CDM

Applications of Finite Fields

Klaus Sutner

Carnegie Mellon University

Fall 2023

Warning

This lecture contains a number of “dirty tricks,” rather than the supremely elegant and deeply rooted mathematical ideas that you have become accustomed to in this class.

Think of it as the engineering approach: some clever trick may just be an opportunistic hack, but, boy, does it ever work well.

What To Implement

In order to implement a finite field we need a data structure to represent field elements and operations on this data structure to perform

- addition
- multiplication
- reciprocals, division
- exponentiation

Subtraction is essentially the same as addition and requires no special attention.

But, anything to do with multiplication is by no means trivial. In fact, there is an annual international conference that is dedicated to the implementation of finite fields.

WAIFI Conference

- Theory of finite field arithmetic:
    - Bases (canonical, normal, dual, weakly dual, triangular ...)
    - Polynomial factorization, irreducible polynomials
    - Primitive elements
    - Prime fields, binary fields, extension fields, composite fields, tower fields ...
    - Elliptic and Hyperelliptic curves
- Hardware/Software implementation of finite field arithmetic:
    - Optimal arithmetic modules
    - Design and implementation of finite field arithmetic processors
    - Design and implementation of arithmetic algorithms
    - Pseudorandom number generators
    - Hardware/Software Co-design
    - IP (Intellectual Property) components
    - Field programmable and reconfigurable systems
- Applications:
    - Cryptography
    - Communication systems
    - Error correcting codes
    - Quantum computing

Classes of Implementations

Our characterization of finite fields as quotients of polynomial rings provides a general method of implementation, and even a reasonably efficient one.

Still, it is a good idea to organize things into several categories, with different algorithmic answers.

- Prime fields $\mathbb{Z}_p$
- Characteristic 2, $\mathbb{F}_{2^k}$
- General case $\mathbb{F}_{p^k}$

For simplicity let's assume that the characteristic p is reasonably small so that arithmetic in $\mathbb{Z}_p$ is $O(1)$.

Note that we are not necessarily looking for asymptotic speed-ups here, constant factors can be quite interesting.

Trick 1: Montgomery Reduction

We know how to handle $\mathbb{F}_p = \mathbb{Z}_p$: we need standard addition and multiplication in combination with remainder computations. The Extended Euclidean Algorithm can be used to compute inverses. No problem.

Yet even in $\mathbb{Z}_p$ there is room for some clever computational tricks.

Here is one way to lower the cost of multiplication in the field. This requires a pre-computation and is only of interest when multiple operations are needed.

Suppose we have characteristic $p > 2$.

- Pick $R > p$ coprime, typically $R = 2^k$.
- Represent modular number x by $\widehat{R}(x)$ where

$$\begin{array}{rccc} \widehat{R}: & \mathbb{Z}_p & \longrightarrow & \mathbb{Z}_p \\ & x & \longmapsto & Rx \bmod p \end{array}$$

  Since p and R are coprime, the map $\widehat{R}$ this is a bijection: we are just permuting the field elements.

No Morphism

We have

$$\widehat{R}(a+b) = \widehat{R}(a) + \widehat{R}(b) \pmod{p}$$

but, unfortunately:

$$\widehat{R}(a \cdot b) = \widehat{R}(a) \cdot \widehat{R}(b) \cdot R^{-1} \pmod{p}$$

To get mileage out of $\widehat{R}$ we need a cheap way to perform reductions

$$x \rightsquigarrow xR^{-1} \bmod p$$

Cheap here just means that we should do better than doing vanilla $\bmod p$ operations.

Performing Montgomery Reduction

How can we do the reduction cheaply?

- Precompute $\alpha = -p^{-1} \bmod R$.
- Given an integer $0 \le x < Rp$ (not a modular number), compute $x_0 = x\alpha \bmod R$.
- Then $(x + x_0 p)/R$ is an integer and

$$(x + x_0 p)/R = xR^{-1} \bmod p.$$

This is more compelling by looking at the actual code.

Montgomery Reduction, Code

Here is a typical implementation for $R = 2^{16}$ (and, of course, $p > 2$).

```
#define MASK  65535UL     // 2^16 - 1
#define SHFT  16;

// precompute alpha (Euclidean algorithm)

x0   = x & MASK;
x0   = (x0 * alpha) & MASK;
x   += x0 * p;
x  >>= SHFT;
return( x > p ? x-p : x );
```

The only expensive operations are two multiplications.

Example $p = 17$

Let $p = 17$ and pick $R = 64$ so that $\alpha = 15$.

Here are the field elements $\neq 0$ and their representations:

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$aR \bmod p$	13	9	5	1	14	10	6	2	15	11	7	3	16	12	8	4

For example, for $5 \times 7 = 1 \bmod p$ we get $x = 14 \times 6 = 84$ and thus

$$\begin{aligned} x_0 &= 84 \times 15 \bmod 64 = 44 \\ (x + x_0 p)/R &= 832/64 = 13 \end{aligned}$$

Indeed 13 corresponds to 1 in our representation, so everything is fine.

Trick 2: Logarithm Tables

Given a generator g for $\mathbb{F}^\times$ in a reasonably small field we can pre-compute and store a logarithm table

$$(h, i) \in \mathbb{F}^\times \times \mathbb{N} \qquad \text{where } h = g^i$$

We also need the inverse table with entries (i, h). For example, for $\mathbb{F}_{2^8}$ the tables require only 512 bytes of memory.

Multiplication is then reduced to table lookups and some (machine-) integer addition and thus very fast.

This technique is used for example in the cryptographic system AES, see below.

Characteristic 2

These log tables can also be used to speed up computation in larger fields of characteristic 2: instead of dealing directly with $\mathbb{F}_{2^k}$ we think of it as an extension of $\mathbb{F}_{2^\ell}$ where ℓ is small. We have a tower of fields

$$\mathbb{F}_2 \subseteq \mathbb{F}_{2^\ell} \subseteq \mathbb{F}_{2^k}$$

For example, suppose we are calculating in $\mathbb{F}_{2^8}$, so a typical element is $a = x^7 + x^6 + x^3 + 1$ or, as a coefficient vector over $\mathbb{F}_2$, $(1, 1, 0, 0, 1, 0, 0, 1)$.

By grouping the coefficients into blocks of 2 we get $(3, 0, 2, 1)$ and can think of a as $3z^3 + 2z + 1 \in \mathbb{F}_{2^2}[z]$.

One can verify that ℓ needs to be a divisor of k for $\mathbb{F}_{2^\ell}$ to be a subfield of $\mathbb{F}_{2^k}$.

A popular choice is $\ell = 8$ so that the coefficients in the intermediate fields are bytes.

Speedup

One uses a log table for the intermediate field $\mathbb{F}_{2^\ell}$, so arithmetic there is very fast (comparable to the prime field $\mathbb{F}_2$).

The main field can now be implemented by using polynomials over $\mathbb{F}_{2^\ell}[x]$ of degree less than k/ℓ rather than the original degree less than k.

So there is a trade-off: the ground field becomes more complicated (though we can keep arithmetic there very fast using tables), but the degree of the polynomials decreases, so we are dealing with smaller bases.

Finding “optimal” implementations of finite fields is not easy.

Trick 3: Polynomial Basis Representation

We have seen that $\mathbb{F}_{p^k}$ can be represented by a quotient ring $\mathbb{Z}_p[x]/(f)$ where $f \in \mathbb{Z}_p[x]$ is irreducible of degree k. Let $q = p^k$ be the size of the field.

As we have seen, the elements of the field are essentially all polynomials of degree less than k. We can represent these polynomials by coefficient vectors of modular numbers:

$$\mathbb{F}_{p^k} = \underbrace{\mathbb{Z}_p \times \mathbb{Z}_p \times \ldots \times \mathbb{Z}_p}_{k}$$

This representation provides addition at $O(k)$ time.

Multiplication is more complicated and comes down to ordinary polynomial multiplication followed by reduction modulo f.

More on Multiplication

By linearity, it suffices to figure out how to multiply $g(x)$ by $x \cong (0, 0, \dots, 1, 0)$.

Writing $f(x) \in \mathbb{F}_2[x]$ in coefficient notation:

$$f(x) = x^k + c_{k-1}x^{k-1} + \ldots + c_1 x + c_0$$

where $c_i \in \mathbb{F}_2$. Given

$$g(x) = a_{k-1}x^{k-1} + \ldots + a_1 x + a_0$$

we get $g(x) \cdot x$ by "reducing" the leading term in

$$g(x) \cdot x = a_{k-1}x^k + a_{k-2}x^{k-1} + \ldots + a_1 x^2 + a_0 x$$

according to f.

The Reduction

If $a_{k-1} = 0$ we simply get

$$a_{k-2}x^{k-1} + \ldots + a_1 x^2 + a_0 x$$

Otherwise we have to update some of the coefficients:

$$(a_{k-2} + c_{k-1})x^{k-1} + \ldots + (a_1 + c_2)x^2 + (a_0 + c_1)x + c_0.$$

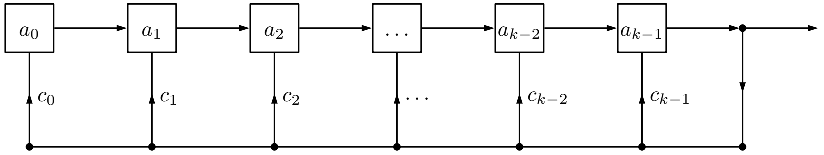
Proposition

$$\boldsymbol{a} \cdot x = \sum_{i=0}^{k-1} (a_{i-1} + a_{k-1}c_i)\, x^i$$

where $a_{-1} = 0$.

Linear Feedback Shift-Registers

In characteristic 2 this all boils down to shifting and xor-ing bit-vectors.



Voila, it's a feedback shift register.

Not so fast . . .

The gizmo is indeed a feedback shift register, but a so-called Galois feedback shift register, as opposed to the slightly more natural Fibonacci feedback shift registers that correspond directly to linear recurrences.

The registers still shift to the right, but the other arrows are reversed (we used to have taps to get a feedback value, now we are flipping some of the registers).

Reciprocals

Suppose $h \neq 0 \in \mathbb{F}$, we need to compute its inverse $1/h$.

Since f is irreducible, this means that h and f are coprime polynomials and we can use the extended Euclidean algorithm to find cofactors $g_1, g_2 \in \mathbb{F}[x]$ such that

$$g_1 h + g_2 f = 1$$

But then cofactor g_1 is the inverse of h modulo f.

Logically this is the same as computing inverses in $\mathbb{Z}_p$, but note that polynomial arithmetic is more expensive than integer arithmetic.

More Reciprocals

Note that we can also use exponentiation to compute the inverse of $h \in \mathbb{F}^\star$: the multiplicative group has size $n = p^k - 1$ so that

$$h^{-1} = h^{n-1}$$

Of course, this approach is computationally fatal unless we have a lightning fast exponentiation algorithm (see below). Specifically, we would like to use the standard divide-and-conquer algorithm based on squaring.

Normal Basis Representation

Thinking of a finite field as a vector space over the prime field naturally leads to the basis

$$1, \alpha, \alpha^2, \ldots, \alpha^{k-1}$$

where $\alpha = x/(f)$ is the “magic root” of f.

Perfectly fine, but a basis can be chosen in many other ways. An element $\alpha \in \mathbb{F}_{p^k}$ is normal if its Frobenius conjugates

$$\alpha, \alpha^p, \alpha^{p^2}, \ldots, \alpha^{p^{k-1}}$$

are linearly independent in $\mathbb{F}$. They are then called the normal basis (generated by α).

Thus, a normal basis is the orbit of a suitable field element under the Frobenius homomorphism.

But Why?

Perhaps surprisingly, this is useful to speed up exponentiation.

For simplicity, consider only characteristic $p = 2$. Then squaring an element $\sum a_i \alpha^{2^i}$ in normal representation comes down to a cyclic shift of coefficients:

$$(a_0, a_1, \ldots, a_{k-2}, a_{k-1}) \mapsto (a_{k-1}, a_0, a_1, \ldots, a_{k-2})$$

Assuming that the exponent e is less than 2^k we can calculate a^e in at most $k - 1$ multiplications of field elements that are trivial to generate.

Trick 4: Redundant Representations

Here is another wild idea. We want to compute in $\mathbb{F}_{p^k}$, usually given as $\mathbb{F}_p[x]/(f)$. Instead we use a subring of the non-field

$$\mathbb{F}_p[x]/(x^n - 1)$$

where $n \geq k$ is "suitably chosen."

This is not a typo. The huge advantage is that the reduction operation here is trivial. Of course, we have to be able to somehow identify the subring.

Example

Consider $\mathbb{F}_2[x]/(x^3 + 1)$. The elements

$$0, x^2 + x, x + 1, x^2 + 1$$

form a subring isomorphic to $\mathbb{F}_4$.

Factoring Polynomials

Of course, there is a little problem in all our approaches: just to get off the ground we need an irreducible polynomial, typically obtained by factoring

$$x^{p^k} - x = x(x^{p^k-1} - 1) \in \mathbb{F}_p[x].$$

This is quite difficult even for moderately large p^k. Fast factoring algorithms for polynomials were one of the big accomplishments of computer algebra in the 1960s.

Elwyn R. Berlekamp

Algebraic Coding Theory

McGraw-Hill 1968

Logarithms in a Group

Suppose G is some cyclic finite group with generator g and cardinality n. We can easily exponentiate in G, the operation

$$e \leadsto g^e$$

takes $O(\mathtt{M} \log e)$ steps where $\mathtt{M}$ is the cost of a single multiplication in G. But going backwards is apparently hard in many groups:

Given $a \in G$, find e such that $g^e = a$.

This is known as the discrete logarithm problem.

Of course, $e = \log_g a$ is trivially computable by a brute force search, but we are here interested in efficient computation when n is large.

Diffie and Hellman

In 1976, Whit Diffie and Martin Hellman seized on this apparent difficulty to propose a cryptographic scheme that promises

Secure communication using only insecure channels.

This almost seems logically impossible: if the eavesdropper has complete knowledge about the encryption method used and has full access to the communication channel it would seem that we cannot keep any secret.

Note that this is the idea that gave rise to RSA.

The group used in Diffie/Hellman is the multiplicative group of a finite field $\mathbb{F}^{\star}$: we can choose the size and implement the group operation quite efficiently.

Diffie/Hellman Algorithm

- Alice and Bob agree on generator β in some finite field $\mathbb{F} = \mathbb{F}_{p^k}$.
- Alice generates random number x, computes $a = \beta^x$ in $\mathbb{F}$, sends a to Bob.
- Bob generates random number y, computes $b = \beta^y$ in $\mathbb{F}$ sends b to Alice.
- Both Alice and Bob can now compute

$$c = \beta^{xy} = a^y = b^x$$

and use it as a secret key (for some other encryption algorithm).

Evilestdoer Charlie knows the algorithm, $\mathbb{F}$, β, a and b, but **not** x and y.

Apparently, Charlie cannot determine c without a huge search, so we only need to make $\mathbb{F}$ large enough to foil his efforts.

A Challenge

Diffie/Hellman seems to hold the promise of secure communication over utterly insecure channels.

Great, but now there is an immediate challenge: can one break Diffie/Hellman, at least in some special cases.

"Break" would mean that we are happy to invest quite a bit of computation, just not the full brute-force exponential seach that seems to be necessary to destray the schema.

Shanks Baby/Giant Steps

In other words: is there anything we an do to speed up computation of logarithms in a finite field?

Suppose g is a generator of the multiplicative subgroup and $a \neq 0$ is some element in a field of size $q = p^k$.

Let $m = \lceil \sqrt{q} \rceil$ an compute two lists

- ag^{-i} where $0 \leq i < m$, and
- g^{mj} where $0 \leq j < m$.

Then check for a common entry in the two lists: this produces $ag^{-i} = g^{mj}$, whence $a = g^{mj+i}$.

So we have essentially written the logarithm as a two-digit number in base m.

Efficiency

To check for a common element we can use hashing. Of course, one need not wait till the lists are complete to check for a match.

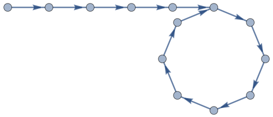
The baby-step/giant-step method requires $O(\sqrt{q}\log q)$ time and space $O(\sqrt{q})$.

This may not seem overly impressive, but it is a huge improvement over the standard $O(q)$ time algorithm (though that runs constant space).

Note that a cryptographic attack may well be worth this much computation. The NSA sure won't mind.

Pollard’s Rho Method

This should be called Pollard’s Lasso method (in particular since the second algorithm in the paper is about “catching kangaroos”), but it’s too late now.



A Rohrschach test:

- If you have a classical education, you will see a ρ.
- If you’re a cowboy, you will see a lasso.

Random Maps

The motivation for this method is a bit strange. Consider a random function $f : A \to A$ where A has size n.

Then the expected value of some key parameters of the functional digraph of f are as follows:

# components	$\frac{1}{2}\log n$
# leaf nodes	$e^{-1}n$
# recurrent nodes	$\sqrt{\pi n/2}$
transient length	$\sqrt{\pi n/8}$
period length	$\sqrt{\pi n/8}$

The expected lengths of the longest transient/cycle are also $c_{1/2}\sqrt{n}$ where $c_1 \approx 1.74$ and $c_2 \approx 0.78$.

Computing Transient and Period

For simplicity we can think of the expected value of transient length t and period length p of a random point a in A as $\sqrt{n}$.

We know an elegant algorithm to compute these parameters: Floyd's trick. More precisely, we can compute t and p in expected time $O(\sqrt{n})$ using $O(1)$ space (we only need to store a small constant number of elements in A).

Here comes the **Wild Idea:**

Can we compute a sequence (x_i) of elements as in Shanks' algorithm that behaves like a (pseudo-) random sequence so that $x_i = x_{2i}$ allows us to compute a discrete logarithm?

The Map

We need a “random” map.

To this end we first split the group G into three sets G_1, G_2 and G_3 of approximately equal size (sets, not subgroups, so this will be easy in practical situations). Any ham-fisted approach will do.

Now, given a generator g and some element a, define $f : G \to G$ as follows:

$$f(x) = \begin{cases} gx & \text{if } x \in G_1, \\ x^2 & \text{if } x \in G_2, \\ ax & \text{otherwise.} \end{cases}$$

Of course, f is perfectly deterministic (at least given the partition of G).

The Orbits

Consider the orbit (x_i) of 1 under f, $x_0 = 1$.

Clearly, all the elements have the form $a^{\alpha_i} g^{\beta_i}$ and the exponents are updated according to

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} (\alpha_i, \beta_i + 1) & \text{if } x \in G_1, \\ (2\alpha_i, 2\beta_i) & \text{if } x \in G_2, \\ (\alpha_i + 1, \beta_i) & \text{otherwise.} \end{cases}$$

Since the partition of G is random, the three steps are chosen randomly (more or less).

Use Floyd

Moreover, using Floyd's method we can find the minimal index e such that $x_e = x_{2e}$, i.e.,

$$a^{\alpha_e} g^{\beta_e} = a^{\alpha_{2e}} g^{\beta_{2e}}$$

But then

$$a^{\alpha_e - \alpha_{2e}} = g^{\beta_{2e} - \beta_e}$$

This equality does not directly solve the discrete logarithm problem but it can help a lot to compute the discrete logarithm.

Note that for cryptographic application any such weakness is potentially fatal: a good method must be secure under any and all circumstances.

Example: $\mathbb{Z}_{999959}$

Consider the multiplicative group of $\mathbb{Z}_p$ where $p = 999959$, and let $g = 7$ and $a = 3$.

Running the algorithm produces $e = 1174$ and $x_e = 11400$, plus the identity

$$3^{310686} = 7^{764000} \pmod{p}$$

A classical case of "close, but no c igar": we want to compute the logarithm of 3 with respect to generator 7, some η such that: $3 = 7^{\eta} \pmod{p}$.

But perhaps we squeeze a bit more information out of this identity? The identity lives over $\mathbb{Z}_p^{\star}$.

Magic

Use the Extended Euclidean algorithm to get

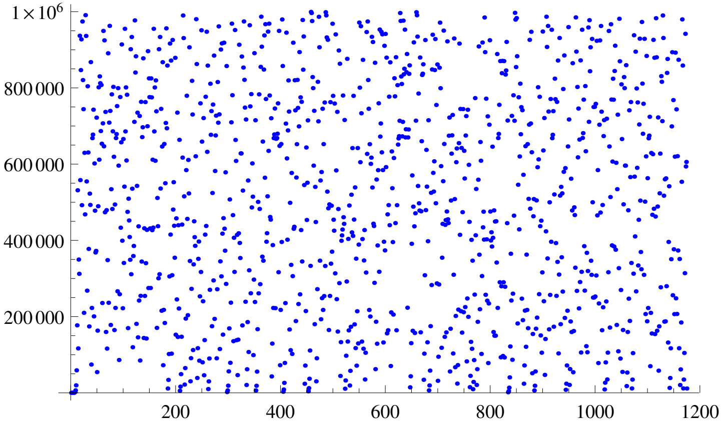
$$\gcd(310686, p-1) = 2 = 148845 \cdot 310686 - 46246 \cdot 999958$$

Raising 3^{310686} to the 148845 power mod p and simplifying produces

$$3^2 = 7^{356324} \pmod{p} \qquad \text{and} \qquad 3 = \pm 7^{178162} \pmod{p}$$

It turns out that in $\mathbb{Z}_p$: $\log_7 3 = 178162$.

How Random Is It?



The History

The now classical DES (data encryption standard) was officially adopted in 1977. It is based on–rather too short–keys of length 56 bits and has since fallen prey to Moore's law: DES can now be broken in a distributed attack in a matter of hours.

In September 1997, NIST issued a Federal Register notice soliciting an unclassified, publicly disclosed encryption algorithm.

15 candidate algorithms were submitted and closely scrutinized. In 2000 the NIST selected the Rijndael algorithm by Joan Daemen and Vincent Rijmen as the new standard.

It is now enshrined in the Federal Information Processing Standard (FIPS) for the Advanced Encryption Standard, FIPS-197, see

http://csrc.nist.gov/CryptoToolkit/aes

Stream Ciphers

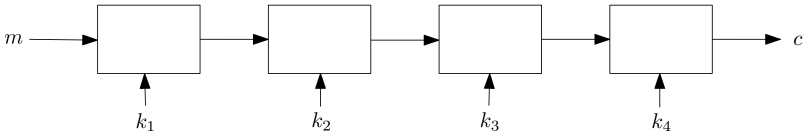
DES and AES are both based on ideas that first appeared in Lucifer, an encryption system developed mainly by IBM: the input message is chopped into fixed-size block of bits, which are then clobbered using a key.

- Use combinatorics and algebra to mangle a block of bits.
- Cleverly incorporate the key into this mangling process.
- Use multiple rounds to make sure the final result is sufficiently complicated.

Each round uses a subkey (aka round key) that is generated from the main key.

And, of course, everything has to be easily reversible when the key is known.

Again: Iteration



Each coding box is relatively simple, and may not provide a safe encoding. But a sufficiently long chain is hard to crack without knowledge of the master key.

Lucifer

There are several patented variants, all based on substitution-permutation networks (so-called Feistel networks) that mangle bits and mix in the key in some clever way. Up to 16 rounds are used to foil attacks.

The block size and key size vary from 48 to 128 bits.

A pleasant feature is that decryption is very similar to encryption, so hardware can be reused.

Alas, the devil is in the details[†], and Lucifer suffered from security issues.

[†]No pun intended.

DES Outline

DES encrypts blocks of 64 bits, using a key of 56 bits.

A 64-bit input block is permuted and then split into two 32-bit blocks (L_0, R_0).

These blocks are then mangled in several rounds according to

$$(L_{i+1}, R_{i+1}) = (R_i, L_i \oplus f(R_i, K_i))$$

Here K_i is a key derived from the original key $K \in \mathbf{2}^{56}$ and $f : \mathbf{2}^{32+48} \to \mathbf{2}^{32}$ is a carefully constructed Boolean map. Tempting, but we won't go there.

The final output is then obtained from (L_{16}, R_{16}).

Eyeballing It

2^{56}	7.2×10^{16}
2^{128}	3.4×10^{38}
2^{256}	1.2×10^{77}

Even with a million processors, the longer keys cannot be brute-forced.

Rijndael

AES Infrastructure

AES encrypts blocks of 128 bits, using a cipher key of 128 (or 192, 256 bits).

Bit-sequences in AES are always divided into bytes, 8-bit blocks.

Finite fields and/or polynomials are used in two places:

- We can think of these bytes as coefficient vectors of elements in $\mathbb{F}_{2^8}$ where the irreducible polynomial for the multiplicative structure is chosen to be

$$f(x) = x^8 + x^4 + x^3 + x^2 + 1$$

- 4-byte vectors are construed as polynomials in $\mathbb{F}_{2^8}[z]/(z^4 + 1)$.

The Appointed Rounds

The algorithm first xors with a subkey, and then proceeds in 10 rounds (actually, the number depends on the key size, but let's just focus on 128-bit keys). As in DES, each round mangles the bits some more (the final round is slightly different, but we will ignore this).

Abstractly, a single round looks like so:

- byte substitution
- shifting rows
- mixing columns
- add key

Terminology

The row/column terminology comes from thinking of the initial input as being given by a 4×4 matrix of bytes (for a total of 128 bits; in reality the input is broken into corresponding pieces).

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

So the current state is described by such a matrix which we may think of as a 4×4 matrix over our favorite field $\mathbb{F}_{2^8}$.

(1) Byte Substitution

Define the patched inverse of an element $\boldsymbol{a} \in \mathbb{B}$ to be

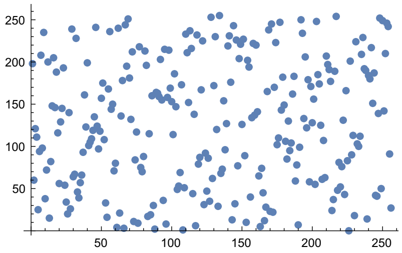
$$\overline{\boldsymbol{a}} = \begin{cases} 0 & \text{if } \boldsymbol{a} = 0, \\ \boldsymbol{a}^{-1} & \text{otherwise.} \end{cases}$$

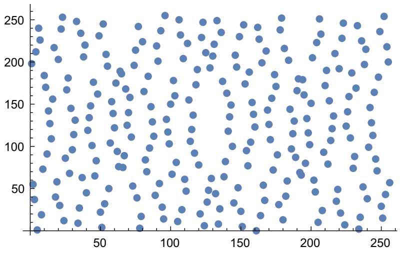
Define an 8×8 bit-matrix and 8-bit vector as follows

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad v = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Then byte substitution is given by the (almost) affine, reversible map $\boldsymbol{a} \mapsto A\,\overline{\boldsymbol{a}} + v$, applied to each byte separately.

Byte Substitution Picture





On the right, we simply use a rather than the patched inverse as on the left.

(2) Shift Rows

Replace the state matrix by the row-shifted version

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{pmatrix}$$

More precisely, we shift the ith row by i places to the left (assuming 0-indexing).

The next operation is a bit more complicated.

Aside: Notation

For phase (3) we will use the same notation as in the Rijndael specification.

To denote an element of $\mathbb{B}$, we think of a byte as two hexadecimal digits.

So, for example, `D4` corresponds to the field element

$$z^7 + z^6 + z^4 + z^2 \bmod \tau$$

in polynomial notation, or 11010100 as a coefficient vector.

Since the byte field $\mathbb{B}$ is quite small, we can easily use lookup tables to make all the field operations very fast.

(3) Mixing Columns

Consider the polynomial (coefficients are written as two hex digits)

$$g(z) = 03\,z^3 + 01\,z^2 + 01\,z + 02 \in \mathbb{B}[z]$$

We can think of each column in the state matrix as another polynomial in $\mathbb{B}[z]$, so in this phase we multiply the column polynomial by g, and then reduce modulo $z^4 + 1$.

Since these operations are all linear, this all comes down to a single matrix multiplication over $\mathbb{B}$:

$$\boldsymbol{c} \rightsquigarrow \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \boldsymbol{c}$$

(4) Add Key

Well . . .

The Standard

Take a look at the NIST specifications for cryptography and Rijndael:

ToolKit

Rijndael

There are lots of implementation details as well as a careful discussion how to decrypt Rijndael encrypted message.

One pleasant aspect: the decryption operations are very similar to encryption, so essentially the same hardware can be used.

Keys and Decryption

Claim: All four phases are reversible.

At each round, a subkey is xor-ed with the state matrix; all the subkeys are 128 bits. We will not discuss how the subkeys are generated from the original key. Incidentally, proper key management is a huge problem in cryptography.

The documents at the links also contain lots of implementation details as well as a careful discussion how to decrypt Rijndael encrypted message. Note that the inverse operations are very similar to the encryption operations, so essentially the same hardware can be used.

Decryption Phases

The xor operation with the keys is self-inverse (we're in characteristic 2), so step 4 in each round is easily undone by anyone with access to the key.

States 1–3 are also invertible, none of the operations loses information.

As with DES, the decryption process is quite similar to encryption.

With appropriate CPU support the throughput is quite high (hundreds of MB/s).