Implementation of Coppersmith's Index Calculus

Letizia D'Achille

Final individual report of the team project for the exam
Algebraic Cryptography: mod. 2 - Finite Fields and Symmetric Cryptography

June 21, 2023

Mathematical overview

The aim of the Index Calculus algorithm proposed by Coppersmith is to solve the Discrete Logarithm Problem over $\mathbb{F}_{2^n}$, i.e., given $a \in \mathbb{F}_{2^n}$ primitive element, and $b \in \mathbb{F}_{2^n}$, find the unique c (modulo $2^n - 1$) such that $b = a^c$. The specific algorithm moves the problem to the representation of $\mathbb{F}_{2^n}$ as ${}^{\mathbb{F}_2[x]}/_{(P(x))}$ where $P(x)$ is an irreducible polynomial over $\mathbb{F}_2[x]$. In particular we take $P(x)$ primitive with root a, so that we can make the computations using as primitive element $x \in {}^{\mathbb{F}_2[x]}/_{(P(x))}$, exploiting the isomorphism:

$$\mathbb{F}_{2^n} \cong \mathbb{F}_2(a) \cong {}^{\mathbb{F}_2[x]}/_{(P(x))}.$$

Every element in $\mathbb{F}_{2^n}$ can then be represented as a polynomial in $\mathbb{F}_2[x]$ of degree less or equal than $\deg(P)$. If we call $G(x)$ the image of b through the isomorphism, the problem to solve is then finding c such that

$$G(x) = x^c \mod P(x).$$

The algorithm is divided in two phases:

- **Precomputation phase**: fixed a bound bd, we create a database containing the logarithms of all the polynomials of degree at most bd;
- **Computation phase**: given a polynomial in $\mathbb{F}_2[x]$ we compute its logarithm by rewriting it as a polynomial function of elements in the previously defined database.

Both of them involve numerous parameters (for now, we have seen bd and $P(x)$) whose choice will be properly investigated in the subsection *Choice of the parameters*. A careful selection is important for the optimization of the algorithm. In particular we choose

$$bd \geq c_1 n^{\frac{1}{3}} (\log n)^{\frac{2}{3}}, \qquad d \sim bd, \qquad k = 2^s \sim \sqrt{\frac{n}{d}}, \qquad h = \left\lceil \frac{n}{k} \right\rceil$$

for c_1 small constant, and we consider the polynomial $Q(x)$ such that $P(x) = x^n + Q(x)$.

Within this context, we must notice that changing the presentation of the field from ${}^{\mathbb{F}_2[x]}/_{(P(x))}$ to ${}^{\mathbb{F}_2[x]}/_{(P'(x))}$ requires special care. Indeed, we might want to construct the database using one specific representation ${}^{\mathbb{F}_2[x]}/_{(P'(x))}$, and then we might be given a primitive element $a \in \mathbb{F}_{2^n}$ that is a root of $P(x)$, but not of the polynomial $P'(x)$ used to construct the field. In this case, we will have to write a, b in the first representation using the isomorphisms

$$\begin{array}{ccccccc} \mathbb{F}_2(a) & \xrightarrow{\sim} & {}^{\mathbb{F}_2[x]}/_{(P(x))} & \xrightarrow{\sim} & {}^{\mathbb{F}_2[x]}/_{(P'(x))} & \xrightarrow{\sim} & \mathbb{F}_2(\alpha) \\ a & \mapsto & x & \mapsto & h(x) & \mapsto & h(\alpha) \\ b & \mapsto & G(x) & \mapsto & G(h(x)) & \mapsto & G(h(\alpha)) \end{array}$$

where α is a root of $P'(x)$ in $\mathbb{F}_{2^n}$ and $h(x)$ is a root of $P(x)$ in ${}^{\mathbb{F}_2[x]}/_{(P'(x))}$.
If we now find c_1, c_2 such that

$$\begin{cases} h(x) = x^{c_1} & \mod P'(x) \\ G(h(x)) = x^{c_2} & \mod P'(x) \end{cases}$$

by solving the DLP twice in ${}^{\mathbb{F}_2[x]}/_{(P'(x))}$, we will have

$$G(h(x)) = h(x)^{-c_1 c_2} \mod P'(x)$$

and looking at the isomorphism we have that $c = -c_1 c_2 \mod 2^n - 1$ solves the original problem.

Precomputation phase

The procedure is divided into two parts:

1. creating a database of linear equations relating the logarithms of all the polynomials of degree at most bd;
2. solving the system of linear equations in order to retrieve the unknown logarithms.

Concerning part 1, let us now outline the method for calculating a linear equation of the type described above. We select a pair of polynomials $(A(x), B(x))$ among those of degree at most d such that $\gcd(A(x), B(x)) = 1$. We then set $C(x) = x^h A(x) + B(x)$ and $D(x) = C(x)^k \mod P(x)$ and check whether they are bd-smooth, i.e, whether they are factorizable in a product of irreducible polynomial of degree at most bd. If this is the case, we write

$$C(x) = \prod_j q_j(x)^{c_j} \qquad D(x) = \prod_j q_j(x)^{d_j}$$

where some of the exponents can possibly be equal to 0, so that we write $C(x)$ and $D(x)$ as product of the same irreducible factors. Exploiting the relation $D(x) = C(x)^k \mod P(x)$ we can derive, passing to the logarithms, the following linear equation:

$$\sum_j d_j \log q_j = \sum_j k c_j \log q_j \mod 2^n - 1$$

or, equivalently,

$$\sum_j (kc_j - d_j) \log q_j = 0 \mod 2^n - 1$$

where $\log q_j$ are the unknowns, and q_js are polynomials of degree at most bd. On the other hand, if $C(x)$ and $D(x)$ are not bd-smooth, the pair $(A(x), B(x))$ is discarded.

Remark. The smoothness testing exploits the fact that over $\mathbb{F}_2$ the polynomial $x^{2^i} - x$ is the product of all irreducible polynomials of degree dividing i. Suppose we want to verify whether $C(x)$ is bd-smooth. One way consists in computing $C'(x) \prod_{i=1}^{bd} (x^{2^i} - x) \mod C(x)$ and checking whether it is 0. Indeed, the derivative $C'(x)$ contains all the repeated factors in $C(x)$, and these appear with multiplicity greater or equal than the multiplicity they have in $C(x)$ diminished by one. When the result is not 0, $C(x)$ can't be bd-smooth. When the result is 0, $C(x)$ still may not be smooth, due to a multiple factor of large degree, but it is sufficient to check this eventuality. In order to avoid the final check, one could simply compute $H(x) = \prod_{i=1}^{bd} (x^{2^i} - x) \mod C(x)$. If we denote with c the degree of $C(x)$, we can then compute $H(x)^c \mod C(x)$ and check whether the result is 0. In this case $C(x)$ will be bd-smooth if and only if it is 0.

Remark. Recalling that $h = \lceil \frac{n}{k} \rceil$ and $P(x) = x^n + Q(x)$, we can define

$$R(x) = x^{hk} = x^{n+(hk-n)} = Q(x) x^{hk-n} \mod P(x)$$

$D(x)$ can then be rewritten as:

$$\begin{aligned} D(x) = C(x)^k = C(x)^{2^s} = & \mod P(x) \\ = (x^h A(x) + B(x))^{2^s} = & \mod P(x)^1 \\ = x^{h2^s} A(x)^{2^s} + B(x)^{2^s} & \mod P(x) \\ = R(x) A(x)^k + B(x)^k & \mod P(x) \end{aligned} \tag{1}$$

Requiring the degree of $Q(x)$ to be small, we have that the degree of $R(x)$ is small as well, and $\deg D(x) \leq \deg(R(x)) + kd \ll n$. This means that the last line of Equation 1 corresponds to the actual writing of $D(x)$ as its representative in $\mathbb{F}_2[x]/(P(x))$.

Remark. The condition $\gcd(A(x), B(x)) = 1$ is required to avoid redundant equations. Indeed, suppose $\gcd(A(x), B(x)) = S(x) \neq 1$, i.e, $A(x) = S(x)A'(x)$ and $B(x) = S(x)B'(x)$ with $\gcd(A'(x), B'(x)) = 1$. We have that

$$\begin{aligned} C(x) &= x^h A(x) + B(x) = S(x)(x^h A'(x) + B'(x)) \\ D(x) &= R(x) A(x)^k + B(x)^k = S(x)^k (R(x) A'(x)^k + B'(x)^k) \end{aligned}$$

where we used Equation 1. $S(x)^k$ is a non-trivial factor of both $C(x)^k$ and $D(x)$, making the linear equation derived from the pair $(A(x), B(x))$ equal to the one associated to $(A'(x), B'(x))$. Indeed, the contribution of factors of $S(x)^k$ cancels out.

[1] We used the fact that we chose k as a power of 2, so that we can use *Freshman's Dream*.

Heuristically, the probability of finding a pair $(A(x), B(x))$ such that $C(x)$ and $D(x)$ are bd-smooth can be computed by assuming that they are independent random polynomials of degree $\sqrt{nd}$. The estimation on the degree is derived from what follows

$$\deg C(x) = \deg(x^h A(x) + B(x)) \leq h + d = \left\lceil \frac{n}{k} \right\rceil + d = \left\lceil n\sqrt{\frac{d}{n}} \right\rceil + d$$

$$\deg D(x) = \deg(R(x)A(x)^k + B(x)^k) \leq \deg(R(x)) + kd = \deg(R(x)) + \sqrt{\frac{n}{d}}d$$

recalling that $A(x)$ and $B(x)$ are polynomials of degree at most d. Removing the small additive constants both the results can be approximated to the aforementioned bound. Since the probability that a polynomial over $\mathbb{F}_2$ of degree m is bd-smooth is $\left(\frac{m}{bd}\right)^{-\frac{m}{bd}}$ (see [3]), then the probability of finding a pair with the desired properties is

$$\sim \left(\frac{\sqrt{nd}}{bd}\right)^{-2\frac{\sqrt{nd}}{bd}} \sim e^{-\sqrt{\frac{n}{bd}}\log\frac{n}{bd}} \sim e^{-\frac{1}{\sqrt{c_1}}\left(\frac{n}{\log n}\right)^{\frac{1}{3}}\log\frac{1}{c_1}\left(\frac{n}{\log n}\right)^{\frac{2}{3}}} \sim e^{c_2 n^{\frac{1}{3}}(\log n)^{\frac{2}{3}}}$$

for some constant c_2, by recalling $bd \geq c_1 n^{\frac{1}{3}}(\log n)^{\frac{2}{3}}, d \sim bd$. The number of irreducible polynomials of degree at most bd can be approximated with $\frac{2^{bd+1}}{bd}$, therefore we must gather at least this number of linear equations, for a total computational cost of

$$O(e^{c_3 n^{\frac{1}{3}}(\log n)^{\frac{2}{3}}})$$

for another constant c_3, since the new multiplicative term is irrelevant asymptotically.

Remark. The sieve on the pairs of polynomials can be carried out by fixing the polynomial $A(x)$ first, and then searching among all the possible pairs $(A(x), B(x))$ as $B(x)$ varies. This can be done by following a sieve described in [2, 4]. The idea is, for each possible $B(x)$ of degree less than d, we compute an entry of a database where there will be the sum of all $\deg g$ for g that ranges among the irreducibles of degree up to bd and for which it is satisfied

$$x^h A(x) + B(x) = 0 \mod g$$

Note that if this is done for each possible irreducible polynomial g and its powers, the entry of the database will contain the degree of the smooth part of $C(x) = x^h A(x) + B(x)$. Then, it would be sufficient to check whether this value coincides with $\deg C(x)$ to accept or reject the correspondent pair.
In practice, the polynomial sieve can be carried out by fixing g and then ranging over all the polynomials $B(x)$ in the equivalence class of $x^h A(x) \mod g$. Indeed, if we define

$$B_0 = x^h A(x) \mod g, \qquad B_i = B_{i-1} + x^{\texttt{pos(i)}} g$$

where `pos(i)` is the index of the least significant bit set in the binary representation of i, we have that these polynomials are exactly the polynomials of degree up to d in the desired equivalence class.

Let us now consider part 2 of the precomputation phase, i.e. the resolution of the system of linear equation. This can be solved using the known methods, but we must notice this is a sparse system. Using specific methods that take advantage of the sparsity of the resulting matrix, the computational cost of this step can be lowered from $O(B^3)$ to $O(B^2)$, where $B \sim \frac{2^{bd+1}}{bd}$ is the number of unknowns.

Computation phase

Given $G(x) \in {}^{\mathbb{F}_2[x]}/_{(P(x))}$ whose discrete logarithm is to be found, the core of the computation phase consists in writing its logatithm as an expression of logarithms of polynomials of degree at most bd, so that we can eventually make use the previously computed database. This is achieved by obtaining relations with respect to polynomials of successively lower degrees, until the desired degree is reached.
We initially note that, given $m \in \mathbb{Z}_{2^n-1}$, we can compute $G'(x) = G(x)x^m \mod P(x)$ and apply the Extended Euclidean Algorithm to the pair $(P(x), G'(x))$ to find a relation of the type

$$P(x)S(x) + G'(x)T(x) = R(x) \quad \equiv \quad (G(x)x^m)T(x) = R(x) \mod P(x) \tag{2}$$

with $T(x), R(x)$ of degree about $\frac{n}{2}$.

Remark. For an efficient implementation of the Extended Euclidean Algorithm, some of its mathematical properties are worth mentioning. Note that in our case we want to apply the Extended Euclidean Algorithm to a pair $(P(x), G'(x))$ where $\deg P > \deg G'$ ($G(x)x^m$ is taken modulo $P(x)$) and $P(x)$ is irreducible. Therefore we have that $\gcd(P(x), G'(x)) = 1$, which will be returned as the final result of the complete execution of the algorithm.
Suppose now that

$$P(x)S_i(x) + G'(x)T_i(x) = R_i(x)$$

is the relation obtained at the i-th step of the algorithm where $R_{i-2} = Q_iR_{i-1} + R_i$ with $\deg R_i < \deg Q_i$, and we use the relations

$$T_i = T_{i-2} - Q_iT_{i-1}, \quad S_i = S_{i-2} - Q_iS_{i-1}$$

Using induction on i, we can show that under our assumptions the following properties hold:

$$\deg Q_i = \deg R_{i-2} - \deg R_{i-1} > 0 \qquad \deg T_i = \sum_{j=0}^{i} \deg Q_j = \deg P - \deg R_{i-1}$$

In particular, since the degree of R_i is strictly decreasing, we have that the degree of T_i is strictly increasing. Moreover, these degrees are balanced with respect to the degree of P, which adds value to the requirement $\deg T, \deg R \sim \frac{n}{2}$.

In practice, we extract a random m and make this computation until the resulting polynomials $T(x), R(x)$ are bd_1-smooth, where $bd_1 = \sqrt{n\,bd} > bd$. Then, we factorize $T(x), R(x)$ as

$$T(x) = \prod_j t_j(x)^{e_j} \qquad R(x) = \prod_j r_j(x)^{f_j}$$

where t_j and r_j have degree at most bd_1, and we can proceed with the computation on these factors. Note that, if the logarithms of those factors were known, we could compute the desired logarithm. Indeed, from Equation 2 it follows

$$\log G(x) + m + \sum_j e_j \log(t_j(x)) = \sum_j f_j \log(r_j(x)) \mod 2^n - 1$$

$$\log G(x) = \sum_j f_j \log(r_j(x)) - \sum_j e_j \log(t_j(x)) - m \mod 2^n - 1$$

If the desired conditions on $C(x), D(x)$ do not hold, the chosen m is instead discarded.

Let it be now $G'(x)$ one of the found factors. If its degree is less that bd, one can immediately find its logarithm by looking at the precomputed database. Otherwise, we try to further lower the degree by finding relations involving logarithms of polynomials of successively lower degree.
To this end, one can use a similar procedure of the one used in the precomputation phase. Suppose that we are given a polynomial $G'(x)$ of low degree g. We fix the temporary parameters

$$bd' = \left\lfloor \sqrt{bd\,g} \right\rfloor, \qquad d' \sim \frac{g + \sqrt{\frac{n}{bd}} \log_2 \left(\frac{n}{bd}\right)}{2}, \qquad k' = 2^{s'} \sim \sqrt{\frac{n}{d'}}, \qquad h' = \left\lceil \frac{n}{k'} \right\rceil$$

As before, we select a pair of polynomials $(A(x), B(x))$ among those of degree at most d' such that $\gcd(A(x), B(x)) = 1$. We then set $C(x) = x^{h'} A(x) + B(x)$ and $D(x) = C(x)^{k'} \mod P(x)$, and check whether the additional condition $G'(x) | C(x)$ holds. If so, we can check whether $\frac{C(x)}{G'(x)}, D(x)$ are bd'-smooth. In this case, we can write

$$\frac{C(x)}{G'(x)} = \prod_j q_j(x)^{e_j} \qquad D(x) = \prod_j d_j(x)^{f_j}$$

and exploiting the equation $D(x) = C(x)^{k'} = \left(\frac{C(x)}{G'(x)}\right)^{k'} G'(x)^{k'} \mod P(x)$ we can retrieve the relation

$$\sum_j f_j \log d_j = k' \sum_j e_j \log q_j + k' \log G'(x) \mod 2^n - 1$$

$$\log G'(x) = \frac{\sum_j f_j \log d_j}{k'} - \sum_j e_j \log q_j \mod 2^n - 1$$

Remark. Note that we can use the polynomial sieve described for the precomputation phase also in this part of the procedure. Since we do not have to find many possible pairs $(A(x), B(x))$, we simply use $G(x)$ as g and sieve over the equivalence class of $x^{h'}A(x)$ mod $G(x)$ using the previously described method. We will find $C(x)$ bd' -smooth, hence it follows that also $\frac{C(x)}{G'(x)}$ will have the same property.

Thereby, we can express the desired logarithms as functions of logarithms of polynomials of smaller degrees (note that $bd' = \lceil \sqrt{bd\, g} \rceil < g$ if $g > bd$), and we can iterate this procedure until all the unknown logarithms have been expressed as a relation of the known logarithms of the polynomial of degree at most bd.
The preceding probabilistic considerations on the single step still apply unchanged. Moreover, this procedure produces less than n irreducible polynomials for each factor. Note that after t steps the required bound will be less than

$$bd_t = (bd\, bd_{t-1})^{\frac{1}{2}} = \cdots = (bd(\ldots(bd\, n)^{\frac{1}{2}}\ldots)^{\frac{1}{2}})^{\frac{1}{2}} = bd^{\sum_{j=1}^{t}\left(\frac{1}{2}\right)^j}\, n^{2^{-t}} = bd(bd^{-1}n)^{2^{-t}}$$

and after $t = \log n$ steps all the polynomials have degree less than bd, therefore the procedure terminates.

Note that we could equivalently use the initial strategy of applying the Extended Euclidean Algorithm to the pair $(P(x), G'(x)x^m)$ for random m, and proceed as outlined above using directly the bound bd. The drawback is that in this case the strategy of lowering the degrees cannot be used and the degree of the factors the resulting polynomials is arbitrary, therefore it may be unuseful if the set bound bd is too low.

Choice of the parameters

The bound bd is chosen is such a way to balance the computational costs of the various phases. An higher bound would be better when considering the computation phase and part 1 of the precomputation, but would be worse when it comes to solving the linear system. Moreover, in the computation phase the bound bd' has to be lower enough not to make the descent to the bound bd too slow, but it must be higher enough to find a pair (A, B) quickly.
The parameters d, d' have to be higher enough to produce the required number of relations in both the phases. On the other hand, as they increase, the probability that $C(x), D(x)$ are smooth (or $\frac{C(x)}{G(x)}, D(x)$ in the computation phase) diminishes, since the degree of these polynomials increases as well.
Regarding the choice of $P(x) = x^n + Q(x)$, we have already remarked that in Equation 1 the degree of $R(x)$ is small if it is that of $Q(x)$. However, it may be of use choosing $Q(x)$ with small factors (see [2, 4]), since this property lowers the probability of smoothness in the distinct steps of the procedure.

Implementation's details

All the computational experiments have been carried out using Magma V2.25-3 on the operating system Windows 11 22H2 with processor 3.20 GHz AMD Ryzen 7 5800H (8 cores, 16 threads) and 16GB RAM 3200 MHz SODIMM. Times were recorded for 100 random test vectors for each value of n ($n = 73, 100$) using the Magma function `time`.

The bound bd was chosen in agreement with the other group, and fixed in advance at $bd = 17$ for both values of n. Indeed, as remarked, an higher bound is only a downside in the precomputation phase, that was not evaluated. In this way it was not possible to produce databases of logarithms of arbitrary size, but these were fixed a priori.
However the other parameters were not fixed, therefore we were free to choose the polynomial $P(x)$. Since we were given elements in an previously fixed representation, it is convenient not to change it if we do not consider the already mentioned downsides. Here we present the code used to change the representation of the field, that is not used in the final version of the project:

```
F := Parent(a);
Q :=  x^73 + x^8 + x^6 + x^2 + 1;
K<alpha> := ext<GF(2)|Q>;
a1 := Roots(P,K)[1][1];
phi := hom < F -> K | a1 >;
log := ComputeLog(R!Eltseq(phi(b)),n,h,Q,b73_1);
return 2814252823474769967350 * log;
```

In this example the code converts a, b from the original representation with $P(x) = x^{73}+x^4+x^3+x^2+1$ to the one with $Q(x) = x^{73}+x^8+x^6+x^2+1$. We chose to implement the isomorphism using the suitable constructor `hom`, and applying the morphism to the element b. Another possibility was to compute the polynomial expression of `a1` in the field `K`, and then use it for the computation of the image of b as showed in the theory. However, the first solution resulted experimentally more convenient. The integer in the last line corresponds to the inverse modulo $2^n - 1$ of the logarithm of `a1` in the field `K`, computed using the command

```
Modinv(ComputeLog(R!Eltseq(a1),bound,n,h,k,Q), 2^n-1)
```

Note that, once one has chosen the desidered representation, this value can be precomputed as the given generator a is fixed.
With regard to the other parameters, $d = 14$ was chosen experimentally (the timings are not reported since the tests on the precomputation take hours to run), $k = 4$ followed from the formula and references from papers (see [1, 2]).

Precomputation phase

We chose to implement the polynomial sieve instead of a straightforward search among random pairs $(A(x), B(x))$ because it produces multiple pairs at a time and resulted overall faster. Its implementation was simply a transcription of the method described

above, whose pseudo-code can be found in [4]. The function `pos` was implemented by dividing by all the possible powers of 2 starting from the smallest one, as follows:

```
function pos(i)
    e := 1;
    while (i mod (2^e)) eq 0 do
        e := e + 1;
    end while;
    return e-1;
end function;
```

Let us now consider the function implementing the testing for smoothness:

```
function SmoothTest(C,b)
    l := Floor(b/2);
    P := Modexp(x,2,C);
    for i in [2..l] do
        P := Modexp(P,2,C);
    end for;
    H := 1;
    for i in [l+1..b] do
        P := Modexp(P,2,C);
        Q := P - x;
        H := (H*Q) mod C;
        if H eq 0 then
            return true;
        end if;
    end for;
    H := Modexp(H,Degree(C),C);
    return H eq 0;
end function;
```

It implements the second strategy which we have described before, i.e. computing $H(x) = \prod_{i=1}^{bd}(x^{2^i} - x) \mod C(x)$ and then computing $H(x)^c \mod C(x)$ where c is the degree of $C(x)$. First of all the function `Modexp` is used to speed up the modular exponentiations. Moreover, x^{2^i} is successively computed by exponentiating a handle variable `P`. Moreover, we actually compute only a part of the product, that is $H(x) = \prod_{i=l+1}^{bd}(x^{2^i} - x) \mod C(x)$, where $l = \lfloor \frac{bd}{2} \rfloor$. Indeed, those polynomials are sufficient to contain as factors all the possible polynomials of degree up to bd. The pseudo-code can be found in [5].

We now consider the main function. In particular, we report in the following only a part of the code, the main loop. As we pointed out above, the use of sparse matrix techniques is of use to solve the final system of linear equations. Therefore we decided to use the functions in Magma for the specific implementation of sparse matrices, starting from the initialisation reported below:

```
    M1 := SparseMatrix(0,#FactorBase);
    M2 := SparseMatrix(0,#FactorBase);
```

Here `FactorBase` contains all the possible irreducibles and is built in the following way:

```
FactorBase := &cat[PrimePolynomials(R,d) : d in [1..bound]];
```

This vector is used to set an order on the columns of the matrices, so that we can properly associate each entry to the correspondent irreducible polynomial.
The main loop is reported below:

```
    while true do
        rows := rows + 1;
        L1,L2 := Explode(CandD(h,d,P));
        for el in L1 do
            SetEntry(~M1, rows, Index(FactorBase, el[1]), V!k*el[2]);
        end for;
        for el in L2 do
            SetEntry(~M2, rows, Index(FactorBase, el[1]), V!el[2]);
        end for;
        if rows ge s+10 and (rows mod 20) eq 0 then
            M := M1 - M2;
            v := ModularSolution(M,2^n-1);
            v := Eltseq(v);
            u := [u[i] ne 0 select u[i] else (elt<F|Coefficients(FactorBase[i])>
    eq Y^v[i] select v[i] else 0) : i in [1..#FactorBase]];
            if &and[u[i] ne 0 : i in [1..#FactorBase]] then
                return u;
            end if;
        end if;
    end while;
```

Here we compute a sequence of pairs to form the rows of the matrix. Note that in each part of the algorithm we keep using the methods made for sparse matrices, as `SetEntry`. However, the most noticeable is `ModularSolution`, that solves the sparse system efficiently. Multiple ideas have been used to speed up the algorithm. We do not try to solve the system from the beginning, but we start when we reach a sufficient number of rows, set to $s+10$ with $s = |\texttt{FactorBase}|$ is the number of columns. If we do not find a solution, we keep adding rows to the matrix and solve the system. Note that we wait 20 rows to be added, so that the computational cost is reduced. The main idea is not to reject a solution if it is not unique or if some entries are not correct. We save in an handle vector all the correct logarithms in the solution we have obtained using `ModularSolution`, and then continue the computation. Experimentally we found that looking for a unique solution is computationally unfeasible for high bounds.

This method could also be combined with the one implemented below:

```
        for rows in [1..param] do
            L1,L2 := Explode(CandD(FactorBase,P,k,h));
            row := Random([1..s+10]);
            RemoveRow(~M1, row);
            RemoveRow(~M2, row);
            for el in L1 do
                SetEntry(~M1, s+10, Index(FactorBase, el[1]), V!k*el[2]);
            end for;
            for el in L2 do
                SetEntry(~M2, s+10, Index(FactorBase, el[1]), V!el[2]);
            end for;
        end for;
```

Here the idea is to keep the size of the matrix fixed, and randomly change some rows to obtain different solutions. This method was designed to overcome the problem of finding

multiple solutions for the same system. Indeed, `ModularSolution` returns always the same solution, and it is unfeasible to search among all the possible solutions, since the kernel can be of arbitrary size. Therefore one can slightly change the system to try to obtain a different solution. This is also beneficial when the number of rows becomes too high and we want to fix the dimension of the system.However, this part has not been implemented in the final code. Indeed, it is not easy to verify whether some solutions are better than others, since a complete precomputation takes hours to run.

Computation phase

Concerning the computation phase, we chose to implement by ourselves the Extended Euclidean Algorithm in order to be able to stop it as soon as we find a suitable relation. The code is the following:

```
1  function ExtEucAlg(A, B, n)
2      R1 := A;
3      R2 := B;
4      T1 := 0;
5      T2 := 1;
6      while R2 ne 0 do
7          Q := R1 div R2;
8          R := R1 - Q * R2;
9          R1 := R2;
10         R2 := R;
11         T := T1 - Q * T2;
12         T1 := T2;
13         T2 := T;
14         if Degree(R2) le (n div 2 + 1) then
15             return R2, T2;
16         end if;
17     end while;
18     return R2, T2;
19 end function;
```

The exit condition in the lines 14-16 is used to ensure that we take the first pair `(R2, T2)` where we can hope that both the polynomials have degree not exceeding $\lfloor \frac{n}{2} \rfloor + 1$. Indeed, due to the stated properties, we know that deg `R2` is decreasing and that the two degrees are balanced with respect to $\frac{n}{2}$. Therefore, it is sufficient to check only the bound for `R2` to take the first pair where it has a reasonable degree, and then check in the enclosing function the degree of `T2`. If the degree of `R2` was not decreasing too quickly, also `T2` will respect the bound. Those lines were initially replaced by the following ones:

```
14         if Degree(R2) le (n div 2 + 1) and Degree(T2) le (n div 2 + 1) then
15             return R2, T2;
16         end if;
```

Here we check both the degrees, but as we said, the check on `T2` is not necessary. Moreover, if there is no suitable pair, the algorithm would not stop as soon as deg `R2`$\leq \lfloor \frac{n}{2} \rfloor + 1$, but it would continue until the end executing useless operations.

In the enclosing function `G1andG2`, we search for a pair $(G1, G2)$ from the Extended Euclidean Algorithm, until both the polynomials respect the bound of $\leq \lfloor \frac{n}{2} \rfloor + 1$ on the

degree. Another possibility is not to directly reject a pair if the condition is not met, but to try the smoothness test anyway. In this case, in the function `ExtEucAlg` would be better to take the pair whose degrees are as close as possible to $\frac{n}{2}$, as follows:

```
        if Degree(R2) le (n div 2) then
            if Abs(Degree(R2) - (n / 2)) gt Abs(Degree(R1) - (n / 2)) then
                return R1, T1;
            end if;
            return R2, T2;
        end if;
```

However, experimentally this solution turned out to be slower than the previous one, because the pairs where one of the polynomials has an high degree are not smooth with a high probability.

We now report the function where we find the actual computation of the logarithm:

```
function G1andG2(G,n,P)
    while true do
        repeat
            m := Random(2^n-1);
            G1, G2 := ExtEucAlg(P, (G * Modexp(x,m,P)) mod P, n);
        until Degree(G2) le (n div 2 + 1) and Degree(G1) le (n div 2 + 1);
        if SmoothTest(G1, bound) and SmoothTest(G2, bound) then
            L1:=Factorization(G1);
            L2:=Factorization(G2);
            break;
        end if;
    end while;
    return L1, L2, m;
end function;

function ComputeLog(G,n,P,c)
    if Degree(G) le bound then
        return c[Index(FactorBase,G)];
    end if;
    L1, L2, m := G1andG2(G,n,P);
    return (&+[ComputeLog(f[1],n,P,c)*f[2] : f in L1] -
    &+[ComputeLog(f[1],n,P,c)*f[2] : f in L2] - m) mod (2^n-1);
end function;
```

Note that we are not following the idea of successively diminishing the degree of the factors, but we use the direct way by applying the Euclidean Algorithm once with the original bound. This method was chosen after having performed several tests on the code with the alternative procedure, since at the end it resulted to be the faster one. The recursive structure of the main function reflects the one that we would use to lower the degree, indeed it is not difficult to adapt this code to the other solution.
Note that we factorize the polynomials using the function `Factorization` when we already know that they are smooth. This also happens in the precomputation phase, even if the dedicated code is not reported. Indeed, factorization is an heavy procedure from the computational point of view, and our new function `SmoothTest` turned out to be a faster alternative.

Since to the last we wanted to solve the problem by using the algorithm contained in [1] but we had to end up choosing the other solution, the final code contained an error

due to the partial adaptation from the previous code. The old code had the lines 7-11 replaced with:

```
7           b := Round(Sqrt(bound*Degree(G)));
8           if SmoothTest(G1, b) and SmoothTest(G2, b) then
9               L1:=Factorization(G1);
10              L2:=Factorization(G2);
11              break;
12          end if;
```

Here we tried to use the Euclidean Algorithm to successively lower the degrees, without recalling that in lines 4-5 we came back to an high degree due to the multiplication by x^m with m random in $\mathbb{Z}_{2^n-1}$. Therefore, at the last recursive call for the final bound bd, we were actually performing the same calculation as we would have done by running the algorithm once. This modification made the code run in about $\frac{1}{5}$ of the time, since we avoid useless operations.

At the end, we report the code in which the alternative solution is implemented:

```
function PolynomialSieve(G,b,n,h,k,P)
    b12 := Round(Sqrt(b*Degree(G)));
    d12 := Floor((Degree(G) + Sqrt(n/b)*Log(2,n/b))/2);
    while true do
        d12 := d12 + 1;
        A := R![Random(GF(2)) : _ in [1..d12+1]];
        t := Degree(A);
        for d in [1..b12] do
            dim := Max(0,d12-d);
            B := (A*(x^h)) mod G;
            if Degree(B) lt d12 then
                for i in [1..2^(dim)] do
                    B := B + (x^l(i))*G;
                    C := (x^(h))*A + B;
                    if C ne 0 and C ne 1 and C ne G then
                        N := C div G;
                        if SmoothTest(N,b12) then
                            L1:=Factorization(N);
                            D := Modexp(C,k,P);
                            if SmoothTest(D,b12) then
                                L2:=Factorization(D);
                                return <L1,L2>;
                            end if;
                        end if;
                    end if;
                end for;
            end if;
        end for;
    end while;
end function;
```

As remarked above, it could be implemented using a modification of the described polynomial sieve, where we do not keep the database for all possible polynomials $B(x)$ and we do not iterate over all possible irreducible polynomials g, but we only consider $G(x)$ and the associated pairs.

The first solution resulted to be the fasted one due to a non-optimal implementation of the sieve, since in theory this second solution is specifically made to optimise the phase.

In our final solution, we no longer look for random relations among logarithms using the strategy that involves the computation of $C(x)$ and $D(x)$, but we only execute the Extended Euclidean Algorithm. Therefore, the choice of the parameter $P(x)$ is no longer influential in the computation phase. As already mentioned, the better choice is not to change the representation, even if $Q(x)$ has an high degree. This was tested experimentally for the polynomials

	$P(x)$	$\deg Q(x)$	$\deg q(x)$
$n = 73$	`Parent(a)`	4	3
	$x^{73} + x^8 + x^6 + x^2 + 1$	8	2
$n = 100$	`Parent(a)`	57	32
	$x^{100} + x^8 + x^7 + x^2 + 1$	8	7
	$x^{100} + x^{10} + x^9 + x^7 + x^6 + x^4 + x^3 + x + 1$	10	2

where $q(x)$ denotes the irreducible factor of $Q(x)$ of highest degree. In particular when using the polynomial sieve the best polynomials to use are the ones with smallest $\deg q(x)$, even if for $n = 73$ there is a minimal difference. However, when using the direct computation with $bd = 17$ it is better to keep the original polynomials to avoid the computation of the isomorphism.

Personal contribution and final considerations

The project described in this report has been implemented by the group consisting of Emanuele Di Lernia, Cecilia Marchi and myself.
Since the code to be implemented was not already well partitioned, we decided to work together and not clearly divide the tasks. Therefore, the main functions come from collaborative work, then for each problem we encountered, each one suggested and implemented different strategies.
Starting from the precomputation, we immediately headed for the sparse matrices as the paper suggested it. It was my idea to save in an handle vector the correct logarithms during the computations, so that we did not have to find the unique solution. Associated with this, I had the idea of changing the system by substituting some rows, and this was the strategy used when we actually computed the databases. I also added the numerical considerations on not solving the system at each iteration.
Concerning the computation phase, I implemented the Extended Euclidean Algorithm, since I noted that this was convenient with respect to the native Magma function. I also took care of all the changes on this algorithm.
Moreover, I found the various irreducible polynomials worth to be tested, in order to choose the best representation of the field. Within this context, I also implemented the change of presentation, identifying the best way to implement the homomorphism and noticing that the logarithm of a could be precomputed.

After the submission of the project, some improvements have been made.
The check on the degree of `T2` was removed from the function `ExtEucAlg`. This has not brought a clear improvement to the timings, since the algorithm is fast anyway.
The essential modification consisted in removing the error in the function `G1andG2` that, as already mentioned, made the code run in about $\frac{1}{5}$ of the original time.

References

[1] D. Coppersmith, *Fast evaluation of logarithms in fields of characteristic two*. In: *IEEE Trans. Inf. Theory* 30 (1984), pp. 587–593.

[2] Emmanuel Thomé, *Computation of Discrete Logarithms in* $\mathbb{F}_{2^{607}}$. In: *Advances in Cryptology, Asiacrypt* (2001), pp. 107–124.

[3] A. M. Odlyzko, *Discrete logarithms in finite fields and their cryptographic significance*. In: *Advances in Cryptology, Eurocrypt* (1984), pp. 224–314.

[4] D. M. Gordon, K. S. McCurley, *Massively Parallel Computation of Discrete Logarithms*. In: *Advances in Cryptology, Crypto* (1992), pp. 312–323.

[5] J.-F. Biasse, M. J. Jacobson Jr., *Smoothness testing of polynomials over finite fields*. In: *Advances in Mathematics of Communications* 8.4 (2014), pp. 459-477.