

CDM

Applications of Finite Fields

Klaus Sutner
Carnegie Mellon University

Fall 2011

Outline:

Outline

- 1 Implementing Finite Fields
- 2 Discrete Logarithms
- 3 Advanced Encryption Standard

Outline:

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.

Think of it as the engineering approach: it's an opportunistic hack, but boy, does it work well.

Implementing Finite Fields:

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.

Implementing Finite Fields:

International Workshop on the Arithmetic of Finite Fields

- 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

Implementing Finite Fields:

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.

However, it is of interest to take a closer look at special cases:

- 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 $O(1)$.

Even speed-ups by a constant factor can be quite interesting.

Here is a bag of assorted dirty tricks.

The Prime Field and Montgomery Reduction

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

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

Here is one way to lower the cost of multiplication in the field.

- Pick $R > p$ coprime, typically $R = 2^k$.
- Represent $a \in \mathbb{F}_p$ by $aR \bmod p$.
- Implement a cheap way to perform reductions:
from $0 \leq x < Rp$ to $xR^{-1} \bmod p$.
- Use these reductions in multiplications:
 $aR \cdot bR \rightsquigarrow abR^2 \rightsquigarrow abR$.

Montgomery Reduction

How can we do the reduction cheaply?

- Precompute $\alpha = -p^{-1} \bmod R$.
- Given $0 \leq x < Rp$ compute $x_0 = x\alpha \bmod R$.
- Then $(x + x_0p)/R$ is an integer and

$$(x + x_0p)/R = xR^{-1} \bmod p.$$

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
#define SHFT  16;

// precompute alpha

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

$$x_0 = 84 \times 15 \bmod 64 = 44$$

$$(x + x_0p)/R = 832/64 = 13$$

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

Logarithm Tables

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

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

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^ℓ} where ℓ is small.

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

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^ℓ} 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^ℓ} , 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.

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 $n = p^k$ be the size of the field.

So 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:

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

Addition is $O(k)$.

Multiplication comes down to ordinary polynomial multiplication followed by reduction modulo f .

More on Multiplication

By linearity, 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:

$$f(x) = x^k + c_{k-1}x^{k-1} + \dots + c_1x + c_0$$

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

$$g(x) = a_{k-1}x^{k-1} + \dots + a_1x + a_0$$

we get $g(x) \cdot x$ by "reducing"

$$g(x) \cdot x = a_{k-1}x^k + \dots + a_1x^2 + a_0x$$

according to f .

The Reduction

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

$$a_{k-2}x^{k-1} + \dots + a_1x^2 + a_0x$$

Otherwise upon reduction we have

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

Proposition

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

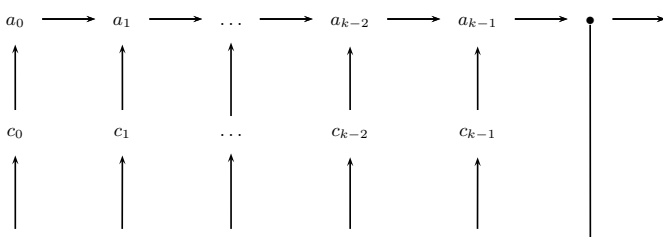
where $a_{-1} = 0$.

Linear Feedback Shift-Registers

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

In the eyes of an electrical engineer:

- linear operations
- delay
- feedback



Linear Feedback Shift-Registers, II

Here • is the output bit.

Basic operation: Shift the a -registers, and feed-back the new bit according to the taps c_i .

Of course, this is lightning fast! And cheap: electronic parts cost nothing.

So we can easily hardwire addition and multiplication for fields of characteristic 2.

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 .

This is very similar to 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 $h \in \mathbb{F}^*$: the multiplicative group has size $n = p^k - 1$ so that

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

and we can use exponentiation (using only multiplication).

Of course, this makes sense only if we have a fast exponentiation algorithm (see below).

Normal Basis Representation

The vector space basis used in the last approach is

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

where $\alpha = x/(f)$ is the "magic root" of f . This is certainly natural.

However, there are other choices. An element $\alpha \in \mathbb{F}_{p^k}$ is **normal** if its conjugates

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

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 $p = 2$. Then squaring an element in normal representation comes down to a cyclic shift of coefficients:

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

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 take at most $O(k^2)$ steps to generate.

Redundant Representations

Here is another wild idea. We want to compute in \mathbb{F}_{p^k} . Instead of using $\mathbb{F}_p[x]/(f)$ to represent the field which requires a relative complicated reduction modulo the irreducible polynomial f use a subring of

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

where $n \geq k$ is suitably chosen so that such a subring exists.

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

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

This is rather hard when p^k is (even moderately) large. Fast factoring algorithms for polynomials were one of the big accomplishments of computer algebra in the 60's.

Another Bible:

Elwyn R. Berlekamp
Algebraic Coding Theory
McGraw-Hill 1968

- Implementing Finite Fields
- Discrete Logarithms
- Advanced Encryption Standard

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

$$e \rightsquigarrow g^e$$

takes $O(M \log e)$ steps where 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}^* : we can choose the size and implement the group operation quite efficiently.

Diffie/Hellman

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

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

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

Eve/estdoer Charlie knows β, \mathbb{F}, 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.

Shanks Baby/Giant Steps

Is there anything we can 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 $m = \lceil \sqrt{q} \rceil$ and compute two lists

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

Then check if there is a common entry in the two lists.

If so we have $ag^{-i} = g^{mj}$ 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 sorting or hashing (operating on vectors of length k over \mathbb{Z}_p).

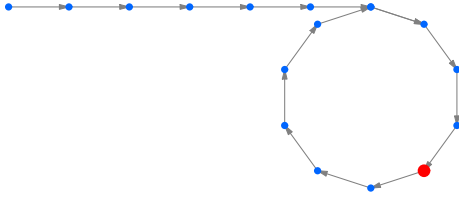
The baby-step/giant-step method requires $O(\sqrt{q} \log q)$ time and space $O(\sqrt{q})$.

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

Note that a cryptographic attack may well be worth this much computation.

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.



If you have a classical education you will see a ∞ .
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 $f : A \rightarrow A$ 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	$e^{-1} n$
# cyclic nodes	$\sqrt{\pi n/2}$
transient	$\sqrt{\pi n/8}$
period	$\sqrt{\pi n/8}$

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 $f : G \rightarrow G$ 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

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

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 $a = 3$.

Then $e = 1174$ and $x_e = 11400$. We obtain the equation

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

Use the Extended Euclidean algorithm to get

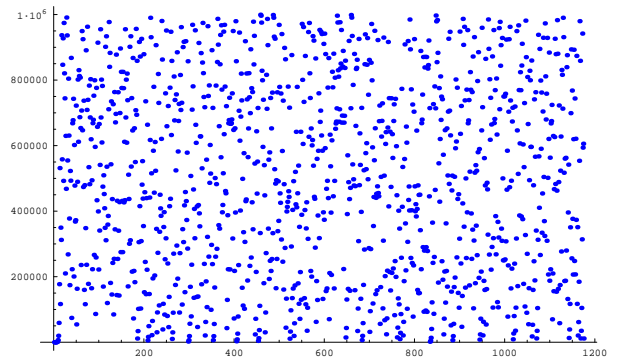
$$\gcd(310686, p-1) = 2 = 148845 \cdot 310686 - 46246 \cdot 999958$$

whence

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

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

How Random Is It?



- Implementing Finite Fields
- Discrete Logarithms
- Advanced Encryption Standard

The History

The now classical DES (data encryption standard) is based on keys of length 56 bits. That was fine in the mid 1970's, but modern technology wreaks havoc on DES (keys 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>

Rijndael



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 $f: \mathbf{2}^{32+48} \rightarrow \mathbf{2}^{32}$ is a carefully constructed boolean map.

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

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

$$x^8 + x^4 + x^3 + x^2 + 1$$

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

The Appointed Rounds

The algorithm 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 for this round.

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).

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

Later on the current state is described by such a matrix.

Byte Substitution

Define the patched inverse of an element $\mathbf{a} \in \mathbb{F}_{2^8}$ to be

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

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

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

Then byte substitution is given by the non-linear map $\mathbf{a} \mapsto A\bar{\mathbf{a}} + v$ applied to each byte separately.

Shift Rows

Replace the state matrix by the row-shifted version

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

Easy, right?

The next operation is a bit more complicated.

Mixing Columns

Let

$$g(z) = 03z^3 + 01z^2 + 01z + 02 \in \mathbb{F}_{2^8}[z]$$

and multiply each column of the state matrix (construed as a polynomial in $\mathbb{F}_{2^8}[z]$) by g , then reduce modulo $z^4 + 1$.

This comes down to a matrix multiplication over \mathbb{F}_{2^8} :

$$\mathbf{a} \rightsquigarrow \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \mathbf{a}$$

Notation

Here we have used the same notation as in the Rijndael specification to denote elements of \mathbb{F}_{2^8} .

Think of kl as two hexadecimal digits, together they produce one byte which can be construed as the coefficient vector of an element of \mathbb{F}_{2^8} .

So, for example, $D4$ corresponds to

$$z^7 + z^6 + z^4 + z^2$$

in classical notation, or 11010100 as a bit-vector.

This is an example where lookup tables (for the small field \mathbb{F}_{2^8}) are useful.

Keys and Decryption

At each round, a round key is xor-ed with the state matrix; the round key has the same size as the state (so this operation is self-inverse).

The real issue here is how the round key is derived from the original cipher key. Incidentally, proper key management is a huge problem in cryptography.

Take a look at the Rijndael specification at

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

The document also contains 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.

Summary

- The implementation of finite fields leads to lots of interesting algorithmic problems, in particular when efficiency is crucial.
- Discrete logarithms, in particular in finite fields, are the foundation of public key cryptography.
- Finite field operations are central to the Advanced Encryption Standard.
- Finite fields in the guise of linear feedback shift registers are important for efficient (low-quality) pseudo-random number generation.