Advanced cryptology (GBX9SY06)
*
Some coding theory aspects (useful) in cryptography

Pierre Karpman

2018-11

1 First definitions; examples

A *linear code* of length n and dimension k over a field $\mathbb{K}$ is a k-dimensional subspace of $\mathbb{K}^n$. In these notes, we will focus on *binary* codes, for which $\mathbb{K} = \mathbb{F}_2$, or possibly an extension thereof.[*] An important characteristic of a code is the *minimum distance* (for the Hamming distance $hd(\cdot,\cdot)$) d between two (distinct) codewords. Define $wt(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{F}_2^n$ as the number of non-zero coordinates of $\boldsymbol{x}$; $hd(\boldsymbol{x},\boldsymbol{y})$ as $wt(\boldsymbol{x}+\boldsymbol{y})$. Then the minimum distance of a code $\mathcal{C}$ is $\min_{\boldsymbol{x}\in\mathcal{C},\boldsymbol{y}\neq\boldsymbol{c}\in\mathcal{C}} hd(\boldsymbol{x},\boldsymbol{y})$, which by linearity of $\mathcal{C}$ is equivalent to $\min_{\boldsymbol{x}\neq\boldsymbol{0}\in\mathcal{C}} wt(\boldsymbol{x})$. The main parameters of length, dimension, and minimum distance of a code $\mathcal{C}$ are summarized by saying that $\mathcal{C}$ is an $[n,k,d]_{\mathbb{F}_2}$ code. While determining n and k is usually straightforward, it is in general a hard problem to compute d.

Given a code $\mathcal{C}$, it will often be necessary to have an explicit *encoding* map $\boldsymbol{x} \in \mathbb{F}_2^k \mapsto \boldsymbol{x}' \in \mathcal{C}$ from *messages* to *codewords*. Such a map can be easily obtained by sampling k linearly-independent codewords $\boldsymbol{g}_1,\ldots,\boldsymbol{g}_k$ and forming the *generator matrix* $\boldsymbol{G}$ whose rows are the $\boldsymbol{g}_i$s.[†] The encoding map is then simply $\boldsymbol{x} \mapsto \boldsymbol{x} \times \boldsymbol{G}$. One should remark that in general the matrix $\boldsymbol{G}$ (and thence the encoding map) will not be unique, as it depends on the selected codewords. A specific class of generator matrices are the ones in *systematic form*, corresponding to block matrices $\begin{pmatrix}\boldsymbol{I}_k & \boldsymbol{A}\end{pmatrix}$, where $\boldsymbol{I}_k$ is the k-dimensional identity matrix and $\boldsymbol{A}$ is a *redundancy* block. A code always admits at least one systematic encoder, up to a permutation of the coordinates of its codewords. One may be obtained by selecting k linearly-independent columns of a generator matrix $\boldsymbol{G}$; applying a column permutation on $\boldsymbol{G}$ such that those columns are in the first k positions; and computing the reduced row-echelon form of $\boldsymbol{G}$. In other words, one obtains an encoder in systematic form by finding a permutation matrix $\boldsymbol{P}$ such that $\boldsymbol{GP}$ is of the form $\boldsymbol{G}' := \begin{pmatrix}\boldsymbol{G}_1 & \boldsymbol{G}_2\end{pmatrix}$ where $\boldsymbol{G}_1$ is invertible, and by computing $\boldsymbol{G}_1^{-1}\boldsymbol{G}'$.

From the existence of a systematic encoder, one deduces that the largest possible minimum distance (or weight) of an $[n,k]$ linear code is $d_{\text{MDS}} = n-k+1$, which is a special case of the Singleton bound. Indeed, the maximum possible weight of any row of a systematic encoder is 1 on the left (identity) block, and $n-k$ on the right (redundancy) block. A code reaching this bound is called *maximum-distance separable*, or MDS.

Finally, note that for some codes, there may exist alternative encoders that do not explicitly use a generator matrix.

[*]Consequently, we may take the liberty of equating subtraction with addition in any formula or algorithm. In order to minimize the confusion, we will try to make this systematic. Nonetheless, most of the discussion seamlessly generalises to other (finite) fields.

[†]Note that the we use the convention that vectors are *row* vectors, if not specified otherwise.

Example 1 (AES MixColumn)**.** Let $\boldsymbol{M}$ be the matrix used in the MixColumn operation of the AES block cipher. The code generated by $\begin{pmatrix}\boldsymbol{I}_4 & \boldsymbol{M}\end{pmatrix}$ is an MDS code of paramateres $[8, 4, 5]_{\mathbb{F}_{2^8}}$.

Example 2 (Binary Reed-Muller codes)**.** The binary Reed-Muller code of order r and in m variables $\mathrm{RM}(r, m)$ is the vector-space formed by the multi-point evaluations of m-variate Boolean functions of degree $\leq r$ over $\mathbb{F}_2^m$. In other words, a message is a Boolean function, and its associated (Reed-Muller) codeword is obtained by evaluating it over its entire domain.

The codewords of this code have length 2^m and form a space of dimension $k := \sum_{i=0}^{r} \binom{m}{i}$. It can be shown that the minimum weight of any codeword is 2^{m-r} [MS06, Ch. 13, Thm. 3]. The parameters of $\mathrm{RM}(r, m)$ are thus $[2^m, k, 2^{m-r}]_{\mathbb{F}_2}$.

A (non-systematic) encoding of a message can be efficiently computed by using a fast Möbius transform. Due to its involutive nature, the same transform can also be used to decode a codeword back to a message. However, this does not correct any error and on its own it is thus of rather limited use.

Reed-Muller codes also follow a recursive "$(u, u + v)$" decomposition. One has that $\mathrm{RM}(r + 1, m + 1) = \{u||(u + v), u \in \mathrm{RM}(r + 1, m), v \in \mathrm{RM}(r, m)\}$. This follows from the fact that an $(m + 1)$-variate Boolean function $F(X_1, \ldots, X_{m+1})$ of degree at most $r + 1$ can be written as $F^0(X_1, \ldots, X_m) + X_{m+1}\, F^1(X_1, \ldots, X_m)$, with $\deg(F^0) \leq r + 1$ and $\deg(F^1) \leq r$. Furthermore, if we write F^{0+} the $m + 1$-variate function whose monomials are identical to F^0, F^{1+} for $X^{m+1}\, F^1$, and $\vec{X}_{1,m}$ a given assignement for the indeterminates $X_1, \ldots, X_m$, then we always have:

— $\mathrm{eval}(F^0, (\vec{X}_{1,m})) = \mathrm{eval}(F^{0+}, (\vec{X}_{1,m}, 0)) = \mathrm{eval}(F^{0+}, (\vec{X}_{1,m}, 1))$;

— $\mathrm{eval}(F^1, (\vec{X}_{1,m})) = \mathrm{eval}(F^{1+}, (\vec{X}_{1,m}, 1))$;

— $\mathrm{eval}(F^{1+}, (\vec{X}_{1,m}, 0)) = 0$;

and the decomposition follows. Finally, one may notice that this is essentially the same induction as the one used in the fast Möbius transform algorithm.

Given a code $\mathcal{C}$, it is often important to be able to determine if a vector of its ambient space is a codeword or not. This may be done using a map $\boldsymbol{x} \mapsto \boldsymbol{y}$ s.t. $\boldsymbol{y}$ is "zero" iff. $\boldsymbol{x} \in \mathcal{C}$. One typically implements this with a *parity-check matrix* $\boldsymbol{H} \in \mathbb{F}_2^{n-k \times n}$ which is a basis of the (right) kernel of a generator matrix $\boldsymbol{G}$ of $\mathcal{C}$; the corresponding map, of codomain $\mathbb{F}_2^{n-k}$, is then $\boldsymbol{x} \mapsto \boldsymbol{H} \times \boldsymbol{x}$. Equivalently, $\boldsymbol{H}$ is made of $(n - k)$ linearly-independent vectors of $\mathbb{F}_2^n$ whose scalar product with any element of $\mathcal{C}$ is zero, and $\boldsymbol{H}\boldsymbol{G}^t$ and $\boldsymbol{G}\boldsymbol{H}^t$ are both zero matrices. A parity-check matrix generates the *dual* of $\mathcal{C}$, written $\mathcal{C}^\perp$, which is thence an $[n, n - k]$ code. A code that is its own dual is called *self-dual.*

2 Information set decoding

In this section we focus on the problem of finding "low-weight" codewords of a code, which is also essentially equivalent to finding a close-by codeword to a given vector, i.e. to decode. This is generally a hard problem for codes that do not exhibit any particular structure (for instance if they are defined from a uniformly random generator matrix), as it is NP-hard [**?**], but efficient algorithms may exist for some specific codes. For now we will focus on "inefficient" generic algorithms that work for any code, but we will later present a good *list decoder* for (punctured and shortened) first-order Reed-Muller codes.

Let $\mathcal{C}$ be an $[n, k, d]$ code for which $\boldsymbol{G}$ is a generator matrix. Enumerating all the codewords of $\mathcal{C}$ can trivially be done in time 2^k by multiplying $\boldsymbol{G}$ by all the vectors of $\mathbb{F}_2^k$.

This immediately allows to find a weight-d codeword of $\mathcal{C}$ or to decode to the (or one of the) closest codeword(s), but the cost is quickly prohibitive.

A first remark on the way to find better alternatives is that the problem that one needs to solve usually does not require to enumerate all the codewords of $\mathcal{C}$. For instance, one may not need to find a codeword of *minimum* weight, but finding one of weight less than a known bound may be enough. Similarly, in a decoding context, one may know an upper-bound on the error weight. It is then possible to use a probabilistic algorithm that stops when a "good-enough" solution has been found.

A second remark is that the decoding problem can be solved by finding low-weight codewords; this will be used to justify the fact that we solely focus on algorithms for the latter. This can be explained in the following way: let $\boldsymbol{c} \in \mathcal{C}$ be an initial codeword, and $\hat{\boldsymbol{c}} = \boldsymbol{c} + \boldsymbol{e}$ be a noisy codeword obtained by adding a noise $\boldsymbol{e}$ of weight $w \leq \mathit{max\ weight} < \lfloor (d-1)/2 \rfloor$. Then $\boldsymbol{c}$ is the unique closest codeword to $\hat{\boldsymbol{c}}$, and $\boldsymbol{e}$ is the unique vector of weight w in the affine subspace $\boldsymbol{e} + \mathcal{C}$. Furthermore, this latter vector can be found by searching for a "codeword" of weight w in the code generated by $\begin{pmatrix} \boldsymbol{G} \\ \hat{\boldsymbol{c}} \end{pmatrix}$. This codeword will even be unique, as for any $\boldsymbol{c}' \neq \boldsymbol{c} \in \mathcal{C}$, $wt(\boldsymbol{c}' + \hat{\boldsymbol{c}}) = wt(\boldsymbol{c}' + \boldsymbol{c} + \boldsymbol{e}) \geq wt(\boldsymbol{c}' + \boldsymbol{c}) - wt(e) \geq d - \mathit{max\ weight} > w$.

The first probabilistic alternative to exhaustive search that we present is quite simple [**?**, **?**]. Given $\boldsymbol{G}$, randomly select k linearly-independent columns; this is called an *information set*. Then permute these columns to the first k positions of $\boldsymbol{G}$, and compute the reduced row-echelon form (i.e. compute an alternative generator matrix $\boldsymbol{G}'$ in systematic form, associated to the selected information set). Finally, check if any of the resulting k rows have a weight less than the input bound. The idea behind this algorithm is that any row of the obtained systematic encoder has by definition a very low weight of exactly one on its first k positions, and the weight on the remaining $n - k$ positions depends on a random codeword linear combination. One then hopes that for *some* information sets, the weight on these latter positions we also be small, resulting in an overall low-weight codeword. In other words, the algorithm will return a weight-w codeword after examining a given information set if it is s.t. there is a codeword of weight 1 over the information set and of weight $w - 1$ over its complement, the *redundancy set*.

There is also a simple interpretation of this algorithm if one directly thinks of it in a "decoding" sense. An information set is by definition a set of positions that carries enough information to fully determine the message corresponding to a codeword. Indeed, given the value of a codeword on an information set, one can reconstruct the entire (non-noisy) codeword by simply applying an encoder systematic w.r.t. this set; it is then easy to invert the encoding to go back to the original message. Thus, what the above does is (randomly) trying to find an information set over which the error vector is all-zero.

A variant of the above first algorithm due to Lee and Brickell [**?**] consists, for each information set, in checking the weight of all linear combinations of rows of $\boldsymbol{G}'$ of weight less than a small value p (typically 2 or 3). This somehow amortizes the cost of computing $\boldsymbol{G}'$ by considering more codewords for each matrix, as now the algorithm returns on a given information set if a codeword's weight splits as $(i, w - i), 1 \leq i \leq p$ over itself and its complement. Also, note that for binary codes, computing all of these can be done particularly efficiently by using Gray codes.

Another variant due to Leon aims to reduce the practical cost of checking the weight of a codeword, and is thus mostly useful for long codes. The idea is simply to first check if a codeword generated from the above procedure has a small weight on a few positions (i.e. to first consider a short *punctured* code), and only to look at the entire codeword in that case. For instance, if one requires the punctured codeword to have weight zero on its redundancy set of size l, one is in effect searching for codewords whose weight splits as

$(i, 0, w-i), 1 \leq i \leq p$ over the information set, and the non-punctured (resp. punctured) redundancy positions.

While this approach looks at fewer candidates per information set as the Lee-Brickell algorithm, this is hoped to be counter-balanced by more efficient implementations.

Another algorithm due to Stern still follows the overall same approach, but improves the time complexity at the cost of some memory [**?**]. The key idea is to split the search space into two lists and to exploit collisions to obtain a quadratic speed-up at some stage of the search. Starting from the initial algorithm, one splits the information set into two subsets I_1 and I_2, and forms the lists Λ_1 and Λ_2 of codewords of weight less than p on each subset respectively. Then one only fully checks the weight of codewords formed by the sum of elements of Λ_1 and Λ_2 that are identical over l prescribed positions Z of the redundancy set. One is then searching for codewords whose weight splits as $(i, j, 0, w-i-j), 1 \leq i, j \leq p$ over I_1, I_2, Z and the remainder.

It is essential to notice that for a given information set, checking for each of the $\#\Lambda_1\#\Lambda_2$ candidate codewords if it is of the above form indeed takes a cost linear in $\#\Lambda_{1,2}$ (by using an appropriate data structure).

Finally, one may remark that this algorithm takes more input parameters than the previous ones. This, together with the fact that it is not memory-less may make it harder to determine what parameter choice is best suited to a given code.

An important observation made by Canteaut and Chabaud [**?**] is that the most expensive step in the above algorithms is the computation of the systematic encoder for a given information set. They then suggest that instead of selecting a new independent information set at every iteration, one may "update" the current set by randomly replacing one of its columns by one column of the redundancy set, which is much more efficient. Furthermore, one can easily be convinced that after a few iterations, the obtained information set will be essentially independent from the starting one, hence there is no risk that one gets stuck in a small subset of the search space considered by the other algorithms.

We will conclude by describing how to efficiently update an information set. Let $\boldsymbol{G} = \begin{pmatrix}\boldsymbol{I} & \boldsymbol{A}\end{pmatrix}$ be a systematic generator matrix; our objective is to compute $\boldsymbol{G}' = \begin{pmatrix}\boldsymbol{I} & \boldsymbol{A}'\end{pmatrix}$ which is a generator matrix for the same code and equal to the reduced row-echelon form of a matrix obtained from $\boldsymbol{G}$ by swapping one column $\boldsymbol{I}_{\cdot,i}$ of the identity with one column $\boldsymbol{A}_{\cdot,j}$ of the redundancy matrix. First notice that this latter process only results in a systematic matrix if $\boldsymbol{A}_{\cdot,j}$ is linearly independent from $\boldsymbol{I}\backslash\boldsymbol{I}_{\cdot,i}$, which is equivalent to requiring that $\boldsymbol{A}_{i,j} = 1 \neq 0$. Second, the matrix $\boldsymbol{A}'$ is simply obtained from $\boldsymbol{A}$ by adding the row $\boldsymbol{A}_i$ to every row $\boldsymbol{A}_{i'}$ where $\boldsymbol{A}_{i',j} = 1$. Indeed, this corresponds to the "reduction" step one needs to perform after swapping the above two columns.

3 Learning Parity with Noise cryptosystems

In this section, we introduce the *Learning Parity with Noise* (LPN) problem, and its application to the design of cryptosystems. The LPN problem is rather attractive because of its very concise description. Let $\boldsymbol{s} \in \mathbb{F}_2^k$, $\boldsymbol{a} \xleftarrow{\$} \mathbb{F}_2^k$, $e \leftarrow \mathrm{Ber}_\eta$, where Ber_η is the Bernoulli distribution of parameter η; that is, $\Pr[e = 1] = \eta$. Then the LPN problem is to guess the value of the scalar product $\boldsymbol{s} \cdot \boldsymbol{a}$ when given $(\boldsymbol{a}, \boldsymbol{s} \cdot \boldsymbol{a} + e)$. An algorithm is then said to solve this problem with advantage ε if it answers correctly with probability p and $2|p - 1/2| = \varepsilon$.

It is clear that if $\eta = 1/2$, the distribution of $\boldsymbol{a} \cdot \boldsymbol{s} + e$ is independent of $\boldsymbol{a}$ and $\boldsymbol{s}$, and no algorithm can succeed with a non-zero advantage. Similarly, without prior knowledge about $\boldsymbol{s}$ and except when $\boldsymbol{a} = \boldsymbol{0}$, one cannot hope to solve the problem given a *single* query of the above form, even in the absence of noise (i.e. even when $\eta = 0$). It is then natural to extend the problem to arbitrarily-many queries q, where the unknown (secret) vector $\boldsymbol{s}$ is kept constant. One may then reformulate the problem as letting $\boldsymbol{A} \xleftarrow{\$} \mathbb{F}_2^{k \times q}$,

$\boldsymbol{e} \leftarrow \mathrm{Ber}_{\eta,q}$ (that is, each bit of $\boldsymbol{e} \in \mathbb{F}_2^q$ has independent probability η to be equal to one), and asking to distinguish $(\boldsymbol{A}, \boldsymbol{sA} + \boldsymbol{e})$ from $(\boldsymbol{A}, \boldsymbol{u})$ where $\boldsymbol{u} \xleftarrow{\$} \mathbb{F}_2^q$. An algorithm is then said to (t, q) solve $\mathrm{LPN}_{k,\eta}$ with advantage ε if it makes q queries and has running time t.

It is now again clear that if $\eta = 0$, the problem is trivially solvable as long as $\mathrm{rank}(\boldsymbol{A}) = k$, as it is then enough to identify k linearly-independent columns of $\boldsymbol{A}$ to recover $\boldsymbol{s}$ (which then allows to predict all the other queries with advantage 1). In the more meaningful case where $\eta > 0$, recovering $\boldsymbol{s}$ becomes equivalent to decoding a noisy codeword for some random code of length q (which is a parameter that may be chosen by the solving algorithm). We will discuss this matter in more details in the next section, and focus for now on some LPN-based cryptosystems whose security depends on the computational hardness of this problem.

We first describe LPN-C, which is a family of symmetric encryption schemes defined by Gilbert et al. [?]. While the confidentiality of an LPN-C instance reduces to the hardness of a corresponding LPN problem, the scheme is inherently malleable and must thus be used in conjunction with a MAC.

An LPN-C instance is parameterized by a random code length k, a noise level η, a message length r, and the parameters of an $[m, r, d]$ binary code $\mathcal{C}$, assumed to be efficiently decodable up to w errors and s.t. $\Pr[wt(\boldsymbol{e}) > w : \boldsymbol{e} \leftarrow \mathrm{Ber}_{\eta,m}]$ is small. The scheme works as follows. The sender and the receiver first share a secret random matrix $\boldsymbol{M} \in \mathbb{F}_2^{k\times m}$. Then, to encrypt an r-bit message $\boldsymbol{x}$, the sender draws a vector $\boldsymbol{a} \xleftarrow{\$} \mathbb{F}_2^k$ and $\boldsymbol{e} \leftarrow \mathrm{Ber}_{\eta,m}$, computes $\boldsymbol{y} = \mathcal{C}(\boldsymbol{x}) + \boldsymbol{aM} + \boldsymbol{e}$, and sends $(\boldsymbol{a}, \boldsymbol{y})$ to the receiver. To decrypt, the receiver computes $\hat{\boldsymbol{x}} = \boldsymbol{y} + \boldsymbol{aM} = \boldsymbol{x} + \boldsymbol{e}$, and uses the decoder of $\mathcal{C}$ to recover $\boldsymbol{x}$. The designers of LPN-C proposed some parameters for secure instantiations of LPN-C, but without specifying which code $\mathcal{C}$ to choose. An example is to take $k = 768$, $m = 160$, $\eta = 1/20$, $r = 75$, $d = 25$.

An informal way to argue about the security of this scheme is that if one uses parameters for which the LPN problem is hard to solve, then the (encoded) message $\mathcal{C}(\boldsymbol{x})$ is whitened by a pseudo-random mask $\boldsymbol{aM} + \boldsymbol{e}$ which is hard to distinguish from random, and is thus encrypted by a secure "stream cipher".[‡]

We now turn to an LPN-based public-key cryptosystem due to Alekhnovich [?]. This is a highly impractical design, as it only encrypts a single bit and the decryption of "1" fails with probability 1/2. It is however of theoretical interest, and is rather simple to describe.

This scheme is parameterized by an integer n, from which one derives $m = 2n$, $k = n^{1/2-\epsilon}$, $\eta = k/n$. The public key is a matrix $\boldsymbol{A}' \in \mathbb{F}_2^{n+1\times m}$, generated as $\begin{pmatrix}\boldsymbol{A}\\ \hat{\boldsymbol{c}}\end{pmatrix}$ where $\boldsymbol{A} \xleftarrow{\$} \mathbb{F}_2^{n\times m}$, $\hat{\boldsymbol{c}} = \boldsymbol{xA} + \boldsymbol{e}$, $\boldsymbol{x} \xleftarrow{\$} \mathbb{F}_2^n$, $\boldsymbol{e} \leftarrow \mathrm{Ber}_{\eta,m}$. The private key is the vector $\boldsymbol{e}$. In other words, one defines a random $[2n, n]$ code with generating matrix $\boldsymbol{A}$ and augments it with a low-weight codeword $\boldsymbol{e}$ to the code $\mathcal{C}$ generated by $\boldsymbol{A}'$. One can indeed check that $\boldsymbol{e}$ is in the span of $\boldsymbol{A}'$, as it is equal to $\begin{pmatrix}\boldsymbol{x} & 1\end{pmatrix}\boldsymbol{A}'$.[§] This augmentation is however done in a "hidden" way, as recovering $\boldsymbol{e}$ from $\boldsymbol{A}'$ is a (hard) decoding problem for the code defined by $\boldsymbol{A}$.

To encrypt one bit for the public key $\boldsymbol{A}'$, the sender proceeds as follows. To encrypt the bit 1, it sends a vector $\boldsymbol{\alpha_1} \xleftarrow{\$} \mathbb{F}_2^m$. To encrypt the bit 0, it computes and sends $\boldsymbol{\alpha_0} = \boldsymbol{d} + \boldsymbol{e}'$, where $\boldsymbol{e}' \leftarrow \mathrm{Ber}_{\eta,m}$ and $\boldsymbol{d}$ is a uniformly random element of $\mathcal{C}^\perp$. The receiver computes the decryption of $\boldsymbol{\alpha}$ as $\boldsymbol{\alpha} \cdot \boldsymbol{e}$.

[‡]There is a slight difference between this setting and the actual definition of LPN that we have used: here the *matrix* is secret and the vector $\boldsymbol{a}$ is public. Yet, this simply corresponds to m "single" queries for m independent secrets batched together into $\boldsymbol{M}$.

[§]One may also remark that for the chosen parameters, $\boldsymbol{e} \notin \mathrm{span}(\boldsymbol{A})$ with high probability. In the unlikely event where this would be the case, one can simply choose another vector and try again.

It is quite immediate to see that upon receiving $\boldsymbol{\alpha_1}$, decryption will fail with probability 1/2. It is a bit less obvious that the decryption of $\boldsymbol{\alpha_0}$ is more successful. In that case, the receiver computes $\boldsymbol{\alpha_0} \cdot \boldsymbol{e} = \boldsymbol{d} \cdot \boldsymbol{e} + \boldsymbol{e} \cdot \boldsymbol{e}'$. Because $\boldsymbol{d} \in \mathcal{C}^\perp$, $\boldsymbol{e} \in \mathcal{C}$, this simplifies to $\boldsymbol{e} \cdot \boldsymbol{e}'$. Finally, the probability that $\boldsymbol{e}$ and $\boldsymbol{e}'$ have a non-disjoint support is $\approx (1-\eta)^k$, which is negligible.

We already have informally argued that computing the secret key $\boldsymbol{e}$ from the public key $\boldsymbol{A}'$ reduces to a hard decoding problem. We may now also remark that distinguishing $\boldsymbol{\alpha_0}$ from $\boldsymbol{\alpha_1}$ reduces to an LPN-like problem for a generating matrix of $\mathcal{C}^\perp$. Alekhnovich then showed that the latter is computationally indistinguishable from a random matrix, which allows to conclude the reduction to LPN.

We conclude this overview by describing an LPN-based symmetric authentication protocol named Lapin [**?**]. Strictly speaking, Lapin is based on the Ring-LPN variant of the problem, whose aim is to decrease the communication complexity. We will first describe it in the LPN framework, and will address this difference next.

A challenger and a verifier share two secret vectors $\boldsymbol{s}$, $\boldsymbol{s}' \in \mathbb{F}_2^n$. To authenticate the challenger, the verifiers draws $\boldsymbol{C} \overset{\$}{\leftarrow} \mathbb{F}_2^{n\times n}$ and send it to the former. The challenger then draws $\boldsymbol{R} \overset{\$}{\leftarrow} \mathrm{GL}(n, \mathbb{F}_2)$, $\boldsymbol{e} \leftarrow \mathrm{Ber}_{\eta,n}$, and sends $(\boldsymbol{R}, (\boldsymbol{sC} + \boldsymbol{s}')\boldsymbol{R} + \boldsymbol{e})$. The verifier then recovers $\boldsymbol{e}$ and validates the challenge if it is of weight less than $\mu\eta n$ for some small acceptance threshold μ.

The Ring-LPN variant of the protocol works similarly, but works over rings of the form $\mathbb{F}_2[X]/\langle f \rangle$, for some polynomial f. The challenge matrix $\boldsymbol{C}$ is replaced by a ring element $\pi(c)$, $c \in \mathbb{F}_2^\lambda$, where π is a mapping verifying some conditions; the matrix $\boldsymbol{R}$ is replaced by an invertible element of the ring r; the secrets are now also ring elements s and s', and so is the error e (still drawn from a Bernoulli distribution, when seen as a vector). The message sent by the challenger is then simply $(r, (s\,\pi(c) + s')r + e)$. The advantage of this variant over the matrix version of the protocol is that the ring elements (esp. r) have a much more compact representation of size $\approx n$ than $n \times n$ random matrices, which essentially decreases the communication cost by a factor n. However, the security now depends on the hardness of decoding codes posessing some structure, which may allow for more efficient algorithms.

Finally, a (still informal) way to argue about the security of this protocol is to notice that the challenger's answer can be written (in the original LPN case) as $\boldsymbol{sCR} + \boldsymbol{s}'\boldsymbol{R} + \boldsymbol{r}$. In other words, one is masking the string challenge-and-secret-dependent string $\boldsymbol{sCR}$ by an LPN query $\boldsymbol{s}'\boldsymbol{R} + \boldsymbol{e}$, which is by assumption computationally indistinguishable from random.

4 LPN solving algorithms

It is quite clear that solving an LPN instance can be done by using generic decoding algorithms. Indeed, one may simply try to solve the *search* variant of the LPN problem, which on $\boldsymbol{sA} + \boldsymbol{e}$ tries to recover $\boldsymbol{s}$, i.e. tries to decode a noisy codeword of the code generated by $\boldsymbol{A}$. As solving the search problem allows to solve the distinguishing problem as well, the hardness of LPN is not more than the one of decoding.

One subtlety that we already mentioned is that an LPN-solving algorithm is free to choose the number of oracle queries it wants to use, i.e. the *length* of the code it wishes to decode. Surely this has to be sufficiently large to even make a successful decoding possible (for instance one may try to ensure that there is at least one error-free information set w.h.p.), but there is no similar constraint on the maximum number of queries.

We will now describe an algorithm that specifically exploits the ability to make many queries in an LPN problem to decrease the time complexity for solving the (search) problem (at the cost of a potentially huge increase in memory and query complexity). This algorithm was first described in this context by Blum, Kalai and Wasserman [?], and is traditionally called *BKW*; we heavily base our presentation on the variant described by Bernstein and Lange [?].

We first describe a simple variant of BKW. An intuition behind the algorithm is that given sufficiently many LPN queries $\boldsymbol{s} \cdot \boldsymbol{a} + e$ with a *fixed*, vector $\boldsymbol{a}$, one can efficiently recover one bit of $\boldsymbol{s}$ (viz. $\boldsymbol{s} \cdot \boldsymbol{a}$) using a majority vote among the samples. This requires $\approx c^{-2}$ samples, where $c = 1 - 2\eta$ $(0 < \eta < 1/2)$ is the correlation of the noise. Repeating this process for k linearly independent masks $\boldsymbol{a}$ leads to a full recovery of $\boldsymbol{s}$.

Of course, as the masks $\boldsymbol{a}$ of LPN queries are uniformly random, collecting enough samples for a single one is equivalent to observing a c^{-2}-multi-collision, which requires $2^{k\cdot(c^{-2}-1)/c^{-2}} \approx 2^k$ samples, so this does not really improve on the exhaustive search of the secret. The idea behind BKW is then to artificially create samples of the above form by combining many random ones; this may provide enough samples to apply majority decoding, but each sample is now "noisier", and one must then find a proper tradeoff.

More precisely, BKW proceeds as follows. To solve an instance of LPN with noise level η (i.e. noise correlation $c = |1 - 2\eta|$) and dimension k with q queries making a pool P_0, we start by fixing a block-size parameter b. Then one creates a table T of size 2^b and an updated pool P_1, both initially empty. Next, for each sample $x = (\boldsymbol{a}, v = \boldsymbol{a} \cdot \boldsymbol{s} + e) \in P$, do the following:

1. Call i the integer value corresponding to the last b bits of $\boldsymbol{a}$. If $T[i] = \emptyset$, update it as $T[i] \hookleftarrow (\boldsymbol{a}, v)$.
2. Else, retrieve $(\boldsymbol{a}', v')$ from $T[i]$ and update x as $(\boldsymbol{a}, v) \hookleftarrow (\boldsymbol{a} + \boldsymbol{a}', v + v')$ and store it in P_1.

At the end of this process, and provided that $q \gg 2^b$, there are $q - 2^b$ samples in P_1 which all have their masks $\boldsymbol{a}$ equal to zero on their last b bits. However, the corresponding dot products indeed became noisier, the correlation c having been squared.[¶] One may now clear the table T of all its entries and start this process again, obtaining a pool P_2 of $q - 2^{b+1}$ samples with masks whose $2b$ last bits are zero and noise correlation c^4, etc., up to a pool P_t of $q - t2^b$ samples with masks with tb zeroes and noise correlation c^{2^t}.

The original algorithm chooses t and b s.t. $tb = k - 1$, that is the last pool is made of samples with masks $\boldsymbol{a}$ of the form $(0, \ldots, 0)$, which are useless, and $(1, 0, \ldots, 0)$, whose corresponding v values can be used to find the first bit of $\boldsymbol{s}$ by a majority vote. This latter step will succeed w.h.p. if $\#P_t \approx c^{-2^{t+1}}$. Finally, the remaining bits of $\boldsymbol{s}$ are iteratively retrieved using the same process.

We now introduce a first optimization to the above algorithm, due to Levieil and Fouque [?]. Let l be a second "block size" parameter, corresponding to an exhaustive search step. The goal is to guess $\boldsymbol{s}$ by blocks of l bits instead of blocks of 1; the first iteration of BKW is then stopped at $tb = k - l$. At that point, one has $N := q - t2^b$ noisy dot products with masks $\boldsymbol{a}_i$ whose l first bits only may be non-zero.[‖] Let $\boldsymbol{y} \in \mathbb{F}_2^k$ be a vector whose $k - l$ last bits are zero; the idea is to notice that if $\boldsymbol{y}$ and $\boldsymbol{s}$ are equal on their first l bits, then $\boldsymbol{a}_i \cdot \boldsymbol{y} = \boldsymbol{a}_i \cdot \boldsymbol{s}$. Consequently, for every sample $(\boldsymbol{a}_i, v_i)$, one has $\boldsymbol{a}_i \cdot \boldsymbol{y} + v_i = e_i$ (where e_i is an aggregated error of some correlation c' coming from the pool creation) which is then one with non-uniform probability $(1 - c')/2$. On the other hand, if $\boldsymbol{y}$ and $\boldsymbol{s}$ disagree on their last bits, $\boldsymbol{a}_i \cdot \boldsymbol{y} + v_i = \boldsymbol{a}_i \cdot (\boldsymbol{y} + \boldsymbol{s}) + e_i$ for a non-zero term $\boldsymbol{y} + \boldsymbol{s}$, and from the uniformity of $\boldsymbol{a}_i$ on its first l bits, this expression is one with

[¶]One can check that the probability of error is $2\eta(1-\eta) = 2(\eta - \eta^2)$, giving a correlation $1 - 4(\eta - \eta^2) = (1 - 2\eta)^2$.

[‖]Note that as $N \gg l$, *non-noisy* approximations would indeed allow to uniquely recover the first l bits of $\boldsymbol{s}$.

probability 1/2 exactly. If $N \approx c'^{-2}$, it is thus possible to distinguish the value of $\boldsymbol{y}$ that matches $\boldsymbol{s}$ to recover its last l bits.

The full process for one block of size l is as follows:

1. For all 2^l candidates $\boldsymbol{y_i}$ for the first l bits of $\boldsymbol{s}$, compute $\hat{c}_{\boldsymbol{y}_i} = \sum_{j=0}^{N}(-1)^{\boldsymbol{a}_j \cdot \boldsymbol{y}_i + v_j}$ (where the sum is over $\mathbb{Z}$).
2. Return the $\boldsymbol{y}_i$ for which $|\hat{c}_{\boldsymbol{y}_i}|$ is largest.

A key algorithmic observation is that, the above is similar to the computation of the Hadamard-Walsh spectrum of the Boolean function $i \mapsto v_i$, for which fast algorithms exist. One can indeed remark that for $\boldsymbol{y}$ and $\boldsymbol{y}'$ that differ only on a single bit, on position l (w.l.o.g.), if we write $\boldsymbol{y}''$ the common part of the two vectors of length $l-1$ and $\boldsymbol{a}_j^1$ (resp. $\boldsymbol{a}_j^0$) the masks whose l^{th} bit is 1 (resp. 0); then we have the following; let

$$A := \sum_{\boldsymbol{a}_j^1}(-1)^{\boldsymbol{a}_j \cdot \boldsymbol{y} + v_j}, \qquad B := \sum_{\boldsymbol{a}_j^0}(-1)^{\boldsymbol{a}_j \cdot \boldsymbol{y}'' + v_j}, \qquad -A = \sum_{\boldsymbol{a}_j^1}(-1)^{\boldsymbol{a}_j \cdot \boldsymbol{y}' + v_j},$$

and $\hat{c}_{\boldsymbol{y}} = A + B$, $\hat{c}_{\boldsymbol{y}'} = -A + B$, and this equality can be applied l times recursively.

The above idea can be further optimized by first applying a transformation on the LPN samples in order to reduce the problem to finding an equivalent *low weight* secret. This idea is due to Kirchner [?] and works as follows. Denote $\boldsymbol{A}$ the full matrix of the masks of q queries to an LPN oracle of dimension k. Let $\boldsymbol{A}_1$ be an information set of k linearly-independent columns of $\boldsymbol{A}$, and let $\boldsymbol{A}_2$ denote k columns of $\boldsymbol{A}$ not in $\boldsymbol{A}_1$. The key observation is that the sum $(\boldsymbol{s}\boldsymbol{A}_1 + \boldsymbol{e}_1)\boldsymbol{A}_1^{-1}\boldsymbol{A}_2 + \boldsymbol{s}\boldsymbol{A}_2 + \boldsymbol{e}_2$ simplifies to $\boldsymbol{e}_1\boldsymbol{A}_1^{-1} + \boldsymbol{e}_2$. In other words, one can transform the $2k$ queries w.r.t. masks $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ for the secret $\boldsymbol{s}$ into k queries for the "secret" $\boldsymbol{e}_1$, whose expected Hamming weight is given by the noise level η, and is then strictly less than $k/2$. The same transformation can be applied many times by changing $\boldsymbol{A}_2$, and one can then run a BKW algorithm to retrieve $\boldsymbol{e}_1$ (which obviously immediately leads to $\boldsymbol{s}$). There is no particular advantage in using this transformation when applying the original BKW algorithm, but when one guesses the secret by block, it becomes enough to guess secrets of low weight, which reduces the search space. Finally, note that the joint computation of the many correlations $\hat{c}_{\boldsymbol{y}}$ can still be done efficiently for those sparse secrets.

We conclude this section by mentioning that even prior the publication of the BKW algorithm, Bleichenbacher described a similar algorithm to exploit biased DSA signatures [?, ?]. Recall that in a DSA (or Schnorr) signature, the signer provides a pair $(c, r + cx)$, where x is a secret exponent, c is random and depends on the message to be signed, and r is a random mask. If r is uniform, the secret cx is blinded by a one-time-pad and nothing can be learned. But if r is biased, then one obtains a problem of noisy decoding similar to LPN. Bleichenbacher's algorithm to retrieve x from many signatures consists in combining samples to zero some of their bits, and to apply a fast Fourier transform to recover, say, 40 bits of the secret. The original bias exploited by Bleichenbacher was a small modulo bias: instead of taking r uniformly in $\mathcal{S}$, $\#\mathcal{S} \approx 2^{160}$, it was uniform over $[0, \ldots, 2^{160} - 1]$ and then reduced modulo $\#\mathcal{S}$. Finally, we remark that similarly to information-set decoding style algorithms, one can also recover a DSA secret using fewer sample signatures with biased masks by finding short vectors in a Euclidean lattice [?].

5 The Goldreich-Levin theorem

In this section, we present the application of list decoding to a proof of existence of *hard-core* predicates for one-way functions, due to Goldreich and Levin [?]. Informally, the objective is to show that if $F : \{0,1\}^n \rightarrow \{0,1\}^m$ is a one-way function, in the sense that it is hard to find a preimage x given $F(x)$, then it is also hard to predict (with probability significantly away from 1/2) the value $\boldsymbol{a}\cdot\boldsymbol{x}$, for any $\boldsymbol{a} \in \mathbb{F}_2^n$ (where $\boldsymbol{x} \in \mathbb{F}_2^n$ is the canonical embedding of $x \in \{0,1\}^n$). A possible proof is to show that if one is given a prediction oracle for (several) $\boldsymbol{a}\cdot\boldsymbol{x}$ with correlation c, then one can reconstruct x w.h.p. with time and memory complexity c^{-2}. Thus, if "it costs at least T to invert F", one has that it is impossible to predict the value of any of the above predicates with correlation ($\approx$ advantage) better than $1/\sqrt{T}$.

One first subtlelty that deserves to be mentioned is that because of the nature of the result we want to prove, the predicate oracle that we will use to invert F can only be called *once* for a given mask $\boldsymbol{a}$: indeed, it makes no sense to define different predictions $\boldsymbol{a}\cdot\boldsymbol{x}$ several times as $\boldsymbol{x}$ itself is fixed. On the other hand, the mask $\boldsymbol{a}$ can be chosen freely. This is to be contrasted with, say, an LPN setting, where one is given $\boldsymbol{a}\cdot\boldsymbol{s}$ for random masks $\boldsymbol{a}$, that may still be potentially equal, in case of unlikely collisions. Yet in either case the effect is the same: it is (by definition or computationally) feasible to recover $\boldsymbol{x}$ by recovering enough biased predictions or samples with the same n-linearly independent masks. Finally, an accurate modelisation of the oracle's power in terms of codes is to say that if $\boldsymbol{A} \in \mathbb{F}_2^{n\times q}$ has q pairwise distinct columns, querying a prediction oracle with q masks $\boldsymbol{A}$ and advantage ε (meaning that a prediction is correct with probability $p > 1/2$; $2p - 1 = \varepsilon$) is equivalent to obtaining a noisy codeword $\boldsymbol{xA} + \boldsymbol{e}$ with $\boldsymbol{e} \leftarrow \mathrm{Ber}_\eta, q$, $\eta = 1/2 - \varepsilon/2$.

A second essential remark is that F and $F(x)$ are both known to the adversary. Thus, as soon as one knows a "small" list L that contains x w.h.p., one can recover x uniquely by mapping F to L and comparing the result to $F(x)$ (here we assume that L does not contain collisions for F, which is true w.h.p. if it is small).

Putting the two remarks together, what we need is an efficient *list-decoding* algorithm for a code generated by a matrix $\boldsymbol{A}$ with pairwise-distinct columns. In the remainder, we will give exactly such an algorithm for a punctured first-order Reed-Muller code.

Let $\boldsymbol{V} \in \mathbb{F}_2^{n\times(r+1)}$ be a projection matrix of rank $r+1$, where $2^r \approx (1-2\eta)^{-2} = \varepsilon^{-2}$ is the number of samples to distinguish the uniform distribution from Ber_η w.h.p., and whose first column is a unit vector $\boldsymbol{b}_i$ (corresponding to the bit x_i that one wishes to recover); let $\boldsymbol{y} = \boldsymbol{xV} \in \mathbb{F}_2^{(r+1)}$ be the projected message; let $\boldsymbol{W} \in \mathbb{F}_2^{(r+1)\times 2^r}$ be the matrix whose first row is all one and whose $r \times 2^r$ lower block is made of all the vectors of $\mathbb{F}_2^r$; let $\boldsymbol{z} = \boldsymbol{yW}$, and remark that if $\boldsymbol{y}$ is seen as a degree-1 Boolean function in r variables, then $\boldsymbol{z}$ corresponds to the first-order Reed-Muller $\mathrm{RM}(1,r)$ encoding of $\boldsymbol{y}$. Now the problem is: given a noisy codeword $\hat{\boldsymbol{z}} = \boldsymbol{z} + \boldsymbol{e}$, recover x_i. If the noise level is low, one can simply use an efficient decoding algorithm for a Reed-Muller code. However, if η is close to 1/2, the expected weight of $\boldsymbol{e}$ is $2^r\eta \approx 2^{r-1}$ which corresponds to the minimum distance of $\mathrm{RM}(1,r)$, and unique decoding is not possible anymore. An observation that may not seem extremely useful at first is that despite a high-level of noise, one can obtain a list of 2^r possible values for x_i in the following way: 1) guess the value of $\boldsymbol{y}[1,\ldots,r]$ and build a matching vector $\boldsymbol{y}'$, with $\boldsymbol{y}'[0] = 0$; 2) compute $\boldsymbol{z}' = \hat{\boldsymbol{z}} + \boldsymbol{y}'\boldsymbol{W}$; 3) if $wt(\boldsymbol{z}') > 2^{r-1}$, guess $x_i = 1$, else guess $x_i = 0$. Now the crucial point is that this process can be in fact jointly applied to all the bits of x by using the *same* projection matrix $\boldsymbol{V}$, up to its first column. By doing so, the guess for the value of the last r bits of $\boldsymbol{y}$ can be reused for every bit of x to obtain a list of 2^r consistent predictions for x in its entirety.

References

[MS06] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The Theory of Error-Correcting Codes*. North-Holland Mathematical Library. North-Holland, 12 edition, 2006.