufficiently large. Under the Independence and Tailoring Heuristic **tailored- $c_i$ -BKW** solves the  $\text{LPN}_k$  problem with overwhelming success probability in time  $T = 2^{\vartheta(1+\varepsilon)}$ , using  $M = 2^{\mu(1+\varepsilon)}$  memory and samples, where*

$$\vartheta = \left(1 - \frac{\alpha \cdot i}{c_i - 1}\right) \cdot \log c_i \cdot \frac{k}{\log k} , \quad \mu = \frac{\alpha \cdot \log c_i}{c_i - 1} \cdot \frac{k}{\log k} .$$

*Proof.* We aim at using **tailored- $c_i$ -sum-Dissect** to instantiate the **c-sum** subroutine of **c-sum-BKW**. Note that **c-sum-BKW** will invoke the **c-sum** subroutine with a list of size  $N^\alpha$ , where  $\log N = \frac{b+c_i \cdot \log c_i+1}{c_{i-1}}$ , which guarantees  $\alpha \log N \geq \frac{b+c_i \cdot \log c_i+1}{c_{i-1}}$  as required by **c-sum-BKW** (see Theorem 3.2).

The time complexity of the **tailored-c<sub>i</sub>-sum-Dissect**, when invoked with a list of size  $N^\alpha$  is given as  $T_{c_i, N} = \tilde{O}(N^{c_{i-1}-i \cdot (\alpha-1)})$  by Lemma 5.8. Using  $\log N = \frac{b+c_i \cdot \log c_i + 1}{c_i - 1}$  and  $c_{i-1} = c_i - i - 1$ , we get

$$\frac{b + c_i \cdot \log c_i + 1}{c_i - 1} \cdot (c_{i-1} - i \cdot (\alpha - 1)) = \left(1 - \frac{\alpha \cdot i}{c_i - 1}\right) \cdot (b + c_i \cdot \log c_i + 1) .$$

As  $c_i$  is constant we have

$$T_{c_i, N} = \tilde{O}\left(2^{\left(1 - \frac{\alpha \cdot i}{c_i - 1}\right) \cdot b}\right) .$$

Using Theorem 3.2 gives  $T = (T_{c_i, N})^{1+o(1)} = 2^{\left(1 - \frac{\alpha \cdot i}{c_i - 1}\right) \cdot b \cdot (1+o(1))}$ . Now recall that the **c<sub>i</sub>-SP<sub>b</sub>** solved as subroutine in the **c-sum-BKW** algorithm operates on stripes of size  $b = \log c_i \cdot \frac{k(1+\varepsilon_a)}{\log k}$  leading to

$$\begin{aligned} \log T &= \left(1 - \frac{\alpha \cdot i}{c_i - 1}\right) \cdot \log c_i \cdot \frac{k \cdot (1 + \varepsilon_a)}{\log k} \cdot (1 + o(1)) \\ &= \left(1 - \frac{\alpha \cdot i}{c_i - 1}\right) \cdot \log c_i \cdot \frac{k}{\log k} \cdot (1 + \varepsilon_a + o(1)) \\ &= \left(1 - \frac{\alpha \cdot i}{c_i - 1}\right) \cdot \log c_i \cdot \frac{k}{\log k} \cdot (1 + \varepsilon) , \end{aligned}$$

for every  $\varepsilon > \varepsilon_a$  and sufficiently large  $k$ , as claimed.

The memory complexity of **tailored-c<sub>i</sub>-sum-Dissect** is  $M_{c_i, N} = \tilde{O}(N^\alpha)$ , according to Lemma 5.8, whereas the memory of **c-sum-BKW** is given by Theorem 3.2 as  $M = (M_{c_i, N})^{(1+o(1))} = N^{\alpha \cdot (1+o(1))}$ . Hence

$$\begin{aligned} \log M &= \frac{b + c_i \cdot \delta_i}{c_i - 1} \cdot \alpha \cdot (1 + o(1)) \\ &= \left(\frac{\log c_i}{c_i - 1} \cdot \frac{k \cdot (1 + \varepsilon_a)}{\log k} + \frac{c_i \cdot \delta_i}{c_i - 1}\right) \cdot \alpha \cdot (1 + o(1)) \\ &= \frac{\alpha \cdot \log c_i}{c_i - 1} \cdot \frac{k}{\log k} \cdot (1 + \varepsilon_a + o(1)) \\ &= \frac{\alpha \cdot \log c_i}{c_i - 1} \cdot \frac{k}{\log k} \cdot (1 + \varepsilon) , \end{aligned}$$

for every  $\varepsilon > \varepsilon_a$  and sufficiently large  $k$ . We have a sample complexity of  $N^{\alpha \cdot (1+o(1))} = 2^{\frac{b+c_i \cdot \log c_i + 1}{c_i - 1} \cdot \alpha \cdot (1+o(1))}$  and therefore equal to the memory complexity  $M$ .  $\square$

The trade-off obtained from Theorem 5.2 as well as both trade-offs previously obtained are given in Figure 6.

At least intuitively it is clear how to choose optimal parameters for **tailored-c<sub>i</sub>-BKW** for any given amount of memory  $M$ : Find the minimal  $c_i \in \text{magic}$  such that **c<sub>i</sub>-Dissect-BKW** uses at most memory  $M$ . Then, resort to **tailored-c<sub>i</sub>-BKW** using memory  $M$  entirely. We now provide the formal justification proving that the approach sketched above is indeed admissible and optimal captured in the following Corollary.

**Corollary 5.1.** *Let  $\varepsilon > 0$  and  $k \in \mathbb{N}$  sufficiently large. Consider memory  $M = 2^{m \cdot \frac{k}{\log k} \cdot (1+\varepsilon)}$  for  $m \in (0, 1]$ . Let*

$$i^* := \min_{i \in \mathbb{N}} \left\{ \frac{\log c_i}{c_i - 1} \leq m \right\} .$$

*Under the Independence Heuristic **tailored-c<sub>i^\*</sub>-BKW** solves the  $\text{LPN}_k$  problem with overwhelming success probability in time  $T = 2^{\vartheta(1+\varepsilon)}$  where*

$$\vartheta = (\log c_{i^*} - i^* \cdot m) \cdot \frac{k}{\log k} .$$

*Proof.* First we show that for any given amount of memory  $M$  there is an instantiation of **tailored-c<sub>i</sub>-BKW** with memory complexity  $M_\alpha = M$ . We do so, by showing that there is a possible parametrization to achieve any memory consumption between the requirements of **c<sub>i-1</sub>-Dissect-** and **c<sub>i</sub>-Dissect-BKW**.

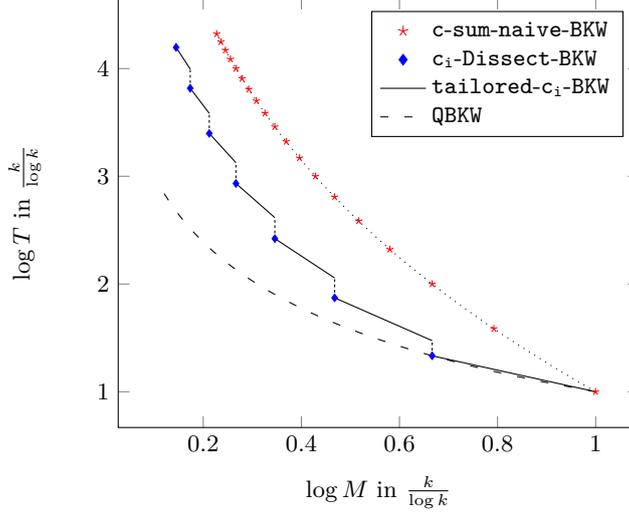


Fig. 6: Time-memory trade-off obtained via Theorem 5.2 (solid lines) as generalization of the Dissection trade-off (blue diamonds, right to left for  $c_i = 4, 7, 11, \dots \in \text{magic}$ ) as given in 5.1. Naive trade-off (red stars, right to left for  $c = 2, 3, 4, \dots$ ) as given in Theorem 4.1, Trade-off obtained through QBKW given as dashed line for orientation.

The memory complexity  $M_\alpha$  of **tailored- $c_i$ -BKW**, where

$$\log M_\alpha = \frac{\alpha \cdot \log c_i}{c_i - 1} \cdot \frac{k}{\log k} \cdot (1 + \varepsilon), \text{ with } \alpha \in \left[1, \frac{c_i - 1}{c_{i-1}}\right] \text{ and } \varepsilon > 0,$$

is obviously continuous in  $\alpha$ . Remember that the **tailored- $c_i$ -BKW** when minimizing the memory consumption, choosing  $\alpha = 1$ , collapses to  **$c_i$ -Dissect-BKW**, as stated in the proof of Lemma 5.8, therefore having the same memory (and time) complexity. On the other hand, when we maximize the memory consumption, choosing  $\alpha = \frac{c_i - 1}{c_{i-1}}$  we get

$$\log M_\alpha = \frac{\log c_i}{c_{i-1}} \cdot \frac{k}{\log k} \cdot (1 + \varepsilon)$$

for sufficiently large  $k$ . If we compare  $M_\alpha$  to the memory requirements  $M_{-1}$  of  **$c_{i-1}$ -Dissect-BKW** we get:

$$\begin{aligned} \log M_\alpha &\geq \log M_{-1} \\ \Leftrightarrow \frac{\log c_i}{c_{i-1}} &\geq \frac{\log c_{i-1}}{c_{i-1} - 1}, \end{aligned} \quad (11)$$

where Equation (11) holds for all  $i \geq 1$ , which can easily be verified numerically. Since  $M_\alpha$  is continuous in  $\alpha$ , this implies a possible instantiation for any given memory in the interval of the memory requirements of  **$c_i$ -Dissect-** and  **$c_{i-1}$ -Dissect-BKW** for all  $i \geq 2$  (because there is no dissection defined for  $c_0$ ). For  $i = 1$  one easily verifies, that **tailored- $c_1$ -BKW** has a maximal memory complexity of  $M_\alpha = \frac{k}{\log k} \cdot (1 + \varepsilon)$  for  $\alpha = \frac{c_1 - 1}{c_0} = \frac{3}{2}$ , which leads to a possible instantiation for any given memory.

We now show that the choice of  $i^* := \min_{i \in \mathbb{N}} \left\{ \frac{\log c_i}{c_i - 1} \leq m \right\}$  is optimal, in terms of minimizing the time complexity  $T_\alpha$  of **tailored- $c_i$ -BKW**

$$\log T_\alpha = \left(1 - \frac{\alpha \cdot i}{c_i - 1}\right) \cdot \log c_i \cdot \frac{k}{\log k} \cdot (1 + \varepsilon),$$

which decreases linearly in  $\alpha$ . Hence, it suffices to show that the running time  $T_\alpha$  of **tailored- $c_i$ -BKW** is at least the running time of  **$c_{i-1}$ -Dissect-BKW**. Let  $T_{-1}$  be the time complexity of  **$c_{i-1}$ -Dissect-BKW**, with

$$\log T_{-1} = \left(1 - \frac{i - 1}{c_{i-1} - 1}\right) \cdot \log c_{i-1} \cdot \frac{k}{\log k} \cdot (1 + \varepsilon)$$

and set  $\alpha = \frac{c_i-1}{c_{i-1}}$ . We already know, that for this choice of  $\alpha$  time  $T_\alpha$  will become minimal. Further

$$\begin{aligned} \log T_\alpha &\geq \log T_{-1} \\ \Leftrightarrow \left(1 - \frac{i}{c_{i-1}}\right) \log c_i &\geq \left(1 - \frac{i-1}{c_{i-1}-1}\right) \cdot \log c_{i-1} \\ \Leftrightarrow \frac{c_{i-1}-i}{c_{i-1}} \cdot \log c_i &\geq \frac{c_{i-1}-i}{c_{i-1}-1} \cdot \log c_{i-1} \\ \Leftrightarrow \frac{\log c_i}{c_{i-1}} &\geq \frac{\log c_{i-1}}{c_{i-1}-1}, \end{aligned}$$

which has already been stated in Equation (11). Hence, **tailored- $c_i$ -BKW** is slower than  **$c_{i-1}$ -Dissect-BKW** regardless of the available memory. Further,  **$c_{i^*}$ -Dissect-BKW** does not require more memory than  $M$  since  $\frac{\log c_{i^*}}{c_{i^*}-1} \leq m$  by choice of  $i^*$ .

One easily verifies that setting  $\alpha^* = \frac{m \cdot (c_i-1)}{\log c_i}$  leads to an overall memory consumption of  $M$  and the stated running time. Thus, it remains to show that  $\alpha^*$  is an admissible choice satisfying  $1 \leq \alpha^* \leq \frac{c_i-1}{c_{i-1}}$ . Because of the minimality of  $i^*$  we have  $m < \frac{\log c_{i^*-1}}{c_{i^*-1}-1}$ , hence  $\alpha^* \leq \frac{(c_i-1) \log c_{i^*-1}}{c_{i^*-1} \cdot \log c_i}$ . Note that  $\alpha^* \leq \frac{c_i-1}{c_{i-1}}$  follows again from (11). Also,  $m \geq \frac{\log c_{i^*}}{c_{i^*}-1}$  and thus  $\alpha^* \geq 1$ .

Since  $T_{\alpha^*}$  is decreasing linearly in  $\alpha^*$ , maximizing  $\alpha^* = \frac{m \cdot (c_i-1)}{\log c_i}$ , i.e., maximizing  $m$  and minimizing  $i$ , is optimal.  $\square$

**The Curious Case of  $i = 1$ .** As already seen various times for  $i \geq 2$  we have

$$\frac{1}{c_{i-1}} > \frac{\log c_{i-1}}{(c_{i-1}-1) \log c_i} \Leftrightarrow \log T_\gamma > \log T_{-1} .$$

That is, for  $i \geq 2$  algorithm **tailored- $c_i$ -BKW** is strictly slower than  **$c_{i-1}$ -Dissect-BKW** (see the leaps in time in Figure 6). However, for  $i = 1$  the trade-off does exactly interpolate between  **$c_1$ -Dissect-BKW** and the complexities of the original **BKW** algorithm. To explain this behavior one observes that **tailored- $c_1$ -BKW** using as much memory as allowed is merely a slight modification of **2-sum-naive-BKW** in turn having the same complexities as the original **BKW** algorithm:

To this end note that a  $c_1 = 4$  tailored-dissection with maximal input lists, operates on lists of size  $2^{b/2}$ , selecting the constrained parameters  $\tau_i$  from  $\mathcal{T} \subseteq \mathbb{F}_2^{b/2}$ . Since the input lists are that large, the iteration over a subset of size  $|\mathcal{T}| = \mathcal{O}(1)$  is sufficient to find a solution to the  **$c_1$ -SP $_b$** . Internally a 4-dissection looks for 2-sums of vectors matching  $\tau_i$  on  $\frac{b}{2}$  bits between lists  $L_1$  and  $L_2$  and  $\tau_i \oplus t$  between  $L_3$  and  $L_4$ , where  $t$  is the target. Then the found 2-sums are matched to  $t$  on the remaining  $\frac{b}{2}$  bits. So one iteration of the  **$c_1$ -Sum-Dissect** can be seen as building a 2-sum of vectors, twice. The  **$c$ -sum-BKW** algorithm operates in this case on stripes of size  $b = \frac{2 \cdot k \cdot (1+\varepsilon_a)}{\log k}$ , so each 2-sum is matched on  $\frac{b}{2} = \frac{k \cdot (1+\varepsilon_a)}{\log k}$  bits, which is exactly the length of the stripes the **2-sum-naive-BKW** operates on. Therefore, one iteration of the  **$c_1$ -Sum-Dissect** correlates to two iterations of the **2-sum-naive-BKW**. But in total we are only iterating  $a = \frac{\log k}{2 \cdot (1+\varepsilon_a)}$  times instead of  $2 \cdot a$  times, like the **2-sum-naive-BKW** does, resulting in an equal amount of 2-sums the final unit vector consists of, while each 2-sum is matched on the same target size of  $\frac{b}{2}$  bits.

## 6 Time-Memory Trade-Offs for solving LWE

Introduced by Regev [32] the *Learning With Errors* (LWE) problem is a versatile generalization of LPN to arbitrary moduli  $q$ . In [2] it is shown how to employ a **BKW**-type algorithm to solve the LWE problem. We briefly discuss that all trade-offs obtained for the **BKW** algorithm to solve LPN carry over to using **BKW** to solve the LWE problem.

In short, transitioning from solving LPN to **BKW** we essentially lose the  $\log k$ -factor in the denominator in our complexity exponents similarly to [2].

**Definition 6.1 (LWE Problem).** Let  $k \in \mathbb{N}$ ,  $q = k^{\nu_q}$ ,  $\mathbf{s} \in \mathbb{F}_q^k$  and  $\mathcal{D}_\sigma$  be a discrete Gaussian distribution with mean zero and variance  $\sigma^2$ , where  $\sigma = k^{\nu_s}$ . Let  $\text{Sample}_q$  denote an oracle that, when queried, samples  $\mathbf{a} \sim \mathcal{U}_{\mathbb{F}_q^k}$ ,  $e \sim \mathcal{D}_\sigma$  and outputs a sample of the form

$$(\mathbf{a}, b) := (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e) .$$

The  $\text{LWE}_k$  problem consists of recovering  $\mathbf{s}$  given access to  $\text{Sample}_q$ .

The BKW algorithm may be slightly tweaked to work in the LWE setting as shown by Albrecht et al. [2]. To this end one still adds two samples in each step to create zeroes (now in  $\mathbb{F}_q$ ) on stripes of width  $b$ . However, the number of samples needed for the majority vote in this setting is

$$m = e^{\frac{4\pi^2 \sigma^2 2^a}{q^2}} ,$$

after  $a$  BKW steps, as shown in [24]. Setting  $a := (1 - \varepsilon_a + 2\nu_q - 2\nu_s) \cdot \log k$  for  $\varepsilon_a > 0$  yields  $m = e^{4\pi^2 k^{1-\varepsilon}}$  and results in a total time, memory and sample complexity of

$$\tilde{\mathcal{O}}\left(q^b \cdot e^{4\pi^2 k^{1-\varepsilon_a}}\right) = q^{b(1+\varepsilon)} = 2^{\frac{1}{2} \cdot \frac{\nu_q}{\nu_q - \nu_s + \frac{1}{2}} k^{(1+\varepsilon)}} ,$$

for any  $\varepsilon > 0$  and sufficiently large  $k$ , which can also be found in [20].

Clearly, we can generalize the  $c$ -sum-problem to arbitrary modulus  $q$ . The main observation is that all given algorithms solving the  $c$ -SP $_b$  may be employed to solve the  $c$ -sum-problem mod  $q$ . To this end we note that the naive approach, as well as the two Dissection algorithms merely require addition in the finite field (denoted  $\oplus$  over  $\mathbb{F}_2$ ) and a total order on list elements, e.g., lexicographic order.

Eventually,  $c$ -sum-BKW has a natural generalization to the LWE setting, too. As the probability of two uniform vectors adding to some fixed pattern on  $b$  coordinates is  $\frac{1}{q^b}$  instead of  $\frac{1}{2^b}$ , we have to adjust the number of samples initially collected. Following the lines of the proof of Lemma 3.1 we see that  $N \geq q^{\frac{b+c\delta}{c-1}}$  list elements are required for the  $c$ -sum-problem to have a solution with overwhelming probability. As before, as we add  $c$  rather than two samples in each step, we adjust  $a$  by a factor of  $\log c$  and set

$$a := (1 - \varepsilon_a + 2\nu_q - 2\nu_s) \cdot \frac{\log k}{\log c}$$

for  $\varepsilon_a > 0$ .

We summarize the resulting complexities. For an algorithm as given by an entry in Table 2 we obtain complexities time  $T = 2^{\vartheta(1+\varepsilon)}$  and memory/samples  $M = 2^{\mu(1+\varepsilon)}$  where

$$\vartheta = t \cdot \frac{1}{2} \cdot \frac{\nu_q}{\nu_q - \nu_s + \frac{1}{2}} \cdot k , \quad \mu = m \cdot \frac{1}{2} \cdot \frac{\nu_q}{\nu_q - \nu_s + \frac{1}{2}} \cdot k , \quad (12)$$

and  $t, m$  as given in Table 2.

Table 2: Partial complexity exponents of our various algorithms solving the LWE problem. For each algorithm using the respective values of  $t$  and  $m$  in Equation (12) gives the corresponding complexities.

	Algorithm	$t$	$m$	for
	BKW	1	1	
classic	$c$ -sum-naive-BKW	$\log c$	$\frac{\log c}{c-1}$	$c \geq 2$
	$c_1$ -Dissect-BKW	$\left(1 - \frac{i}{c_i-1}\right) \cdot \log c_i$	$\frac{\log c_i}{c_i-1}$	$c_i \in \text{magic}$
	tailored- $c_1$ -BKW	$\left(1 - \alpha \cdot \frac{i}{c_i-1}\right) \cdot \log c_i$	$\alpha \cdot \frac{\log c_i}{c_i-1}$	$c_i \in \text{magic}, \alpha \in [1, \frac{c_i-1}{c_i-1}]$
quantum	QBKW	$\frac{1}{2} \cdot \frac{c}{c-1} \cdot \log c$	$\frac{\log c}{c-1}$	$c \geq 2$

## 7 Experimental Verification of Heuristics

We present experimental results to verify our Independence Heuristic as well as our Tailoring Heuristic.

### 7.1 Experiments for the Independence Heuristic

We tested the Independence Heuristic for  $c_i \in \{4, 7\}$ . We iteratively monitored the number of single-solutions found after a run of `ci-sum-Dissect` on successive stripes starting with a list of size  $N$ . For each  $c_i$  we repeatedly called `ci-sum-Dissect` on three stripes. After each call we stored the number of single-solutions found. To analyze the impact of dependencies amongst the list elements—rather than influences due to variations in the number of single-solutions found—we pruned lists of more than  $N$  single-solutions down to  $N$  before starting the next run, and discarded lists where less than  $N$  single-solutions were found.

Note that during the first run all list elements are independent and uniformly distributed, even if their  $c$ -sums are not. While the list elements remain uniform on a “fresh” stripe in subsequent runs of `ci-sum-Dissect` they are not independent anymore what could affect (besides  $c$ -sums being dependent) the distribution of existing single-solutions even further. Under the Independence Heuristic the number of single-solutions found is close to being  $\text{Bin}_{(N/c_i)^{c_i}, 2^{-b}}$  distributed after any run of `ci-sum-Dissect`.

Our experiments with parameters as given in Table 3 lead to the plots given in Figure 7. The measured relative frequencies are given by points, while the continuous plots are the benchmark distributions  $\text{Bin}_{(N/c_i)^{c_i}, 2^{-b}}$  for our Independence Heuristic.

Table 3: Parameters for testing the Independence Heuristic. Parameter  $a$  denotes the number of stripes of width  $b$ .

$c_i$	$a$	$b$	$N$	run	sample size in thous.	given in
$c_1 = 4$	3	25	2048	1	108	Figure 7a
				2	100	Figure 7b
				3	58	Figure 7c
$c_2 = 7$	3	33	441	1	29	Figure 7d
				2	23	Figure 7e
				3	18	Figure 7f

We see from Figure 7 that the distribution of the output list-size is close to their benchmark, even after three iterations of the `c-sum` subroutine, where already  $4^3 = 64$ -sums (resp.  $7^3 = 343$ -sums) haven been built.

We also used the Independence Heuristic in Lemma 5.8 to show that the random variables  $Z_j$  of the number of single-solutions gained in the  $j^{\text{th}}$  iteration over a constraint is close to being binomially  $\text{Bin}_{x \cdot y, 2^{-(b-i\lambda)}}$  distributed, where  $x \sim \text{Bin}_{2^{(i+1)\lambda}, 2^{-i\lambda}}$  and  $y \sim \text{Bin}_{2^{\lambda c_{i-1}}, 2^{-i\lambda}}$ . In order to verify this experimentally, we computed several instances with parameter set

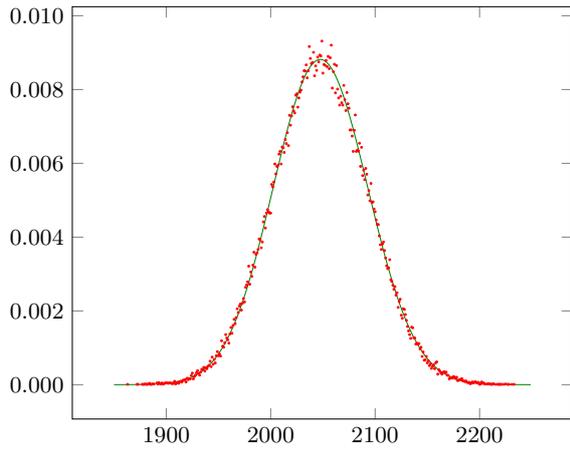
$$i = 1 \ (c_1 = 4), \ b = 25, \ a = 3, \ N = 8192 \ .$$

Each time we performed three consecutive runs of `4-sum-Dissect` on successive stripes and stored the number of single-solutions obtained per constraint after each iteration. The results are given in Figure 8. Again, the obtained relative frequencies accurately match the benchmark curve  $\text{Bin}_{x \cdot y, 2^{-(b-i\lambda)}}$ .

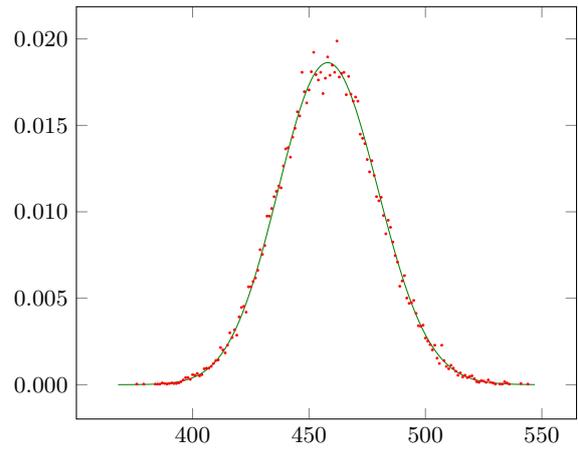
### 7.2 Experiments on the Tailoring Heuristic

As in the previous subsection, let  $Z_j$  be the number of single-solutions obtained per constraint iteration in `ci-sum-Dissect`. In Tailored Dissection it is required that the sum of these random variables is close to its expectation. The Tailoring Heuristic states that this is true with overwhelming probability, if we slightly increase the expected number of required iterations by a polynomial factor.

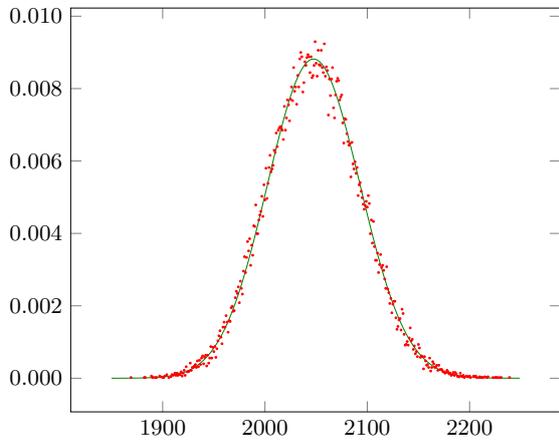
To test the Tailoring Heuristic we ran `4-sum-Dissect` (without tailoring) on three stripes ( $a = 3$ ) with  $N = 8192$  and varying choice of  $b$ . We summed the numbers  $Z_j$  of single-solutions found per randomized



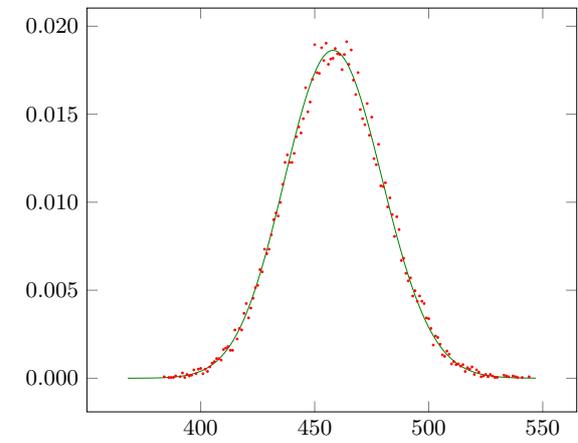
(a) Distribution of the number of single-solutions after run 1 of 4-sum-Dissect. Sample size in thousands: 108.



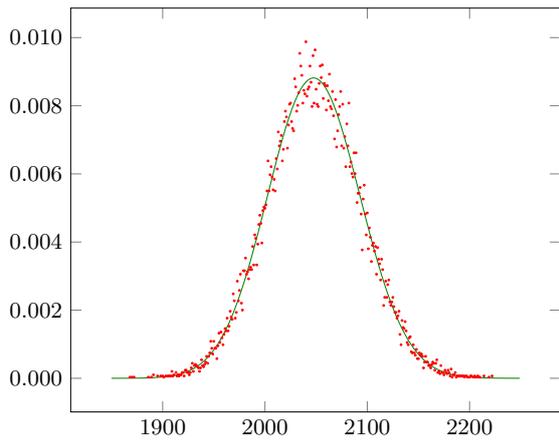
(d) Distribution of the number of single-solutions after run 1 of 7-sum-Dissect. Sample size in thousands: 29.



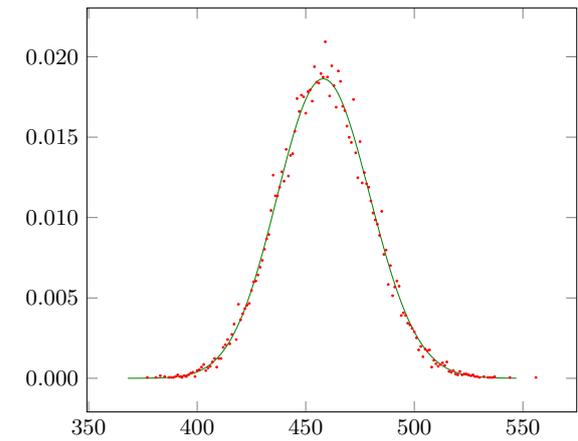
(b) Distribution of the number of single-solutions after run 2 of 4-sum-Dissect. Sample size in thousands: 100.



(e) Distribution of the number of single-solutions after run 2 of 7-sum-Dissect. Sample size in thousands: 23.

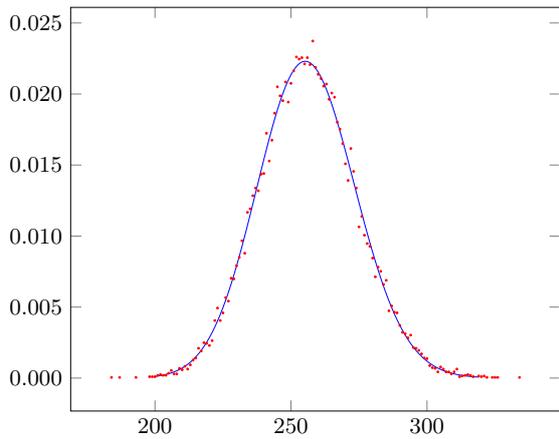


(c) Distribution of the number of single-solutions after run 3 of 4-sum-Dissect. Sample size in thousands: 58.

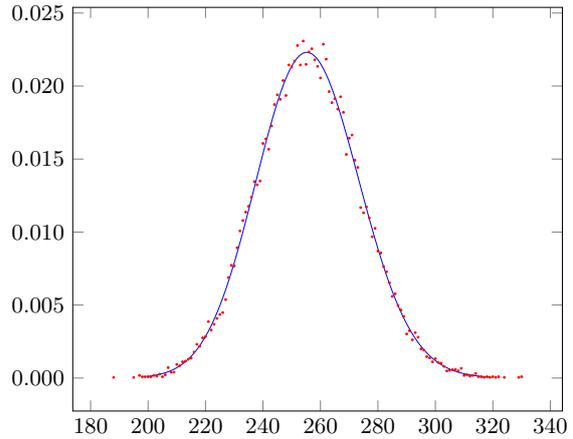


(f) Distribution of the number of single-solutions after run 3 of 7-sum-Dissect. Sample size in thousands: 18.

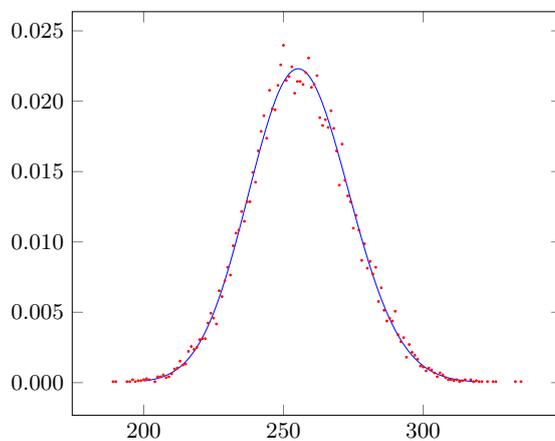
Fig. 7: Distribution of the number of single-solutions over successive runs of  $c_1$ -sum-Dissect. Under the Independence Heuristic this distribution is close to  $\text{Bin}_{(N/c_1)^{c_i}, 2^{-b}}$ . All parameters are given in Table 3.



(a) First run. Sample size in thous.: 20.



(b) Second run. Sample size in thous.: 22.



(c) Third run. Sample size in thous.: 14.

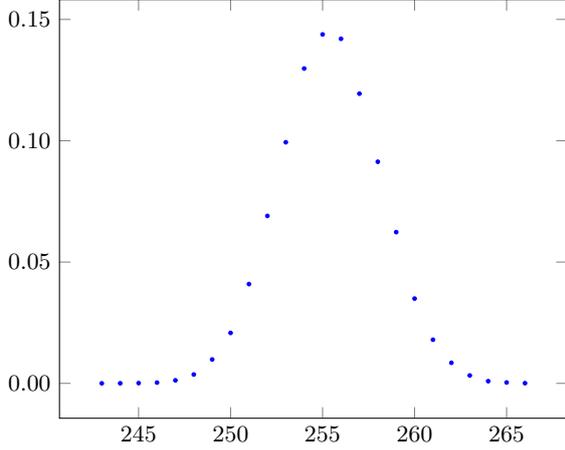
Fig. 8: Distribution of the number of single-solutions per constraint in the first (Figure 8a), second (Figure 8b) and third run (Figure 8c) of 4-sum-Dissect.

Table 4: Parameters and results for testing the Tailoring Heuristic for 4-sum-Dissect,  $N = 8192$ .

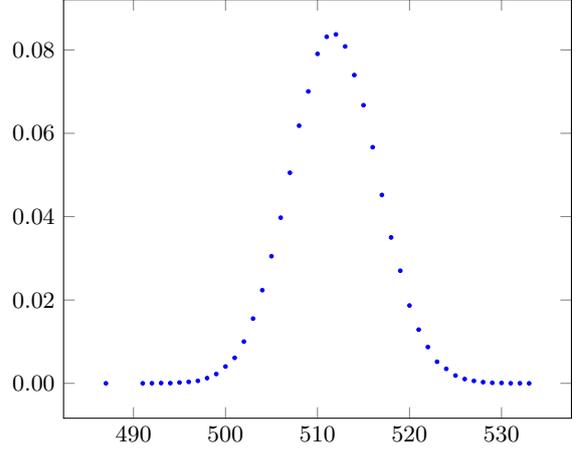
$b$	$\mathbb{E}[\text{iterations to reach } N]$		99% confidence interval	sample size in thous.	given in
	theory	experiments			
25	32	32.50	$32.50 \cdot (1 \pm 0.031)$	50	
28	256	255.40	$255.40 \cdot (1 \pm 0.031)$	100	Figure 9a
29	512	511.78	$511.78 \cdot (1 \pm 0.026)$	150	Figure 9b
30	1024	1024.20	$1024.20 \cdot (1 \pm 0.021)$	250	Figure 9c

constraint  $\tau_j$  during the last run in a list, until their sum exceeded  $N$ . The results can be found in Table 4 and Figure 9.

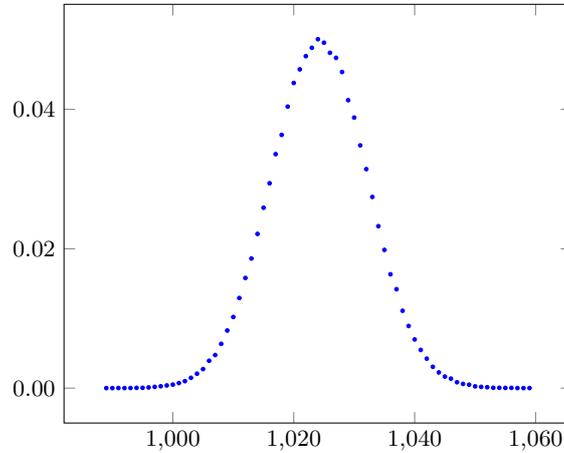
We see in Table 4 that the experimentally required numbers of iteration very accurately match their theoretical predictions (that were computed under the Independence Heuristic). Moreover, we experimentally need only a small factor to achieve a 99% confidence interval, even for low expectations. This means that the distribution has small variance and sharply concentrates around its mean, as can also be seen in Figure 9. This all supports the validity of our Tailoring Heuristic for the analysis of Tailored Dissection BKW.



(a)  $b = 28$ . Parameters of sample set  $X_1$ :  $\mathbb{E}[X_1] = 255.40$ ,  $\text{Var}[X_1] = 7.55$ ,  $\sigma_{X_1} = 2.75$ , 99% confidence interval  $\mathbb{E}[X_1] \pm 8$ .



(b)  $b = 29$ . Parameter of sample set  $X_2$ :  $\mathbb{E}[X_2] = 511.78$ ,  $\text{Var}[X_2] = 22.81$ ,  $\sigma_{X_2} = 4.78$ , 99% confidence interval  $\mathbb{E}[X_2] \pm 13$ .



(c)  $b = 30$ . Parameter of sample set  $X_3$ :  $\mathbb{E}[X_3] = 1024.20$ ,  $\text{Var}[X_3] = 63.99$ ,  $\sigma_{X_3} = 7.99$ , 99% confidence interval  $\mathbb{E}[X_3] \pm 21$ .

Fig. 9: Required number of iterations to collect at least  $N$  single-solutions.  $N = 8192$ .

*Acknowledgements.* We would like to thank Eamonn Postlethwaite for his detailed feedback and helpful suggestions on an earlier version of this paper.

Andre Esser was supported by DFG Research Training Group GRK 1817. Felix Heuer, Alexander May and Christian Sohler were supported by Mercator Research Center Ruhr, project ‘‘LPN-Krypt’’.

## A Las Vegas to Monte Carlo Boosting Technique

**Corollary A.1 (Corollary of Markov’s Inequality, [30]).** *Let  $X$  denote a non-negative random variable with expectation  $\mathbb{E}[X]$ . Then for any  $c > 0$  we have*

$$\Pr[X \geq c \cdot \mathbb{E}[X]] \leq \frac{1}{c} .$$

For a randomized algorithm  $A$  we let random variable  $T_A$  (resp.  $M_A$ ) denote the running time (resp. memory consumption) of  $A$  taken over its randomness.

**Corollary A.2 (Las Vegas to Monte-Carlo).** *Let  $A$  be a randomized algorithm that outputs a solution with probability 1. Then for any  $k \in \mathbb{N}$  there exists a randomized algorithm  $B := B_k$  with running time  $T_B \leq 4k \cdot \mathbb{E}[T_A]$ , memory consumption  $M_B \leq 4 \cdot \mathbb{E}[M_A]$  that outputs a solution with probability at least  $1 - 2^{-k}$ .*

*Proof.* Let  $k \in \mathbb{N}$ . We define algorithm  $B_k$  as follows: Run  $A$   $k$  times on fresh randomness and abort a run if  $T_A > 4 \cdot \mathbb{E}[T_A]$  or  $M_A > 4 \cdot \mathbb{E}[M_A]$ . If any run of  $A$  output a solution, output it. Otherwise, return a failure symbol  $\perp$ .

The running time and memory consumption of  $B$  are easy to verify. For  $B$ ’s failure probability we have

$$\begin{aligned} \Pr[B \text{ outputs } \perp] &= \Pr[\text{All } k \text{ runs of } A \text{ were aborted}] \\ &= \Pr[A \text{ was aborted}]^k \\ &= (\Pr[T_A > 4 \cdot \mathbb{E}[T_A] \cup M_A > 4 \cdot \mathbb{E}[M_A]])^k \\ &\leq (\Pr[T_A > 4 \cdot \mathbb{E}[T_A]] + \Pr[M_A > 4 \cdot \mathbb{E}[M_A]])^k \\ &\leq 2^{-k} , \end{aligned}$$

where we employed Corollary A.1 for the last bound.

## References

1. <http://csrc.nist.gov/groups/ST/post-quantum-crypto/>
2. Albrecht, M.R., Cid, C., Faugere, J.C., Fitzpatrick, R., Perret, L.: On the complexity of the bkw algorithm on lwe. *Designs, Codes and Cryptography* 74(2), 325–354 (2015)
3. Alekhnovich, M.: More on average case vs approximation complexity. In: 44th FOCS. pp. 298–307. IEEE Computer Society Press (Oct 2003)
4. Bai, S., Laarhoven, T., Stehlé, D.: Tuple lattice sieving. *LMS Journal of Computation and Mathematics* 19(A), 146–162 (2016)
5. Blum, A., Furst, M.L., Kearns, M.J., Lipton, R.J.: Cryptographic primitives based on hard learning problems. In: Stinson, D.R. (ed.) CRYPTO’93. LNCS, vol. 773, pp. 278–291. Springer, Heidelberg (Aug 1994)
6. Blum, A., Kalai, A., Wasserman, H.: Noise-tolerant learning, the parity problem, and the statistical query model. In: 32nd ACM STOC. pp. 435–440. ACM Press (May 2000)
7. Bogos, S., Tramèr, F., Vaudenay, S.: On solving L P N using B K W and variants - implementation and analysis. *Cryptography and Communications* 8(3), 331–369 (2016), <https://doi.org/10.1007/s12095-015-0149-2>
8. Bogos, S., Vaudenay, S.: Optimization of LPN solving algorithms. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016, Part I. LNCS, vol. 10031, pp. 703–728. Springer, Heidelberg (Dec 2016)
9. Boyer, M., Brassard, G., Høyer, P., Tapp, A.: Tight bounds on quantum searching. arXiv preprint quant-ph/9605034 (1996)
10. Devadas, S., Ren, L., Xiao, H.: On iterative collision search for LPN and subset sum. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017, Part II. LNCS, vol. 10678, pp. 729–746. Springer, Heidelberg (Nov 2017)
11. 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.) CRYPTO 2012. LNCS, vol. 7417, pp. 719–740. Springer, Heidelberg (Aug 2012)
12. Dohotaru, C., Hoyer, P.: Exact quantum lower bound for grover’s problem. arXiv preprint arXiv:0810.3647 (2008)
13. Ducas, L.: Shortest vector from lattice sieving: a few dimensions for free. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 125–145. Springer (2018)

14. Esser, A., Kübler, R., May, A.: LPN decoded. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part II. LNCS, vol. 10402, pp. 486–514. Springer, Heidelberg (Aug 2017)
15. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: 28th ACM STOC. pp. 212–219. ACM Press (May 1996)
16. Guo, Q., Johansson, T., Löndahl, C.: Solving LPN using covering codes. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part I. LNCS, vol. 8873, pp. 1–20. Springer, Heidelberg (Dec 2014)
17. Guo, Q., Johansson, T., Stankovski, P.: Coded-BKW: Solving LWE using lattice codes. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part I. LNCS, vol. 9215, pp. 23–42. Springer, Heidelberg (Aug 2015)
18. Herold, G., Kirshanova, E.: Improved algorithms for the approximate  $k$ -list problem in euclidean norm. In: Fehr, S. (ed.) PKC 2017, Part I. LNCS, vol. 10174, pp. 16–40. Springer, Heidelberg (Mar 2017)
19. Herold, G., Kirshanova, E., Laarhoven, T.: Speed-ups and time–memory trade-offs for tuple lattice sieving. In: PKC (2018)
20. Herold, G., Kirshanova, E., May, A.: On the asymptotic complexity of solving LWE. Des. Codes Cryptography 86(1), 55–83 (2018), <https://doi.org/10.1007/s10623-016-0326-0>
21. Howgrave-Graham, N., Joux, A.: New generic algorithms for hard knapsacks. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 235–256. Springer, Heidelberg (May 2010)
22. Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: 15th ACM STOC. pp. 193–206. ACM Press (Apr 1983)
23. Kaplan, M.: Quantum attacks against iterated block ciphers. CoRR abs/1410.1434 (2014), <http://arxiv.org/abs/1410.1434>
24. Kirchner, P., Fouque, P.A.: An improved BKW algorithm for LWE with applications to cryptography and lattices. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part I. LNCS, vol. 9215, pp. 43–62. Springer, Heidelberg (Aug 2015)
25. Laarhoven, T.: Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part I. LNCS, vol. 9215, pp. 3–22. Springer, Heidelberg (Aug 2015)
26. Laarhoven, T., Mariano, A.: Progressive lattice sieving. In: International Conference on Post-Quantum Cryptography. pp. 292–311. Springer (2018)
27. Laarhoven, T., de Weger, B.: Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In: Lauter, K.E., Rodríguez-Henríquez, F. (eds.) LATINCRYPT 2015. LNCS, vol. 9230, pp. 101–118. Springer, Heidelberg (Aug 2015)
28. Leveil, É., Fouque, P.A.: An improved LPN algorithm. In: Prisco, R.D., Yung, M. (eds.) SCN 06. LNCS, vol. 4116, pp. 348–359. Springer, Heidelberg (Sep 2006)
29. Lyubashevsky, V., Peikert, C., Regev, O.: A toolkit for ring-LWE cryptography. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 35–54. Springer, Heidelberg (May 2013)
30. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York, NY, USA (2005)
31. Newman, D.J., Shepp, L.: The Double Dixie Cup Problem. The American Mathematical Monthly 67(1) (1960), <http://www.jstor.org/stable/2308930>
32. Regev, O.: New lattice based cryptographic constructions. In: 35th ACM STOC. pp. 407–416. ACM Press (Jun 2003)
33. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM 56(6), 34:1–34:40 (2009), <http://doi.acm.org/10.1145/1568318.1568324>
34. Schroepel, R., Shamir, A.: A  $t=o(2^{n/2})$ ,  $s=o(2^{n/4})$  algorithm for certain np-complete problems. SIAM J. Comput. 10(3), 456–464 (1981), <https://doi.org/10.1137/0210033>
35. Wagner, D.: A generalized birthday problem. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer, Heidelberg (Aug 2002)
36. Zhang, B., Jiao, L., Wang, M.: Faster algorithms for solving LPN. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part I. LNCS, vol. 9665, pp. 168–195. Springer, Heidelberg (May 2016)