Applications of polynomials

Application of discrete models

1 Shamir's secret sharing (SSS)

1.1 Lagrange interpolation

We know that a polynomial of degree n has at most n roots (at least in rings with no zero divisors, e.g. for numbers). This also means that if a polynomial of degree *at most* n has $n+1$ roots, then it can only be the zero polynomial (0) (which has arbitrarily many roots).

This statement can be generalized from roots to other values:

Theorem 1.1. *If two polynomials of degree $\leq n$ are equal in $n+1$ places, then they are equal.*

For example, if p and q have degree ≤ 2, and they are equal in three places, e.g. $p(1) = q(1)$, $p(2) = q(2)$ and $p(5) = q(5)$, then the theorem says that $p = q$. For linear polynomials, two values are sufficient (visually: two points define a line), and for constant polynomials, one is enough.

(The proof of the theorem is simple: the difference of the two polynomials has zeros on these $n+1$ places, i.e. roots, but then it can only be the zero polynomial.)

So $n+1$ values uniquely determine a polynomial of degree n.

But how can we calculate this unique polynomial from the $n+1$ values? First, consider a special case where all values are zero except one. For example, let's find the quadratic polynomial p for which $p(1) = 0$, $p(2) = 0$ and $p(5) = 8$. The first two conditions can be easily handled: $p := (x-1)(x-2)$. But the third one is not satisfied yet: $p(5) = (5-1)(5-2) = 12 \neq 8$. So we multiply this polynomial by a constant c such that $c \cdot p(5) = 8$, i.e. $c := 8/p(5) = 8/12 = 2/3$, because this transformation not only "fixes" the third value, but it also preserves the zeros. So let p be rather $p := 2/3 \cdot (x-1)(x-2) = 2x^2/3 - 2x + 4/3$.

How can we generalize this to arbitrary (not necessarily zero) values? For example, let's find the quadratic polynomial p for which $p(1) = 4$, $p(2) = 2$ and $p(5) = 8$. For this, we use the above method three times, and combine the results:

$$p_1(1) = 4,\ p_1(2) = 0,\ p_1(5) = 0 \implies p_1 = 4 \cdot \frac{(x-2)(x-5)}{(1-2)(1-5)} = x^2 - 7x + 10$$

$$p_2(1) = 0,\ p_2(2) = 2,\ p_2(5) = 0 \implies p_2 = 2 \cdot \frac{(x-1)(x-5)}{(2-1)(2-5)} = -\frac{2}{3}x^2 + 4x - \frac{10}{3}$$

$$p_3(1) = 0,\ p_3(2) = 0,\ p_3(5) = 8 \implies p_3 = 8 \cdot \frac{(x-1)(x-2)}{(5-1)(5-2)} = \frac{2}{3}x^2 - 2x + \frac{4}{3}$$

$$p(1) = 4,\quad p(2) = 2,\quad p(5) = 8 \implies p = p_1 + p_2 + p_3 \qquad = x^2 - 5x + 8$$

This method is called *Lagrange interpolation*. In general:

Theorem 1.2 (Lagrange interpolation)**.** *If a polynomial p of degree $\leq n$ takes the following values on given $n+1$ different places: $p(x_1) = y_1$, $p(x_2) = y_2$, ..., $p(x_{n+1}) = y_{n+1}$, then:*

$$p = \sum_{i=1}^{n+1} y_i \prod_{\substack{j=1 \\ j \neq i}}^{n+1} \frac{x - x_j}{x_i - x_j}.$$

The method works for *any* $x_1, \ldots, x_{n+1}$ and $y_1, \ldots, y_{n+1}$ values (if all x_i's are different), i.e. we don't need to know beforehand whether the polynomial exists, because the above formula always gives such a polynomial (and it is the only one).

Because of the divisions, the Lagrange interpolation works only over fields, e.g. in $\mathbb{R}[x]$ or $\mathbb{Q}[x]$, but *not* in $\mathbb{Z}[x]$ (as in the above example, even though all values were integers, we still needed fractions).

Sage has a built-in function for Lagrange interpolation:

```
sage: P.<x> = QQ[]    # (doesn't work for ZZ, only for fields)
sage: P.lagrange_polynomial([(1,4), (2,2), (5,8)])
x^2 - 5*x + 8
```

1.2 Secret sharing

Imagine a secret information (a number, e.g. an encryption key) which needs to be shared among multiple people (e.g. among the employees of a company).

The simplest way to do this is to give the secret to all participants. The problem with this of course is that if any of them misuses it, or accidentally leaks it, then the whole secret is compromised. We need a solution where no single person can access the secret.

The opposite end is to split the secret into parts, and give each participant a single part. Then none of them can individually misuse their own part – not even a small group of them can do that –, so the secret can only be restored when all participants are involved. But the problem with this is that if any of them loses their part (or they are temporarily unavailable), then the common secret cannot be restored.

But there is also another problem with this approach: altough none of the participants have the *whole* secret, they still have a part of it, which brings them one step closer to finding out the secret using brute force. For example, if a number lock has a four-digit code, then someone who doesn't know anything about the code needs to try $10^4 = 10000$ possibilities to open it (in the worst case). But if someone knows one digit of the code, then they only need to try $10^3 = 1000$ possibilities, which is ten times faster. With two digits, it is hundred times faster.

The ideal solution would need a threshold k out of the n participants to restore the secret (e.g. three out of five), so that any k participants would be able to calculate the secret, but less than k people not only couldn't recover the whole secret, but they couldn't even get one step closer to finding it out than an outsider with no information at all.

A solution for this is called *Shamir's secret sharing* (SSS).

Warning: the method as described right below is not yet completely secure, but this way it is easier to understand for the first time; the improved version will be presented afterwards.

Let n be the number of participants, and k the threshold to restore the secret ($1 \leq k \leq n$). Let $S \in \mathbb{Z}$ be the secret to be shared, and let's "hide" it in a polynomial of degree $k-1$ as follows:

$$p = a_{k-1}x^{k-1} + \ldots + a_2x^2 + a_1x + S,$$

where the constant term is the secret S, and the other coefficients $(a_1, \ldots, a_{k-1})$ are randomly chosen integers. Evaluate this polynomial in n different places, e.g. $p(1), p(2), \ldots, p(n)$, and give each participant one of the values (a pair (x_i, y_i)). Then any k participants can use their k values to construct the original polynomial using Lagrange interpolation (see above), and thus get its constant term, S. But no group of less than k people can restore the polynomial (because they could fill the missing values in any way, resulting in very different polynomials, and they wouldn't know which one was the original).

Let's see an example for $n = 5$ participants and threshold $k = 3$. Let the secret be the value $S = 67$, and generate two more random coefficients, e.g. $a_1 = 38$ and $a_2 = 13$, to construct the following quadratic polynomial: $p = 13x^2 + 38x + 67$. Then evaluate p in five places, e.g.:

$$p(1) = 118, \quad p(2) = 195, \quad p(3) = 298, \quad p(4) = 427, \quad p(5) = 582,$$

and give each participant one of these values. Then any three of them can restore S, e.g. the 1st, 2nd and 5th can do this as follows (using Lagrange interpolation, see above):

$$p = 118 \cdot \frac{(x-2)(x-5)}{(1-2)(1-5)} + 195 \cdot \frac{(x-1)(x-5)}{(2-1)(2-5)} + 582 \cdot \frac{(x-1)(x-2)}{(5-1)(5-2)} = 13x^2 + 38x + 67.$$

We can optimize this by not calculating the whole polynomial, which we don't need anyway, only its constant term S, which can be done by calculating $p(0)$, i.e. by substituting $x = 0$:

$$S = p(0) = 118 \cdot \frac{(-2)(-5)}{(1-2)(1-5)} + 195 \cdot \frac{(-1)(-5)}{(2-1)(2-5)} + 582 \cdot \frac{(-1)(-2)}{(5-1)(5-2)} = 67.$$

But the method described above still has a problem. If two participants cooperate, then although they can't recover the whole secret, but they can still figure out some properties of it. For example, if they know $p(1)$ and $p(5)$, then they can use the fact that the polynomial is quadratic with integer coefficients, i.e. it can be written as $p = ax^2 + bx + S$ with $a, b \in \mathbb{Z}$, to calculate the following:

$$\begin{aligned} p(5) &= 25a + 5b + S = 582 \\ p(1) &= a + b + S = 118 \\ p(5) - p(1) &= 24a + 4b = 464 \quad \Longrightarrow \quad b = 116 - 6a \\ \Longrightarrow \quad p(1) &= a + (116 - 6a) + S = 118 \quad \Longrightarrow \quad S = 5a + 2. \end{aligned}$$

So they know that S is congruent to 2 modulo 5, so if they want to find the secret by brute force, then they can do it five times faster.

This problem can be eliminated if the coefficients are not integers, but members of a field. The obvious choice would be the field of rationals ($\mathbb{Q}$), but fractions can be inconvenient, and also, the random number generation would be problematic (e.g. what distribution do we use?). Therefore in practice, a finite field $\mathbb{Z}_m$ is used instead, where m is a large enough prime number (so that all possible values of S can be represented).

For example, in $\mathbb{Z}_{89}$, the above polynomial, $p = \overline{13}x^2 + \overline{38}x + \overline{67}$ is again evaluated in five places (which are the same values but modulo 89):

$$p(\overline{1}) = \overline{29}, \quad p(\overline{2}) = \overline{17}, \quad p(\overline{3}) = \overline{31}, \quad p(\overline{4}) = \overline{71}, \quad p(\overline{5}) = \overline{48}.$$

The secret can be recovered with exactly the same calcalations as above, but in $\mathbb{Z}_{89}$ instead of $\mathbb{Q}$.

In this way, the problem mentioned above is no longer present, because although they can still calculate that $S = \overline{5}a + \overline{2}$, but now this provides no information about S at all, because in a field, these kinds of equations have a solution for each possible $S \in \mathbb{Z}_{89}$: $a = \overline{5}^{-1}(S - \overline{2})$.

2 Cyclic redundancy check (CRC)

When data is transmitted through a physical channel (e.g. in a wire or with radio signals), errors can happen, i.e. the received data may not be exactly the same as what was sent. How can we detect such errors?

The solution is *redundancy*: extend the message by a new part called the *check value*, which is calculated from the message, i.e. it doesn't add any new information, but it can be used to check whether the data is correct. If the receiver finds that the check value doesn't correspond to the message, than they can be sure that the message or the check value (or both) was corrupted.

A simple example is the *parity bit*: the message, which is a bit sequence, is extended by a single new bit: if the message contains an even number of 1 bits, then append 0, otherwise append 1. Then the result always has an even number of 1's. For example: $0101 \to 01010$, and $1101 \to 11011$. If we get 11010 on the other end of the channel, then we know that an error must have happened, because it contains an odd number of 1's. But we don't know *where* the error is, e.g. both of the above two example bit sequences differ in only one bit from this one. The parity bit method always detects a single-bit error, but it can't detect two bits (because then the number of 1's remains even).

A more advanced method is *Cyclic Redundancy Check (CRC)*, which is used in many places for error-detection (e.g. Ethernet, USB, Wi-Fi).

CRC treats the bit sequences as polynomials, where each coefficient is one of the bits (0 or 1). For example, the bit sequence 101011 corresponds to the polynomial $x^5 + x^3 + x + 1$ (the last bit is the constant term, the previous one is the linear term, and so on). The polynomials are in $\mathbb{Z}_2[x]$, i.e. all calculations are done modulo 2, e.g. $1 + 1 = 0$, $0 - 1 = 1$ etc. As a consequence of this, for all polynomials, $p+p = 0$ (e.g. $(x^2+1)+(x^2+1) = (1+1)x^2+(1+1) = 0$), so $p = -p$, therefore addition is the same as subtraction: $p - q = p + (-q) = p + q$. This addition/subtraction is also the same as the bitwise "exclusive or" (XOR, $\oplus$) operation (e.g. $1 + 1 = 0 \longrightarrow$ "true" $\oplus$ "true" = "false").

The main operation of CRC is polynomial division with remainder in $\mathbb{Z}_2[x]$. The divisor is a fixed polynomial g, the *generator polynomial* of CRC. If $n := \deg g$ (i.e. g is $n+1$ bits long), then we call this an *n-bit CRC* (CRC-n), since the remainder by g fits in n bits (as it has degree $\leq n-1$).

Sending a message (for a given generator polynomial $g \in \mathbb{Z}_2[x]$):

1. Input: the message m to be transmitted (bit sequence, or it can be treated as a polynomial).
2. Extend m by n zeros, i.e. calculate the polynomial mx^n.
3. Divide mx^n by g, and let c be the remainder – this is the *check value*.
4. Extend m by the n-bit check value c, i.e. calculate $mx^n + c$ – this is the transmitted *codeword*.

Checking the message, method 1:

1. Input: the received codeword $mx^n + c$ (see above).
2. Split the codeword into m and c, i.e. separate the last n bits.
3. Divide mx^n by g (as above).
4. If the remainder is different from c, then the codeword is invalid.

Checking the message, method 2:

1. Input: the received codeword $mx^n + c$ (see above).
2. Divide $mx^n + c$ by g.
3. If the remainder is not 0, then the codeword is invalid.

The two methods are equivalent, because if the remainder of mx^n is c, i.e. $mx^n = qg + c$, then $mx^n - c = qg$; but in $\mathbb{Z}_2[x]$, subtraction is the same as addition, so $mx^n - c = mx^n + c$, therefore $g \mid mx^n + c$, i.e. $mx^n + c$ indeed gives the remainder 0.

For example, let the generator be $g = x^4 + x + 1$, or as a bit sequence, $g = 10011$ (so $n = 4$), and let the message be $m = 11000101$. For transmission, calculate the polynomial $mx^n = 11000101|0000$ (i.e. append four zeros to m), and divide it by g (see below on the left). The remainder is $c = 110$, extend it to n bits: $c = 0110$, and append it to the message: $mx^n + c = 11000101|0110$, to get the transmitted codeword. The receiver checks this value by dividing it by g (see below on the right), and since the remainder is 0, the message is accepted as valid. (Again: these are not numbers in binary, but polynomials in $\mathbb{Z}_2[x]$, so subtraction is really XOR.)

Sending ($mx^n \bmod g$):

```
 11000101|0000 mod 10011
−10011
  10111
 −10011
    10001
  −10011
     10000
   −10011
      110
```

Checking ($mx^n + c \bmod g$):

```
 11000101|0110 mod 10011
−10011
  10111
 −10011
    10001
  −10011
     10011
   −10011
      00
```

The advantage of CRC is that this polynomial division can be done very efficiently on computers, because it needs only bit manipulations, which computers are already good at.

Unfortunately, it is impossible to detect *all* errors, and the check will occasionally accept some invalid codewords too. The goal is to make this as unlikely as possible. An error means that instead of the transmitted $mx^n + c$, the receiver gets $mx^n + c + e$, where e is the *error polynomial*, i.e. which has a 1 coefficient for each bit that was corrupted during transmission. If an error is undetected, it is because $g \mid mx^n + c + e$ (i.e. the remainder is 0), therefore $g \mid e$ (since for the correct codeword, $g \mid mx^n + c$). A good g therefore does not divide e for the most frequently occuring error patterns.

Here are some aspects of choosing a generator polynomial (g):

1. Its degree (n) equals the number of check bits, so the bigger it is, the smaller the error probability can be (a random bit sequence is accepted with $1/2^n$ probability).
2. The constant term (last bit) of g should be 1, because:
   - If it were 0, then $x \mid g$, so the last (few) bits of $c = mx^n \bmod g$ were always 0, which would reduce the number of useful bits of c.
   - Then all one-bit errors are detected (because then $e = x^k$, which is only divisible by x^l).
   - Then all such errors are detected where the failed bits are within a single n-bit window.
3. If $x+1 \mid g$ (or equivalently, g has an even number of 1 bits), then all errors with an odd number of bits are detected (because the multiples of g also have an even number of 1 bits).

Some frequently used generator polynomials in practice:

- CRC-16: $g = 1\,10000000\,00000101$ (e.g. USB);
- CRC-32: $g = 1\,00000100\,11000001\,00011101\,10110111$ (e.g. Ethernet, Wi-Fi).

Also, it can be easily shown that CRC-1 with the generator polynomial $g = 11$ ($g = x + 1$) is equivalent to the parity bit method described above.

3 Advanced Encryption Standard (AES)

3.1 Structure of finite fields

We have discussed field extensions and finite fields in the previous part (Algebraic structures). We have seen that all finite fields have p^k elements, where p is a prime number and $k \geq 1$, and they are extensions of the corresponding field $\mathbb{Z}_p$. Now let's discuss how the elements of these fields look like:

Theorem 3.1. *Let $\mathbb{F}_{p^k}$ be a finite field, and let θ be one of its primitive elements (i.e. for which $\mathbb{F}_{p^k} = \mathbb{Z}_p(\theta)$). Then all $\alpha \in \mathbb{F}_{p^k}$ elements can be written uniquely in the following form:*

$$\alpha = a_0 + a_1\theta + a_2\theta^2 + \ldots + a_{k-1}\theta^{k-1},$$

where $a_0, a_1, \ldots, a_{k-1} \in \mathbb{Z}_p$.

In other words, each element of the field corresponds to a polynomial of degree $\leq k-1$ in $\mathbb{Z}_p[x]$ (more precisely, it is its value at $x = \theta$). For example, as demonstrated in the previous part, $\mathbb{F}_9 = \mathbb{Z}_3(\sqrt{2})$, and all $\alpha \in \mathbb{F}_9$ elements can be uniquely written as $\alpha = a + b\sqrt{2}$, where $a, b \in \mathbb{Z}_3$. This theorem also explains why finite fields have prime power number of elements: the polynomial can have k coefficients, and each one has p possible values, which gives p^k possibilities.

Finite fields can have more than one primitive element. For example, $\mathbb{F}_9 = \mathbb{Z}_3(\sqrt{2}) = \mathbb{Z}_3(\overline{2}\sqrt{2} + \overline{1})$, i.e. not only $\sqrt{2}$, but also $\theta = \overline{2}\sqrt{2} + \overline{1}$ can be used to write all elements of $\mathbb{F}_9$ in the form $a + b\theta$ (but of course with different a and b). For example, if $\alpha = \sqrt{2} + \overline{1} \in \mathbb{F}_9$, then by $\sqrt{2}$, $\alpha = \overline{1} + \overline{1}\sqrt{2}$, and by θ, $\alpha = \overline{2} + \overline{2}\theta$.

When creating a finite field in Sage, we can specify which primitive element is used to represent the elements. It must be given by its minimal polynomial, i.e. the irreducible polynomial for which the primitive element is a root of (e.g. $\sqrt{2}$ is a root of $x^2 - 2$, and $\theta = \overline{2}\sqrt{2} + \overline{1}$ is a root of $x^2 + x + 2$):

```
sage: F = GF(9, "g", modulus = x^2 - 2)   # "g" denotes the primitive element
sage: F.list()
[0, g + 2, g, 2*g + 2, 2, 2*g + 1, 2*g, g + 1, 1]
sage: g = F.gen()
sage: g^2
2
```

Similarly to polynomials, we have a simplified syntax to create both `F` and `g` simultaneously:

```
sage: F.<g> = GF(9, modulus = x^2 - 2)   # F = GF(...) and g = F.gen()
sage: g^2
2
```

3.2 Overview of AES

The *Advanced Encryption Standard (AES)* (originally called Rijndael) is one of the most popular cryptographic algorithms. It is used in a wide variety of places, including internet communication (e.g. HTTPS), disk encryption and the classified documents of the US government. Its popularity comes from not only its security, but also its high speed.

AES is a symmetric-key encryption, i.e. it uses the same key to encrypt and decrypt the message (as opposed to e.g. the RSA, which is asymmetric, see the part about RSA).

The full AES algorithm is not detailed here, only some of its mathematically interesting parts.

The message is split into 128-bit blocks, which are encrypted separately. Each such block has 16 bytes, which are written as a 4×4 matrix in column major order:

$$B = \begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}.$$

The bytes are represented as elements of the finite field $\mathbb{F}_{2^8} = \mathbb{F}_{256}$, using the formula in the above theorem, where the coefficients of the polynomial are the 8 bits of the byte, and the primitive element $\theta \in \mathbb{F}_{256}$ is the root of the fixed minimal polynomial $m = x^8 + x^4 + x^3 + x + 1$. For example, the byte 01001011 corresponds to $\theta^6 + \theta^3 + \theta + 1 \in \mathbb{F}_{256}$.

One round of the encryption algorithm consists of four steps:

1. *SubBytes*: transforming each individual byte;
2. *ShiftRows*: shifting the rows cyclically;
3. *MixColumns*: multiplying the matrix by a constant matrix;
4. *AddRoundKey*: applying the key to the block.

The full algorithm runs these steps (with minor changes) 10-14 times (depending on the parameters). For decryption, the inverse steps are performed in reverse order. The four steps are described below.

3.3 AES: *SubBytes*

In this step, the bytes of the matrix are transformed indepentently using an invertible function $S : \mathbb{F}_{256} \to \mathbb{F}_{256}$, without changing their place in the matrix (i.e. $\forall i : b'_i := S(b_i)$). This transformation is also called *Rijndael S-box* (S = substitution).

The transformation S works in two steps ($S(b) = S_2(S_1(b))$):

1. First, take the multiplicative inverse of the element in $\mathbb{F}_{256}$: $S_1(b) := b^{-1}$ – except for $b = 0$, which is not invertible, so let $S_1(0) := 0$.
2. The second step is an affine transformation, which is an exception in that it is not calculated in $\mathbb{F}_{256}$, but in another representation of the elements. It can be stated in two equivalent ways:
   (a) Using the polynomial representation of $b \in \mathbb{F}_{256}$ ($b = b_7x^7 + \ldots + b_1x + b_0 \in \mathbb{Z}_2[x]$):
   $$S_2(b) := \big(b \cdot (x^4 + x^3 + x^2 + x + 1)\big) \bmod (x^8 + 1) \;\; + \;\; (x^6 + x^5 + x + 1).$$
   (b) Using the coefficient vector of $b \in \mathbb{F}_{256}$ (as $b \in \mathbb{Z}_2^8$), by a matrix multiplication ($\mathbb{Z}_2^{8\times 8}$):
   $$S_2(b) := \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$
   It is important to note again that for both representations, the bits (i.e. the polynomial coefficients or the vector components) are still in $\mathbb{Z}_2$, i.e. they are calculated modulo 2.

This S-box gives the essence of AES: it is a *nonlinear* transformation (a linear transformation would be just a matrix multiplication), therefore it is very difficult to follow what happens with the bytes. This transformation has only 256 possible values, so it is usually precalculated and stored in a table, to make encryption more efficient.

3.4 AES: *ShiftRows*

This step simply rearranges the bytes within the rows of the matrix: the first row is unchanged, the second row is cyclically shifted left by one, the third row by two, and the fourth one by three. So it means the following rearrangement (the original columns are marked to make it easier to see):

$$\begin{pmatrix} b_0 & b_4 & \underline{b_8} & \underline{\boldsymbol{b_{12}}} \\ b_1 & b_5 & \underline{b_9} & \underline{\boldsymbol{b_{13}}} \\ b_2 & b_6 & \underline{b_{10}} & \underline{\boldsymbol{b_{14}}} \\ b_3 & b_7 & \underline{b_{11}} & \underline{\boldsymbol{b_{15}}} \end{pmatrix} \longrightarrow \begin{pmatrix} b_0 & b_4 & \underline{b_8} & \underline{\boldsymbol{b_{12}}} \\ b_5 & \underline{b_9} & \underline{\boldsymbol{b_{13}}} & b_1 \\ \underline{b_{10}} & \underline{\boldsymbol{b_{14}}} & b_2 & b_6 \\ \underline{\boldsymbol{b_{15}}} & b_3 & b_7 & \underline{b_{11}} \end{pmatrix}$$

This permutation step ensures that the columns are mixed with each other, instead of just being manipulated independently.

3.5 AES: *MixColumns*

In this step, the 4×4 matrix is simply multiplied by another matrix (both are in $\mathbb{F}_{256}^{4\times4}$):

$$B' := \begin{pmatrix} \theta & \theta{+}1 & 1 & 1 \\ 1 & \theta & \theta{+}1 & 1 \\ 1 & 1 & \theta & \theta{+}1 \\ \theta{+}1 & 1 & 1 & \theta \end{pmatrix} B.$$

Reminder: $\theta \in \mathbb{F}_{256}$ is the primitive element of the field. Using bit sequences, $1 = 00000001$, $\theta = 00000010$ and $\theta{+}1 = 00000011$. (These values were chosen to make calculation easier.)

The name of this step (*MixColumns*) indicates that this matrix multiplication is equivalent to multiplying each column vector of B separately by the same constant matrix.

This step ensures that the elements are not only rearranged and transformed separately, but they are also combined with each other (within the column).

3.6 AES: *AddRoundKey*

This step adds the key of the current round to the block. The key is also a 128-bit sequence, which can be written in the same matrix form, so this step is just a matrix addition in $\mathbb{F}_{256}^{4\times4}$. Or alternatively, using bit sequences, it is a bitwise "exclusive or" (XOR) operation of the two 128-bit sequences.

The key is different in each round of the encryption (hence the name *RoundKey*), and it can be calculated from the master key, which is the input of the algorithm, and its length is between 128 and 256 bits (depending on the parameters). These calculations are omitted here.

The key is what makes the algorithm secure: anyone who knows it can easily decrypt the message, but for others, the encrypted text is just a meaningless sequence of bits. Without knowing the key, one has to check all of its possible values, which, even for the shortest 128-bit key, means $2^{128} = 340282366920938463463374607431768211456$ possibilities. This is practically impossible.

4 Author

These lecture notes were written by Uray M. János, partially using Nagy Ádám's notes in Hungarian:
https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa
Please report errors and suggestions here: uray.janos@inf.elte.hu.