Asymptotics and Improvements of Sieving for Codes

Léo Ducas[1,2*], Andre Esser[3†], Simona Etinski[2*], and Elena Kirshanova[3,4‡]

[1] Centrum Wiskunde & Informatica, NL
{leo.ducas, simona.etinski}@cwi.nl
[2] Leiden University, NL
[3] Technology Innovation Institute, UAE
{andre.esser, elena.kirshanova}@tii.ae
[4] Immanuel Kant Baltic Federal University, Russia

Abstract. A recent work by Guo, Johansson, and Nguyen (Eprint'23) proposes a promising adaptation of Sieving techniques from lattices to codes, in particular, by claiming concrete cryptanalytic improvements on various schemes. The core of their algorithm reduces to a Near Neighbor Search (NNS) problem, for which they devise an ad-hoc approach. In this work, we aim for a better theoretical understanding of this approach. First, we provide an asymptotic analysis which is not present in the original paper. Second, we propose a more systematic use of well-established NNS machinery, known as Locality Sensitive Hashing and Filtering (LSH/F). LSH/F is an approach that has been applied very successfully in the case of sieving over lattices. We thus establish the first baseline for the sieving approach with a decoding complexity of $2^{0.117n}$ for the conventional worst parameters (full distance decoding, where complexity is maximized over all code rates). Our cumulative improvements eventually enable us to lower the hardest parameter decoding complexity for SievingISD algorithms to $2^{0.101n}$. This approach outperforms the BJMM algorithm (Eurocrypt'12) but falls behind the most advanced conventional ISD approach by Both and May (PQCrypto'18). As for lattices, we found the Random-Spherical-Code-Product (RPC) to give the best asymptotic complexity. Moreover, we also consider an alternative that seems specific to the Hamming Sphere, which we believe could be of practical interest as it plausibly hides less sub-exponential overheads than RPC.

1 Introduction

One of the central problems in coding theory is given as follows. Given a linear code, find a small codeword in this code. Concretely, given a parity check matrix

[*] Supported by ERC Starting Grant 947821 (ARTICULATE).

[†] Supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID MA 2536/12

[‡] Supported by the Young Russian Mathematics scholarship and by the Russian Science Foundation grant N 22-41-04411, https://rscf.ru/project/22-41-04411/

$\mathbf{H} \in \mathbb{F}^{(n-k)\times n}$ of a code of dimension k, length n, and defined over a field $\mathbb{F}$, find $\mathbf{e} \in \mathbb{F}^n$ such that

$$\mathbf{He} = 0 \quad \text{and} \quad |\mathbf{e}| < w$$

for some bound $0 \le w \le n$ and $|\cdot|$ is a metric defined over $\mathbb{F}$. In this work, we focus on the case where $\mathbb{F} = \mathbb{F}_2$ and $|\cdot|$ is the Hamming metric. Thus, we are interested in finding small Hamming weight codewords in a binary linear code. Specifically, we consider the case of random binary linear codes, i.e., $\mathbf{H}$ is chosen uniformly at random from $\mathbb{F}_2^{(n-k)\times n}$.

The problem of finding small Hamming weight codewords is a building block in all known efficient decoding algorithms for random linear codes. Information Set Decoding (ISD) algorithms [30,23], for example, construct such codewords by enumerating them in a clever way, while the Statistical Decoding approach [5] requires an oracle that returns a set of small weight codewords. In order to instantiate the oracle, [5] uses the above-mentioned ISD algorithms.

In the world of Euclidean lattices, a very similar problem occurs, namely, the problem of finding a short lattice vector in the Euclidean metric. For finding those short vectors there are (at least) two different established approaches. Concretely, there exist *enumeration*-based algorithms [14,18] as well as *sieving* algorithms [27,1,20]. While the former carefully prune the enumeration space, the latter saturate the space with many lattice vectors to the point where pairwise sums start producing short vectors. Drawing inspiration from sieving-based techniques in lattices, one can naturally ask:

Is there a sieving-type algorithm for finding small-weight codewords?

Given how natural this question is, it seems fair to assume that it has been investigated by various experts over the years. However, it was not until recently that the first satisfying answer was given by Guo, Johansson, and Nguyen [16] (GJN) in the form of their sieving-style ISD algorithm.

Any sieving algorithm (either for codes or for lattices) starts by generating a (large) list of vectors (either codewords or lattice vectors). A sieving step consists of finding a pair $\mathbf{e}, \mathbf{e}'$ from the list such that their sum produces a short(-er) vector. Codes resp. lattices are closed under addition, hence the newly produced vector is a codeword resp. a lattice vector, and is qualitatively better than the original elements from the list.

The sieving-style ISD approach from [16] now uses two key ingredients that differ from the lattice setting and make the sieving especially effective for finding short codewords. First, instead of applying the sieving technique to the full code, it is applied only to a subcode within the conventional ISD framework [15]. Essentially, the enumeration routine of the ISD procedure is substituted with a sieving-style algorithm for finding small codewords. The second main difference to the lattice setting is that, instead of starting with large codewords which become shorter through the sieving steps, the weight is kept equal throughout all sieving iterations. However, the "quality" of elements improves in each step as lists contain codewords from supercodes, where the codimension increases in each step until codewords eventually belong to the input code.

The fundamental task of finding a pair $\mathbf{e}, \mathbf{e}'$ that produces a short sum is called *the near neighbor problem* and has been extensively studied in various settings [17,24,28,1,8]. Concretely, if we denote by

$$\mathcal{S}_w^n \subset \mathbb{F}_2^n \text{ the set of binary vectors of weight } w,$$

we are interested in the following formulation of the problem.

Definition 1.1 **(w-Near Neighbor Search (informal)).** *Given a list of vectors $L \subset \mathcal{S}_w^n$ of weight w, find all pairs $\mathbf{x}, \mathbf{y} \in L^2$ s.t. $|\mathbf{x} + \mathbf{y}| = w$.*

Interestingly, this problem variant, where the input vectors all lie on the sphere $\mathcal{S}_w^n$, has not attracted much attention yet. It was studied in the context of *different input distributions* in [11]. It was shown there that the fastest known algorithms for a uniformly random list $L \subset \mathbb{F}_2^n$, without further tweaks, do not perform well in the case of fixed-weight input vectors. Recent works [4,9] studied a slightly more general version of the problem, where the input and output weight can differ. Esser [9] shows that advanced algorithms for this problem have the potential to improve the state of the art of ISD algorithms and provides a first algorithm for solving the problem. Carrier [4] provides advanced algorithms by showing how to efficiently adapt the concepts from the uniformly random input list case to the sphere. Most recently, in the context of the introduction of sieving-style ISD, GJN [16] specified a new algorithm for solving the w-near neighbor search used as a subroutine in the sieving step.

Here we see room for improvement and systematization: lattice sieving has benefited greatly from the Locality-Sensitive Filtering (LSF) framework, both in terms of clarity and efficiency. We study the translation of this framework to the Hamming case resulting in improved algorithms for the near neighbor search and, consequently, in improved SievingISD instantiations.

1.1 Our contributions

The contribution of this work is twofold. First, motivated by the relevance of the w-near neighbor search in the context of SievingISD [16] and in general ISD algorithms [9], we provide improved algorithms solving the problem from Definition 1.1. As this problem might be of independent interest, we provide those results in their full generality, allowing application in an arbitrary context. Our second contribution is to provide improved SievingISD instantiations based on these new near neighbor routines. In this context, we initiate the asymptotic study of the SievingISD framework and establish the asymptotic complexity exponent of the GJN algorithm. Further, we show that the new algorithms significantly improve the GJN running time and provide a comparison to the state-of-the-art of conventional ISD algorithms.

Near Neighbor Algorithms. In order to construct new algorithms solving the w-near neighbor search we formulate the Locality-Sensitive-Filtering framework in the Hamming metric; this framework is a generic method for solving the near neighbor problem and was originally proposed in the context of lattices [1] as a generalization of Locally-Sensitive-Hashing techniques [19]. We show how to adapt it to the Hamming metric and provide several concrete instantiations of this framework.

We also obtain the GJN algorithm as one of those instantiations. In this context we establish the asymptotic complexity of the GJN near neighbor algorithm, later serving as a foundation when analyzing its use in the SievingISD framework. We then give a series of algorithms resulting in significantly improved asymptotic complexities. The asymptotically fastest algorithm uses the most recent techniques based on Random Product Codes (RPCs). We were only recently pointed to an existing analysis of RPCs for the Hamming sphere in the Thesis of Carrier [4]. Because Carrier's Thesis [4] is only available in French we preferred to leave our own analysis in Section 4.2, but original credit should go to [4].

Moreover, we give an additional algorithm (Hash-opt) particular to the Hamming case which has high potential in practice: paying only a slight asymptotic penalty in comparison to RPC, it improves hidden sub-exponential factors considerably. In Fig. 1 we illustrate the complexity exponent ϑ, where the time complexity of the algorithms is equal to $|L|^{\vartheta}$, for varying weight and fixed list size. We compare the previous approaches of GJN (GJN) and Esser (Esser) against the fastest instantiation RPC-opt and the more practical instantiation Hash-opt, as well as a quadratic search baseline, corresponding to $\vartheta = 2$. It can be observed that the new algorithms improve the running time significantly for all weights.

SievingISD Instantiations. We study the asymptotics of SievingISD algorithms. We focus on the worst-case complexity in the full-distance decoding setting, the established measure for comparing the performance of decoding procedures. We establish the asymptotic worst case complexity of the GJN SievingISD algorithm as $2^{0.117n}$. This shows that the algorithm improves on Prange's original ISD algorithm [29] but, opposed to initial assumptions [16], falls behind the modern ISD algorithm by May, Meurer and Thomae (MMT) [23]. The new SievingISD instantiations based on RPC-opt and Hash-opt improve significantly by decreasing worst case complexity to $2^{0.1001n}$ and $2^{0.1007n}$ respectively. As illustrated in Fig. 2, this improvement is larger than the improvement made by any previous ISD algorithm over its predecessor.[1] Moreover, RPC-opt and Hash-opt improve drastically over the MMT algorithm and even slightly over the ISD algorithm by Becker, Joux, May, Meurer (BJMM) [2].

[1] Due to the chosen precision, Fig. 2 shows equality between Sisd-RPC-opt and Sisd-Hash-opt. However, in higher precision and for fixed rate Sisd-RPC-opt outperforms Sisd-Hash-opt.

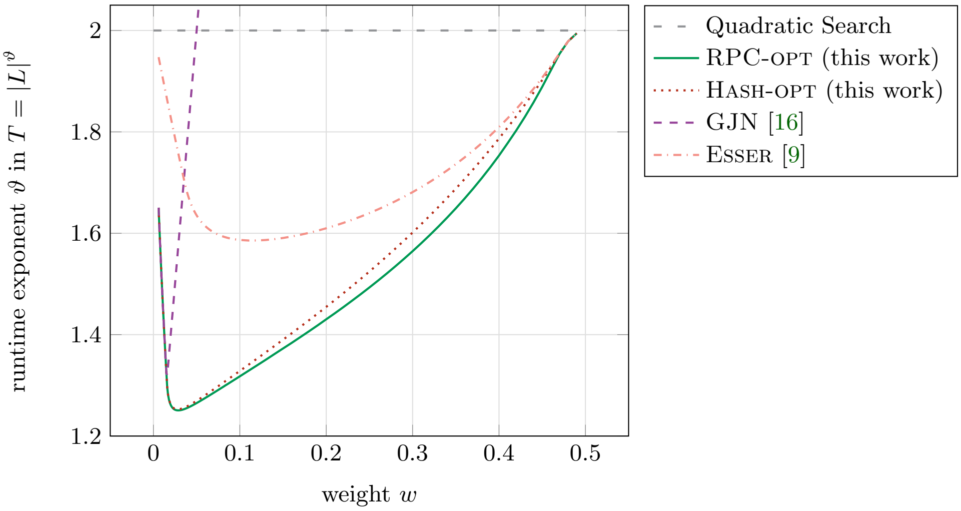


Fig. 1: Comparison of the running time of different algorithms solving the w-near neighbor search for fixed list size $|L| = 2^{0.05n}$.

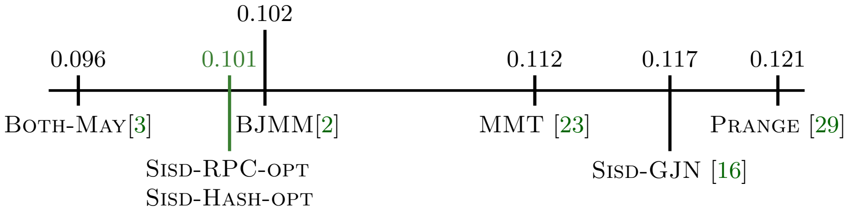


Fig. 2: Comparison of the asymptotic worst case runtime exponent c in the full distance setting for different SievingISD and conventional ISD algorithms. Runtime is of the form 2^{cn}.

Note that the conventional ISD algorithm by Both and May [3], which incorporates the algorithmic refinements of more than a decade, still has the lowest runtime exponent. However, we show that the recently introduced framework of SievingISD allows for competitive instantiations, already coming close to the best conventional ISD procedures. Moreover, practical applications usually resort to the MMT algorithm [12,13] due to lower overheads. We propose a practical SievingISD variant SISD-HASH-OPT which has a strong potential to lead to more efficient implementations, as it improves significantly on the MMT runtime.

In practical scenarios, memory is often limited, which puts a burden on ISD algorithms, SievingISD as well as conventional ISD, which require high amounts of memory for their enumeration subroutines. However, those algorithms are able to reduce the enumeration effort and with it the memory requirements at the cost of an increased runtime, resulting in a time-memory trade-off. In the extreme case of only a polynomial amount of memory being available, they interpolate to the running time of the original ISD algorithm by Prange. In Fig. 3, we compare the resulting time-memory trade-offs of Sisd-Hash-opt and Sisd-RPC-opt against those of Sisd-GJN and Both-May. Additionally, we compare against two recently proposed improvements of the MMT and BJMM trade-offs due to Esser and Zweydinger [13], labeled EZ-MMT and EZ-BJMM respectively.

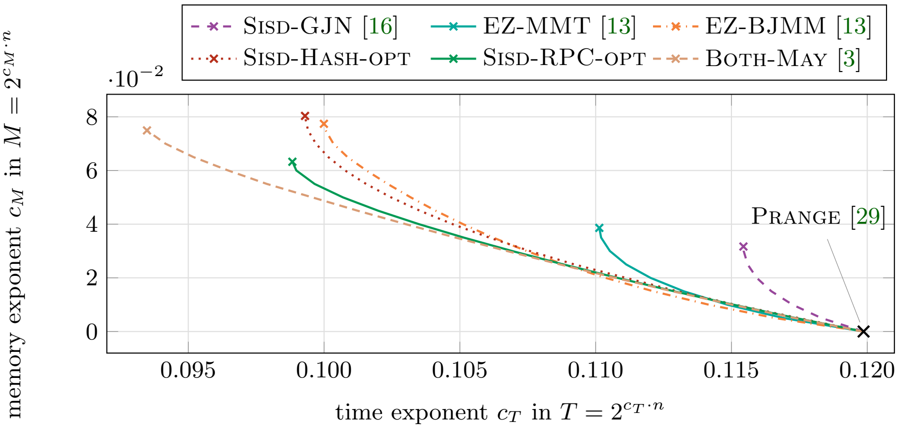


Fig. 3: Time-memory trade-off curves of SievingISD instantiations in comparison to conventional ISD trade-offs ($k = 0.5n$, full distance, i.e., $w \approx 0.11n$).

We observe that the new SievingISD instantiations outperform Sisd-GJN for all memory parameters. Moreover, the Sisd-RPC-opt trade-off behavior comes close to the one of Both-May for moderate amounts of memory. In terms of practical instantiations, we find that Sisd-Hash-opt outperforms the recent trade-offs by Esser and Zweydinger for any memory larger than $2^{0.015n}$ (EZ-MMT) or $2^{0.035n}$ (EZ-BJMM), respectively, further supporting its practical potential.

On heuristics. Our LSF algorithms that perform the near neighbor search do not rely on any heuristics. We rely on heuristics only when we apply these algorithms to solve the decoding problem. Note that, in the application to ISD, the input vectors to a near neighbor routine are not independent since they are constructed as pairwise sums of (potentially non-independent) vectors in the previous sieving step. Roughly speaking, we assume that the input list elements provided at any step behave like uniformly random and independent vectors from the sphere $\mathcal{S}_w^n \subset \mathbb{F}_2$ (for a more formal statement see Heuristic 1). However, we show in extensive experiments (see Section 6) that this building of iterative sums does not negatively influence the output list distribution. We note that exactly the same situation occurs in lattices: LSF-based near neighbor search techniques exhibit provable correctness and runtime [1], but efficient lattice sieving algorithms that rely on these LSF routines are heuristic. Moreover, similar assumptions arose in other contexts [31,21,10], which have later been substantiated by the corresponding proofs [25,7,22].

1.2 Technical Overview

This section aims to provide intuition and a simpler description of the algorithms following the LSF framework in the Hamming metric to solve the problem from Definition 1.1. Therefore, we omit some technical details (including Landau notations) for the sake of clarity. Rigorous descriptions and proofs are presented later in the main chapters.

The input contains a list $L \subset \mathcal{S}_w^n$ of uniform random and independent vectors. We denote by $|L| = N$ the list size. In the following, we call any pair $\mathbf{x}, \mathbf{y} \in L$ with $|\mathbf{x}+\mathbf{y}| = w$ a solution to the near neighbor search. Notice that $|\mathbf{x}\wedge\mathbf{y}| = w/2$ for $\mathbf{x}, \mathbf{y} \in \mathcal{S}_w^n$, implies that $\mathbf{x}, \mathbf{y}$ is a solution to the near neighbor search,[2] where $\wedge$ is applied coordinate-wise. Therefore, we can also search for pairs with a predefined coordinate-wise AND.

The idea of LSF is to apply a certain relation to list vectors such that if two vectors collide under this relation, they are likely to be a solution. Specifically, in LSF we create a set $\mathcal{C}_\mathrm{f} \subset \mathbb{F}_2^n$ of *filters* or *centers*[3] that divide the Hamming space into (possibly overlapping) regions. Each element $\mathbf{x} \in L$ is assigned to a filter $\mathbf{c}$ if and only if $|\mathbf{x} \wedge \mathbf{c}| = \alpha$ for some integer α. List elements assigned to the same $\mathbf{c}$ form a *bucket*:

$$\texttt{Bucket}_{\mathbf{c},\alpha} = \{\mathbf{x} \in L \colon |\mathbf{x} \wedge \mathbf{c}| = \alpha\}.$$

Note that if two uniform random vectors $\mathbf{x}, \mathbf{y}$ happen to be assigned to the same bucket, they have a certain (large) overlap in support (positions of 1's) with $\mathbf{c}$, so they are more likely to have overlap in support with each other. This principle lies at the heart of Algorithm 1, which is a simplified version of the more formal Algorithm 4, specified later.

To ease the description of the algorithm we introduce $\mathcal{B}_{\alpha,\mathbf{x}}$ – the set of *valid filters* which a fixed $\mathbf{x}$ was assigned to.

$$\mathcal{B}_{\alpha,\mathbf{x}} := \{\mathbf{c} \in \mathcal{C}_\mathrm{f} \colon |\mathbf{x} \wedge \mathbf{c}| = \alpha\}.$$

With that, the near neighbor search in Algorithm 1 consists of two steps: bucketing, which assigns each $\mathbf{x}$ to $\texttt{Bucket}_{\mathbf{c},\alpha}$ for $\mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{x}}$, and checking, which for each $\mathbf{x}$ searches for a matching element in $\texttt{Bucket}_{\mathbf{c},\alpha}$ for all $\mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{x}}$.

Notice that Algorithm 1 does not specify how $\mathcal{C}_\mathrm{f}$ should be chosen, nor the parameter α that determines the bucketing phase. By specifying these two inputs, we obtain an instantiation of Algorithm 1. Interestingly, the recent GJN

[2] Precisely, those pairs are guaranteed to be of distance *smaller* or equal to w. However, the overwhelming fraction is of distance exactly w.

[3] We use those terms interchangeably and even sometimes use the term *filter centers* to refer to the elements from the set $\mathcal{C}_\mathrm{f}$.

Algorithm 1: Near Neighbor Search (simplified)

```
Input  : $L \subseteq \mathcal{S}_w^n$,
         $\mathcal{C}_\mathrm{f}$ set of filter centers,
         $\alpha$ – parameter
Output: list $L'$ containing pairs $\mathbf{x}, \mathbf{y} \in L^2$ with $|\mathbf{x} + \mathbf{y}| = w$
BUCKETING PHASE:
for $\mathbf{x} \in L$ do
    Put $\mathbf{x}$ into $\texttt{Bucket}_{\mathbf{c},\alpha}$ $\forall c \in \mathcal{B}_{\alpha,\mathbf{x}}$
CHECKING PHASE:
$L' = \emptyset$
for $\mathbf{x} \in L$ do
    for $\mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{x}}$ do
        for $\mathbf{y} \in \texttt{Bucket}_{\mathbf{c},\alpha}$ do
            if $|\mathbf{x} \wedge \mathbf{y}| = w/2$ then
                store $(\mathbf{x}, \mathbf{y})$ in $L'$
return $L'$
```

approach [16] can be obtained as an instantiation of Algorithm 1 as we detail below. However, as we show next, other choices of $\mathcal{C}_\mathrm{f}$ and α lead to faster routines. In all the instantiations that we describe below, the following notations should be kept in mind

- ⋄ $N = |L|$ s.t. the expected number of solutions is N,
- ⋄ $F = |\mathcal{S}_\alpha^n| = \binom{n}{\alpha}$,
- ⋄ $P = |\mathcal{S}_\alpha^w| = \binom{w}{\alpha}$,
- ⋄ $D = \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}} \cap \mathcal{B}_{\alpha,\mathbf{y}}|\big]$ for some fixed pair $\mathbf{x}, \mathbf{y}$ s.t. $|\mathbf{x} + \mathbf{y}| = w$.

We describe the improvements in a progressive manner for didactic reasons starting at the GJN approach. In the later rigorous analysis in Section 4 we then skip certain, less effective variants. Whenever a variant has a counterpart in that section we specify the corresponding statement in parenthesis for fast reference.

Sieving by Guo-Johansson-Nguyen (Lemma 4.2). The main idea of the GJN sieving algorithm is to exploit the fact that for $\mathbf{x}, \mathbf{y}$ satisfying $|\mathbf{x}| = |\mathbf{y}| = w$ such that $|\mathbf{x}+\mathbf{y}| = w$, there exists a $\mathbf{c}$ of weight $w/2$ such that $|\mathbf{x} \wedge \mathbf{c}| = |\mathbf{y} \wedge \mathbf{c}| = w/2$. Moreover, given two vectors $\mathbf{x}, \mathbf{y}$ of weight w, the existence of such a $\mathbf{c}$ implies that $|\mathbf{x} + \mathbf{y}| \leq w$.

The GJN algorithm enumerates all those $\mathbf{c}$ and assigns $\mathbf{x}$ to a filter $\mathbf{c}$ if $|\mathbf{x} \wedge \mathbf{c}| = w/2$. In the context of Algorithm 1 this means the set of filters contains all vectors on the Hamming sphere of radius $w/2$, i.e., $\mathcal{C}_\mathrm{f} = \mathcal{S}_{w/2}^n$, and the bucketing parameter is chosen as $\alpha = w/2$. This implies that there are $|\mathcal{C}_\mathrm{f}| = F = \binom{n}{w/2}$ filters, while any vector $\mathbf{x} \in L$ is stored within $P = \binom{w}{w/2}$ buckets.

Let us now consider the runtime of this instantiation. From the above, the cost for the bucketing phase amounts to NP. For the cost of the checking phase, we note that this parametrization gives (almost) no false positives and no false negatives. Put differently, a pair of vectors found in the same bucket is (almost always)[4] of distance w and each pair of distance w is found. Furthermore, those pairs are found exactly once, i.e., there are no duplicate pairs, since the valid $\mathbf{c}$ is in fact unique as $\mathbf{c} = \mathbf{x} \wedge \mathbf{y}$. This implies that the cost of the checking phase is exactly the number of solutions, which, due to our choice of N, is N, giving time and memory

$$T = NP \quad \text{and} \quad M = T = NP.$$

Sieving with False-Positives. While the GJN algorithm fits the LSF framework, this LSF instantiation is just too restrictive. In particular, efficient LSF instantiations try to balance the cost of the bucketing and checking phases to minimize the time complexity. Usually, those instantiations give rise to false positives, that is, pairs ending up in the same bucket, but not being as close as desired. Those are then simply discarded during the checking phase.

To implement this idea, we change the parameters of the filters from having weight $w/2$ to any smaller value. In particular, we choose the centers $\mathbf{c}$ now on the α-sphere for $\alpha < w/2$, i.e., $\mathcal{C}_{\mathrm{f}} = \mathcal{S}^n_\alpha$. Note that this changes the amount of filters to $|\mathcal{C}_{\mathrm{f}}| = F = \binom{n}{\alpha}$, while each element $\mathbf{x} \in L$ can be found in $P = \binom{w}{\alpha}$ buckets. Finding all centers associated with a vector $\mathbf{x}$, i.e., all $\mathbf{c} \in \mathcal{C}_{\mathrm{f}}$ such that $|\mathbf{x} \wedge \mathbf{c}| = \alpha$ remains efficient, by simple subset enumeration.

Therefore the bucketing phase has cost NP as before (now for updated P) and on expectation, there are NP/F elements in each bucket as the probability that $\mathbf{x}$ lands in a certain bucket is P/F. The checking phase iterates for every list element over all elements in the associated buckets, which gives a total of $NP \cdot (NP/F) = (NP)^2/F$ checks. The overall complexity is summarized as

$$T = NP + (NP)^2/F \quad \text{and} \quad M = NP.$$

Note that this instantiation still does not allow for any false negative, meaning all pairs of distance w are detected. In fact, each such pair is detected by exactly $\binom{w/2}{\alpha}$ many centers.

Sieving with False Negatives. While it is optimal if every pair of distance w is detected exactly once, the previous instantiation detects any such pair $D = \binom{w/2}{\alpha}$ times. In the following, we therefore discard most of the bucket centers $\mathbf{c}$ randomly, only keeping a $1/D$ fraction of them. Then on expectation, every pair is still detected in one of the non-discarded buckets. This can be realized by defining the set $\mathcal{C}_{\mathrm{f}}$ to only include those centers for which $\mathcal{H}(\mathbf{c}) = 0$ for some random function $\mathcal{H}\colon \mathcal{S}^n_\alpha \to [D]$.

Since every list element $\mathcal{H}$ has to be evaluated for all P possible centers to determine which centers are valid, the cost of the bucketing phase remains unchanged. However, since the expected amount of considered filters is now only $|\mathcal{C}_{\mathrm{f}}| = F/D$, every element is found in only $P/D = \binom{w}{\alpha}/D$ different buckets,

[4] The *almost* is related to the fact that $|\mathbf{x} \wedge \mathbf{y}| = w/2$ actually implies $|\mathbf{x} + \mathbf{y}| \leq w$.

which reduces the cost of the checking phase and the memory consumption by a factor of D, resulting in

$$T = NP + (NP)^2/(DF) \quad \text{and} \quad M = NP/D. \tag{1}$$

Faster Sieving with False-Negatives (Theorem 4.3). Next, we mitigate the necessity of looping over all P possible centers in order to decide which centers belong to $\mathcal{C}_\mathrm{f}$ by specially crafting $\mathcal{H}$. Precisely, we craft $\mathcal{H}$ such that for a given $\mathbf{x}$ the set of valid centers $\mathcal{B}_{\alpha,\mathbf{x}}$ (of expected size P/D) can be computed in time less than P. For our concrete construction, consider a random binary linear code $\mathcal{C}_\mathcal{H}$ (independent from the original input code) with co-dimension $r \approx \log D$. With this code, define the hash function as follows

$$\mathcal{H}(\mathbf{c}) = 0 \iff \mathbf{c} \in \mathcal{C}_\mathcal{H}.$$

In turn, we expect only $1/2^r \approx 1/D$ random centers to evaluate to zero under this hash function.

Determining a valid center boils down to finding weight-α codewords in a random binary code. It might appear that we came back to the original problem of finding small-weight codewords, but it turns out that the effective length of the code $\mathcal{C}_\mathcal{H}$ is much smaller than n, hence the search for small-weight codewords is easier. In particular, denoting by $T_\mathtt{decode}$ the running time of finding weight-α codewords in $\mathcal{C}_\mathcal{H}$, the overall complexity of this sieving subroutine is

$$T = N \cdot (P/D + T_\mathtt{decode}) + (NP)^2/(DF) \quad \text{and} \quad M = NP/D. \tag{2}$$

Later in our formal analysis we use Prange's algorithm [29] to instantiate a decoder for $\mathcal{C}_\mathcal{H}$.

Repeating Faster Sieving with False Negatives (Corollary 4.1). In order to improve the memory of the above algorithm, we do not consider all filters at once but rather repeat bucketing and checking phases for smaller sets of size $|\mathcal{C}_\mathrm{f}| = F/(D \cdot R)$, for a repetition parameter R. Each individual run then uses less time and memory while after R repetitions we expect to find all the solutions. Concretely, we reduce the size of $|\mathcal{C}_\mathrm{f}|$ by choosing smaller codes $\mathcal{C}_\mathcal{H}$, with co-dimension $r \approx \log(DR)$, for the construction of $\mathcal{H}$. Since in each individual run, we consider only a $1/R$ fraction of the filters, the relevant expectations are reduced by that factor, leading to a time and memory complexity of

$$\begin{aligned} T &= R \cdot \left(N \cdot (P/(DR) + T'_\mathtt{decode}) + (NP)^2/(DRF)\right) \quad \text{and} \quad M = NP/(DR) \\ &= N \cdot (P/D + R \cdot T'_\mathtt{decode}) + (NP)^2/(DF) \quad \text{and} \quad M = NP/(DR). \end{aligned} \tag{3}$$

Here $T'_\mathtt{decode}$ denotes the time complexity to determine the set $\mathcal{B}_{\alpha,\mathbf{x}}$ for a given $\mathbf{x}$, corresponding to finding weight-α codewords in the, now smaller, code $\mathcal{C}_\mathcal{H}$. Note that interestingly this technique also allows decreasing the time complexity in comparison to Eq. (2), since by tweaking the code parameters we can ensure that over all executions we still reduce the overhead for finding valid centers, i.e., we ensure $R \cdot T'_\mathtt{decode} < T_\mathtt{decode}$.

Sieving with Random Product Codes (RPC) (Theorem 4.4). Finally, we describe a technique that does not introduce any asymptotic overhead for finding valid centers, i.e., we construct the set $\mathcal{B}_{\alpha,\mathbf{x}}$ for any element $\mathbf{x}$ in time $|\mathcal{B}_{\alpha,\mathbf{x}}|$.

The way we achieve this is by using *random product spherical codes.* We construct $\mathcal{C}_\mathrm{f} = \mathcal{C}_\mathrm{f}^{(1)} \times \mathcal{C}_\mathrm{f}^{(2)} \times \ldots \times \mathcal{C}_\mathrm{f}^{(t)}$ as the Cartesian product of t sets (or non-linear codes) $\mathcal{C}_\mathrm{f}^{(i)}$. The sets themselves contain a random selection of vectors of length n/t on the α/t-sphere, i.e., $\mathcal{C}_\mathrm{f}^{(i)} \subset \mathcal{S}_{\alpha/t}^{n/t}$. The cardinality of each set is $|\mathcal{C}_\mathrm{f}^{(i)}| = \sqrt[t]{F/D}$, such that overall $|\mathcal{C}_\mathrm{f}| = F/D$ centers are considered.

A list element $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_t)$ is now stored in the bucket associated with $\mathbf{c} = (\mathbf{c}_1, \ldots, \mathbf{c}_t)$. Interestingly, bucketing remains efficient because of the product structure. For an element $\mathbf{x} \in L$ first all partial centers $\mathbf{c}_1$ of $\mathcal{C}_\mathrm{f}^{(1)}$ are found that can be extended to valid bucket center, i.e., those with $|\mathbf{x}_1 \wedge \mathbf{c}_1| = \alpha/t$. Then the algorithm iteratively proceeds with the next partial center. Once all partial centers have been found, those are again combined product-wise to determine all buckets in which $\mathbf{x}$ has to be stored. Using this strategy together with a careful selection of parameters allows to decrease the cost of the bucketing phase to NP/D. In total the complexities become

$$T = NP/D + (NP)^2/(DF) \quad \text{and} \quad M = NP/D.$$

Smaller RPCs with Repetitions (Corollary 4.2). We again apply the technique of repeating the algorithm R times on smaller initial sets $|\mathcal{C}_\mathrm{f}| = F/(DR)$. However, since in comparison to Eq. (3) there is no overhead in finding valid centers involved, we fix the repetition amount to $R = P/D$. This leads to an optimal memory complexity that is linear in the initial list size N, while maintaining the same time complexity, which gives

$$T = NP/D + (NP)^2/(DF) \quad \text{and} \quad M = N.$$

Experiments. The source code for our experiments from Section 6 as well as the scripts used for the numerical optimization of the SievingISD instantiations are available at `https://github.com/setinski/Sieving-For-Codes`.

2 Preliminaries

We use non-bold letters for scalars, small bold letters for vectors, and capital bold letters for matrices. We denote by $\mathbb{F}_2$ the binary finite field and by $\mathbb{F}_2^n$ the corresponding vector space of dimension n. We use standard Landau notation, where $\tilde{\mathcal{O}}\left(\cdot\right)$ omits polylogarithmic factors. All logarithms are base 2. We define $H(\omega) := -\omega \log(\omega) - (1-\omega)\log(1-\omega)$ to be the binary entropy function. For a vector $\mathbf{x}$ we denote by $|\mathbf{x}| := |\{x_i \mid x_i \neq 0\}|$ the Hamming weight of $\mathbf{x}$, which counts the number of non-zero coordinates in $\mathbf{x}$. The sphere of radius w in $\mathbb{F}_2^n$ is defined as $\mathcal{S}_w^n := \{\mathbf{x} \in \mathbb{F}_2^n \mid |\mathbf{x}| = w\}$, which is of size $|\mathcal{S}_w^n| = \binom{n}{w}$.

Coding theory. A binary linear $[n, k]$ code $\mathcal{C}$ is a k-dimensional subspace of $\mathbb{F}_2^n$, where n is called its length and k its dimension. Such a code can be represented efficiently via a *parity-check matrix* $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$. The code $\mathcal{C}$ is then given as

$$\mathcal{C} := \{\mathbf{c} \in \mathbb{F}_2^n \mid \mathbf{Hc} = \mathbf{0}\}.$$

We make use of common transformations referred to as puncturing and shortening of codes.

Definition 2.1 (Code puncturing). *For a linear $[n, k]$ code $\mathcal{C}$ and a binary vector $\mathbf{x} \in \mathbb{F}_2^n$ with $|\mathbf{x}| = n'$ we define by $\pi_{\mathbf{x}} : \mathbf{c} \mapsto \mathbf{c} \wedge \mathbf{x}$ the puncturing function relative to the support of $\mathbf{x}$, and $\pi_{\mathbf{x}}(\mathcal{C})$ to be the corresponding punctured code.*

Note that if the support of $\mathbf{x}$ is an information set of $\mathcal{C}$, $\pi_{\mathbf{x}}$ is bijective (when implicitly restricted to $\mathcal{C}$), in which case we can define $\pi_{\mathbf{x}}^{-1}$ to return the unique pre-image in $\mathcal{C}$.

Definition 2.2 (Code shortening). *For a linear $[n, k]$ code $\mathcal{C}$ and a binary vector $\mathbf{x} \in \mathbb{F}_2^n$ with $|\mathbf{x}| = n'$ we define by $\sigma_{\mathbf{x}}(\mathcal{C}) := \{\mathbf{c} \in \mathcal{C} \mid \mathbf{c} \wedge \bar{\mathbf{x}} = \mathbf{0}\}$, the code shortened in the coordinates where $\mathbf{x}$ has support. Here $\bar{\mathbf{x}} := \mathbf{1} + \mathbf{x}$ is the bitwise complement of $\mathbf{x}$.*

Problem definitions. A central problem in coding theory underlying the security of many code-based primitives is the syndrome decoding problem, defined as follows.

Definition 2.3 (Syndrome Decoding Problem (SDP)). *Let $\mathcal{C} \subseteq \mathbb{F}_2^n$ be a linear $[n, k]$ code given via a parity check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$. Given a weight $w \in \mathbb{N}$ and a syndrome $\mathbf{s} \in \mathbb{F}_2^{n-k}$, find a vector $\mathbf{e} \in \mathcal{S}_w^n$ satisfying $\mathbf{He} = \mathbf{s}$.*

In the remainder of this work, we study the problem of codeword finding instead, given in the following definition.

Definition 2.4 (Codeword Finding Problem (CFP)). *Let $\mathcal{C} \subseteq \mathbb{F}_2^n$ be a linear $[n, k]$ code. Given a fixed weight $w \in \mathbb{N}$, find a vector $\mathbf{e} \in \mathcal{S}_n^w \cap \mathcal{C}$.*

We consider w to be linear in n, concretely for our asymptotic results, we choose w to match the Gilbert-Varshamov bound, i.e., $w = H^{-1}(1-k/n)n$, where $H^{-1}(\cdot)$ is the inverse of the binary entropy function on the interval $[0, 0.5]$. This guarantees that for both the SDP as well as the CFP the solution is unique.

Note that both problems are equivalent under the weight, length, and dimension preserving[5] polynomial reductions, implying that our results translate one-to-one to the SDP case. Observe that any solution $\mathbf{e}$ to codeword finding satisfies $\mathbf{He} = 0$ and, hence, forms a solution to SDP for syndrome $\mathbf{s} = \mathbf{0}$. Now, any SDP instance with solution $\mathbf{e}'$ defined by $\mathbf{H}, \mathbf{s}$ can be transformed into an

[5] Precisely, either length and weight or dimension increase by one depending on the chosen reduction.

instance $(\mathbf{H}', \mathbf{s}' = \mathbf{0})$, by letting $\mathbf{H}' = (\mathbf{H} \mid \mathbf{s})$. Now, this forms a CFP instance with increased weight $w+1$, length $n+1$, and solution $(\mathbf{e}', 1)$.

To solve those problems, ISD algorithms can be applied. The following lemma states the complexity of the original ISD algorithm by Prange to find all codewords of weight w in a given code.

Lemma 2.1 (Prange, [29]). *Given a binary linear $[n, k]$ code $\mathcal{C}$. Then for $w \leq n-k$, Prange's algorithm returns all weight w codewords in $\mathcal{C}$ in time $T = \tilde{\mathcal{O}}\left(\binom{n}{w} / \binom{n-k}{w}\right)$ and memory $M = \tilde{\mathcal{O}}(1)$.*

3 The Information Set Decoding (ISD) Framework

The information set decoding (ISD) framework consists of the following 3 steps. The first step samples $\mathbf{x} \in \mathcal{S}^n_{n'}$ and verifies if the dimension of the punctured code $\pi_{\mathbf{x}}(\mathcal{C})$ is equal to k. If that is the case, the support of $\mathbf{x}$ contains an information set of $\mathcal{C}$, and the algorithm continues.[6] In the second step, the algorithm computes N weight w' codewords of the punctured code $\pi_{\mathbf{x}}(\mathcal{C})$ using a sieving oracle. In the third and final step, the algorithm checks if any of these codewords (from the punctured code) yields a codeword of weight w in the original code when lifted using $\pi_{\mathbf{x}}^{-1}$. The procedure is detailed in Algorithm 2.

Algorithm 2: ISD

```
Input  : An $[n,k]$ code $\mathcal{C}$, parameters $w$ and $n' > k$. An oracle $\mathsf{O}$ returning $N$
         distinct uniformly random weight-$w'$ codewords in a given code.
Output: $\mathbf{e} \in \mathcal{C}$ such that $|\mathbf{e}| = w$
repeat
    Choose random $\mathbf{x} \in \mathbb{F}_2^n$ with $|\mathbf{x}| = n'$ and $\dim(\pi_{\mathbf{x}}(\mathcal{C})) = k$
    $L \leftarrow \mathsf{O}(\pi_{\mathbf{x}}(\mathcal{C}), w')$
    if $\exists\ \mathbf{y} \in L$: $|\pi_{\mathbf{x}}^{-1}(\mathbf{y})| = w$ then
        return $\pi_{\mathbf{x}}^{-1}(\mathbf{y})$
```

There are many ways to instantiate the oracle O in Algorithm 2, and we will refer to Algorithm 2 as SievingISD when this oracle is a sieving algorithm, that is an oracle as templated in Algorithm 3.

Theorem 3.1 (Complexity of ISD). *Let $\mathcal{C}$ be an $[n, k]$ code and $w \leq H^{-1}(1 - k/n)n$ be an integer. Let T_{O} and M_{O} be the expected time and the expected memory complexities of the oracle O used in Algorithm 2. Then Algorithm 2 returns a weight-w codeword in $\mathcal{C}$, if such exists, in expected time and memory*

$$T = \tilde{\mathcal{O}}\left((p_1 p_2)^{-1} \cdot T_{\mathsf{O}}\right) \quad \textit{and} \quad M = M_{\mathsf{O}},$$

[6] This happens at least with constant probability [6] so it will be omitted from the asymptotic analysis of the running time of the algorithm.

for any n', w' ensuring $p_2 \leq 1$, where

$$p_1 := \binom{n'}{w'}\binom{n-n'}{w-w'}\Big/\binom{n}{w} \quad \textit{and} \quad p_2 := N \cdot 2^{n'-k}\Big/\binom{n'}{w'}.$$

Proof. Note that $\mathbf{x}$ is chosen randomly and as long as $\mathbf{y}' := \pi_{\mathbf{x}}(\mathbf{e}) = w'$, we have $\mathbf{y}' \in \pi_{\mathbf{x}}(\mathcal{C}) \cap \mathcal{S}^{n'}_{w'}$, which implies that $\mathbf{y}'$ can be contained in L. Further, as long as $\dim(\pi_{\mathbf{x}}(\mathcal{C})) = k$, $\pi_{\mathbf{x}}$ is bijective which reveals $\mathbf{e} = \pi_{\mathbf{x}}^{-1}(\mathbf{y}')$, once $\mathbf{y}'$ is found.

Regarding the success probability of the algorithm, first, it must be the case that $\mathbf{e}$ has weight w' when projected onto the support of $\mathbf{x}$, that is $|\mathbf{e} \wedge \mathbf{x}| = w'$. This happens with probability

$$p_1 = \binom{n'}{w'}\binom{n-n'}{w-w'}\Big/\binom{n}{w}$$

If this first condition is fulfilled, we need to consider whether $\mathbf{e}\wedge\mathbf{x}$ is included in L. Note that there are on expectation $\binom{n'}{w'}/2^{n'-k}$ codewords of weight w' in $\pi_{\mathbf{x}}(\mathcal{C})$. The probability that $\mathbf{e}\wedge\mathbf{x}$ is included in a list of size N sampled uniformly at random from the set of small codewords is therefore

$$p_2 = 1 - \left(1 - 2^{n'-k}\Big/\binom{n'}{w'}\right)^N = \Theta\left(N \cdot 2^{n'-k}\Big/\binom{n'}{w'}\right). \tag{4}$$

Here, the last equality follows from the fact that for the oracle to be feasible it must hold

$$N \leq \binom{n'}{w'} \cdot 2^{k-n'}, \tag{C 1}$$

i.e., there must exist N distinct codewords of weight w' in $\pi_{\mathbf{x}}(\mathcal{C})$. Note that this inequality translates to $p_2 \leq 1$.

The time complexity of the algorithm is the time per iteration divided by the success probability. We already saw that the success probability is $p_1 p_2$, while one iteration is dominated by the time it takes to query the oracle, which is T_{O}, resulting in the claimed running time. Besides the list L, the algorithm stores only elements of polynomial size, therefore the memory complexity is equal to the memory complexity of the oracle, which is at least N. □

The Sieving Subroutine Algorithm 2 has access to an oracle O returning N distinct weight-w' codewords for a given code. In our work, the oracle is instantiated using the *sieving* routine detailed in Algorithm 3.

This routine starts with an arbitrary list of small-weight words of length n', i.e., a list $L \subset \mathbb{F}_2^{n'} = \mathcal{C}_0$. Note that choosing small-weight words from $\mathcal{C}_0$ is efficient. The algorithm proceeds iteratively using a tower of codes $\mathcal{C}_0 \subset \mathcal{C}_1 \subset \ldots \subset \mathcal{C}_{n'-k} = \mathcal{C}'$. In each iteration i a new list of short codewords belonging to code $\mathcal{C}_i$ is constructed from sums of elements of the current list; until in iteration $n'-k$ the constructed list finally contains codewords from $\mathcal{C}' = \mathcal{C}_{n'-k}$. A possible choice for the tower of codes is, for example, $\mathcal{C}_i$'s whose parity-check matrix consists of the first i rows of the parity-check matrix from $\mathcal{C}'$.

```
Algorithm 3: O: Sieving
Input  : $[n', k]$-code $\mathcal{C}'$, $N$ and $w'$.
Output: set $L = \{\mathbf{e} \in \mathcal{C}' : |\mathbf{e}| = w'\}$ with $|L| = N$
Choose a tower of codes $\mathbb{F}_2^{n'} = \mathcal{C}_0 \subset \mathcal{C}_1 \subset \cdots \subset \mathcal{C}_{n'-k} = \mathcal{C}'$, with dimension decrements of 1.
Choose $N$ random distinct vectors of $\mathbb{F}_2^{n'}$ of weight $w'$ as initial set $L$
for $i = 1$ to $n' - k$ do
    $L \leftarrow \{\mathbf{x} + \mathbf{y}$ s.t. $|\mathbf{x} + \mathbf{y}| = w'$ and $(\mathbf{x}, \mathbf{y}) \in L^2\} \cap \mathcal{C}_i$
    Discard some elements if $|L| > N$
Return $L$
```

For constructing the list in iteration i we first apply a *near neighbor* subroutine, which finds all words of weight w' that can be constructed via pairwise sums from the current list; after which we filter the list for codewords belonging to the current code $\mathcal{C}_i$.

Maintaining the list size. Note that, since all used codes $\mathcal{C}_i$ are linear and two subsequent codes' dimension differs by one, the filtering discards on expectation half of the constructed elements. Through all iterations, we aim at maintaining a steady list size of N, by discarding elements if necessary. Therefore, the list L must be large enough so that at least N many pairs $(\mathbf{x}, \mathbf{y}) \in L^2$ sum to short vectors. Accounting for a loss of half of the vectors, since on expectation $\#\mathcal{C}_i / \#\mathcal{C}_{i+1} = 2$, and for the fact that we take every pair twice, this requires

$$N \geq 4 \cdot \binom{n'}{w'} \Big/ \binom{w'}{w'/2} \binom{n' - w'}{w'/2} \tag{C 2}$$

In the following, we choose N up to a constant factor equal to this lower bound.

Finding Short Sums. In the application of the near neighbor routine to ISD we rely on a certain heuristic, common to the sieving setting. Informally, we treat elements contained in the lists in each iteration as independently and uniformly sampled from the w'-sphere $\mathcal{S}_{w'}^{n'}$. This allows us to study algorithms that solve the *w-near neighbor search* to construct L. This problem is defined as follows.

Definition 3.1 (w-Near Neighbor Search). *Given a list of uniformly and independently distributed vectors $L \subset \mathcal{S}_w^n$ of weight w with $|L| = N$, find a $(1 - o(1))$-fraction of pairs $\mathbf{x}, \mathbf{y} \in L^2$ s.t. $|\mathbf{x} + \mathbf{y}| = w$.*

We refer to this problem as $\mathtt{NNS}(N, n, w)$ while we refer to (L, n, w) as an instance of the problem.

Our heuristic assumption is that the time and memory complexity of algorithms solving the w-near neighbor search is only mildly affected by the dependencies between list elements if constructed as pairwise sums over multiple iterations as in Algorithm 3. We formalize this in the following heuristic.

Binary-Sieve Heuristic *Let $n' \in \mathbb{N}$, κ, ω, λ be positive constants. Let $k = \kappa n'$, $w' = \omega n'$ and $|L| = N = 2^{\lambda n'}$ satisfying Constraints Eq. (C 1) and Eq. (C 2). Then, we assume that:*

1. *The running time and memory complexity of any algorithm applied to the near neighbor search instance (L, n', w') for L from Line 4 of Algorithm 3 is at most affected by a factor of $2^{o(n')}$ in comparison to L being sampled uniformly and independently from $\mathcal{S}_{w'}^{n'}$.*
2. *The probability of any element being present in the finally returned list L in Line 6 of Algorithm 3 is up to a $2^{o(n')}$ factor equal to the probability that $L \subset C' \cap \mathcal{S}_{w'}^{n'}$ is drawn uniformly at random. Formally, $\Pr\left[\mathbf{c} \in L \mid \mathbf{c} \in \mathcal{C}' \cap \mathcal{S}_{w'}^{n'}\right] \geq p_2/2^{o(n')}$ for p_2 from Eq. (4).*

The first part of the heuristic ensures that we can use algorithms solving the w-near neighbor search in order to construct the list L in each iteration. The second part is necessary to ensure that the success probability (see Eq. (4)) of Algorithm 2 is not significantly impacted and the runtime statement of Theorem 3.1 remains valid when instantiating the oracle with Algorithm 3.

Note that an analogous heuristic is used by lattice sieving algorithms [1, Section 7]. Also in the binary case, heuristics about the mild effect of stochastic dependencies from iterative sums are commonly used as, for example, in the context of Learning Parity with Noise (LPN) [21,10], the generalized birthday problem (GBP) [31] or even by other ISD algorithms [23,2,24]. Those heuristics have been put to the test experimentally [10,12] and most of them have been proven in later works [25,7,22]. In addition, we provide experiments verifying the heuristic in our precise context in Section 6.

Relying on this heuristic, the running time of the oracle (Algorithm 3) is asymptotically equal to the time required to solve the w-near neighbor search. We summarize this in the following theorem.

Theorem 3.2 (Complexity of the Sieving Oracle). *Let $n' \in \mathbb{N}$, κ, ω, λ be positive constants. Let $k = \kappa n'$, $w' = \omega n'$ and $|L| = N = 2^{\lambda n'}$ satisfying Constraints Eq. (C 1) and Eq. (C 2). Further, let T_{NNS} and M_{NNS} be the time and memory complexities to solve the $\mathtt{NNS}(N, n', w')$ Then, under the* Binary-Sieve Heuristic *the time and memory complexity of Algorithm 3 is*

$$T_{\mathsf{O}} = \tilde{\mathcal{O}}\left(T_{\mathrm{NNS}}\right) \quad \text{and} \quad M_{\mathsf{O}} = \tilde{\mathcal{O}}\left(M_{\mathrm{NNS}}\right).$$

Proof. Note that N is exponential in n'. Therefore, the running time of Algorithm 3 is dominated by the construction of the list in Line 4. Under the Binary-Sieve Heuristic the running time to construct this list is $\tilde{\mathcal{O}}\left(T_{\mathrm{NNS}}\right)$ and the memory needed is $\tilde{\mathcal{O}}\left(M_{\mathrm{NNS}}\right)$. □

Theorem 3.2 motivates the further study of algorithms to solve the w-near neighbor search problem in the following section. Later in Section 5 we study the performance of SievingISD, i.e., Algorithm 2 in combination with Algorithm 3, instantiated using those near neighbor search routines.

4 Near Neighbor Search in the Hamming Metric

In this section, we present different algorithms solving the w-near neighbor search from Definition 3.1. We first recall the general locality sensitive hashing (LSH) or locality sensitive filter (LSF) framework for near neighbor search – a framework that forms the basis for many of the best known algorithms to find near neighbors [24,1,11]. We then show that the recently presented algorithm by Guo-Johansson-Nguyen [16] already falls into this framework. We proceed by presenting and analyzing different improvements.

Locality Sensitive Filtering. Define first a set $\mathcal{C}_{\mathrm{f}} \subset \mathbb{F}_2^n$ of filter vectors $\mathbf{c}$ that divide the Hamming space into regions. Concretely, for an integer α let

$$\mathrm{Region}_{\mathbf{c},\alpha} = \{\mathbf{x} \in \mathbb{F}_2^n \ : \ |\mathbf{x} \wedge \mathbf{c}| = \alpha\}. \tag{5}$$

Notice that for a sufficiently large α, two vectors that lie in the same region have large overlapping support with a fixed vector $\mathbf{c}$, hence their sum has a high chance of being of small Hamming weight. A *bucket* associated to center $\mathbf{c}$ is defined as $\mathrm{Bucket}_{\mathbf{c},\alpha} = \mathrm{Region}_{\mathbf{c},\alpha} \cap L$.

The idea of LSH/F is to assign all vectors from L to $\mathrm{Bucket}_{\mathbf{c},\alpha}, \forall \mathbf{c} \in \mathcal{C}_{\mathrm{f}}$. Therefore, for all $\mathbf{x} \in L$, we first find all *valid filters* defined as the set

$$\mathcal{B}_{\alpha,\mathbf{x}} := \{\mathbf{c} \in \mathcal{C}_{\mathrm{f}} \colon |\mathbf{x} \wedge \mathbf{c}| = \alpha\}.$$

For a fixed $\mathbf{x}$, the procedure of determining and returning those valid filters is called `ValidFilters` in Algorithm 4. We denote the step of assigning all elements to each of its buckets as *bucketing phase.* Subsequently, the search for close pairs is carried out only within each $\mathrm{Bucket}_{\mathbf{c},\alpha}$, which we refer to as *checking phase.*

Complexity of Algorithm 4 Let us give a general lemma stating the complexity and correctness of Algorithm 4 to which we refer in our later analyses.

Lemma 4.1 (Complexity of Algorithm 4). *Let $\mathbf{x}, \mathbf{y}$ be s.t. $|\mathbf{x}| = |\mathbf{y}| = |\mathbf{x} + \mathbf{y}| = w$ and $T_{\texttt{ValidFilters}}$ denote the time to compute the set $\mathcal{B}_{\alpha,\mathbf{x}}$ for any given $\mathbf{x} \in L$. Then Algorithm 4 returns a list containing $\mathbf{x}, \mathbf{y}$ in expected time T and expected memory M, where*

$$T = \tilde{\mathcal{O}}\left(N \cdot (T_{\texttt{ValidFilters}} + \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}|\big] \cdot \mathbb{E}\big[|\mathrm{Bucket}_{\mathbf{c},\alpha}|\big])\right) \ \textit{and}$$
$$M = \tilde{\mathcal{O}}\left(N \cdot \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}|\big]\right)$$

with probability $q := \Pr\left[\exists \mathbf{c} \in \mathcal{C}_{\mathrm{f}} \colon \mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{x}} \cap \mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{y}}\right]$ whenever $\mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}|\big] \geq 1$.

Proof. Note that the algorithm recovers a w-close pair $\mathbf{x}, \mathbf{y}$ whenever there is a $\mathbf{c} \in \mathcal{C}_{\mathrm{f}}$ for which $\mathbf{c}$ is a valid filter for both, $\mathbf{x}$ and $\mathbf{y}$. More formally, a w-close pair $\mathbf{x}, \mathbf{y}$ whenever $\exists \mathbf{c} \in \mathcal{C}_{\mathrm{f}} \colon \mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{x}}$ and $\mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{y}}$ since, in that case, $\mathbf{x}$ is stored in $\mathrm{Bucket}_{\mathbf{c},\alpha}$ in the bucketing phase, while $\mathbf{y}$ checks $\mathrm{Bucket}_{\mathbf{c},\alpha}$ in the checking phase for close pairs.

Algorithm 4: Near Neighbor Search

```
Input  : NNS$(L, n, w)$ instance where $L \subseteq \mathcal{S}_w^n$, description of the set $\mathcal{C}_\mathrm{f}$ of
         bucket centers $\mathbf{c}$, bucketing parameter $\alpha$
Output: list $L'$ containing pairs $\mathbf{x}, \mathbf{y} \in L^2$ with $|\mathbf{x} + \mathbf{y}| = w$
BUCKETING PHASE:
for $\mathbf{x} \in L$ do
    for $\mathbf{c} \in$ ValidFilters$(\mathcal{C}_\mathrm{f}, \mathbf{x}, \alpha)$ do
        store $\mathbf{x}$ in Bucket$_{\mathbf{c},\alpha}$
CHECKING PHASE:
$L' = \emptyset$
for $\mathbf{x} \in L$ do
    for $\mathbf{c} \in$ ValidFilters$(\mathcal{C}_\mathrm{f}, \mathbf{x}, \alpha)$ do
        for $\mathbf{y} \in$ Bucket$_{\mathbf{c},\alpha}$ do
            if $|\mathbf{x} \wedge \mathbf{y}| = w/2$ then
                store $(\mathbf{x}, \mathbf{y})$ in $L'$
return $L'$
```

The running time of the algorithm is dominated by the checking phase. The bucketing can be performed in the expected time

$$T_{\text{Bucket}} = \tilde{\mathcal{O}}\left(N \cdot T_{\texttt{ValidFilters}}\right),$$

where $T_{\texttt{ValidFilters}}$ is the expected time to retrieve the set $\mathcal{B}_{\alpha,\mathbf{x}}$ for a fixed $\mathbf{x}$ via the `ValidFilters` function. The checking phase performs the same identification of valid centers. Additionally, for all returned valid centers it explores the corresponding bucket to find w-close pairs. Note that this exploration can be performed in time linear in the size of the bucket. Hence, we have

$$T_{\text{Check}} = T_{\text{Bucket}} + \tilde{\mathcal{O}}\left(N \cdot \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}|\big] \cdot \mathbb{E}\big[|\text{Bucket}_{\mathbf{c},\alpha}|\big]\right).$$

Note that the expected bucket size is given by

$$\mathbb{E}\big[|\text{Bucket}_{\mathbf{c},\alpha}|\big] = \frac{N \cdot \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}|\big]}{|\mathcal{C}_\mathrm{f}|}, \tag{6}$$

since there are expected $N \cdot \mathbb{E}\big[||\mathcal{B}_{\alpha,\mathbf{x}}||\big]$ elements stored among all buckets and the probability of any of those elements being located in a specific bucket is $1/|\mathcal{C}_\mathrm{f}|$. The total running time of Algorithm 4 therefore amounts to

$$T = T_{\text{Bucket}} + T_{\text{Check}} = \tilde{\mathcal{O}}\left(N \cdot \left(T_{\texttt{ValidFilters}} + \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}|\big] \cdot \mathbb{E}\big[|\text{Bucket}_{\mathbf{c},\alpha}|\big]\right)\right),$$

while the expected memory is given by

$$M = \tilde{\mathcal{O}}\left(N\left(1 + \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}|\big]\right)\right) = \tilde{\mathcal{O}}\left(N \cdot \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}|\big]\right),$$

as long as $\mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}|\big] \geq 1$. □

The main differences of all following instantiations of Algorithm 4 lie in the precise choice of $\mathcal{C}_\mathrm{f}$ and the definition of the `ValidFilters` function.

The GJN algorithm We first show that the GJN algorithm already falls into the framework of Algorithm 4 and establish its asymptotic complexity for a later classification of our improvements.

The main idea of the GJN near neighbor algorithm is to exploit the fact that for $\mathbf{x}, \mathbf{y}$ satisfying $|\mathbf{x}| = |\mathbf{y}| = w$ and $|\mathbf{x} + \mathbf{y}| = w$, there exists a $\mathbf{c}$ of weight $w/2$ such that $|\mathbf{x} \wedge \mathbf{c}| = |\mathbf{y} \wedge \mathbf{c}| = w/2$. Moreover, given two vectors $\mathbf{x}, \mathbf{y}$ of weight w, the existence of such a $\mathbf{c}$ implies that $|\mathbf{x} + \mathbf{y}| \leq w$.

In the context of Algorithm 4 the GJN algorithm chooses $\mathcal{C}_\mathrm{f} = \mathcal{S}^n_{w/2}$ and $\alpha = w/2$. For a given $\mathbf{x}$ the valid centers $\mathbf{c}$ are found by simple enumeration of all weight $w/2$ words restricted to the support of $\mathbf{x}$. That is the function `ValidFilters` is defined as

$$\texttt{ValidFilters}(\mathcal{S}^n_{w/2}, \mathbf{x}, w/2) \quad \text{returns} \quad \mathcal{B}_{\mathcal{S}^n_{w/2}, w/2, \mathbf{x}} := \{\mathbf{c} \in \mathcal{S}^n_{w/2} \colon |\mathbf{x} \wedge \mathbf{c}| = w/2\}. \tag{7}$$

Note that this set can be efficiently enumerated in time $|\mathcal{B}_{\mathcal{S}^n_{w/2}, w/2, \mathbf{x}}| = \binom{w}{w/2}$.

Lemma 4.2 (LSF via GJN). *Let $n, w \in \mathbb{N}$, $w < n$. Further, let $\mathcal{C}_\mathrm{f} := \mathcal{S}^n_{w/2}$, $\alpha := w/2$ and `ValidFilters` as defined in (7). Then Algorithm 4 solves the `NNS`(N, n, w) using expected time T and expected memory M, where*

$$T = M = \tilde{\mathcal{O}}\left(N \cdot \binom{w}{w/2}\right).$$

Proof. Note that for any w-close pair $\mathbf{x}, \mathbf{y}$ with $|\mathbf{x}| = |\mathbf{y}| = w$, it holds that $\mathbf{c}^* = \mathbf{x} \wedge \mathbf{y}$ is of weight $|\mathbf{c}^*| = w/2$. Also it implies $|\mathbf{x} \wedge \mathbf{c}^*| = |\mathbf{y} \wedge \mathbf{c}^*| = w/2$. Therefore we have $\mathbf{c}^* \in \mathcal{B}_{\mathcal{S}^n_{w/2}, w/2, \mathbf{x}}$ as well as $\mathbf{c}^* \in \mathcal{B}_{\mathcal{S}^n_{w/2}, w/2, \mathbf{y}}$ implying that any such pair $\mathbf{x}, \mathbf{y}$, is recovered with probability $q = 1$ (compare to Lemma 4.1).

The `ValidFilters` function can be computed in time $T_{\texttt{ValidFilters}} = \tilde{\mathcal{O}}\left(|\mathcal{B}_{\mathcal{S}^n_{w/2}, w/2, \mathbf{x}}|\right) = \tilde{\mathcal{O}}\left(\binom{w}{w/2}\right)$, while the expected bucket size is given (compare to Eq. (6)) as

$$\mathbb{E}\big[|\mathrm{Bucket}_{\mathbf{c},\alpha}|\big] = \frac{N \cdot \mathbb{E}\big[|\mathcal{B}_{\mathcal{S}^n_{w/2}, w/2, \mathbf{x}}|\big]}{|\mathcal{S}^n_{w/2}|} = \frac{N \cdot \binom{w}{w/2}}{\binom{n}{w}}.$$

The expected time complexity therefore becomes (see Lemma 4.1)

$$T = \tilde{\mathcal{O}}\left(N\binom{w}{w/2} \cdot \left(1 + \frac{\binom{w}{w/2}}{\binom{n}{w}}\right)\right) = \tilde{\mathcal{O}}\left(N\binom{w}{w/2}\right),$$

while the expected memory amounts to the same value, since $M = \tilde{\mathcal{O}}\left(N \cdot \mathbb{E}\big[|\mathcal{B}_{\mathcal{S}^n_{w/2}, w/2, \mathbf{x}}|\big]\right) = \tilde{\mathcal{O}}\left(N\binom{w}{w/2}\right)$. □

Improved instantiations of the Framework While the GJN algorithm chooses the set $\mathcal{C}_{\mathrm{f}} = \mathcal{S}^n_{w/2}$ to be all vectors on the $w/2$-sphere, our following algorithms choose $\mathcal{C}_{\mathrm{f}} \subset \mathcal{S}^n_v$ with $v < w/2$.

Notice here that choosing the subset $\mathcal{C}_{\mathrm{f}}$ too small might lead to false negatives, i.e., close pairs that never fall into the same bucket and, hence, remain undetected. On the other hand, to optimize the running time, we aim at choosing $\mathcal{C}_{\mathrm{f}}$ of minimal size while still detecting all pairs. To determine this lower bound on $|\mathcal{C}_{\mathrm{f}}|$, we analyze the number of centers on the v-sphere that can identify a given close pair, which we call D in the following (analogous to Section 1.2). We then show that a $1/D$ fraction of all centers, i.e. $|\mathcal{C}_{\mathrm{f}}| \geq |S^n_v|/D$, is sufficient to identify all pairs.

Aligned with the lattice sieving literature, our analysis uses a geometric interpretation of the algorithm. Let us first recall the definition of a region from Eq. (5)):

$$\mathrm{Region}_{\mathbf{c},v} := \{\mathbf{x} \in \mathbb{F}_2^n \ : \ |\mathbf{x} \wedge \mathbf{c}| = \alpha\}.$$

We can then define a *spherical cap* as the intersection of the sphere with a region[7] and thus obtain the following definition.

Definition 4.1 (Spherical cap). *For $\mathbf{c} \in \mathcal{S}^n_w$, integers $0 \leq \alpha, w \leq n$, a spherical cap is defined by $\mathcal{C}_{\mathbf{c},w,\alpha} := \mathcal{S}^n_w \cap \mathrm{Region}_{\mathbf{c},v} \equiv \{\mathbf{x} \in \mathcal{S}^n_w \ : \ |\mathbf{x} \wedge \mathbf{c}| = \alpha\}$.*

The volume of a cap is defined as the number of elements included in the cap and can be computed as follows.

Theorem 4.1 (Cap volume). *Fix integers $0 \leq \alpha \leq w \leq n$ and fix $\mathbf{c} \in \mathcal{S}^n_v$. Then the volume of $\mathcal{C}_{\mathbf{c},w,\alpha}$ is $\mathscr{C}^n_{v,w,\alpha} := \mathrm{Vol}(\mathcal{C}_{\mathbf{c},w,\alpha}) = \binom{v}{\alpha} \cdot \binom{n-v}{w-\alpha}$.*

Proof. The first binomial in the product defines the number of possible placements of α-many 1's in $\mathbf{x} \in \mathcal{C}_{\mathbf{c},w,\alpha}$ that we should put in the support of $\mathbf{c}$. The second binomial defines the number of possible placements of the remaining $(w - \alpha)$-many 1's of $\mathbf{x}$ in the 0-positions of $\mathbf{c}$. □

Note that the spherical cap $\mathcal{C}_{\mathbf{c},w,\alpha}$ includes all values $\mathbf{x}$ on the w-sphere which are associated with the bucket center $\mathbf{c}$. In turn $\mathcal{C}_{\mathbf{x},v,\alpha} = \mathcal{B}_{\mathcal{S}^n_v,\alpha,\mathbf{x}}$ describes the set of bucket centers $\mathbf{c}$ on the v-sphere to which a fixed element $\mathbf{x}$ is associated. Therefore, the set of bucket centers that is able to identify a fixed pair of distance w, is formed as the intersection of two spherical caps, which we call a *spherical wedge* in the following.

Definition 4.2 (Spherical wedge). *Fix integers $0 \leq \alpha, v \leq n$. For $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^n$ of weight w a (spherical) wedge is defined as*

$$\begin{aligned} \mathcal{W}^n_{\mathbf{x},\mathbf{y},v,\alpha} &:= \mathcal{C}_{\mathbf{x},v,\alpha} \cap \mathcal{C}_{\mathbf{y},v,\alpha} \equiv \mathcal{S}^n_v \cap \mathrm{Region}_{\mathbf{x},\alpha} \cap \mathrm{Region}_{\mathbf{y},\alpha} \\ &\equiv \{\mathbf{c} \in \mathcal{S}^n_v \colon |\mathbf{c} \wedge \mathbf{x}| = |\mathbf{c} \wedge \mathbf{y}| = \alpha\}. \end{aligned}$$

[7] The previously defined regions can also be interpreted as half-spaces common in the lattice sieving literature.

Now the number of centers able to identify a fixed pair $\mathbf{x}, \mathbf{y}$ is the number of elements in $\mathcal{W}^n_{\mathbf{x},\mathbf{y},v,\alpha}$, or alternatively its volume $\mathrm{Vol}(\mathcal{W}^n_{\mathbf{x},\mathbf{y},v,\alpha})$. The following lemma specifies this volume for a fixed pair $\mathbf{x}, \mathbf{y}$ of distance w.

Theorem 4.2 (Wedge volume). *Fix integers $0 \leq \alpha, v \leq n$. For $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^n$ of weight w s.t. $|\mathbf{x} + \mathbf{y}| = w$ it holds that*

$$\mathscr{W}^n_{w,v,\alpha} := \mathrm{Vol}(\mathcal{W}^n_{\mathbf{x},\mathbf{y},v,\alpha}) = \sum_{e=0}^{w/2} \binom{w/2}{e} \binom{w/2}{\alpha - e}^2 \binom{n - 3w/2}{v - 2\alpha + e}$$

Proof. The statement of the theorem follows from counting the possibilities to place the v ones in $\mathbf{c}$ on the positions where either $\mathbf{x}$ or $\mathbf{y}$ have support, none of them have support or both of them have support.

Concretely, denote the number of 1-entries of $\mathbf{c}$ on the positions where $\mathbf{x}$ and $\mathbf{y}$ have support by e. We have $e \in [0, w/2]$, since $|\mathbf{x} \wedge \mathbf{y}| = w/2$. Since $\mathbf{c} \in \mathcal{W}^n_{\mathbf{x},\mathbf{y},v,\alpha}$ implies that $\mathbf{c}$ overlaps with the support of $\mathbf{x}$ (resp. $\mathbf{y}$) in exactly α positions there must be additional $\alpha - e$ ones in $\mathbf{c}$ among the $w/2$ positions where only $\mathbf{x}$ (resp. $\mathbf{y}$) has support. The remaining $v - 2\alpha + e$ ones of $\mathbf{c}$ then have to be placed among the $n - 3w/2$ positions where neither $\mathbf{x}$ nor $\mathbf{y}$ have support. □

The volume of the wedge describes how often a close pair is identified considering all bucket centers $\mathbf{c}$ on $\mathcal{S}_v$. Throughout this quantity is labeled D. The following remark shows how to obtain the previous value of D from Theorem 4.2.

Remark 4.1 (Obtain D via Theorem 4.2). Note that for $\alpha = v$, it follows that the only e for which the term of the sum in Theorem 4.2 is well defined is $e = v$. This in turn gives $\mathrm{Vol}(\mathcal{W}^n_{\mathbf{x},\mathbf{y},v,\alpha}) = \binom{w/2}{v}$ which exactly matches the previously stated value of D in Section 1.2.

Note that asymptotically $\mathscr{W}^n_{w,v,\alpha}$ is equal to the maximal addend of the sum in Theorem 4.2. The following remark shows how to obtain the value of e for which the term in the sum is maximized numerically.

Remark 4.2 (Maximal addend in Theorem 4.2). The value of e for which the addend in the sum of Theorem 4.2 becomes maximized does not seem to have a compact representation. However, it can be computed numerically. In particular, approximating the binomials via $\binom{a}{b} \approx 2^{aH(b/a)}$ and then taking the partial derivative wrt. e, leads to the following cubic

$$e\,(w'/2 + \alpha - e)^2\,(v - 2\alpha + e) = (w'/2 - e)(\alpha - 2)^2(n - 3w'/2 - v + 2\alpha - e). \quad (8)$$

A similar equation appears in the Thesis of Carrier [4, Eq. 8.10]. This cubic has one real and two imaginary roots. The real root gives the maximal addend. To obtain the integer solution the value can be rounded up- or downwards, depending on which one is larger.

In the next lemma, we formalize that choosing the size of $\mathcal{C}_\mathrm{f}$ to be larger than $\mathrm{Vol}(S^n_v)/D = \mathrm{Vol}(S^n_v)/\mathscr{W}^n_{w,v,\alpha}$ indeed guarantees to identify every w-close pair with overwhelming probability.

Lemma 4.3 (Amount of Filters). *Let $n \in N$ be sufficiently large, $w, \alpha, v = \Theta(n)$ be integers. Let $\mathbf{x}, \mathbf{y} \in \mathcal{S}_w^n$ satisfy $|\mathbf{x} \wedge \mathbf{y}| = w$. Further, let $\mathcal{C}_\mathrm{f} \subset \mathcal{S}_v^n$ be a random subset of size $|\mathcal{C}_\mathrm{f}| \geq \mathrm{poly}(n) \cdot \frac{\mathrm{Vol}(\mathcal{S}_v^n)}{\mathrm{Vol}(\mathcal{W}_{\mathbf{x},\mathbf{y},v,\alpha}^n)} = \frac{\mathrm{poly}(n) \cdot \binom{n}{v}}{\mathscr{W}_{w,v,\alpha}^n}$. Then we have*

$$q = \Pr\left[\exists \mathbf{c} \in \mathcal{C}_\mathrm{f} \colon \mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{x}} \cap \mathcal{B}_{\alpha,\mathbf{y}}\right] = \Pr\left[\exists \mathbf{c} \in \mathcal{C}_\mathrm{f} \colon \mathbf{c} \in \mathcal{W}_{\mathbf{x},\mathbf{y},v,\alpha}^n\right] \geq 1 - \mathrm{negl}(n).$$

Proof. We have

$$q = 1 - \left(1 - \frac{\mathscr{W}_{w,v,\alpha}^n}{\mathrm{Vol}(\mathcal{S}_v^n)}\right)^{|\mathcal{C}_\mathrm{f}|} \geq 1 - \left(1 - \frac{\mathscr{W}_{w,v,\alpha}^n}{\mathrm{Vol}(\mathcal{S}_v^n)}\right)^{\frac{\mathrm{poly}(n) \cdot \mathrm{Vol}(\mathcal{S}_v^n)}{\mathscr{W}_{w,v,\alpha}^n}} \geq 1 - \exp(-\,\mathrm{poly}(n))$$

□

4.1 LSF via Coded Hashing

Our first improved version relies on a hash function to select the random subset $\mathcal{C}_\mathrm{f}$. While not leading to the asymptotically fastest variant, it already comes close and has comparably low overhead and therefore might be well suited for practical settings.

Note that in order to select a random subset of filters $\mathcal{C}_\mathrm{f} \subset \mathcal{S}_v^n$ of size $\mathrm{Vol}(\mathcal{S}_v^n)/2^r$ we can define a random hash function $\mathcal{H} \colon \mathcal{S}_v^n \to [2^r]$ and define $\mathcal{C}_\mathrm{f} := \{\mathbf{c} \in \mathcal{S}_v^n \mid \mathcal{H}(\mathbf{c}) = 0\}$. Put differently, we discard all filters $\mathbf{c} \in \mathcal{S}_v^n$ with $\mathcal{H}(\mathbf{c}) \neq 0$. However, without further tweaks, this would still require looping over all possible filters in $\mathcal{S}_v^n$ and evaluating $\mathcal{H}$ in order to decide if the respective filter should be discarded or not. In turn, this would only improve the checking phase, but not the bucketing phase.

To overcome this problem, we design a hash function that, for any given $\mathbf{x} \in L$, identifies more efficiently the valid centers $\mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{x}}$. The hash function is instantiated via a random binary linear code $\mathcal{C}_\mathcal{H}$ of length n and dimension $n - r$. Notice here that for such a code, any filter $\mathbf{c} \in \mathcal{C}_\mathrm{f}$ is contained as a codeword with probability $\Pr[\mathbf{c} \in \mathcal{C}_\mathcal{H}] = \frac{1}{2^r}$. The hash function thus outputs 0 if and only if $\mathbf{c} \in \mathcal{C}_\mathcal{H}$. Therefore, the problem of identifying valid bucket centers reduces to finding codewords of weight v in $\mathcal{C}_\mathcal{H}$.

Let us now take $\alpha = v$, where $\mathcal{C}_\mathrm{f} \subset \mathcal{S}_v^n$. For a given list element $\mathbf{x} \in L$, the support of valid bucket centers $\mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{x}}$ overlaps entirely with the support of $\mathbf{x}$, i.e. $\mathbf{x} \wedge \mathbf{c} = \mathbf{c}$. This implies that for $\mathbf{c} \in \mathcal{B}_{\alpha,\mathbf{x}}$ we have

$$\mathcal{H}(\mathbf{c}) = \mathbf{0} \Leftrightarrow \mathbf{c} \in \mathcal{C}_\mathcal{H} \Leftrightarrow \mathbf{c} \in \sigma_\mathbf{x}(\mathcal{C}_\mathcal{H}),$$

where $\sigma_\mathbf{x}(\mathcal{C}_\mathcal{H})$ denotes a shortened code. This further means we only need to find short codewords in $\sigma_\mathbf{x}(\mathcal{C}_\mathcal{H})$, which is presumably easier. We detail the procedure to identify valid bucket centers for a given list element $\mathbf{x}$ in Algorithm 5.

Lemma 4.4 (`ValidFilters` for Coded Hashing). *Let $\mathcal{C}_\mathcal{H}$ be a $[n, n-r]$ code and $\mathcal{C}_\mathrm{f} = \mathcal{S}_v^n \cap \mathcal{C}_\mathcal{H}$, $v \in N$. Then Algorithm 5 returns the set $\mathcal{B}_{v,\mathbf{x}}$ in time $\binom{w}{v}/\binom{r}{v}$.*

```
Algorithm 5: ValidFilters (coded hashing)
  Input  : $\mathcal{S}^n_v$ and random $[n, n-r]$ code $\mathcal{C}_\mathcal{H}$ describing $\mathcal{C}_\mathrm{f} = S^n_v \cap \mathcal{C}_\mathcal{H}$, list
           element $\mathbf{x} \in \mathcal{S}^n_w$, bucketing parameter $v$
  Output: $\mathcal{B}_{v,\mathbf{x}} := \{\mathbf{c} \in \mathcal{C}_\mathrm{f} \colon |\mathbf{x} \wedge \mathbf{c}| = v\}$
  return $\{\sigma_\mathbf{x}^{-1}(\mathbf{c}) \mid \mathbf{c} \in \sigma_\mathbf{x}(\mathcal{C}_\mathcal{H})$ with $|\mathbf{c}| = v\}$
```

Proof. Note that by the above argumentation the sets

$$\mathcal{B}_{v,\mathbf{x}} := \{\mathbf{c} \in \mathcal{S}^n_v \cap \mathcal{C}_\mathcal{H} \colon |\mathbf{x} \wedge \mathbf{c}| = v\} \quad \text{and} \quad \{\mathbf{c} \in \sigma_\mathbf{x}(\mathcal{C}_\mathcal{H}) \colon |\mathbf{c}| = v\}$$

are identical, implying the correctness of the algorithm. We use Prange's algorithm to find all short codewords in $\sigma_\mathbf{x}(\mathcal{C}_\mathcal{H})$. Note that $\sigma_\mathbf{x}(\mathcal{C}_\mathcal{H})$ has an effective length of $|\mathbf{x}| = w$ and dimension $w - r$. The asymptotic cost of Prange's algorithm to find all weight v codewords in $\sigma_\mathbf{x}(\mathcal{C}_\mathcal{H})$ is, as per Lemma 2.1, $\binom{w}{v}/\binom{r}{v}$. □

The following theorem establishes the running time using our approach of a coded hash function.

Theorem 4.3 (LSF via Coded Hashfunction). *Let $n \in \mathbb{N}$, $w, v = \Theta(n)$. Further let $\alpha := v$, $\mathcal{C}_\mathcal{H}$ be a random binary $[n, n-r]$ code for $r := \log\binom{w/2}{v} - \log n$, $\mathcal{C}_\mathrm{f} = \mathcal{S}^n_v \cap \mathcal{C}_\mathcal{H}$ and* `ValidFilters` *as defined in Algorithm 5. Then Algorithm 4 solves the* `NNS`(N, n, w) *using expected time T and expected memory M, where*

$$T = \tilde{\mathcal{O}}\left(N \cdot \binom{w}{v} \cdot \left(\binom{r}{v}^{-1} + \frac{N\binom{w}{v}}{\binom{n}{v} \cdot 2^r}\right)\right) \quad \textit{and} \quad M = \tilde{\mathcal{O}}\left(N \cdot \binom{w}{v} \Big/ \binom{w/2}{v}\right).$$

Proof. Note that $|\mathcal{C}_\mathrm{f}| = \mathcal{S}^n_v \cap \mathcal{C}_\mathcal{H} = \{\mathbf{c} \in \mathcal{C}_\mathcal{H} \colon |\mathbf{c}| = v\}$. Therefore we have

$$\mathbb{E}\big[|\mathcal{C}_\mathrm{f}|\big] = \frac{\binom{n}{v}}{2^r} = \frac{n \cdot \mathrm{Vol}(\mathcal{S}^n_v)}{\binom{w/2}{v}} = \frac{n \cdot \mathrm{Vol}(\mathcal{S}^n_v)}{\mathscr{W}_{w,v,\alpha}},$$

where the last equality follows from the fact that $\alpha = v$ (compare to Remark 4.1). Assuming that this construction of $\mathcal{C}_\mathrm{f}$ via a random linear code resembles a random subset of $\mathcal{S}^n_v$ of size $\mathbb{E}\big[|\mathcal{C}_\mathrm{f}|\big]$, we can apply Lemma 4.3, which ensures that every close pair is stored in the same bucket at least once with overwhelming probability. The correctness now follows from the correctness of the `ValidFilters` function (see Lemma 4.4) and Algorithm 4 (see Lemma 4.1).

Note that the set of valid filters is of size

$$\mathbb{E}\big[|\mathcal{B}_{v,\mathbf{x}}|\big] = \mathbb{E}\big[|\{\mathbf{c} \in \sigma_\mathbf{x}(\mathcal{C}_\mathcal{H}) \colon |\mathbf{c}| = v\}|\big] = \binom{w}{v}/2^r = \tilde{\Theta}\left(\binom{w}{v}/\binom{w/2}{v}\right).$$

Therefore the condition $\mathbb{E}\big[|\mathcal{B}_{v,\mathbf{x}}|\big] \geq 1$ of Lemma 4.1 is satisfied. Due to Lemma 4.4 the set $\mathcal{B}_{v,\mathbf{x}}$ can be computed in time $T_{\texttt{ValidFilters}} = \binom{w}{v}/\binom{r}{v}$. The expected bucket size is given by

$$\mathbb{E}\big[|\mathrm{Bucket}_{\mathbf{c},\alpha}|\big] = \frac{N \cdot \mathbb{E}\big[|\mathcal{B}_{v,\mathbf{x}}|\big]}{|\mathcal{C}_{\mathrm{f}}|} = \frac{N\binom{w}{v}}{\binom{n}{v}}.$$

Eventually, by plugging in those quantities into the time complexity given by Lemma 4.1 we obtain the claim, namely

$$T = \tilde{\mathcal{O}}\left(N \cdot \left(\binom{w}{v}/\binom{r}{v} + \frac{N\binom{w}{v}^2}{\binom{n}{v}\cdot 2^r}\right)\right) \quad \text{and} \quad M = \tilde{\mathcal{O}}\left(N \cdot \binom{w}{v}/2^r\right). \square$$

We also explored the use of more advanced ISD algorithms for the `ValidFilters` definition from Algorithm 5. However, this resulted only in very small improvements, which is why we stay with the simple Prange formula here.

Saving Memory Through Repetitions In order to ensure a high success probability we only need to classify the input elements according to enough filters $\mathcal{C}_{\mathrm{f}}$ (see Lemma 4.3). Thereby, it is possible to interleave the bucketing and checking phases. We can, for example, first execute the bucketing phase for half of the filters, perform the checking phase, and then repeat the process for the second half of the filters. Note that the size of all buckets is halved (on expectation) in the repeated execution. Hence, as long as the buckets dominate the memory consumption, we obtain a memory improvement with such modification.

More generally, in the following, we execute the algorithm on an initial set of filters $\mathcal{C}'_{\mathrm{f}}$ of size $|\mathcal{C}'_{\mathrm{f}}| = |\mathcal{C}_{\mathrm{f}}|/2^d$. We compensate for the reduced size of the filter set by repeating the algorithm 2^d times. Overall, this improves the memory complexity by a factor of 2^d as formalized in the following corollary.

Corollary 4.1 (LSF via Coded Hashfunction with Repetitions). *Let $n \in \mathbb{N}$, $w, v = \Theta(n)$. Further let, $\alpha := v$, $\mathcal{C}_{\mathcal{H}}$ be a random binary $[n, n-r]$ code for $\log\binom{w}{v} - \log n \geq r \geq \log\binom{w/2}{v} - \log n$, $\mathcal{C}_{\mathrm{f}} = \mathcal{S}^n_v \cap \mathcal{C}_{\mathcal{H}}$ and* `ValidFilters` *as defined in Algorithm 5. Define $d := r - (\log\binom{w/2}{v} - \log n)$. Then 2^d sequential repetitions of Algorithm 4 on fresh randomness solve the* $\mathtt{NNS}(N, n, w)$ *using expected time T and expected memory M, where*

$$T = \tilde{\mathcal{O}}\left(2^d \cdot N \cdot \binom{w}{v} \cdot \left(\binom{r}{v}^{-1} + \frac{N\binom{w}{v}}{\binom{n}{v}\cdot 2^r}\right)\right) \quad \textit{and} \quad M = \tilde{\mathcal{O}}\left(N \cdot \binom{w}{v}/2^r\right).$$

Proof. Over all 2^d iterations, the list elements are still classified with respect to

$$2^d \cdot \mathbb{E}\big[|\mathcal{C}_{\mathrm{f}}|\big] = \frac{2^d\binom{n}{v}}{2^r} = \frac{n \cdot \mathrm{Vol}(\mathcal{S}^n_v)}{\binom{w/2}{v}} = \frac{n \cdot \mathrm{Vol}(\mathcal{S}^n_v)}{\mathscr{W}_{w,v,\alpha}},$$

filters as required by Lemma 4.3.

Note that the time of the algorithm and the memory consumption remain the same as before, now for potentially updated r. Overall, the running time suffers an additional 2^d factor due to the sequential repetitions. □

Interestingly, as we show in Section 5, this repetition approach also leads to an improvement in the time complexity, due to the more optimal choice of r with respect to the decoding routine used within the `ValidFilters` function.

4.2 LSF via Random Product Codes

Our fastest instantiation of Algorithm 4 uses random product codes (RPC) to define the set of centers $\mathcal{C}_\mathrm{f}$. Similarly to the Coded Hashing algorithm, LSF with RPC also comes with a memory-optimal version. Throughout this section, we refer to these algorithms as RPC and RPC-OPT for the usual and memory optimal versions respectively. We now give the description of both algorithms.

Prior work. The techniques and analysis of this subsection turned out to be quite similar to part of the Thesis of Carrier [4, Sec. 8.2]. As it is only available in French, we preferred to keep the details in this document, but the original credit should got to Carrier.

Definition 4.3 (Random product code). *A random product code C is an element drawn uniformly at random from the set*

$$R_{n,v,t} := \{C = C^{(1)} \times \ldots \times C^{(t)} \mid C^{(i)} \subseteq \mathcal{S}^{n/t}_{v/t} \textit{ with } |C^{(i)}| = \sqrt[t]{|C|}\}.$$

In the following we choose $\mathcal{C}_\mathrm{f} = \mathcal{C}_\mathrm{f}^{(1)} \times \mathcal{C}_\mathrm{f}^{(2)} \times \ldots \times \mathcal{C}_\mathrm{f}^{(t)} \in R_{n,v,t}$. Analogous to this product structure, we redefine our regions, or half-spaces, as

$$\mathrm{Region}^{(t)}_{\mathbf{c},\alpha} := \{\mathbf{x} = (\mathbf{x}_1, \ldots \mathbf{x}_t) \in (\mathbb{F}_2^{n/t})^t \mid |\mathbf{x}_i \wedge \mathbf{c}_i| = \alpha/t \ \forall i\}. \tag{9}$$

Similarly, redefine $\mathcal{B}_{\alpha,\mathbf{x}}$ – the set of valid centers for an $\mathbf{x} \in \mathcal{S}^n_w$ as

$$\mathcal{B}^{(t)}_{\alpha,\mathbf{x}} = \{\mathbf{c} \in \mathcal{C}_\mathrm{f} \mid \mathbf{x} \in \mathrm{Region}^{(t)}_{\mathbf{c},\alpha}\}. \tag{10}$$

We first detail how to efficiently find all valid filters $\mathbf{c} \in \mathcal{B}^{(t)}_{\alpha,\mathbf{x}}$ for a given list element. Afterward, we show that the product structure affects the necessary size of $\mathcal{C}_\mathrm{f}$ to guarantee success only by a subexponential factor in comparison to the non-product case.

Identifying valid Bucket Centers We identify valid bucket centers by exploiting the product structure of $\mathcal{C}_\mathrm{f}$. For a given list element $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_t) \in (\mathbb{F}_2^{n/t})^t$ we first find all valid partial centers $\mathbf{c}_i \in \mathcal{C}_\mathrm{f}^{(i)}$ with $|\mathbf{x}_i \wedge \mathbf{c}_i| = \alpha/t$ individually. Then we obtain the full list of valid bucket centers $\mathbf{c} \in \mathcal{C}_\mathrm{f}$ by product-wise combination of the valid partial centers. This procedure is detailed in Algorithm 6. The following lemma shows that the runtime of the algorithm for large enough t is optimal, i.e., it is asymptotically equal to the output size.

Lemma 4.5. *Let $\mathcal{C}_\mathrm{f} \in R_{n,v,t}$ be an RPC. Then Algorithm 6 returns the set $\mathcal{B}^{(t)}_{\alpha,\mathbf{x}}$ in time*

$$T = t \cdot \sqrt[t]{|\mathcal{C}_\mathrm{f}|} + |\mathcal{B}^{(t)}_{\alpha,\mathbf{x}}|. \tag{11}$$

Algorithm 6: `ValidFilters` (RPC)

```
Input  : sets $\mathcal{C}_{\mathrm{f}}^{(i)} \subset \mathcal{S}_{v/t}^{n/t}$ defining an RPC $\mathcal{C}_{\mathrm{f}} = \mathcal{C}_{\mathrm{f}}^{(1)} \times \ldots \times \mathcal{C}_{\mathrm{f}}^{(t)}$, list element
         $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_t) \in \mathcal{S}_w^n$, bucketing parameter $\alpha$
Output: $\mathcal{B}_{\alpha,\mathbf{x}}^{(t)}$
1 $L_i = \emptyset$
2 for $i = 1$ to $t$ do
3     $L_i := \{\mathbf{c}_i \in \mathcal{C}_{\mathrm{f}}^{(i)} : |\mathbf{x}_i \wedge \mathbf{c}_i| = \alpha/t\}$
4 return $L_1 \times L_2 \times \ldots \times L_t$
```

Proof. Note that each list L_i computed in Line 3 of Algorithm 6 contains exactly the elements from $C^{(i)}$ which have the desired overlapping support with $\mathbf{x}_i$. Therefore, elements of the product $\mathbf{c} \in L_1 \times \ldots \times L_t$ are precisely those for which $\mathbf{x} \in \mathrm{Region}_{\mathbf{c},\alpha}$ and vice versa.

The time complexity of Algorithm 6 is composed of the time it takes to construct the L_i's and the time to construct the final product. All these times are linear in the involved list sizes, leading to the lemma's statement. □

Product of random subcodes. Recall that our centers $\mathbf{c} \in \mathcal{C}_{\mathrm{f}}$ are now a concatenation of t vectors of length n/t on the v/t-sphere rather than length-n vectors on the v-sphere. A similar analysis as in the previous section thus requires, instead of considering caps or wedges from $\mathbb{F}_2^n$, to consider the volume of the Cartesian product of t caps or wedges in $\mathbb{F}_2^{n/t}$. The following lemmata show that for $t = o(\frac{n}{\log n})$ the Cartesian product of t caps (resp. wedges) in $\mathbb{F}_2^{n/t}$ approximates a cap (resp. wedge) in $\mathbb{F}_2^n$ up to a subexponential factor in n.

The proofs of both lemmata require the following fact on binomial coefficients.

Fact 1 (Bounds on binomial coefficient [26, Lemma 10.2]) *For $0 \leq b \leq a$, it the following holds*

$$\frac{1}{a+1} 2^{aH(b/a)} \leq \binom{a}{b} \leq 2^{aH(b/a)},$$

where $H := -x\log(x) - (1-x)\log(1-x)$ is the binary entropy function.

Lemma 4.6 (Approximating Caps in $\mathbb{F}_2^n$). *Let $t = o\left(\frac{n}{\log n}\right)$. Then for $\alpha, w, v = \Theta(n)$ the following holds*

$$\left(\mathscr{C}_{v/t,w/t,\alpha/t}^{n/t}\right)^t \frac{1}{\mathrm{poly}(n)} \leq \mathscr{C}_{v,w,\alpha}^n \leq \left(\mathscr{C}_{v/t,w/t,\alpha/t}^{n/t}\right)^t \cdot 2^{o(n)}.$$

Proof. The result follows directly from the application of Fact 1 to Theorem 4.1. As $\mathscr{C}_{v,w,\alpha}^n$ is a product of two binomials, the lower bound on $\mathscr{C}_{v,w,\alpha}^n$ comes applying twice the fact that for any $b \leq a$, $\binom{a}{b} / \binom{a/t}{b/t}^t \geq 1/(a+1)$. The upper bound follows from $\binom{a}{b} / \binom{a/t}{b/t}^t \leq \left(\frac{a}{t}+1\right)^t$ and our choice of t. □

Lemma 4.7 (Approximating Wedges in $\mathbb{F}_2^n$). *Let $t = o\left(\frac{n}{\log n}\right)$. Then for $\alpha, w, v = \Theta(n)$, the following holds*

$$\left(\mathscr{W}^{n/t}_{w/t,v/t,\alpha/t}\right)^t \cdot 2^{-o(n)} \leq \mathscr{W}^n_{w,v,\alpha} \leq \left(\mathscr{W}^{n/t}_{w/t,v/t,\alpha/t}\right)^t \cdot 2^{o(n)}.$$

Proof. From Theorem 4.2 and Remark 4.2 we know that $\mathscr{W}^n_{w,v,\alpha}$ can be approximated up to a polynomial factor by only considering its maximum addend. Precisely this follows from the fact that the volume is the sum of $w/2 = \Theta(n)$ addends, while each of the addends is the product of three binomials, all exponential in n.

Similarly, if we consider the maximal addend in the sum of $\mathscr{W}^{n/t}_{w/t,v/t,\alpha/t}$ we approximate the total sum within a factor of $(w/2)/t = \Theta\left(\frac{n}{t}\right)$. Therefore, via this we obtain an approximation of $\left(\mathscr{W}^{n/t}_{w/t,v/t,\alpha/t}\right)^t$ within a factor of $(n/t)^t = 2^{o(n)}$, since $t = o\left(\frac{n}{\log n}\right)$.

Recall that the maximal addend in $\mathscr{W}^n_{w,v,\alpha}$ is defined by the real solution to Eq. (8). Let us denote it by $e^\star$. Replacing w, α, v in Eq. (8) by respectively $w/t, \alpha/t, v/t$, it follows that $e^\star/t$ is the real solution to this modified qubic equation. Hence, $e^\star/t$ defines the maximal addend in $\mathscr{W}^{n/t}_{w/t,v/t,\alpha/t}$.

Therefore, the statement of the lemma follows if we can show that those two volume approximations still differ at most by a factor of $2^{o(n)}$. To this end, we compare the approximation of $\mathscr{W}^n_{w,v,\alpha}$ which is

$$\binom{w/2}{e^\star}\binom{w/2}{\alpha - e^\star}^2\binom{n - 3w'/2}{v - 2\alpha + e^\star} \tag{12}$$

against the corresponding approximation of $\left(\mathscr{W}^{n/t}_{w/t,v/t,\alpha/t}\right)^t$ given by

$$\binom{w/2t}{e^\star/t}^t\binom{w/2t}{\alpha/t - e^\star/t}^{2t}\binom{n/t - 3w'/2t}{v/t - 2\alpha/t + e^\star/t}^t. \tag{13}$$

From Fact 1 it holds that

$$\left(\frac{1}{a/t+1}\right)^t 2^{aH(b/a)} \leq \binom{a/t}{b/t}^t \leq 2^{aH(b/a)}.$$

Now to show the lower bound on $\mathscr{W}^n_{w,v,\alpha}$ as in the lemma's statement, consider $\binom{a}{b}/\binom{a/t}{b/t}^t \geq \frac{1}{a+1} = \Omega(1/n)$, where the inequality follows from taking the lower bound on $\binom{a}{b}$ and the upper bound on $\binom{a/t}{b/t}^t$. The statement follows from noticing that $a = \Theta(n)$ in the three binomial coefficients from Equation (12).

Similarly, the upper bound on $\mathscr{W}^n_{w,v,\alpha}$ follows again from the inequalities on the binomial coefficients. It holds that $\binom{a}{b}/\binom{a/t}{b/t}^t \leq \left(\frac{1}{a/t+1}\right)^{-t} \leq \left(\frac{a}{t}\right)^t$. Since $a = \Theta(n)$, for any $t = o\left(\frac{n}{\log n}\right)$, it holds that $\left(\frac{a}{t}\right)^t < 2^{o(n)}$, from which the statement follows. □

The main application of random product codes in sieving is, for a given $\mathbf{x} \in L$, finding all relevant filters efficiently. This implies that we require two properties to be satisfied by a random product code: it should be efficiently decodable and it should behave like a random code in the sense that the success probability for $C \leftarrow R_{n,v,t}$ to 'capture' a pair $\mathbf{x}, \mathbf{y} \in L$ should be (up to subexponential in n factors) the same as for a random code $C \subset \mathcal{S}_v^n$. The first property – the decodability – is elaborated on in Lemma 4.5, while the next theorem shows the 'randomness' property.

Lemma 4.8 (Amount of Filters for RPCs). *Let n be sufficiently large, $w, \alpha, v = \Theta(n)$, $t = o\left(\frac{n}{\log n}\right)$ be integers. Let $\mathbf{x}, \mathbf{y} \in \mathcal{S}_w^n$ satisfy $|\mathbf{x} \wedge \mathbf{y}| = w$. Further, let $C \in R_{n,v,t}$ be an RPC and $P = \{\pi_i\}_{i \in [n(n/t+1)^{3t}]}$ be a selection of independent random permutations on n elements. Denote by q the probability that there exists a $\pi \in P$ and a $\mathbf{c} \in C$ such that $\mathbf{c} \in \mathcal{W}^n_{\pi(\mathbf{x}),\pi(\mathbf{y}),v,\alpha}$. Then*

$$q \geq \min\left\{|C| \cdot \frac{\mathscr{W}^n_{w,v,\alpha}}{\mathrm{Vol}(\mathcal{S}_v^n) \cdot 2^{o(n)}},\ 1 - \mathrm{negl}(n)\right\}.$$

Proof. Denote by $\bar{\mathbf{x}} = (\bar{\mathbf{x}}_1, \ldots, \bar{\mathbf{x}}_t) = \pi(\mathbf{x})$, $\bar{\mathbf{y}} = (\bar{\mathbf{y}}_1, \ldots, \bar{\mathbf{y}}_t) = \pi(\mathbf{y})$, for $\pi \in P$. We first show that with overwhelming probability over the choice of P there is at least one $\pi \in P$ for which

$$|\bar{\mathbf{x}}_i| = |\bar{\mathbf{y}}_i| = |\bar{\mathbf{x}}_i + \bar{\mathbf{y}}_i| = \frac{w}{t} \quad \text{for all } i = 1, \ldots, t. \tag{14}$$

Note that this requires that exactly $w/(2t)$ out of the $w/2$ coordinates where only $\mathbf{x}$, exactly $w/(2t)$ out of the $w/2$ coordinates where only $\mathbf{y}$, and exactly $w/(2t)$ out of the $w/2$ coordinates where $\mathbf{x}$ and $\mathbf{y}$ are non-zero, must be present in each of the $\bar{\mathbf{x}}_i$ (resp. $\bar{\mathbf{y}}_i$). For any permutation π this happens with probability

$$\begin{aligned} q_\pi &= \Pr\left[|\bar{\mathbf{x}}_i| = |\bar{\mathbf{y}}_i| = |\bar{\mathbf{x}}_i + \bar{\mathbf{y}}_i| = \frac{w}{t} \colon |\mathbf{x}| = |\mathbf{y}| = |\mathbf{x} + \mathbf{y}| = w\right] \\ &= \frac{\left(\binom{n/t}{w/(2t)}\binom{n-w/(2t)}{w/(2t)}\binom{n-w/t}{w/(2t)}\right)^t}{\binom{n}{w/2}\binom{n-w/2}{w/2}\binom{n-w}{w/2}} \geq \left(\frac{1}{n/t+1}\right)^{3t}. \end{aligned}$$

The probability that there exists at least one π among the $n(n/t+1)^{3t}$ permutations in P that is suitable is then given by

$$1 - (1 - q_\pi)^n (n/t+1)^{3t} > 1 - \exp(-n) = 1 - \mathrm{negl}(n).$$

In the following, we restrict our analysis to a suitable permutation π satisfying Eq. (14).

We denote the product of subwedges defined by the $\bar{\mathbf{x}}_i, \bar{\mathbf{y}}_i$ as

$$\mathit{\Pi} := \prod_{i=1}^{t} \mathcal{W}^{n/t}_{\bar{\mathbf{x}}_i, \bar{\mathbf{y}}_i, v/t, \alpha/t}.$$

Note that it holds

$$\mathit{\Pi} \subset \mathcal{W}^n_{\bar{\mathbf{x}},\bar{\mathbf{y}},v,\alpha}. \tag{15}$$

Indeed, any $\mathbf{z} = (\mathbf{z}_1, \ldots, \mathbf{z}_t) \in \mathit{\Pi}$ satisfies $|\mathbf{z}_i \wedge \bar{\mathbf{x}}_i| = |\mathbf{z}_i \wedge \bar{\mathbf{y}}_i| = \alpha/t$ and $|\mathbf{z}_i| = v/t$. It implies that $|\mathbf{z} \wedge \bar{\mathbf{x}}| = |\mathbf{z} \wedge \bar{\mathbf{y}}| = \alpha$ and $|\mathbf{z}| = v$. Therefore, $\mathbf{z} \in \mathcal{W}^n_{\bar{\mathbf{x}},\bar{\mathbf{y}},v,\alpha}$. Denote by q_i the probability that $\mathbf{c}_i \in C^{(i)}$ is in $\mathcal{W}^{n/t}_{\bar{\mathbf{x}}_i,\bar{\mathbf{y}}_i,v/t,\alpha/t}$, where $C = C^{(1)} \times \ldots \times C^{(t)}$. From the Inclusion (15), it follows that $q > q_1 \cdot q_2 \cdot \ldots \cdot q_t$. Moreover,

$$q_i = 1 - \left(1 - \frac{\mathscr{W}^{n/t}_{w/t,v/t,\alpha/t}}{\mathrm{Vol}(\mathcal{S}^{n/t}_{v/t})}\right)^{|C^{(i)}|}.$$

Now, since we restrict our analysis to the permutation π satisfying Eq. (14) we can apply Lemma 4.7 to approximate the wedge volume as

$$\mathscr{W}^{n/t}_{w/t,v/t,\alpha/t} \geq \frac{\left(\mathscr{W}^n_{w,v,\alpha}\right)^{1/t}}{2^{o(n)/t}},$$

from which it follows that

$$q_i \geq 1 - (1 - \bar{q}_i)^{|C^{(i)}|} \quad \text{with} \quad \bar{q}_i = \frac{(\mathscr{W}^n_{w,v,\alpha})^{1/t}}{\mathrm{Vol}(\mathcal{S}^{n/t}_{v/t})2^{o(n)/t}}.$$

Let $|C^{(i)}| = X/\bar{q}_i$. Recall that for all $i = 1, \ldots, t$ we have $|C^{(i)}| = \sqrt[t]{|C|}$ or, equivalently,

$$|C| = |C^{(i)}|^t = \frac{\mathscr{W}_{w,v,\alpha} X^t}{\mathrm{Vol}(\mathcal{S}^n_v) \cdot 2^{o(n)}}, \tag{16}$$

where we use the fact that $\mathrm{Vol}(\mathcal{S}^{n/t}_{v/t}) = 2^{o(n)}\mathrm{Vol}(\mathcal{S}^n_v)$.

We now make three case distinctions based on the size of X.

Case $X > 2^{o(n)/t}$. We directly obtain

$$q_i \geq 1 - (1 - \bar{q}_i)^{X/\bar{q}_i} > 1 - \exp(-X) = 1 - \mathrm{negl}(n),$$

and ultimately $q > \prod_{i=1}^t q_i = 1 - \mathrm{negl}(n)$.

Case $1 \leq X < 2^{o(n)/t}$. Here we can bound q_i as in the previous case, namely

$$q_i \geq 1 - \exp(-X) \geq 1 - \exp(-1),$$

giving $q > \prod_{i=1}^t q_i = 2^{-o(n)}$. Note that for this choice of X we also have (compare to Eq. (16))

$$|C| = \frac{\mathscr{W}_{w,v,\alpha}}{\mathrm{Vol}(\mathcal{S}^n_v) \cdot 2^{o(n)}},$$

and hence $q \geq 2^{-o(n)} = |C| \cdot \frac{\mathscr{W}^n_{w,v,\alpha}}{\mathrm{Vol}(\mathcal{S}^n_v) \cdot 2^{o(n)}}$ as claimed.

Case $X < 1$. In that case, we obtain

$$q_i \geq 1 - (1 - \bar{q}_i)^{|C^{(i)}|} = |C^{(i)}| \cdot \bar{q}_i - \Theta\big((|C^{(i)}| \cdot \bar{q}_i)^2\big) = \Theta(|C^{(i)}| \cdot \bar{q}_i),$$

where the last equality follows from the fact that $|C^{(i)}| \cdot \bar{q}_i = X < 1$ This immediately gives $q > \prod_{i=1}^{t} q_i = \Theta(|C^{(i)}| \cdot \bar{q}_i)^t = |C| \cdot \frac{\mathscr{W}^n_{w,v,\alpha}}{\mathrm{Vol}(\mathcal{S}^n_v) \cdot 2^{o(n)}}$, concluding the proof. □

Now we are ready to give the complexity of Algorithm 4 when $\mathcal{C}_\mathrm{f}$ is instantiated with a random product code. Before giving the statement, we remind the reader that $\mathscr{C}^n_{v,w,\alpha} = \mathrm{Vol}(\mathcal{C}_{\mathbf{c},w,\alpha})$ represents the volume of a cap centered at $\mathbf{c}, |\mathbf{c}| = v$, on the sphere $\mathcal{S}^n_w$, while $\mathscr{C}^n_{w,v,\alpha} = \mathrm{Vol}(\mathcal{C}_{\mathbf{x},v,\alpha})$ represents the volume of a cap centered at $\mathbf{x}, |\mathbf{x}| = w$, on the sphere $\mathcal{S}^n_v$. The former gives rise to the expected size of a bucket, while the latter describes all valid centers for $\mathbf{x}$.

Theorem 4.4 (LSF via RPC). *Let $n \in \mathbb{N}$, $w, v, \alpha = \Theta(n) \in \mathbb{N}$ and $t = \Theta(\sqrt{n}) \in \mathbb{N}$. Further, let $\mathcal{C}_\mathrm{f} \in R_{n,v,t}$ be an RPC such that $|\mathcal{C}_\mathrm{f}| = \frac{2^{o(n)} \cdot \mathrm{Vol}(\mathcal{S}^n_v)}{\mathscr{W}^n_{w,v,\alpha}}$, with $|\mathcal{C}_\mathrm{f}| \cdot q = 1 - \mathrm{negl}(n)$ for q as defined in Lemma 4.8.*
Denote by $P = \{\pi_i\}_{i \in [n(n/t+1)^{3t}]}$ a selection of independent random permutations on n elements. Instantiate the function `ValidFilters` *using Algorithm 6. Let (L, n, w) be a $\mathtt{NNS}(N, n, w)$ instance. Then the iterative execution of Algorithm 4 on input $(\pi(L), n, w), \mathcal{C}_\mathrm{f}, \alpha$, for all $\pi \in P$ solves the $\mathtt{NNS}(N, n, w)$ with overwhelming probability within expected time T and expected memory M, where*

$$T = 2^{o(n)} \cdot N \cdot \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\mathscr{W}^n_{w,v,\alpha}} \left(1 + \frac{N\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\binom{n}{v}}\right) \quad \text{and} \quad M = 2^{o(n)} \cdot N \cdot \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\mathscr{W}^n_{w,v,\alpha}}.$$

Proof. Fix any pair $\mathbf{x}, \mathbf{y} \in L$ of distance w. We consider in the following an execution of Algorithm 4 on the list $\pi(L)$ for a $\pi \in P$ that satisfies Eq. (14), i.e., it gives the desired weight distribution on $\mathbf{x}, \mathbf{y}$. We have already shown in the proof of Lemma 4.8 that for a collection of size, $|P| = n(n/t+1)^{3t}$ such a π exists with overwhelming probability over the choice of P. Now Lemma 4.8 ensures that for our choice of $|\mathcal{C}_\mathrm{f}|$, there is at least one filter $\mathbf{c} \in \mathcal{C}_\mathrm{f}$ that leads to the recovery of the pair $\mathbf{x}, \mathbf{y}$ with probability $1 - \mathrm{negl}(n)$.

Let us now analyze the complexity using Lemma 4.1. Thanks to Lemma 4.5, we have $T_{\mathtt{ValidFilters}} = t\sqrt[t]{|\mathcal{C}_\mathrm{f}|} + |\mathcal{B}^{(t)}_{\alpha,\mathbf{x}}|$. Note that for our choice of $\mathcal{C}_\mathrm{f}$ we have

$$\mathbb{E}\big[|\mathcal{B}^{(t)}_{\alpha,\mathbf{x}}|\big] = \sum_{\mathbf{c} \in \mathcal{C}_\mathrm{f}} \Pr\Big[|\mathbf{c}_i \wedge \mathbf{x}_i| = \frac{\alpha}{t}\ \forall i\Big] = |\mathcal{C}_\mathrm{f}| \left(\frac{\mathscr{C}^n_{w/t,v/t,\alpha/t}}{\mathrm{Vol}(\mathcal{S}^{n/t}_{v/t})}\right)^t = 2^{o(n)} \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\mathscr{W}^n_{w,v,\alpha}},$$

where the last equality follows from our choice of $|\mathcal{C}_\mathrm{f}|$ and the approximation $\left(\frac{\mathscr{C}^n_{w/t,v/t,\alpha/t}}{\mathrm{Vol}(\mathcal{S}^{n/t}_{v/t})}\right)^t \geq \frac{\mathscr{C}^n_{w,v,\alpha}}{2^{o(n)}\mathrm{Vol}(\mathcal{S}^n_v)}$ given by Lemma 4.6. Note that $\mathbb{E}\big[|\mathcal{B}^{(t)}_{\alpha,\mathbf{x}}|\big] \geq 1$ as for any $\mathbf{x} \in \mathcal{S}_w^n$, a cup defined by $\mathbf{x}$ contains a wedge defined by this $\mathbf{x}$, therefore the ratio between their volumes is greater than 1 and hence, the condition of Lemma 4.1 is satisfied. Therefore, on expectation

$$T_{\texttt{ValidFilters}} = 2^{o(n)} \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\mathscr{W}_{w,v,\alpha}^n},$$

since $\mathcal{B}_{\alpha,\mathbf{x}}$ as well as $\mathcal{C}_\mathrm{f}$ are of size exponential in n and $t = \Theta(\sqrt{n})$.

What remains is to argue on $\mathbb{E}\big[|\mathrm{Bucket}_{\mathbf{c},\alpha}|\big]$. From the proof of Lemma 4.1, it follows that

$$\mathbb{E}\big[|\mathrm{Bucket}_{\mathbf{c},\alpha}|\big] = \frac{N \cdot \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}^{(t)}|\big]}{|\mathcal{C}_\mathrm{f}|} = 2^{o(n)} \cdot N \cdot \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\mathrm{Vol}(\mathcal{S}_v^n)}$$

Collecting all the expectations and applying Lemma 4.1, obtain

$$T = N \cdot \left(2^{o(n)} \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\mathscr{W}_{w,v,\alpha}^n} + 2^{o(n)} \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\mathscr{W}_{w,v,\alpha}^n} \cdot N \cdot \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\mathrm{Vol}(\mathcal{S}_v^n)} \right),$$

which is equivalent to the theorem's statement.

Due to Lemma 4.1, the memory complexity is given by $M = N \cdot \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}^{(t)}|\big]$ as it is stated in the theorem. □

Saving Memory Through Repetitions in case of RPC Analogously to Corollary 4.1, we exploit the idea of choosing a smaller set of filters and repeating Algorithm 6 for many of these smaller sets.

Concretely, we independently choose (omitting subexponential factors for brevity) $d := \frac{\mathscr{C}_{w,v,\alpha}^n}{\mathscr{W}_{w,v,\alpha}^n}$-many RPCs $\mathcal{C}_\mathrm{f} \in R_{n,v,t}$ each of size $|\mathcal{C}_\mathrm{f}| = \frac{\mathrm{Vol}(\mathcal{S}_v^n)}{\mathscr{C}_{w,v,\alpha}^n}$. First, this choice of $|\mathcal{C}_\mathrm{f}|$ yields the expected number of buckets per $\mathbf{x} \in L$ to be 1, i.e., $\mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}^{(t)}|\big] = 1$, which follows from the proof of Theorem 4.4. Second, thanks to Lemma 4.8, the success probability of each run with a smaller RPC is still at least $|\mathcal{C}_\mathrm{f}| \cdot \frac{\mathscr{W}_{w,v,\alpha}^n}{\mathrm{Vol}(\mathcal{S}_v^n)}$, thus repeating the algorithm d times bounds from below the success probability q of finding a good pair $\mathbf{x}, \mathbf{y}$ as

$$q \geq 1 - \left(1 - |\mathcal{C}_\mathrm{f}| \cdot \frac{\mathscr{W}_{w,v,\alpha}^n}{\mathrm{Vol}(\mathcal{S}_v^n)}\right)^d = \left(1 - \frac{\mathscr{W}_{w,v,\alpha}^n}{\mathscr{C}_{w,v,\alpha}^n}\right)^d \geq 1 - \frac{1}{e}.$$

This success probability can be made overwhelming by choosing a slightly larger d, e.g., $d = 2^{o(n)} \frac{\mathscr{C}_{w,v,\alpha}^n}{\mathscr{W}_{w,v,\alpha}^n}$.

This leads to the following corollary. Compared to the result from Theorem 4.4, it has asymptotically the same running time but achieves optimal memory. A similar statement is given in the Thesis of Carrier [4, Cor. 8.2.6].

Corollary 4.2 (Memory optimal LSF via PRC). *Let $n \in \mathbb{N}$, $w, v, \alpha = \Theta(n) \in \mathbb{N}$ and $t = \Theta(\sqrt{n}) \in \mathbb{N}$. Further, let $R = \{\mathcal{C}_\mathrm{f} \in R_{n,v,t}\}_{i \in [d]}$ be a set of*

independently chosen RPCs with $|\mathcal{C}_\mathrm{f}| = 2^{o(n)} \cdot \frac{\mathrm{Vol}(\mathcal{S}_v^n)}{\mathscr{C}_{w,v,\alpha}^n}$*, and* $d = 2^{o(n)} \frac{\mathscr{C}_{w,v,\alpha}^n}{\mathscr{W}_{w,v,\alpha}^n}$*. Denote by* $P = \{\pi_i\}_{i \in [n(n/t+1)^{3t}]}$ *a selection of independent random permutations on* n *elements. Instantiate the function* `ValidFilters` *using Algorithm 6.*
Let (L, n, w) *be a* $\mathtt{NNS}(N, n, w)$ *instance. Then the iterative execution of Algorithm 4 on input* $(\pi(L), n, w), \mathcal{C}_\mathrm{f}, \alpha$*, for all* $\pi \in P, \mathcal{C}_\mathrm{f} \in R$*, solves the* $\mathtt{NNS}(N, n, w)$ *with overwhelming probability within expected time* T *and expected memory* M*, where*

$$T = 2^{o(n)} \cdot N \cdot \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\mathscr{W}_{w,v,\alpha}^n} \left(1 + \frac{N\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\binom{n}{v}}\right) \quad \text{and} \quad M = 2^{o(n)} \cdot N.$$

Proof. As argued above, for our choice of $|\mathcal{C}_\mathrm{f}|$, per each execution of Algorithm 4, we have $\mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}^{(t)}|\big] = 2^{o(n)}$. From Lemma 4.1 this already gives the statement for the memory M.

Smaller choice of $|\mathcal{C}_\mathrm{f}|$ also leads to $T_{\mathtt{ValidFilters}} = 2^{o(n)}$, and

$$\mathbb{E}\big[|\mathrm{Bucket}_{\mathbf{c},\alpha}|\big] = \frac{N \cdot \mathbb{E}\big[|\mathcal{B}_{\alpha,\mathbf{x}}^{(t)}|\big]}{|\mathcal{C}_\mathrm{f}|} = 2^{o(n)} N \cdot \frac{\mathscr{C}_{w,v,\alpha}^n}{\mathrm{Vol}(\mathcal{S}_v^n)} = 2^{o(n)} N \cdot \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\binom{n}{v}}.$$

Therefore, one iteration of Algorithm 4 has complexity

$$2^{o(n)} N \cdot \left(1 + N \cdot \frac{\binom{w}{\alpha}\binom{n-w}{v-\alpha}}{\binom{n}{v}}\right).$$

Repeating Algorithm 4 $\left(2^{o(n)} \frac{\mathscr{C}_{w,v,\alpha}^n}{\mathscr{W}_{w,v,\alpha}^n}\right)$-many times gives the statement. □

5 Results and Performance Comparisons

Each of the presented algorithms to solve the w-near neighbor search from Section 4 leads to an instantiation of the SievingISD algorithm (Algorithm 2) via the machinery presented in Section 3. The SievingISD framework dictates specific parameters for the w-near neighbor search problem $\mathtt{NNS}(N, n', w')$ solved within the ISD routine. While n' and w' are optimization parameters chosen to minimize the running time, N is chosen equal to the lower bound given in Constraint (C 2), to ensure that there are again N close vectors.

Note that algorithms solving the w-near neighbor search might be of independent interest for a broader range of parameters. Therefore, in order to allow for a more general categorization, we compare the performance of the near neighbor search algorithms for a wider range of parameters first, independent of the choices in SievingISD. However, this comparison already allows us to draw conclusions on possible speedups obtained via those algorithms in the context of the SievingISD framework. Subsequently, we study the resulting SievingISD instantiations in more detail.

5.1 Performance of Near Neighbor Algorithms

In the comparison of algorithms to solve the w-near neighbor search we refer to the algorithms as GJN ([16], Lemma 4.2), Hash (Theorem 4.3), Hash-opt (Corollary 4.1), RPC (Theorem 4.4) and RPC-opt (Corollary 4.2). Additionally, we compare those algorithms against a quadratic search baseline that naively computes all list pairs to find those which are close, and against an algorithm recently proposed by Esser [9].

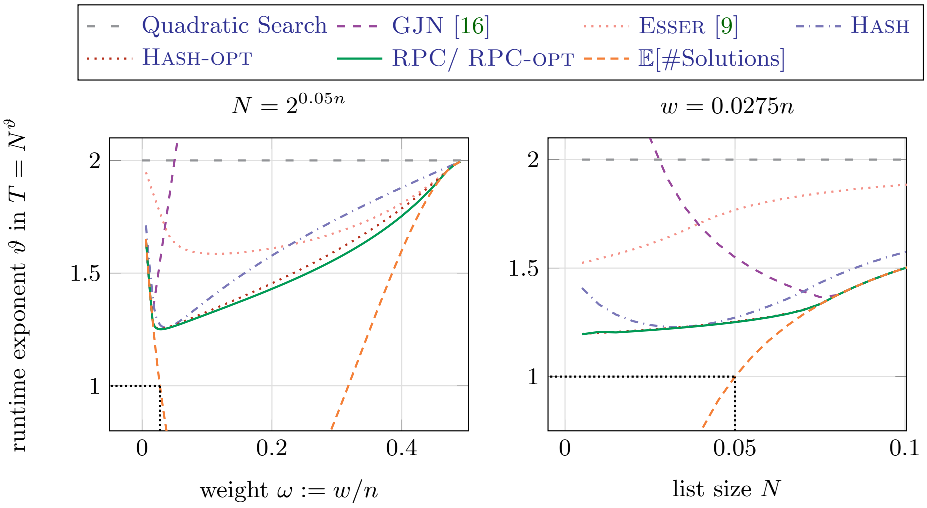


Fig. 4: Comparison of the running time of different algorithms solving the w-near neighbor search for fixed list size (left) and fixed weight (right).

On the left in Fig. 4 we compare the running time of the different algorithms for different relative weights $\omega := w/n$ and fixed list size $N = 2^{0.05n}$. A choice that roughly corresponds to the list sizes encountered in the later SievingISD application. All algorithms, with the exception of GJN, outperform the quadratic-search baseline for all weights $\omega < 0.5$. Furthermore, the fastest algorithms presented in this work outperform the previous approaches, GJN and Esser, for all weights. Note that RPC and RPC-opt obtain the same running time since RPC-opt corresponds to a pure memory improvement. Interestingly, the same repetition approach leads to a significant time improvement in the context of Hash-opt over Hash. This is because the extra degree of freedom allows to optimization of the code parameters to reduce the overhead for finding valid bucket centers in Hash-opt via Lemma 4.4.

Additionally, the graph depicts the expected amount of solutions, calculated as $\mathbb{E}[\#\text{Solutions}] = \binom{n}{w}/2^{n-k}$, as an orange dashed line. It can be observed that all algorithms, except Esser, obtain a running time that is (roughly) linear in the number of existing solutions for very small weights. For larger weights the complexities diverge, while all algorithms (except GJN) converge to the quadratic search baseline for weight $\omega = 0.5$.

In the SievingISD application, we are interested in the performance of the algorithms when the amount of solutions is equal to the list size, i.e., the point on the dashed orange line at $\vartheta = 1$. This is the case for $\omega \approx 0.02748$, which is highlighted by a black dotted line in the plot. We find that the new algorithm from Section 4.1 significantly improves on GJN in that regime, indicating an improved SievingISD algorithm. On the other hand, all new algorithms obtain similar complexities in that regime, implying that they show similar performance within the SievingISD framework. The algorithm by Esser performs worse than GJN in that regime indicating no improvement in the SievingISD context.

On the right in Figure 4 we consider for completeness the running time of the algorithms for fixed weight and variable list size. Again, the SievingISD relevant instantiation, where the amount of solutions is equal to the list size, is highlighted via a black dotted line.

5.2 Performance of SievingISD Instantiations

In this section, we detail the performance of the SievingISD instantiations obtained via the w-near neighbor search algorithms from Section 4. We then compare the obtained complexities against the state of the art of ISD algorithms.

Obtaining Different SievingISD Instantiations Recall, that Theorem 3.1 states the running time of any SievingISD algorithm depending on the time complexity of an oracle to find short codewords of weight w' in a given code. We then instantiate this oracle via a sieving routine (see Algorithm 3). Under the Binary-Sieve Heuristic (Heuristic 1) the complexity of this sieving algorithm is equal to the complexity of solving the w'-near neighbor search.

Different SievingISD algorithms are obtained by instantiating the near neighbor search routine, used within the sieving routine, with the different algorithms from Section 4. We refer to the obtained instantiations as: Sisd-GJN (Lemma 4.2), Sisd-Hash (Theorem 4.3), Sisd-Hash-opt (Corollary 4.1), Sisd-RPC (Theorem 4.4) and Sisd-RPC-opt (Corollary 4.2).

Notice here that the near neighbor search instance solved within the sieving routine corresponds to the $\texttt{NNS}(N, n', w')$ problem, for N matching Eq. (C 2), and n', w' as defined in Theorem 3.1. We then obtain the running time of different instantiations by replacing T_{NNS} from Theorem 3.2 with appropriate statements.

In order to compare the complexities of different instantiations, we follow the common practice of modeling the running time and memory as $2^{c(k,w)n}$, where c is a constant that depends on k and w. Therefore, we approximate all binomial coefficients via the upper bound given in Fact 1. Note that this leads to at most a polynomial divergence, asymptotically subsumed by the fact that we always round the constant $c(k, w)$ upwards. We then consider $k = \kappa n$, $w = \omega n$, for constants κ, ω, and model any additional optimization parameter o_i, such as n'

Type	Algorithm	κ	$c_T(\kappa, \omega)$	$c_M(\kappa, \omega)$
SievingISD	Sisd-GJN [16]	0.44	0.1169	0.0279
	Sisd-Hash	0.44	0.1007	0.0849
	Sisd-Hash-opt	0.44	0.1007	0.0830
	Sisd-RPC	0.44	0.1001	0.0852
	Sisd-RPC-opt	0.44	0.1001	0.0636
Conventional ISD	Prange [29]	0.45	0.1207	0.0000
	MMT [23]	0.45	0.1116	0.0541
	BJMM [2]	0.43	0.1020	0.0728
	Both-May [3]	0.42	0.0951	0.0754

Table 1: Worst case running time $2^{c_T(\kappa,\omega)n}$ and corresponding memory usage $2^{c_M(\kappa,\omega)n}$ for different ISD algorithms. Running time is maximized for given κ using $\omega = H^{-1}(1-\kappa)$ equal to the Gilbert-Varshamov bound.

and w', as $o_i = \hat{o}_i n$. For given κ, ω, we then perform a numerical minimization of the running time over the choice of the $\hat{o}_i$, resulting in the complexity exponent $c(k, w)$, or $c(\kappa, \omega)$, as the constant only depends on κ and ω.

Worst case Complexities A common measure to compare the performance of algorithms to solve the syndrome decoding problem is their worst case complexity. Therefore one considers $w = \omega n$ matching the Gilbert-Varshamov bound, i.e., $\omega = H^{-1}(1-\kappa)$, with H^{-1} being the inverse of the binary entropy function in the interval $[0, 0.5]$. The worst case running time is then obtained by maximizing the constant $c(\kappa, \omega)$ over all possible choices of the rate κ. The following table states the worst case running times for the different SievingISD instantiations in comparison to the best known ISD algorithms.[8]

We observe that the new SievingISD instantiations obtain a significant improvement over the running time of the original Sisd-GJN proposal from [16]. Still, they do not yet reach the best time complexity exponent for conventional ISD algorithms, given by the Both-May algorithm [3][9]. However, the new algorithms yield the first improvement over the running time of the BJMM algorithm, which does not follow the conventional ISD paradigm. Furthermore, our more practical instantiations Sisd-Hash and Sisd-Hash-opt, still slightly outperform the BJMM algorithm, while significantly improving on the MMT algorithm, which is usually the preferred choice in practice [12,13].

In Fig. 5 we compare the running time exponent of the different SievingISD instantiations, Sisd-GJN, Sisd-Hash-opt and Sisd-RPC-opt for all rates κ against conventional ISD procedures. We find that the Sisd-GJN instantiation falls in between the running times of Prange and of MMT. The improved

[8] For obtaining the numerical exponents of conventional ISD procedures we use the code available at `https://github.com/Memphisd/Revisiting-NN-ISD`.

[9] See [5,9] for a correction of the initial result.

SievingISD instantiations offer BJMM comparable running times. We observe that for rates $\kappa \leq 0.6$ our best SievingISD instantiations even outperform the BJMM algorithm. It can also be observed that our more practical SISD-HASH-OPT instantiation generally suffers only a slight overhead in terms of time complexity compared to our best SISD-RPC-OPT variant, as it was also suggested by the comparison in Section 5.1.

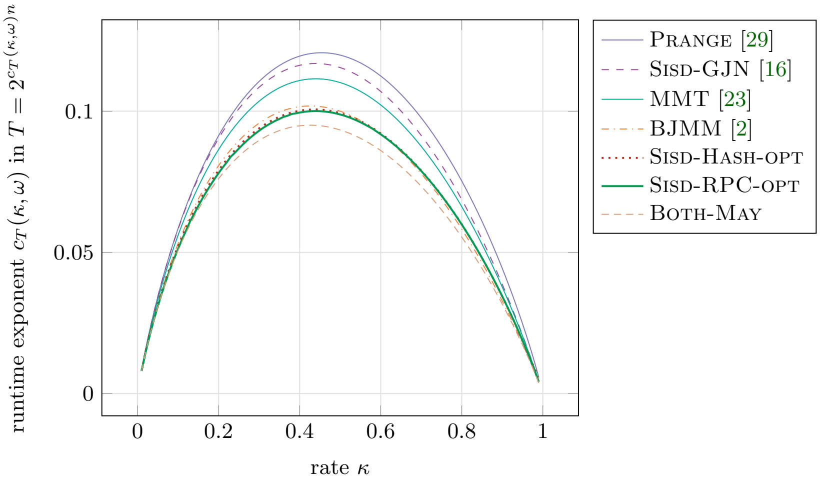


Fig. 5: Runtime exponent for different ISD and SievingISD variants as a function of the rate κ using $\omega := H^{-1}(1-\kappa)$.

Time-Memory Trade-Offs Table 1 indicates that more advanced algorithms obtain time improvement by spending higher amounts of memory. In fact, all modern ISD algorithms, conventional as well as those following the SievingISD framework, use an exponential amount of memory in order to obtain their best runtime exponents. In the following, we show that our SievingISD instantiations also significantly improve the time-memory trade-off potential of the early algorithm by Guo-Johansson-Nguyen and also outperform some of the conventional ISD trade-offs and recently proposed improvements.

Note that all ISD algorithms can be interpolated to the memoryless algorithm by Prange. This memoryless endpoint for the SievingISD algorithms corresponds to the choice of $n' = k$, $w' = 0$ in Theorem 3.1. Then, by gradually increasing the sieving effort through larger choices of n', w' one obtains a continuous time-memory trade-off, i.e., an instantiation of the algorithm for any fixed amount of memory. A similar interpolation is possible for conventional ISD algorithms. In Fig. 6 we compare the time-memory trade-off curves resulting from our SievingISD instantiations for rate $\kappa = 0.5$ and $\omega = H^{-1}(0.5) \approx 0.11$ against the initial approach by Guo, Johansson and Nguyen. We observe that the new algorithms improve the time complexity for any fixed amount of memory. Interestingly, our practical SISD-HASH-OPT instantiation yields a better time-memory trade-off curve than SISD-RPC, even though SISD-RPC offers the better time complexity in the unlimited memory case. The best trade-off is obtained via our SISD-RPC-OPT instantiation.

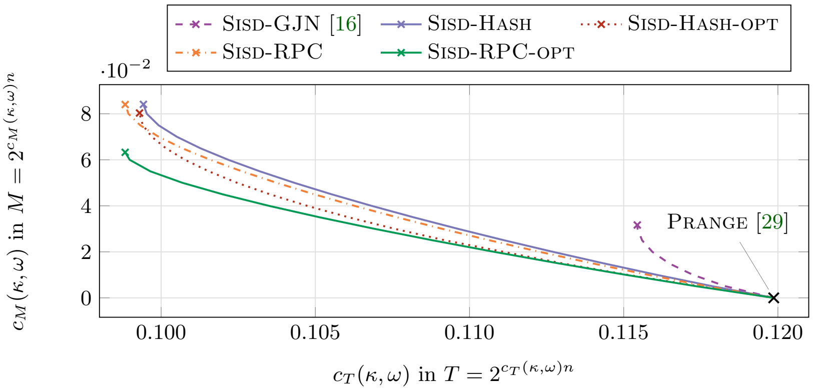


Fig. 6: Time-memory trade-off curves of different SievingISD instantiations, for $\kappa = 0.5$ and $\omega = H^{-1}(0.5)$.

Additionally, in Fig. 7 we compare our best theoretical and practical instantiations, i.e., SISD-RPC-OPT and SISD-HASH-OPT, against time-memory trade-offs based on the MMT and BJMM algorithm, recently proposed by Esser and Zweydinger [13], labeled EZ-BJMM and EZ-MMT. We also give the implicit trade-off resulting from the interpolation of the Both-May algorithm to Prange's memoryless procedure.

For high amounts of memory, unsurprisingly, BOTH-MAY offers the best instantiations, as it achieves the best running time in the unlimited memory case. However, note that our SISD-RPC-OPT instantiation for memories smaller than $2^{0.04n}$ achieves a similar trade-off behaviour. For very small instantiations with less than $2^{0.02n}$ the EZ-BJMM achieves the best runtime. Considering practical instantiations, we find that our SISD-RPC-OPT outperforms the usually applied EZ-MMT for any memory larger than $2^{0.015n}$ and even the EZ-BJMM for memories larger than $2^{0.035n}$.

6 Collisions and unique solutions

In this section, we provide practical experiments verifying our heuristic assumption. Informally, our Heuristic 1 states that the dependencies introduced by constructing the elements as iterative sums do not affect the performance of the near neighbor algorithms, especially in later sieving iterations. Further, we

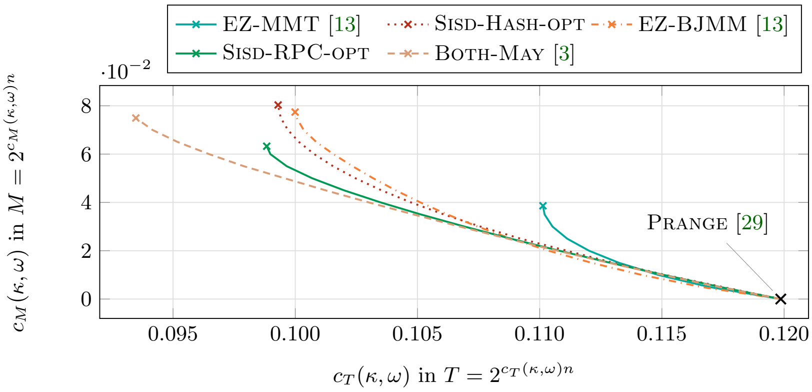


Fig. 7: Time-memory trade-off curves of best SievingISD instantiations in comparison to conventional ISD trade-offs, for $\kappa = 0.5$ and $\omega = H^{-1}(0.5)$.

assume that the probability of any short vector being contained in the final list is also not influenced by those dependencies.

Note that both assumptions hold true if the list throughout all sieving iterations remains close enough to a list where each element is drawn uniformly and independently at random.

In the following we first model the amount of uniquely generated vectors in each near neighbor search iteration in the uniform random case. We then compare this theoretical model against our experimental data.

6.1 Model

Let $N \in \mathbb{N}$ be the input list size to the near neighbor algorithm. Recall that we want to choose N according to Eq. (C 2) such that there exist again N close vectors as pairwise sums of this input list. On average, half of these vectors already belong to the next code in the tower and could be directly forwarded to the next sieving step. However, as this complicates the theoretical model and might introduce additional dependencies, we disregard those vectors. Instead, we only use the other half as an input list for the near neighbor search.[10] The expected number of newly generated vectors of weight w is, hence

$$K := \frac{M(M-1)}{2} \frac{\binom{w}{w/2}\binom{n-w}{w/2}}{\binom{n}{w}}, \tag{17}$$

where $M = N/2$ is the expected number of vectors used for forming the pairs, which gives $\frac{M(M-1)}{2}$ pairs in total, and $\frac{\binom{w}{w/2}\binom{n-w}{w/2}}{\binom{n}{w}}$ is the probability that a given pair sums to weight w.

[10] As a side-effect, all generated sums are then included in the next code in the tower since the sum of two vectors, each of which does not belong to the next code, does belong to this next code.

Amount of unique vectors. We observe here that two different pairs of vectors might have the same sum, in which case we obtain a collision between the newly generated vectors. Next we estimate the number of (multi-)collisions that occur or, equivalently, we count the number of uniquely generated vectors.

If we model the newly generated pairs as being uniform and independent in the set of possible solutions $\mathcal{D}_i = \mathcal{S}_w^n \cap \mathcal{C}_i$, the answer is given via the following lemma, where the set $\mathcal{D}_i$ has expected size $D_i \approx \binom{n}{w}2^{-i}$.

Lemma 6.1 (Unique Solutions). *Let U_K be a random variable that counts the number of different elements obtained in a list that consists of K uniformly and independently sampled elements from a set of size D. Its expectation is*

$$\mathbb{E}[U_K] = \frac{1 - c^K}{1 - c},$$

where $c = 1 - 1/D$.

Proof. We set up the following recurrence relation that describes the effect on U_K when adding a newly generated sample to the list

$$U_{K+1} = U_K + X_K.$$

In this relation $X_K \in \{0, 1\}$ is a random variable that is equal to 1 if a newly generated vector does not collide with the previously generated ones and 0 otherwise. The probability that $X_K = 1$ knowing U_K is $p = 1 - U_K/D$. By linearity of expectation, we have

$$E[U_{K+1}] = E[U_K] + E[X_K] = 1 + E[U_K](1 - 1/D), \tag{18}$$

Note further that $E[U_1] = 1$ (if there is only one element generated, there can be no collision). It remains to observe that $E[U_K] = \frac{1-c^K}{1-c}$ where $c = \big(1 - 1/D\big)$ is a solution to Eq. (18) and that it satisfies the initial condition $E[U_1] = 1$.

6.2 Experiments

We implemented a simple quadratic sieve to obtain practical data points for the comparison to the theoretical model. Our code is available at `https://github.com/setinski/Sieving-For-Codes`.

In Fig. 8a we present the comparison between the modeled prediction for the number of newly generated vectors from Eq. (17) and the experimental data we obtained by running the sieve algorithm. The subsequent Fig. 8b illustrates the number of collisions generated in each sieving step in 10 independent experiments in comparison to the theoretical model provided by Lemma 6.1.

In the experiments, we set $N = 4.1 \cdot \binom{n}{w} \Big/ \binom{w}{w/2}\binom{n-w}{w/2}$ according to the Constraint Eq. (C 2) for maintaining the list size, with a very small margin (note the constant 4.1 rather than 4). We start from a random list of weight-w codewords from the full code $\mathbb{F}_2^n$, and perform iterative sieving steps until no short vectors are found anymore. Note that Constraint Eq. (C 1) states that we require $N \leq \binom{n}{w} 2^{-i} = D_i$ for N different solutions to exist. We therefore expect to start seeing more collisions when D_i approaches N, i.e., we reach saturation of the sphere, which leads to a degeneration of the list size.

Model vs. Experiments. Recall that the theoretical model from Eq. (17) predicts the outcome of the experiment in case of independent and uniformly at random drawn list elements. We observe in our experiments that the dependencies introduced through multiple iterations lead to a slightly higher number of newly generated vectors than predicted (Fig. 8a). However, note that the number quickly converges to a stable value after a few sieving steps, indicating that the effect does not amplify further.

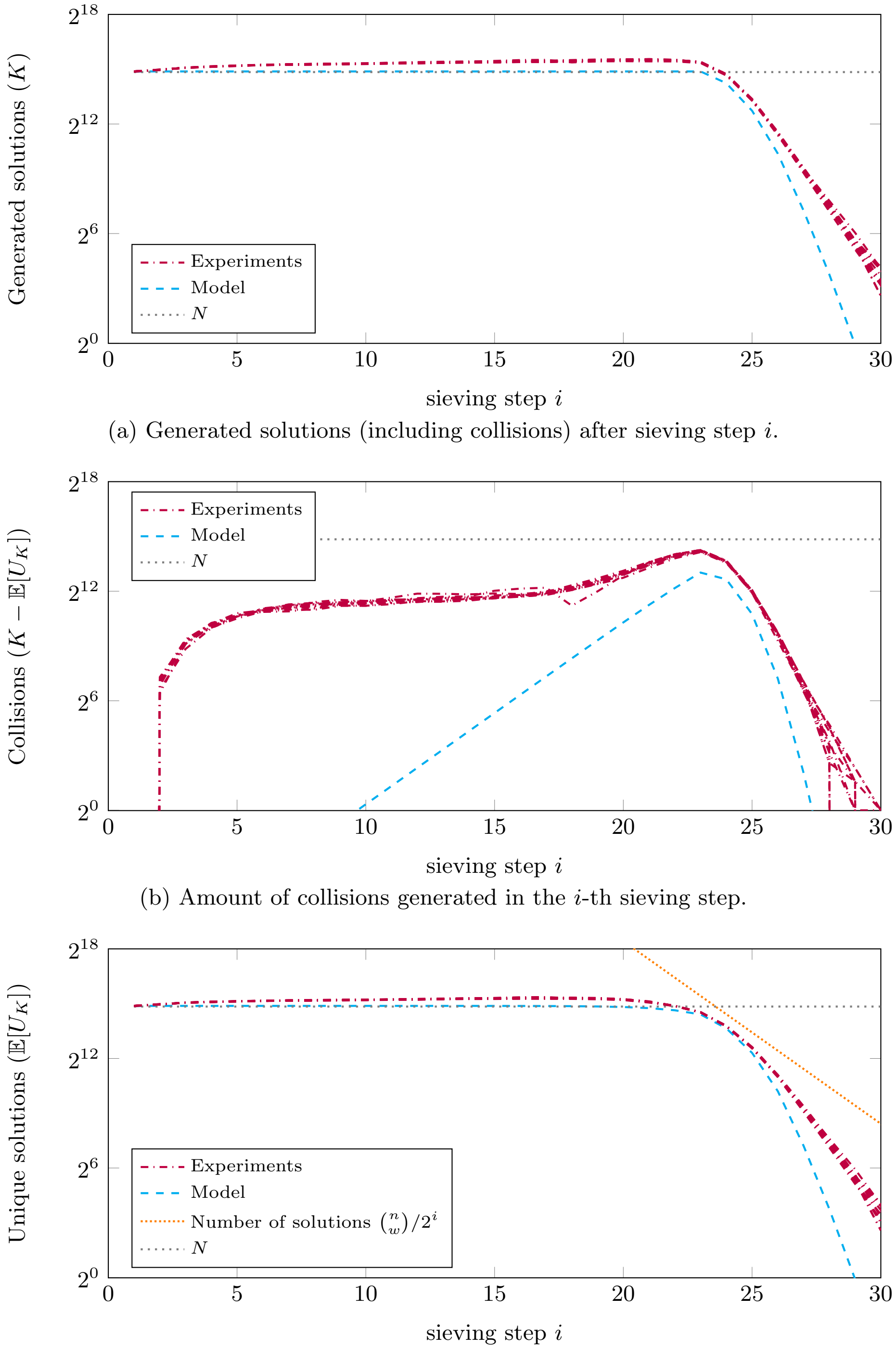
On the other hand, we also observe that the overall number of collisions is higher than predicted (Fig. 8b). The theoretical model for the number of collisions is obtained as the difference between the newly generated vectors (Eq. (17)) and the expected amount of uniquely generated vectors (Lemma 6.1). Here, the theoretical model from Lemma 6.1 predicts the outcome of the experiment in case the weight-w vectors are sampled uniformly and independently at random from the set $\mathcal{D}_i$ in each sieving step i. This assumption makes the theoretical analysis cleaner, but potentially introduces deviations as vectors are, even in the case of lists containing independent elements, constructed as pairwise sums. The difference between the experimental data and the theoretical prediction is therefore likely caused by dependencies as well as a slight bias in the theoretical modeling.

However, we find that the number of extra solutions outweighs the number of collisions, implying that in total we find *more* unique solutions in the experiments than given by the theoretical prediction (see Fig. 8c). This effect can also be observed for other sets of parameters, as depicted in Fig. 9, where we observed the number of uniquely generated vectors in 10 independent experiments for different code lengths n and a fixed small weight w.

Relation to Heuristic 1. Let's first discuss the second part of Heuristic 1. Here the heuristic states that the probability of any vector being contained in the list after i sieving steps is about N/D_i. This ensures that once a suitable subcode in Algorithm 2 is chosen, the target vector is returned with this probability. Our experiments now show that we generate even more unique solutions than the theoretical model predicts. Note that this does not imply that those unique solutions are uniformly distributed. However, in the context of the SievingISD algorithm the subcode used for the sieving routine (and with it the target vector) are randomly drawn in every iteration. For a uniformly random target vector, the probability bound is implied as long as N distinct weight-w codewords are returned.

The first part of the heuristic makes an assumption about the time complexity of the iterative application of the near neighbor algorithms. Overall our experiments indicate that the list distributions do not significantly deviate from the theoretical model. While this supports the heuristic, full verification can only be achieved by implementing the different near neighbor approaches. Our simple quadratic sieve does not deviate in runtime, as it always obtains the worst case complexity of N^2.

Extending the Experiments. Our experiments indicate the validity of the heuristic assumption. However, an in-depth verification requires a more elaborate implementation effort. This includes especially the more advanced near neighbor routines to verify the first part of the heuristic for those concrete procedures. This would then also allow us to extend the experiments to even larger parameters currently inaccessible to our implementation.

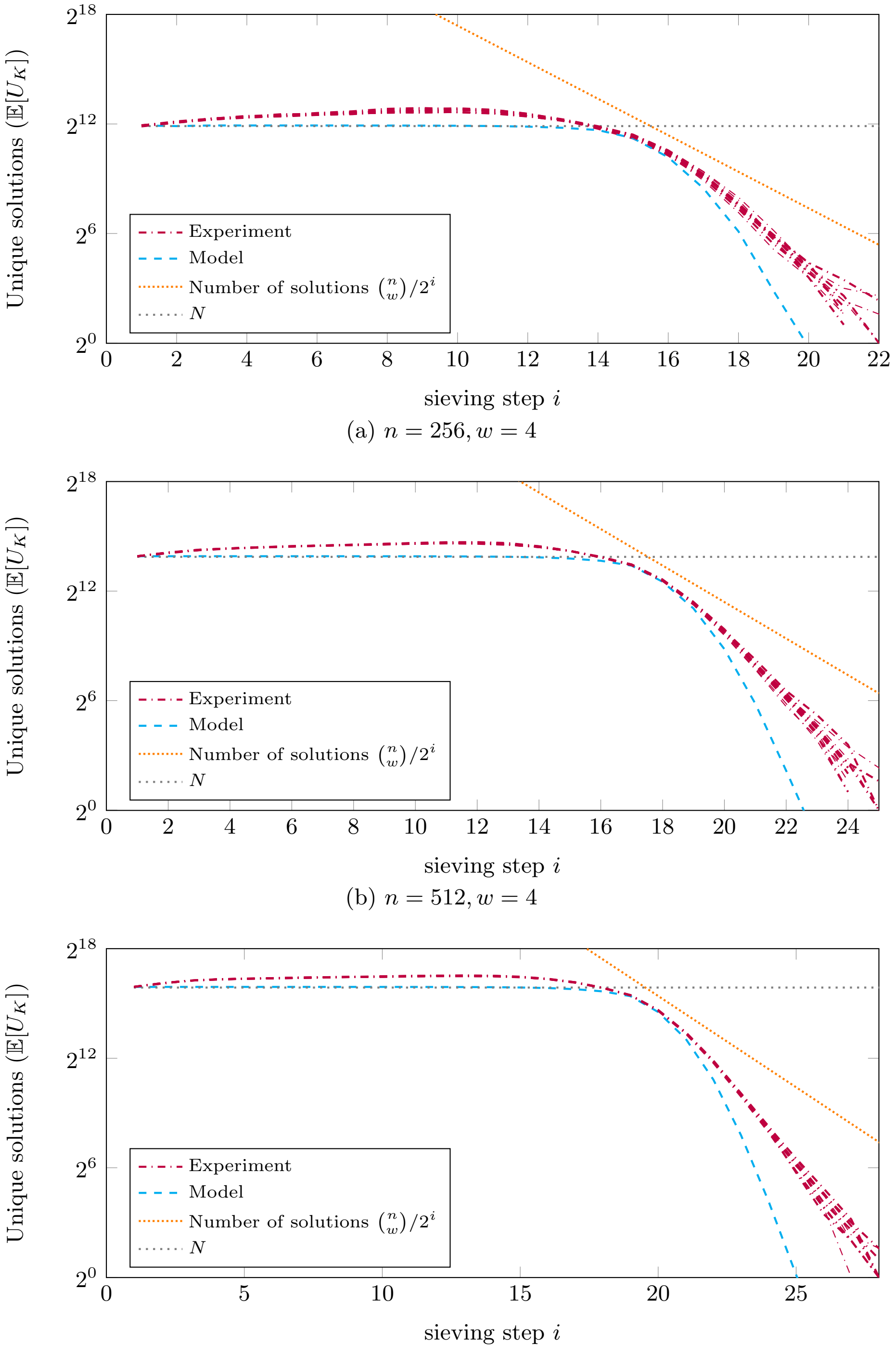


(a) Generated solutions (including collisions) after sieving step i.

(b) Amount of collisions generated in the i-th sieving step.

(c) Amount of uniquely generated solutions after sieving step i.

Fig. 8: Theoretical prediction and experimental data for $n = 256, w = 6$ in a logarithmic scale



(a) $n = 256, w = 4$

(b) $n = 512, w = 4$

(c) $n = 1024, w = 4$

Fig. 9: Amount of unique solutions in theoretical prediction vs. experimental data for different parameter sets (in a logarithmic scale)

References

1. Becker, A., Ducas, L., Gama, N., Laarhoven, T.: New directions in nearest neighbor searching with applications to lattice sieving. In: Krauthgamer, R. (ed.) 27th SODA. pp. 10–24. ACM-SIAM (Jan 2016). https://doi.org/10.1137/1.9781611974331.ch2
2. Becker, A., Joux, A., May, A., Meurer, A.: Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 520–536. Springer, Heidelberg (Apr 2012). https://doi.org/10.1007/978-3-642-29011-4_31
3. Both, L., May, A.: Decoding linear codes with high error rate and its impact for LPN security. In: Lange, T., Steinwandt, R. (eds.) Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018. pp. 25–46. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-79063-3_2
4. Carrier, K.: Recherche de presque-collisions pour le décodage et la reconnaissance de codes correcteurs. Ph.D. thesis, Sorbonne université (2020)
5. Carrier, K., Debris-Alazard, T., Meyer-Hilfiger, C., Tillich, J.P.: Statistical decoding 2.0: Reducing decoding to LPN. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part IV. LNCS, vol. 13794, pp. 477–507. Springer, Heidelberg (Dec 2022). https://doi.org/10.1007/978-3-031-22972-5_17
6. Cooper, C.: On the distribution of rank of a random matrix over a finite field. Random Struct. Algorithms **17**(3-4), 197–212 (2000)
7. 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). https://doi.org/10.1007/978-3-319-70503-3_24
8. Dubiner, M.: Bucketing coding and information theory for the statistical high-dimensional nearest-neighbor problem. IEEE Transactions on Information Theory **56**(8), 4166–4179 (2010)
9. Esser, A.: Revisiting nearest-neighbor-based information set decoding. In: IMA International Conference on Cryptography and Coding. pp. 34–54. Springer (2023)
10. Esser, A., Heuer, F., Kübler, R., May, A., Sohler, C.: Dissection-BKW. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 638–666. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96881-0_22
11. Esser, A., Kübler, R., Zweydinger, F.: A faster algorithm for finding closest pairs in hamming metric. In: Bojanczyk, M., Chekuri, C. (eds.) 41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2021, December 15-17, 2021, Virtual Conference. LIPIcs, vol. 213, pp. 20:1–20:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.FSTTCS.2021.20, https://doi.org/10.4230/LIPIcs.FSTTCS.2021.20
12. Esser, A., May, A., Zweydinger, F.: McEliece needs a break - solving McEliece-1284 and quasi-cyclic-2918 with modern ISD. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part III. LNCS, vol. 13277, pp. 433–457. Springer, Heidelberg (May / Jun 2022). https://doi.org/10.1007/978-3-031-07082-2_16
13. Esser, A., Zweydinger, F.: New time-memory trade-offs for subset sum: Improving ISD in theory and practice. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Part V. LNCS, vol. 14008, pp. 360–390. Springer, Heidelberg (Apr 2023). https://doi.org/10.1007/978-3-031-30589-4_13

14. Fincke, U., Pohst, M.: A procedure for determining algebraic integers of given norm. In: van Hulzen, J.A. (ed.) Computer Algebra. pp. 194–202 (1983)
15. Finiasz, M., Sendrier, N.: Security bounds for the design of code-based cryptosystems. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 88–105. Springer, Heidelberg (Dec 2009). https://doi.org/10.1007/978-3-642-10366-7_6
16. Guo, Q., Johansson, T., Nguyen, V.: A new sieving-style information-set decoding algorithm. Cryptology ePrint Archive, Report 2023/247 (2023), https://eprint.iacr.org/2023/247
17. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: 30th ACM STOC. pp. 604–613. ACM Press (May 1998). https://doi.org/10.1145/276698.276876
18. Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing. p. 193–206. STOC '83 (1983). https://doi.org/10.1145/800061.808749
19. 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). https://doi.org/10.1007/978-3-662-47989-6_1
20. 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). https://doi.org/10.1007/978-3-319-22174-8_6
21. Levieil, 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). https://doi.org/10.1007/11832072_24
22. Liu, H., Yu, Y.: A non-heuristic approach to time-space tradeoffs and optimizations for BKW. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part III. LNCS, vol. 13793, pp. 741–770. Springer, Heidelberg (Dec 2022). https://doi.org/10.1007/978-3-031-22969-5_25
23. May, A., Meurer, A., Thomae, E.: Decoding random linear codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 107–124. Springer, Heidelberg (Dec 2011). https://doi.org/10.1007/978-3-642-25385-0_6
24. May, A., Ozerov, I.: On computing nearest neighbors with applications to decoding of binary linear codes. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part I. LNCS, vol. 9056, pp. 203–228. Springer, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46800-5_9
25. Minder, L., Sinclair, A.: The extended k-tree algorithm. Journal of Cryptology **25**(2), 349–382 (Apr 2012). https://doi.org/10.1007/s00145-011-9097-y
26. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York, NY, USA (2005)
27. Nguyen, P.Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. Journal of Mathematical Cryptology **2**(2), 181–207 (2008). https://doi.org/doi:10.1515/JMC.2008.009, https://doi.org/10.1515/JMC.2008.009
28. Pagh, R.: Locality-sensitive hashing without false negatives. In: Krauthgamer, R. (ed.) 27th SODA. pp. 1–9. ACM-SIAM (Jan 2016). https://doi.org/10.1137/1.9781611974331.ch1
29. Prange, E.: The use of information sets in decoding cyclic codes. IRE Transactions on Information Theory **8**(5), 5–9 (1962)

30. Stern, J.: A method for finding codewords of small weight. In: International Colloquium on Coding Theory and Applications. pp. 106–113. Springer (1988)
31. Wagner, D.: A generalized birthday problem. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer, Heidelberg (Aug 2002). https://doi.org/10.1007/3-540-45708-9_19