Mathematical basis of the Quadratic Sieve

Alexandre Joly

July 25, 2025

Abstract

This document present the Quadratic Sieve algorithm. First, we explain why we need an involving algorithm to factorize a large number. Then, we walk throught the mathematical basis of the algorithm. Finally, we present the Wiedemann's algorithm which is used to solve the linear algebra problem in the Quadratic Sieve.

Contents

1 RSA cryptosystem

RSA is an asymmetric encryption system *(public key for encryption and private key for decryption)*. It was published in 1977 [6] by Ron Rivest, Adi Shamir and Leonard Adleman, from whose surnames the initials R, S and A are derived.

1.1 Principle

The aim of RSA is to build a **one-way** function, meaning it is easy to compute in one direction but computationally hard to reverse.

First, let's posit two required lemmas:

1. **Bézout's Identity:** for any two integers p and q, there exist integers x and y such that:

$$px + qy = \gcd(p, q) \tag{1}$$

   Then if p and q are distinct prime numbers:

$$px + qy = 1 \tag{2}$$

2. **Euler's theorem (generalization of Fermat's Little Theorem)**: if n is a positive integer and a is an integer coprime with n, and $\varphi(n)$ is Euler's totient function, then:

$$a^{\varphi(n)} \equiv 1 \mod n \tag{3}$$

   The Euler's totient function can be computed using the Euler's product formula:

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right) \tag{4}$$

   where p are the distinct prime factors of n.

Now, given:

- $n = p \times q$ with p and q distinct prime numbers
- a an integer coprime with n
- e an integer coprime with $\varphi(n)$

Then, according to Bézout's identity, there exist two integers d and m such that:

$$ed - m\varphi(n) = 1 \iff ed = 1 + m\varphi(n) \tag{5}$$

Hence:

$$a^{ed} = a^{1+m\varphi(n)} = a \times (a^{\varphi(n)})^m \tag{6}$$

Using Euler's theorem:

$$a^{ed} \equiv a \times ((1 \mod n)^m) \tag{7}$$

Resulting in the **RSA theorem**:

$$a^{ed} \equiv a \tag{8}$$

Hence using e and d it is possible to encrypt and decrypt a message.

1.2 Keys generation

1. Choose two distinct large prime numbers: p and q
2. Compute the product named the **modulus**: $n = p \times q$
3. Compute the *Euler's totient function* in n: $\varphi(n) = \varphi(pq) = (p-1)(q-1)$ [1]
4. Choose an integer e smaller and coprime to $\varphi(n)$, e is called **encryption exponent** or **public key exponent**.
5. Compute d the **decryption exponent** or **private key exponent** which is the *modular multiplicative inverse* of e modulo $\varphi(n)$: $d \equiv e^{-1} \mod \varphi(n)$. This is solved in the form $de \equiv 1 \mod \varphi(n)$ using the *extended Euclidean algorithm.*

Then we have both keys:

- Public key: $\{n, e\}$
- Private key: d

The size of the modulus n in bits is the key length. Nowadays, RSA system use 2048 or 4096 bits keys. To visualize how long it is we can use the logarithm to estimate the number of decimal digits in function of the key length k:

$$2^k = 10^n \implies n = k \log_{10}(2) \tag{9}$$

Key Size (bits)	Approximate Number of Decimal Digits
1024	308
2048	617
4096	1233

Table 1: RSA key size and corresponding number of decimal digits

The record for the largest RSA number factorization is 829 bits, which is equivalent to 250 decimal digits. This record was achieved in 2020 by a team of researchers from the University of Limoges, using the General Number Field Sieve (GNFS) algorithm and 2700 CPU core-years (2.1 GHz Intel Xeon Gold 6130 as reference).

1.3 Encryption and decryption of a message

Let the message M be a natural number smaller than n.

The ciphertext is then: $C \equiv M^e \mod n$

To recover the initial message we need the private key: $C^d \equiv M^{ed} \equiv M \mod n$

1.4 Reverse operation

We saw that having the decryption exponent d and the modulus n can be used to decode the ciphertext.

d is found by solving $de \equiv 1 \mod \varphi(n)$. However, if n is large and we don't know its factorisation then the computation of $\varphi(n)$ is too long.

Thus finding the factorisation $n = p \times q$ is the key to the key.

[1] While the original version of RSA use Euler's totient, modern implementation use Carmichael's function for efficiency but the process is the same as $\lambda(n)$ divide $\varphi(n)$.

2 Quadratic Sieve by Carl Pomerance (1981)

The Quadratic Sieve [5] is an **integer factorization algorithm** for large N. It was invented by Carl Pomerance in 1981 as an improvement to Schroeppel's linear sieve that builds on the ideas of Dixon and Kraitchik.

In practice, it is the second fastest algorithm for factorizing integers of more than 100 digits, after the General Number Field Sieve. It is, however, the fastest for integers of less than 100 digits. Given our resources, we will focus on the factorization of integers of less than 100 digits and therefore, the Quadratic Sieve algorithm is relevant.

2.1 Principle

In the field of factorization, *Pierre de Fermat (17th century)* is a major figure. He was the first to propose a method to factorize a number N using the representation of an odd number as the difference of two squares:

$$N = a^2 - b^2 = (a + b) \times (a - b) \tag{10}$$

Fermat proposed to search the smallest $a \geq \lceil \sqrt{N} \rceil$ such that $a^2 - N$ is a perfect square, requiring $O(\sqrt{N})$ trials in the worst case. In its simplest form, Fermat's method might be slower than trial division but this idea inspired many other mathematicians, including *Maurice Kraitchik*.

In the 1920s, Kraitchik proposed that instead of looking for a difference of two squares $N = a^2 - b^2$, we could look for a difference of two squares that is a multiple of N. In other words, a number that is a square modulo N:

$$a^2 \equiv b^2 \mod N, \quad \text{with} \quad a \not\equiv \pm b \mod N \iff a^2 - b^2 \equiv 0 \mod N \tag{11}$$

Equivalent to:

$$\begin{cases} (a - b) \times (a + b) \text{ is a multiple of N} \\ (a - b) \text{ and } (a + b) \text{ are not multiples of N (trivial case)} \end{cases} \tag{12}$$

Thus, N has a non-trivial factor with at least one of $(a - b)$ and $(a + b)$.

So, $\gcd(a + b, N)$ **and/or** $\gcd(a - b, N)$ **are non-trivial factors of** N**.**

2.2 Kraitchik Method

To find squares a^2 and b^2 satisfying the necessary congruence conditions, Kraitchik [2] computed values from what we now call the **Kraitchik polynomial**:

$$Q(x_i) = x_i^2 - N \tag{13}$$

Kraitchik then sought combinations of these values that produce a perfect square. An effective strategy involves factoring each $Q(x_i)$ and identifying combinations such that the product of these factors yields even exponents.

Specifically, the combination yields the desired square:

$$(x_1^2 - N)(x_2^2 - N) \dots (x_k^2 - N) = b^2 \tag{14}$$

This implies:

$$\prod_{i=1}^{k} (x_i^2 - N) \equiv \prod_{i=1}^{k} x_i^2 \mod N, \quad \text{with} \quad a = \prod_{i=1}^{k} x_i \tag{15}$$

Therefore, we have:

$$a^2 \equiv b^2 \mod N \tag{16}$$

which can then be exploited to factor the integer N.

2.3 Dixon technique

Before discussing Dixon's technique, we define the **smoothness of a number**. A number is said to be B-smooth if all its prime factors are less than or equal to B.

Dixon's [1] key improvement was collecting only values $Q(x_i)$ that are B-smooth. With a suitably chosen small B, identifying combinations of these smooth values becomes significantly easier. We thus define a **factor base** consisting of all primes less than or equal to B and denote its size by $\pi(B)$. The B-smooth values we select are termed **relations**.

Next, we construct a matrix M where each row represents the exponent vector of prime factors for each relation $Q(x_i)$. Hence, the matrix M has $\pi(B)$ columns, one for each prime in the factor base.

A key observation is that a perfect square has only even exponents in its prime factorization. Thus, if we find a combination of relations whose exponents sum to an even vector, we obtain a perfect square. Formally, this corresponds to finding a combination of rows of M whose sum is a zero vector modulo 2.

To identify such combinations, we find the kernel of the transposed matrix M^T over the finite field $\mathbb{F}_2$. A non-trivial kernel requires the rank of M to be strictly less than the number of columns, necessitating at least $\pi(B) + 1$ relations.

In general, finding the kernel of a matrix can be performed using Gaussian elimination, which has a complexity of $O(n^3)$. However, in this context, the matrix M is typically large and sparse, motivating the use of more efficient methods specialized for sparse linear algebra.

2.4 Schroeppel's linear sieve

Richard C. Schroeppel's original analysis was never formally published but circulated in personal communications in the mid-1970s.

Schroeppel proposed an important refinement of Dixon's random-sampling approach by replacing it with an efficient sieving process on a two-dimensional polynomial:

$$Q(i, j) = (r + i)(r + j) - N, \quad \text{where} \quad r = \left\lceil \sqrt{N} \right\rceil \tag{17}$$

Sieving in two dimensions offers two main advantages. First, it generates a richer set of candidate values around zero by varying both i and j, one explores a full grid of residues, increasing the density of potential smooth numbers. Second, the 2D layout improves memory locality and allows parallel sieving across rows or columns, yielding practical speedups on modern hardware.

To clearly understand Schroeppel's sieve, we first recall the concept of **residue classes**. In modular arithmetic, the residue class modulo n of an integer a is the set of all integers congruent to a modulo n:

$$[a]_n = a + kn : k \in \mathbb{Z} \tag{18}$$

All elements within a residue class $[a]_n$ are indistinguishable modulo n.

Using this concept, the sieving proceeds as follows:

For each prime p in our chosen factor base, we first solve the congruence

$$Q(x) \equiv 0 \mod p \tag{19}$$

to find the two distinct roots α, β modulo p. Positions corresponding to these residue classes $x \equiv \alpha, \beta \mod p$ in a sieve array are marked (or "crossed off") by adding the value $\log p$.

The rationale behind adding logarithms is to efficiently approximate factorization. When a prime p divides $Q(x)$, the logarithmic value of $|Q(x)|$ decreases by approximately $\log p$ for each occurrence of p as a factor. We initialize an array $S[x]$ to zero, representing the logarithmic accumulations. Each time we identify a residue class modulo p, we increment by $\log p$:

$$S[x] \leftarrow S[x] + \log p \tag{20}$$

After processing all primes in the factor base, we compare the accumulated values $S[x]$ with the logarithmic magnitude of $|Q(x)|$. If the total of recorded logarithms in $S[x]$ closely matches $\log |Q(x)|$ (within a small threshold accounting for rounding errors), this strongly indicates that $Q(x)$ factors completely over the factor base, identifying it as B-smooth. These identified smooth values are exactly the "relations" required to build the necessary matrix for subsequent steps.

This sieve reduces the cost of candidate testing dramatically and laid the groundwork for the first large-scale Quadratic Sieve implementations.

2.5 Pomerance and the Quadratic Sieve

Building on Schroeppel's two-dimensional linear sieve, Pomerance proposed a much more practical and efficient method by simplifying the polynomial used and refining the process of detecting B-smooth.

At the heart of Pomerance's method is a carefully chosen quadratic polynomial:

$$Q(x) = (x + r)^2 - N, \quad \text{where} \quad r = \lceil \sqrt{N} \rceil. \tag{21}$$

This formulation ensures that $Q(x)$ is close to zero when x is small, making it more likely that $Q(x)$ is B-smooth.

2.5.1 Roots Modulo p and Residue Classes

A key insight lies in analyzing the congruence

$$(x + r)^2 \equiv N \mod p. \tag{22}$$

This equation seeks values of x for which $Q(x) \equiv 0 \mod p$. Solving it is equivalent to finding the square roots of N modulo p. Specifically, we seek $s \in \mathbb{Z}$ such that:

$$s^2 \equiv N \mod p. \tag{23}$$

If N is a quadratic residue modulo p, then there exist exactly two such square roots, say s and $-s \mod p$. Rewriting the original congruence:

$$x + r \equiv \pm s \mod p \implies x \equiv s - r \mod p \quad \text{or} \quad x \equiv -s - r \mod p, \tag{24}$$

reveals two residue classes α_p, β_p for which $p \mid Q(x)$. These positions are used to initialize the sieve for each prime p in the factor base.

2.5.2 Sieve Centered Around $\sqrt{N}$

The sieve is centered at $x = 0$, which corresponds to evaluating $Q(x)$ near $r = \lceil \sqrt{N} \rceil$. Around this point, we have the approximation:

$$Q(x) = (r + x)^2 - N = r^2 - N + 2rx + x^2 \approx 2rx + x^2. \tag{25}$$

Since $r \approx \sqrt{N}$, the dominant term is linear in x for small x. Thus, $Q(x)$ grows approximately linearly with x, meaning that the values of $Q(x)$ increase steadily and predictably as x increases or decreases. This behavior keeps $|Q(x)|$ small in a neighborhood around $x = 0$, significantly increasing the probability that $Q(x)$ will be B-smooth.

2.5.3 Choosing the Factor Base Using the Legendre Symbol

To ensure efficiency, only those primes p for which N is a quadratic residue modulo p are used in the factor base. This is efficiently tested via the Legendre symbol [3]:

$$\left(\frac{N}{p}\right) = N^{\frac{p-1}{2}} \bmod p = \begin{cases} 1, & \text{if } N \text{ is a quadratic residue mod } p, \\ -1, & \text{otherwise.} \end{cases} \tag{26}$$

This ensures that for each p in the factor base, the congruence $(x+r)^2 \equiv N \mod p$ has exactly two solutions, and thus $Q(x)$ can potentially be divisible by p.

2.5.4 Optimal Choice of the Smoothness Bound B

Selecting the bound B is a balancing act: it must be large enough to include enough primes for factoring $Q(x)$ values, but small enough to keep the linear algebra manageable. Pomerance suggested, after a study of the smooth number distribution, the following heuristic bound to minimize total complexity:

$$B = e^{\frac{1}{2}\sqrt{\log N \log\log N}}. \tag{27}$$

2.5.5 Key Improvements Beyond the Basic QS

Pomerance and later researchers introduced several refinements that significantly improved the performance and practicality of the algorithm:

- **Self-initializing sieve:** Rather than hardcoding the positions α_p, β_p where each p divides $Q(x)$, this approach dynamically computes them using the modular square roots of N modulo p. These positions are then used to mark the sieve efficiently.
- **Large-prime variation:** Instead of requiring $Q(x)$ to be fully B-smooth, relations with one or two larger primes in (B, B^2) are accepted. These can be recombined later to form usable full relations, boosting the yield significantly.
- **Multiple-polynomial variant (MPQS):** To extend the method to larger values of N, several different polynomials $(ax+b)^2 - N$ are used, each producing smaller $Q(x)$ values and allowing parallelization.
- **Bucket/block sieving:** To reduce memory usage and improve cache efficiency, sieve updates are batched by prime in blocks or "buckets," which avoids touching all sieve positions for each p.

Initially, the quadratic sieve employed the Lanczos algorithm for linear algebra computations. However, contemporary implementations typically use Wiedemann's algorithm, which offers linear algebra solutions with complexity $O(n\omega)$, where ω is the number of nonzero entries.

3 Wiedemann's algorithm

The **Wiedemann algorithm** [7] is a probabilistic iterative, projection-based Krylov-subspace method for solving sparse systems of linear equations over finite fields.

It is based on the fact that when a square matrix is repeatedly applied to a vector, the sequence of vectors satisfies a linear recurrence.

Consider a sparse $n \times n$ matrix A and a vector b of size n over a finite field $\mathbb{F}$, the problem is to find a vector x such that:

$$Ax = b \tag{28}$$

3.1 Origin

The **Cayley-Hamilton theorem** asserts that the square matrix A satisfies its own characteristic equation of degree n, giving the identity:

$$P_A(A) = A^n + c_{n-1}A^{n-1} + \cdots + c_1 A + c_0 I = 0 \tag{29}$$

However, the **minimal polynomial** may be of smaller degree $d \leq n$:

$$m_A(A) = A^d + m_{d-1}A^{d-1} + \cdots + m_1 A + m_0 I = 0 \tag{30}$$

This minimal polynomial retains the same relation but with possibly fewer terms, we don't know d yet but the smaller it is, the faster we will find a solution. Also note that this minimal polynomial is by definition monic (highest-degree coefficient equal to 1).

In the singular case the minimal polynomial has $m_0 = 0$, so A is not invertible, thus the minimal polynomial is of the form:

$$m_A(A) = A^d + m_{d-1}A^{d-1} + \cdots + m_1 A = 0 \tag{31}$$

We can therefore factor out A at least once (more if $m_1, m_2, \ldots, m_{d-1}$ are also zero):

$$A(A^{d-1} + m_{d-1}A^{d-2} + \cdots + m_1 I) = 0 \tag{32}$$

We rewrite this as:

$$m_A(A) = A^r q(A) = 0, \quad r \geq 1 \tag{33}$$

Hence, $q(A)$ is a member of the kernel of A^r:

$$im(q(A)) \subseteq \ker(A^r) \iff im(q(A)) \subseteq \ker(A) \tag{34}$$

Then we want to find a vector w such that:

$$w = q(A)v = \sum_{i=1}^{d} m_i A^i v \tag{35}$$

3.2 Build the vector w

The definition above of w is a sequence of vectors in the Krylov subspace of A and v:

$$\mathcal{K}_d(A, v) = \text{span}\{v, Av, A^2v, \ldots, A^{d-1}v\} \tag{36}$$

Now, we need to find the coefficients m_i. Note that m_i are field elements, so in our context, they are elements of $\mathbb{F}_2$.

The idea of Wiedemann is to do a **scalar projection** to turn that matrix problem over $\mathbb{F}^n$ into a one-dimensional recurrence in $\mathbb{F}$.

1. Let's choose two random vectors u and v in $\mathbb{F}^n$.
2. Define the scalar sequence of length $2n$ (we will see why next):

$$s_i = u^T A^i v, \quad i \in [0, 2n-1]$$

3. Using the relation of the minimal polynomial we have:

$$\sum_{i=0}^{d} m_i s_i = 0$$

The only unknowns are the d terms m_i but at this time we only have 1 equation, which is not enough. In order to build d equations we shift the identity by multiplying it by any non-negative power A^k because it doesn't alter the identity:

$$m_A(A)A^k = 0, \quad k \geq 0$$

Giving the scalar sequence:

$$\sum_{i=0}^{d} m_i u^T A^{i+k} v = \sum_{i=0}^{d} m_i s_{k+i} = 0$$

Now we have enough equation to find and fix d terms m_i.

3.3 Berlekamp-Massey algorithm

The **Berlekamp-Massey** algorithm [4], given a finite-length scalar sequence $\{s_0, s_1, \ldots, s_{N-1}\}$, computes the shortest linear recurrence (i.e., the minimal polynomial) that generates it:

$$C(x) = 1 + C_1 x + C_2 x^2 + \cdots + C_L x^L \tag{37}$$

However, for the moment we only know the value of n but not d. Berlekamp-Massey can only be certain that no shorter recurrence exists once it has processed at least twice the true degree. That is why we will have to use a Krylov sequence of length $2n$.

Algorithm:

1. Initialization:

$$C(x) = 1, \quad B(x) = 1, \quad T(x) = 1, \quad L = 0, \quad m = 1, \quad b = 1$$

- $C(x)$ is the current connection (candidate minimal) polynomial
- $B(x)$ is the previous version of C at the last update

- L is the current length (degree) of C
- m counts how many steps since the last update
- b stores the last nonzero discrepancy

2. Iteration, for each s_i:
   Compute the discrepancy d, it measures how far the current polynomial fails to annihilate the sequence at position i:

$$d = s_i - \sum_{j=1}^{L} C_j s_{i-j}$$

   - If $d = 0$, the current minimal polynomial $C(x)$ correctly predicts the term i, so no update is needed, simply increment $m \leftarrow m + 1$.
   - If $d \neq 0$, we correct C by adding a shifted copy of the last "good" polynomial $B(x)$, scaled so as to cancel that error:
     - Store a copy of the current minimal polynomial $C(x)$ as the previous minimal polynomial $T(x)$: $T(x) \leftarrow C(x)$
     - Update the current minimal polynomial $C(x)$ as:

$$C(x) = C(x) - \frac{d}{b} x^m B(x)$$

     - if $2L \leq i$:
       * Update the length L: $L \leftarrow i + 1 - L$
       * Update the previous minimal polynomial $B(x) \leftarrow T(x)$, the scaling factor $b \leftarrow d$ and the step counter $m \leftarrow i + 1 - L$

3.3.1 Complexity of Berlekamp-Massey:

Each step requires $O(L)$ operations to compute d, resulting in a total $O(L^2)$ complexity. $C(x)$ updates cost $O(L)$.

Therefore, the Berlekamp-Massey algorithm has a complexity of $O(d^2)$, with $d \leq n$.

3.4 Probability of success

Wiedemann's algorithm relies on the fact that the random vector u chosen from $\mathbb{F}^n$ will project the Krylov sequence onto a space that captures the minimal polynomial of A.

It succeeds when the projected sequence $s_i = u^T A^i v$ has the same minimal polynomial as A. Failure occurs only if u lies in the orthogonal complement $\mathcal{K}_d(A, v)^\perp$ because then the projection $u^T A^i v$ will not capture the full span of $\mathcal{K}_d(A, v)$.

Because $\dim \mathcal{K}_d^\perp = n - d$, the probability that a random u lies in this orthogonal complement is given by the ratio of the dimensions:

$$p_{failure} = \frac{\dim \mathcal{K}_d^\perp}{\dim \mathbb{F}_2^n} = \frac{|\mathbb{F}_2|^{n-d}}{|\mathbb{F}_2|^n} = |\mathbb{F}_2|^{-d} \tag{38}$$

Thus a single random u already works with probability $1 - |\mathbb{F}_2|^{-d}$, and repeating with fresh u's drives the failure rate down exponentially: after t trials it is $|\mathbb{F}_2|^{-dt}$.

In the Quadratic Sieve, $\mathbb{F} = \mathbb{F}_2$ and d is typically small, so the probability of success is very high with just a few random vectors.

Also, in the case of singular matrices, the minimal polynomial has $m_0 = 0$, so in order to ensure Berlekamp-Massey finds the minimal polynomial such as $m_A(0) = 0$, we need to ensure that $s_0 = u^T v = 0$. This can be easily checked and if not satisfied, we can simply choose a new random u.

3.5 Complexity of Wiedemann's Algorithm:

We saw that the complexity of the Berlekamp-Massey algorithm is $O(d^2)$, the other major computation is the construction of the Krylov subspace.

It requires n matrix-vector multiplication that costs $O(\omega)$ field ops where ω is the number of non-zeros in A. Giving a complexity of $O(n\omega)$.

For very sparse matrices, as in the Quadratic Sieve, it is typical that $d \ll \omega$, so the Krylov subspace construction dominate and give an overall complexity of $O(n\omega)$.

3.6 Conclusion

Wiedemann's algorithm is a powerful method for solving large sparse linear systems over finite fields, particularly in the context of exact linear algebra tasks. By leveraging the properties of the minimal polynomial and the Berlekamp-Massey algorithm, it efficiently reduces the problem to a manageable scalar sequence, allowing for the recovery of solutions with a relatively low computational cost.

4 Full example

Let's take an example with $N = 2041$. As N is small, we can choose $B = 7$, so $\pi(B) = 4$ and the factor base is:

$$\mathcal{F} = \{2, 3, 5, 7\} \tag{39}$$

4.1 Data collection phase: sieving

$$\sqrt{N} \approx 45.17, \text{so we start the sieve at } 46 \tag{40}$$

- $46^2 \equiv 75 \mod N, 75 = 3 \times 5^2$, 75 is B-smooth
- $47^2 \equiv 168 \mod N, 168 = 2^3 \times 3 \times 7$, 168 is B-smooth
- $48^2 \equiv 263 \mod N, 263$ is prime, 263 is not B-smooth
- ...

At the end of the sieve, we have a set of B-smooth numbers:

$$a_i = \{46, 47, 49, 51, 53\}, \quad x_i = \{75, 168, 360, 560, 768\} \tag{41}$$

4.2 Data processing phase:

The exponent matrix M is built, reduced modulo 2 and then transposed.

$$M = \begin{pmatrix} 0 & 1 & 2 & 0 \\ 3 & 1 & 0 & 1 \\ 3 & 2 & 1 & 0 \\ 4 & 0 & 1 & 1 \\ 8 & 1 & 0 & 0 \end{pmatrix}, \quad M_{mod2} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \quad M_{mod2}^T = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{42}$$

Kernel search using Wiedemann's algorithm:

As we will use the Wiedemann algorithm we need a square matrix. Thus we build $M_{mod2}M_{mod2}^T$ which kernel is the same as M_{mod2}^T.

$$M_{mod2}M_{mod2}^T = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{43}$$

1. Check that the matrix is singular:

   Row 1 and row 5 are identical, so its rank is at most 4. Hence $\ker M \neq \{0\}$.

2. We choose the random vector u and v, for example:

$$u = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \quad \text{and} \quad v = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} \tag{44}$$

   We check that $s_0 = u^T v \equiv 0 \mod 2$:

$$s_0 = u^T v = 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 0 + 0 \cdot 1 + 1 \cdot 1 = 2 \equiv 0 \mod 2 \tag{45}$$

3. We compute the scalar sequence $s_i = u^T M^i v$ for $i = 0, 1, \ldots, 9$:

$$\{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\} = \{0, 1, 0, 0, 1, 1, 0, 0, 1, 1\} \tag{46}$$

4. We apply the Berlekamp-Massey algorithm to the scalar sequence $\{s_i\}$ to find the minimal polynomial coefficients m_i:

$$m_i = \{1, 1, 1, 1, 0\} \tag{47}$$

5. The minimal polynomial is then:

$$m_M(x) = 1 + x + x^2 + x^3 \tag{48}$$

6. We can now compute the solution x using the minimal polynomial:

$$x = (M^3 + M^2 + M + I)v \equiv \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} \mod 2 \tag{49}$$

7. Finally, we can verify that $Mx = 0$:

$$Mx = 0 \tag{50}$$

This confirms that the Wiedemann's algorithm has successfully found a solution to the system $Mx = 0$.

Tentative solution computation:

The solution vector x corresponds to the couples: $\{46, 75\}, \{47, 168\}, \{49, 360\}, \{51, 560\}$.

Thus we can compute the tentative solution:

$$(46 \times 47 \times 49 \times 51)^2 \equiv 75 \times 168 \times 360 \times 560 \mod N \tag{51}$$

$$(46 \times 47 \times 49 \times 51)^2 \equiv 2^{10} \times 3^4 \times 5^4 \times 7^2 \mod N \tag{52}$$

$$(46 \times 47 \times 49 \times 51)^2 \equiv (2^5 \times 3^2 \times 5^2 \times 7)^2 \mod N \rightarrow \text{We found a square modulo N} \tag{53}$$

Now,

$$a = 5402838 \equiv 311 \mod N, \quad b = 50400 \equiv 1416 \mod N \tag{54}$$

$$pgcd(1416 - 311, N) = 13 \text{ and } pgcd(311 + 1416, N) = 157 \tag{55}$$

We found the two prime factors of N:

$$2041 = 157 \times 13 \tag{56}$$

A Wiedemann's algorithm pseudo-code

In pseudo-code, Wiedemann's algorithm can be summarized as follows:

```
function Wiedemann(A, b):
    n = len(b)
    u = random_row_vector(n)
    s = [0] * (2 * n)  # Initialize scalar sequence of length 2n
    for i in range(2 * n):
        s[i] = u @ (A ** i) @ b  # Compute scalar sequence
    C = BerlekampMassey(s)  # C contains the coefficients of the minimal polynomial
    x = compute_solution(A, b, C)  # Use C to compute the solution
    return x  # Return the solution vector x
```

B Berlekamp-Massey algorithm pseudo-code

In pseudo-code, the Berlekamp-Massey algorithm can be summarized as follows:

```
function BerlekampMassey(s):
    C = [1]  # current polynomial coefficients
    B = [1]  # previous polynomial coefficients
    L = 0    # current length of C
    m = 1    # step counter since last update
    b = 1    # last nonzero discrepancy
    for i from 0 to len(s) - 1:
        d = s[i] + sum(C[j] * s[i - j] for j in range(1, L + 1))
        if d == 0:
            m += 1
        else:
            T = C[:]  # store current C as T
            C = [C[j] - (d / b) * (C[j - m] if j >= m else 0) for j in range(len(C))]
            if 2 * L <= i:
                L = i + 1 - L
                B = T[:]
                b = d
                m = i + 1 - L
    return C  # returns the coefficients of the minimal polynomial
```

References

[1] John D. Dixon. Asymptotically fast factorization of integers. *Mathematics of Computation*, 36(153):255–260, January 1981.

[2] Marcel Kraitchik. *Recherches sur la théorie des nombres, Tome II: Factorisation.* Gauthier-Villars, Paris, France, 1929.

[3] Adrien-Marie Legendre. *Essai sur la théorie des nombres.* Duprat, Paris, France, 1798. Originally published An VI (1798).

[4] James L. Massey. Shift-register synthesis and bch decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, January 1969.

[5] Carl Pomerance. The quadratic sieve factoring algorithm. In *Advances in Cryptology: Proceedings of EUROCRYPT 84*, volume 209 of *Lecture Notes in Computer Science*, pages 169–182. Springer, Berlin, Heidelberg, January 1985.

[6] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[7] Douglas H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, January 1986.