Relation collection for the Function Field Sieve

Jérémie Detrey, Pierrick Gaudry, Marion Videau

Relation collection for the Function Field Sieve

Jérémie Detrey, Pierrick Gaudry and Marion Videau
INRIA – CNRS – Université de Lorraine

September 27, 2012

Abstract

In this paper, we focus on the relation collection step of the Function Field Sieve (FFS), which is to date the best known algorithm for computing discrete logarithms in small-characteristic finite fields of cryptographic sizes. Denoting such a finite field by $\mathbb{F}_{p^n}$, where p is much smaller than n, the main idea behind this step is to find polynomials of the form $a(t) - b(t)x$ in $\mathbb{F}_p[t][x]$ which, when considered as principal ideals in carefully selected function fields, can be factored into products of low-degree prime ideals. Such polynomials are called "relations", and current record-sized discrete-logarithm computations require billions of them.

Collecting relations is therefore a crucial and extremely expensive step in FFS, and a practical implementation thereof requires heavy use of cache-aware sieving algorithms, along with efficient polynomial arithmetic over $\mathbb{F}_p[t]$. This paper presents the algorithmic and arithmetic techniques which were put together as part of a new implementation of FFS, aimed at medium- to record-sized computations, and planned for public release in the near future.

Keywords: function field sieve; discrete logarithm; polynomial arithmetic; finite-field arithmetic.

1 Introduction

The computation of discrete logarithms is an old problem that has received a lot of attention after the discovery of public-key cryptography in 1976. Indeed, together with integer factorization, it is the most famous computationally hard problem on which the security of public-key algorithms relies. We are interested here in the case of discrete logarithms in finite fields of small characteristic, for which the best known algorithm to date is the function field sieve (FFS) first introduced by Adleman [1] in 1994. Since then, and until 2005, there have been several works on this algorithm, both on the theoretical side [2, 19, 9] and on the practical side [10, 16, 15]. This culminated with a record set by Joux and Lercier who computed discrete logarithms in $\mathbb{F}_{2^{613}}$ and surpassed the previous record set by Thomé [23] in 2001 using Coppersmith's algorithm [7]. A long period with essentially no activity on the topic has followed, and has recently ended with a new record [14, 13] set over $\mathbb{F}_{3^{6\times 97}}$, which took advantage of the fact that the extension was composite; this particular case is of practical interest in pairing-based cryptography.

The FFS algorithm can be decomposed into several steps. Two of them are by far the most time-consuming: the relation collection step and the linear algebra step. While the latter is not specific to FFS—it is a sparse-matrix kernel computation that is essentially the same as for other discrete-logarithm algorithms based on combining relations—the relation

collection, on the other hand, is specific to each family of algorithms. The purpose of this paper is to present a new implementation of this important step for the function field sieve, targeting in our case finite fields of small characteristic with no special Galois structure. We choose two benchmark finite fields, in characteristic 2 and in characteristic 3, that are the most interesting for cryptographic applications. The extension degrees were chosen so as to match the "kilobit milestone": we consider $\mathbb{F}_{2^{1039}}$ and $\mathbb{F}_{3^{647}}$. Note that the factorization of $2^{1039}-1$ has been completed only recently [3]. This is relevant to our benchmark as, among others, an 80-digit (265-bit) prime factor was exhibited: had $2^{1039}-1$ been a product of only moderately large primes, there would have been better algorithms than FFS for computing discrete logarithms in $\mathbb{F}_{2^{1039}}$.

In the literature, many implementation details about the relation collection are merely sketched. In this article, we wish to explain as many of them as possible. Some of them are "folklore" or easy adaptations of techniques used in the Number Field Sieve (NFS) for integer factorization. For instance, the notion of skewness, or the use of bucket sorting during sieving that we describe below will come as no surprise to the NFS *cognoscenti*. However, to the best of our knowledge, some techniques go beyond what was previously known. We mention two of them. The sieving step starts with a phase called *norm initialization*, where a large array is initialized with the degree of a polynomial, which is different for each cell. In several sieving implementations, only an approximation of the size of the norm is used, in order to save time. We propose several strategies that allow us to get an exact result in this step, at a very reasonable computational cost. This includes using an analog of floating-point arithmetic for polynomials. Another original contribution lies in the heart of the relation collection, during a sieving step, which is similar in essence to the sieve of Eratosthenes: in the same large array as above, we have to visit all the positions that correspond to some arithmetic property. We show that, in FFS, the set of these positions forms a vector space for which we give the general form of a basis, along with a method to compute it quickly following some Euclidean algorithm. We also adapt the well-known Gray code walk to our case where we are only interested in monic linear combinations.

To complement the description of our implementation, we plan to make its source code freely available[1]. This code is used to propose runtime estimates for the relation collection step in the two kilobit-sized finite fields that we target. It reveals that these sizes are easily reachable with only moderate computing resources, since it is a matter of months on a typical 1000-core cluster. The question of the subsequent linear algebra step is left open for future research. Still, it is already fair to claim that the cryptosystems whose security relies on the discrete-logarithm problem in a small-characteristic finite field of kilobit size provide only marginal security. Even more so, due to the fact that the relation collection and the linear algebra steps are to be done just once and for all for a given finite field.

Outline. The paper is organized as follows. A short overview of the function field sieve, along with a focus on its relation collection step, is first given in Section 2. Section 3 then highlights several key aspects of the proposed implementation, while low-level architectural and arithmetic issues are addressed in Sections 4 and 5, respectively. Timing estimates for two kilobit-sized finite fields are presented in Section 6, before a few concluding remarks and perspectives in Section 7.

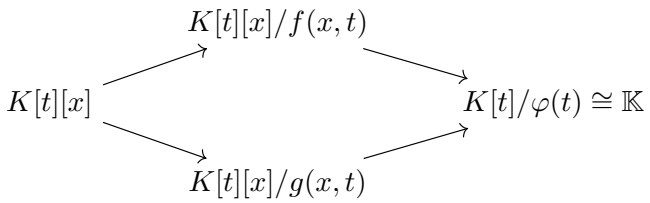
[1]It is currently a private repository; please contact the authors for request of a snapshot.

2 Collecting relations in FFS

2.1 A primer on the FFS algorithm

The principle of the Function Field Sieve algorithm is very similar to that of the Number Field Sieve algorithm for computing discrete logarithms in prime fields, where the ring of integers $\mathbb{Z}$ is replaced by the ring of polynomials $K[t]$, with K a finite field of size $\#K = \kappa$.

Assume that we want to compute discrete logarithms in $\mathbb{K}$, a degree-n algebraic extension of the base field K, of cardinality $\#\mathbb{K} = \kappa^n$. We consider two polynomials f and g in $K[t][x]$ such that their resultant in the x variable contains an irreducible factor $\varphi(t)$ of degree n. The following diagram, where all maps are ring homomorphisms, is commutative:



The key to FFS is to study the behavior of an element of the form $a(t) - b(t)x$ in this diagram. Along the two paths, the element is tested for *smoothness* in the appropriate algebraic structures. Pushing the factored versions into the finite field $\mathbb{K}$, we get a linear relation between the discrete logarithms of "small" elements. Once enough relations have been collected, they are packed in a matrix for which a kernel element gives the discrete logarithms of these "small" elements. Finally, the logarithm of a given element is obtained by a procedure called special-$\mathfrak{q}$ descent that uses again the diagram to produce a relation between the logarithm of the given element and the logarithms of smaller elements, and recursively so until we get a link to logarithms of the "small" elements.

The smoothness notion is as follows. The element $a(t) - b(t)x$ is mapped to an element of $K[t][x]/f(x,t)$ (substitute g for f for the other side), which is viewed as a principal ideal in the ring of integers of the corresponding function field $K(t)[x]/f(x,t)$. Since this is a Dedekind domain, there is a unique factorization in prime ideals. And if these prime ideals have degrees less than a parameter called the *smoothness bound*, we say that the principal ideal is smooth. Just like with NFS, there are complications arising from the fact that the ring of integers is not necessarily principal and that there are non-trivial units. The Schirokauer maps and the notion of virtual logarithms [21] are the tools to solve these complications; we do not need to worry about this here, since it does not interfere with the relation collection step.

2.2 Overview of the relation collection step

In this paper, we focus on this so-called *relation collection* step: given a smoothness bound—which might be different for each side—the goal is to produce (a, b) pairs, called *relations*, that simultaneously yield smooth ideals for both sides. Solving the resulting linear system requires finding at least as many relations as there are prime ideals of degree less than the smoothness bound.

The relation collection step is independent of the other steps in the FFS algorithm as long as we allow for any kinds of polynomials f and g. This is the most time-consuming part, both in practice and in theory: optimizing the parameters for this step will optimize

the overall algorithm. We remark however that the linear algebra step has a high space complexity that can become a practical issue. The usual approach to circumvent this problem is to let the relation collection run longer than what theory alone would recommend. This produces a larger set of relations, and there exist algorithms for selecting among them a subset that is better suited for linear algebra than the original set of relations obtained without this *oversieving* strategy. The RSA-768 factorization record [18] is a good example of this approach.

Ideals and norms. The highbrow description of the smoothness notion deals with ideals. Without loss of generality, we focus here on the side of the diagram corresponding to the polynomial f, the other side being similar. For our purposes, it is enough to see a prime ideal $\mathfrak{p}$ as a pair of polynomials $(p(t), r(t))$, where p is a monic irreducible polynomial and $r(t)$ is a root in x of the polynomial $f(x,t)$ modulo $p(t)$. More formally, the notation $\mathfrak{p} = (p(t), r(t))$ represents the ideal $\langle p(t), x - r(t)\rangle$. Restricting to ideals of that form means that, in the article, we ignore the places at infinity and the problems that ramification can cause. Our implementation takes care of these more complicated cases, but we prefer to keep the presentation simple.

An element (a, b) that potentially leads to a relation must be tested for smoothness. It means that we want the corresponding principal ideal on the f side to be the product of "small" prime ideals. Here, we face a terminology issue, since in our setting the good notion of size of a prime ideal is the degree of its p polynomial. Therefore, we are going to talk about the degree of a prime ideal $\mathfrak{p} = (p, r)$ as the degree of p (even though, formally, all of these have degree one).

The factorization of the principal ideal is dictated by the factorization of its *norm* which, in our case, takes a very nice form. Denoting by $F(X, Y, t)$ the polynomial obtained from $f(x, t)$ by homogenization with respect to the x variable, the norm corresponding to a pair (a, b) is $F(a, b)$, a polynomial in t. Then, to each irreducible factor p of $F(a, b)$ corresponds a prime ideal $\mathfrak{p} = (p, r)$ that divides the principal ideal, and such that $r \equiv a/b \pmod{p}$.

As a consequence, a simple restatement of the relation search task is to find (a, b) pairs for which the two norms $F(a, b)$ and $G(a, b)$ have all their irreducible factors in $K[t]$ of degree less than a given bound.

An important remark is that only *primitive* pairs are interesting, in the sense that if (a, b) gives a valid relation, then multiplying both a and b by the same small factor also gives another relation which, however, is essentially the same and is of no use to the next steps of the algorithm. Therefore, we are only interested in the case where $\gcd(a, b) = 1$ and, furthermore, where (say) b is monic.

Sieving and cofactorization. In order to test the smoothness of the norms corresponding to a pair (a, b), we proceed using a two-pass strategy as is now usual for NFS. The first pass consists in using a sieving process *à la* Eratosthenes in order to efficiently remove the contribution of the prime ideals of degree less than a given *factor base bound*—chosen to be smaller than the actual smoothness bound—to the norm of the principal ideals that they divide. All of those (a, b) pairs such that what then remains of their norm is small enough will have better chances to completely factor as products of prime ideals of degree less than the smoothness bound. As such, the promising pairs which have survived sieving on both the f and the g sides are selected to undergo a full-blown smoothness test in the second pass of

the algorithm, which is sometimes called *cofactorization* or *large prime separation*.

Note that, there is no need to keep track of the actual norm of the principal ideal in the first pass. In fact, it is enough to consider only the *degree* of the norm here, to which we simply subtract $\deg p$ for every prime ideal $\mathfrak{p} = (p, r)$ which divides the corresponding principal ideal.

Also note that, because of the sieving techniques used in the first pass, we cannot process each pair (a, b) one after the other: the sieving region has to be considered as a whole and allocated *a priori* as a large array, delimited by upper bounds on the degrees of a and b, and in which all the (a, b) positions will be considered simultaneously by the sieving process.

3 Organization of the computation

3.1 Sieving by special-$\mathfrak{q}$

Since the set of (a, b) pairs that are tested for smoothness is very large, we have to subdivide this search space into pieces that are to be handled separately. Following the current trend for NFS and FFS, we split the (a, b) space according to so-called *special*-$\mathfrak{q}$ lattices.

A special-$\mathfrak{q}$ is just a prime ideal $\mathfrak{q} = (q, \rho)$ corresponding to either the f or the g side, and the $\mathfrak{q}$-lattice is exactly the set of (a, b) pairs for which the principal ideal on this side is divisible by $\mathfrak{q}$. This corresponds to the set of (a, b) pairs such that $b\rho \equiv a \pmod{q}$, which is a $K[t]$-lattice of dimension 2 and determinant q. The relation search is then seen as a set of independent tasks, each task corresponding to finding relations in a given $\mathfrak{q}$-lattice.

Basis of the $\mathfrak{q}$-lattice. Let $\mathfrak{q} = (q, \rho)$ be a special-$\mathfrak{q}$. A basis of the $\mathfrak{q}$-lattice is given by the two vectors $(q, 0)$ and $(\rho, 1)$. It is a very unbalanced basis, and we can improve it using Gaussian lattice reduction to obtain two vectors u_0 and u_1 whose coordinates have a degree about half of the degree of q. We then look for relations coming from (a, b) pairs of the form $iu_0 + ju_1$, with i of degree less than I and j of degree less than J. This leads to a and b of degree approximately equal to $\max(I, J) + \frac{1}{2}\deg q$.

Skewness. There are cases where the polynomials f and g are *skewed*, in the sense that the degree in t of the x^i coefficients decreases when i increases. In that case, a relevant search space for (a, b) that yields norms of more or less constant degree is a rectangle: the bound on the degree on a must be larger than the bound on the degree of b. The difference of these two bounds on the degrees is called the *skewness*. In the $\mathfrak{q}$-lattice, to obtain the target skewness adapted to the f and g polynomials, the Gaussian lattice reduction must be modified, so that the vectors u_0 and u_1 both have a difference of degrees between their coordinates that is close to the skewness. In the implementation, this translates into a small modification of the stopping criterion of the lattice reduction.

From now on, we consider that the sieving space for a given special-$\mathfrak{q}$ is a κ^I by $\left(\frac{\kappa^J-1}{\kappa-1} + 1\right)$ rectangle in the (i, j) plane, covering all the polynomial pairs of degrees less than I and J, respectively, and such that the j coordinate is either 0 or monic. The correspondence with (a, b) is of the form $a = ia_0 + ja_1$ and $b = ib_0 + jb_1$, where $u_0 = (a_0, b_0)$ and $u_1 = (a_1, b_1)$ is the skew-reduced basis of the $\mathfrak{q}$-lattice.

In our current implementation, I and J are fixed and equal. It could make sense to let them vary, for instance giving them a smaller value for larger special-$\mathfrak{q}$'s so as to keep more

or less always the same bound on the (a, b) pairs.

3.2 Initializing the norms

Initializing the norm at a given position is a simple task; however it can become costly if done naively. One could use an estimate of the degree (*e.g.*, take an upper bound) but it would impact the number of positions to test for smoothness by either losing or keeping too many positions. To avoid these drawbacks, we decided to work on being precise and even exact on the degree computation at a reasonable cost (ideally a negligible one).

We present our method with the polynomial f; of course, it works similarly for the polynomial g. Recall that F denotes the homogenization of the polynomial f. For each special-$\mathfrak{q}$, we start by computing a linear transformation on f to get $f_\mathfrak{q}$ such that $F_\mathfrak{q}(i, j) = F(a, b)$. Then we have to compute the degree (in t) of $F_\mathfrak{q}(i, j)$ and put it into the array to initialize the corresponding position. We assume from now on that the loop is organized row by row, *i.e.*, that j is fixed and then i varies.

Saving many degree computations. Recall that, for a pair (i, j), the norm that has to be tested for smoothness is given by $F_\mathfrak{q}(i, j) = \mathfrak{f}_d i^d + \mathfrak{f}_{d-1} i^{d-1} j + \cdots + \mathfrak{f}_0 j^d$, where each term $\mathfrak{f}_k i^k j^{(d-k)}$ is a polynomial in t of degree equal to $\deg(\mathfrak{f}_k) + k \deg(i) + (d - k) \deg(j)$.

When a position (i_0, j_0) has been initialized, it is possible to deduce the initialization of many following positions $(i_0 + i', j_0)$ since for small values of i' the degree of the norm does not change. To precisely characterize these i', we define the notion of *gap* on the norm evaluated at (i_0, j_0).

Definition 1. *With the previous notation, the* gap *of $F_\mathfrak{q}$ at (i_0, j_0), denoted by γ, is defined by $\gamma = -1$ if only one term $\mathfrak{f}_k i_0^k j_0^{(d-k)}$ has maximal degree and by*

$$\gamma = \max_{0 \le k \le d} \deg\left(\mathfrak{f}_k i_0^k j_0^{(d-k)}\right) - \deg\left(F_\mathfrak{q}(i_0, j_0)\right), \text{ otherwise.}$$

If there is only one term $\mathfrak{f}_k i_0^k j_0^{(d-k)}$ of maximal degree (*i.e.*, $\gamma = -1$), this property will not change as long as $(i_0 + i')$ has the same degree as i_0. We thus get the degree of the norm for free until the degree of $(i_0 + i')$ increases.

If there are several terms of maximal degree, then only the γ most significant coefficients of $(i_0 + i')$ can change the degree of the norm. Therefore, for all i' such that $\deg(i') < \deg(i_0) - \gamma$, we also get the degree of the norm for free.

Since computing the gap comes at almost no additional cost when computing the degree of the norm, this method saves many computations.

Computing the degree of the norm. If there is only one term, let say $\mathfrak{f}_{k_0} i_0^{k_0} j_0^{(d-k_0)}$, of maximal degree, then $\deg\left(F_\mathfrak{q}(i_0, j_0)\right) = \deg(\mathfrak{f}_{k_0}) + k_0 \deg(i_0) + (d - k_0) \deg(j_0)$, $\gamma = -1$ and we do not have to actually compute the norm to obtain its degree.

If there are several terms of maximal degree, we need to take care of the possible cancellations. As a cheaper alternative to computing the actual norm, we resort to an adaptive-precision approximation, using an analog of the floating-point number system for polynomials. The precision N representation of a polynomial $p(t) = \sum p_k t^k$ is then given by a pair

$(\deg p, \mathrm{mant}_N(p))$, where the mantissa $\mathrm{mant}_N(p)$ is a polynomial of degree $N-1$ corresponding to the N most significant coefficients of p:

$$\mathrm{mant}_N(p) = \frac{p - (p \bmod t^{\deg p - N + 1})}{t^{\deg p - N + 1}} = p_{\deg p} t^{N-1} + \cdots + p_{\deg p - N + 1}.$$

Starting from an initial precision $N = N_0$, we compute the "floating-point" approximations of the terms $\mathfrak{f}_k i_0^k j_0^{(d-k)}$, using truncated high products for the multiplications. Before summing those terms, we first align them to the maximal degree, as is the case for regular floating-point additions. This can be seen as a conversion to a fixed-point representation with precision N. Finally, from the degree of the sum of the terms, we deduce the degree of the norm (and the gap), unless the result is zero, meaning that the precision N was not high enough to conclude.

Therefore, the computation takes the form of a loop where the precision N increases iteratively until the computation of the norm with N significant coefficients is enough to infer its degree.

A few more improvements can be added to this strategy. When the precision N is small, it might happen that several terms have a degree that is too small to contribute to the final sum, so that their computation can be skipped from the beginning. Also, when summing the terms, it can make sense to do so starting with the terms of higher degrees. If there are not too many cancellations, it can be the case that we can then guess the final degree before having to take the remaining terms into account.

We finally remark that the first step where we consider only the degrees of the terms can be seen as the particular case $N = 0$ of the subsequent loop.

3.3 Bases for the $\mathfrak{p}$-lattices

Let $\mathfrak{p} = (p, r)$ be a prime ideal. The goal of this section is to describe the set of (i, j) positions for which the corresponding ideal is divisible by $\mathfrak{p}$ (hence the norm is divisible by p). We recall that in terms of (a, b) position, it translates into the condition $a - br \equiv 0 \pmod p$, and therefore, the set of such (a, b) positions is a $K[t]$-lattice for which $(p, 0)$ and $(r, 1)$ is a basis.

Using the definition of a and b in terms of i and j, we find that the condition becomes

$$i(a_0 - rb_0) \equiv -j(a_1 - rb_1) \pmod p.$$

This is a K-linear condition on the polynomials i and j, and therefore the set of solutions is a K-vector space. The degree of i must be less than I and the degree of j less than J, so that this vector space is of finite dimension. More precisely, the linear system corresponding to the condition has $I + J$ unknowns and $\deg p$ equations, so that the dimension of the set of solutions is at least $I + J - \deg p$. We will present a basis for this vector space, that takes a different shape, depending on the degree of p being smaller or larger than I.

As a warm-up, we deal with the case where $a_0 - rb_0 \equiv 0 \pmod p$. The main condition on i and j simplifies to $j \equiv 0 \pmod p$. A basis is then given by the set of vectors $\mu_k = (t^k, 0)$ for $0 \le k < I$, and $\nu_\ell = (0, t^\ell p)$ for $0 \le \ell < J - \deg p$. The dimension is then $I + J - \deg p$.

From now on, we will assume that $a_0 - rb_0$ is invertible modulo p, and we define

$$\lambda_\mathfrak{p} = -\frac{a_1 - rb_1}{a_0 - rb_0} \mod p,$$

so that the main condition becomes $i \equiv \lambda_\mathfrak{p} j \pmod p$.

3.3.1 Case of small primes

Lemma 2. *Let $\mathfrak{p} = (p, r)$ be a prime ideal of degree $L < I$. The vector space of (i, j) positions for which the corresponding ideal is divisible by $\mathfrak{p}$ is of dimension $I + J - L$ and admits the basis given by the vectors $\mu_k = (t^k p, 0)$ for $0 \leq k < I - L$, and $\nu_\ell = (t^\ell \lambda_\mathfrak{p} \mod p, t^\ell)$ for $0 \leq \ell < J$.*

Proof. The vectors of the given set are indeed in the target set of positions, and they are linearly independent, due to their echelonized degrees property. To see that it is a basis, let us consider (i, j) with $\deg i < I$, $\deg j < J$, and $i \equiv j\lambda_\mathfrak{p} \pmod p$. Let then i_0 be the polynomial of degree less than p such that $i_0 \equiv j\lambda_\mathfrak{p} \pmod p$. Then the vector (i_0, j) can be obtained by combining the ν_ℓ's with the coefficients of j. Furthermore i and i_0 differ by a multiple of p that is of degree less than I, so that (i, j) can be written as a linear combination of the claimed basis, which is hence, indeed a basis. □

3.3.2 Case of large primes

We assume now that $\mathfrak{p}$ is of degree greater or equal to I. In fact, we can assume furthermore that its degree is also greater or equal to J. Indeed, the symmetric role of I and J makes it possible to write a similar basis as before if the degree of J is larger than the degree of $\mathfrak{p}$. In practice, this would raise questions to keep the locality when visiting a vector space given by a basis of this shape, but for the moment, we always take $I = J$ in our code.

The $K[t]$-lattice of the valid (i, j) positions corresponding to $\mathfrak{p}$ is generated by $(p, 0)$ and $(\lambda_\mathfrak{p}, 1)$. To study this lattice, we consider the sequence of vectors computed by the Euclidean algorithm starting with these two vectors: let $v_\ell = (\sigma_\ell, \tau_\ell)$, defined by $v_0 = (p, 0)$, $v_1 = (\lambda_\mathfrak{p}, 1)$ and for all $\ell \geq 1$,

$$\sigma_{\ell+1} = \sigma_{\ell-1} - \sigma_\ell q_\ell, \quad \text{and} \quad \tau_{\ell+1} = \tau_{\ell-1} - \tau_\ell q_\ell,$$

where q_ℓ is the quotient in the Euclidean division of $\sigma_{\ell-1}$ by σ_ℓ. Since p is prime, $\lambda_\mathfrak{p}$ is coprime to p and the final vector of the sequence is $(0, p)$. The finite family of vectors (v_ℓ) is a K-free family of vectors with the degrees of both coordinates less than or equal to L. In the following lemma it is shown how to complete it to form a basis of such vectors.

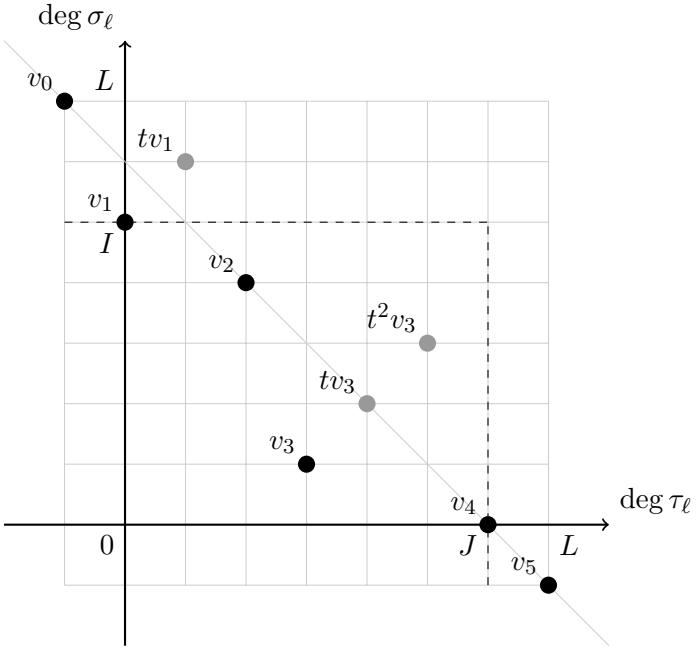
Lemma 3. *Let $\mathfrak{p} = (p, r)$ be a prime ideal of degree L and $(v_\ell = (\sigma_\ell, \tau_\ell))$ the corresponding Euclidean sequence. The vector space of (i, j) positions for which the corresponding ideal is divisible by $\mathfrak{p}$ with degrees at most L in each coordinate is of dimension $L + 2$ and admits the following basis:*

$$\{v_0\} \cup \left(\bigcup_{\ell \geq 1} \{t^s v_\ell \mid 0 \leq s < \deg \sigma_{\ell-1} - \deg \sigma_\ell\} \right).$$

Proof. The claimed basis is such that the degree in the first coordinate takes all values from -1 to L exactly once[2], and the same for the second coordinate. Indeed, since, by induction, the degrees of the σ_ℓ's are strictly decreasing, and those of the τ_ℓ's are strictly increasing, we have that $\deg \tau_{\ell+1} - \deg \tau_\ell = \deg q_\ell$, which is itself, by construction, equal to $\deg \sigma_{\ell-1} - \deg \sigma_\ell$. Therefore, when iterating from $v_{\ell-1}$ to v_ℓ, a drop δ in the degree of σ_ℓ is directly followed by an identical increase in the degree of τ_ℓ in the next iteration, thus showing that the vectors $t^s v_\ell$, for $0 \leq s < \delta$, bridging the gap between $v_{\ell-1}$ and $v_{\ell+1}$, will all have distinct degrees in

[2]We define the degree of the zero polynomial to be -1 for convenience.

Figure 1: The vectors obtained during the Euclidean algorithm (black dots), completed with some of their multiples (gray dots) to form the full K-basis of vectors of the $\mathfrak{p}$-lattice with coordinates of degree at most $\deg p$.



both of their coordinates, as illustrated in Figure 1. Consequently, the proposed family of vectors is K-free and contains $L + 2$ elements.

In order to get the exact dimension of the vector space, we just count its elements. The vector space is

$$\{(i, j) \mid i \equiv \lambda_{\mathfrak{p}} j \pmod{p}, \text{ with } \deg i \leq L \text{ and } \deg j \leq L\},$$

and we split it into κ^{L+1} disjoint subsets S_j, one for each possible value of j with $\deg j \leq L$. For any j, S_j has the same cardinality as the set of polynomials i of degree at most L such that $i \equiv j\lambda_{\mathfrak{p}} \pmod{p}$. Since L is precisely the degree of p, this set has the same cardinality κ as the base field K. Adding the contributions, we obtain a K-vector space of cardinality κ^{L+2}, and thus of dimension $L + 2$ as claimed. □

For the algorithm, we are interested only in those elements for which the degrees of their coordinates are (strictly) bounded by I and J respectively. Due to the echelonized form of the basis, it is enough to keep the basis elements that satisfy the degree constraints to generate all the elements with these constraints.

3.4 Enumeration of the $\mathfrak{p}$-lattice

Once the K-basis of a $\mathfrak{p}$-lattice is known, the next step is to enumerate all the (i, j) pairs in this lattice and to subtract $\deg p$ to the degree of the norm of the corresponding ideals. Obviously, this task can be achieved by considering all the possible K-linear combinations of the basis vectors. However, attention must be paid to a few details.

In order to address these issues for all possible types of $\mathfrak{p}$-lattices at once, whether $\mathfrak{p}$ be a small or a large prime ideal, or even when $a_0 - rb_0 \equiv 0 \pmod{p}$, we introduce the following notations, designed to cover all cases without loss of generality: given a prime ideal $\mathfrak{p}$, we denote the basis of the corresponding lattice seen as a K-vector space, as constructed in Section 3.3, by the vectors $(\mu_0, \ldots, \mu_{d_0-1}, \nu_0, \ldots, \nu_{d_1-1})$, where:

- the μ_k vectors are of the form $\mu_k = (\gamma_k, 0)$, for $0 \le k < d_0$;
- the ν_ℓ vectors are of the form $\nu_\ell = (\alpha_\ell, \beta_\ell)$, for $0 \le \ell < d_1$;
- the degrees of the γ_k's are echelonized: $0 \le \deg \gamma_0 < \deg \gamma_1 < \cdots < \deg \gamma_{d_0-1} < I$; and
- the degrees of the β_ℓ's are echelonized: $0 \le \deg \beta_0 < \deg \beta_1 < \cdots < \deg \beta_{d_1-1} < J$.

Since in the end we are only interested in primitive (a, b) pairs, we impose j to be monic (the condition $\gcd(i, j) = 1$ is not compatible with sieving). Therefore, we normalize the ν_ℓ vectors of the basis so that their second coordinate β_ℓ is monic. Using the fact that the β_ℓ's are echelonized and sorted by increasing degree, enumerating only those lattice vectors whose j-coordinates are monic boils down to considering only *monic* linear combinations of the ν_ℓ's, that is to say, K-linear combinations whose most significant nonzero coefficient, if any, is 1.

On the other hand, since the j-coordinates of the μ_k's are all zero, all K-linear combinations of these vectors should be considered. We start by explaining this easy case and come back to the ν_ℓ later.

Assuming that K is a prime field, the enumeration of all K-linear combinations can be carried out by means of a κ-ary Gray code, as already suggested by Gordon and McCurley in the binary case [8].

Let us consider the infinite sequence $\Delta = (\delta_i)_{i \ge 1}$ of the t-adic valuations of the nonzero polynomials of $K[t]$ sorted by increasing lexicographic order. For a given integer $d \ge 0$, the prefix sequence $\Delta_d = (\delta_1, \ldots, \delta_{\kappa^d - 1})$ can be computed via a recursive definition as $\Delta_0 = ()$ (the empty sequence), and $\Delta_{i+1} = (\Delta_i, i, \Delta_i, i, \ldots, i, \Delta_i)$, where Δ_i and i are repeated κ and $\kappa - 1$ times, respectively. For instance, over the base field $K = \mathbb{F}_3$, one would have $\Delta = (0, 0, 1, 0, 0, 1, 0, 0, 2, 0, 0, 1, \ldots)$.

In fact, Δ_d corresponds to the transition sequence of a d-digit κ-ary Gray code: starting from the polynomial $\pi_1 = 0$, and repeating the construction $\pi_{i+1} = \pi_i + t^{\delta_i}$, we end up after $\kappa^d - 1$ iterations with the set $\{\pi_1, \ldots, \pi_{\kappa^d}\}$ covering precisely all the polynomials of $K[t]$ of degree less than d. Additionally, given a K-free family of vectors $(\mu_k)_{0 \le k < d}$, we can use the iteration $\pi_{i+1} = \pi_i + \mu_{\delta_i}$ instead, this way enumerating the whole set of K-linear combinations of the μ_k's, at the cost of only one vector addition per combination. This technique can therefore be used to efficiently go through all the κ^{d_0} linear combinations of the μ_k basis vectors.

However, as far as the ν_ℓ's are concerned, this method cannot be applied directly, as only the *monic* linear combinations of these basis vectors need be considered. To that intent, we define a so-called *monic* variant of κ-ary Gray codes by considering the infinite sequence $\Delta' = (\delta'_i)_{i \ge 1}$ defined as the t-adic valuations of the nonzero *monic* polynomials of $K[t]$ sorted by increasing lexicographic order. As for Δ, the prefix sequence $\Delta'_d = (\delta'_1, \ldots, \delta'_{(\kappa^d - 1)/(\kappa - 1)})$ can be computed, for any nonnegative integer d, as $\Delta'_0 = ()$ and $\Delta'_{i+1} = (\Delta'_i, i, \Delta_i)$. For instance, over $K = \mathbb{F}_3$, one would have $\Delta' = (0, 1, 0, 0, 2, 0, 0, 1, 0, 0, 1, 0, 0, 3, \ldots)$.

And as in the non-monic case above, Δ'_d can also be interpreted as the transition sequence of a so-called d-digit κ-ary *monic* Gray code. Starting from 0, the construction $\pi_{i+1} = \pi_i + t^{\delta'_i}$ would indeed end up, after $\frac{\kappa^d-1}{\kappa-1}$ iterations, enumerating all the monic polynomials in $K[t]$ of degree less than d. Finally, using the iteration $\pi_{i+1} = \pi_i + \nu_{\delta'_i}$ instead can then be used to enumerate efficiently all the monic K-linear combinations of any K-free family of monic vectors $(\nu_\ell)_{0\leq\ell<d}$.

Remark: The above technique works when K is a prime field, as its additive group is then cyclic of order κ. In the case where K is not prime but an algebraic extension of a prime field K_0 of size κ_0, a similar technique can be applied. It simply involves considering the $\mathfrak{p}$-lattice as a K_0-vector space—with its associated K_0-basis—instead of just a K-vector space, and using κ_0-ary Gray codes to perform the enumeration. However, attention still has to be paid to the monic part of the enumeration, which then becomes a bit trickier.

3.5 Cofactorization

We did not put a lot of efforts in the implementation of this step. The classical approach, coming back to Coppersmith and Davenport [7] has been used. To test whether a polynomial $P(t)$ is B-smooth, we compute

$$P'(t) \prod_{k=\lceil \frac{B}{2} \rceil}^{B} (t^{\kappa^k} - t) \mod P(t).$$

If $P(t)$ is smooth, this quantity is zero, and the converse is true, unless we are in the unlucky case where there is a large irreducible factor that occurs with a multiplicity that is a multiple of κ. To speed-up the reductions modulo $P(t)$, its preinverse is precomputed once and for all. Then, each product modulo $P(t)$ can be done at a cost of one polynomial product in degree $\deg P$, one short product and one high product (see for instance [12]). At the moment, the special products are implemented as plain products, so that there is room for improvement.

Finally, at least in characteristic 2, the overall cost of cofactorization is small compared to the sieving step. Therefore, we did not yet fully implement a resieving step. For the moment the input given to the cofactorization step is the full norm, in which we do not remove small irreducible factors that have been detected during the sieving step. Again, this leaves room for improvement, especially in characteristic 3 where the cofactorization step takes a large share of the total runtime (see Section 6 below).

3.6 Sub-lattices

Since we want only to consider primitive (i, j) pairs, a classical improvement is to split the (i, j) plane according to the congruence class of i and j modulo one or a few small irreducibles.

Let h be an irreducible polynomial of degree d; then the probability that it divides i is κ^{-d}, and the probability that it divides both i and j is κ^{-2d}. This probability gives the proportion of the sieving space that is useless. To save some sieving time, we can therefore decompose the sieving space into κ^{2d} translates of the sub-lattice corresponding to h, and sieve only $\kappa^{2d} - 1$ of them.

The previous sections about efficient enumeration of the $\mathfrak{p}$-lattices can be easily modified to consider only its intersection with (i, j) pairs that are congruent to some prescribed (i_0, j_0) modulo h. The key observation is that once a first point has been found, the others can

be found by adding the elements of the vector space that has been precisely described. The modifications to make are therefore mostly concerned with the initial position for the enumeration that is no longer the point $(0, 0)$ but a point that is deduced with a few computations modulo h. These computations of the starting point are cheap, but they are to be done for each $\mathfrak{p}$ and for each of the $\kappa^{2d} - 1$ translates of sublattice that we consider. As a consequence, only polynomials h of small degrees d must be considered. However, it is possible to combine several h of small degrees.

The case where this strategy gives the best practical improvement is when the base field K is $\mathbb{F}_2$. Then the 2 irreducible polynomials of degree 1 are t and $t + 1$. Using them, we can sieve only 9/16 of the original sieving space at a reasonable additional cost in the arithmetic cost of the initialization. It was already used by Joux and Lercier for their 2002 record [15]. Going further would require to consider the only irreducible of degree 2, namely $t^2 + t + 1$. This could in theory provide an additional 15/16 shrink factor of the sieving space. On the other hand, this would multiply by 15 the overheads, and with our current implementation, we estimate that it is not worth the effort.

The next interesting case is when $K = \mathbb{F}_3$. The irreducibles of degree 1 are t, $t + 1$ and $t + 2$. Each of them can provide a 8/9 reduction of the sieving space and using all of them could give a 512/729 one. Again, we consider that with our current implementation the potential gain would be overpassed by the overhead. This could however change in the future with further optimization of the small arithmetic building blocks.

For larger base fields, the overall probability for a polynomial to be divisible by an irreducible polynomial h of degree 1 decreases. It makes this strategy less and less attractive. For instance, in [13] where the authors took $K = \mathbb{F}_{27}$, it is not considered.

4 Memory locality issues

When enumerating the elements of a given $\mathfrak{p}$-lattice, the visited positions are scattered across the (i, j) plane. It results in random accesses to the array containing the degree of the norm. For instance, for a kilobit-sized field K, the number of visited positions lies between a few hundreds of million and a few billion, meaning that data locality rapidly becomes an important issue on typical modern-day computers with strongly hierarchized memory.

Being able to access the elements of the lattice in a local fashion is therefore key to avoid cache miss penalties when updating the norm degrees in the (i, j) array. To that intent, we detail in the following paragraphs two techniques designed to improve the spatial locality of memory accesses, one for small primes and the other for large primes.

4.1 Sieving by rows

Let us first consider the case of small prime ideals $\mathfrak{p}$, of degree $L < I$. From Lemma 2, the first $I - L$ vectors $(\mu_k)_{0 \le k < I-L}$ of the basis of the corresponding $\mathfrak{p}$-lattice all have their second coordinates equal to 0. Consequently, all the κ^{I-L} linear combinations of these vectors lie in the same horizontal line in the (i, j) plane. Therefore, if this (i, j) plane is stored as a row-major array, and assuming that at least a single row—*i.e.*, κ^I elements—can fit into the L1 data cache, then memory locality can be drastically improved by sieving row by row. This technique was first described by Pollard in the case of the Number Field Sieve [20].

Furthermore, since the remaining J vectors $(\nu_\ell)_{0 \le \ell < J}$ of the basis all have j-coordinates equal to t^ℓ, sieving rows by increasing lexicographic order on their second coordinates is simply

a matter of enumerating in the same order monic polynomials over K of degrees less than J, then using their coefficients for computing linear combinations of the ν_ℓ's.

However, using the monic Gray code technique presented in Section 3.4 to speed up the enumeration process would break the lexicographic ordering of the j-coordinates. In order to maintain this ordering, we propose a simple change of basis which would allow us to still benefit from the efficiency of Gray code enumeration at the expense of a slight overhead.

Let us then denote by $(\nu'_\ell)_{0\leq\ell<J}$ the family of vectors defined by $\nu'_\ell = \sum_{i=0}^{\ell} \nu_i$. For any $0 \leq \ell < J$, the second coordinate of ν'_ℓ is $\sum_{i=0}^{\ell} t^i$, from which it follows that the ν'_ℓ's form a K-free family, and that $(\mu_0, \ldots, \mu_{I-L-1}, \nu'_0, \ldots, \nu'_{J-1})$ is also a K-basis of the $\mathfrak{p}$-lattice.

One can already note that a non-monic Gray code enumeration of the ν'_ℓ's would go through all linear combinations thereof, sorted by increasing lexicographic order in their j-coordinates. Indeed, intuitively, the second coordinate $\sum_{i=0}^{\ell} t^i$ of ν'_ℓ emulates the propagation of the "carry" through the low-weight coefficients of j, just as a lexicographic enumeration would do right before reaching a polynomial j of t-adic valuation ℓ. For instance, over $K = \mathbb{F}_3$, the sequence $(0, \nu'_0, 2\nu'_0, \nu'_1 + 2\nu'_0, \nu'_1, \nu'_1 + \nu'_0, 2\nu'_1 + \nu'_0, 2\nu'_1 + 2\nu'_0, 2\nu'_1, \nu'_2 + 2\nu'_1, \ldots)$, obtained thanks to a ternary Gray code enumeration, corresponds in fact to a lexicographic enumeration in the j-coordinates.

However, this change of basis is not sufficient when only monic j-coordinates should be considered. One can easily verify that, when applying the previous technique with monic Gray codes, the first enumerated vector to have its second coordinate of degree equal to ℓ will be $\nu_\ell + 2\nu_{\ell-1}$, and not ν_ℓ as required by the lexicographic order. Since this happens only $J-1$ times throughout the whole enumeration of the $\mathfrak{p}$-lattice, a simple fix to tackle this issue is to subtract $2\nu_{\ell-1}$ from the current vector whenever the degree of its j-coordinate has just increased from $\ell - 1$ to ℓ.

Remark: Note that sieving by rows is also compatible with the case when $a_0 - rb_0 \equiv 0 \mod p$, where the only elements of the $\mathfrak{p}$-lattice are those entire rows of the (i, j) plane whose j-coordinates are multiples of p.

4.2 Bucket sieving

When the degree L of the ideal $\mathfrak{p}$ increases, the corresponding lattice positions are much more sparsely dispersed over the (i, j) plane, as at most one hit per row can be expected when $L \geq I$. A different strategy is therefore necessary in the case of large prime ideals.

The main idea is to coalesce the norm updates in the (i, j) array according to their j-coordinates, using an algorithm inspired from bucket sort. Partitioning the sieving area into small groups of consecutive rows, or *bucket regions*, it is possible to go through all the large prime ideals and fill each bucket with all the hits falling into the corresponding bucket region. The actual updating of the norms in the (i, j) array is deferred to a second phase, where bucket regions are processed individually and sequentially. This ensures that all the norm updates corresponding to a same bucket region are applied before moving to the next region, therefore enforcing spatial locality of the memory accesses. This method was described in details by Aoki and Ueda in [4].

There is however a trade-off to keep in mind when dimensioning the buckets: if smaller bucket regions might fit better in the data cache during the norm update phase, this also implies more buckets to fill when enumerating the hit positions, which might end up exhausting the TLB (Translation Lookaside Buffer) of the CPU. Careful tuning is therefore necessary to balance cache misses against TLB misses.

Finally, since this technique forces the sieving area to be processed one bucket region at a time, it is possible to split the other steps of the relation collection algorithm (norm initialization, sieving by rows and cofactorization) so that they also operate on only one bucket region at a time. This way, there is no need to store the whole (i, j) array in memory, but only one bucket region.

5 A library for polynomial arithmetic

5.1 Rationale

Where it be for computing the bases of the `q`-lattices, for initializing the degrees of the norms of the principal ideals in the (i, j) array, for enumerating the elements of the `p`-lattices, or for testing promising ideals for smoothness, polynomial arithmetic over $K[t]$ plays a central role throughout the whole relation collection phase in FFS. Efficient implementation of this arithmetic is therefore crucial, and should not be overlooked.

However, compared to existing, general-purpose libraries such as NTL [22], ZEN [6], or `gf2x` [5], we have the following specific constraints:

- Memory footprint: we need types for polynomials of various fixed sizes, and we cannot allow to loose too much in term of memory. *E.*g., polynomials of degree less than 16 over $\mathbb{F}_2$ should fit in 16 bits, and no more (no padding to 64 bits, no additional integer storing the degree, *etc.*).

- Efficiency: not all operations are critical. The library should have a reasonably fast fall-back for everything, while also allowing one to write specific code for a single operation, such as addition of two ternary polynomials of degree less than 32.

- Compile-time optimization for a given base field K: even if the library should be able to support different base fields, this field is fixed when compiling. This knowledge must be exploited so as to properly inline and optimize all the field-specific low-level primitives, without having to pay for function call overheads at runtime.

To meet all these requirements, we have developed our own library for polynomial arithmetic. Written in C, it defines one type per polynomial size: `fppol16_t`, `fppol32_t`, and `fppol64_t` for 16-, 32-, and 64-coefficient polynomials, respectively. An extra type, `fppol_t`, is defined for multiprecision polynomials. Functions also follow this simple naming scheme: for instance, `fppol32_add` represents the addition of two 32-coefficient polynomials. Heavy use of the C preprocessor allows us to easily define new functions for all supported sizes.

Most of the provided functions are independent of the size of K, and only a few field-specific core functions, such as addition or multiplication by an element of K, should be defined in order to add support for another base field to the library. For the time being, only the fields $\mathbb{F}_2$ and $\mathbb{F}_3$ are supported.

5.2 Representation

The current version of the library only supports bitsliced representation of the polynomials. Therefore, if a single element of K can be represented on k bits (typically, $k = \lceil \log_2 \kappa \rceil$), then an ℓ-coefficient polynomial, of type `fppol`ℓ`_t`, will be represented as an array of k ℓ-bit words.

For $K = \mathbb{F}_2$, this is equivalent to taking the evaluation of the corresponding integer polynomial at 2. Over $\mathbb{F}_3$, an ℓ-coefficient polynomial $p(t) = \sum_{i=0}^{\ell-1} p_i t^i$ will be represented by an array `p` of two ℓ-bit words `p[0]` and `p[1]` such that $p_i = 2\texttt{p[1]}_i + \texttt{p[0]}_i$, where $\texttt{p[}j\texttt{]}_i$ denotes the i-th bit of word `p[`j`]`.

5.3 Basic operations

On top of being quite a compact format (as opposed to packed representations, where extra zeros have to be inserted between coefficients), bitsliced representation allows us to use the bitwise instructions supported by the CPU to perform coefficient-wise operations on the polynomials. For instance, the polynomial addition $r(t) \leftarrow p(t) + q(t)$ over $\mathbb{F}_3[t]$ can be computed in only 7 bitwise operations, as in [11] or, equivalently:

```
   t = (p[0] | p[1]) & (q[0] | q[1]);
r[0] = (p[0] | q[0]) ^ t;
r[1] = (p[1] | q[1]) ^ t;
```

The default implementation of the polynomial multiplication follows a simple serial/parallel (also known as shift-and-add) scheme. It is however possible to plug in optimized functions to accelerate this critical operation. For instance, over $K = \mathbb{F}_2$, one can build the library against `gf2x` and use the faster `gf2x_mul1` function as a drop-in replacement.

Other supported functions include degree computation (using fast leading-zero counting primitives such as GCC's `__builtin_clz`), various conversions between types of different sizes, division, modular reduction, GCD, and so on.

5.4 Enumeration and κ-ary Gray codes

As already mentioned in the previous sections, enumerating (monic) polynomials by increasing lexicographic order is required at various stages of the relation collection process. If this operation is merely incrementing the ℓ-bit word representing the polynomial over $\mathbb{F}_2$, it rapidly becomes trickier over larger base fields. For instance, given $p(t) \in \mathbb{F}_3[t]$, computing the next polynomial $r(t)$ requires 7 operations:

```
  t0 = p[1] + 1;
  t1 = p[1] ^ t0;
  t2 = (t1 >> 1) ^ t1;
r[0] = p[0] ^ t2;
r[1] = (r[0] & t2) ^ t0;
```

This piece of code artificially propagates the "carry" through all the coefficients of $p(t)$ which are equal to 2, starting from the least-significant one. This way, we end up with `t2` representing the integer $2^{\mathrm{ord}_t(r)}$, where $\mathrm{ord}_t(r)$ denotes the t-adic valuation of $r(t)$.

To provide a unified interface for all base fields, stepping through the (monic) enumeration is achieved thanks to the field-specific functions `fppol`ℓ`_set_next` and `fppol`ℓ`_set_next_monic`, respectively.

Enumerating (monic) polynomials in such a way also computes the transition sequence Δ (Δ', respectively) of the corresponding (monic) κ-ary Gray code, as it is the sequence of the t-adic valuations of the successive (monic) polynomials taken in that order. In the previous example for $K = \mathbb{F}_3$, the sequence Δ can be retrieved simply by computing the number of trailing zeros of the successive values of `t2`.

5.5 Addressing the (i, j) array

When sieving by rows or when applying norm updates from a bucket, the accesses to the (i, j) array are mostly random, albeit restricted to a few rows. Converting a given position in the (i, j) plane to an integer offset in the memory representation of this plane is thus critical to the sieving process. It should allow for efficient implementation in order to minimize the computational overhead it incurs, all the while providing a compact integer representation of the polynomials to avoid invalid offsets which would unnecessarily increase the memory footprint of the (i, j) array.

Over the binary field $K = \mathbb{F}_2$, this conversion is trivial, since the bitsliced representation of a polynomial $p(t) \in K[t]$ already corresponds to the integer $\tilde{p}(2)$, where $\tilde{p}(t) \in \mathbb{Z}[t]$ denotes the corresponding integer polynomial. One can then simply address the (i, j) array by offsets of the form $\tilde{p}_{i,j}(2)$, with $p_{i,j}(t) = i(t) + t^I j(t)$.

The situation is however not as simple over $K = \mathbb{F}_3$. Given the representation `p` of a polynomial $p(t) \in \mathbb{F}_3[t]$ as two ℓ-bit words, interleaving the bits of `p[0]` and `p[1]` would correspond to evaluating $\tilde{p}(t)$ at 4, which would result in only $(3/4)^\ell$ of the 2ℓ-bit integers being valid representations of polynomials. On the other hand, evaluating $\tilde{p}(t)$ at 3 would provide us with a compact integer representation, but would be more expensive to compute.

In the current version of our library, we have settled for an intermediate solution between those two extremes: noting that $3^5 = 243 \lessapprox 256 = 2^8$, we can split $p(t)$ into 5-coefficient chunks (or *pentatrits*) and, thanks to a simple look-up table, convert the 10 bits representing each chunk into its evaluation at 3, which then fits on a single octet (8 bits). A ternary polynomial of degree less than d can then be represented using at most $8\lceil d/5 \rceil$ bits (if $d \not\equiv 0 \pmod 5$, the leading pentatrit is not fully used, and its evaluation at 3 will require less than 8 bits). Furthermore, this representation is more compact than using 2 bits for each trit, as $243/256 \approx 0.95$ is much closer to 1 than $3/4$ is.

It is also possible to easily adapt this method to account for monic polynomials by using a specific look-up table for the most-significant pentatrit. Addressing the (i, j) array is then just a matter of converting $p_{i,j}(t) = i(t) + t^I j(t)$ in the same manner.

6 Benchmarks and time estimates

We propose two finite fields $\mathbb{K}$ as benchmarks for our implementation: the fields $\mathbb{F}_{2^{1039}}$ and $\mathbb{F}_{3^{647}}$. Both are prime-degree extensions, so that the improvements of [17] and [13] making use of the Galois action are not available. The sizes of both problems are similar and correspond to the "kilobit" milestone.

We performed a basic polynomial selection, based on an exhaustive search of a polynomial with many small roots. This step would benefit from much further research but this is not the topic of the present paper. So we give without further explanations the polynomials used for this benchmark, where hexadecimal numbers encode polynomials in t represented by their evaluation at 2 (resp. 4) in characteristic 2 (resp. 3):

	$\mathbb{F}_{2^{1039}}$	$\mathbb{F}_{3^{647}}$
f	$x^6 + \texttt{0x7}\,x^5 + \texttt{0x6}\,x + \texttt{0x152a}$	$x^6 + \texttt{0x2}\,x^2 + \texttt{0x11211}$
g	$x + t^{174} + \texttt{0x1ef9a3}$	$x + t^{109} + \texttt{0x1681446166521980}$

It can be checked that the resultants of f and g are polynomials in t that have irreducible factors of degree 1039 and 647, respectively, and are therefore suitable for discrete-logarithm

computations in the target fields.

Parameters. For our choices of polynomials, when taking special-$\mathfrak{q}$ on the g side (the so-called *rational side*), norms have more or less similar degrees on both f and g sides. Therefore we take the same smoothness parameters given in the following table.

Field	$\mathbb{F}_{2^{1039}}$	$\mathbb{F}_{3^{647}}$
Sieving range $I = J$	15	9
Skewness	3	1
Maximum degree of sieved primes (factor base bound)	25	16
Maximum degree of large primes (smoothness bound)	33	21
Threshold degree for starting cofactorization	99	63

Timings. We give the running time of our code with these parameters on a single core of an Intel Core i5 2500 clocked at 3.3 GHz. The code was linked with the `gf2x` library version 1.1 so that we can take advantage of the presence of the `PCLMULQDQ` instruction for fast polynomial multiplication in characteristic 2. For various degrees of special-$\mathfrak{q}$'s, we provide the average time to compute one relation, and the average number of relations per special-$\mathfrak{q}$. Since the number of special-$\mathfrak{q}$'s of a given degree is reliably approximated by the number of monic irreducible polynomials of this degree, this gives enough information to deduce how many relations can be computed in how much time.

$\mathbb{F}_{2^{1039}}$	$\deg q$	26	27	28	29	30	31	32
	Yield (s/rel)	0.59	0.71	0.88	1.07	1.31	1.70	2.05
	Rels per special-$\mathfrak{q}$	32.6	26.2	20.6	16.5	13.3	10.2	8.3

$\mathbb{F}_{3^{647}}$	$\deg q$	17	18	19	20
	Yield (s/rel)	2.24	2.88	3.52	4.71
	Rels per special-$\mathfrak{q}$	23.5	15.7	11.8	8.0

For $\mathbb{F}_{2^{1039}}$, since the large prime bound is set to degree 33, we can give a rough estimate of $2 \times 2^{34}/33 \approx 1.04 \cdot 10^9$ for the number of ideals in both factor bases. Collecting relations up to special-$\mathfrak{q}$'s of degree 30 will provide about $1.19 \cdot 10^9$ relations in about 28 600 days. Taking into account the fact that not all ideals are involved but that there are duplicate relations, this should be just enough to get a full set of relations. Collecting relations for all special-$\mathfrak{q}$'s up to degree 31 will provide about $1.90 \cdot 10^9$ relations in about 51 000 days, which will give a lot of excess and plenty of room to decrease the pressure on the linear algebra. The tuning of this amount of oversieving cannot be done without a linear algebra implementation and is, therefore, out of the scope of this paper. Nevertheless, we expect that in practice the relation collection will be done for a large proportion of special-$\mathfrak{q}$'s of degree 31.

For $\mathbb{F}_{3^{647}}$, similar estimates based on the large prime bound give a theoretical number of ideals of $1.53 \cdot 10^9$. Collecting relations up to special-$\mathfrak{q}$'s of degree 19 will provide about $1.24 \cdot 10^9$ relations in about 45 300 days while going to degree 20 will yield about $2.63 \cdot 10^9$ relations in about 121 000 days.

The yield and the number of relations vary a lot from one special-$\mathfrak{q}$ to the other, even at the same degree. The average values given above have been obtained by letting the program run with special-$\mathfrak{q}$'s of the same degree until we reach a point where the measured yield is statistically within a ± 3 % interval with a confidence level of 95.4 %.

To convert these large numbers into more practical notions, let us assume that we have a cluster of 1000 cores similar to the one we used. Then a full set of relations with enough redundancy can be computed in a bit more than a month for $\mathbb{F}_{2^{1039}}$, and in about 4 months for $\mathbb{F}_{3^{647}}$.

Runtime breakdown. For $\mathbb{F}_{2^{1039}}$ (resp. $\mathbb{F}_{3^{647}}$), we give details on where the time is spent for special-$\mathfrak{q}$'s of degree 30 (resp. degree 19), in percentage and in average number of cycles per (a, b) candidate.

		Norm init	Sieve by rows	Fill buckets	Apply buckets	Cofactor	Total
$\mathbb{F}_{2^{1039}}$	Percent	2.04 %	18.15 %	59.21 %	5.12 %	13.87 %	100 %
	Cyc/pos	1.10	9.73	31.73	2.74	7.43	53.59
$\mathbb{F}_{3^{647}}$	Percent	6.17 %	25.45 %	29.84 %	0.64 %	37.66 %	100 %
	Cyc/pos	43.65	180.11	211.23	4.53	266.53	707.77

From this table, it is clear that we have not yet spent as much time on optimizing the case $K = \mathbb{F}_3$ as we have for $K = \mathbb{F}_2$. We expect that there is a decent room for improvement in characteristic 3. On the other hand, the arithmetic cost is intrinsically higher in characteristic 3 than in characteristic 2: the addition of two polynomials takes 7 instructions instead of just one, and the multiplication in characteristic 2 is implemented in hardware whereas in characteristic 3 this is a software implementation based on the addition. This difference is due to our choice to implement on general purpose CPUs, but would decrease or even disappear if we allowed ourselves to resort to specific hardware.

7 Conclusion

We have presented in this paper a new implementation of the relation collection step for the function field sieve algorithm, which combines state-of-the-art algorithmic, arithmetic and architectural techniques from previous works on both FFS and NFS (where applicable), along with original contributions on several key steps of the algorithm. We intend to release the source code of our implementation under a public license. To the best of our knowledge, this will be the first FFS public code for handling record sizes. We hope this will serve as a reference point for future developments on this subject as well as for dimensioning key-sizes in DLP-based cryptosystems.

Of course, this project is still under heavy development, especially for base fields K larger than $\mathbb{F}_2$. Among other perspectives, we plan to investigate the relevance of delegating some parts of the computation to dedicated hardware accelerators such as GPUs or FPGAs.

Finally, we plan to integrate our code for the relation collection step into a full implementation of FFS, in order to complete the ongoing computations of discrete logarithms over $\mathbb{F}_{2^{1039}}$ and $\mathbb{F}_{3^{647}}$, whence setting kilobit-sized records as new landmarks.

Acknowledgements. The authors wish to thank the other members of the CARAMEL group for numerous interesting discussions regarding preliminary versions of this work, in particular Razvan Barbulescu, Cyril Bouvier, Emmanuel Thomé and Paul Zimmermann.

References

[1] Leonard M. Adleman. The function field sieve. In *Algorithmic Number Theory, First International Symposium, ANTS-I Proceedings*, volume 877 of *LNCS*, pages 108–121. Springer, 1994.

[2] Leonard M. Adleman and Ming-Deh A. Huang. Function field sieve method for discrete logarithms over finite fields. *Inf. Comput.*, 151(1–2):5–16, May 1999.

[3] Kazumaro Aoki, Jens Franke, Thorsten Kleinjung, Arjen Lenstra, and Dag Arne Osvik. A kilobit special number field sieve factorization. In *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 1–12. Springer, 2007.

[4] Kazumaro Aoki and Hiroki Ueda. Sieving using bucket sort. In *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 of *LNCS*, pages 92–102. Springer, 2004.

[5] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. `gf2x`: A library for multiplying polynomials over the binary field. `http://gf2x.gforge.inria.fr/`.

[6] Florent Chabaud and Reynald Lercier. ZEN: A toolbox for fast computation in finite extension over finite rings. `http://zenfact.sourceforge.net/`.

[7] Don Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Transactions on Information Theory*, 30(4):587–594, 1984.

[8] Daniel M. Gordon and Kevin S. McCurley. Massively parallel computation of discrete logarithms. In *Advances in Cryptology – CRYPTO' 92*, volume 740 of *LNCS*, pages 312–323. Springer, 1993.

[9] Robert Granger. Estimates for discrete logarithm computations in finite fields of small characteristic. In *Cryptography and Coding, IMA International Conference Proceedings*, volume 2898 of *LNCS*, pages 190–206. Springer, 2003.

[10] Robert Granger, Abdrew J. Holt, Dan Page, Nigel P. Smart, and Frederik Vercauteren. Function field sieve in characteristic three. In *Algorithmic Number Theory, ANTS VI Proceedings*, volume 3076 of *LNCS*, pages 223–234. Springer, 2004.

[11] Keith Harrison, Dan Page, and Nigel P. Smart. Software implementation of finite fields of characteristic three, for use in pairing-based cryptosystems. *LMS Journal of Computation and Mathematics*, 5:181–193, Jan 2002.

[12] Laszlo Hars. Applications of fast truncated multiplication in cryptography. *EURASIP Journal on Embedded Systems*, 2007(1):61721, 2007.

[13] Takuya Hayashi, Takeshi Shimoyama, Naoyuki Shinohara, and Tsuyoshi Takagi. Breaking pairing-based cryptosystems using η_T pairing over $GF(3^{97})$. Cryptology ePrint Archive, Report 2012/245, 2012. `http://eprint.iacr.org/`.

[14] Takuya Hayashi, Naoyuki Shinohara, Lihua Wang, Shin'ichiro Matsuo, Masaaki Shirase, and Tsuyoshi Takagi. Solving a 676-bit discrete logarithm problem in $GF(3^{6n})$. In *Public Key Cryptography – PKC 2010*, volume 6056 of *LNCS*, pages 351–367. Springer, 2010.

[15] Antoine Joux and Reynald Lercier. The function field sieve is quite special. In *Algorithmic Number Theory, ANTS V Proceedings*, volume 2369 of *LNCS*, pages 343–356. Springer, 2002.

[16] Antoine Joux and Reynald Lercier. Discrete logarithms in $GF(2^{607})$ and $GF(2^{613})$. Posting to the Number Theory List, September 2005. `https://listserv.nodak.edu/cgi-bin/wa.exe?A2=ind0509&L=NMBRTHRY&F=&S=&P=19612`.

[17] Antoine Joux and Reynald Lercier. The function field sieve in the medium prime case. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 254–270. Springer, 2006.

[18] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *LNCS*, pages 333–350. Springer, 2010.

[19] Ryutaroh Matsumoto. Using C_{ab} curves in the function field sieve. *IEICE Trans. Fundamentals*, E82-A(3):551–552, Mar 1999.

[20] John M. Pollard. The lattice sieve. In Arjen K. Lenstra and Hendrik W. Lenstra, editors, *The development of the number field sieve*, volume 1554 of *LNM*, pages 43–49. Springer, 1993.

[21] Oliver Schirokauer. Discrete logarithms and local units. *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, 345(1676):409–423, 1993.

[22] Victor Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl/`.

[23] Emmanuel Thomé. Computation of discrete logarithms in $\mathbb{F}_{2^{607}}$. In *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 107–124. Springer, 2001.