Discrete logarithm in GF(2^{809}) with FFS

Razvan Barbulescu, Cyril Bouvier, Jérémie Detrey, Pierrick Gaudry, Hamza Jeljeli, Emmanuel Thomé, Marion Videau, Paul Zimmermann

Discrete logarithm in $\mathrm{GF}(2^{809})$ with FFS

Razvan Barbulescu, Cyril Bouvier, Jérémie Detrey, Pierrick Gaudry,
Hamza Jeljeli, Emmanuel Thomé, Marion Videau, and Paul Zimmermann

CARAMEL project-team, LORIA, INRIA / CNRS / Université de Lorraine,
Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France
`<first-name>.<last-name>@loria.fr`

Abstract. The year 2013 has seen several major complexity advances for the discrete logarithm problem in multiplicative groups of small-characteristic finite fields. These outmatch, asymptotically, the Function Field Sieve (FFS) approach, which was so far the most efficient algorithm known for this task. Yet, on the practical side, it is not clear whether the new algorithms are uniformly better than FFS. This article presents the state of the art with regard to the FFS algorithm, and reports data from a record-sized discrete logarithm computation in a prime-degree extension field.

Keywords: Discrete logarithm, Function field sieve, Cryptanalysis, Number theory

1 Introduction

The discrete logarithm problem (DLP) is the cornerstone of a large part of public-key cryptography. Multiplicative groups of finite fields were the first proposed groups for cryptography, meeting the requirements of fast arithmetic and a hard discrete logarithm problem. Since almost the beginning though, the discrete logarithm problem is known to be of subexponential complexity in these groups, with the most efficient algorithms being those from the Number Field Sieve family, of complexity $L_{\#G}(1/3, c)$ for computing discrete logarithms in $G = \mathrm{GF}(p^n)^\times$, where we use the conventional notation

$$L_{\#G}(\alpha, c) = \exp\left((c + o(1))(\log \#G)^\alpha (\log\log \#G)^{1-\alpha}\right).$$

To this regard, alternatives to these choices of groups, such as elliptic curve cryptography, are of course considered. With elliptic curves, a smaller group size allows for the same security against potential attackers. In spite of the existence of such alternatives, multiplicative groups of finite fields remain of primary importance for cryptography, because of their widespread use in many protocols and their implementations, such as the Diffie–Hellman key exchange [12], the ElGamal encryption system [13] or the Digital Signature Algorithm [26]. Another typical situation where the DLP in multiplicative groups of finite fields is of crucial importance is the area of pairing-based cryptography.

We focus here on multiplicative groups of finite fields of small characteristic, the archetypal examples of which being the binary fields $\mathrm{GF}(2^n)$. These fields offer some implementation advantages for the cryptographer, which make them a natural choice (the most usually chosen fields being prime fields). Invented by Adleman [2] in 1994, the Function Field Sieve has been the most efficient algorithm known to solve the DLP in finite fields of small characteristic for quite a long time. It earned its first successes from 2001 onwards [17,18], the latest record computations being in the fields $\mathrm{GF}(2^{613})$ [21] and $\mathrm{GF}(2^{619})$ [8].

In 2013, several works have significantly changed the landscape and improved the asymptotic complexity for solving the DLP in small-characteristic finite fields. Successive improvements lowered the complexity from $L(1/3)$ down to $L(1/4 + o(1))$ [19], and later down to quasi-polynomial complexity [9]. A common feature of these improved algorithms is their reliance on an appropriately sized subfield. Hence, the new algorithms perform well for finite fields which readily provide such a subfield. All published DLP records which have used these improved algorithms satisfy this criterion, the culminating computation to date being in the field $\mathrm{GF}(2^{6168}) = \mathrm{GF}((2^{24})^{257})$. On the other hand, finite fields which have no such subfield, such as fields of prime extension degree $\mathrm{GF}(2^p)$, are less amenable to the newer approaches. One must first apply the new algorithms to the DLP in an extension $\mathrm{GF}(2^{kp})$ for an appropriate k, and only then solve the DLP in $\mathrm{GF}(2^p)$ as a by-product. Complexity analysis shows that this approach is asymptotically faster than FFS, despite the large intermediary field $\mathrm{GF}(2^{kp})$.

Motivation. In practical terms, though, determining the cut-off point where FFS is surpassed by the new methods is not obvious. No practical data exists as of yet where the $L(1/4+o(1))$ or quasi-polynomial algorithms have been applied to fields of prime extension degree. The FFS approach, on the other hand, is complicated enough so that its runtime is difficult to extrapolate from previous experiments, all the more so given tha experimental data with FFS is quite scarce (the computation from [21] is 8 years old, and the report [8], while doing better than the former, involved only a small amount of resources). One of the motivations of the present article is to provide new record computation data for fields of prime extension degree by computing discrete logarithms in $\mathrm{GF}(2^{809})$.

Another line of work made important by the recent developments in the computation of discrete logarithms in small characteristic finite fields is the assessment of their reach in the context of pairing-based cryptography, as studied for instance by the preprint [1]. Indeed, the fields of definition of pairings on elliptic curves and Jacobians of genus-2 curves offer almost by construction the required criterion for the new algorithms to apply optimally, or fall close at the very least. Re-assessing the security levels for finite fields which were once thought to offer strong DLP hardness calls again for a serious FFS study, because of its potential use in this context. The new algorithms allow to do the so-called descent step all the way down to tiny elements, much smaller than what any FFS/NFS-DL variant can reach. For best performance, it is perfectly reasonable to organize this descent by combining several algorithms, as proposed in [1]. It is expected that the early steps of the descent are best done with the same techniques as for in the FFS algorithm, including in particular the sieving technique.

It is an important feature of our work that all software used is original work freely available for download, the main component being the `cado-nfs` software suite [6] (although `cado-nfs` originally focuses on the Number Field Sieve, recent additions cover FFS as well). The development of the present work has led to several preprints and articles providing detailed insights on the improvements of the individual steps of FFS [7,10,11,16]. For brevity, this article does not repeat these details, and the interested reader is referred to these articles for more detail.

Roadmap. This article is organized as follows. Section 2 recalls classical facts about the FFS algorithm. Section 3 gives the details about the different steps involved in the computation of discrete logarithms in (a prime order subgroup of) GF(2^{809})$^\times$. Section 4 discusses ways to optimize the computation cost by balancing the sieving and linear algebra step. Section 5 explores the cost of solving the discrete logarithm problem in a kilobit-sized field, namely GF(2^{1039})$^\times$.

2 A brief overview of FFS

The goal of this section is to provide the reader with a primer on FFS, so that they can have a better insight into the various steps of the algorithm along with the computations they involve. However, for brevity purposes, this presentation is in no way complete nor exhaustive. For more details, we refer the interested reader to the extensive literature on this topic, starting from the first theoretical articles [2,3,24] to the algorithmic advances and implementation reports which followed [18,14,15,7,11,10,16].

2.1 Index calculus

The Function Field Sieve algorithm belongs to the family of so-called *index calculus* methods for computing discrete logarithms in a cyclic group G.

Writing G multiplicatively and assuming that it is generated by an element g of prime order $\ell = \#G$, the core idea is to collect many equalities, or *relations*, of the form

$$\prod_i \alpha_i^{e_i} = 1,$$

where the elements α_i all belong to a predefined subset of G called the *factor base*, and where the exponents e_i all lie in $\mathbb{Z}/\ell\mathbb{Z}$. When taking the discrete logarithm in base g, each relation then yields a linear equation in $\mathbb{Z}/\ell\mathbb{Z}$,

$$\sum_i e_i \log_g(\alpha_i) \equiv 0 \pmod{\ell},$$

whose unknowns are the discrete logarithms of the elements α_i of the factor base. This is known as the *relation collection* step.

Once enough such relations have been found (typically, more than the size of the factor base), one can solve the corresponding system of linear equations and retrieve the logarithms of the factor base elements. This step is usually referred to as the *linear algebra* step. It is often directly preceded by a *filtering* step, whose chief purpose is to prepare the matrix from the collected relations; some simplifications and Gaussian elimination are typically performed at this stage.

Finally, the discrete logarithm of a given element $h \in G$ is computed thanks to the last step, known as the *individual logarithm* step. The idea here is to rewrite h as a product $\prod_i \alpha_i^{f_i}$ of elements of the factor base. The discrete logarithm of h is then

$$\log_g(h) \equiv \sum_i f_i \log_g(\alpha_i) \pmod{\ell}.$$

2.2 FFS-specific considerations

In the case of FFS, one can detail further the various steps of the index calculus algorithm, mostly due to the fact that G is the multiplicative subgroup of a finite field $\mathrm{GF}(p^n)$, where the characteristic p is a small prime (*e.g.*, $p = 2$ in this work).

Relation collection. Let us write k to denote the base field $\mathrm{GF}(p)$, and let f and g be two monic polynomials in $k[t][x]$ such that their resultant in x contains an irreducible factor $\varphi(t)$ of degree n. The key to collecting relations in FFS is then based on the following commutative diagram, whose maps are ring homomorphisms:

$$\begin{tikzcd}
 & k[t][x] \arrow[dl] \arrow[dr] & \\
k[t][x]/f(x,t) \arrow[dr] & & k[t][x]/g(x,t) \arrow[dl] \\
 & k[t]/\varphi(t) \cong \mathrm{GF}(p^n) &
\end{tikzcd}$$

Let us now consider an element of the form $a(t) - b(t)x \in k[t][x]$. Following the maps on the left-hand side of the diagram, we first obtain an element of $k[t][x]/f(x,t)$, which can be viewed as a principal ideal in the ring of integers of the corresponding function field $k(t)[x]/f(x,t)$. As this is a Dedekind domain, there is a unique factorization in prime ideals. The same also applies to the right-hand side of the diagram, corresponding to the function field $k(t)[x]/g(x,t)$, thus yielding two factorizations of the same element $a(t) - b(t)x$. Strictly speaking, there is no reason for $k[t][x]/f(x,t)$ to be the full ring of integers of $k(t)[x]/f(x,t)$, but this technicality can easily be dealt with after a study of the singularities of the curve of equation $f(x,t) = 0$.

When pushing these two factorizations into the finite field $k[t]/\varphi(t) \cong \mathrm{GF}(p^n)$, we then obtain an equality between two products of elements of the finite field. Should all these elements belong to the factor base, we would then have computed a *relation*.

In FFS, there are in fact two factor bases: one for each side of the diagram. Indeed, when considering the principal ideal corresponding to $a(t) - b(t)x$ in $k(t)[x]/f(x,t)$, we will say that it is *smooth* if its factorization involves only prime ideals whose norms have degree less than or equal to a parameter called the *smoothness bound*. Therefore, the factor base on this side will correspond to the prime ideals $\langle q(t), x - r(t)\rangle$ of $k(t)[x]/f(x,t)$ for which $\deg(q)$ is less than or equal to this smoothness bound. In general, by degree of an ideal of the factor base we understand the degree of its norm. The factor base for the g side can be constructed in a similar fashion.

All in all, finding relations in FFS amounts to looking for elements of the form $a(t) - b(t)x$ in $k[t][x]$ whose corresponding images in $k[t][x]/f(x,t)$ and $k[t][x]/g(x,t)$ are both smooth. Whether or not these images factor nicely can be decided by considering the factorizations of the polynomials $\mathrm{Res}_x(a(t) - b(t)x, f(x,t))$ and $\mathrm{Res}_x(a(t) - b(t)x, g(x,t))$. These resultants are commonly referred to as *norms* because of the link to the norm of the ideal generated by $a(t) - b(t)x$ in the two rings.[1]

The relation collection process can be greatly accelerated by using sieving techniques. Another important acceleration can be achieved thanks to the so-called *sieving by special-*$\mathfrak{q}$ technique. It relies on the fact that one can easily restrict the relation collection to elements $a(t) - b(t)x$ which are divisible by an arbitrary prime ideal $\mathfrak{q}$ on the (say) f side. This way, when considering $a(t) - b(t)x$ over $k[t][x]/f(x,t)$, one knows that the corresponding principal ideal is divisible by $\mathfrak{q}$, thus lowering by $\deg(\mathfrak{q})$ the degree of the remaining part and therefore increasing the probability that it is smooth. One can then apply this technique for many special-$\mathfrak{q}$'s of degree below the smoothness bound in order to achieve a large speed-up.

Polynomial selection. In order to further increase the probability to find doubly-smooth elements $a(t) - b(t)x$ all the while keeping arithmetic computations to a minimum in the relation collection step, one has to pay extra care to the selection of the polynomials f and $g \in k[t][x]$. Similar to the case of the Number Field Sieve (NFS), several criteria exist in the literature in order to rate polynomials to be used in FFS [7].

A typical choice of polynomials involves a g which is linear in x, but many degrees of freedom remain. Due to the bad impact a poor choice of polynomials can have on the relation collection step, it is deemed important to dedicate some

[1] As in the case for the Number Field Sieve, it is possible to generalize the setting somewhat by allowing non-monic polynomials f and g. In that case the norms and the resultants do not coincide exactly. This is not a concern for the present work.

CPU time to carefully select good polynomial pairs. This preliminary step is usually known as the *polynomial selection* step.

Individual logarithms. In the FFS context, individual logarithms are computed thanks to the *descent* step, which reuses several key aspects of the relation collection step. Indeed, the sieving methods developed for the latter can help us rewrite "large" elements (*i.e.*, of high degree) of $\mathrm{GF}(p^n)$ into products of smaller elements. This way, starting from h and recursively working our way down, we end up considering only products of elements of the factor base, all of whose discrete logarithms are known after the linear algebra step. We can then backtrack this descent and reconstruct the discrete logarithms of the larger elements and, ultimately, of h.

3 Discrete logarithm computation in GF(2^{809})

In this work, we have chosen to attack the discrete logarithm problem in a subgroup which is not $\mathrm{GF}(2^{809})^\times$ itself, but rather one of its prime-order subgroups, namely the subgroup of prime order ℓ, where ℓ is the 202-bit prime factor of $2^{809} - 1$ given by

$$\ell = 4148386731260605647525186547488842396461625774241327567978137.$$

The other factor of $2^{809} - 1$ is also prime, and 608 bits long.

The motivation for this choice is related to the cryptographic applications, for which the discrete logarithm problem is to be solved only in a subgroup whose size is large enough to resist Pollard's ρ attack. We recall, as a comparison, that the original DSA (digital signature algorithm) setup recommends a 160-bit prime order subgroup in the multiplicative group of a 1024-bit finite field [26]. Here, the chosen subgroup is rather over-sized than under-sized, given the expected difficulty of Pollard's ρ attack on a 202-bit group.

Extrapolations from the hardness of our discrete logarithm computation to the hardness of the full discrete logarithm computation in the group $\mathrm{GF}(2^{809})^\times$ are easy to obtain, as one can get satisfactory estimates by keeping most timings unchanged, and scaling the linear algebra cost by a roughly 4-fold linear factor (the complexity of the linear algebra step being dominated by the cost of multiplying an integer modulo ℓ by a word-size integer).

In the following sub-sections, we discuss the choices of parameters made for this computation, and present benchmarks and estimates for the various steps of the FFS algorithm.

Notations. For the sake of compactness, in the following, we represent a polynomial of $\mathrm{GF}(2)[t]$ by the integer obtained when considering the polynomial over $\mathbb{Z}$ then evaluating it at $t = 2$; we write this integer in hexadecimal so that sparseness is still visible. For instance, `0x11` represents $t^4 + 1$, and `0xb00001` is $t^{23} + t^{21} + t^{20} + 1$.

3.1 Polynomial selection

We used the criteria defined in [7] to select the polynomials. It appears that monic polynomials $f(x,t)$ and $g(x,t)$ of degree 6 and 1 in x, respectively, were the best choice. We therefore computed the α value—a quantity that is similar to the one used when estimating the quality of NFS polynomials [25]—for all the irreducible polynomials of degree 6 in x for which the degree in t of the coefficient of x^i is at most $12-2i$. Relying on sieving techniques for acceleration purposes, this computation took about 2,760 hours on one core of an Intel Core i5-2500 CPU running at 3.3 GHz. We see it as a pre-computation since, when associated with different polynomials g, f can be used to compute discrete logarithms in any field GF(2^n) with n ranging roughly from 700 and 900.

With this setting, the degree of the resultant in x of f and g is 6 times the degree in t of the constant coefficient of g. Since we want this resultant to have an irreducible factor of degree 809, this imposes this constant coefficient to have a degree at least 135. According to [7], our choice may be driven by the α value and by the factorization pattern of the resultant. As it turns out, among the few polynomials $f(x,t)$ that were preselected for having an α value around -6, all but one have arithmetic properties that force a small factor of degree 2 in the resultant, independently of the linear polynomial g. For those polynomials, the degree in t of the constant coefficient of g would have to be at least 136 in order to leave enough room for an irreducible factor of degree 809 to exist in the resultant. For this reason, we chose for f the only polynomial that we have found to have a nice α-value and that forces only a factor of degree 1 in the resultant, which is expected to be better [7, §3.3]:

$$f(x,t) = x^6 + \texttt{0x7}x^5 + \texttt{0x6b}x^3 + \texttt{0x1ab}x^2 + \texttt{0x326}x + \texttt{0x19b3}.$$

As far as $g(x,t)$ is concerned, no special care was taken for its selection, as it can be found in a few seconds and it did not make any difference in the subsequent computations. We therefore picked a linear monic polynomial whose constant term (with respect to x) is a sparse polynomial of degree 135 in t:

$$g(x,t) = x + \texttt{0x8000000000000000000000000000001e7eaa}.$$

The resultant in x of $f(x,t)$ and $g(x,t)$ is $(t+1)$ times an irreducible factor $\varphi(t)$ of degree 809 that we take as defining polynomial for GF(2^{809}):

$$\begin{aligned}\varphi(t) = {} & \texttt{0x3ffffffffffffffffffffffffffffffffe80000000000000000000000} \\ & \texttt{0cc0cfaeb0e000000000000000000000000000000004dfffffffffffffc} \\ & \texttt{3c3ffc3c3c3fce3c2133ffffffffffffe9697fe96804c84c97e0b0000c0} \\ & \texttt{0cf9b0f7675354e79f4cf7c97e29}.\end{aligned}$$

This choice of polynomials was driven solely by the efficiency of the relation collection. The genus of the curve corresponding to $f(x,t) = 0$ is 19, which is not especially low. The class number is 2073600, which is rather large, and there are some singular points. The only ones that we have to take care of for the computation are the ones at $(x,t) = (\omega,\omega)$, where $\omega^2+\omega+1 = 0$, which splits into 2 places, and of course its conjugate. All these complications have essentially no influence on the running time but require some care in the implementation.

N.B. Since g was chosen to be linear in x, the corresponding function field $\mathrm{GF}(2)(t)[x]/g(x,t)$ is a rational function field. Therefore, to remain consistent with the established terminology, we will refer to the corresponding side (*i.e.*, the right-hand side in the commutative diagram of Section 2.2) as the *rational side*. Conversely, the f side of the diagram will be referred to as the *algebraic side*.

3.2 Relation collection

The relation collection was performed using the implementation described in [11]. This is a rather classical sieving method using lattice-sieving for various special-$\mathfrak{q}$'s. We actually ran the relation collection step for two different sets of parameters, in order to compare and be able to see how the tuning of this phase influences the filtering and linear algebra steps.

The terminology used in [11] is the classical one, where the *factor base bound* is the limit for the degree of the irreducible polynomials that are sieved, and the *large-prime bound* refers to the limit for the degree of the polynomials allowed in a relation (*i.e.*, the smoothness bound). These notions are very similar to the ones used when sieving relations for factoring integers using NFS (see for instance [23]); irreducible polynomials of a given degree playing the role of prime numbers of a given number of bits, we keep the "large-prime" terminology for convenience. Likewise, the discussion below uses terminology which is heavily borrowed from NFS implementation folklore. In particular, the I and J parameters directly relate to the dimensions of the sieved area in what is customarily called the (i, j)*-plane* in the lattice sieving context. Typically, the number of $a(t) - b(t)x$ polynomials considered for a given special-$\mathfrak{q}$ is given by 2^{I+J}.

For the two parameter sets that we considered, we used a skewness of 2 (the skewness being the degree gap in the coefficients $a(t)$ and $b(t)$ of the functions $a(t) - b(t)x$ considered in the algorithm), and we only considered the special-$\mathfrak{q}$'s on the rational side, since we are in a range where the rational side yields larger norms than the algebraic side. We used a factor base bound of degree 23 (inclusive). The main difference between our two sets of parameters is the large-prime bound.

Case 1: large-prime bound of 27. In that case, we used $I = J = 15$, which were chosen to yield enough relations per special-$\mathfrak{q}$, despite a low large-prime bound. Only $(a(t), b(t))$ pairs for which the norms on both sides (once removed the factors obtained from sieving) have degree at most 81 were actually considered as candidates for a complete factorization attempt. In other words, we allowed for three large primes of maximum degree on each side.

All the special-$\mathfrak{q}$'s of degree 24 to 27 (inclusive) were sieved, producing a bit more than 52 million relations (possibly non-unique). The relevant data is summarized in the following table. The running times are given for a single core of an Intel Core i5-2500 CPU running at 3.3 GHz (Sandy Bridge microarchitecture). In particular, this assumes the presence of the PCLMULQDQ instruction for carry-less multiplication. In practice, most of our computations were done using the idle time of a cluster[2] whose 4-year old processors do not support this instruction, and therefore took twice as much time. On the other hand, we expect faster runtime on Intel's new Haswell processor family that provides a much better implementation of the PCLMULQDQ instruction than the Sandy Bridge and Ivy Bridge microarchitectures.

deg $\mathfrak{q}$	number of rels	s/rel	rels/sp-$\mathfrak{q}$	accumulated rels	accumulated time
24	6,940,249	1.48	9.93	6,940,249	2,853 h
25	9,926,294	1.91	7.39	16,866,543	8,119 h
26	14,516,775	2.42	5.62	31,383,318	17,877 h
27	20,645,456	3.38	4.15	52,028,774	37,260 h

Case 2: large-prime bound of 28. In that case, we used $I = J = 14$, which was enough to get a decent rate of relations per special-$\mathfrak{q}$. The threshold was again set to 3 times the large-prime bound, that is, 84 for both sides. We sieved all the special-$\mathfrak{q}$'s of degree 24 to 28 (inclusive), and produced more than 117 million relations, distributed as in the following table:

deg $\mathfrak{q}$	number of rels	s/rel	rels/sp-$\mathfrak{q}$	accumulated rels	accumulated time
24	9,515,069	0.41	13.61	9,515,069	1,083 h
25	13,816,908	0.54	10.29	23,331,977	3,155 h
26	20,538,387	0.65	7.95	43,870,364	6,863 h
27	29,652,781	0.86	5.96	73,523,145	13,946 h
28	43,875,232	1.07	4.57	117,398,377	26,986 h

In both cases, we obtained a number of relations that provided a reasonable excess (the excess is defined as the difference between the number of unique relations collected and the number of ideals involved).

3.3 Filtering

The filtering step is split in three stages:

- duplicate: remove duplicate relations from the relation collection step;
- purge: remove singletons (ideals that appear in only one relation) and remove relations while the excess is positive (*i.e.*, there are still more relations than ideals);
- merge: beginning of Gaussian elimination.

The filtering step was performed using the implementation described in [10]. It was run identically on the two sets of relations produced by the relation collection step.

[2] We acknowledge the support of the Région Lorraine and the CPER MISN TALC project who gave us access to this cluster.

Case 1: large-prime bound of 27. In total, 52,028,774 relations were collected. They produced 30,142,422 unique relations (42 % were duplicates). After the first singleton removal, about 29M relations remained as well as 19M ideals (so the excess was around 10M). At the end of the purge algorithm, there were 9.6M relations and as many ideals. The final matrix (after the merge algorithm) had 3.68M rows and columns (with, in average, 100 non-zero coefficients per row, which is close to optimal for our linear algebra implementation).

Case 2: large-prime bound of 28. In total, 117,398,377 relations were collected. They produced 67,411,816 unique relations (43 % duplicates). After the first singleton removal, about 65M relations remained as well as 37M ideals (so the excess was around 28M). At the end of the purge algorithm, there were 13.6M relations and as many ideals. The final matrix (after the merge algorithm) had 4.85M rows and columns (with, in average, 100 non-zero coefficients per row).

For the actual computation, relations collected with both values of the large-prime bound were considered to produce the matrix. This is of course not "fair", in the sense that if the computation were to be run again, we would have only one of the two relation sets. On the other hand, it was a pity not to use all what we had at hand to reduce the cost of the linear algebra.

In this setting, starting from an input set of 78.8M unique relations, we obtained a matrix with 3,602,667 rows and columns, and an average of 100 non-zero coefficients per row.

3.4 Linear algebra

The linear system to be solved is of the form $Mw = 0$, where M is the sparse matrix produced by the filtering step. We solved the linear system modulo the subgroup of order ℓ, which is a 202-bit prime, using the Wiedemann algorithm as a solver. This algorithm iterates a very large number of sparse-matrix–vector products of the form $v' \leftarrow Mv$, where v and v' are dense vectors.

The computation was carried out on NVIDIA GPUs. The implementation used Residue Number System (RNS) arithmetic to accelerate arithmetic over $\mathbb{Z}/\ell\mathbb{Z}$, since this representation system offers the opportunity to increase the parallelism between the computational units, and is well suited to the GPU execution framework. This approach is described in details in [16].

This linear algebra step was actually computed twice on two different setups, whose choice was driven by the hardware which was available to us at that time. A CPU implementation was also developed to serve as a reference point.

Dual-GPU setup. A simple Wiedemann algorithm was run on a single node equipped with two NVIDIA GeForce GTX 680 graphics processors. On this setup, the sparse-matrix–vector product takes 144 ms, including 16 ms for inter-GPU communication. The total computation time sums up to 18 days: 12 days on both GPUs for computing the initial sequence, 35 minutes for computing the minimal polynomial, and 6 days on both GPUs for computing a vector of the kernel.

Eight-GPU setup. Another option was tried, using a different computing facility[3] equipped with slightly different hardware. We used four distinct nodes, each equipped with two NVIDIA Tesla M2050 graphics processors, and ran the block Wiedemann algorithm [22] with blocking parameters $m = 8$ and $n = 4$. An iteration took 169 ms on each node, with 27 ms for the inter-GPU communication. The initial sequence computation required 2.6 days in parallel on the 4 nodes. The linear generator computation required 2 hours in parallel using 16 jobs on a 4-node cluster with Intel Xeon E5-2609 CPUs at 2.4 GHz connected with Infiniband QDR network. Finally, computing the kernel vector required 1.8 days in parallel on the 4 GPU nodes. All in all, this computation took a total wall-clock time of about 4.5 days.

CPU implementation. For comparison purposes, we also developed a software implementation of block Wiedemann based on the RNS representation for the arithmetic and using the SSE-4.2 and AVX SIMD instruction sets. We exploited the data-level parallelism in order to multiply the sparse matrix by several vectors in parallel. For instance, using 64-bit RNS moduli, a 128-bit SSE register can hold two RNS residues coming from two distinct vectors, thus allowing us to multiply the matrix by these two vectors in parallel. The SSE implementation offers the advantage of using integer arithmetic while AVX only supports packed floating-point arithmetic.

The experiment was run on a cluster of Intel Core i5-2500 CPU running at 3.3 GHz connected with Infiniband QDR network. Each node of the cluster contains 4 cores. The matrix was split into 4 balanced parts and distributed over 4 threads running on a single node. With the SSE-4.2 implementation, an iteration requires 5 seconds to compute the product of M by two vectors, while the AVX version computes the product with four vectors in 12.1 seconds. The SSE implementation using integer arithmetic ends up being the fastest, despite its lower data parallelism. Even though we did not run the full computation of the linear algebra, our SSE implementation running on 4 nodes with the blocking parameters $m = 16$ and $n = 8$ should take a total of 68.4 days, or 26,267 core-hours (15,120 core-hours for computing the initial sequence, 1,067 core-hours for the linear generators, and 10,080 core-hours for computing a kernel vector).

From the non-zero kernel vector coming out of the linear algebra step, we obtained the discrete logarithms of 39,319,911 elements of the factor base, among which 98.6 % of the degree-28 ideals.

[3] This work was realized with the support of HPC@LR, a Center of Competence in High-Performance Computing from the Languedoc-Roussillon region, funded by the Languedoc-Roussillon region, the European Union and the Université Montpellier 2 Sciences et Techniques. The HPC@LR Center is equipped with an IBM hybrid Supercomputer.

3.5 Descent

Once the discrete logarithms of almost all elements up to the large-prime bound have been found, we compute individual logarithms using the classical strategy of descent by special-`q`.

More precisely, we start by splitting the target element into the quotient of two elements of about half the degree, using an Euclidean algorithm that we stop in the middle. Randomizing the target allows to repeat that step until the two elements are smoother than average. In our case, after a dozen of minutes, we managed to rewrite the target in terms of elements of degree at most 90 (in comparison, straight out of the Euclidean algorithm, we have a numerator and a denominator whose degree is at most 405).

Then we “descended” these elements of degree less than or equal to 90 but above 28, by considering them as special-`q`’s in the relation collection implementation, so that they are rewritten as ideals of smaller degree. Hence a tree is built where the discrete logarithm of a node can be deduced from the discrete logarithms of each of its children, which are of smaller degree. At the end of this process, one of the degree-28 ideals involved in the tree was not known from the linear algebra step, and was therefore “re-descended” to known degree-28 ideals.

The overall cost of the individual logarithm step is less than one hour, and therefore was not thoroughly investigated.

There are cases, however, where the cost of the descent step could become critical. First, we could imagine that if such a finite field is used in real life for securing communications, then the number of individual logarithms to solve becomes very large. Besides this admittedly highly hypothetical scenario, it is important to take into account the new discrete logarithm algorithms where the bottleneck is the descent step. For instance, in [1], the estimates for computing a discrete logarithm in $\mathrm{GF}(3^{6\times 509})$ is around 2^{74} operations, with about half of the time spent in the Euclidean algorithm and classical descent steps which are exactly the same algorithms as in FFS.

As was already performed in [1], in order to optimize our descent implementation, it would be necessary to make a careful study of the cost required to descend an element of a given degree down to the large-prime bound. Indeed, once we know the average costs to fully descend elements of degree less than a given integer d, it is possible to deduce an estimate for the cost of a complete descent of a polynomial of degree d. For this, we run a few special-`q` sievings for random ideals of degree d and, for each relation, we compute the corresponding cost to go down to the large-prime bound. It is then possible to tune the parameters of the sieving for a special-`q` of degree d in order to optimize the expected total time to fully descend it. Then we can continue with degree $d+1$, and so on. This approach is for instance implemented in the NFS context in the `cado-nfs` software package [5].

As for the Euclidean step, it is also possible to optimize it using the sieving strategy explained in [20]. It is presented in the case of the Number Field

Sieve, but it applies *mutatis mutandis* to FFS after having replaced integers with polynomials.

3.6 Computation result

Since it is a generator of the order-ℓ subgroup of GF$(2^{809})^{\times}$, we considered the element t as the basis for the discrete logarithm computations. Then, the logarithms of the elements of the factor base were readily available after the linear algebra step. For instance,

$$\begin{aligned}\log_t(t+1) \equiv\ & 107082105171602535431582987436 \setminus \\ & 798986525914273094868488570257 4 \pmod{\ell}.\end{aligned}$$

As an illustration of the descent step, we computed the discrete logarithm of a "random" element. We decided to step away from the tradition of taking the logarithm of decimals of π, and took as a "random" input the integer RSA-1024 (converted into a polynomial of degree 1023 in t using the same encoding as above). It was reduced modulo $\varphi(t)$ before taking its discrete logarithm. We then obtain

$$\begin{aligned}\log_t(\text{RSA-1024}) \equiv\ & 299978707191164348545002008342 \setminus \\ & 083497715498790833812541647079 6 \pmod{\ell}.\end{aligned}$$

A short Magma script is given in Appendix for verification purposes.

4 Balancing sieving and linear algebra

In retrospect, it is now clear that the strategy of using a large-prime bound of 27 is better than 28: in the same amount of sieving time, one obtains a post-merge matrix that is smaller.

The question of where to stop sieving is not so easy to answer in advance, but with the data that we have collected, we can give some hints for future choices.

With this objective in mind, we ran the filtering step for various numbers of relations (always produced with a large-prime bound of 27), and estimated both the sieving time for getting these relations, along with the linear algebra time for the corresponding matrix. The relations were added in increasing lexicographical order of the special-$\mathfrak{q}$'s. We estimate the cost of the linear algebra step as the product of the size by the total weight of the matrix, which is theoretically proportional to the running time. Using this metric, the linear algebra step described in Section 3.4 has a cost of about 1298 and corresponds to 36 days on one GTX 680 GPU (this running time is the same for both the dual- and the eight-GPU setups). Estimates for the CPU running time of the linear algebra step are also reported, based on our software reference implementation.

# rels	matrix size after singleton removal	matrix size after merge	lin. alg. cost	sieve CPU time ($\times 10^3$ h)	linear algebra GPU time ($\times 10^3$ h)	linear algebra CPU time ($\times 10^3$ h)
27.7M	14.1M × 14.0M	4.99M	2493	15.4	1.65	55.4
31.3M	16.6M × 15.1M	4.46M	1995	17.8	1.32	40.4
33.9M	18.6M × 16.1M	4.28M	1837	20.2	1.22	37.2
36.5M	20.4M × 16.8M	4.15M	1723	22.7	1.14	34.9
39.1M	22.1M × 17.4M	4.04M	1633	25.1	1.08	33.7
41.7M	23.7M × 17.9M	3.94M	1560	27.5	1.03	31.6
44.2M	25.1M × 18.3M	3.87M	1498	29.9	0.99	30.3
46.8M	26.5M × 18.6M	3.80M	1444	32.4	0.96	29.2
49.4M	27.7M × 18.9M	3.73M	1396	34.8	0.92	28.3
52.0M	28.9M × 19.1M	3.68M	1354	37.2	0.90	27.4

One can then use the above estimates to tailor the balance between the sieving and the linear algebra steps to the available CPU and GPU resources. For instance, in a CPU-only context, a minimal running time of $57.4 \cdot 10^3$ core-hours is achieved after sieving around 34M relations, even though most of the other options fall very close. Similarly, if one has 50 CPU cores and 4 GPUs at hand, an optimal wall-clock time of 686 hours is obtained when stopping sieving after only about 31M relations.

5 Towards GF(2^{1039})

After GF(2^{809}), the next natural milestone is to reach a prime-degree extension field of kilobit size. A nice target is GF(2^{1039}), because it echoes the rather recent factorization of $2^{1039} - 1$ which was the first kilobit SNFS integer factorization [4]. In particular, there are subgroups in the multiplicative group that are not amenable to a discrete logarithm computation based on Pollard's ρ, since the two largest prime factors of $2^{1039} - 1$ are 752 and 265 bits long, respectively, and are both far beyond what is feasible with an exponential algorithm.

5.1 Relation collection

As mentioned in [11], it is a rather easy task to obtain a set of relations that is complete—in the sense that it yields a system with (far) more equations than unknowns—so that we can expect it to be full-rank or close to full-rank. We ran our software with exactly the same parameters as in [11], namely:

- a polynomial $f(x,t)$ of degree 6 in x, which is not as good as the one we used for GF(2^{809}), since its α-value is around -5;
- a large-prime bound of 33 on each side;
- a factor base bound of 25 on each side;
- a threshold value of 99;
- a sieving range with parameters $I = J = 15$.

The relation collection was performed for a large range of special-$\mathfrak{q}$'s, in the following order: first the rational special-$\mathfrak{q}$'s of degree 26 to 30, then the algebraic special-$\mathfrak{q}$'s of degree 28 to 29, and finally, again the rational special-$\mathfrak{q}$'s of degree 31 and 32. For the rational special-$\mathfrak{q}$'s of degree 32, we sieved only about half of the range. The computing time for this relation collection step was equivalent to 124 core-years on an Intel i5-2500 at 3.3 GHz as the one used in [11]. On a 768-core cluster of Intel E5-2650 at 2.0 GHz to which we have access, this corresponds to 3 months.

After having computed a bit more than 1.2 billion relations (which corresponds to the start of the rational degree-31 range), we got our first complete system. The corresponding matrix had 110 million rows and columns, with an average weight of 100 non-zero coefficients per row. The rate of duplicates, at this stage, was 32.5 %.

As we sieved more and more special-$\mathfrak{q}$'s, the time to get a relation increased with the degree of the special-$\mathfrak{q}$'s, the rate of duplicates increased as well, but on the other hand, the matrix produced by the filtering stage was getting smaller.

After having computed about 2 billion relations, we got a duplicate rate of 37.7 % and a matrix with 67 million rows and columns. In total, we obtained 2.6 billion relations, leading to 1.6 billion unique relations (duplicate rate of 38.5 %). The final matrix had then 60 million rows and columns, with an average row-weight of 100.

We then stopped the relation collection because we had reached a point where the size of the matrix was decreasing very slowly, and we could not really expect to save much more by just sieving larger and larger special-$\mathfrak{q}$'s. Still, we remark that the estimates in [11] were indeed accurate.

5.2 Linear algebra

We ran some computer experiments in order to estimate the running time that the linear algebra step would take. For these experiments, we worked in the subgroup whose order is the 265-bit prime factor of $2^{1039} - 1$.

The maximum available memory on our graphics cards is 4 GB, which is far below what is needed to store the matrix and the temporary vectors. However, this fits in the 64 GB of RAM of the CPU host of our cards. Therefore, our timings include the additional cost of on-the-fly data transfers between the CPU and the GPU. The sparse-matrix–vector product on a single GPU then requires 14.3 seconds, which leads to an estimate of 82 GPU-years for the whole linear algebra phase (assuming a simple Wiedemann implementation).

The CPU version of our implementation could run two interleaved sparse-matrix–vector products on a 16-core dual Intel E5-2650 node in 37 seconds, leading to an estimate of 1408 core-years for the whole linear algebra step, or 22 months on a 768-core cluster to which we have access (assuming blocking parameters $n = 96$, $m = 192$). Because of the important memory requirements for this step, we were unable to perform benchmarking experiments on our Intel i5-2500 machines.

Although this computation is clearly feasible, we did not start it for real. The first reason is that we still need to investigate how the minimal polynomial computation would behave in practice for the huge blocking factors we used in our CPU estimates. The second reason is that, based on our experiments with $\mathrm{GF}(2^{809})$, we want to try other parameters than the ones proposed in [11]. For instance, lowering the large-prime bound to 32 would certainly make life easier for the linear algebra; there is still some work to do in order to make our sieving code efficient in that case.

6 Conclusion

Our computation of discrete logarithms in $\mathrm{GF}(2^{809})^\times$ claimed a total time which is not immense, especially in comparison to large integer factorization records [23]. It brings important data, however, towards the assessment of the feasibility limit of discrete logarithms in $\mathrm{GF}(2^p)$ for prime extension degrees p. Given our experimental data and our running time projections, it is apparent that reaching larger sizes is possible using current hardware and software technology, going even to kilobit-sized fields provided one affords the necessary time.

References

1. Adj, G., Menezes, A., Oliveira, T., Rodríguez-Henríquez, F.: Weakness of $\mathbb{F}_{3^{6*509}}$ for discrete logarithm cryptography (2013), preprint, 24 pages, available at `http://eprint.iacr.org/2013/446`
2. Adleman, L.M.: The function field sieve. In: Adleman, L.M., Huang, M.D.A. (eds.) ANTS-I. Lecture Notes in Comput. Sci., vol. 877, pp. 108–121. Springer–Verlag (1994), proceedings
3. Adleman, L.M., Huang, M.D.A.: Function field sieve method for discrete logarithms over finite fields. Inf. Comput. 151(1–2), 5–16 (1999)
4. Aoki, K., Franke, J., Kleinjung, T., Lenstra, A.K., Osvik, D.A.: A kilobit special number field sieve factorization. In: Kurosawa, K. (ed.) Advances in Cryptology–ASIACRYPT 2007. pp. 1–12. Springer (2007)
5. Bai, S., Bouvier, C., Filbois, A., Gaudry, P., Imbert, L., Kruppa, A., Morain, F., Thomé, E., Zimmermann, P.: `cado-nfs`, an implementation of the number field sieve algorithm (2013), development version, `http://cado-nfs.gforge.inria.fr/`
6. Bai, S., Filbois, A., Gaudry, P., Kruppa, A., Morain, F., Thomé, E., Zimmermann, P.: CADO-NFS, Crible Algébrique: Distribution, Optimisation - Number Field Sieve, `http://cado-nfs.gforge.inria.fr/`
7. Barbulescu, R.: Selecting polynomials for the Function Field Sieve (2013), preprint, 23 pages, available at `http://hal.inria.fr/hal-00798386`
8. Barbulescu, R., Bouvier, C., Detrey, J., Gaudry, P., Jeljeli, H., Thomé, E., Videau, M., Zimmermann, P.: The relationship between some guy and cryptography (2012), ECC2012 rump session talk (humoristic), see `http://ecc.2012.rump.cr.yp.to/`
9. Barbulescu, R., Gaudry, P., Joux, A., Thomé, E.: A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic (2013), preprint, 8 pages, available at `http://hal.inria.fr/hal-00835446`

10. Bouvier, C.: The filtering step of discrete logarithm and integer factorization algorithms (2013), preprint, 22 pages, available at `http://hal.inria.fr/hal-00734654`
11. Detrey, J., Gaudry, P., Videau, M.: Relation collection for the Function Field Sieve. In: Nannarelli, A., Seidel, P.M., Tang, P.T.P. (eds.) Proceedings of ARITH-21. pp. 201–210. IEEE (2013)
12. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
13. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory 31(4), 469–472 (1985)
14. Granger, R., Holt, A.J., Page, D., Smart, N.P., Vercauteren, F.: Function field sieve in characteristic three. In: ANTS VI. Lecture Notes in Comput. Sci., vol. 3076, pp. 223–234 (2004)
15. Hayashi, T., Shimoyama, T., Shinohara, N., Takagi, T.: Breaking pairing-based cryptosystems using η_T pairing over $GF(3^{97})$. In: ASIACRYPT 2012. Lecture Notes in Comput. Sci., vol. 7658, pp. 43–60 (2012)
16. Jeljeli, H.: Accelerating iterative SpMV for Discrete Logarithm Problem using GPUs (2013), preprint, 11 pages, available at `http://hal.inria.fr/hal-00734975`
17. Joux, A., Lercier, R.: Discrete logarithms in GF(2^n) (521 bits) (Sep 2001), email to the NMBRTHRY mailing list. Available at `http://listserv.nodak.edu/archives/nmbrthry.html`
18. Joux, A., Lercier, R.: The function field sieve is quite special. In: Fieker, C., Kohel, D.R. (eds.) ANTS-V. Lecture Notes in Comput. Sci., vol. 2369, pp. 431–445. Springer–Verlag (2002), proceedings
19. Joux, A.: A new index calculus algorithm with complexity $L(1/4 + o(1))$ in very small characteristic (2013), preprint, 12 pages, available at `http://eprint.iacr.org/2013/095`. To appear in *Selected Areas in Cryptography*, 2013.
20. Joux, A., Lercier, R.: Improvements to the general number field sieve for discrete logarithms in prime fields. a comparison with the Gaussian integer method. Mathematics of computation 72(242), 953–967 (2003)
21. Joux, A., Lercier, R.: Discrete logarithms in GF(2^{607}) and GF(2^{613}). E-mail to the NMBRTHRY mailing list; `http://listserv.nodak.edu/archives/nmbrthry.html` (Sep 2005)
22. Kaltofen, E.: Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. Mathematics of Computation 64(210), 777–806 (1995)
23. Kleinjung, T., Aoki, K., Franke, J., Lenstra, A.K., Thomé, E., Bos, J., Gaudry, P., Kruppa, A., Montgomery, P.L., Osvik, D.A., te Riele, H., Timofeev, A., Zimmermann, P.: Factorization of a 768-bit RSA modulus. In: Rabin, T. (ed.) Advances in Cryptology – CRYPTO 2010. Lecture Notes in Comput. Sci., vol. 6223, p. 333–350. Springer–Verlag (2010)
24. Matsumoto, R.: Using C_{ab} curves in the function field sieve. IEICE Trans. Fund. E82-A(3), 551–552 (1999)
25. Murphy, B.A.: Polynomial selection for the number field sieve integer factorisation algorithm. Phd thesis, Australian National University (1999)
26. National Institute of Standards and Technology: Digital signature standard (DSS) (2013), `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`

Magma verification script

```
UP<t> := PolynomialRing(GF(2));
hex2pol := func<x | UP!Intseq(x, 2)>;

phi := hex2pol(0x3ffffffffffffffffffffffffffffffffe80000000000000\
  0000000000cc0cfaeb0e000000000000000000000000000000004dfffffffff\
  ffffc3c3ffc3c3c3fce3c2133ffffffffffffe9697fe96804c84c97e0b0000c\
  00cf9b0f7675354e79f4cf7c97e29);
assert Degree(phi) eq 809 and IsIrreducible(phi);

N   := 2^809-1;
ell := Factorization(N)[1][1];
h   := N div ell;

rsa1024 := hex2pol(0xc05748bbfb5acd7e5a77dc03d9ec7d8bb957c1b95d9b\
  206090d83fd1b67433ce83ead7376ccfd612c72901f4ce0a2e07e322d438ea4\
  f34647555d62d04140e1084e999bb4cd5f947a76674009e2318549fd102c5f7\
  596edc332a0ddee3a355186b9a046f0f96a279c1448a9151549dc663da8a6e8\
  9cf8f511baed6450da2c1cb);
log1024 :=
  2999787071911643485450020083420834977154987908338125416470796;

Modexp(rsa1024, h, phi) eq Modexp(t, h*log1024, phi);
```