LPN Decoded

Andre Esser, Robert Kübler(✉), and Alexander May

Faculty of Mathematics, Horst Görtz Institute for IT Security,
Ruhr University, Bochum, Germany
{andre.esser,robert.kuebler,alex.may}@rub.de

Abstract. We propose new algorithms with *small memory consumption* for the Learning Parity with Noise (LPN) problem, both classically and quantumly. Our goal is to predict the hardness of LPN depending on both parameters, its dimension k and its noise rate τ, as accurately as possible both in theory and practice. Therefore, we analyze our algorithms asymptotically, run experiments on medium size parameters and provide bit complexity predictions for large parameters.

Our new algorithms are modifications and extensions of the simple Gaussian elimination algorithm with recent advanced techniques for decoding random linear codes. Moreover, we enhance our algorithms by the dimension reduction technique from Blum, Kalai, Wasserman. This results in a hybrid algorithm that is capable for achieving the best currently known run time for any fixed amount of memory.

On the asymptotic side, we achieve significant improvements for the run time exponents, both classically and quantumly. To the best of our knowledge, we provide the first quantum algorithms for LPN.

Due to the small memory consumption of our algorithms, we are able to solve for the first time LPN instances of medium size, e.g. with $k = 243, \tau = \frac{1}{8}$ in only 15 days on 64 threads.

Our algorithms result in bit complexity prediction that require relatively large k for small τ. For instance for small noise LPN with $\tau = \frac{1}{\sqrt{k}}$, we predict 80-bit classical and only 64-bit quantum security for $k \geq 2048$. For the common cryptographic choice $k = 512, \tau = \frac{1}{8}$, we achieve with limited memory classically 102-bit and quantumly 69-bit security.

Keywords: LPN key size · Information set decoding · Grover · BKW

1 Introduction

With the upcoming NIST initiative for recommending quantum-secure public key cryptosystems [1], it becomes even more urgent and mandatory to properly select cryptographic key sizes with a well-defined security level, both classically and of course also quantumly. Therefore, the cryptographic community has to establish for the most prominent hardness problems, e.g. in the areas of codes, lattices, multivariate and isogenies, predictions for solving cryptographic instances with security levels of 128 bit and above.

J. Katz and H. Shacham (Eds.): CRYPTO 2017, Part II, LNCS 10402, pp. 486–514, 2017.
DOI: 10.1007/978-3-319-63715-0_17

The choice of key sizes has naturally been a tradeoff between efficiency and security. On the one hand, one would like to choose small parameters that allow for efficient implementations. On the other hand, one is usually quite conservative in estimating which parameters can be broken within say 2^{128} steps. While giving conservative security estimates is in general good, we believe that this practice is often disproportionate in cryptographic research.

For instance, when selecting the best algorithm, cryptographers usually completely ignore memory consumption. And quite often, the best time complexity T is only achieved with memory consumption as large as T. An example with such huge memory requirement is the Blum-Kalai-Wasserman (BKW) algorithm [7] for solving LPN. But when implementing an algorithm in practice, memory consumption is the main limiting factor. While performing 2^{60} steps is even doable on smallish computing clusters in a reasonable amount of time, getting an amount of 2^{60} of RAM is clearly out of reach. If one has to rely on additional hard disk space, the running time will increase drastically.

An Internet investigation shows that nowadays the largest supercomputers[1] have a RAM of at most 1.6 PB $< 2^{54}$ bits. Putting some safety margin, it seems to be fair to say that any algorithm with memory consumption larger than 2^{60} bits cannot be instantiated in practice. In the course of the paper we will also consider a higher safety margin of 2^{80} bits.

Hence, there is a need for finding algorithms for post-quantum problems that can be instantiated with *small memory*, in order to run them on medium size instances for an accurate extrapolation to cryptographic key sizes. For the selection of key sizes, one might safely restrict to algorithms that do not exceed a certain amount of memory, like e.g. 2^{60} bits. Belaïd et al. [5] considered a related model in which an attacker has limited LPN samples and memory. However, we do not want to limit the number of LPN samples.

Ideally, we would design algorithms whose running time benefit from any fixed amount of memory. Let us assume that we have M bits of RAM on our computing facility. The main research question is then which optimal running time can be achieved when (fully) using this amount.

Our goal is to answer this question for Learning Parity with Noise (LPN). LPN is the basis for many code-based constructions and can be seen as a special instance of Learning with Errors (LWE) [29]. In the LPN problem, one has to learn a secret $\mathbf{s} \in \mathbb{F}_2^k$ using access to an oracle that provides samples of the form $(\mathbf{a}_i, b_i)$, where $\mathbf{a}_i$ is uniformly at random from $\mathbb{F}_2^k$, and $b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i$ for some error $e_i \in \{0, 1\}$ with $\Pr[e_i = 1] = \tau$. Hence, LPN is a two-parameter problem with dimension k and error rate $\tau \in [0, \frac{1}{2})$.

Naturally, the problem becomes harder with increasing k and τ. For $\tau = 0$ we can easily draw k samples with linearly independent $\mathbf{a}_i$ and solve for $\mathbf{s}$ via Gaussian elimination. This algorithm can simply be generalized to any $\tau \in [0, \frac{1}{2})$, by drawing k samples in each iteration, computing a candidate $\mathbf{s}'$, and test whether $\mathbf{s} = \mathbf{s}'$. Notice that $\mathbf{s} = \mathbf{s}'$ iff in this iteration all samples are error-free.

[1] E.g. the IBM 20-Petaflops cluster installed in Sequoia, Lawrence Livermore National Laboratory, California [2].

This algorithm, that we call `Gauss`, seems to be somewhat folklore. To the best of our knowledge it was first used in 2008 by Carrijo et al. [11], and has been e.g. analyzed in Bogos et al. [8]. The benefits of `Gauss` are that it consumes only small memory and performs well for small noise τ, e.g. for the currently required choice of $\tau = \frac{1}{\sqrt{k}}$ in the public key encryption schemes of Alekhnovich [3], Damgård, Park [12], Döttling, Müller-Quade, Anderson [13] and Duc, Vaudenay [14].

For constant noise τ, as used e.g. in the HB family of protocols [16,20,21] and their extensions [19,23], currently the best known algorithm is `BKW`, due to Blum, Kalai and Wasserman [7] with running time, memory consumption and sample complexity $2^{\mathcal{O}(k/\log k)}$. `BKW` has been widely studied in the cryptographic literature and there are several improvements in practice due to Fossorier et al. [15], Levieil, Fouque [24], Lyubashevsky [25], Guo, Johansson, Löndahl [18] and Zhang, Jiao, Mingsheng [30]. While `BKW` offers for large τ the best running time, it cannot be implemented even for medium size LPN parameters due to its huge memory consumption. But without having any experimental results, it is an error-prone process to predict security levels. This also led to some discussion about the accuracy of predictions [9].

`Gauss` and `BKW` are the starting point of our paper. We revisit both in Sect. 2, where we analyze them asymptotically and show that `BKW` has a very bad dependency on τ with a running time of $2^{\mathcal{O}(k/\log(\frac{k}{\tau}))}$. So even for τ as small as $\tau = \frac{1}{k}$, the running time remains $2^{\mathcal{O}(k/\log k)}$.

Another drawback of `Gauss` and `BKW` is their large sample complexity, i.e. the number of calls to an LPN oracle, which is for both algorithms as large as their running time. Since the LPN oracle is by definition classical, this prevents any possible speed-ups by quantum search techniques, e.g. by Grover search [17].

Therefore, we will first reduce the number of samples to a pool of only $n = \text{poly}(k)$ samples. Out of these n samples, we look for a set of k error-free samples similar to `Gauss`. The resulting algorithm `Pooled Gauss` (Sect. 4) has the same time and memory complexity as `Gauss`, while consuming far fewer samples. This immediately gives rise to a quantum version, for which we save a square root in the run time via Grover search.

Another benefit of having small sample complexity is that we can add a preprocessing step that reduces the dimension of our LPN instances via intensive use of the LPN oracle. The resulting algorithm that we call `Well-Pooled Gauss` (Sect. 5.1) offers a significantly reduced time complexity.

In a nutshell, `Well-Pooled Gauss` has a simple *preprocessing step* that decreases the LPN dimension, and then a *decoding step* via Gaussian elimination. The preprocessing step can be improved by more sophisticated dimension reduction methods such as `BKW`. This comes at the cost of using some memory, but we can control the required memory by adjusting the amount of dimension reduction. Altogether, this results in Algorithm `Hybrid` (Sect. 5.3) that for any given memory M first reduces the dimension with full memory use, and second runs Gaussian elimination on the dimension-reduced, and thus easier, LPN instance.

Another nice feature of `Hybrid` is that its preprocessing step allows to easily include many of the recent `BKW` optimizations [18,24,30].

Moreover, we are also able to improve on the decoding step by replacing Gaussian elimination with more sophisticated information set decoding algorithms, like `Ball-Collision Decoding` of Bernstein, Lange, Peters [6], `MMT` of May, Meurer, Thomae [26], `BJMM` of Becker et al. [4] or `May-Ozerov` [27]. For our purpose of decoding LPN instances, it turns out that the `MMT` algorithm tuned to the LPN setting performs best. The resulting algorithm that we call `Well-Pooled MMT` is studied in Sect. 5.4.

Table 1 provides a more detailed overview of our algorithms and results. For ease of exposition, in Table 1 we omit all small error terms in run times, like $(1 + o(1))$-factors or $\tilde{\mathcal{O}}$-notation.

Table 1. Overview of our results, $f(\tau) := \log\left(\frac{1}{1-\tau}\right)$

Algorithm	Time	Samples	Memory	Quantum
`BKW` (Theorem 1)	$2^{\frac{k}{\log\left(\frac{k}{\tau}\right)}}$	=Time	=Time	Inapplicable
`Gauss` (Theorem 2)	$2^{f(\tau)k}$	=Time	k^2	Inapplicable
`Pooled Gauss` (Theorems 3 & 4)	$2^{f(\tau)k}$	k^2	k^3	$2^{\frac{f(\tau)k}{2}}$
`Pooled Gauss`, $\tau(k) \to 0$ (Corollary 1)	$e^{\tau k}$	k^2	k^3	$e^{\frac{\tau k}{2}}$
`Well-Pooled Gauss` (Theorems 5 & 6)	$2^{\frac{f(\tau)}{1+f(\tau)}k}$	=Time	k^3	$2^{\frac{f(\tau)}{2+f(\tau)}k}$
`Hybrid` (Theorem 7)	$2^{f(\tau)k}$ to $2^{\frac{k}{\log\left(\frac{k}{\tau}\right)}}$	=Time	k^3 to $2^{\frac{k}{\log\left(\frac{k}{\tau}\right)}}$	Applicable
`Well-Pooled MMT` (Sect. 5.4)	$\approx 2^{\frac{f\left(\frac{7}{6}\tau\right)}{\log\left(\frac{12}{5}\right)+f\left(\frac{7}{6}\tau\right)}}$	=Time	$< \sqrt{\text{Time}}$	Applicable [22]

2 Preliminaries and the LPN Problem

2.1 Preliminaries

Let us first fix some notation. With log we denote the binary logarithm. For a positive integer $n \in \mathbb{N}$ we define $[n] := \{1, 2, \ldots, n\}$. Let M be a set and $k \in \mathbb{N}$. Then $\binom{M}{k}$ is the set of all subsets of M of size k. In particular, $\binom{[n]}{k}$ is the set of size-k subsets of $\{1, \ldots, n\}$.

Let $A \in \mathbb{F}_2^{n \times k}$, $\mathbf{b} \in \mathbb{F}_2^k$ and $I = \{i_1, \ldots, i_\ell\} \subseteq [k]$. Then A_I consists of the rows indexed by I and $\mathbf{b}_I$ consists of the entries indexed by I, e.g.

$$A_I := \begin{pmatrix} — & a_{i_1} & — \\ & \vdots & \\ — & a_{i_\ell} & — \end{pmatrix} \text{ and } \mathbf{b}_I := (b_{i_1}, \ldots, b_{i_\ell})^t.$$

Let $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbb{F}_2^n$. Then we call $\mathrm{wt}(\mathbf{v}) := \|\mathbf{v}\|_1 = |\{i \in [n] \mid v_i \neq 0\}|$ the Hamming weight (or just weight) of $\mathbf{v}$. A linear code $\mathcal{C}$ is a subspace of $\mathbb{F}_2^n$. If $\dim(\mathcal{C}) = k$ and $d := \min_{0 \neq c \in \mathcal{C}} \{\mathrm{wt}(c)\}$, then we call $\mathcal{C}$ an $[n, k, d]$ code.

This implies $\mathcal{C} = \mathrm{im}(G)$ for some matrices $G \in \mathbb{F}_2^{n \times k}$ with rank k. We call G a *generator matrix* of $\mathcal{C}$. For a random matrix G we call $\mathcal{C} = \mathrm{im}(G)$ a *random linear code*.

For a finite set M we write the uniform distribution on M by $\mathcal{U}(M)$. Moreover, we denote by Ber_τ the Bernoulli distribution with parameter τ, i.e., $e \sim \mathrm{Ber}_\tau$ means that we draw a 0–1 valued random variable e with $\Pr[e = 1] = \tau$.

The binomial distribution is denoted as $\mathrm{Bin}_{n,p}$ and can be seen as the sum of n independently identically distributed Ber_p variables. If $X \sim \mathrm{Bin}_{n,p}$, we have $\Pr[X = i] = \binom{n}{i} p^i (1-p)^{n-i}$. In the course of the paper we will deal with the question: Given $p = p(k)$ and $N = N(k)$, how large does the number of Bernoulli trials $n = n(k)$ have to be, such that $\Pr[X \geq N] \geq 1 - \mathrm{negl}(k)$? Here k is a security parameter and $\mathrm{negl}(k) = o(\frac{1}{\mathrm{poly}(k)}) = k^{-\omega(1)}$. We call probabilities of the form $1 - \mathrm{negl}(k)$ overwhelming.

For example, setting $n = \frac{N}{p}$ only yields $\Pr[X \geq N] \geq \frac{1}{2}$, so n has to be larger than that. How much larger it has to be is answered by

Lemma 1. *Let $X \sim Bin_{n,p>0}$ and $0 \leq N \leq np$. If $n = \Theta\left(\frac{N + \log^2 k}{p}\right)$, then we have $\Pr[X \geq N] \geq 1 - k^{-\omega(1)}$.*

Proof. The Chernoff bounds give us $\Pr[X \geq N] \geq 1 - e^{-\frac{(np-N)^2}{2np}}$. If we set $n \geq \frac{N + \log^2 k + \sqrt{N \cdot \log^2 k + \log^4 k}}{p}$, for example $n = \frac{2N + \log^2 k}{p}$, we get

$$1 - e^{-\frac{(np-N)^2}{2np}} = 1 - e^{-\frac{(N+\log^2 k)^2}{4N + 2\log^2 k}} \geq 1 - e^{-\frac{N + \log^2 k}{4}} \geq 1 - e^{\frac{\log^2 k}{4}} = 1 - k^{-\omega(1)}.$$

□

For $N = 1$, Lemma 1 gives us the amount of Bernoulli trials we need, until we get a success with overwhelming probability, i.e. $\frac{\log^2 k}{p}$ suffices. One can see, that this is only slightly more than $\frac{1}{p}$, the expectation value of a geometric distributed random variable with parameter p.

2.2 The LPN Problem

Let us now formally define the LPN problem.

Definition 1. *In the* $\mathrm{LPN}_{k,\tau}$ problem*, for some secret $\mathbf{s} \in \mathbb{F}_2^k$ and error parameter $\tau \in [0, \frac{1}{2})$ we are given access to an oracle that provides* samples *of the form*

$$(\mathbf{a}_i, b_i) := (\mathbf{a}_i, \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i), \text{for } i = 1, 2, \ldots,$$

where $\mathbf{a}_i \sim \mathcal{U}(\mathbb{F}_2^k)$ and $e_i \sim Ber_\tau$, independently. Our goal is to recover $\mathbf{s}$. We call b_i the corresponding label of $\mathbf{a}_i$.

Notation: *Upon asking m queries, we write $(A, \mathbf{b}) \leftarrow \mathrm{LPN}_{k,\tau}^{m}$ meaning that $A\mathbf{s} = \mathbf{b} + \mathbf{e}$, where the i^{th} row of $A \in \mathbb{F}_2^{m \times k}$ and $\mathbf{b} \in \mathbb{F}_2^m$ present the i^{th} sample.*

Remark 1. We say that an algorithm $\mathtt{A}$ with overwhelming probability solves $\mathrm{LPN}_{k,\tau}$ in running time T, if it *both* terminates within time T and outputs the correct $\mathbf{s}$ with probability $1 - \mathrm{negl}(k)$. This means that $\mathtt{A}$ might not terminate in time T or that $\mathtt{A}$ might output an incorrect $\mathbf{s}'$, but we bound both events by some negligible function in k. Notice that our notion is stronger than just *expected running time* T_e, where the real running time might significantly deviate from T_e with even constant probability.

The error-free case $\mathrm{LPN}_{k,0}$ can be easily solved by obtaining k sample $(\mathbf{a}_i, b_i)$ with linearly independent $\mathbf{a}_i$, and computing via Gaussian elimination

$$\mathbf{s} = A^{-1}\mathbf{b}. \tag{1}$$

However, in case of errors we obtain $\mathbf{s} = A^{-1}\mathbf{b} + A^{-1}\mathbf{e}$, with an accumulated error of $A^{-1}\mathbf{e}$, where $\mathrm{wt}(A^{-1}\mathbf{e})$ is usually large. In other words, Gaussian elimination lets the error grow too fast by adding together too many samples.

The error growth can be made precise in terms of the number n of additions via the following lemma, usually called Piling-up Lemma in the cryptographic literature.

Lemma 2 (Piling-up Lemma). *Let $e_i \sim Ber_\tau$, $i = 1, \ldots, n$ be identically, independently distributed. Then we have $\sum_{i=1}^{n} e_i \sim Ber_{\frac{1}{2} - \frac{1}{2}(1-2\tau)^n}$.*

Proof. $n = 1$ is immediate. Induction over n yields

$$\begin{aligned}
\Pr\left[\sum_{i=1}^{n} e_i = 1\right] &= \Pr\left[\sum_{i=1}^{n-1} e_i = 0\right] \cdot \Pr[e_n = 1] + \Pr\left[\sum_{i=1}^{n-1} e_i = 1\right] \cdot \Pr[e_n = 0] \\
&= \left(\frac{1}{2} + \frac{1}{2}(1-2\tau)^{n-1}\right)\tau + \left(\frac{1}{2} - \frac{1}{2}(1-2\tau)^{n-1}\right)(1-\tau) \\
&= \frac{1}{2} - \frac{1}{2}(1-2\tau)^n.
\end{aligned}$$

□

3 Revisiting Previous Work

3.1 The BKW Algorithm

Blum, Kalai and Wasserman [7] proposed a variant of Gaussian elimination, called BKW algorithm, that performs elimination of whole blocks instead of single coordinates. This results in way less additions of samples, thus controlling the error, at the cost of requiring way more initial LPN samples to perform eliminations.

The following high-level description of BKW eliminates blocks of size d in each of its $c-1$ iterations, resulting in vectors that are sums of 2^{c-1} original samples. We describe only how to compute the first bit of $\mathbf{s}$, the other bits are analogous.

Input: $\mathrm{LPN}_{k,\tau}$ oracle, $\tau > 0$
Output: First bit s_1 of the secret $\mathbf{s} = (s_1, \ldots, s_k)$
Choose $\varepsilon > 0$;
$c := (1-\varepsilon)\log\left(\frac{k}{\tau}\right)$;
$d := \frac{k}{c}$;
$N := \left(c - 1 + \frac{\log^2 k}{(1-2\tau)^{2^c}} + \log^2 k\right)2^d$;
$(A, \mathbf{b}) \leftarrow \mathrm{LPN}_{k,\tau}^N$;
for $i = 1, \ldots, c-1$ **do**
  foreach $j \in \mathbb{F}_2^d$ **do**
    Pick a row $\mathbf{a}_k$ of A with suffix $j|0^{(i-1)d}$ (if any); add $\mathbf{a}_k$ to all the other rows of A with suffix $j|0^{(i-1)d}$, also add corresponding labels;
    Remove the k^{th} row from A and $\mathbf{b}$;
  end
end
$I := \{i \in [N] \mid a_i = u_1 = (1, 0, \ldots, 0)\}$;
return $s_1 =$ the bit which is the majority of all bits in $\mathbf{b}_I$.

Algorithm 1. `BKW`

Blum, Kalai and Wasserman show that, for constant τ, instantiating their algorithm with blocks of size roughly $d = \frac{k}{\log k}$ and $c = \log k$ iterations while using $N = 2^{\mathcal{O}(k/\log k)}$ samples results in running time and memory complexity also $2^{\mathcal{O}(k/\log k)}$.

Since for concrete cryptographic instantiations, we are also interested in the dependence on τ and the constant hidden in the $\mathcal{O}$-notion, we give a slightly more detailed analysis in the following.

Theorem 1. *`BKW` solves $\mathrm{LPN}_{k,\tau}$ for $\tau > 0$ with overwhelming success probability in time, memory and sample complexity* $2^{\frac{k}{\log\left(\frac{k}{\tau}\right)}(1+o(1))}$.

Proof. By our choice in `BKW` we initially start with $N := \left(c - 1 + \frac{\log^2 k}{(1-2\tau)^{2^c}} + \log^2 k\right)2^d$ samples. Every foreach loop reduces the number of samples by at most 2^d, resulting in at least $\left(\frac{\log^2 k}{(1-2\tau)^{2^c}} + \log^2 k\right)2^d$ samples after loop termination.

Let $\mathbf{u}_1 = (1, 0, 0, \ldots, 0)$ be the first unit vector. Among the remaining samples there will be at least $r = \frac{\log^2 k}{(1-2\tau)^{2^c}}$ samples of the form $(\mathbf{u}_1, \mathbf{s}_1 + e)$ for some error $e \in \{0, 1\}$ with overwhelming probability according to Lemma 1. Since our r remaining samples are generated as a sum of 2^{c-1} initial samples, the Piling-up lemma (Lemma 2) yields $e \sim \mathrm{Ber}_{\frac{1}{2} - \frac{1}{2}(1-2\tau)^{2^{c-1}}}$.

Hence, e has a bias of $\bar{b} = \frac{1}{2}(1-2\tau)^{2^{c-1}}$. An easy Chernoff bound argument shows that having $\bar{b}^{-2}$ samples is sufficient to obtain s_1 with constant success

probability by majority vote. Since our number r is larger than $\bar{b}^{-2}$ by a factor of $\frac{\log^2 k}{4}$, we even obtain s_1 with overwhelming success probability. By repeating this process for all bits $s_1, \ldots, s_k$ a union bound shows that we lose a factor of at most k in the success probability, meaning that we can recover $\mathbf{s}$ with overwhelming success probability.

The algorithm's run time and memory consumption is (up to polynomial factors) dominated by its sample complexity, which by our choice of c, d is

$$N = \big(c - 1 + \frac{\log^2 k}{(1-2\tau)^{2^c}} + \log^2 k\big)2^d = 2^{\mathcal{O}(k^{1-\varepsilon}) + \frac{1}{1-\varepsilon}\cdot\frac{k}{\log(\frac{k}{\tau})}} = 2^{\frac{k}{\log(\frac{k}{\tau})}(1+o(1))}.$$

□

We would like to point out that in Theorem 1 the running time $2^{k/\log(\frac{k}{\tau})(1+o(1))}$ only very slowly decreases with τ. Notice that even for τ as small as $\Theta(\frac{1}{k})$ we still obtain a running time of $2^{\frac{1}{2}k/\log k(1+o(1))}$, while $\mathrm{LPN}_{k,\mathcal{O}(\frac{1}{k})}$ clearly can be solved in polynomial time via correcting $\mathcal{O}(1)$ errors and running Gaussian elimination.

3.2 Gauss

The following simple Algorithm 2, that we call `Gauss`, is the most natural extension of Gaussian elimination from Sect. 2.2, where one repeats sampling k linearly independent $\mathbf{a}_i$ until they are all error-free.

In each iteration of `Gauss` we simply assume error-freeness and compute a candidate secret key $\mathbf{s}' = A^{-1}\mathbf{b}$ as in Eq. (1). We take fresh samples to test our hypothesis, whether we were indeed in the error-free case and hence $\mathbf{s}' = \mathbf{s}$.

Notice that we are in the error-free case with probability $(1-\tau)^k$. Hence, Algorithm 2 has up to polynomial factors expected running time $(\frac{1}{1-\tau})^k$, provided that `Test` can be carried out in polynomial time. Thus in comparison to `BKW` in Sect. 3.1, we obtain a much better dependence on τ. For instance for $\tau = \mathcal{O}(\frac{1}{k})$, we obtain polynomial running time, as one would expect.

Input: $\mathrm{LPN}_{k,\tau}$ oracle, τ
Output: secret $\mathbf{s}$
repeat
  repeat
    $(A, \mathbf{b}) \leftarrow \mathrm{LPN}_{k,\tau}^k$;
  until $A \in \mathrm{GL}_k(\mathbb{F}_2)$;
  $s' := A^{-1}\mathbf{b}$;
until $\mathtt{Test}(\mathbf{s}', \tau, \frac{1}{2^k}, (\frac{1-\tau}{2})^k\ ) = \mathrm{Accept}$;
return $\mathbf{s}'$;

Algorithm 2. `Gauss`

Basically our algorithm `Test` computes for sufficiently many fresh LPN sample $(A, \mathbf{b}) \leftarrow \mathrm{LPN}_{k,\tau}^m$ whether $A\mathbf{s}' + \mathbf{b}$ is closer to $\mathrm{Ber}_{m,\tau}$ or to $\mathrm{Ber}_{m,\frac{1}{2}}$ via checking whether its weight is close to τm or $\frac{m}{2}$, respectively.

We have designed `Test` in a flexible way that allows us to control the two-sided error probabilities $\Pr[\texttt{Test rejects} \mid \mathbf{s}' = \mathbf{s}]$ for rejecting the right candidate and $\Pr[\texttt{Test accepts} \mid \mathbf{s}' \neq \mathbf{s}]$ for accepting an incorrect $\mathbf{s}'$ via two parameters α, β. Throughout this paper, we will tune these parameters α, β to guarantee that all subsequent algorithms have overwhelming success probability $1-\mathrm{negl}(k)$.

Input: $\mathbf{s}'$, τ, error levels $\alpha, \beta \in (0,1]$
Output: Accept or Reject
$m := \left(\frac{\sqrt{\frac{3}{2}\ln(\frac{1}{\alpha})} + \sqrt{\ln(\frac{1}{\beta})}}{\frac{1}{2}-\tau} \right)^2$;
$(A, \mathbf{b}) \leftarrow \mathrm{LPN}^m_{k,\tau}$;
$c := \tau m + \sqrt{3(\frac{1}{2}-\tau)\ln(\frac{1}{\alpha})m}$;
if $\mathrm{wt}(A\mathbf{s}' + \mathbf{b}) \leq c$ **then**
  return Accept;
end
else
  return Reject;
end

Algorithm 3. `Test`

Notice that by our definition of m in `Test` even an exponentially small choice of $\alpha = \beta = \frac{1}{2^k}$ leads to only $m = \Theta\left(\frac{k}{(\frac{1}{2}-\tau)^2}\right)$ samples, which is linear in k and quadratic in $\left(\frac{1}{2}-\tau\right)^{-1}$. Thus, our hypothesis test can be carried out efficiently even for exponentially small error probabilities.

Lemma 3 (Hypothesis Testing). *For any $\alpha, \beta \in (0,1]$, `Test` accepts the correct LPN secret $\mathbf{s}$ with probability at least $1-\alpha$, and rejects incorrect $\mathbf{s}'$ with probability at least $1-\beta$, using m samples in time and space $\Theta(mk)$.*

Proof. Inputting the correct $\mathbf{s}$ to `Test` implies, that $\mathrm{wt}(A\mathbf{s}' + \mathbf{b}) \sim \mathrm{Bin}_{m,\tau}$. In this case we have

$$\begin{aligned} \Pr\left[\mathrm{wt}(A\mathbf{s}' + \mathbf{b}) \geq c\right] &\overset{\text{Chernoff}}{\leq} \exp\left(-\tfrac{1}{3} \cdot \min\left(\tfrac{c}{\tau m} - 1, \left(\tfrac{c}{\tau m} - 1\right)^2\right) \cdot \tau m\right) \\ &\leq \exp\left(-\tfrac{1}{3} \cdot \tfrac{\tau}{\frac{1}{2}-\tau} \left(\tfrac{c}{\tau m} - 1\right)^2 \cdot \tau m\right) \overset{!}{=} \alpha. \end{aligned}$$

We need that the last term is equal to α, which leads to the threshold weight of

$$c := \tau m + \sqrt{3(\frac{1}{2}-\tau)\ln\left(\frac{1}{\alpha}\right)m},$$

as defined in `Test`. If $\mathbf{s}' \neq \mathbf{s}$, then $\mathrm{wt}(A\mathbf{s}' + \mathbf{b}) \sim \mathrm{Bin}_{m,\frac{1}{2}}$. We want to upper bound the acceptance probability in this case.

$$\Pr\left[\mathrm{wt}(A\mathbf{s}' + \mathbf{b}) \leq c\right] \overset{\text{Chernoff}}{\leq} \exp\left(-\frac{1}{2} \cdot \left(1 - \frac{2c}{m}\right)^2 \cdot \frac{m}{2}\right) \overset{!}{=} \beta$$

Using the c from above, the last equation holds, if

$$m := \left(\frac{\sqrt{\frac{3}{2} \ln\left(\frac{1}{\alpha}\right)} + \sqrt{\ln\left(\frac{1}{\beta}\right)}}{\frac{1}{2} - \tau} \right)^2 . \qquad \square$$

Remark 2. As defined, `Test` takes m fresh samples on every invocation for achieving independence. However, for efficiency reasons we will in practice use the same m samples for `Test` on every invocation. Our experiments confirm that the introduced dependencies do not noticeably affect the algorithms' performance and success probability.

Now that we are equipped with an efficient hypothesis test, we can carry out the analysis of `Gauss`. For ease of notation, we use for the running time soft-Theta notion $\tilde{\Theta}$ to suppress factors that are polynomial in k.

Theorem 2. *`Gauss` solves $\mathrm{LPN}_{k,\tau}$ with overwhelming success probability in time and sample complexity $\tilde{\Theta}\left(\frac{1}{(1-\tau)^k}\right)$ using $\Theta(k^2)$ memory.*

Proof. We already noted that the outer repeat loop of `Gauss` takes an expected number of $\frac{1}{(1-\tau)^k}$ to produce a batch of k error-free LPN samples. In particular, Lemma 1 tells us that we will find an error-free batch after at most $\frac{\log^2 k}{(1-\tau)^k}$ trials with overwhelming probability.

The inner loop is executed an expected number of $\mathcal{O}(1)$ times until $A \in \mathrm{GL}_k(\mathbb{F}_2)$. Here again, after at most $\mathcal{O}\left(\log^2 k\right)$ iterations it is ensured that we get an invertible A with overwhelming probability. This already proves the upper bound on the running time.

Since, we only have to store k samples for A of length $\Theta(k)$ each, our memory consumption is $\Theta(k^2)$. In `Test` we do not necessarily have to store our $m = \Theta(k)$ samples, since we can process them on the fly. However, in practice (see Remark 2) it is useful to reserve for them another $\Theta(k^2)$ memory cells.

Considering the success probability, `Gauss` solves $\mathrm{LPN}_{k,\tau}$ when it rejects all false candidates $\mathbf{s}'$, and accepts the secret key $\mathbf{s}$ (if it appears). The first event happens by Lemma 3 with probability at least $1 - \beta = (\frac{1-\tau}{2})^k$ for each incorrect candidate by our choice in `Gauss`. The second event happens by Lemma 3 with probability at least $1 - \alpha = 1 - 2^{-k}$.

Let X be a random variable for the number of iterations of the outer loop until we are for the first time in the error-free case. Then

$$\begin{aligned} \Pr[\text{Success}] &= \sum_{i=1}^{\infty} \Pr[\text{Success} \mid X = i] \cdot \Pr[X = i] \\ &\geq \sum_{i=1}^{\infty} (1-\beta)^{i-1} (1-\alpha) \cdot \left(1 - (1-\tau)^k\right)^{i-1} (1-\tau)^k \end{aligned}$$

$$= \frac{(1-\alpha)\,(1-\tau)^k}{1-(1-\beta)\,(1-(1-\tau)^k)}$$
$$\geq \frac{(1-\alpha)(1-\tau)^k}{\beta+(1-\tau)^k} = 1-\mathrm{negl}(k). \qquad \square$$

Notice that `Gauss'` sample complexity is as large as its running time by Theorem 2. We will show in the following section that the sample complexity can be decreased to poly(k) without affecting the run time. This will be the starting point for further improvements.

4 LPN and Its Relation to Decoding

Let us slightly modify the `Gauss` algorithm from Sect. 3.2. Instead of taking in each iteration a fresh batch of k LPN samples, we initially fix a large enough pool of n samples. Then in each iteration we take k out of our pool of n samples, with linearly independent $\mathbf{a}_i$. This results in the following Algorithm 4 that we call `Pooled Gauss`.

Input: $\mathrm{LPN}_{k,\tau}$ oracle, τ
Output: secret $\mathbf{s}$
$n := k^2 \log^2 k$;
$(A, \mathbf{b}) \leftarrow \mathrm{LPN}^n_{k,\tau}$;
repeat
  repeat
    $I \leftarrow \mathcal{U}(\binom{[n]}{k})$;
  until $A_I \in \mathrm{GL}_k(\mathbb{F}_2)$;
  $s' := A_I^{-1}\mathbf{b}_I$;
until $\mathtt{Test}(\mathbf{s}', \tau, \frac{1}{2^k}, (\frac{1-\tau}{2})^k\ ) = \mathrm{Accept}$;
return $\mathbf{s}'$;

Algorithm 4. `Pooled Gauss`

Before we analyze `Pooled Gauss`, we want to clarify its connection to the decoding of random linear codes. Notice that we fix a sample matrix $A \in \mathbb{F}_2^{n\times k}$ with uniformly random entries. A can be considered a generator matrix of some random linear $[n, k]$ code $\mathcal{C}$, which is the column span of $\mathcal{C}$. The secret $\mathbf{s} \in \mathbb{F}_2^k$ is a message and the label vector $\mathbf{b} \in \mathbb{F}_2^n$ is an erroneous encoding of $\mathbf{s}$ with some error vector $\mathbf{e} \in \mathbb{F}_2^n$ having components $e_i \sim \mathrm{Ber}_\tau$. Thus, decoding the codeword $\mathbf{b}$ to the original message $\mathbf{s}$ solves $\mathrm{LPN}_{k,\tau}$.

Decoding such a codeword $\mathbf{b}$ can be done by finding an error-free index set as in `Pooled Gauss`. In coding theory language, such an error-free index set is called an information set. Thus, our `Pooled Gauss` algorithm is in this language an information set decoding algorithm, namely it resembles the well-known algorithm of Prange [28] from 1962. One should notice however that as opposed to the decoding scenario, we can fix the length n of $\mathcal{C}$ ourselves.

Theorem 3. *`Pooled Gauss` solves $LPN_{k,\tau}$ with overwhelming success probability in time $\tilde{\Theta}\left(\frac{1}{(1-\tau)^k}\right)$ using $\tilde{\Theta}(k^2)$ samples and $\tilde{\Theta}(k^3)$ memory.*

Proof. `Pooled Gauss`' run time follows with the same reasoning as for `Gauss`' running time. The outer loop will with overwhelming probability be executed at most $\frac{\log^2 k}{(1-\tau)^k}$ times, and all other parts can be performed in time $\mathcal{O}(k^3)$. The sample complexity follows by our choice of n in `Pooled Gauss`. Storing n samples requires $\tilde{\Theta}(k^3)$ memory.

For the success probability, we would first like to notice that the probability for drawing k linearly independent vectors out of a pool even as small as $n' = 2k$ without replacement can easily be lower-bounded by $\frac{1}{4}$. We will see, that the pool in our algorithm will be even bigger than that in the following. Therefore, by our choice of n and similar to the reasoning in the proof of Theorem 2, the inner loop of `Pooled Gauss` will always find an invertible A_I with overwhelming probability. So we condition our further analysis on this event.

Let Y be the number of error-free samples in the pool of n vectors. On expectation, we have $\mathbb{E}[Y] = (1-\tau)n$. By using a Chernoff bound, we can show that we deviate by a factor of $1 - \frac{1}{k}$ from the expectation with probability at most

$$\Pr[Y \geq (1 - \frac{1}{k})(1-\tau)n] \geq 1 - e^{-\frac{(1-\tau)n}{2k^2}}.$$

By our choice of $n = \omega\left(k^2 \log k\right)$ the right hand side is $1 - k^{-\omega(1)}$, which is overwhelming.

We call any pool with at least $(1 - \frac{1}{k})(1-\tau)n$ error-free samples *good*. Conditioned on the event G that our pool is good, we draw a batch of k error-free samples with probability

$$\begin{aligned} p &\geq \prod_{i=0}^{k-1} \frac{(1-\frac{1}{k})(1-\tau)n - i}{n} \geq \left(\frac{(1-\frac{1}{k})(1-\tau)n - k}{n}\right)^k \\ &= (1-\frac{1}{k})^k (1-\tau)^k \left(1 - \frac{k}{n(1-\frac{1}{k})(1-\tau)}\right)^k = \Omega\left((1-\tau)^k\right). \end{aligned}$$

Now, following the same arguments with p instead of $(1-\tau)^k$ as in Theorem 2 gives us an overwhelming probability of success. □

4.1 Low-Noise LPN

Some interesting cryptographic applications require that the LPN error $e_i \sim \mathrm{Ber}_\tau$ has an error term $\tau = \tau(k)$ depending on k. E.g. public key encryption seems to require some $\tau(k)$ as small as $\frac{1}{\sqrt{k}}$.

As a corollary from Theorem 3, we obtain that for any $\tau(k)$ that approaches 0 for $k \to \infty$, our `Pooled Gauss` algorithm runs – up to polynomial factors – in time $e^{\tau(k)k(1+o(1))}$. This implies that for $\tau(k) = o(\frac{1}{\log k})$ the run time of `Pooled Gauss` asymptotically outperforms the run time of `BKW` from Theorem 1.

Corollary 1 (Low Noise). *Let $\tau(k) \overset{k\to\infty}{\longrightarrow} 0$. `Pooled Gauss` solves $LPN_{k,\tau(k)}$ with overwhelming success probability in time $\tilde{\Theta}\left(e^{\tau k(1+o(1))}\right)$ using $\tilde{\Theta}(k^2)$ samples and $\tilde{\Theta}(k^3)$ memory.*

Proof. The run time statement follows by observing that

$$\left(\frac{1}{1-\tau}\right)^k = \left(\frac{1}{(1-\tau)^{\frac{1}{\tau}}}\right)^{\tau k} = \left(\frac{1}{\frac{1}{e}-o(1)}\right)^{\tau k} = (e+o(1))^{\tau k} = e^{\tau k(1+o(1))}. \quad \square$$

For small noise $\tau(k) = \Omega(\frac{1}{\sqrt{k}})$, i.e. a case that covers the mentioned encryption application above, we can also remove the error term $(1 + o(1))$ in the exponent, meaning that `Pooled Gauss` achieves – up to polynomial factors – run time $e^{\tau(k)k}$.

Corollary 2 (Really Low Noise). *Let $\tau(k) = \frac{1}{k^c}$ for $c \geq \frac{1}{2}$. `Pooled Gauss` solves $LPN_{k,\tau}$ with overwhelming success probability in time $\tilde{\Theta}\left(e^{k^{1-c}}\right)$, using $\tilde{\Theta}(k^2)$ samples and $\tilde{\Theta}(k^3)$ memory.*

Proof. Since $\ln(\frac{1}{1-x}) = x + \frac{x^2}{2} + \mathcal{O}\left(x^3\right)$ for $x \in [-1, 1)$ we get

$$\left(\frac{1}{1-\frac{1}{k^c}}\right)^k = e^{\ln\left(\frac{1}{1-\frac{1}{k^c}}\right)k} = e^{k^{1-c}+\frac{k^{1-2c}}{2}+\mathcal{O}\left(k^{1-3c}\right)}.$$

We see, that for $c \geq \frac{1}{2}$, the last term is in $\mathcal{O}\left(e^{k^{1-c}}\right)$ and for $c < \frac{1}{2}$ it is not. $\square$

4.2 Quantum Pooled Gauss

In a nutshell, `Pooled Gauss` runs until it finds an error-free batch of k LPN samples from a pool of n samples. The expected number of error-free samples in such a pool is $(1 - \tau)n$. Hence, we search for an index set I in a total search space of size $\binom{n}{k}$, in which we expect $\binom{(1-\tau)n}{k}$ good index sets. Therefore, we expect

$$T = \frac{\binom{n}{k}}{\binom{(1-\tau)n}{k}}$$

iterations of `Pooled Gauss` until we hit an error-free batch. It is not hard to show that T equals up to a polynomial the run time from Theorem 3.

The event of hitting an error-free batch can be modeled by the function $f\colon \binom{[n]}{k} \to \{0, 1\}$ that takes value $f(I) = 1$ iff I is an index set of k error-free LPN samples. More formally, we can define

$$f\colon \binom{[n]}{k} \to \{0,1\},\ I \mapsto \begin{cases} 1_{A_I^{-1}\mathbf{b}_I=\mathbf{s}}, & A_I \in \mathrm{GL}(\mathbb{F}_2^k) \\ 0, & A_I \notin \mathrm{GL}(\mathbb{F}_2^k) \end{cases}. \tag{2}$$

Here, the characteristic function $1_{A_I^{-1}\mathbf{b}_I=\mathbf{s}}$ takes value 1 iff we compute the correct secret key $\mathbf{s}$, which is equivalent to I being an index set of k error-free LPN

samples. In our algorithm `Pooled Gauss` the evaluation of $1_{A_I^{-1}\mathbf{b}_I=\mathbf{s}}$ is done by `Test`, which may err with negligible probability. But assume for a moment that we have a perfect instantiation of f.

Using f, the task of `Pooled Gauss` is to find an index set I^* among all index sets from $\binom{[n]}{k}$ such that $f(I^*) = 1$, which can be done *classically* in expected time

$$T = \frac{\binom{n}{k}}{|f^{-1}(1)|}.$$

We can now speed up `Pooled Gauss` *quantumly* by applying Boyer et al.'s [10] version of Grover search [17], which results in run time $\sqrt{T}$. It is worth to point out that Boyer et al.'s algorithm works even in our case, where we do not know the number $|f^{-1}(1)|$ of error-free index sets. All that the algorithm requires is oracle access to the function f, for which we show that this oracle access can be perfectly simulated by `Test`. This results in Algorithm 5 that we call `Quantum Pooled Gauss`.

Input: $\mathrm{LPN}_{k,\tau}$ oracle, τ
Output: secret $\mathbf{s}$
$n := k^2 \log^2 k$;
$(A, \mathbf{b}) \leftarrow \mathrm{LPN}_{k,\tau}^n$;
Define
$$\tilde{f}\colon \binom{[n]}{k} \to \{0,1\},\ I \mapsto \begin{cases} \texttt{Test}(A_I^{-1}\mathbf{b}_I, \tau, \binom{n}{k}^{-2}, \binom{n}{k}^{-2}), & A_I \in \mathrm{GL}(\mathbb{F}_2^k) \\ 0, & A_I \notin \mathrm{GL}(\mathbb{F}_2^k) \end{cases};$$
$I^* \leftarrow \texttt{Grover}(\tilde{f})$;
return $\mathbf{s} = A_{I^*}^{-1}\mathbf{b}_{I^*}$;

Algorithm 5. `Quantum Pooled Gauss`

Theorem 4. *`Quantum Pooled Gauss` quantumly solves $LPN_{k,\tau}$ with overwhelming probability in time* $\tilde{\Theta}\left(\left(\frac{1}{1-\tau}\right)^{\frac{k}{2}}\right)$, *using* $\tilde{\Theta}(k^2)$ *queries and* $\tilde{\Theta}(k^3)$ *memory.*

Proof. According to [10], `Grover` succeeds with overwhelming success probability. Hence, the proof of Theorem 3 essentially carries over to the quantum setting.

However, it remains to show that we can safely replace oracle access to the function f as defined in Eq. (2) by `Test`, which in turn defines some function

$$\tilde{f}\colon \binom{[n]}{k} \to \{0,1\},\ I \mapsto \begin{cases} \texttt{Test}(A_I^{-1}\mathbf{b}_I, \tau, \binom{n}{k}^{-2}, \binom{n}{k}^{-2}), & A_I \in \mathrm{GL}(\mathbb{F}_2^k) \\ 0, & A_I \notin \mathrm{GL}(\mathbb{F}_2^k) \end{cases}.$$

We will show that by our choice of $\alpha = \beta = \binom{n}{k}^{-2}$ with overwhelming probability $f(I) = \tilde{f(I)}$ for *all* I, i.e. we perfectly simulate f. Let us define a random variable X that counts the number of inputs in which f and $\tilde{f}$ disagree, i.e.

$$X := \left| \{ I \in \binom{[n]}{k} \mid \tilde{f}(I) \neq f(I) \} \right|.$$

Notice that $X \sim \mathrm{Bin}_{\binom{n}{k},\leq\alpha}$, since by Lemma 3 `Test` errs with probability (at most) $\alpha = \beta$ for all of the $\binom{n}{k}$ sets I. We obtain

$$\Pr[\tilde{f} = f] = \Pr[X = 0] \geq \left(1 - \frac{1}{\binom{n}{k}^2}\right)^{\binom{n}{k}} \geq 1 - \frac{1}{\binom{n}{k}},$$

where we use Bernoulli's inequality for the last step. Since we chose $n = \omega(k^2)$, we have $\binom{n}{k} \geq \left(\frac{n}{k}\right)^k = \omega(k^k)$. This implies $\Pr[\tilde{f} = f] = 1 - \mathrm{negl}(k)$, as required. □

Remark 3. Notice that our slight modification from `Gauss` to `Pooled Gauss` enables the use of quantum techniques. While both algorithms `Gauss` and `Pooled Gauss` achieve the same running time T, `Gauss` also requires (roughly) T samples. But any algorithm with sample complexity T has automatically run time lower bound $\Omega(T)$, since our LPN oracle is by Definition 1 classical and each oracle access costs $\Omega(1)$.

So while there is good motivation to reduce the number of samples, it is somewhat unsatisfactory from a cryptanalysts' point of view to make only limited use of an LPN oracle by restricting to a polynomial number of samples. In the next section, we will show how more extensive queries give rise to a better suited pool of vectors that will further speed up `Pooled Gauss`.

5 Decoding LPN with Preprocessing

Our idea is to add some preprocessing to `Pooled Gauss` that produces LPN samples with $\mathbf{a}_i$ of *smaller dimension* k' by zeroing some columns in the A-matrix. This may come at the cost of slightly increasing the noise parameter τ. This idea gives rise to the following meta algorithm `Dim-Decode`.

Input: $\mathrm{LPN}_{k,\tau}$ oracle, τ
Output: secret $\mathbf{s}$
(1) *Modify*: Use a large number of samples to produce a small number of dimension-reduced samples, resulting in a new $\mathrm{LPN}_{k',\tau'}$ instance with $k' < k$ and $\tau' \geq \tau$;
(2) *Decode*: Use a decoding algorithm to solve $\mathrm{LPN}_{k',\tau'}$, e.g. use `Pooled Gauss`;
(3) *Complete*: Recover the remaining coordinates of $\mathbf{s}$, e.g. via enumeration or by iterating (1) and (2) accordingly;

Algorithm 6. `Dim-Decode`

In the following, we give different instantiations of `Dim-Decode`. We start by looking at techniques for the *Modify* step for dimension reduction.

5.1 Improvements Using only Polynomial Memory

Our first simple technique is to keep only those LPN samples $(\mathbf{a}, b)$ that have zeros in the last $k - k'$ coordinates of $\mathbf{a}$. We will balance the running time for steps *Modify* and *Decode* by choosing k' accordingly. This results in Algorithm 7 that we call `Well-Pooled Gauss`.

Notice that by our choice of k', `Well-Pooled Gauss` reduces the dimension to a $\frac{1}{1+\log(\frac{1}{1-\tau})}$-fraction of k. Since $\tau \in [0, \frac{1}{2})$ we have

$$\frac{1}{1+\log(\frac{1}{1-\tau})} \in (\frac{1}{2}, 1],$$

meaning that $k' \geq \frac{k}{2}$ or in other words that `Pooled Gauss` in its first run recovers at least the first half of the bits of $\mathbf{s}$, and in its second run the remaining half.

Input: $\mathrm{LPN}_{k,\tau}$ oracle, τ
Output: secret $\mathbf{s}$
$k' := \frac{1}{1+\log(\frac{1}{1-\tau})}k$;
Set parameters n, m as in `Pooled Gauss` for an $\mathrm{LPN}_{k',\tau}$ instance;
(1) *Modify*
repeat
  $(\mathbf{a}, b) \leftarrow \mathrm{LPN}_{k,\tau}$;
  if $\mathbf{a}_{\{k'+1,\ldots,k\}} = 0^{k-k'}$ **then**
    Add $(\mathbf{a}_{\{1,\ldots,k'\}}, b)$ to sample pool;
  end
until pool contains more than $n + m$ elements;
(2) *Decode*
$(s_1, \ldots, s_{k'}) \leftarrow$Run `Pooled Gauss` on the pool containing the first n $\mathrm{LPN}_{k',\tau}$ samples, while taking the remaining m samples for `Test`;
(3) *Complete*
$(A, \mathbf{b}) \leftarrow \mathrm{LPN}_{k,\tau}^{n+m}$. Reduce A's dimension to $k - k'$ using $(s_1, \ldots, s_{k'})$;
$(s_{k'+1}, \ldots, s_k) \leftarrow$ Run `Pooled Gauss` on the pool containing the first n $\mathrm{LPN}_{k-k',\tau}$ samples, while taking the remaining m samples for `Test`;
return $\mathbf{s}$;

Algorithm 7. Well-Pooled Gauss

Hence the run time of `Pooled Gauss`' first application dominates the run time of its second application. Since `Pooled Gauss`' run time depends exponentially on k, we can gain up to a square root in the running time when we reduce the dimension up to $\frac{k}{2}$.

Theorem 5 (Well-Pooled Gauss). *`Well-Pooled Gauss` solves $LPN_{k,\tau}$ with overwhelming probability in time and query complexity*

$$\tilde{\Theta}\left(\left(\frac{1}{(1-\tau)^k}\right)^{\frac{1}{1+\log\left(\frac{1}{1-\tau}\right)}}\right)$$

using $\tilde{\Theta}(k^3)$ memory.

Proof. `Pooled Gauss`'s first application runs in time $T := \tilde{\Theta}\left(\frac{1}{(1-\tau)^{k'}}\right)$, which is the claimed total running time. Furthermore, by Lemma 1 with overwhelming probability the run time of the *Modify* step for finding $n+m$ samples with last $k-k'$ 0-coordinates is bounded by

$$2^{k-k'}\left(n+m+\log^2 k\right) = \tilde{\Theta}\left(2^{k-k'}\right) = \tilde{\Theta}\left(2^{\left(1-\frac{1}{1+\log\left(\frac{1}{1-\tau}\right)}\right)k}\right)$$
$$= \tilde{\Theta}\left(2^{\frac{\log\left(\frac{1}{1-\tau}\right)}{1+\log\left(\frac{1}{1-\tau}\right)}k}\right) = T.$$

□

5.2 Quantum Improvements with Polynomial Memory

In the quantum version of `Well-Pooled Gauss`, called `Quantum Well-Pooled Gauss`, we simply replace in Algorithm 7 the `Pooled Gauss` procedure by its quantum version. Notice that by Remark 3 we cannot provide a quantum version of our *Modify* step. So we cannot expect to gain another full square root by going to the quantum version of `Well-Pooled Gauss`.

The following theorem shows that in `Quantum Well-Pooled Gauss` one should take the parameter choice $k' := \frac{2}{2+\log\left(\frac{1}{1-\tau}\right)}k > \frac{2}{3}k$. The `Quantum Pooled Gauss` routine then runs in time

$$T := \tilde{\Theta}\left(\left(\frac{1}{1-\tau}\right)^{\frac{k'}{2}}\right).$$

Hence in comparison with `Well-Pooled Gauss` (Theorem 5) we gain at most an additional factor of $\frac{2}{3}$ in the exponent.

Theorem 6 (Quantum Well-Pooled Gauss). *`Quantum Well-Pooled Gauss` quantumly solves $LPN_{k,\tau}$ with overwhelming probability in time and query complexity*

$$\tilde{\Theta}\left(\left(\frac{1}{(1-\tau)^k}\right)^{\frac{1}{2+\log\left(\frac{1}{1-\tau}\right)}}\right),$$

using $\tilde{\Theta}(k^3)$ space.

Proof. Analogous to the proof of `Well-Pooled Gauss`' run time (Theorem 5), in its quantum version the run time of the first `Quantum Pooled Gauss` routine is $T = \tilde{\Theta}((\frac{1}{1-\tau})^{\frac{k'}{2}})$, which is the claimed total run time. T dominates the run time of the second call for `Quantum Pooled Gauss`, since $k' > \frac{2}{3}k$. T also upper bounds the run time of the *Modify* step, since by Lemma 1 with overwhelming probability for obtaining $n+m$ of the desired form it takes at most time

$$2^{k-k'}\left(n+m+\log^2 k\right) = \tilde{\Theta}(2^{k-k'}) = \tilde{\Theta}\left(2^{\frac{\log\left(\frac{1}{1-\tau}\right)}{2+\log\left(\frac{1}{1-\tau}\right)}k}\right) = T. \qquad \square$$

5.3 Using Memory – Building a Bridge Towards BKW

Up to now, for instantiating Algorithm 6 in `Well Pooled Gauss` we made only somewhat naive use of our LPN oracle by storing only those vectors in the stream of all oracle answers that were already dimension-reduced. The reason for this was our restriction to polynomial memory consumption in order to achieve highly efficient algorithms in practice.

Optimally, we could tune our LPN algorithm by the memory that is available on our machine. Let us say, we have memory M and we are looking for the fastest algorithm that uses at most M memory cells. In this scenario, we are free to store more LPN samples and to add them until they provide us dimension-reduced samples, at the cost of a growing error $\tau' > \tau$ determined by the Piling-up Lemma (Lemma 2).

Notice that this is exactly the strategy of `BKW`. But as opposed to the `BKW` algorithm, which basically reduces the dimension k all the way down to 1, we allow – due to our constraint memory M – only limited dimension reduction down to some k'. Since afterwards, we will in Algorithm 6 resort again to a *Decoding* step, the optimal choice of k' is not determined by the size of M alone, but also by the growth of the error τ' that our decoding procedure can handle.

More precisely, the following Algorithm 8, called `Hybrid`, in a first step uses the naive strategy of `Well-Pooled Gauss` to decrease the dimension by k_1, while leaving τ unchanged. Then in a second step it uses c `BKW`-iterations on blocks of size d to further reduce the dimension by k_2, thereby increasing to error τ' which grows double-exponentially in c.

`Hybrid` instantiated with only polynomial memory leaves out all `BKW`-iterations and boils down to `Well-Pooled Gauss`. `Hybrid` instantiated with sufficiently memory achieves `BKW`'s time complexity $2^{k/\log(\frac{k}{\tau})(1+o(1))}$. Thus, `Hybrid` provides a perfect interpolation between both algorithms.

Theorem 7 (Hybrid). *Using $\tilde{\mathcal{O}}(M)$ space, `Hybrid` solves $\mathrm{LPN}_{k,\tau}$ with overwhelming probability in time and query complexity $\tilde{\mathcal{O}}(M \cdot 2^{k_1})$, where k_1 is as defined in `Hybrid`.*

For $M = \mathrm{poly}(k)$ we get the same time, memory and sample complexity as in `Well-Pooled Gauss` (Theorem 5).

Choosing $M = 2^{k/\log(\frac{k}{\tau})(1+o(1))}$ gives us the complexities of `BKW` (Theorem 1).

Proof. For the *Decoding* step, we need $m = \tilde{\Theta}\left(\frac{1}{(\frac{1}{2}-\tau')^2}\right) = \tilde{\Theta}\left(\frac{1}{(1-2\tau)^{2^{c+1}}}\right)$ samples for the hypothesis test and $n = \mathrm{poly}(k)$ samples for the pool at the end. That's why we initially need to feed the `BKW` step $N = m + n + c2^d = \tilde{\Theta}\left(2^{\max\left(\frac{k_2}{c}, 2^c \log\left(\frac{1}{(1-2\tau)^2}\right)\right)}\right)$ samples. As in `Well-Pooled Gauss` we can create

Input: $\mathrm{LPN}_{k,\tau}$ oracle, τ, memory M
Output: secret $\mathbf{s}$
Choose $\varepsilon > 0$;
$c := \max\left(0, \log\left(\frac{1}{\tau^{1-\varepsilon}k^{\varepsilon}}\log(M)\log(\frac{k}{\tau})\right)\right)$;
Choose $k_2 \leq c\frac{k-1}{\log\left(\frac{k}{\tau}\right)}$ s.t. $\max\left(\frac{k_2}{c}, 2^c\log(\frac{1}{(1-2\tau)^2})\right) \leq \log(M)$;
$d := \frac{k_2}{c}$;
$\tau' = \frac{1}{2} - \frac{1}{2}(1-2\tau)^{2^c}$;
$k_1 := \frac{\log\left(\frac{1}{1-\tau'}\right)(k-k_2)+2^c\log\left(\frac{1}{(1-2\tau)^2}\right)-\log M}{1+\log\left(\frac{1}{1-\tau'}\right)}$;
$k' := k - k_1 - k_2$;
Set parameters n, m as in `Pooled Gauss` for an $\mathrm{LPN}_{k',\tau'}$ instance;
(1) *Modify*
repeat
  $(\mathbf{a}, \ell) \leftarrow \mathrm{LPN}_{k,\tau}$;
  if $\mathbf{a}_{\{k-k_1+1,\ldots,k\}} = 0^{k_1}$ **then**
    Add $(\mathbf{a}_{1,\ldots,k-k_1}, b)$ to sample pool;
  end
until pool contains more than $n + m + c2^d$ samples;
for $i = 1, \ldots, c$ **do**
  foreach $j \in \mathbb{F}_2^d$ **do**
    Pick a row $\mathbf{a}_k$ of A with suffix $j|0^{(i-1)d}$ (if any); add $\mathbf{a}_k$ to all the other rows of A with suffix $j|0^{(i-1)d}$, also add corresponding labels;
    Remove the k^{th} row from A and $\mathbf{b}$;
  end
end
(2) *Decode*
$(s_1, \ldots, s_{k'}) \leftarrow$Run `Pooled Gauss` on the pool containing the first n $\mathrm{LPN}_{k',\tau'}$ samples, while taking the remaining m samples for `Test`;
(3) *Complete*
While there are still unknown bits of $\mathbf{s}$ go to (1), using the known bits of $\mathbf{s}$ to create smaller dimension samples;
return $\mathbf{s}$;

Algorithm 8. `Hybrid`

these samples in time $\tilde{\Theta}(N \cdot 2^{k_1})$. Therefore with overwhelming probability the *Modify* and *Decoding* steps in `Hybrid` take time $\tilde{\Theta}\left(N \cdot 2^{k_1} + N + \left(\frac{1}{1-\tau'}\right)^{k'} m\right)$

$$= \tilde{\Theta}\left(2^{k_1+\max\left(\frac{k_2}{c}, 2^c\log(\frac{1}{(1-2\tau)^2})\right)} + 2^{\log\left(\frac{1}{1-\tau'}\right)(k-k_1-k_2)+2^c\log\left(\frac{1}{(1-2\tau)^2}\right)}\right),$$

using $\tilde{\Theta}\left(N \cdot 2^{k_1}\right)$ samples and $\tilde{\Theta}(N)$ memory. Our choice of k_1 in `Hybrid` balances both run time summands. Our choice of k_2 then gives us the stated time, sample and memory complexity. Notice that the *Complete* step takes less time, queries and memory, since here we solve smaller LPN instances.

It remains to show that Hybrid contains as special cases Well-Pooled Gauss and BKW. For some $\varepsilon > 0$, let us define similar to Hybrid

$$c(k, \tau, M) := \max\left(0, \log\left(\frac{1}{\tau^{1-\varepsilon} k^{\varepsilon}} \log(M) \log\left(\frac{k}{\tau}\right)\right)\right).$$

For the choice $M = \text{poly}(k)$ we get $c = o(1)$ and $k_2 = o(1)$, which means that we do not perform any BKW steps. Thus, Hybrid is identical to Well-Pooled Gauss. For the choice $M = 2^{(1+o(1))k/\log\left(\frac{k}{\tau}\right)}$ we obtain $c = (1-\varepsilon)\log(\frac{k}{\tau}) + o(1)$, $k_2 = k - o(1)$ and $k_1 = o(1)$, giving us the complexities of BKW from Theorem 1. □

5.4 Using Memory – Advanced Decoding Algorithms

In the previous Sect. 5.3 we provided a time-memory tradeoff for Algorithm 6 Dim-Decode by looking at the *Modify* step only. In this section, we will focus on the *Decode* step. So far, we only used the quite naive Gauss decoding procedure, which resembles Prange's information set decoding algorithm [28]. However, within the last years there has been significant progress in information set decoding, starting with Ball-Collision Decoding [6], then followed by a series of papers, MMT [26], BJMM [4] and May-Ozerov [27], using the so-called representation technique.

In principle, one can freely choose the preferred decoding procedure in *Decode*. Our starting point was a simplified version of BJMM, but then on optimizing the BJMM parameters we found out that for the LPN instances under consideration (see Sect. 6), MMT performs actually best. Since BJMM offers asymptotically a better run time than MMT, the situation should however change for (very) large LPN dimension k. We also did not consider the Nearest Neighbor algorithm May-Ozerov, since its large polynomial run time factor currently prevents speedups in the parameter ranges that we consider.

Instantiating Dim-Decode with MMT decoding results in Algorithm 9, called Well-Pooled MMT.

Input: $\text{LPN}_{k,\tau}$ oracle, τ
Output: secret $\mathbf{s}$
Set parameters n, m as in Pooled Gauss for a $\text{LPN}_{k',\tau}$ instance;
(1) *Modify*
Use the same procedure as in Well-Pooled Gauss (Algorithm 7).
(2) *Decode*
$(s_1, \ldots, s_{k'}) \leftarrow$Run MMT on the pool containing the first n $\text{LPN}_{k',\tau}$ samples, while taking the remaining m samples for Test;
(3) *Complete*
$(A, \mathbf{b}) \leftarrow \text{LPN}_{k,\tau}^{n+m}$. Reduce A's dimension to $k - k'$ using $(s_1, \ldots, s_{k'})$;
$(s_{k'+1}, \ldots, s_k) \leftarrow$ Run MMT on the pool containing the first n $\text{LPN}_{k-k',\tau}$ samples, while taking the remaining m samples for Test;
return $\mathbf{s}$;

Algorithm 9. Well-Pooled MMT

Unfortunately, [26] does not provide a closed run time formula for `MMT`. Therefore, we cannot give a theorem for `Well-Pooled MMT` that provides a precise bound for the time complexity as a function of k, τ as in the previous sections. However, we are able to optimize the run time of `Well-Pooled MMT` for every fixed τ as a function of k.

Let us conjecture that `Well-Pooled MMT`'s time complexity is $2^{c(\tau)k}$. Recall from Theorem 5 that `Well-Pooled Gauss`'s time complexity is

$$\tilde{\Theta}\left(2^{\frac{\log\left(\frac{1}{(1-\tau)}\right)}{1+\log\left(\frac{1}{1-\tau}\right)}k}\right) = \tilde{\Theta}\left(2^{c'(t)k}\right).$$

We can plot both functions $c(\tau)$ and $c'(\tau)$ as a function of τ. The following graph in Fig. 1 visualizes that the run time exponent $c(\tau)$ of `Well-Pooled MMT` is smaller than the run time exponent $c'(\tau)$ of `Well-Pooled Gauss` for every $\tau \in [0, \frac{1}{2})$, as one would expect. The largest gap appears at $\tau = \frac{1}{4}$, where `Well-Pooled MMT` achieves time $2^{0.282k}$, whereas `Well-Pooled Gauss` requires time $2^{0.293k}$.

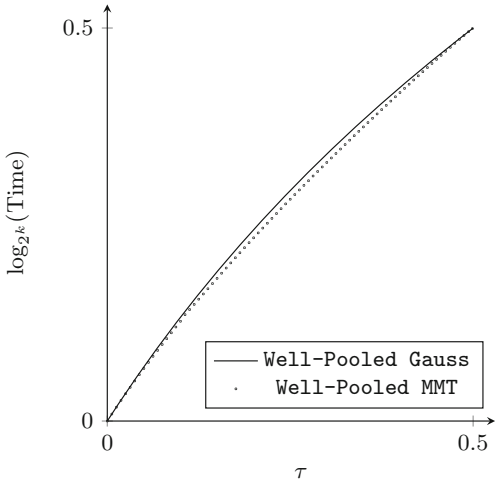


Fig. 1. `Well-Pooled Gauss` and `Well-Pooled MMT`

Remark 4. Since we do not know how to express the running time of `MMT` as a closed formula, a function that approximates $c(\tau)$ reasonably well is

$$\frac{\log\left(\frac{1}{1-\frac{7}{6}\tau}\right)}{\log\left(\frac{12}{5}\right) + \log\left(\frac{1}{1-\frac{7}{6}\tau}\right)}.$$

It is worth noticing that MMT, as opposed to Prange, consumes exponential memory. However, the memory consumption is still quite moderate. For the LPN instances that we considered the memory never exceeded $2^{\frac{c(\tau)k}{2}}$.

Quantumly we can also speed up our *Decode* step. Naively, i.e. just using Grover search, MMT collapses to Quantum Gauss. But using quantum random walks, one obtains a Quantum Well-Pooled MMT algorithm, as recently proposed by Kachigar and Tillich [22], that slightly outperforms Quantum Gauss.

6 Classical and Quantum Bit Security Estimates

In LPN cryptanalysis, it is currently practice to give estimated bit complexities, that is the binary logarithm of the running time, for newly developed algorithms in table form. In order to allow some comparison with existing work and to give an impression how our algorithms might perform for LPN parameters of cryptographic interest, we also give tables in the following. However, at the same time we want to express a clear warning that *all* LPN tables should be taken with care, since the given bit security levels might over- or underestimate the real performance in practice.

Therefore, we also provide experimental results in Sect. 7 on medium-size LPN parameter, which we extrapolate to cryptographic size by our asymptotic formulas. In our opinion, this is the only reliable way of predicting key sizes with good accuracy. Nevertheless, experimental results might not be possible even for medium-size parameters in some cases, e.g. when an algorithm consumes large memory or when we predict quantum running-times. Hence, performance tables are in these cases unavoidable.

Since we care about practical memory consumption in this work, we enforce upper limits on the available memory for the considered algorithms. We first consider 2^{60} bits, since that is more memory than the biggest RAM we are aware of. Second, we consider a bound of 2^{80} bits for an extra safety margin.

For these upper bounds on the memory M, we compute the bit complexities of our algorithms and compare them to the Coded-BKW [18] table from Bogos et al. [8].

The formulas we used for computing the bit complexities are available at https://github.com/Memphisd/LPN-decoded. In these formulas, we tried to take any polynomial factors into account. As in [8], the cost of the *Complete* step is not taken into account in the tables, but the cost of *Complete* would not significantly change the data.

One can see in Table 2 that for the chosen LPN parameters only 7 parameter sets can be solved with Coded-BKW within the memory limit of 2^{60}. Using instead 2^{80} bits of memory as in Table 3, Coded-BKW can be used or a larger range of instances, as one would expect.

Let us compare this to Tables 4 and 5, where we used either Hybrid, marked as bold face entries, or Well-Pooled MMT (WP MMT). Interestingly, most instances are optimally solved with Well-Pooled MMT using always less than 2^{30} bits of memory. For small memory, Hybrid collapses to Well-Pooled Gauss.

Table 2. `Coded-BKW` [8], $M = 2^{60}$

τ	k							
	256	384	448	512	576	640	768	1280
$\frac{1}{\sqrt{k}}$	44	55	59	-	-	-	-	-
0.05	42	54	59	-	-	-	-	-
0.125	52	-	-	-	-	-	-	-
0.25	-	-	-	-	-	-	-	-
0.4	-	-	-	-	-	-	-	-

Table 3. `Coded-BKW` [8], $M = 2^{80}$

τ	k							
	256	384	448	512	576	640	768	1280
$\frac{1}{\sqrt{k}}$	44	55	59	64	70	73	-	-
0.05	42	54	59	65	72	78	-	-
0.125	52	67	74	-	-	-	-	-
0.25	70	-	-	-	-	-	-	-
0.4	-	-	-	-	-	-	-	-

Table 4. `WP MMT` or `Hybrid`, $M = 2^{60}$

τ	k							
	256	384	448	512	576	640	768	1280
$\frac{1}{\sqrt{k}}$	43	49	51	52	54	56	59	70
0.05	39	48	52	56	60	64	72	108
0.125	**58**	81	91	102	113	123	144	230
0.25	**65**	**124**	153	172	192	211	250	406
0.4	**85**	**153**	**186**	**219**	**251**	**286**	**357**	584

Table 5. `WP MMT` or `Hybrid`, $M = 2^{80}$

τ	k							
	256	384	448	512	576	640	768	1280
$\frac{1}{\sqrt{k}}$	43	49	51	52	54	56	59	70
0.05	39	48	52	56	60	64	72	108
0.125	**58**	**77**	91	102	113	123	144	230
0.25	**65**	**88**	**119**	**151**	**184**	211	250	406
0.4	**76**	**122**	**154**	**186**	**219**	**251**	**318**	**576**

Table 6. `Quantum Hybrid`, $M = 2^{60}$

τ	k							
	256	384	448	512	576	640	768	1280
$\frac{1}{\sqrt{k}}$	33	37	39	40	42	43	46	54
0.05	30	37	40	42	45	48	53	73
0.125	56	57	63	69	75	81	93	140
0.25	**63**	89	101	112	123	135	158	248
0.4	**76**	**121**	**144**	163	181	198	234	373

Table 7. `Quantum Hybrid`, $M = 2^{80}$

τ	k							
	256	384	448	512	576	640	768	1280
$\frac{1}{\sqrt{k}}$	33	37	39	40	42	43	46	54
0.05	30	37	40	42	45	48	53	73
0.125	56	57	63	69	75	81	93	140
0.25	**63**	**83**	**96**	112	123	135	158	248
0.4	**73**	**100**	**122**	**144**	**165**	**187**	**232**	373

However, `Well-Pooled MMT` outperforms `Well-Pooled Gauss` for the given instances. Interestingly, most instances – especially those of cryptographic interest – are solved via `Well-Pooled MMT`, that is with pure decoding and *without* using the given memory.

Note that according to our predictions, 80-bit security on classical computers for $\mathrm{LPN}_{\frac{1}{\sqrt{k}},k}$ can only be achieved for $k \geq 2048$. This makes current applications of LPN for encryption quite inefficient.

`Coded-BKW` shows its strength for errors around $\frac{1}{8}$. However, the same techniques could be used in the *Modify* step of our `Hybrid` algorithm, which would result in a similar running time.

Tables 6 and 7 finally state the quantum bit security levels when taking `Quantum Well-Pooled Gauss` inside `Hybrid`. Here again, the bold marked entries are those where the optimization suggests to use `BKW` steps. In comparison to the classical case, these are even less instances. We see that the prominent cryptographic choice $\mathrm{LPN}_{512,\frac{1}{8}}$ offers only 69-bit security on quantum computers.

Table 8. $\tau = \frac{1}{\sqrt{k}}$

k	Classic	Quantum
6100	**128**	91
15000	**192**	127
26500	**256**	158
2200	86	**64**
4300	113	**80**
15400	199	**128**

Table 9. $\tau = \frac{1}{8}$

k	Classic	Quantum
670	**128**	84
1060	**192**	120
1410	**256**	152
460	93	**64**
630	121	**80**
1150	208	**128**

Table 10. $\tau = \frac{1}{4}$

k	Classic	Quantum
470	**128**	105
610	**192**	128
790	**256**	162
260	66	**64**
370	86	**80**
610	192	**128**

Choice of k for security levels $128, 192, 256$ (classic) and $64, 80, 128$ (quantum)

NIST's Post-Quantum Call. NIST [1] asks for classical security levels of $128, 192, 256$ bit and quantum security levels of $64, 80, 128$ bit. Tables 8, 9 and 10 define the minimal k that fulfill these levels for τ taking values $\frac{1}{\sqrt{k}}$, $\frac{1}{8}$, $\frac{1}{4}$, respectively. The memory is constrained to $M = 2^{80}$ bits.

7 Experiments

All our implementations are available via
https://github.com/Memphisd/LPN-decoded.

Our experiments were done on a server with four 16-core-processors, allowing a parallelization of 64 threads, using 256 GB $< 2^{41}$ bit RAM.

We executed `Well-Pooled Gauss` and `Well-Pooled MMT` for $\tau = \frac{1}{8}, \frac{1}{4}$ and various k. In order to get reliable run times for $\tau = \frac{1}{8}$ and $k < 170$, respectively $\tau = \frac{1}{4}$ and $k < 100$, we averaged the run time over 30 instances. For larger k, we solved only a single instance.

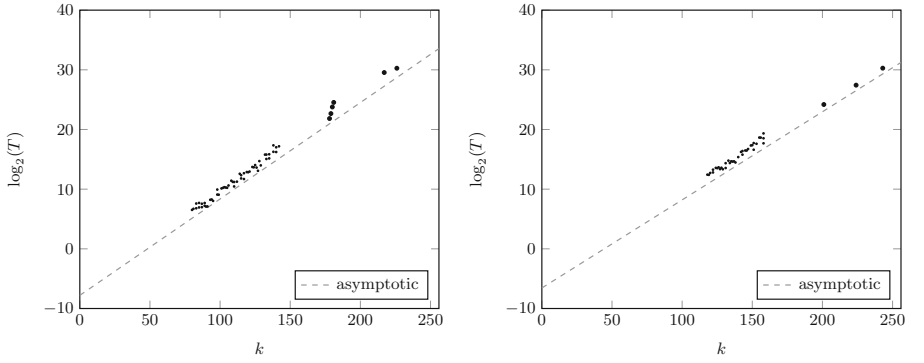
The results for $\tau = \frac{1}{8}$ and $\tau = \frac{1}{4}$ are shown in Figs. 2 and 3, respectively. Here we plot the logarithm of the running time in msec as a function of k. Hence negative values mean that it takes only a fraction of a msec to solve the instance. For `Well-Pooled Gauss` we plotted as a comparison the asymptotic line with slope

$$\frac{\log_2(\frac{1}{1-\tau})}{1+\log_2(\frac{1}{1-\tau})},$$

which follows from Theorem 5. For `Well-Pooled MMT` we numerically computed the slopes 0.177 and 0.381 for $\tau = \frac{1}{8}$ and $\tau = \frac{1}{4}$, respectively, similar to the computation in Fig. 1.

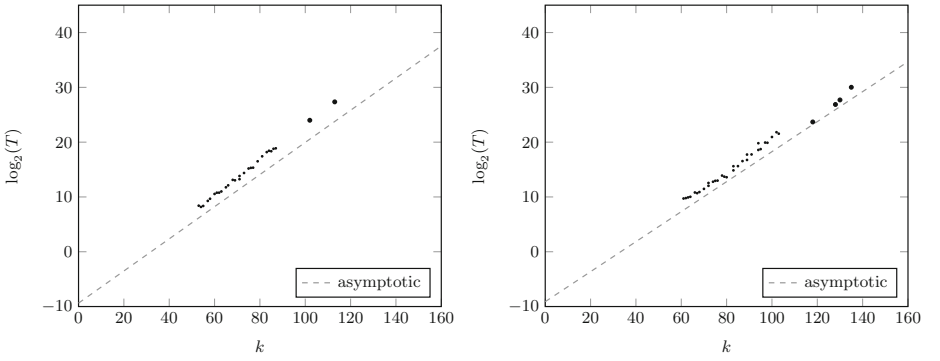
As can be seen in Figs. 2 and 3, as expected the experiments take slightly longer than the asymptotic prediction, since the asymptotic hides polynomial factors. But especially for `Well-Pooled MMT` large values of k are close to the asymptotic line, which means that the asymptotic quite accurately predicts the running time.

Since `Well-Pooled MMT`'s run time includes quite large polynomials factors, due to `MMT`, we expected that it outperforms `Well-Pooled Gauss` only for large



(a) `Well-Pooled Gauss` (b) `Well-Pooled MMT`

Fig. 2. Experimental results for $\tau = \frac{1}{8}$



(a) `Well-Pooled Gauss` (b) `Well-Pooled MMT`

Fig. 3. Experimental results for $\tau = \frac{1}{4}$

values of k. To our surprise, the break even point for both algorithms was only $k = 78$ for $\tau = \frac{1}{8}$, and $k = 9$ for $\tau = \frac{1}{4}$. Hence, for these error rates τ one should always prefer `Well-Pooled MMT` over `Well-Pooled Gauss` even for relatively small sizes of k.

Largest instances. The largest instances that we solved with `Well-Pooled MMT` were $k = 243, \tau = \frac{1}{8}$ and $k = 135, \tau = \frac{1}{4}$. Let us provide more details for these computations.

For $k = 243, \tau = \frac{1}{8}$, we first computed in almost 7 days a pool of samples $(\mathbf{a}_i, b_i)$ where the $\mathbf{a}_i$ had their last 35 coordinates equal to zero. The resulting $\mathrm{LPN}_{208,\frac{1}{8}}$ was solved with `MMT` in 8 days, resulting in a total of 15 days. This recovers already 208 coordinates of $\mathbf{s}$. The *Complete* step in `Well-Pooled MMT` that recovers the remaining 35 coordinates took less than a second.

For $k = 135, \tau = \frac{1}{4}$, the preprocessing step on again 35 coordinates took almost 6 days, the decoding step 8 days and *Complete* less than a second, resulting in a total of 14 days. We summarize the results in Table 11.

Table 11. Solved instances

Algorithm	k	τ	Pool gen.	BKW	Decoding	Total
Well-Pooled MMT	243	0.125	6.73 d	-	8.34 d	15.07 d
Well-Pooled MMT	135	0.25	5.65 d	-	8.19 d	13.84 d
Hybrid	135	0.25	2.21 d	1.72 h	3.41 d	5.69 d
Well-Pooled Gauss	113	0.25	0.77 d	-	1.21 d	1.98 d
Well-Pooled MMT	113	0.25	1.64 h	-	2.18 h	3.82 h
Hybrid	113	0.25	0.13 h	0.98 h	0.57 h	1.68 h

Extrapolation to $k = 512$. Let $T(k, \tau)$ be the time to solve an $\mathrm{LPN}_{k,\tau}$ instance via Well-Pooled MMT as computed numerically in Fig. 1. Then it would take us a factor of $\frac{T(512,\frac{1}{8})}{T(243,\frac{1}{8})} \approx 2^{41}$ longer to break an $\mathrm{LPN}_{512,\frac{1}{8}}$ than an $\mathrm{LPN}_{243,\frac{1}{8}}$ instance. For $\tau = \frac{1}{4}$ we would even need an additional factor of $\frac{T(512,\frac{1}{4})}{T(135,\frac{1}{4})} \approx 2^{113}$.

To make these kind of statements more trustworthy, we should check, if the numerically computed running times resemble the experimental data. Therefore consider the following: It took about $2^{6.4}$ times as long to solve $\mathrm{LPN}_{135,\frac{1}{4}}$ than solving $\mathrm{LPN}_{113,\frac{1}{4}}$ in the experiments. Using formulas, there is a gap of 2^6 between these instances, which is close to what we actually measured.

Hence, both instances seem to provide sufficient classical security, but also recall from Sect. 6 that $\mathrm{LPN}_{512,\frac{1}{8}}$ only offers 69-bit quantum security.

Hybrid implementation. We solved $\mathrm{LPN}_{113,\frac{1}{4}}$ in less than 2 h, which in comparison took for Well-Pooled Gauss around 2 days and for Well-Pooled MMT still almost 4 h. We were also able to solve $\mathrm{LPN}_{135,\frac{1}{4}}$ using Hybrid in 5.69 days. In comparison, solving this instance with Well-Pooled MMT took 13.84 days.

Let us provide some more details of both computations, starting with $\mathrm{LPN}_{113,\frac{1}{4}}$. We first computed a pool of 2^{33} samples with $k_1 = 3$ bits fixed in 8 min. BKW then eliminated $k_2 = 93$ bit in $c = 3$ iterations with block-size $d = 31$ in 56 min. This gave an $\mathrm{LPN}_{17,\frac{255}{512}}$ instance, which Gauss solved in 9 min.

Thus, we recovered the first 17 bits of $\mathbf{s}$, which we then eliminated from our pool, resulting in an $\mathrm{LPN}_{93,\frac{1}{4}}$ instance. In a second iteration, BKW eliminated $k_2 = 78$ bit in $c = 3$ iterations with block-size $d = 26$ in 3 min. The resulting $\mathrm{LPN}_{15,\frac{255}{512}}$ instance was solved by Gauss in 6 min.

After eliminating these further 15 bits from our pool, we are left with an $\mathrm{LPN}_{78,\frac{1}{4}}$ instance, which was directly solved by MMT in another 19 min. The remaining $k_1 = 3$ bits were brute-forced in 125 msec.

Thus, in total it took us only 101 min to solve $\mathrm{LPN}_{113,\frac{1}{4}}$ with Hybrid.

For solving $\mathrm{LPN}_{135,\frac{1}{4}}$, we computed again a pool of 2^{33} samples with $k_1 = 10$ coordinates fixed in 1.47 days. `BKW` then eliminated $k_2 = 99$ bits in $c = 3$ iterations taking 36 min, which resulted in $2^{21.4}$ samples with $k_1 + k_2 = 109$ coordinates fixed.

This amount of samples is not yet sufficient to achieve a good success probability for solving the remaining $\mathrm{LPN}_{26,\frac{255}{512}}$ by `Gauss`. In order to increase the input samples to `Gauss` we computed a new pool of 2^{32} samples with $k_1 = 10$ coordinates fixed in 0.74 days, and exchanged half of the samples of the bigger pool that we compute in the beginning. This altered pool of size 2^{33} was then used as input to `BKW` to eliminate again $k_2 = 99$ bits in $c = 3$ iterations in another 35 min. This allowed us to double the size of the input pool for `Gauss` while increasing the runtime only by a factor of ≈ 1.5.

After acquiring enough input samples, solving $\mathrm{LPN}_{26,\frac{255}{512}}$ by `Gauss` took 3.12 days. The remaining $\mathrm{LPN}_{109,\frac{1}{4}}$ instance could be solved by another iteration and subsequent execution of `MMT` in less then 7 h. In total, solving $\mathrm{LPN}_{135,\frac{1}{4}}$ took us 5.69 days with `Hybrid`.

References

1. http://csrc.nist.gov/groups/ST/post-quantum-crypto/
2. https://computing.llnl.gov/?set=resources&page=SCF_resources#sequoia
3. Alekhnovich, M.: More on average case vs approximation complexity. In: 44th FOCS, pp. 298–307. IEEE Computer Society Press, October 2003
4. 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 (2012). doi:10.1007/978-3-642-29011-4_31
5. Belaïd, S., Coron, J.-S., Fouque, P.-A., Gérard, B., Kammerer, J.-G., Prouff, E.: Improved side-channel analysis of finite-field multiplication. In: Güneysu, T., Handschuh, H. (eds.) CHES 2015. LNCS, vol. 9293, pp. 395–415. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48324-4_20
6. Bernstein, D.J., Lange, T., Peters, C.: Smaller decoding exponents: ball-collision decoding. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 743–760. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22792-9_42
7. Blum, A., Kalai, A., Wasserman, H.: Noise-tolerant learning, the parity problem, and the statistical query model. In: 32nd ACM STOC, pp. 435–440. ACM Press, May 2000
8. Bogos, S., Tramer, F., Vaudenay, S.: On solving LPN using BKW and variants. Cryptology ePrint Archive, Report 2015/049 (2015). http://eprint.iacr.org/2015/049
9. Bogos, S., Vaudenay, S.: Observations on the LPN solving algorithm from eurocrypt'16 (2016). http://eprint.iacr.org/2016/437
10. Boyer, M., Brassard, G., Høyer, P., Tapp, A.: Tight bounds on quantum searching. arXiv preprint quant-ph/9605034 (1996)
11. Carrijo, J., Tonicelli, R., Imai, H., Nascimento, A.C.A.: A novel probabilistic passive attack on the protocols HB and HB+ (2008). http://eprint.iacr.org/2008/231
12. Damgård, I., Park, S.: How practical is public-key encryption based on LPN and ring-LPN? Cryptology ePrint Archive, Report 2012/699 (2012). http://eprint.iacr.org/2012/699

13. Döttling, N., Müller-Quade, J., Nascimento, A.C.A.: IND-CCA secure cryptography based on a variant of the LPN problem. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 485–503. Springer, Heidelberg (2012). doi:10.1007/978-3-642-34961-4_30
14. Duc, A., Vaudenay, S.: HELEN: a public-key cryptosystem based on the LPN and the decisional minimal distance problems. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) AFRICACRYPT 2013. LNCS, vol. 7918, pp. 107–126. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38553-7_6
15. Fossorier, M.P., Mihaljevic, M.J., Imai, H., Cui, Y., Matsuura, K.: A novel algorithm for solving the LPN problem and its application to security evaluation of the HB protocol for RFID authentication. Cryptology ePrint Archive, Report 2006/197 (2006). http://eprint.iacr.org/2006/197
16. Gilbert, H., Robshaw, M.J.B., Seurin, Y.: HB$^{\#}$: Increasing the security and efficiency of HB^{+}. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 361–378. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78967-3_21
17. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: 28th ACM STOC, pp. 212–219. ACM Press, May 1996
18. Guo, Q., Johansson, T., Löndahl, C.: Solving LPN using covering codes. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 1–20. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45611-8_1
19. Heyse, S., Kiltz, E., Lyubashevsky, V., Paar, C., Pietrzak, K.: Lapin: an efficient authentication protocol based on ring-LPN. In: Canteaut, A. (ed.) FSE 2012. LNCS, vol. 7549, pp. 346–365. Springer, Heidelberg (2012). doi:10.1007/978-3-642-34047-5_20
20. Hopper, N.J., Blum, M.: Secure human identification protocols. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 52–66. Springer, Heidelberg (2001). doi:10.1007/3-540-45682-1_4
21. Juels, A., Weis, S.A.: Authenticating pervasive devices with human protocols. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 293–308. Springer, Heidelberg (2005). doi:10.1007/11535218_18
22. Kachigar, G., Tillich, J.-P.: Quantum information set decoding algorithms. arXiv preprint arXiv:1703.00263 (2017)
23. Kiltz, E., Pietrzak, K., Cash, D., Jain, A., Venturi, D.: Efficient authentication from hard learning problems. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 7–26. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20465-4_3
24. Levieil, É., Fouque, P.-A.: An improved LPN algorithm. In: Prisco, R., Yung, M. (eds.) SCN 2006. LNCS, vol. 4116, pp. 348–359. Springer, Heidelberg (2006). doi:10.1007/11832072_24
25. Lyubashevsky, V.: The parity problem in the presence of noise, decoding random linear codes, and the subset sum problem. In: Chekuri, C., Jansen, K., Rolim, J.D.P., Trevisan, L. (eds.) APPROX/RANDOM -2005. LNCS, vol. 3624, pp. 378–389. Springer, Heidelberg (2005). doi:10.1007/11538462_32
26. 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 (2011). doi:10.1007/978-3-642-25385-0_6
27. May, A., Ozerov, I.: On computing nearest neighbors with applications to decoding of binary linear codes. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 203–228. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46800-5_9
28. Prange, E.: The use of information sets in decoding cyclic codes. IRE Trans. Inf. Theory **8**(5), 5–9 (1962)

29. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N., Fagin, R. (eds.) 37th ACM STOC, pp. 84–93. ACM Press, May 2005
30. Zhang, B., Jiao, L., Wang, M.: Faster algorithms for solving LPN. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 168–195. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49890-3_7