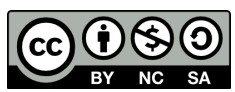
CDM

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.

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.

- 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

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.

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 bit of pre-computation and is only of interest when lots of field multiplications are needed. Suppose we have characteristic $p > 2$.

The key is to use a strange representation of $\mathbb{F}^\times$ that seems to just make a mess.

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

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

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

No Morphism

Careful, though, $\widehat{R}$ is not a field isomorphism. Actually, for addition, there is no problem:

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

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.

How can we do the reduction cheaply? First, precompute $\alpha = -p^{-1} \bmod R$. Suppose we have an integer x where $0 \leq x < Rp$.

- Compute $x' = x\alpha \bmod R$.
- Then $(x + x'p)/R$ is an integer and

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

Recall that $R = 2^k$, so these operations only require integer multiplication and some shifting.

This is more compelling by looking at the actual code.

Here is a typical implementation for $R = 2^{16}$ and some prime p, $2 < p < R$.

```
#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.

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
$\widehat{R}(a)$	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

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

$$(x + x_0 p)/R = 832/64 = 13$$

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.

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

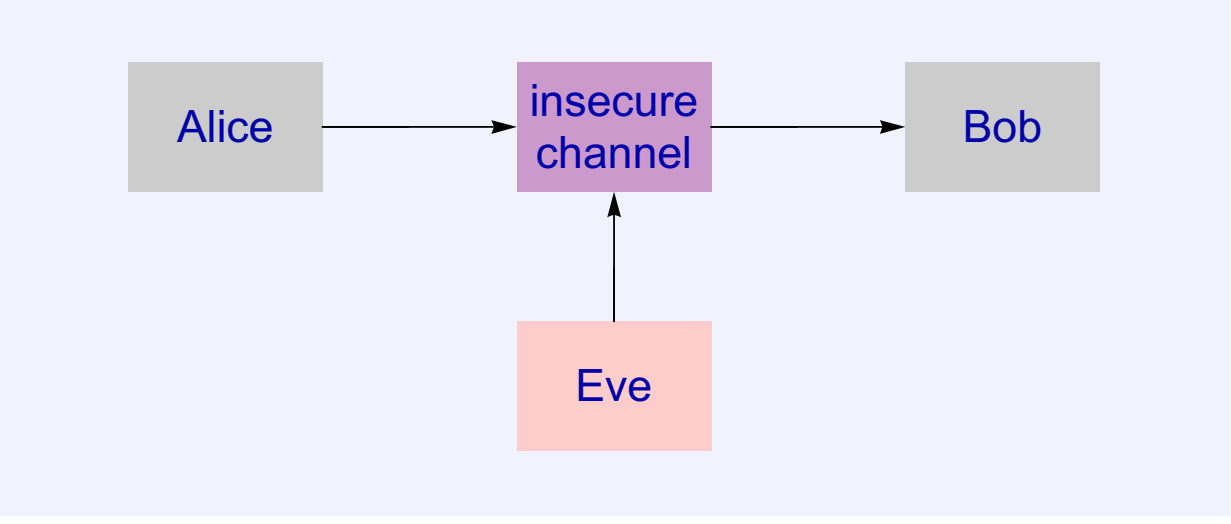
One uses a log table for the intermediate field $\mathbb{F}_{2^\ell}$, so arithmetic there is very fast (comparable to the prime field $\mathbb{F}_2$).

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.

Thanks to coding theory we can assume that messages are sent over a noiseless channel (a probabilistic assertion).



- Alice sends an encrypted message
- Bob receives and decrypts the message
- Eve, the eavesdropper, intercepts the message

Goal: even though Eve has full access to the encrypted message, she cannot decrypt it with available computational resources.

$$\text{encode} : \text{message space} \longrightarrow \text{cipher space}$$

$$\text{decode} : \text{cipher space} \longrightarrow \text{message space}$$

Requirement:

$$\text{decode}(\text{encode}(x)) = x$$

Terminology: The encoding function translates a

message or plaintext

into a

cipher text or coded message or cryptogram.

Usually coding and decoding involves an additional special parameter, called a key, that specifies the particular details of the encoding used.

$$\text{encode} : \text{message space} \times \text{key space} \longrightarrow \text{cipher space}$$
$$\text{decode} : \text{cipher space} \times \text{key space} \longrightarrow \text{message space}$$

where

$$\text{decode}(\text{encode}(x, K), K) = x$$

- $\text{encode}(x, K) = z$ must be easy to compute.
- $\text{decode}(z, K) = x$ must be easy to compute.
- $\text{decode}(z, ???) = x$ must be hard to compute.

Key spaces are usually finite and may well be known to the adversary, but so large that exhaustive search is impossible.

There is an old method known as Vernon’s Xor code that is perfect in a certain sense.

Suppose the message space is $\mathbf{2}^n$ and let K be a random binary sequence of length n. Code by bit-wise xor between message and key:

$$\text{encode}(x, K) = x \oplus K$$

The decoding function is exactly the same as the encoding function.

Importantly, with probability 1, the cipher text is just a random bit sequence no matter what the original message is. But then there is no way it can be decrypted without the key K.

Big Problem:
The key K must be kept secret at all cost.

In other words, it needs to be transmitted by a secure channel. We're back to square 0.

There are other glitches. Notably, if Eve can make Alice send (parts of) a specific message x we get

$$\text{encode}(x, K) \oplus x = K$$

so we must change the key with some frequency.

An extreme case of Vernon is to use one-time pads: use a new key every time a message is sent. The secure channel now is used as often as the insecure one, though presumably for far shorter messages.[†]

[†] Supposedly this method was used at the American Embassy in Moscow.

Quantum physics provides a potentially unassailable way of getting one-time pads: send a pair of entangled photons to both Alice and Bob, measure to obtain 1 bit per photon. For bizarre reasons, Alice and Bob will measure the same bit, despite the fact that

- the measuring devices are far apart,
- the photons will produce a random bit stream.

Any eavesdropping would destroy the stream of photons.

In practice works at distances of 100km, enough to wire a financial district. With satellites get up to 1200 km.

In 1976, W. Diffie and M. Hellman proposed cryptographic schemes that accomplish a feat that may seem logically impossible at first glance:

> Secure communication using only insecure channels.

The eavesdropper has complete knowledge about the encryption method used and has full access to the communication channel.

And yet, Eve lacks some critical piece of information and is utterly incapable of decrypting the cipher texts she has full access to.

The concept of secure communication over entirely insecure channels seems rather paradoxical at first glance. And yet ...

The central idea to distinguish between a public key and a private key. Say, Bob wants to message Alice.

- Alice's public key is available to anyone interested in communicating, there are no protections whatsoever.
- Alice's private key is available only to her, no one else has access.

Bob can use Alice's public key to encode a message and then send it over any insecure channel. Alice is able to decode the cipher text using her private key.

Presumably, Eve cannot decode since she has no access to the private key (at least within the constraints of feasible computation).

For a one-time pad, the critical information is the key, and it has to be kept in two places. To get it from one place to the other, a secure channel is critical.

On the upside, the cipher text is random.

In public key encryption, the private key is kept in one place only, there is no need whatsoever for secure communication.

But the cipher text is no longer random and in fact far from it. There is the danger that someone could find an attack that allows for decryption without knowledge of the private key.

We have to ensure that the computation

$$\text{decode}(z, ???) = x$$

is difficult to impossible if the proper key K is not known.

It is natural to try to use computational complexity theory to make this more precise. For example, "easy" will translate into polynomial time computable.

One the other hand, "difficult" should translate into XX-hard where XX is some suitable complexity class, safely removed from polynomial time.

Here is a formalization of the idea that a function is easy to compute in the forward direction, but it's hard to go backward. We use polynomial time as a stand-in for easy, and lack thereof for hard.

Definition

A function $f : \mathbf{2}^\star \to \mathbf{2}^\star$ is a one-way function if

- f is polynomial time computable
- $|f(x)| = \Theta(|x|)$
- For any randomized polynomial time algorithm $\mathcal{A}$, sufficiently large n, $x \in \mathbf{2}^n$, and $k > 0$

$$\Pr_{\boldsymbol{r}}[\mathcal{A}(f(x)) = x] \leq n^{-k}$$

Here the probability distribution is supposed to be uniform over $x \in \mathbf{2}^n$ and random bits in $\boldsymbol{r}$.

Comments

$\mathcal{A}$ represents an attack on our cryptographic scheme.

In light of the importance of randomized algorithms it makes sense that $\mathcal{A}$ can use random bits trying to compute $f^{-1}(z)$.

As per our definition, an adversary will succeed in inverting the function only with negligible probability.

Note that k is unbounded here, so if we try to use error reduction based on polynomially many independent runs of $\mathcal{A}$ we will not achieve constant error probability.

The "sufficiently large" clause is a nuisance, but cannot be eliminated: $\mathcal{A}$ could could "cheat" and simply hardwire a finite number of cases. Constraints on $\mathcal{A}$ that prevent that kind of unintended behavior are difficult to formalize; in the end one winds up with program size complexity and things get very messy.

At present, there is no theorem guaranteeing the existence of a one-way function. Much less do we know how to construct a specific one.

Note that a non-constructive existence theorem would not be mildly annoying, to actually use the function we need to get our algorithmic hands on a concrete example.

The good news is that there are several plausible candidates. For example, plain multiplication seems to be hard to invert: for two primes p and q, and $n = pq$, we do not currently know how to factor n in polynomial time. And a lot of people are counting on things staying that way—though the quantum guys may wreck this dream.

For cryptography, the closely related concept of a trapdoor function is even more important: we want one-way functions that become easily invertible if a secret key (the trapdoor) is available.

A formal definition becomes a bit more involved, we now have to deal with a family of functions $f_k : \mathbf{2}^\star \to \mathbf{2}^\star$ where $k \in K \subseteq \mathbf{2}^\star$, K being the set of keys.

There has to be a probabilistic polynomial time algorithm that, on input 0^n, generates a key/trapdoor pair (k, t_k) where $|k| = n$.

Given k and x, we can compute $f_k(x)$ in polynomial time.

Given $z = f_k(x)$, k and t_k we can find a preimage in probabilistic polynomial time.

However, without the trapdoor t_k, the likelihood of success is negligible.

For RSA we choose some number n, the intended number of bits in the RSA primes.

First, we construct two suitable primes p and q of length around n bits.

The key consists of $N = pq$ and some random $e \in \mathbb{Z}_N^*$.

The corresponding trapdoor is $d = e^{-1} \pmod{\Phi(N)}$.

Encoding happens via $x \mapsto x^e \pmod{N}$.

Presumably $\Phi(N) = (p-1)(q-1)$ and thus d are hard to compute.

Merkle and Hellman proposed a cryptographic scheme based on on the SubsetSum problem.

Problem:	**SubsetSum**
Instance:	A vector $\boldsymbol{a} \in \mathbb{N}^n$ and $s \in \mathbb{N}$.
Question:	Is there a vector $\boldsymbol{x} \in \mathbf{2}^n$ such that $\boldsymbol{a} \circ \boldsymbol{x} = s$?

This problem is well-known to be $\mathbb{NP}$-complete.

Careful, though: some instances of SubsetSum are easily solved using a brute-force, linear-time greedy algorithm.

This is a general problem in the application of computational hardness to cryptography.

Call $\boldsymbol{a}$ superincreasing iff $a_i > \sum_{j<i} a_j$. E.g., $a_i = 2^i$ is superincreasing.

Alice constructs some superincreasing sequence $\boldsymbol{b}$.

She also picks some modulus m, $m > \sum b_i$, and a number c coprime to m.

Alice computes $a_i = c\, b_i \bmod m$.

Alice's public key is $\boldsymbol{a}$, her trapdoor pair is m and c.

When Bob wants to send a message $\boldsymbol{x} \in \mathbf{2}^n$ he encodes it as $s = \boldsymbol{a} \circ \boldsymbol{x}$.

Alice can decode using her private key: she computes $s' = c^{-1} \bmod m$ and then solves the corresponding superincreasing $\boldsymbol{b}$ instance greedily.

The vector $\boldsymbol{a}$ looks rather random, and there is no reason why the $\boldsymbol{a}$ instance of SubsetSum should be easy for Eve to solve.

Since Eve knows neither m nor c, she cannot translate into the easy instance over $\boldsymbol{b}$.

Burning Question:

Is this really hard for someone who knows neither m nor c?

One needs to make sure that there is no possible angle of attack, using whatever ideas might be applicable.

Alas . . .

In 1982, Shamir published a polynomial time attack on the Merkle-Hellman method that is a perfect example of out-of-the-box thinking:

Instead of trying to somehow get access to m and c, his method uses Lenstra's integer programming algorithm to find some pair m' and c' that behaves like m, c but will usually be different.

That's fine, m' and c' suffice to perform the translation into a superincreasing instance which can then be solved greedily.

One can also use the famous Lenstra–Lenstra–Lovász (LLL) lattice basis reduction algorithm to tackle Merkle-Hellman.

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

$$e \rightsquigarrow g^e$$

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:

Discrete Logarithm Problem:

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

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

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

We need a nice cyclic group G of sufficient size where the group operation is easily computable and we have access to a generator.

Diffie/Hellman used the multiplicative group of a finite field $\mathbb{F}_{p^k}$. So all we need is a primitive polynomial $\tau \in \mathbb{F}_p[x]$ of sufficiently high degree.

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

$$c = g^{xy} = a^y = b^x$$

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

Eavesdropper Eve knows the algorithm, $\mathbb{F}$, g, a and b, but **not** the trapdoor exponents x and y.

On the face of it, Diffie/Hellman seems to work in the sense that no one has produced any systematic attack on the method.

Alas, for cryptography this is not really enough: even a partial attack that only works in some special cases may not be tolerable. This is very different from ordinary computational hardness where one only needs to establish that some difficult instances exist.

“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.

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

Suppose g is a generator of the multiplicative subgroup of $\mathbb{F} = \mathbb{F}_{p^k}$ and $a \neq 0$ is some given element in the field whose logarithmic we want to compute.

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 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.

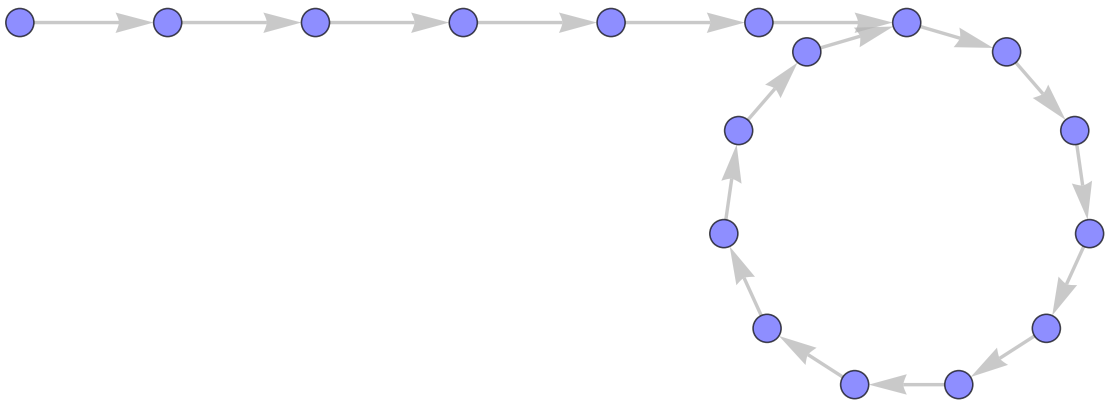
To check for a common element we can use hashing. Of course, one need not wait till the lists are complete to check for a match.

The baby-step/giant-step method requires $O(\sqrt{q})$ field operations, and storage of $O(\sqrt{q})$ field elements.

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.

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.

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 $c_2 \approx 0.78$.

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).

Wild Idea:

Can we compute a (pseudo-)random sequence (x_i) of group elements so that $x_i = x_{2i}$ helps us to compute a discrete logarithm?

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 given the partition of G.

Consider the orbit (x_i) of 1 under f.

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.

Use Floyd to find the minimal index e such that $x_e = x_{2e}$:

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

But then

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

This equality does not solve the discrete logarithm problem directly but it can help at least sometimes.

Again, for cryptographic applications 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 $p = 999959$. Pick generator $g = 7$ and let $a = 3$.

Perhaps the most simpleminded partition is to chop $[p-1]$ into thirds. This produces an orbit with transient and period

$$928 \qquad 587$$

A similarly obvious partition would use $x \bmod 3$. This produces an orbit with transient and period

$$919 \qquad 575$$

Note the values are order-of-magnitude close to $\sqrt{p}$, looks like our maps are sufficiently random.

Running a suitably modified version of Floyd's algorithm with the first partition produces $e = 1174$ and $x_e = 11400$, plus the identity

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

Close, but no cigar: we need to somehow clobber the exponent 310686.

The last identity lives in $\mathbb{Z}_p^\star$, a group of order $p-1$.

So we could try to simplify exponents modulo $p-1$.

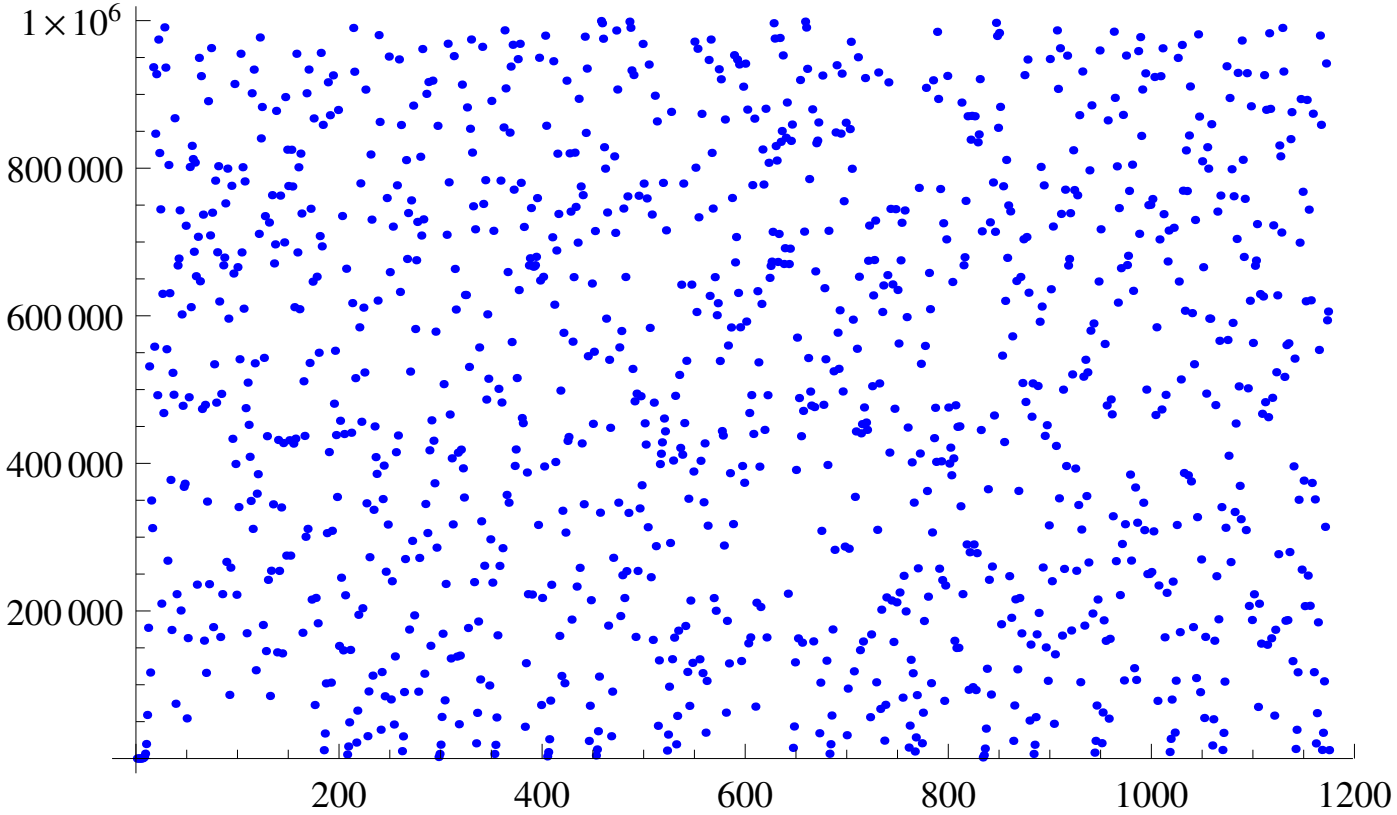
Use the Extended Euclidean algorithm to get

$$\begin{aligned}\gcd(310686, p-1) &= 2 \\ &= 148845 \cdot 310686 - 46246 \cdot 999958\end{aligned}$$

Then raise 3^{310686} to the 148845 power mod p to obtain

$$\begin{aligned}3^2 &= 7^{356324} \pmod{p} \\ 3 &= \pm 7^{178162} \pmod{p}\end{aligned}$$

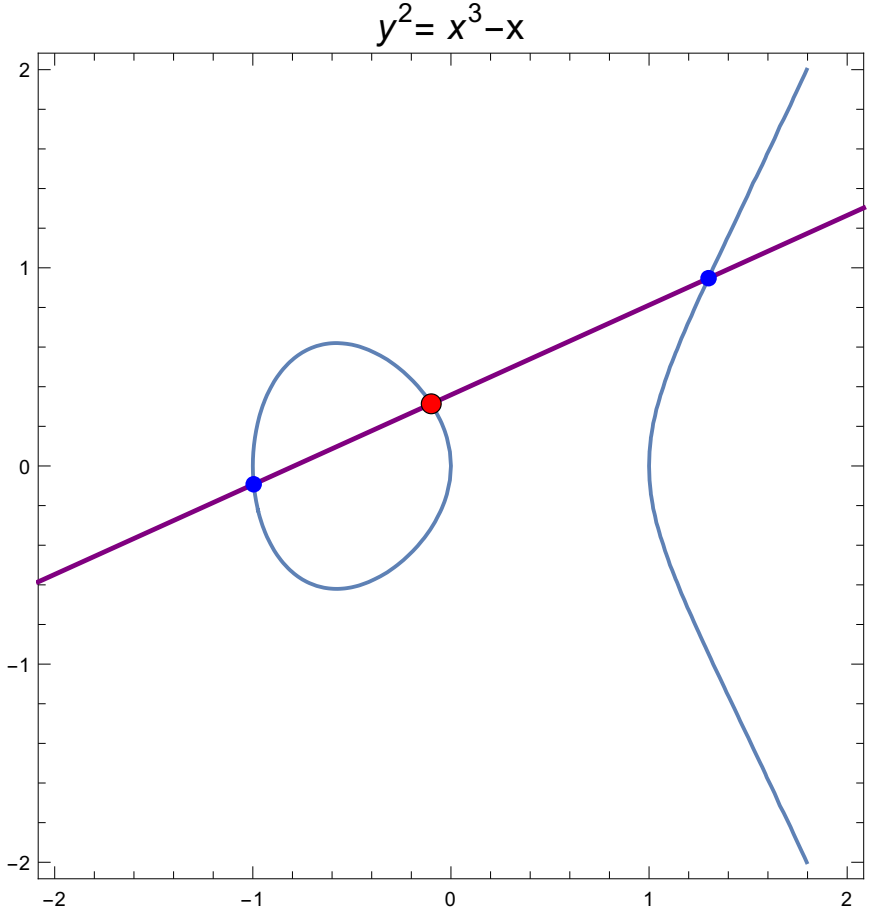
We can simply check the two cases and find that in $\mathbb{Z}_p$: $\log_7 3 = 178162$.



A plot of the orbit of 1 given our "random" partition.

Aside: Elliptic Curves

Curves of the form $y^2 = x^3 + ax + b$ over a finite field produce a nice group that can be used for discrete logarithm methods.



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, 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.

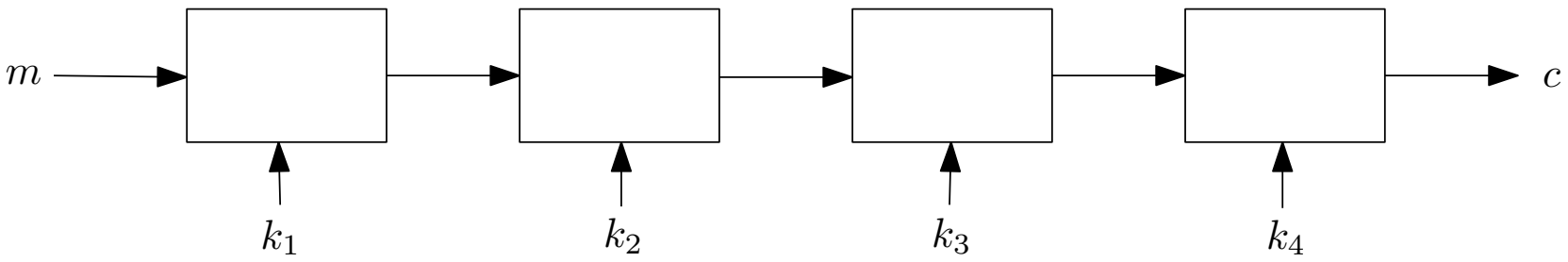
DES and AES are both based on ideas that first appeared in Lucifer, an encryption system developed mainly by IBM: the input message is chopped into fixed-size block of bits, which are then clobbered using a key.

- 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 anything resembling a safe encoding. But a sufficiently long chain is hard to crack without knowledge of the master key.

As an aside, a long message obviously needs to be chopped up into blocks of the right side. This apparently trivial task is handled by a "block cipher mode of operation."

Electronic codebook mode: divide the message is divided into blocks, encrypt each block separately. Not a great idea.

Cipher block chaining: Xor each block of plaintext with the previous ciphertext block, then encrypt.

Propagating cipher block chaining: Xor not just with previous ciphertext but also with plaintext.

And so on, and so forth.

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 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}).

$$2^{56} \qquad 7.2 \times 10^{16}$$

$$2^{128} \qquad 3.4 \times 10^{38}$$

$$2^{256} \qquad 1.2 \times 10^{77}$$

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

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 $\mathbb{F}_{2^8}[z]/(z^4 + 1)$. I.e., we smash all higher powers down below 4.

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

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 $\mathbb{F}_{2^8}$.

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

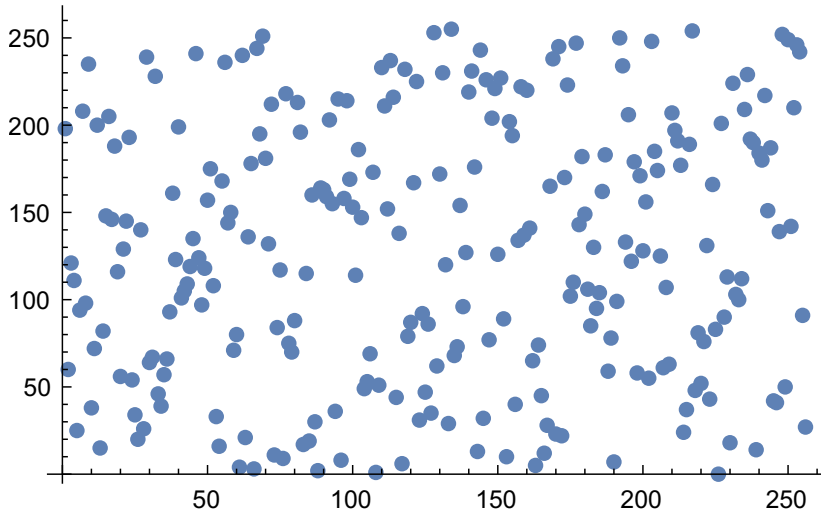
$$\overline{\boldsymbol{a}} = \begin{cases} 0 & \text{if } \boldsymbol{a} = 0, \\ \boldsymbol{a}^{-1} & \text{otherwise.} \end{cases}$$

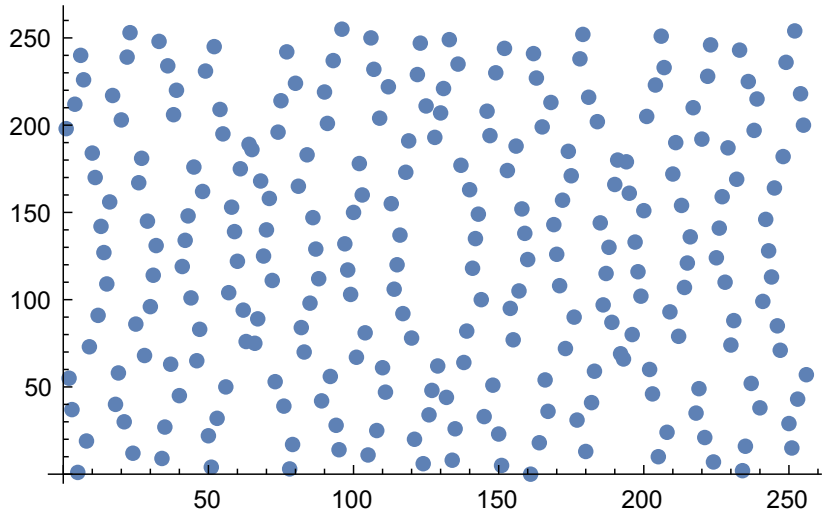
Define an 8×8 bit-matrix and 8-bit vector as follows

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

Then byte substitution is given by the not-quite 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.

The visuals suggest that patched inverses help quite a bit to make a bigger mess.

(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). So e.g. the diagonal winds up in the first column.

The next operation is a bit more complicated.

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 $1101\,0100$ as a coefficient vector, or the field element

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

as a polynomial.

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

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

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

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

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

AES is a symmetrical block cipher, meaning the same key is used for encryption and decryption. E.g., we can use xor of the key and (parts of) the cipher text.

Let's ignore the details.

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 a Rijndael encrypted message.

One pleasant aspect: the decryption operations are very similar to encryption, so essentially the same hardware can be used. With appropriate support the throughput is quite high (hundreds of MB/s).

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.