Dissection-BKW

Andre Esser[1(✉)], Felix Heuer[1], Robert Kübler[1], Alexander May[1], and Christian Sohler[2]

[1] Horst Görtz Institute for IT Security, Ruhr University Bochum, Bochum, Germany
{andre.esser,felix.heuer,robert.kuebler,alexander.may}@rub.de
[2] Department of Computer Science, TU Dortmund, 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 [4] and its generalization to arbitrary moduli, the Learning with Errors (LWE) problem [29], lie at the heart of our most promising coding-based and lattice-based post-quantum cryptographic constructions [2,26,28]. 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 [5] 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

H. Shacham and A. Boldyreva (Eds.): CRYPTO 2018, LNCS 10992, pp. 638–666, 2018.
https://doi.org/10.1007/978-3-319-96881-0_22

$\Theta(\frac{k}{\log k})$ 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 [21] takes time $2^{\mathcal{O}(n \log n)}$ with polynomial memory only, lattice sieving methods [12,22–24] 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 [3,18,19].

For BKW, the research so far mainly concentrated on run time, where many optimizations have been proposed in the cryptographic literature, e.g. [7,16,17,25,32]. While these improvements may significantly 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 h 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. [32] as an extension

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 $N_c := (M_{\mathrm{BKW}})^{\frac{\log c}{c-1}} = 2^{\frac{\log c}{c-1}\frac{k}{\log k}}$.

	c-sum Algorithm	Memory	Time	for		
classic	sorting (BKW)	N_2	N_2	$c = 2$		[5]
	Naive	N_c	${N_c}^{c-1}$	$c \geq 2$		Sect. 4.1
	Dissection	N_c	${N_c}^{c-\sqrt{2c}}$	$c = 4, 7, 11, 16, \ldots$		Sect. 5.2
	Tailored Dissection	${N_c}^{\alpha}$	${N_c}^{c-\alpha\sqrt{2c}}$	$c = 4, 7, 11, 16, \ldots$	$\alpha \in [1, \frac{\sqrt{c}}{\sqrt{c}-1}]$	Sect. 5.3
quantum	Naive + Grover	N_c	${N_c}^{c/2}$	$c \geq 2$		Sect. 4.2

$LF(k)$ of the BKW variants $LF1$ and $LF2$ by Levieil and Fouque [25], using Wagner's k-list algorithm [31].

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 $N_c := {M_{\mathrm{BKW}}}^{\frac{\log c}{c-1}}$ for any constant $c \in \mathbb{N}$, where $M_{\mathrm{BKW}} := 2^{\frac{k}{\log k}}$ 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 [10,30], 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^{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 Fig. 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^{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 Fig. 1 (triangles) for the resulting trade-off curve.

Dissection. We replace our naive c-sum algorithm by more advanced time-memory techniques like Schroeppel-Shamir [30] and its generalization, Dissection [10], to reduce the classic running time. We call the resulting algorithm Dissection-BKW. To give some illustrative results, with the Schroeppel-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 diamond at $\log M = \frac{2}{3}\frac{k}{\log k}$ in Fig. 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 diamond at $\log M \approx 0.47\frac{k}{\log k}$ in Fig. 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 diamonds in Fig. 1. Interestingly, when classically employing Schroeppel-Shamir we are on par (see point $(\frac{2}{3}, \frac{4}{3})$ in Fig. 1) with the complexities from the quantum trade-off as Schroeppel-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 (line segments depicted in Fig. 1) covering the sub-exponential memory regime entirely.

The full version of this work [13] also contains a discussion how our results translate from the LPN setting to LWE.

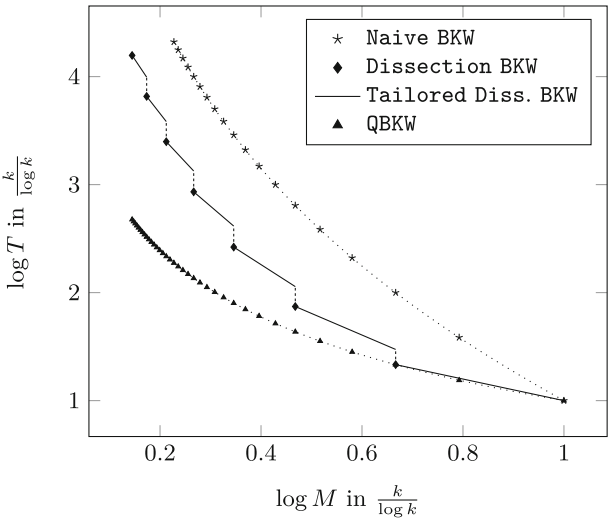


Fig. 1. Illustration of our BKW trade-offs. Instantiations exist exactly for marks as well as everywhere on solid lines. Naive BKW trade-off in stars (see Theorem 1.1), QBKW trade-off in triangles and Dissection-BKW trade-offs in diamonds and solid line segments (see Theorem 1.2).

2 Preliminaries

2.1 Notation

For $a \leq b \in \mathbb{N}$ let $[a, b] := \{a, a+1, \ldots, b\}$ 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* $L = (l_1, \ldots, l_i)$ is an element $L \in S^i$ and is of length $|L| = i$. We let $\emptyset$ 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 $L_1 = (l_1, \ldots, l_i)$ and $L_2 = (l_{i+1}, \ldots, l_j)$ we let $L_1 \cup L_2 := (l_1, \ldots, l_i, l_{i+1}, \ldots, l_j)$. 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 $\mathrm{low}_b(v)$. We let $\mathbf{u}_i$ denote the i^{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 Ber_p we denote the Bernoulli distribution, i.e., $X \sim \mathrm{Ber}_p$ means that $\Pr[X = 1] = 1 - \Pr[X = 0] = p$. For n independent random variables $X_1, \ldots, X_n \overset{iid}{\sim} \mathrm{Ber}_p$ their sum X is binomial distributed with parameters n and p, denoted $X \sim \mathrm{Bin}_{n,p}$. 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, [27]). *Let $X \sim \mathrm{Bin}_{n,p}$. Then*

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

2.2 The LPN Problem

Definition 2.1 ((Search) LPN Problem). *Let $k \in \mathbb{N}$, $\boldsymbol{s} \in \mathbb{F}_2^k$ and $p \in [0, \frac{1}{2})$ be a constant. Let* `Sample` *denote an oracle that, when queried, samples $\boldsymbol{a} \sim \mathcal{U}_{\mathbb{F}_2^k}$, $e \sim \mathrm{Ber}_p$ and outputs a* sample *of the form $(\boldsymbol{a}, b) := (\boldsymbol{a}, \langle \boldsymbol{a}, \boldsymbol{s} \rangle + e)$. The* $\mathtt{LPN}_k$ problem *consists of recovering $\boldsymbol{s}$ given access to* `Sample`*. In the following we call k the* dimension*, $\boldsymbol{s}$ the* secret*, p the* error rate*, b the* label *of $\boldsymbol{a}$ 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 $X_1, \ldots, X_n \sim \mathrm{Ber}_q$ independently. Then*

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

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

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

□

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, $j = 2, \ldots, k$ 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 $(1 - \exp(-k))^k \geq 1 - k \cdot \exp(-k) = 1 - \mathrm{negl}(k)$.

2.3 Combining Samples

In [5], 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

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

For such a 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 $e_1 \oplus e_2 \sim \mathrm{Ber}_{p'}$ 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 $e_i \sim \mathrm{Ber}_p$, $i \in [n]$ be identically, independently distributed. Then*

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

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^{th} *stripe*. Essentially, `c-sum-BKW` consists of a **for**-loop (line 4) that generates zeros (resp. the first unit vector) on the j^{th} stripe for $j \in [a-1]$ (resp. the a^{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-sum`(L, j, t) denote an algorithm that outputs a new list L where the coordinates of each entry match t on the j^{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^{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 $t \in \mathbb{F}_2^b$.

Algorithm 1. c-sum-BKW(k, p, ϵ_a, N) ▷ $\mathtt{c} \in \mathbb{N}$

Input: dimension k, error rate p, $\varepsilon_a > 0$, $N \geq 2^{\frac{b+c\log c+1}{c-1}}$, access to Sample
Output: $\mathbf{s} \in \mathbb{F}_2^k$

$a := \frac{\log k}{(1+\varepsilon_a)\log c}$, $b := \frac{k}{a}$, $n := \frac{8(1-p)k}{(1-2p)^{2c^a}}$
for $i \leftarrow 1, \ldots, n$ **do**
  Query N samples from Sample and save them in L.
  for $j \leftarrow 1, \ldots, a-1$ **do**
    $L \leftarrow \mathtt{c\text{-}sum}(L, j, 0^b)$
  $L \leftarrow \mathtt{c\text{-}sum}(L, a, \mathbf{u}_1)$
  if $L = \emptyset$ **then**
    return $\perp$
  Pick $(\mathbf{u}_1, b_i)$ uniformly from L.
$s_1 \leftarrow \text{majorityvote}(b_1, \ldots, b_n)$
Determine $s_2, \ldots, s_k$ the same way.
return $\mathbf{s} = s_1 \ldots s_k$

Definition 3.1 (The c-Sum-Problem (c-SP)). *Let $b, c, N \in \mathbb{N}$ with $c \geq 2$. Let $L := (l_1, \ldots, l_N)$ 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 $\mathcal{L} \in \binom{[N]}{c}$ 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 [6,7,32].

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 [9] 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 Sect. 6.1 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 Sect. 5.3 also requires $\alpha > 1$.

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

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

Then, under the Independence Heuristic, (L, t) has at least N^α 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 $X \sim \mathrm{Bin}_{\binom{N^\alpha}{c},2^{-b}}$ is binomially distributed. Hence,

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

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

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

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^α single-solutions solution using our Chernoff bound from Theorem 2.1 as

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

Observation 3.1. Clearly, 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). As a consequence 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*

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

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

Proof. Let $a = \frac{\log k}{(1+\varepsilon_a)\log c}$ 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^{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

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

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 $b_1,\ldots,b_n$ 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

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

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

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

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

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

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 $\tilde{\mathcal{O}}(\max\{N, a \cdot T_{c,N}\}) = \tilde{\mathcal{O}}(T_{c,N})$. Thus, for recovering the whole secret we get a running time of $\tilde{\mathcal{O}}(n \cdot T_{c,N})$. 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* $\tilde{\mathcal{O}}(M_{c,N}) = (M_{c,N})^{1+o(1)}$. The sample complexity of c-sum-BKW is $\tilde{\mathcal{O}}(knN) = \tilde{\mathcal{O}}(nN) = N^{1+o(1)}$.

It remains to remove the expectations from the complexity statements for time and 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 the full version [13]. □

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 $\tilde{\mathcal{O}}(N)$.
The following lemma shows correctness and the complexities of Algorithm 2.

Lemma 4.1. c-sum-naive *finds a solution of the* c-SP$_b$ *in time* $\tilde{\mathcal{O}}\left(N^{c-1}\right)$ *and memory* $\tilde{\mathcal{O}}(N)$.

Proof. See full version [13].

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.

Algorithm 2. `c-sum-naive`(L, t) ▷ $c \in \mathbb{N}$

Input: Sorted list $L = (v_1, \ldots, v_N) \in (\mathbb{F}_2^b)^N$, target $t \in \mathbb{F}_2^b$
Output: $S \subseteq \binom{[N]}{c}$ or $\perp$
1: **for all** $\mathcal{V} = \{i_1, \ldots i_{c-1}\} \subseteq [N]$ **do**
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: $S \leftarrow S \cup \{\{i_1, \ldots, i_c\}\}$
5: **if** $|S| = N$ **then**
6: **return** S
7: **return** $\perp$

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* $\mathtt{LPN}_k$ *problem with overwhelming success probability in time $T = 2^{\vartheta(1+\varepsilon)}$, using $M = 2^{\mu(1+\varepsilon)}$ memory and samples, where*

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

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)

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

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

Figure 1 shows the time-memory trade-off from Theorem 4.1 depicted by stars.

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 Donotaru and Høyer [11] shows that Grover's algorithm is optimal. We use a modification of Grover's algorithm due to [8], denoted `Grover`, that applies even in case the number of marked elements is unknown.

Theorem 4.2 (Grover Algorithm [8]). *Let $f\colon D \to \{0,1\}$ 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*

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

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^{th} entry in L, we now employ a Grover-search to immediately obtain $(c-1)$-sums for which there exists a suitable c^{th} element in L. Let us define the Grover function f_t as

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

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) ▷ `c` $\in \mathbb{N}$

Input: Sorted list $L = (v_1, \ldots, v_N) \in (\mathbb{F}_2^b)^N$, target $t \in \mathbb{F}_2^b$
Output: $S \subseteq \binom{[N]}{c}$ or $\perp$

1: **repeat** $\tilde{\mathcal{O}}(N)$ times
2: $\quad \mathcal{V} = (i_1, \ldots, i_{c-1}) \leftarrow$ `Grover`^{f_t}
3: $\quad$ **for all** $i_c \in [N] \setminus \mathcal{V}$ satisfying $v_{i_c} = t \oplus (\oplus_{i \in \mathcal{V}} v_i)$ **do**
4: $\quad\quad S \leftarrow S \cup \{\{i_1, \ldots, i_c\}\}$
5: $\quad\quad$ **if** $|S| = N$ **then**
6: $\quad\quad\quad$ **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* $\tilde{\mathcal{O}}\left(N^{c/2}\right)$ *and memory* $\tilde{\mathcal{O}}(N)$.

Proof. See full version [13].

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* `LPN`$_k$ *problem with overwhelming success probability in time $T = 2^{\vartheta(1+\varepsilon)}$, using $M = 2^{\mu(1+\varepsilon)}$ memory and samples, where*

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

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

The trade-off from Theorem 4.3 is depicted in Fig. 1 on Page 5 by triangles.

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 Sect. 5.1. As a consequence, we resort to the more advanced Dissection technique [10,30] in Sect. 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 Sect. 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* $\texttt{c-SP}_b$ *instance. Partition L into lists $L_1,\ldots,L_c$ of size $\frac{L}{c}$ each. A single-solution is* equally-split (wrt. $L_1,\ldots,L_c$) *if it corresponds to list elements $l_1,\ldots,l_c$ 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* $\texttt{c-SP}_b$ *instance with*

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

Then, under the Independence Heuristic, (L,t) has at least N^{α} 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, Eq. (2) is satisfied, and the rest follows as in the proof of Lemma 3.1. □

5.1 Meet-in-the-Middle and Beyond

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

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

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 $\tilde{\mathcal{O}}(\max(|L|,|L|^{\frac{c}{2}})) = \tilde{\mathcal{O}}(|L|^{\frac{c}{2}})$ 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`.

Schroeppel and Shamir's Meet-in-the-Middle. We present a heuristic simplification of the Schroeppel-Shamir algorithm [30] due to Howgrave-Graham and Joux [20].

In a nutshell, the idea of Schroeppel and Shamir is to run a meet-in-the-middle attack but impose an artificial constraint τ 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 τ over its whole domain. $L_1^{\frac{c}{2},\tau}$ is in turn built from smaller lists as follows. Let $t = v_{i_1} \oplus \ldots \oplus v_{i_c}$ and write

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

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 $\mathrm{low}_{\frac{c}{4}\log|L|}(\ell_{11} \oplus \ell_{12}) = \tau$ for some fixed τ. 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 $\tilde{\mathcal{O}}(|L|^{\frac{c}{4}})$. 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 $L_{2,1}^{\frac{c}{4}}$ and $L_{2,2}^{\frac{c}{4}}$ 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 $\tilde{\mathcal{O}}(\max\{|L|, |L|^{\frac{c}{4}}\})$.

As we iterate over $2^{|\tau|} = |L|^{\frac{c}{4}}$ choices of τ requiring expected time $\tilde{\mathcal{O}}(|L|^{\frac{c}{4}})$ per iteration, the overall expected running time is given by $\tilde{\mathcal{O}}(|L|^{\frac{c}{2}})$ 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 $\tilde{\mathcal{O}}(|L|)$.

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 Schroeppel-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 [31]. A Dissection algorithm is given c lists $L_c, \ldots, L_1$ 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 τ. For each τ:
- **(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 τ 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. Found 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 τ on some more bits.

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

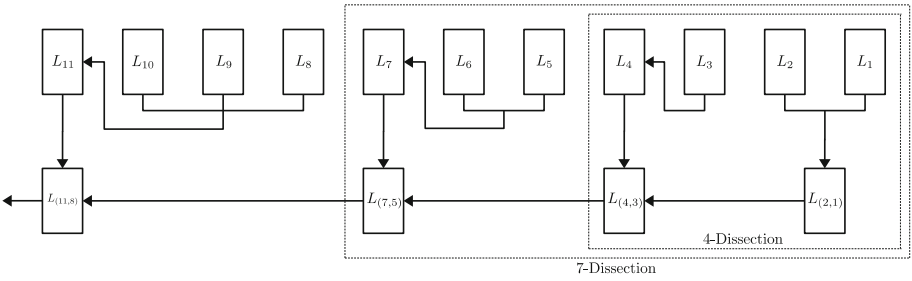


Fig. 2. High-level structure of an 11-Dissection on input lists $L_{11}, \ldots, L_1$. Recursively called 7- and 4-Dissection enclosed in dashed boxes. Arrows entering a list from the right side indicate a check on-the-fly. Arrows leaving a list on the left side indicate that a found match is returned 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 $L_{11}, \ldots, L_1$ and some target t. It loops through an artificial constraint τ_3. Within each iteration: A list $L_{(11,8)}$ containing sums $l_{11} \oplus \ldots \oplus l_8$ consistent with constraint τ_3 where $l_{11} \in L_{11}, \ldots, l_8 \in L_8$ is computed and stored. Then, still within the first iteration of the loop, a 7-Dissection is run (see Fig. 2) on lists $L_7, \ldots, L_1$ and a modified target $t \oplus \tau_3$. The 7-Dissection itself introduces a constraint τ_2 and stores a list $L_{(7,5)}$ containing sums of elements from L_7, L_6, L_5 fulfilling τ_2. It recursively calls a 4-Dissection, i.e. Schroeppel-Shamir (Sect. 5.1), on target $t \oplus \tau_3 \oplus \tau_2$. Internally, the 4-Dissection introduces another constraint τ_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 Fig. 2 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*

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

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

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

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 [10]). *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* $\mathsf{magic} := (c_i)_{i \in \mathbb{N}^{\geq 1}}$.

One easily verifies that, alternatively,

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

As Dissection formulated in the language of *k-list algorithms* might be of independent interest we deter from adding $\mathtt{c\text{-}SP}_b$-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

$$\mathtt{c_0\text{-}Dissect}(L_2, L_1, t, \mathtt{inner}) := L_2 \bowtie_t L_1 \ . \tag{7}$$

Here, "`inner`" indicates that $\mathtt{c_0}$-`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 $\mathtt{c_i}$-`Dissect` for $i \geq 1$. Lists are denoted by L and input-lists are numbered L_{c_i} down to L_1 for $c_i \in \mathsf{magic}$. 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 $L_j \bowtie \ldots \bowtie L_i$ 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 $\kappa = j, \ldots, i$.

The optimality of this recursive structure is shown in [10]. We now establish the required properties of Algorithm 4 in a series of lemmata. We give detailed proofs in the full version [13].

Algorithm 4. $\mathtt{c_i\text{-}Dissect}(L_{c_i}, \ldots, L_1, t, \mathsf{pos})$ ▷ $\mathtt{c_i} \in$ magic

Input: Lists $L_{c_i}, \ldots, L_1 \in (\mathbb{F}_2^b)^{2^\lambda}$ where $\lambda \leq \frac{b}{i}$, target $t \in \mathbb{F}_2^b$, $\mathtt{pos} \in \{\mathtt{outer}, \mathtt{inner}\}$
Output: $S \subseteq \binom{[N]}{c_i}$ or $\perp$

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

Algorithm 5. $\mathtt{c_i\text{-}sum\text{-}Dissect}(L, t)$ ▷ $\mathtt{c_i} \in$ 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 $L_{c_i}, \ldots, L_1$ of size $\frac{N}{c_i}$ each
2: $S \leftarrow \mathtt{c_i\text{-}Dissect}(L_{c_i}, \ldots, L_1, t, \mathtt{outer})$
3: **if** $|S| < N$ **then**
4: $\quad$ **return** $\perp$
5: **return** S

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

Proof. See full version [13].

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

Proof. See full version [13].

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

Proof. See full version [13].

Lemma 5.5. *Let $i \geq 1$. Algorithm $\mathtt{c_i\text{-}Dissect}(\cdots, \mathtt{inner})$ runs in expected time $\tilde{\mathcal{O}}(2^{c_{i-1} \cdot \lambda})$.*

Proof. See full version [13].

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

Proof. See full version [13].

Lemma 5.7. *Let $b \in \mathbb{N}$ and $c_i \in$ magic. For $t \in \mathbb{F}_2^b$ let (L, t) be an instance of the $\texttt{c}_\texttt{i}$-SP$_b$ where $|L| = N := 2^{\frac{b+c_i\cdot\log c_i+1}{c_i-1}}$. Under the Independence Heuristic $\texttt{c}_\texttt{i}$-sum-Dissect solves the $\texttt{c}_\texttt{i}$-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 $\texttt{c}_\texttt{i}$-Dissect recovers all of them. Note that the lists defined in line 1 are of length

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

One easily verifies that $\lambda = \frac{b+\log c_i+1}{c_i-1} \leq \frac{b}{i}$ as syntactically required by $\texttt{c}_\texttt{i}$-Dissect. Hence, Algorithm 5 solves the $\texttt{c-SP}_b$ with overwhelming probability.
From Lemma 5.3 we have

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

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

From Lemma 5.6 we conclude

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

Since $c_{i-1} = c_i - i - 1$ by Eq. (5) and $i \approx \sqrt{2c_i}$ by Eq. (6), we see that $\texttt{c}_\texttt{i}$-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 $\texttt{c}_\texttt{i}$-Dissect-BKW denote the variant of c-sum-BKW, where c-sum is instantiated using $\texttt{c}_\texttt{i}$-sum-Dissect.

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

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

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 $\texttt{c}_\texttt{i}$-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 $\texttt{c}_\texttt{i}$-Dissect-BKW. We have

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

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

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

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 $N^{1+o(1)} = M$. □

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 $(1 - \frac{i}{c_i-1})\cdot\log c_i$.

The trade-off from Theorem 5.1 is given in Fig. 1 on Page 5 as diamonds. Although it improves over the naive tradeoff from Sect. 4.1, this improvement comes at the price of covering only c_i-sums with $c_i \in$ magic. Given some available memory M, one would choose the minimal $c_i \in$ 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 $L_{c_i}, \ldots, L_1$, where $N = |L_{c_i}| + \ldots + |L_1|$. 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.[1]

[1] Imagine many single-solutions concentrated on few constraints τ_i as a counterexample.

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

$$\Pr\left[\sum_{j\in\mathcal{J}} Z_j < \frac{1}{\mathsf{poly}(\lambda)} \cdot \mathbb{E}\left[\sum_{j\in\mathcal{J}} Z_j\right]\right] \leq \mathsf{negl}(\lambda) \ . \tag{8}$$

In particular, it follows from Equation (8) that for all $\iota \leq 2^{i\lambda}$ we have

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

That is, we can compensate the deviation of the sums of Z_j below its expectation by iterating $\mathsf{poly}(\lambda)$ more often.[2]

As for the Independence Heuristic we ran experiments to verify the Tailoring Heuristic (Sect. 6.2).

Algorithm 6. $\mathtt{tailored\text{-}c_i\text{-}sum\text{-}Dissect}(L, t)$ ▷ $\mathtt{c_i} \in \mathsf{magic}$

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: $S \subseteq \binom{\lfloor N^\alpha \rfloor}{c_i}$ or $\perp$

1: Partition L into c_i lists $L_{c_i}, \ldots, L_1$ of size $2^\lambda := \frac{N^\alpha}{c_i}$ each
2: $S \leftarrow \mathtt{c_i\text{-}Dissect}(L_{c_i}, \ldots, L_1, t, \mathtt{outer})$ ▷ halt $\mathtt{c_i\text{-}Dissect}$ once $|S| = N^\alpha$
3: **if** $|S| < N^\alpha$ **then**
4: **return** $\perp$
5: **return** S

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

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^α for $\alpha \geq 1$ and the halt condition added as a comment in line 2.

Lemma 5.8 (Tailored Dissection). *Let $b \in \mathbb{N}, c_i \in \mathsf{magic}$ 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 $\mathtt{c_i\text{-}SP}_b$ 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 $\mathtt{c_i\text{-}SP}_b$ with overwhelming probability in expected time $T = \tilde{\mathcal{O}}\left(N^{c_{i-1} - i \cdot (\alpha - 1)}\right)$ and expected memory $M = \tilde{\mathcal{O}}\left(N^\alpha\right)$.*

Proof. See full version [13].

The complexities of `tailored-c_i-sum-Dissect` are given in Table 1 where we simplified α'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 $c_i \in \mathsf{magic}$ 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 $\mathtt{LPN}_k$ problem with overwhelming success probability in time $T = 2^{\vartheta(1+\varepsilon)}$, using $M = 2^{\mu(1+\varepsilon)}$ memory and samples, where*

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

Proof. See full version [13].

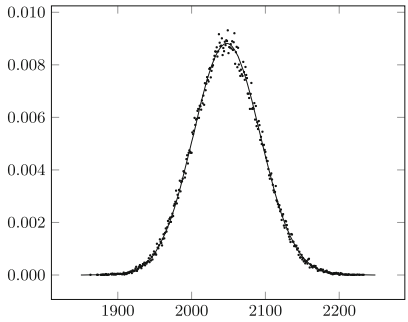
Intuitively, it is clear how to choose optimal parameters for `tailored-c_i-BKW` for any given amount of memory M: Find the minimal $c_i \in \mathsf{magic}$ such that `c_i-Dissect-BKW` uses at most memory M. Then, resort to `tailored-c_i-BKW` using memory M entirely by choosing the right α. The trade-off achieved via this approach is given in Fig. 1 on page 5 (line segments). A formal justification of the approach is contained in the full version [13].

6 Experimental Verification of Heuristics

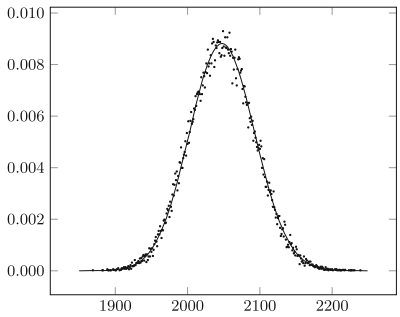
We present experimental results to verify our Independence Heuristic as well as our Tailoring Heuristic.

6.1 Experiments for the Independence Heuristic

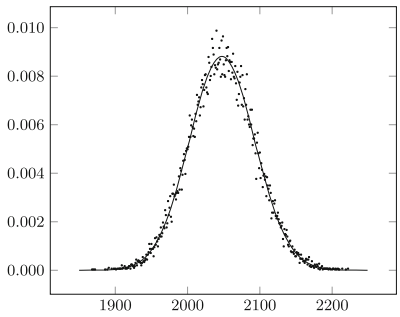
We tested the Independence Heuristic for $c_i \in \{4, 7\}$. We iteratively monitored the number of single-solutions found after a run of `c_i-sum-Dissect` on successive stripes starting with a list of size N. For each c_i we repeatedly called `c_i-sum-Dissect` on three stripes. After each call we stored the number of single-solutions found. To analyze the impact of dependencies amongst the



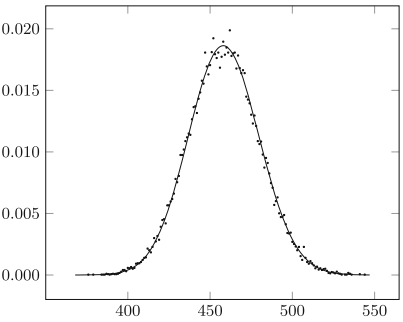
(a) Distribution of the number of single-solutions after run 1 of 4-sum-Dissect. Sample size in thousands: 108.



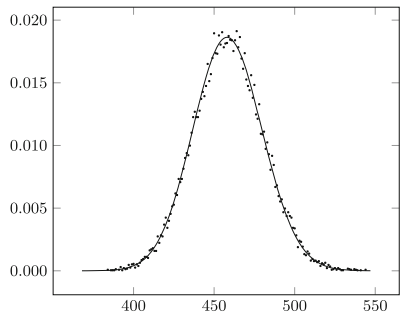
(b) Distribution of the number of single-solutions after run 2 of 4-sum-Dissect. Sample size in thousands: 100.



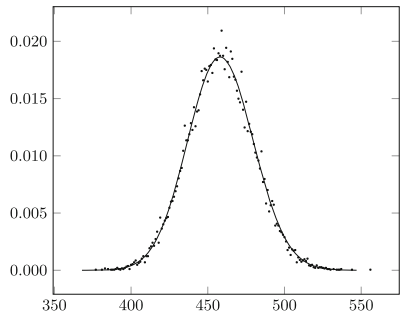
(c) Distribution of the number of single-solutions after run 3 of 4-sum-Dissect. Sample size in thousands: 58.



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



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



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

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

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 `c_i-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 $\mathrm{Bin}_{(N/c_i)^{c_i},2^{-b}}$ distributed after any run of `c_i-sum-Dissect`.

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

Table 2. Parameters for testing the Independence Heuristic. Parameter a denotes the number of stripes of width b.

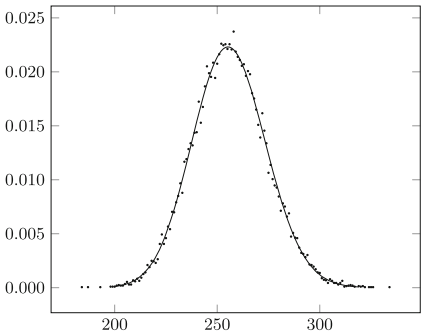
c_i	a	b	N	run	sample size in thous.	given in
				1	108	Fig. 3a
$c_1 = 4$	3	25	2048	2	100	Fig. 3b
				3	58	Fig. 3c
				1	29	Fig. 3d
$c_2 = 7$	3	33	441	2	23	Fig. 3e
				3	18	Fig. 3f

We see from Fig. 3 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.

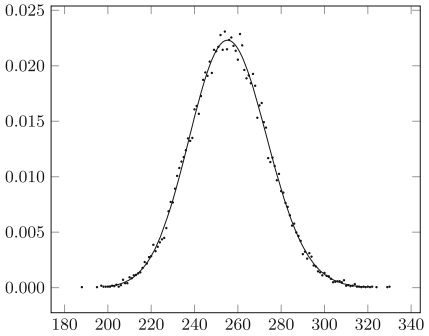
We also used the Independence Heuristic in Lemma 5.8 to show that the random variables Z_j of the number of single-solutions gained in the j^{th} iteration over a constraint is close to being binomially $\mathrm{Bin}_{x\cdot y,2^{-(b-i\lambda)}}$ distributed, where $x \sim \mathrm{Bin}_{2^{(i+1)\cdot\lambda},2^{-i\lambda}}$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

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

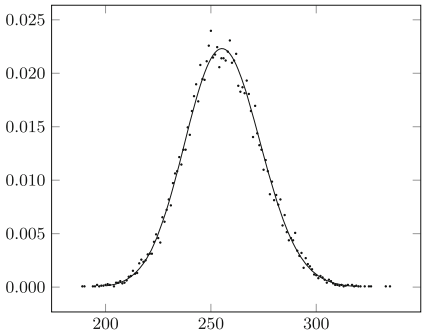
Each time we performed three consecutive runs of `4-sum-Dissect` on successive stripes and stored the number of single-solutions obtained per constraint after each iteration. The results are given in Fig. 4. Again, the obtained relative frequencies accurately match the benchmark curve $\mathrm{Bin}_{x\cdot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



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

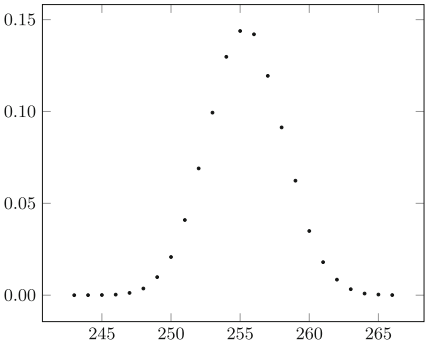
6.2 Experiments on the Tailoring Heuristic

As in the previous subsection, let Z_j be the number of single-solutions obtained per constraint iteration in $\texttt{c}_\texttt{i}$-`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.

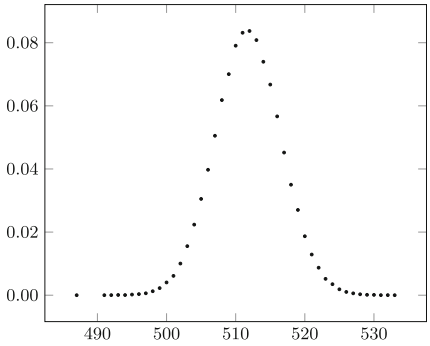
To test the Tailoring Heuristic we ran 4-`sum-Dissect` (without tailoring) on three stripes ($a = 3$) with $N = 8192$ and varying choice of b. We summed the numbers Z_j of single-solutions found per randomized constraint τ_j during the last run in a list, until their sum exceeded N. The results can be found in Table 3 and Fig. 5.

Table 3. Parameters and results for testing the Tailoring Heuristic for 4-`sum-Dissect`, $N = 8192$.

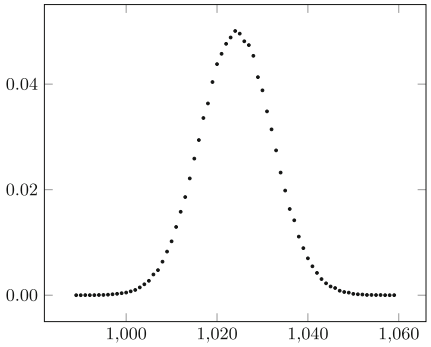
	$\mathbb{E}$[iterations to reach N]				
b	theory	experiments	99% confidence interval	sample size in thous.	given in
25	32	32.50	$32.50 \cdot (1 \pm 0.031)$	50	
28	256	255.40	$255.40 \cdot (1 \pm 0.031)$	100	Fig. 5a
29	512	511.78	$511.78 \cdot (1 \pm 0.026)$	150	Fig. 5b
30	1024	1024.20	$1024.20 \cdot (1 \pm 0.021)$	250	Fig. 5c



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



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



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

Fig. 5. Required number of iterations to collect at least N single-solutions. $N = 8192$.

We see in Table 3 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 been seen in Fig. 5. This all supports the validity of our Tailoring Heuristic for the analysis of Tailored Dissection BKW.

Acknowledgements. We would like to thank Eamonn Postlethwaite for his detailed feedback and helpful suggestions on an earlier version of this paper. We are grateful to the anonymous CRYPTO reviewers for their valuable comments.

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".

References

1. http://csrc.nist.gov/groups/ST/post-quantum-crypto/
2. Alekhnovich, M.: More on average case vs approximation complexity. In: 44th FOCS, pp. 298–307. IEEE Computer Society Press, October 2003
3. Bai, S., Laarhoven, T., Stehlé, D.: Tuple lattice sieving. LMS J. Comput. Math. **19**(A), 146–162 (2016)
4. Blum, A., Furst, M.L., Kearns, M.J., Lipton, R.J.: Cryptographic primitives based on hard learning problems. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 278–291. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-48329-2_24
5. 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
6. Bogos, S., Tramèr, F., Vaudenay, S.: On solving LPN using BKW and variants - implementation and analysis. Crypt. Commun. **8**(3), 331–369 (2016). https://doi.org/10.1007/s12095-015-0149-2
7. 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 (2016). https://doi.org/10.1007/978-3-662-53887-6_26
8. Boyer, M., Brassard, G., Høyer, P., Tapp, A.: Tight bounds on quantum searching. arXiv preprint quant-ph/9605034 (1996)
9. 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, Cham (2017). https://doi.org/10.1007/978-3-319-70503-3_24
10. 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 (2012). https://doi.org/10.1007/978-3-642-32009-5_42
11. Dohotaru, C., Hoyer, P.: Exact quantum lower bound for grover's problem. arXiv preprint arXiv:0810.3647 (2008)
12. Ducas, L.: Shortest vector from lattice sieving: a few dimensions for free. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part I. LNCS, vol. 10820, pp. 125–145. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78381-9_5

13. Esser, A., Heuer, F., Kübler, R., May, A., Sohler, C.: Dissection-BKW. Cryptology ePrint Archive, Report 2018/569 (2018). https://eprint.iacr.org/2018/569
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, Cham (2017). https://doi.org/10.1007/978-3-319-63715-0_17
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 (2014). https://doi.org/10.1007/978-3-662-45611-8_1
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 (2015). https://doi.org/10.1007/978-3-662-47989-6_2
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 (2017). https://doi.org/10.1007/978-3-662-54365-8_2
19. Herold, G., Kirshanova, E., Laarhoven, T.: Speed-Ups and time–memory tradeoffs for tuple lattice sieving. In: Abdalla, M., Dahab, R. (eds.) PKC 2018, Part I. LNCS, vol. 10769, pp. 407–436. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-76578-5_14
20. 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 (2010). https://doi.org/10.1007/978-3-642-13190-5_12
21. Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: 15th ACM STOC, pp. 193–206. ACM Press, April 1983
22. 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 (2015). https://doi.org/10.1007/978-3-662-47989-6_1
23. Laarhoven, T., Mariano, A.: Progressive lattice sieving. In: Lange, T., Steinwandt, R. (eds.) PQCrypto 2018. LNCS, vol. 10786, pp. 292–311. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-79063-3_14
24. 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, Cham (2015). https://doi.org/10.1007/978-3-319-22174-8_6
25. Levieil, É., Fouque, P.-A.: An improved LPN algorithm. In: De Prisco, R., Yung, M. (eds.) SCN 2006. LNCS, vol. 4116, pp. 348–359. Springer, Heidelberg (2006). https://doi.org/10.1007/11832072_24
26. 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 (2013). https://doi.org/10.1007/978-3-642-38348-9_3
27. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York (2005)
28. Regev, O.: New lattice based cryptographic constructions. In: 35th ACM STOC, pp. 407–416. ACM Press, June 2003
29. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM **56**(6), 34:1–34:40 (2009). https://doi.org/10.1145/1568318.1568324

30. Schroeppel, 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
31. Wagner, D.: A Generalized birthday problem. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 288–304. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45708-9_19
32. 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 (2016). https://doi.org/10.1007/978-3-662-49890-3_7