Practically Solving LPN

Thom Wiggers
Radboud University, Nijmegen, The Netherlands
thom@thomwiggers.nl

Simona Samardjiska
Radboud University, Nijmegen, The Netherlands
simonas@cs.ru.nl

***Abstract*—The best algorithms for the Learning Parity with Noise (LPN) problem require sub-exponential time and memory. This often makes memory, and not time, the limiting factor for practical attacks, which seem to be out of reach even for relatively small parameters. In this paper, we try to bring the state-of-the-art in solving LPN closer to the practical realm. We improve upon the existing algorithms by modifying the Coded-BKW algorithm to work under various memory constrains. We correct and expand previous analysis and experimentally verify our findings. As a result we were able to mount practical attacks on the largest parameters reported to date using only 2^{39} bits of memory.**

I. Introduction

The Learning Parity with Noise (LPN) problem has its roots in machine learning, where it is connected to a crucial question of learning functions in the presence of noise. But LPN is also a fundamental problem in the fields of coding theory and cryptography [3]. In essence, the LPN problem asks to recover a secret vector given noisy system of linear equations over $\mathbb{F}_2$, where the noise follows a Bernoulli distribution. The extension of LPN to fields larger than $\mathbb{F}_2$, Learning With Errors (LWE), forms the basis of many submissions in the second and third round of the NIST Post-Quantum standardization process [4]. Both LPN and LWE are believed to be hard even for adversaries with access to a quantum computer.

A lot of effort has been put in determining the hardness of the LPN problem. The best algorithms run in sub-exponential time, but also require sub-exponential amounts of memory [1], [5]–[7]. This is the main practical limitation, even for small sizes of the problem. Only recently have a series of algorithms been proposed that try to balance the demands on memory and time [2], [8], [9]. This line of research is however far from closed, since it is still not clear what the limits are of time-memory trade-offs for LPN algorithms.

A. Contributions

The focus of this paper are algorithms for solving LPN in a low memory regime. We show that it is possible to modify and enhance the Coded-BKW algorithm [7] to be used when only restricted memory is available, and that it is possible to achieve scalable time-memory trade-off for various parameters. We adopt the approach from [1] and devise an improved and more efficient chain finding algorithm under memory restrictions. Unlike suggested in [2], we show that for mid-range parameters the WHT decoding method is superior to the Gauss decoding method and is more suitable for combining with other reduction steps. This can be seen by our concrete complexity estimates in Table I that improve significantly over the similar Hybrid algorithm from [2]. Note, that without any reduction steps, using MMT [10] is still superior for larger parameters as reported in [2], however these parameters are already far from practical reach. We verify our results by practically mounting an attack against LPN for the largest parameters reported so far, using only 2^{39} bits of memory.

We thank the authors of [1] for providing the implementation of their chain-finding algorithm and the authors of [2] for providing their complexity calculation script. This work was supported by ERC Starting Grant No. 805031 (EPOQUE).

B. Organization

In Section II we give the necessary preliminaries, and in Section III we present the known solving techniques for the LPN problem. Section IV provides analysis and comparison of the two decoding methods we are interested in: `WHT` and `Gauss`. In Section V and Section VI we present our main results and the obtained best reduction chains under different memory constrains. Finally in Section VII we experimentally verify our findings.

II. Preliminaries

We will denote vectors and matrices with bold-face letters, like $\mathbf{v}$ or $\mathbf{M}$. We write inner product of two vectors as $\langle \mathbf{v}_1, \mathbf{v}_2 \rangle$ The Hamming weight of $\mathbf{v}$ is $wt(\mathbf{v})$. We write Ber_τ for a Bernoulli distribution with parameter τ. Bin_τ^n is the binomial distribution with n trails and success rate τ. We write $y \xleftarrow{\$} Y$ when we uniformly sample y from Y.

The LPN Search problem can be defined using the following definition from [11].

Definition 1: (Search LPN problem). Let $\mathbf{s} \xleftarrow{\$} \mathbb{F}_2^k$ be a secret vector of length k and let $0 \leqslant \tau < \frac{1}{2}$ be a constant noise parameter. An LPN oracle $\mathcal{O}_{\mathbf{s},\tau}^{\mathsf{LPN}}$ outputs independent random samples $(\mathbf{a}, c)$ according to the distribution:

$$\left\{ (\mathbf{a}, c) \,\middle|\, \mathbf{a} \xleftarrow{\$} \mathbb{F}_2^k, c = \langle \mathbf{a}, \mathbf{s} \rangle + e, e \leftarrow \mathrm{Ber}_\tau \right\}.$$

The Search LPN Problem, denoted by $\mathsf{LPN}_{k,\tau}^n$ is to find the (secret) vector $\mathbf{s}$, given access to the LPN oracle.

We will be interested in algorithms that solve $\mathsf{LPN}_{k,\tau}^n$ in time t, using at most n samples and using at most m bits of memory. Such an algorithm may fail with a certain probability θ. Sometimes, instead of the noise parameter τ we will use the *bias* of an LPN instance $\mathsf{LPN}_{k,\tau}$, defined as $\delta = E\left((-1)^X\right) = 1 - 2\tau$ where $X \sim \mathrm{Ber}_\tau$. We will refer to the bias of the secret as δ_s.

III. Solving LPN problems

The known algorithms that solve an LPN instance $\mathsf{LPN}_{k,\tau}$ typically follow a common structure. We can usually split them in two phases: a *reduction phase* in which a *reduction* algorithm reduces the problem to a smaller one $\mathsf{LPN}_{k',\tau'}$, $k' \leqslant k$; and a *decoding phase* in which a *decoding* algorithm recovers the secret of the smaller LPN instance. Intuitively, a smaller LPN problem is easier to decode, but a reduction typically increases the level of noise and may change the number of samples.

It is possible to apply a sequence of reduction algorithms before decoding the reduced $\mathsf{LPN}_{k',\tau'}$ instance. This is already

Input: n samples $(\mathbf{a}, c)$ from $O^{\mathsf{LPN}}_{\mathbf{s},\tau}$, list of reduction algorithms $\mathcal{R}$, and decoding algorithm D
Output: Information on s

for $R \in \mathcal{R}$ **do**
  Apply R to obtain $\mathsf{LPN}_{k',\tau'}$, $k' \leqslant k$ and n' samples.
  $k \leftarrow k'$, $n \leftarrow n'$
end for
Use decoding algorithm D, consuming n samples.
return Information on s

Figure 1. General LPN decoding algorithm

implied by the original BKW algorithm. Bogos et al. [1] proposed using chains of different reduction algorithms before applying a decoding algorithm. We summarize this meta-algorithm in Figure 1.

Note that most decoding algorithms recover only part of the secret. However, the algorithm can be repeated to obtain more information. We will, as in the literature, only discuss the first run of the algorithm, since this is the most resource-intensive of recovering the full s.

A. Reduction algorithms

We will now briefly discuss algorithms that reduce an $\mathsf{LPN}^{n}_{k,\tau}$ problem to an $\mathsf{LPN}^{n'}_{k',\tau'}$ problem. For more details on these algorithms, we refer to the cited works.

1) `drop-reduce`(b)*:* Deletes all samples that do not have b zero bits at the end.

$k' = k - b$; $n' = n2^{-b}$; $\delta' = \delta$; $\delta'_s = \delta_s$; $t = O(kn)$; $m = O(kn)$.

2) `xor-reduce`(b) *[6]:* Partitions samples by the last b bits and sums all pairs of vectors within each partition. This cancels out the last b bits. The bias of the LPN problem is squared as per the piling-up lemma: the bias of the sum of n Bernoulli variables with bias δ is δ^n.

$k' = k - b$; $n' = \frac{n(n-1)}{2^{b+1}}$; $\delta' = \delta^2$; $\delta'_s = \delta_s$;
$t = O(k \max(n, n'))$; $m = O(\max(kn, k'n'))$.

3) `sparse-secret` *[1], [7], [12]:* Transforms the problem so that the secret is Bernoulli-distributed with $\tau < \frac{1}{2}$ instead of uniform. This reduction does not simplify the LPN problem, but is necessary for `code-reduce`.

$k' = k$; $n' = n - k$; $\delta' = \delta$; $\delta'_s = \delta$; $m = O(kn)$
$t = O\left(\min_{\chi\in\mathbb{N}}\left(\frac{n'k^2}{\log_2 k - \log_2\log_2 k} + k^2, kn'\lceil\frac{k}{\chi}\rceil + k^3 + k\chi 2^{\chi}\right)\right)$.

4) `code-reduce`$([k, k']\ code)$ *[1], [7], [11]:* Uses the covering property of codes to reduce the LPN problem size. Using a linear $[k, k']$ code, `code-reduce` approximates the samples to the closest codeword of the code. The effect on the bias is called bc and depends on the original δ and the properties of the code. A bigger bc is better, as it will maximize the bias of the reduced LPN instance.

Theorem 1: (Upper bound for bc [1]). A $[k, k', D]$ linear code C has for any $r \in \mathbb{N}$ and $\delta_s \in [0, 1]$:

$$\mathsf{bc} \leqslant 2^{k'-k}\sum_{w=0}^{r}\binom{k}{w}\left(\delta_s^w - \delta_s^{r+1}\right) + \delta_s^{r+1}.$$

Equality for any δ_s implies that C is a *(quasi-)perfect* code, in which case r equals the packing radius $R = \lfloor\frac{D-1}{2}\rfloor$.
In [7] the analysis of `code-reduce` was done for codes that reach the bound in Theorem 1. This overestimates the efficiency of the reduction. In practice we know few codes that are close to the bound and have efficient decoding. Instead, [1] concatenates small codes that either reach or are close to the bound. We use

Input:A set V of s k'-bit samples $(\mathbf{a}, c) \in O^{\mathsf{LPN}}_{\mathbf{s}',\tau'}$.
Output: $(\mathbf{s}'_1, \ldots, \mathbf{s}'_{k'})$ from $\mathbf{s}'$
$f(\mathbf{x}) = \sum_{(\mathbf{a},c)\in V} 1_{V_{1,\ldots,k'}=\mathbf{x}}(-1)^c$
$\hat{f}(\mathbf{x}) = \sum_x (-1)^{\langle \mathbf{a},\mathbf{x}\rangle} f(x)$
return $(\mathbf{s}'_1, \ldots, \mathbf{s}'_{k'}) = \arg\max_{\mathbf{a}\in\mathbb{Z}_2^{k'}}(\hat{f}(\mathbf{a}))$

Figure 2. WHT algorithm [6]

function GAUSS($O^{\mathsf{LPN}}_{\mathbf{s}',\tau'}$, τ')
  repeat
    $(\mathbf{A}, \mathbf{c}) \leftarrow \left(O^{\mathsf{LPN}}_{\mathbf{s}',\tau'}\right)^{k'}$ such that $\mathbf{A}$ is full rank
    $\mathbf{s}' = \mathbf{A}^{-1}\mathbf{c}$
  until TEST($\mathbf{s}'$, τ', $\frac{1}{2^k}$, $\left(\frac{1-\tau'}{2}\right)^k$)
  return $\mathbf{s}'$
end function
function TEST($\mathbf{s}'$, τ', α, β)
  $s = \left(\frac{\sqrt{\frac{3}{2}\ln(\frac{1}{\alpha})} + \sqrt{\ln(\frac{1}{\beta})}}{\frac{1}{2} - \tau'}\right)^2$, $c = \tau' s + \sqrt{3\left(\frac{1}{2} - \tau'\right)\ln\left(\frac{1}{\alpha}\right)s}$
  $(\mathbf{A}, \mathbf{c}) \leftarrow \left(O^{\mathsf{LPN}}_{\mathbf{s}',\tau'}\right)^{s}$
  return $wt(\mathbf{A}\mathbf{s}' + \mathbf{c}) \leqslant c$
end function

Figure 3. Gauss algorithm [2]

the same approach. As the modified δ_s is hard to quantify, we only allow `code-reduce` to be applied once.

$k = k'$; $n' = n$; $\delta' = \delta \cdot \mathsf{bc}$; $t = O(kn)$; $m = O(kn)$.

5) c`-sum-Dissection`(b)*:* It is possible to sum up more than just two samples, such that the last bits add up to 0. This was initially proposed in [13] as LF(4). [9] rephrased it as a time-memory trade-off for solving LPN problems. They use the Dissection technique [14] to solve c-sum problems in lists of samples. Dissection requires that c is one of $c_i \in \left\{\frac{1}{2}\left(i^2 + 3i + 4\right) \mid i \in \mathbb{N}\right\}$. The first few c are 2, 4, 7, 11. It also requires that $\log_2(n/c_i) \leqslant b/i$.

$k' = k - b$; $n' = \binom{n}{c} \cdot 2^{-b}$; $\delta' = \delta^c$; $\delta'_s = \delta_s$; $t = O\left(2^{c_{i-1}\frac{n}{c_i}}\right)$;
$m = O(kn)$.

Note that [8] further improved c`-sum-Dissection` by using the Van Oorschot-Wiener Parallel Collision Search (PCS) algorithm [15]. We denote this variant as c`-sum-PCS`(b).

B. Decoding algorithms

The general algorithm from Figure 1 for solving LPN reduces $\mathsf{LPN}^{n}_{k,\tau}$ to a smaller instance $\mathsf{LPN}^{n'}_{k',\tau'}$ through a number of reduction steps. It then solves the final instance using some sort of decoding algorithm. The original BKW used majority decoding [5]. This was improved by using the Walsh-Hadamard transform (`WHT`) [6] and subsequently used in [1], [7].

$t = k' \cdot 2^{k'-1}(\log s + 1) + k's$; $m = k'(2^{k'} + s)$.

Esser et al. [2] used the folklore `Gauss` algorithm that performs simple Gaussian eliminations using k' samples, assuming error-freeness. The obtained candidate $\mathbf{s}'$ is then tested against s samples to determine whether the error's distribution is closer to Bin^s_τ or $\mathrm{Bin}^s_{\frac{1}{2}}$. The `Pooled-Gauss` variant randomly selects samples from a re-used pool.

$t = (k'^3 + k's) \cdot \log^2 k' \cdot (1 - \tau')^{-k'}$; $m = k'(k' + s)$.

The two algorithms are given in Figure 2 and Figure 3.

C. Finding the best reduction chain

Bogos et al. proposed in [1] to search for the most efficient combination of reductions (*reduction chain*) before decoding the problem. They present their algorithm as an automaton that defines all possible reduction paths. They used (concatenated) perfect, quasi-perfect and random codes for the `code-reduce` reduction and failure probability $\theta = 0.33$. We modify the algorithm to include the `Pooled-Gauss` decoding algorithm, as well as more reduction techniques. We present the updated automaton in Figure 4.

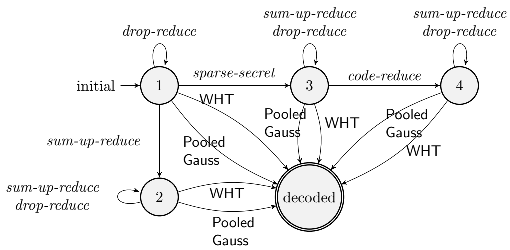


Figure 4. The automaton accepting valid LPN reduction chains. `sum-up-reduce` represents any of the reductions combining samples, i.e. `xor-reduce`, `lf4-reduce`, c-`sum-Dissection` or c-`sum-PCS`.

IV. Fair Comparison between WHT and Gauss

We revisit both `WHT` and `Gauss` and provide a unified analysis in order to compare them. Our analysis shows that assuming negligible decoding error $1/2^k$, both algorithms require (almost) the same number of samples. However, their efficiency depends very differently on the size of the problem and the bias. As a consequence, they are suitable for different scenarios. This has several implications.

First, we show that there is no obstacle in obtaining a negligible error in `WHT` by choosing an appropriate number of samples. This was overlooked in [2].

Second, we provide the basis for a fair comparison between chains of reduction steps ending in `Gauss` and `WHT`. As we will see later, this disproves the claim in [2] that `Gauss` can be combined with various reduction steps and give better results than performing reduction steps and using `WHT`. This further explains the experimental results from [2] which imply that `Gauss` almost always performs better without any `sum-up-reduce` reduction steps.

As a side result, we improve the efficiency of `Gauss` by obtaining a better bound for the sample complexity.

Proposition 1: If $s = \left(\frac{4}{(1-2\tau)^2} - 2\right) \ln \frac{1}{\sqrt{2\pi}\gamma}$ samples are available, where $\gamma \in (0, \frac{1}{\sqrt{2\pi e}}]$, the `WHT` algorithm applied to $\mathrm{LPN}^n_{k,\tau}$ outputs the correct solution with probability at least $1-\gamma$.

Proof: We detail the analysis for a positive bias following the approach of [1]. For a negative bias, the analysis is equivalent. `WHT` outputs the candidate with the largest value of $\hat{f}$. A failure occurs when there exists another $\bar{\mathbf{s}} \neq \mathbf{s}$ such that $\hat{f}(\bar{\mathbf{s}}) > \hat{f}(\mathbf{s})$ i.e. when $\mathrm{HW}(\mathbf{A}\bar{\mathbf{s}} + \mathbf{c}) < \mathrm{HW}(\mathbf{A}\mathbf{s} + \mathbf{c})$. Let $\bar{\mathbf{y}} = \mathbf{A}\bar{\mathbf{s}} + \mathbf{c}$ and $\mathbf{y} = \mathbf{A}\mathbf{s} + \mathbf{c}$. Then the expectation and variance of random variables $\mathbf{x}_i = \mathbf{y}_i - \bar{\mathbf{y}}_i$ is $\mathrm{E}(\mathbf{x}_i) = \frac{2\tau-1}{2}$ and $\mathrm{Var}(\mathbf{x}_i) = \frac{1}{2} - \left(\frac{2\tau-1}{2}\right)^2$. Let $Z = \sqrt{s}(S_s - \mathrm{E}(\mathbf{x}_i))/\sqrt{\mathrm{Var}(\mathbf{x}_i)}$, where $S_s = \frac{\mathbf{x}_1+\cdots+\mathbf{x}_s}{s}$. By the Central Limit Theorem $Z \xrightarrow{d} N(0,1)$. Using standard upper-tail inequalities for the standard normal distribution $N(0,1)$, we obtain

$$Pr\left[\hat{f}(\bar{\mathbf{s}}) > \hat{f}(\mathbf{s})\right] = Pr\left[Z \geqslant \frac{(1-2\tau)\sqrt{s}}{\sqrt{2-(1-2\tau)^2}}\right] \leqslant \frac{e^{-\frac{(1-2\tau)^2 s}{2(2-(1-2\tau)^2)}}}{\sqrt{2\pi}} \quad (1)$$

Taking $s = \left(\frac{4}{(1-2\tau)^2} - 2\right) \ln \frac{1}{\sqrt{2\pi}\gamma}$, inequality (1) becomes

$$Pr\left[\hat{f}(\bar{\mathbf{s}}) > \hat{f}(\mathbf{s})\right] \leqslant \gamma.$$

We can make the probability of an error in the WHT procedure arbitrarily small if we take $\gamma = \mathrm{negl}(k)$. ■

Proposition 2: If $s = \left(\frac{\sqrt{2\tau(1-\tau)\ln(\frac{1}{\sqrt{2\pi}\alpha})} + \sqrt{\frac{1}{2}\ln(\frac{1}{\sqrt{2\pi}\beta})}}{\frac{1}{2}-\tau}\right)^2$ samples are available for $\alpha, \beta \in (0, \frac{1}{\sqrt{2\pi e}}]$, the Test function from the Gauss algorithm applied on $\mathrm{LPN}^n_{k,\tau}$ accepts the correct solution with probability at least $1-\alpha$, and rejects incorrect solutions with probability at least $1-\beta$.

Proof: A correct $\mathbf{s}'$ input to the Test algorithm, means that $\mathbf{e} = \mathbf{A}\mathbf{s}' + \mathbf{c}$ follows the Binomial distribution Bin^s_τ i.e. $\mathbf{e}_i \sim \mathrm{Ber}_\tau$, $i \in \{1, \ldots, s\}$. Then $\mathrm{E}(\mathbf{e}_i) = \tau$ and $\mathrm{Var}(\mathbf{e}_i) = \tau(1-\tau)$. Using the same approach as in Proposition 1 for $Z = \frac{\sqrt{s}(S_s - \mathrm{E}(\mathbf{e}_i))}{\sqrt{\mathrm{Var}(\mathbf{e}_i)}}$, and $S_s = \frac{\mathbf{e}_1+\cdots+\mathbf{e}_s}{s}$, and we obtain

$$Pr\left[\mathrm{HW}(\mathbf{A}\mathbf{s}' + \mathbf{c}) \geqslant c\right] \leqslant \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2s} \cdot \frac{(c - s\tau)^2}{\tau(1-\tau)}\right) \quad (2)$$

Taking $c = s\tau + \sqrt{2s\tau(1-\tau)\ln\frac{1}{\sqrt{2\pi}\alpha}}$ (similarly as in [2]), Equation (2) turns into $Pr\left[\mathrm{HW}(\mathbf{A}\mathbf{s}' + \mathbf{c}) \geqslant c\right] \leqslant \alpha$. For the chosen c, the probability that a correct $\mathbf{s}'$ will produce an error $\mathbf{e}$ of larger weight than c can be made negligible. Therefore we can use this c as a threshold value in the Test algorithm.

We estimate $Pr\left[\mathrm{HW}(\mathbf{A}\mathbf{s}' + \mathbf{c}) \leqslant c\right]$ similarly,

$$Pr\left[\mathrm{HW}(\mathbf{A}\mathbf{s}' + \mathbf{c}) \leqslant c\right] \leqslant \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(s-2c)^2}{2s}\right) \quad (3)$$

Taking $s = \left(\frac{\sqrt{2\tau(1-\tau)\ln(\frac{1}{\sqrt{2\pi}\alpha})} + \sqrt{\frac{1}{2}\ln(\frac{1}{\sqrt{2\pi}\beta})}}{\frac{1}{2}-\tau}\right)^2$ and the previously found c, Equation (3) turns into $Pr\left[\mathrm{HW}(\mathbf{A}\mathbf{s}' + \mathbf{c}) \leqslant c\right] \leqslant \beta$. With this we have also estimated the required amount of samples needed for the Test function. ■

In order to compare fairly the two decoding algorithms, the errors γ for `WHT` and $\alpha + \beta$ for `Gauss` should be approximately the same. For simplicity we take $\alpha = \beta = \gamma = 1/(2^k\sqrt{2\pi})$. Then we get approximately the same amount of needed samples i.e.

$$s_G \approx \frac{8k\ln 2}{(1-2\tau)^2}, \quad s_{WHT} \approx \frac{4k\ln 2}{(1-2\tau)^2}$$

This shows that we can ignore the sample number s from the time and memory expressions of both decoding algorithms and look at them as functions in k' and τ'. Interestingly, the time complexity of both algorithms is exponential in k', but with different bases: 2 for WHT and $((1-\tau)^{-1})$ for Gauss. As we add more reduction steps, $((1-\tau)^{-1})$ grows and the `Gauss` algorithm quickly overruns `WHT`. Hence, we can expect that having more reduction steps favors `WHT` instead of `Gauss` as this reduces the LPN problem, and it becomes more likely that we can fit the WHT algorithm in memory. This observation is shown in Figure 5 and further confirmed in Subsection III-C.

V. Combining code-reduce with Gauss

In [2] it was suggested that the low-memory `Gauss` decoding algorithms can be combined with various reduction algorithms. The intuitive combination with the `code-reduce` reduction that uses little memory and does not consume any samples,

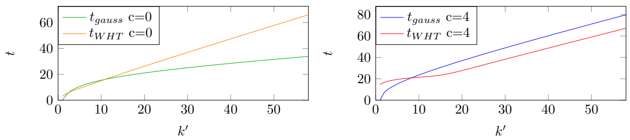


Figure 5. Comparing WHT and Gauss for different $\delta' = \delta^{2^c}$. c indicates the number of reduction steps.

Input: $n = k + k^2 \log_2^2 k + s$ samples from $O^{\mathsf{LPN}}_{\mathrm{s},\tau}$,
a $[k, k']$ code C with generator matrix G
Output: Linear relations on s

sparse-secret()
code-reduce(k, k', C)
$s \leftarrow$ Pooled-Gauss(k')
return s' of size k' such that $\mathrm{s}G^T = \mathrm{s}'$

Figure 6. Coded Pooled Gauss

would appear to make sense. Using `Pooled-Gauss`, a variant that does not regenerate samples, this combination looks like Figure 6. However, we will show that this approach is not more viable than just applying `Pooled-Gauss` to the full problem. Even hypothetical codes that reach the Hamming Bound [16] don't have good enough bc that makes Coded (Pooled) Gauss better.

In our analysis we assume that we can decode a sample in insignificant time. We explore whether even under this assumption, Coded Gauss can be competitive. In practice, constant decoding time is only feasible for (concatenations of) small codes. Those are not the best possible covering codes theoretically.

A. Analysis of the required bias of the code

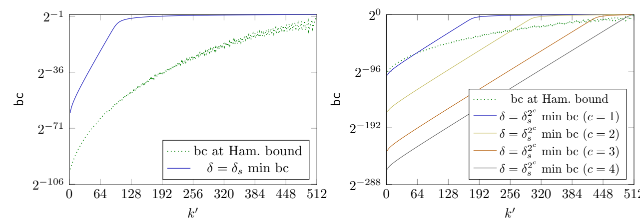
In order for Coded Pooled Gauss to have advantage over Plain Pooled Gauss, we need the time complexity of Coded Pooled Gauss to be better, i.e.

$$\frac{(k^3 + ks)\log_2^2 k}{\left(\frac{1}{2} + \frac{1}{2}\delta\right)^k} \geqslant \frac{(k'^3 + k's)\log_2^2 k'}{\left(\frac{1}{2} + \frac{1}{2}\delta \mathsf{bc}\right)^{k'}} + s + n. \tag{4}$$

Recall that Theorem 1 bounds the bc of any $[k, k']$ code and that the bound is met for perfect or quasi-perfect codes. Combining it with the Hamming bound, reached by perfect codes, $(2^{k'} \geqslant \sum_{w=0}^{R} \binom{k}{w})$, we can compute the upper bound on bc for any $[k, k']$ code. In turn, this gives us the best possible time complexity for Coded (Pooled) Gauss using any $[k, k']$ code. Unfortunately, our calculations show (see Figure 7(a)) that the required bc can not be reached even for codes on the Hamming bound. This implies that Coded (Pooled) Gauss is always worse than immediately applying (Pooled) Gauss.

Note that here, since we only combine `code-reduce` and `Gauss` we have $\delta = \delta_s$ (the sparse-secret transformation is performed right before `code-reduce`). However, in order for the code-reduce step to be worth applying we actually need $\delta < \delta_s$. This corresponds to applying other reduction steps in between sparse-secret and code-reduce. Figure 7(b) depicts this scenario. As before, c indicates the number of reduction steps. Note that as c increases, so does the possible advantage of applying `code-reduce`.

The previous analysis does not give the full picture. We have neglected the running time of the in-between steps for the sake of argument and to show that the only favorable case involves several reduction steps before Coded Gauss.



(a) $\delta = \delta_s; \tau = \frac{1}{8}$ (b) $\delta = \delta_s^{2^c} < \delta_s = \frac{3}{4}$

Figure 7. Minimal bc for Coded Gauss to be faster than just applying Gauss and the bc obtained at the Hamming bound. (b) actually requires additional reduction steps before `code-reduce`.

B. Memory Cost

The samples used by `Gauss` to test if candidate s′ are correct greatly contributes to its memory consumption. With small bias, `Gauss` is not memory-efficient. For quite realistic $\delta \cdot \mathsf{bc} \approx 10^{-6}$ and $k' \gtrapprox 16$, `Gauss` needs many terabytes of memory. When $\delta \cdot \mathsf{bc} \approx 10^{-7}$, it even crosses into the exabytes. This further limits realistic attacks. We note that relaxing the failure probability reduces the memory consumption, though not by many orders of magnitude. However, this could make the difference for a practical attack to fit in memory.

VI. Finding memory restricted reduction chains

Our main goal here is to find the best reduction chains in the spirit of [1] but under memory constraints. As a first step, we modified the chain finding algorithm from [1] to only allow branches to be taken if the memory consumed by the reduction or decoding is below a set limit. Although in theory this approach should yield the best chain in the end, it is extremely inefficient, time consuming and does not scale well. This was especially visible after adding new reduction steps to the algorithm. However, we noticed that the automaton can be greatly simplified due to many impossible branches and some clear optimization steps due to the memory restrictions.

Proposition 3: The sequence `sum-up-reduce` $\rightarrow$ `drop-reduce` can never occur in the best reduction chain for solving a given $\mathsf{LPN}_{k_0,\tau_0}$ search problem under memory constrains.

Proof: We will prove the claim for `sum-up-reduce=xor-reduce`. The rest can be shown very similarly. Suppose that after a number of reduction steps we need to reduce the problem $\mathsf{LPN}_{k,\tau}$. Using the sequence `xor-reduce` $\rightarrow$ `drop-reduce`, we can reduce it first to $\mathsf{LPN}_{k-b,\tau'}$ using `xor-reduce`, and then to $\mathsf{LPN}_{k',\tau'}$ using `drop-reduce`. Here $\tau' = (1 - (1 - 2\tau)^2)/2$ and $b \in [0, k - k']$. The sequence takes time $t = k \max\{n, \frac{n(n-1)}{2^{b+1}}\} + (k - b)\frac{n(n-1)}{2^{b+1}}$ and memory $m = \max\{kn, (k-b)\frac{n(n-1)}{2^{b+1}}\}$. For some constants A, B, C, these can be written as functions in b as $t(b) = A + n(n-1)\frac{Bk-b}{2^{b+1}}$ and $m(b) = A + Cn(n-1)\frac{k-b}{2^{b+1}}$. It is easy to see that both functions are strictly decreasing in b, so the minimum on $[0, k - k']$ is achieved when $b = k - k'$. Note further that the number of remaining samples does not depend on b, so the choice of b does not affect subsequent reduction steps. Summarizing, in the best chain any sequence `xor-reduce` $\rightarrow$ `drop-reduce` collapses to just `xor-reduce`. ∎

We also looked into the sequences `code-reduce` $\rightarrow$ `drop-reduce` and `code-reduce` $\rightarrow$ `sum-up-reduce`. However, due to the very complex relation between the time complexity and the bias bc of the code, we could not make a compact analysis similar to Proposition 3. Instead, we performed an extensive set of experiments where we tested the appearance of these sequences just before a decoding algorithm is applied,

Table I. Complexities of solving $\mathsf{LPN}_{k,\tau}$ in restricted memory

	k=	**128**			**256**			**384**			**512**		
τ	**m=**	**40**	**60**	**80**	**40**	**60**	**80**	**40**	**60**	**80**	**40**	**60**	**80**
0.05	**Our work**	$\mathbf{26}^{W}$	$\mathbf{26}^{W}$	$\mathbf{26}^{W}$	$\mathbf{38}^{W}$	$\mathbf{38}^{W}$	$\mathbf{38}^{W}$	$\mathbf{58}^{G}$	$\mathbf{49}^{W}$	$\mathbf{49}^{W}$	$\mathbf{68}^{W}$	$\mathbf{58}^{W}$	$\mathbf{58}^{W}$
	Hybrid / MMT	37 34	37 34	37 34	54 40	54 40	54 40	70 48	68 48	68 48	87 57	87 57	84 57
0.10	**Our work**	$\mathbf{31}^{W}$	$\mathbf{31}^{W}$	$\mathbf{31}^{W}$	$\mathbf{50}^{W}$	$\mathbf{46}^{W}$	$\mathbf{46}^{W}$	$\mathbf{81}^{G}$	$\mathbf{60}^{W}$	$\mathbf{60}^{W}$	$\mathbf{99}^{W}$	$\mathbf{92}^{W}$	$\mathbf{73}^{W}$
	Hybrid / MMT	41 38	41 38	41 38	76 53	61 53	61 53	106 70	106 70	74 70	136 87	136 87	101 87
0.125	**Our work**	$\mathbf{33}^{W}$	$\mathbf{33}^{W}$	$\mathbf{33}^{W}$	$\mathbf{56}^{W}$	$\mathbf{49}^{W}$	$\mathbf{49}^{W}$	$\mathbf{92}^{G}$	$\mathbf{71}^{W}$	$\mathbf{64}^{W}$	$\mathbf{114}^{W}$	$\mathbf{105}^{W}$	$\mathbf{78}^{W}$
	Hybrid / MMT	41 41	41 41	41 41	86 61	61 61	61 61	121 81	110 81	81 81	157 102	157 102	101 102
0.25	**Our work**	$\mathbf{38}^{W}$	$\mathbf{38}^{W}$	$\mathbf{38}^{W}$	$\mathbf{102}^{G}$	$\mathbf{58}^{W}$	$\mathbf{58}^{W}$	$\mathbf{140}^{G}$	$\mathbf{92}^{W}$	$\mathbf{77}^{W}$	$\mathbf{179}^{G}$	$\mathbf{186}^{G}$	$\mathbf{115}^{W}$
	Hybrid / MMT	47 57	47 57	47 57	113 95	69 95	69 95	175 134	135 134	104 134	230 172	202 172	171 172
0.40	**Our work***	$\mathbf{51}^{W}$	$\mathbf{48}^{W}$	$\mathbf{48}^{W}$	$\mathbf{136}^{G}$	$\mathbf{84}^{W}$	$\mathbf{71}^{W}$	$\mathbf{189}^{G}$	$\mathbf{176}^{G}$	$\mathbf{116}^{W}$	$\mathbf{245}^{G}$	$\mathbf{241}^{G}$	$\mathbf{209}^{W}$
	Hybrid / MMT	62 75	57 75	57 75	129 132	93 132	81 132	197 189	160 189	139 189	264 245	228 245	207 245

^{G}: Gauss decoding method. ^{W}: WHT decoding method.
Hybrid / MMT per [2], generated by a version of their script that contains a bug-fix acknowledged by the authors. *: 0.40 results do not use random codes from [1].

i.e. sequences of type `code-reduce` → `reduce` → `decode`. Our experiments showed that such sequences never appear, and that they collapse to `code-reduce` → `decode`. Therefore, we decided to not allow in the automaton any other reduction steps after `code-reduce`.

As a final modification, we put `drop-reduce` as a first step. This is a logical choice in a memory restricted environment and has been used in previous works as well [2]. Samples can be generated on the fly and discarded immediately if they don't satisfy the requirements of `drop-reduce`. This creates a time-memory trade-off since only the reduced samples from `drop-reduce` remain in memory.

We updated the automaton from Figure 4 using our findings, and what we get is depicted in Figure 8.

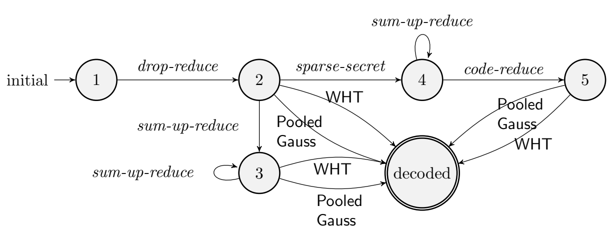


Figure 8. The updated automaton using the results from Section VI. The notation is the same as in Figure 4.

A. Experimental Results

We applied our algorithm to find reduction chains that fit in 2^{40} (128 GiB), 2^{60} (128 PiB) and 2^{80} (128 ZiB) bits of memory. 2^{40} bits is an amount of memory that is readily available from server vendors in common configurations. 2^{60} bits is a much larger, but not necessarily impractical amount of memory. A top supercomputer, Summit, has over 250 PB of storage [17].[1] Finally, 2^{80} is included to give some safety margin.

In Table I, we show that solving most LPN instances is fastest using the `WHT` decoding algorithm. Only when we get severely memory-restricted, does the algorithm find chains with `Gauss`. This improves upon the results of [2], who were not able to fit any `WHT`-based algorithm in 2^{60} bits of memory. We also see that the found reduction and decoding chain is able to recover $\mathsf{LPN}_{256,0.25}$ in 2^{58} time. This is a significant improvement on the complexity of 2^{63} for their best attack on $\mathsf{LPN}_{256,0.25}$, which involved a quantum algorithm. Going up to $m = 80$ shows that more memory does not necessarily allow for better algorithms. This is probably related to the fact that the most significant factor affecting memory requirements is the number of samples, which in turn affects the required time.

[1]While this is networked storage, Summit nodes have over 10 PB of local storage combined. 128 PB of RAM is probably within reach in the near future.

VII. Practical attack on LPN

Using our results, with memory limit $m = 39$, we have executed several attacks. Results are listed in Table II. We implemented the reductions and solving algorithms in Rust.[2] We hope these results and memory bounds are meaningful and illustrate what some time complexities mean in practice. We ran them on a computer with 192 GB RAM and two Intel Xeon Gold 6230s totaling 80 threads. Their runtime, due to the tight memory restriction, is dominated by drop-reduce, so we also give the number of bits dropped.

Table II. Solved $\mathsf{LPN}_{k,\tau}$ instances with $m = 39$

k	τ	Exp. time	Init. samples	`drop` bits	runtime
190	1/8	$2^{40.9}$	$2^{31.0}$	7	33 minutes
200	1/8	$2^{44.4}$	$2^{31.2}$	12	290 minutes
150	1/4	$2^{44.5}$	$2^{31.4}$	12	281 minutes
154	1/4	$2^{48.4}$	$2^{31.4}$	16	3 741 minutes

We see that our results scale in line with the theoretical complexity. For $k = 512$, we see that for $\tau = \frac{1}{8}$ the theoretical time complexity is $t = 2^{114}$. Extrapolating to this complexity, we would expect to need 2^{77} minutes to run an attack in practice, with our implementation. Of course, both extrapolations assume the exact same hardware and software for attacking a problem of this size. There is potential for acceleration by using GPUs or trivially distributing e.g. `drop-reduce` over multiple computers. We leave this for future work.

VIII. Conclusion

In this paper we focused on practical consideration for solving the LPN problem, in particular the issue of memory consumption. We improved the state-of-the-art by modifying and enhancing the Coded-BKW algorithm to work under various memory constraints. Our analysis of Coded (Pooled) Gauss disproved that this intuitive combination of low-memory algorithms is generally feasible. We further showed that when combined with several reduction steps, Gauss is generally always worse than using WHT, especially for practical parameters. The practicality of our approach was demonstrated by mounting attacks on the largest parameters reported so far, in only 2^{39} bits of memory.

[2]Our software is available at https://thomwiggers.nl/publication/lpn/.

References

[1] S. Bogos and S. Vaudenay, "Optimization of LPN solving algorithms," in Advances in Cryptology – ASIACRYPT 2016, Part I, ser. Lecture Notes in Computer Science, J. H. Cheon and T. Takagi, Eds., vol. 10031. Springer, Heidelberg, Dec. 2016, pp. 703–728.

[2] A. Esser, R. Kübler, and A. May, "LPN decoded," in Advances in Cryptology – CRYPTO 2017, Part II, ser. Lecture Notes in Computer Science, J. Katz and H. Shacham, Eds., vol. 10402. Springer, Heidelberg, Aug. 2017, pp. 486–514.

[3] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in 37th Annual ACM Symposium on Theory of Computing, H. N. Gabow and R. Fagin, Eds. ACM Press, May 2005, pp. 84–93.

[4] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, "Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process," National Institute of Standards and Technology, Tech. Rep. NIST Internal or Interagency Report (NISTIR) 8309, Jul. 2020. [Online]. Available: https://csrc.nist.gov/publications/detail/nistir/8309/final

[5] A. Blum, A. Kalai, and H. Wasserman, "Noise-tolerant learning, the parity problem, and the statistical query model," in 32nd Annual ACM Symposium on Theory of Computing. ACM Press, May 2000, pp. 435–440.

[6] É. Levieil and P.-A. Fouque, "An improved LPN algorithm," in SCN 06: 5th International Conference on Security in Communication Networks, ser. Lecture Notes in Computer Science, R. D. Prisco and M. Yung, Eds., vol. 4116. Springer, Heidelberg, Sep. 2006, pp. 348–359.

[7] Q. Guo, T. Johansson, and C. Löndahl, "Solving LPN using covering codes," in Advances in Cryptology – ASIACRYPT 2014, Part I, ser. Lecture Notes in Computer Science, P. Sarkar and T. Iwata, Eds., vol. 8873. Springer, Heidelberg, Dec. 2014, pp. 1–20.

[8] C. Delaplace, A. Esser, and A. May, "Improved low-memory subset sum and LPN algorithms via multiple collisions," Cryptology ePrint Archive, Report 2019/804, 2019, https://eprint.iacr.org/2019/804.

[9] A. Esser, F. Heuer, R. Kübler, A. May, and C. Sohler, "Dissection-BKW," in Advances in Cryptology – CRYPTO 2018, Part II, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds., vol. 10992. Springer, Heidelberg, Aug. 2018, pp. 638–666.

[10] A. May, A. Meurer, and E. Thomae, "Decoding random linear codes in $\tilde{O}(2^{0.054n})$," in Advances in Cryptology – ASIACRYPT 2011, ser. Lecture Notes in Computer Science, D. H. Lee and X. Wang, Eds., vol. 7073. Springer, Heidelberg, Dec. 2011, pp. 107–124.

[11] S. Bogos, F. Tramer, and S. Vaudenay, "On solving LPN using BKW and variants," Cryptology ePrint Archive, Report 2015/049, 2015, http://eprint.iacr.org/2015/049.

[12] B. Applebaum, D. Cash, C. Peikert, and A. Sahai, "Fast cryptographic primitives and circular-secure encryption based on hard learning problems," in Advances in Cryptology – CRYPTO 2009, ser. Lecture Notes in Computer Science, S. Halevi, Ed., vol. 5677. Springer, Heidelberg, Aug. 2009, pp. 595–618.

[13] B. Zhang, L. Jiao, and M. Wang, "Faster algorithms for solving LPN," in Advances in Cryptology – EUROCRYPT 2016, Part I, ser. Lecture Notes in Computer Science, M. Fischlin and J.-S. Coron, Eds., vol. 9665. Springer, Heidelberg, May 2016, pp. 168–195.

[14] I. Dinur, O. Dunkelman, N. Keller, and A. Shamir, "Efficient dissection of composite problems, with applications to cryptanalysis, knapsacks, and combinatorial search problems," in Advances in Cryptology – CRYPTO 2012, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, Heidelberg, Aug. 2012, pp. 719–740.

[15] P. C. van Oorschot and M. J. Wiener, "Parallel collision search with cryptanalytic applications," Journal of Cryptology, vol. 12, no. 1, Jan. 1999, pp. 1–28.

[16] R. W. Hamming, "Error detecting and error correcting codes," The Bell System Technical Journal, vol. 29, no. 2, Apr. 1950, pp. 147–160.

[17] Oak Ridge National Laboratory. Summit FAQs. Accessed 2021-01-25. [Online]. Available: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/summit-faqs/