Course Notes for Introduction to Cryptography

Eric Bach
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Foreword

These are lecture notes from an undergraduate course in cryptography, which I have taught since 1999. The course had two goals. First, to use classical cryptography to introduce the essential mathematics (elementary number theory, probability and statistics) needed to understand more complicated modern systems. Second, to provide the student with a survey of the major ideas in modern cryptography, illustrated as far as possible with current cryptographic algorithms.

I thank my former students in this course, especially Jeff Chard, for pointing out many errors in the original notes.

Table of Contents

Lecture 1

What's this about?

If we were being precise, our course would be called Introduction to Cryptology. "Cryptography" is more common among non-experts, of course.

Cryptology = Cryptography + Cryptanalysis

Cryptology involves techniques for ensuring the integrity and authenticity of digital information in an adversarial environment

Cryptography is the design and use of secret codes

Cryptanalysis involves techniques for unauthorized reading of encoded data

It is essential that you learn both halves of the subject. The only way to assess the security of a code or system is to know what your opponent is capable of.

The subject has a long history.

The standard reference is David Kahn, The Codebreakers.

Until about 1970, cryptographic experts were either professionals (working for governments, and in rare cases, private companies) or amateurs (who enjoyed the puzzle-solving aspects of cryptanalysis). After 1970, cryptography became popular among academics.

Because cryptography combines mathematical knowledge and practical skill, writing in the field can be very opinionated. When reading sources, you should figure out which group the author belongs to (professional, amateur, academic) and what biases, if any, result from this affiliation.

We will begin our study by reviewing some classical systems. None of these will survive cryptanalysis (especially in the computer age) but it is a good way to get started in the field without the extra overhead of learning complicated algorithms.

The Caesar cipher

The earliest references to cryptography are classical. The Roman historian Suetonius stated that Julius Caesar, wishing to keep his writings from casual observers, would replace each letter by the third one following.

This means we replace `A` by `D`, `B` by `E`, ..., and `Z` by `C`. This mapping is called the *encryption transformation*.

Example: suppose the message is

```
NON ILLEGITIMI CARBORUNDUM .
```

This transforms into

```
QRQ LLOHJLWLPL FDUERUXQGXP .
```

(Actually we are cheating – Latin didn't use spaces.)

The inverse mapping, called *decryption*, replaces `A` by `X`, `B` by `Y`, and so on.

Notice that encryption and decryption are a matched pair of functions.

There is nothing special about the number 3. We could, if we wanted, shift by any amount from 0 to 25. This is called the *shift cipher*, and the amount of shift is called the *key*.

The shift cipher persists into modern times. The Russian army used it in WWI, probably with disastrous effect [Bauer, Decrypted Secrets, p. 47]. Shifting a message by 13 (called `rot13`) became popular on the Internet, also as a way of preventing casual reading. Note that this transformation is its own inverse.

For a more recent example, see Discovery News Brief, April 21,2006: Mafia Boss's Encrypted Messages Deciphered, by Rossella Lorenzi.

Classical cryptosystems

There are a great variety of cryptographic systems. We need some mathematical framework for talking about them. Here is one way to do it.

P is a set of *plaintexts*

C is a set of *ciphertexts*

K is a set of *keys*

For each $k \in K$, there are two functions

$$e_k : P \to C \qquad \textit{encryption}$$

and

$$d_k : C \to P \qquad \textit{decryption}$$

such that

$$d_k(e_k(x)) = x \qquad \text{for all } x \in P$$

Often $P = C$, and it is called the *message space*.

For our example above (shift ciphers) we have

$P = \{\mathtt{A}, \ldots, \mathtt{Z}\}$

$C = \{\mathtt{A}, \ldots, \mathtt{Z}\}$

$K = \{0, \ldots, 25\}$

$e_k(x)$ is the k-th letter following x

$d_k(x)$ is the k-th letter preceding x

Note that, if we take the indices mod 26, $e_k = d_{-k}$.

Classically, cryptographic transformations were applied letter-by-letter. That is, if the message was

$$x = x_1 x_2 \ldots x_n$$

with $x_i \in P$, then it was encrypted as

$$e_k(x) = e_k(x_1)e_k(x_2)\ldots e_k(x_n)$$

Decryption proceeded similarly.

Other systems

Block ciphers

Instead of applying the same transformation to each letter, we can apply them to blocks of letters.

Example: Let us shift the 1st, 3rd, 5th, ... letters by 1 and shift the 2nd, 4th, 6th, ... letters by 2. So

```
VENI VIDI VICI
```

becomes

```
WGOJ WJEJ WJDJ
```

This can be subsumed in the previous setup if we take P to be all "words" of b symbols. The number b is called the *block length*.

Many of the commonly used ciphers, like DES and AES, are block ciphers, in which the underlying "symbol" is a bit (0 or 1).

Stream ciphers

These are systems in which the transformation of a symbol depends on its position (and possibly on previous symbols).

Example: Let us shift the i-th letter by p_i, the i-th prime. Since primes can be larger than 26, we agree that the shift amount is p_i mod 26. To encrypt

```
VENI VIDI VICI
```

the shifts would be

$$2, 3, 5, 7,\ 11, 13, 17, 19,\ 23, 3, 5, 11$$

– this is a kind of *pseudo-random sequence* – which gives

```
XHSP GVUJ ILHT
```

You should observe that the encrypted message is more "random looking" than in the block cipher example. A good stream cipher will have this property.

Before the advent of public-key cryptography, secure long-haul telecommunication relied on stream ciphers, and many of the systems were secret and/or proprietary. (A cynic might go further and say that only the bad stream ciphers got published.) For this reason, we make no claim to completeness in this area.

Public-key systems

For a classical system, anyone who knows the encryption key, or the encryption transformation, can easily determine the decryption transformation.

The realization that this need not be true led to the invention of public-key cryptography. Public key systems are those in which it it computationally infeasible to determine d_k from k or e_k.

Public-key cryptography has many applications beyond message transmission over insecure channels. An example is the idea of a digital signature: something that can be associated with a message that could only have been produced by a particular person holding that message.

Lecture 2

This lecture covers *modular arithmetic*.

Letters as Numbers

Recall the shift cipher:

Plaintexts $P = \{\mathtt{A}, \ldots, \mathtt{Z}\}$

Ciphertexts $C = \{\mathtt{A}, \ldots, \mathtt{Z}\}$

Keys $K = \{0, \ldots, 25\}$

Encryption function $e_k(x) = k$-th letter following x

Decryption function $d_k(x) = k$-th letter preceding x

More complicated transformations will be harder to describe in this informal way. It is easier to encode letters by numbers and work directly with the numbers. One system is

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Congruences

Suppose N is a positive integer. We write

$$a \equiv b \pmod{N}$$

whenever $b - a$ is a multiple of N. Thus

$$32 \equiv 2 \pmod{3}$$

whereas

$$10 \not\equiv 5 \pmod{3}$$

Observe that congruence mod N is a *relation*, that is, two numbers are either congruent mod N or they aren't. This relation has properties similar to equality, for example if $a \equiv b$ and $b \equiv c$, you can conclude that $a \equiv c$.

The integers mod N

Let $\mathbf{Z}_N = \{0, 1, \ldots, N-1\}$.

This is the complete set of remainders that can occur after division by N.

If x is an integer then x mod N is the unique y in $\mathbf{Z}_N$ for which $x - y$ is divisible by N.

This is called the *unary mod* operation.

Example: $346 \bmod 5 = 1$.

Our definition is standard in mathematics. Certain programming languages do things a little differently. For example, in C (on my workstation!) we have

$$(-1)\%3 = -1$$

whereas for us

$$(-1) \bmod 3 = 2$$

For certain purposes, one can replace $\mathbf{Z}_N$ by other complete sets of remainders. Popular choices include $\{1, \ldots, N\}$ (used by Sinkov) and the symmetric remainder set $\{-\lfloor N/2 \rfloor, \ldots, 0, 1, \ldots, \lfloor (N-1)/2 \rfloor\}$.

We can now describe the shift cipher more concisely as

$P = C = K = \mathbf{Z}_{26}$.

$e_k(x) = x + k \bmod 26$

$d_k(x) = x - k \bmod 26$

We could replace 26 by any other N if wanted.

Algebraic laws for $\mathbf{Z}_N$

$\mathbf{Z}_N$ obeys the usual laws of algebra if we agree to do the operations mod N. More precisely, let us define

$x \oplus y = x + y \bmod N$

$x \otimes y = xy \bmod N$

Then the following laws hold:

$$\begin{aligned} x \oplus y &= y \oplus x & x \otimes y &= y \otimes x \\ x \oplus (y \oplus z) &= (x \oplus y) \oplus z & x \otimes (y \otimes z) &= (x \otimes y) \otimes z \\ x \oplus 0 &= x & x \otimes 1 &= x \end{aligned}$$

$$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$$

If we use $x \ominus y$ as an abbreviation for $x \oplus -y$, then

$$x \ominus x = 0$$

Some examples: Take $N = 26$. Then

$24 \oplus (14 \oplus 15) = 24 \oplus 3 = 1$

$(24 \oplus 14) \oplus 15 = 12 \oplus 15 = 1$

and

$$10 \otimes (5 \oplus 4) = 90 \bmod 26 = 12$$
$$(10 \otimes 5) \oplus (10 \otimes 4) = 24 \oplus 14 = 12$$

You should prove these laws as exercises for yourself. To see how this goes, let's consider the associative law for addition mod N:

$$\begin{aligned} x \oplus (y \oplus z) &= x \oplus (y + z \bmod N) \\ &= x \oplus (y + z + kN) \\ &= (x + y + z + kN) \bmod N \\ &= (x + y + z) \bmod N. \end{aligned}$$

Reasoning similarly, we also have

$$(x \oplus y) \oplus z = (x + y + z) \bmod N,$$

so

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z.$$

The system $(\mathbf{Z}_N, \oplus, \otimes)$ is called the *ring of integers mod* N, or, less pedantically, *arithmetic mod* N.

There is usually no ambiguity in replacing $\oplus, \ominus, \otimes$ by $+, -, \times$ so we'll omit the circles from now on.

What happens with division?

Anything you can derive in ordinary algebra using only the properties of addition, subtraction, and multiplication listed above will still be true in $\mathbf{Z}_N$. This is not so, however, when you want to use division.

In ordinary algebra, if $x \neq 0$ then there is a y making $xy = 1$. This need not hold mod N.

Example: take $N = 26$ and $x = 2$. Could we have $2y \bmod 26 = 1$? Suppose this were true. Then

$$2y + 26k = 1$$

for some integer k, but the left side of this equation is an even number and the right side is odd.

A system in which this law does hold is called a *field*. It can be shown that $\mathbf{Z}_N$ is a field iff N is prime.

When p is a prime, you will sometimes see the notation $GF(p)$ used for our $\mathbf{Z}_p$. ("GF" stands for Galois field – after the French mathematician who did a lot of early work on these systems.)

In ordinary algebra, if $a \neq 0$ then $ax = b$ has a unique solution. In arithmetic mod N, there can be multiple solutions.

Example: consider $a = b = 2$ and $N = 26$. The equation

$$2x = 2$$

has two solutions in $\mathbf{Z}_{26}$, namely $x = 1$ and $x = 14$.

Affine ciphers

The results we have proved have implications for the design of cryptographic systems. Let's consider the *affine cipher*, for which

$P = C = \mathbf{Z}_{26}$

$e_k(x) = (ax + b) \bmod 26$.

We'd like to take the key space to be all pairs (a, b) with $a, b \in \mathbf{Z}_{26}$. Examples show, however, that not all values of a can be used.

Suppose $a = 13$. Let's use the algebraic laws we noted above. If x is even, then in arithmetic mod 26,

$$13x = 13(2y) = 26y = 0y = 0$$

whereas if x is odd, we have

$$13x = 13(2y + 1) = 26y + 13 = 13.$$

Thus, if the numbers $\{0, \ldots, 25\}$ stand for the letters $\mathtt{A}, \ldots, \mathtt{Z}$ in order, the letters $\mathtt{A}, \mathtt{C}, \mathtt{E}, \ldots, \mathtt{Y}$ all get encrypted in the same way. The same is true of the letters $\mathtt{B}, \mathtt{D}, \mathtt{F}, \ldots, \mathtt{Z}$. So the receiver will not be able to recover very much information about the message!

Which values of a will work? Let's think of the mapping $x \rightarrow ax + b$ as multiplication by a followed by shift by b. Only the first operation can destroy information, because shifting is always reversible. So we can only use values of a for which $x \rightarrow ax$ is 1-1 as a mapping on $\mathbf{Z}_{26}$. This is true for

$$a = 1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25$$

and no others. All of these numbers have the property that they have no prime factors in common with 26.

Sets of this type are so important that we give them a special name.

$$\mathbf{Z}_N^* = \{x \in \mathbf{Z}_N : x \text{ and } N \text{ have no common prime factors}\}$$

When is multiplication by a mod N a 1-1 function?

Theorem: The mapping $x \rightarrow ax$ is 1-1 on $\mathbf{Z}_N$ iff $a \in \mathbf{Z}_N^*$.

Proof.

(Forward) If the mapping is 1-1, it is onto, so there will be some x for which

$$ax = 1 \qquad \text{in } \mathbf{Z}_N$$

So there is an integer k for which

$$ax - kN = 1.$$

If some prime divided a and N, it would also divide 1, which is impossible. So a and N have no prime factors in common, that is, $a \in \mathbf{Z}_N^*$.

(Reverse) We will use the unique factorization law, which says that every positive number is a product of primes in an essentially unique way. Suppose the mapping is not 1-1. Then there are x and x' with $0 \leq x' < x < N$ making

$$ax = ax' \qquad \text{in } \mathbf{Z}_N$$

So

$$a(x - x') = 0 \qquad \text{in } \mathbf{Z}_N$$

so for some integer k

$$a(x - x') = kN \qquad \text{as integers}$$

If N has the prime factorization $p_1 \cdots p_m$, then

$$a(x - x') = kp_1 \cdots p_m$$

The number $x - x'$ is between 1 and $N - 1$, so one of the p_i's must divide a. Therefore, a is not in $\mathbf{Z}_N^*$.

Lecture 3

This lecture covers *Euclid's algorithm and modular inversion.*

Arithmetic mod N

$\mathbf{Z}_N = \{0, 1, \ldots, N-1\}$, with $+, -, *$ taken mod N

$\mathbf{Z}_N^* = \{x \in \mathbf{Z}_N : x \text{ is relatively prime to } N\}$.

Two integers are *relatively prime* if they have no prime factors in common.

We have proved that $x \to ax$ is 1-1 iff $a \in \mathbf{Z}_N^*$.

Notation:

$$a^{-1} \bmod N = \text{ the unique } x \in \mathbf{Z}_N \text{ with } ax = 1$$

Note that this is only defined for $a \in \mathbf{Z}_N^*$.

Decryption for the Affine Cipher

The Affine Cipher has

Message space $\mathbf{Z}_N$

Encryption function $e_k(x) = ax + b \bmod N$

Decryption is only possible if $a \in \mathbf{Z}_N^*$, so the key space is $\mathbf{Z}_N^* \times \mathbf{Z}_N$.

Example:

Let's take $N = 26$, with $a = 7$ and $b = 2$.

The easiest way to make a table of the function $ax + b$ is to start with b and repeatedly add a mod N. We get

0	1	2	3	4	5	6	7	8	9	10	11	12
2	9	16	23	4	11	18	25	6	13	20	1	8

13	14	15	16	17	18	19	20	21	22	23	24	25
15	22	3	10	17	24	5	12	19	0	7	14	21

As usual, we'll think of the numbers in $\mathbf{Z}_N$ as standing for letters, so that `A` $= 0$, `B` $= 1$, and so on. To encrypt the message `CRYPTO`, we convert each letter into a number, apply the encryption function, and then change numbers back to letters. This gives

C	R	Y	P	T	O
2	17	24	15	19	14
16	17	14	3	5	22
Q	R	O	D	F	W

Observe that `R` is encrypted as itself.

The Decryption Function

We guess that decryption has the same general form as encryption, so we look for a function

$$d_k(y) = a'y + b'$$

making $d_k(e_k(x)) = x$. Plugging in, this equation becomes

$$x = a'(ax + b) + b' = a'ax + (a'b + b').$$

This will be satisfied if

$$a'a = 1$$

and $a'b + b' = 0$, which is equivalent to

$$b' = -a'b.$$

Thus if

$$e_k(x) = ax + b$$

we have

$$d_k(y) = a^{-1}y - a^{-1}b$$

We can see that this is the right formula if we observe that $ax + b$ involves multiplication by a, followed by a shift of b. To undo this, we should shift by $-b$, then undo the multiplication, that is, multiply by a^{-1}. So

$$d_k(y) = a^{-1}(y - b),$$

which is equivalent to the first formula.

Example (continued)

If $a = 7$ and $b = 2$, then $a^{-1} = 15$ in $\mathbf{Z}_{26}$. So our decryption function is

$$a^{-1}(y - b) = 15y - 15 \cdot 2 = 15y - 30 = 15y - 4 = 15y + 22.$$

We can check this by computing

$$d_k(\mathtt{Q}) = 15 \cdot 16 + 22 = 2 = \mathtt{C}.$$

Euclid's algorithm

If u and v are integers, we define $\gcd(u, v)$ to be the largest integer that divides both u and v, or zero if both are zero.

Examples:

$$\gcd(12, 15) = 3$$

$$\gcd(u, 0) = u$$

This leads to a recursive algorithm. Let the inputs be $u \geq v \geq 0$. We have

$$\gcd(u,v) = \begin{cases} u, & \text{if } v = 0; \\ \gcd(v, u \bmod v), & \text{if } v > 0. \end{cases}$$

This clearly works if $v = 0$. Otherwise, write $u = qv + r$ with $0 \leq r < v$. Then

$$d \text{ divides } u, v \Leftrightarrow d \text{ divides } v, r$$

so the gcd of u, v is the same as the gcd of v, r.

The algorithm terminates because v decreases.

Let's use this to compute the gcd of 26 and 7. We have

$$\begin{aligned} 26 &= 3 \cdot 7 + 5 && (u = 26, v = 7) \\ 7 &= 1 \cdot 5 + 2 && (u = 7, v = 5) \\ 5 &= 2 \cdot 2 + 1 && (u = 5, v = 2) \\ 2 &= 2 \cdot 1 + 0 && (u = 2, v = 1) \\ & && (u = 1, v = 0) \end{aligned}$$

so the gcd is 1.

Inverses mod N

Fact: There are integers x and y making $ux + vy = \gcd(u, v)$.

This explains why we agree to take $\gcd(0, 0) = 0$. If it were any other number this would not be true.

We can compute x and y recursively, using the results of Euclid's algorithm.

Base case: if $v = 0$ then take $x = 1$, $y = 0$.

Recursion: Since $v > 0$ we have $u = qv + r$ with $0 \leq r < v$. By induction, there are x', y' making $\gcd(u, v) = \gcd(v, r) = vx' + ry'$. So

$$\gcd(u,v) = vx' + ry' = vx' + (u - qv)y' = y'u + (x' - qy')v$$

Thus,

$$\begin{aligned} x &= y' \\ y &= x' - qy' \end{aligned}$$

Continuing with the example, let's solve $26x + 7y = 1$. A solution to $7x' + 5y' = 1$ is given by $x' = -2$ and $y' = 3$. On the other hand, $26 = 3 \cdot 7 + 4$, so $q = 3$. Using our recursive formula we get

$$\begin{aligned} x &= y' = 3 \\ y &= x' - qy' = -2 - 3 \cdot 3 = -11 = 15, \end{aligned}$$

in arithmetic mod 26.

Thus, a^{-1} mod N can be obtained by solving $Nx + ay = 1$ in integers, and setting $a^{-1} = y \bmod N$.

Remarks

These algorithms are very efficient. It can be shown that Euclid's algorithm uses $O(\log u)$ division steps. [See e.g. Section 4.2 of E. Bach and J. Shallit, Algorithmic Number Theory I, MIT Press.] The worst case is when u and v are consecutive Fibonacci numbers, say $u = F_n$ and $v = F_{n-1}$. In this case, all quotients will equal 1, and there will be about n division steps. Because

$$u = F_n \sim \text{ const } (1.618...)^n$$

we have $n \sim \log_{1.618...}(F_n) = O(\log u)$ as claimed.

Recursion uses extra space, but this can be eliminated by using a matrix form of the algorithm. We will illustrate this with our example. From the first step of Euclid's algorithm we have

$$\begin{pmatrix} 7 \\ 5 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} 26 \\ 7 \end{pmatrix}.$$

From the next step, we get

$$\begin{aligned} \begin{pmatrix} 5 \\ 2 \end{pmatrix} &= \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 7 \\ 5 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} 26 \\ 7 \end{pmatrix} \\ &= \begin{pmatrix} 1 & -3 \\ -1 & 4 \end{pmatrix} \begin{pmatrix} 26 \\ 7 \end{pmatrix} \end{aligned}$$

Similarly,

$$\begin{pmatrix} 2 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 & 4 \\ 3 & -11 \end{pmatrix} \begin{pmatrix} 26 \\ 7 \end{pmatrix}$$

and

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -3 & -11 \\ -7 & 26 \end{pmatrix} \begin{pmatrix} 26 \\ 7 \end{pmatrix}.$$

This is called the *Extended Euclidean Algorithm*. If this is done cleverly, one need only store 3 numbers. For good code, see Knuth, vol. 2.

Lecture 4

This lecture covers *Polyalphabetic ciphers.*

Alphabets

So far we have studied two systems (both with message space $\mathbf{Z}_N$):

Shift ciphers: $K = \mathbf{Z}_N$, $e_k(x) = x + a$

Affine ciphers: $K = \mathbf{Z}_N^* \times \mathbf{Z}_N$, $e_k(x) = ax + b$

In each case, we have a class of 1-1 onto mappings. By choosing a key, we select a transformation and its inverse.

In traditional cryptography, these mappings are called *alphabets.* This terminology persists in the literature (e.g. it is used by Sinkov), so here is a short translation guide.

Direct standard alphabet = shift cipher

Reversed standard alphabet = special affine transformations

$$x \rightarrow -x + a$$

We note that if $y = -x + a$, then $x = -y + a$, so such a transformation is its own inverse. The term arises because a table giving the mapping reverses the order of the letters. Consider for example $a = 2$ and $N = 26$, with the usual numerical values for letters. The mapping is

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
C B A Z Y X W V U T S R Q P O N M L K J I H G F E D
```

Decimated alphabet = affine transformations of the form

$$x \rightarrow ax$$

The operation of extracting every a-th element of a sequence is called *decimation.* For example, if $a = 3$, we take every third letter in order, so the mapping is

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A D G J M P S V Y B E H K N Q T W Z C F I L O R U X
```

Observe that in each case, the encryption and decryption mappings are of the same type. This is obviously useful in practice, since we then don't need two sets of hardware.

Monoalphabetic substitution

The most general family of mappings we can consider is the set of all *permutations* (1-1 onto mappings) on $\mathbf{Z}_N$.

For this system $K = \Sigma_N$, the set of all permutations on $\{0, \ldots, N-1\}$.

Traditionally, these are called random or mixed alphabets.

Pencil-and-paper cryptography involved lots of mnemonic tricks for determining the permutation from a key word. We'll be content with one demonstration. Choose a key word that does not repeat any letters, for example the word `NUMERICAL`. Use these as the images of the first letters of the alphabet, and then fill in the remaining letters in order, thus:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
N U M E R I C A L B D F G H J K O P Q S T V W X Y Z
```

There are obvious problems with this method, for example the last letters will probably map to themselves. So in practice other methods were used.

The goal of modern block ciphers (DES, AES, etc.) is to simulate, in some sense, randomly chosen permutations of the message space.

Polyalphabetic methods

Ciphers of the type we have discussed can be improved by changing the mapping in a periodic way.

The classic system of this type is called Vigenère. It was state of the art from the 1600's until about 1850.

The idea is to use a periodically repeating shift. Traditionally, this was done with the aid of a key word. For example, let the key, and its numerical equivalent, be

C	R	Y	P	T	O
2	17	24	15	19	14

We write the message followed by its numerical equivalent, and the key, repeated as many times as needed. Adding the two rows of numbers mod 26 and transforming back to letters produces the ciphertext:

L	I	A	I	S	O	N	S	D	A	N	G	E	R	E	U	S	E	S
11	8	0	8	18	14	13	18	3	0	13	6	4	17	4	20	18	4	18
2	17	24	15	19	14	2	17	24	15	19	14	2	17	24	15	19	14	2
13	25	24	23	11	2	15	25	1	15	5	20	6	8	2	9	11	18	20
N	Z	Y	X	L	C	P	Z	B	P	F	U	G	I	B	J	L	S	U

The receiver, also knowing the key, can undo the transformation and read the message.

Encryption and decryption were done with the aid of a table for addition mod 26 (written with letters), called the *Vigenére square*. One table suffices, since the decryptor can just use the complementary key word, which in our case is

Y	J	C	L	H	M
24	9	2	11	7	12

Beaufort is a related system based on reversed alphabets.

Of course, one can use other transformations besides shifts.

Modern-day Vigènere systems

One reason to learn about the classic methods is that they keep getting recycled into modern systems. The following example comes from an article by John Bennett [Cryptologia 11:4, 1987, pp. 206-210], which describes the encryption used by Word Perfect 4.2.

The system is based on the ASCII character set, which uses eight bits to store a character. A character can thus be written down using two elements of

$$\mathbf{Z}_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}.$$

Base 16 digits are called *hexadecimal*. We will think of these as elements of $\mathbf{Z}_2^4$.

Instead of addition mod N, the system uses bitwise exclusive or. We will denote this operation by $\oplus$. On hex digits we have, for example,

$$B \oplus 6 = 1011 \oplus 0110 = 1101 = D.$$

Note that this is just addition in $\mathbf{Z}_2$ on each bit. This can be extended to sequences of bits or hex digits in the obvious way.

Encryption works as follows. The user supplies a key of up to 75 characters. (Presumably, the length 75 was chosen so the key will fit on one line.)

First, a 16-bit check sum S is formed from the key. Each bit of the check sum is a linear combination of the bits of the key.

Suppose the user wishes to encrypt the text T. If the key has length ℓ, the sequence

$$A = \ell, \ell + 1, \ell + 2, \ldots, \mathrm{FF}, 00, 01, \ldots$$

(an ascending sequence mod 256, starting with the key length) is formed, followed by

$$U = T \oplus A.$$

As in the classic Vigènere system, the key is written down, as many times as needed, to form a sequence K. The encryption of T is then

$$C = U \oplus K = T \oplus A \oplus K.$$

What is actually stored in the file? The file on the user's disk contains

$$\mathrm{FE\ FF\ 61\ 61} \circ S \circ C,$$

where we have denoted concatenation by $\circ$. The first block apparently tags the file as encrypted by Word Perfect. The second block is the check sum, and the remainder of the file is encrypted text.

Carroll points out various weaknesses of this system (marketing claims to the contrary!). One obvious one is that the check sum is stored in the file, reducing the set of possible keys somewhat.

In defense of the system, one can say that it is probably strong enough for every-day use, offering roughly the same level of security as the lock on a car door.

The cell phone system CAVE apparently also used Vigenère. [See Diffie and Landau, Privacy on the Line.]

Lecture 5

This lecture covers *Hill ciphers*.

Mixing

Simple substitutions, even polyalphabetic ones, just relabel the letters of the plaintext. It is generally preferable to have each ciphertext letter depend not just on corresponding cleartext letter but on neighboring ones as well.

An elegant system for doing this was invented by Lester Hill in the 1920's. [L. Hill, Cryptography in an Algebraic Alphabet, Amer. Math. Monthly, 1929.] This work had an important effect: abstract algebra became a fundamental tool in cryptography.

Linear Transformations and Matrices

Suppose we have a system like

$$z = 2x + 5y$$
$$w = 3x + 7y$$

We'll think of this as a transformation taking x and y to z and w. We abbreviate this by using matrices:

$$\begin{pmatrix} z \\ w \end{pmatrix} = \begin{pmatrix} 2 & 5 \\ 3 & 7 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

There are corresponding formulas for linear transformations on n variables. In particular, given an $n \times n$ matrix (m_{ij}), we let

$$\begin{pmatrix} m_{11} & \cdots & m_{1n} \\ m_{21} & \cdots & m_{2n} \\ & \vdots & \\ m_{n1} & \cdots & m_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} m_{11}x_1 + \cdots + m_{1n}x_n \\ m_{21}x_1 + \cdots + m_{2n}x_n \\ \vdots \\ m_{n1}x_1 + \cdots + m_{nn}x_n \end{pmatrix}$$

Hill Ciphers

This is a system with $P = C = \mathbf{Z}_N{}^n$.

The key space consists of certain $n \times n$ matrices M with entries in $\mathbf{Z}_N$. In particular, if

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

and $k = M$, then

$$e_k(x) = Mx.$$

As we will see, the existence of a decryption function forces some requirements on M.

Example for $n = 2$.

Let's take $N = 26$ as usual, and $M = \begin{pmatrix} 2 & 5 \\ 3 & 7 \end{pmatrix}$. A sample message, converted to numerical form, is

R	U	T	H	S	T	R	I	K	E	S	O	U	T
17	20	19	7	18	19	17	8	10	4	18	14	20	19

We encrypt block 1 as follows (doing arithmetic mod 26)

$$\begin{pmatrix} 2 & 5 \\ 3 & 7 \end{pmatrix} \begin{pmatrix} 17 \\ 20 \end{pmatrix} = \begin{pmatrix} 2 \cdot 17 + 5 \cdot 20 \\ 3 \cdot 17 + 7 \cdot 20 \end{pmatrix} = \begin{pmatrix} 8 + 22 \\ 25 + 10 \end{pmatrix} = \begin{pmatrix} 4 \\ 9 \end{pmatrix}$$

Doing the other blocks similarly, we get

4	9	21	2	1	5	22	3	14	6	2	22	5	11
E	J	V	C	B	F	W	D	O	G	C	W	F	L

How do we decrypt? For 2×2 matrices there is a formula for the inverse matrix (that is, the matrix representing the inverse transformation):

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = (ad - bc)^{-1} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

So

$$\begin{pmatrix} 2 & 5 \\ 3 & 7 \end{pmatrix}^{-1} = (-1) \begin{pmatrix} 7 & -5 \\ -3 & 2 \end{pmatrix} = \begin{pmatrix} 19 & 5 \\ 3 & 24 \end{pmatrix}.$$

To check this, observe that it correctly decrypts the first block:

$$\begin{pmatrix} 19 & 5 \\ 3 & 24 \end{pmatrix} \begin{pmatrix} 4 \\ 9 \end{pmatrix} = \begin{pmatrix} 17 \\ 20 \end{pmatrix}$$

Decryption in general

Note that only certain matrices will work for M. For example, if $M = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}$, then only even numbers can be produced as ciphertext. This violates our basic assumption that encryption mappings are invertible.

Theorem. For $n \times n$ matrices M over $\mathbf{Z}_N$, the following are equivalent: a) the mapping given by M is 1-1; b) the mapping given by M is onto; c) M has an inverse matrix with coefficients in $\mathbf{Z}_N$; d) $\det(M) \in \mathbf{Z}_N^*$.

We leave the equivalence of a) – c) for you to prove. The most interesting part is that c) and d) are equivalent.

First, if M is invertible, then we have

$$1 = \det(I) = \det(MM^{-1}) = \det(M)\det(M^{-1}),$$

so $\det(M)$ is in $\mathbf{Z}_N^*$.

Conversely, if $\det(M) \in \mathbf{Z}_N^*$, we can use Cramer's rule to solve

$$Mx = y_i,$$

where y_i is the vector with a 1 in the i-th position and 0's everywhere else. Doing this for $i = 1, \ldots, n$, we obtain the columns of M^{-1}.

Inverting matrices mod N.

When n is large, inverse matrices can be found by a procedure similar to Gauss-Jordan elimination.

Example: Let

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{pmatrix}$$

We will solve the system

$$\begin{aligned} x + 2y + 3z &= a \\ 4x + 5y + 6z &= b \\ 7x + 8y + 10z &= c \end{aligned}$$

mod N using elementary row operations. These are: i) multiply any equation by an element of $\mathbf{Z}_N^*$, and ii) replace row i by row i plus a multiple of row j, where $i \neq j$. As usual we take $N = 26$.

We only need to write down the coefficients. We start with

$$\begin{pmatrix} 1 & 2 & 3 & 1 & 0 & 0 \\ 4 & 5 & 6 & 0 & 1 & 0 \\ 7 & 8 & 10 & 0 & 0 & 1 \end{pmatrix}$$

Let

$$\text{row } 2 = \text{ row } 2 - 4 \text{ row } 1$$

and then

$$\text{row } 3 = \text{ row } 3 - 7 \text{ row } 1.$$

This gives

$$\begin{pmatrix} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 23 & 20 & 22 & 1 & 0 \\ 0 & 20 & 15 & 19 & 0 & 1 \end{pmatrix}$$

Multiplying row 2 by $23^{-1} = 17$, we get

$$\begin{pmatrix} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & 2 & 10 & 17 & 0 \\ 0 & 20 & 15 & 19 & 0 & 1 \end{pmatrix}$$

Now we let

$$\text{row } 1 = \text{ row } 1 - 2 \text{ row } 2$$

and

$$\text{row } 3 = \text{ row } 3 - 20 \text{ row } 2$$

and get

$$\begin{pmatrix} 1 & 0 & 25 & 7 & 18 & 0 \\ 0 & 1 & 2 & 10 & 17 & 0 \\ 0 & 0 & 1 & 1 & 24 & 1 \end{pmatrix}$$

Finally, we let

$$\text{row } 1 = \text{ row } 1 - 25 \text{ row } 3$$

$$\text{row } 2 = \text{ row } 2 - 2 \text{ row } 3$$

which produces

$$\begin{pmatrix} 1 & 0 & 0 & 8 & 16 & 1 \\ 0 & 1 & 0 & 8 & 21 & 24 \\ 0 & 0 & 1 & 1 & 24 & 1 \end{pmatrix}$$

Therefore, we have

$$\begin{aligned} x &= 8a + 16b + c \\ y &= 8a + 21b + 24c \\ z &= a + 24b + c \end{aligned}$$

so the inverse matrix is

$$M^{-1} = \begin{pmatrix} 8 & 16 & 1 \\ 8 & 21 & 24 \\ 1 & 24 & 1 \end{pmatrix}.$$

A potential snag with this procedure is that we always need to have a pivot that is an element of $\mathbf{Z}_N^*$. If one is not found in the correct position it can always be manufactured. We note the following result without proof.

Lemma: Let $a_1, \ldots, a_n \in \mathbf{Z}_N$ have the property that some linear combination of these elements is 1. Then there are $x_2, \ldots, x_n \in \mathbf{Z}_N$ and $u \in \mathbf{Z}_N^*$ such that

$$a_1 + \sum_{i \geq 2} x_i a_i = u.$$

Elements x_i as needed can always be found efficiently. In particular, if you choose random $x_i \in \mathbf{Z}_N$, your chance that this will make a unit is $\varphi(N)/N$. As we will see shortly, this implies that the number of trials before this works is $O(\log \log N)$.

Now, suppose we are trying to apply our elimination procedure to an invertible matrix M. If the upper left element is in $\mathbf{Z}_N^*$, we have no problem and can use it as a pivot. If not, we observe that the mapping $x \to xM$ on row vectors is onto, so some linear combination of the elements in the first column must be 1. By the lemma, we can

apply row operations to make the upper left element 1, and use further row operations to make all elements below this 0. Now, after k steps of this procedure, we will have a matrix of the form

$$\begin{pmatrix} I_k & X_k & \cdots \\ 0 & M_k & \cdots \end{pmatrix}$$

where I_k is a $k \times k$ identity matrix, and M_k is $(n-k) \times (n-k)$. Row operations do not change the determinant, so $\det(M) = \det(I_k)\det(M_k) = \det(M_k)$, implying that M_k is invertible. By the same argument we used for the full matrix, we can use row operations on the last $n-k$ rows to make the upper left element of M_k into a suitable pivot (i.e. an element of $\mathbf{Z}_N^*$). This allows elimination to proceed.

Notes

If we follow a Hill encipherment by Vigenère encryption, we get the *affine block cipher* $x \longrightarrow Ax + b$.

The algorithm FEAL uses an affine block cipher with $N = 256$ as an internal component. (Reference?)

Here is an exact computation of the chance that the construction of the lemma will work. First,

$$\Pr[a_1 + \sum_{i \geq 2} x_i a_i = u \in \mathbf{Z}_N^*] = \prod_{p|N} \Pr[a_1 + \sum_{i \geq 2} x_i a_i \not\equiv 0 \bmod p].$$

Considering the situation mod p, there are two cases. If one of $a_2, \ldots, a_n$ is not divisible by p, then

$$\Pr[a_1 + \sum_{i \geq 2} x_i a_i \not\equiv 0 \bmod p] = \sum_{u \in \mathbf{Z}_p^*} \Pr[\sum_{i \geq 2} x_i a_i = a_1 - u] = (p-1)/p.$$

If, on the other hand, p divides all of $a_2, \ldots, a_n$, then our assumption implies that p does not divide a_1. In this case

$$\Pr[a_1 + \sum_{i \geq 2} x_i a_i \not\equiv 0 \bmod p] = 1.$$

So the probability of success is $\prod(1 - 1/p)$, where the product is taken over primes p dividing N, but not dividing all of $a_2, \ldots, a_n$.

Lecture 6

This lecture discusses *transposition*.

Permutation matrices

Last time we studied a block cipher based on $n \times n$ matrices over $\mathbf{Z}_N$. In this system, the key is the matrix M, which we multiply by the block x:

$$e(x) = Mx.$$

The corresponding decryption function is

$$d(x) = M^{-1}x.$$

Inverting the matrix M is possible but not pleasant. This suggests we look for classes of matrices whose inverses are easy to find.

One such class is the class of *permutation matrices*. We say that M is a permutation matrix if its entries are 0 or 1, with exactly one 1 in each row and column.

Multiplication by a permutation matrix rearranges elements. For example:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_2 \\ x_3 \\ x_1 \end{pmatrix}$$

Transposition ciphers

A special case of the Hill cipher, then, involves encrypting by simply rearranging the symbols in a block.

This is a classical idea, called a *transposition cipher*. Usually the matrices are not written down explicitly, rather, a permutation of $\{1, \ldots, n\}$ is implied by the choice of key.

We'll discuss one classic method, called *keyed columnar transposition*.

Choose a key word, say

B O M B E

Note that letters can be repeated. Let ℓ be its length, so that here $\ell = 5$.

This determines a permutation of $\{1, \ldots, \ell\}$, e.g.

1	5	4	2	3
B	O	M	B	E

That is, we assign 1 to the earliest letter in the alphabet occurring in the key. If there is a tie, choose the first one in the key. Cross this letter off and repeat

the same process to assign 2,3,4, and so on until every letter has been given a number.

Now, take the message, say

C R Y P T O G R A P H Y I S F U N

Write the numbers in the first row of an array with ℓ columns, followed by the message letters left-to-right as usual:

1	5	4	2	3
C	R	Y	P	T
O	G	R	A	P
H	Y	I	S	F
U	N	X	Y	Z

The last 3 letters, called *nulls*, are used to fill out the rectangle.

Finally, read out the message by columns, in numerical order:

C O H U P A S Y T P F Z Y R I X R G Y N

The receiver gets this and uses the key to reconstruct the message.

In computer programming terms, one can see that what is going on is this. The message is put into a 2-dimensional array in row major order. It is read out in column major order, with a column ordering derived from a key.

With this system, factoring the message length gives strong clues as to the structure of the cryptogram. To prevent this, one can leave off the nulls, in which case the cryptogram becomes

C O H U P A S T P F Y R I X R G Y N

There are an enormous number of variations on this basic idea. For example, one could read in by columns, and out by diagonals, or use a spiral pattern, or ...

Transposition as a cipher ingredient

In modern systems, transposition is commonly used in conjunction with other scrambling operations. We give two examples.

ADFGX

This cipher was used by the German army in World War I. Its name derives from five letters whose Morse codes are easy to distinguish:

A	· —
D	— · · ·
F	· · — ·
G	— — ·
X	— · · —

Cryptograms consisted only of these letters, and were derived by a two-step process involving substitution, then transposition.

We assume an underlying alphabet of 25 letters, not distinguishing between I and J. Using a key word, first write the alphabet into a 5×5 square. Suppose, for example the key word is KAISER. The square is

	A	D	F	G	X
A	K	A	I	S	E
D	R	B	C	D	F
F	G	H	L	M	N
G	O	P	Q	T	U
X	V	W	X	Y	Z

The square gives us a two-symbol encoding for each letter of the alphabet, so that A is AD, B is DD, and so on. We take the message, for example

I M W E S T E N N I C H T S N E U E S

and encode it using the square to get

AF FG XD AX AG GG AX FX FX AF DF FD GG AG GX AX GX AX

(We have preserved the pairs to make it easier for you to see what is going on.) This intermediate message was read row-by-row into a 10 (?) column square, whose columns were indexed using a key:

3	1	4	5	9	2	6	8	7	0
A	F	F	G	X	D	A	X	A	G
G	G	A	X	F	X	F	X	A	F
D	F	F	D	G	G	A	G	G	X
A	X	G	X	A	X				

Reading out the columns in key order gave the cryptogram:

G F X F G F X D X G X A G D A F A F G G X D X A F A A A G X X G X F G A

Transposition in DES

The Data Encryption Standard, of which we'll see more later, uses transposition as a preprocessing step and a postprocessing step.

DES encrypts 64-bit blocks. That is, input to the encryption process is a vector from $\mathbf{Z}_2^{64}$. Keys are 56 bits long.

DES does 18 transformations on the block. The first is a permutation σ of the entries

$$b_1 b_2 \cdots b_{64},$$

which does not depend on the key. The next 16 are different, and use information from the key in a way we will not describe here. (Each of these is called a "round.") Finally, the last transformation is the inverse of σ.

Since the initial and final permutations do not involve the key, one should ask whether any security is gained thereby. One answer is that they slow down software implementations of DES, since rearranging individual bits usually requires several machine instructions. This penalty would be felt by anyone using a program to cryptanalyze DES by trial and error.

It is of interest to see the 8×8 table which describes σ.

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

We interpret this to mean that bit 58 of the input becomes bit 1 after the initial permutation, bit 50 of the input becomes bit 2, and so on.

One can see this is in some sense the direct descendant of classical transposition schemes. There is an important lesson here for anyone studying cryptography (or any other technology, for that matter). Namely, there are a limited number of basic methods that get combined over and over again in the design of systems. (Less charitably, it is hard to be original!)

Lecture 7

This lecture begins our study of *Cryptanalysis.*

What's the goal here?

So far we have considered ciphers from the point of view of the code designer. We now want to take the opposite point of view, and consider what unauthorized users of a cryptographic system can do.

In analyzing cryptanalytic attacks on a system, it is usual to assume that the enemy knows the system being used. That is, any security must derive solely from the choice of key, not from ignorance of the general mechanism in use. This is called Kerchoff's Principle. [A. Kerckhoffs, La Cryptographie Militaire, ca. 1880.]

It is also not wise to think of systems as "unbreakable." Rather, we should focus on two questions:

For how long will an encrypted message be secure?

What is the cost to recover the key or the message?

Attack models.

It is standard to consider three possibilities:

Ciphertext only. Here, the cryptanalyst knows $y = e_k(x)$, or more generally,

$$y_1 y_2 \cdots y_n = e_k(x_1) e_k(x_2) \cdots e_k(x_n),$$

and wishes to determine the message or (even better) the key.

Known plaintext. The cryptanalyst possesses one or more matched pairs $(y_i, e_k(x_i))$, and wishes to determine k.

Chosen plaintext. The cryptanalyst has obtained a copy of the encryption device and can choose $x_1, \ldots, x_n$ and observe the ciphertexts $y_1 = e_k(x_1), \ldots, y_k = e_k(x_n)$, in an attempt to obtain the key k.

Our goal will be to estimate the probability of success of various attacks, or to estimate the average time until a given attack is successful.

We note that randomization enters into cryptanalysis in (at least) three ways:

Messages are not deterministic (otherwise there would be no need to send them!). This is usually modeled by invoking randomness.

Keys are usually chosen randomly. It is standard to assume that each key is equally likely.

The cryptanalyst can use randomization as part of an algorithm.

To study this properly, we need some notions of probability and statistics.

Probability models (in general).

Probability theory arose from the study of problems arising in gambling and insurance. Nowadays, one prefers an axiomatic approach, which was first stated clearly by Kolmogorov in the 1930's. We will start with the simplest case of this theory.

Let Ω be a finite or countable *sample space*. We assign each element x of Ω a *probability* p_x, satisfying

$$0 \leq p_x \leq 1$$

and

$$\sum_{x \in \Omega} p_x = 1$$

Example: Let us consider the following mini-scrabble game. There are 10 tiles labelled with the letters A through F. There are three E, two A, two C, and one each of the remaining letters.

Drawing a random tile can be modeled by taking

$$\Omega = \{\mathtt{A}, \ldots, \mathtt{F}\},$$

with

$$\Pr[\mathtt{A}] = 0.2, \Pr[\mathtt{B}] = 0.1, \Pr[\mathtt{C}] = 0.2, \Pr[\mathtt{D}] = 0.1, \Pr[\mathtt{E}] = 0.3, \Pr[\mathtt{F}] = 0.1.$$

Selection of probabilities in such a model must ultimately be based on experience.

A subset of Ω is called an *event*. We assign probabilities to events using the formula

$$\Pr[A] = \sum_{x \in A} p_x.$$

This has the consequence that

$\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$, with equality if A and B have no elements in common

$0 \leq \Pr[A] \leq 1$.

Observe also that

$\Pr[\Omega] = 1$ and $\Pr[\emptyset] = 0$.

We should think of $\Pr[A]$ as the long-term average frequency with which the event A occurs, assuming that we sample from Ω.

In our mini-scrabble example, we have

$$\Pr[\text{ vowel }] = 0.5$$

so

$$\Pr[\text{ consonant }] = 0.5$$

This illustrates the general fact that

$$\Pr[\bar{A}] = 1 - \Pr[A].$$

Usually, we are not interested in the outcome x per se but only in some function of it, say $f(x)$. Such functions are called *random variables*. The most important property of a random variable is its weighted average, called the *expected value*:

$$E[f] = \sum_{x \in \Omega} f(x) p_x.$$

Think of playing a game where you win $f(x)$ dollars if the outcome x occurs. If you play the game many times, you should expect to win an amount close to $E[f]$ per game.

Expectation has properties of an integral:

$$E[af + bg] = aE[f] + bE[g]$$

and

$$\min\{f\} \leq E[f] \leq \max\{f\}.$$

Random variables f, g are called *independent* if

$$\Pr[f(x) = a, g(y) = b] = \Pr[f(x) = a]\Pr[g(y) = b]$$

for all possible outcomes a and b.

Let us consider a standard 52-card deck. The random variables "suit" and "rank" are independent.

If f and g are independent, then

$$E[fg] = E[f]E[g].$$

Computing probabilities of events and expected values directly from the definition is often tedious, and we will seek and use other, less formal, ways to do these computations. One of the hallmarks of a skilled probabilist is a large bag of tricks for this purpose.

Finally, a warning. Uncountable sample spaces can be used, but they require special care. In particular, only certain sets can be used as events, and only certain functions can be used as random variables. Sets and functions that cannot be so used are rather difficult to construct unless one is trying to do so, however.

Probablistic models for search

Confronted with a long enciphered message, the crudest thing the cryptanalyst could do is to search for the key. We'll assume that decrypting with any key but the one that was used to encode the message produces incomprehensible junk.

This can be modelled using the following setup. Suppose we have an urn with 1 white ball and $n-1$ black balls. Choosing balls from the urn at random (this means the probability of obtaining any particular ball is $1/n$), we sample until we find the white ball.

First, suppose we return each ball to the urn and shake it before obtaining the next. This is called *sampling with replacement.* We let the random variable T denote the number of balls drawn up to and including the first white ball.

We will analyze $E[T]$ using a *decision tree.* This is a generally useful tool for computing probabilities and expected values. At the top level, the tree has branches labelled $1/n$ (leading to a leaf) and $(n-1)/n$ (leading to another tree of the same kind).

Because of the self-replicating nature of the tree, we have

$$E[T] = 1/n + \frac{n-1}{n}(1 + E[T])$$

that is,

$$E[T] = 1 + \frac{n-1}{n}E[T].$$

Solving this we get

$$E[T] = n.$$

This result can be stated another way. Suppose we flip coins for which heads occur with probability p. The expected number of coins we must flip to get the first head is exactly $1/p$.

We'll now consider another model, that of *sampling without replacement.* In this version, any ball removed from the urn is not returned. It is reasonable that this should find the white ball more quickly, since the proportion of black balls decreases with each step.

Let's think about this for $n = 4$. Imagine that all the balls are lined up and revealed to us one by one. There are four possible ways this could happen, each with probability $1/4$.

$$\begin{array}{ll} WBBB & T = 1 \\ BWBB & T = 2 \\ BBWB & T = 3 \\ BBBW & T = 4 \end{array}$$

Hence $E[T] = 1/4 \times (1 + 2 + 3 + 4) = 5/2$.

In general, the same argument works, and

$$E[T] = 1/n \sum_{i=1}^{n} i = (n+1)/2.$$

Lecture 8

This lecture covers key enumeration and some chosen plaintext and known plaintext attacks on our example ciphers.

Results on searching (proved last time)

Suppose we have an urn with n balls (the keys), of which one is white (the "correct" key) and the rest are black (the "incorrect" key).

If we sample from the urn with replacement the expected number of balls until the first white ball is n.

If we sample from the urn without replacement, the expected number is $n/2$.

These results have the following implications.

Key spaces should be large. Of course a large key space is necessary, but not sufficient, for security.

Unless there is good reason to do otherwise, the encoder should choose a key uniformly from the key space. (This should be formalized.)

Some keyspace estimates

Shift ciphers:

We have $e_k(x) = x + k \bmod N$, so $|K| = N$.

Affine ciphers:

We have $e_k(x) = ax + b \bmod N$ with $a \in \mathbf{Z}_N^*$.

The number of elements in $\mathbf{Z}_N^*$ is called *Euler's phi function*:

$$\varphi(N) = \#\{a \in \mathbf{Z}_N : \gcd(a, N) = 1\}.$$

Using the combinatorial principle of inclusion and exclusion it can be shown that

$$\varphi(N) = N\left(1 - \sum_{p|N} 1/p + \sum_{p,q|N} 1/pq - \cdots\right) = N\prod_{p|N}(1 - 1/p).$$

Let's verify this for $N = 26$. The possible primes p are 2 and 13, so we have

$$\varphi(26) = 26(1/2)(12/13) = 12.$$

Around 1900 Landau showed that

$$\varphi(N) \geq c\frac{N}{\log\log N}.$$

Since $K = \mathbf{Z}_N^* \times \mathbf{Z}_N$ we have the estimates

$$\frac{cN^2}{\log\log N} \leq |K| \leq N^2.$$

General substitutions:

Here K is the set of all permutations on N symbols.

According to Stirling's formula, we have

$$|K| = N! \sim \sqrt{2\pi N} N^N e^{-N}.$$

Vigenère:

If the block length is n, this is equivalent to choosing n separate shift ciphers. Hence

$$|K| = N^n$$

Hill ciphers:

We encrypt a block x of length n by multiplying it by an $n \times n$ invertible matrix M. Hence

$$K = \{M : \det(M) \in \mathbf{Z}_N^*\}.$$

Jordan proved (date?) that there are

$$N^{n^2} \prod_{p|N} (1 - 1/p)(1 - 1/p^2) \cdots (1 - 1/p^n)$$

invertible $n \times n$ matrices over $\mathbf{Z}_N$.

As an example, let's count the invertible 2×2 matrices with elements in $\mathbf{Z_2}$. The first row must be nonzero, so there are 3 possible first rows. Given any first row, the second row can be anything but a multiple of it. This leaves two possible second rows for each first row. Note that Jordan's formula gives

$$2^4(1 - 1/2)(1 - 1/4) = 6.$$

It can be shown that

$$\prod_p \prod_{i=2}^{\infty} (1 - 1/p^i) \geq 0.435...$$

so

$$c' \frac{\varphi(N)}{N} \leq \frac{|K|}{N^{n^2}} \leq \frac{\varphi(N)}{N}$$

Using Landau's bound for $\varphi(N)$ we get

$$c'' \frac{N^{n^2}}{\log\log N} \leq |K| \leq N^{n^2}$$

Let's summarize what we have done, and see how the message and key spaces compare in size. An interesting thing to do here is to estimate how many bits would be needed to write down a message, and how many bits for a key.

Cipher	$\log_2 \lvert M \rvert$	$\log_2 \lvert K \rvert$
Shift	$\log_2 N$	$\log_2 N$
Affine	$\log_2 N$	$\sim 2 \log_2 N$
General	$\log_2 N$	$\sim N \log_2 N$
Vigènere	$n \log_2 N$	$n \log_2 N$
Hill	$n \log_2 N$	$\sim n^2 \log_2 N$

Chosen plaintext attacks

If the cryptanalyst is allowed to choose plaintext and form the corresponding ciphertext, each of these systems can be attacked in an evident way. Remember that the sole goal here is to determine the key.

We will measure the cost of these attacks by the number of plaintexts required.

For the shift cipher, if $y = x + k$, then $k = x - y$. This requires 1 plaintext.

For the affine cipher, if

$$y = ax + b$$

the cryptanalyst can choose $x = 0$ to obtain b. Then taking $a = 1$ gives $a = y - b$. This requires 2 plaintexts.

For the general monoalphabet, the analyst must encrypt all possible plaintexts, of which there are N. (Actually $N - 1$ will do.)

The Vigenère system is attacked by encrypting $(0, \ldots, 0)$ which immediately gives the key. This requires encryption of 1 block.

The key for a Hill system can be determined by encrypting the elementary basis vectors $(0, \ldots, 0, 1, 0, \ldots, 0)^T$. This requires encryption of n blocks.

Note that:

All of these attacks are optimal, in the sense that one uses no more information, asymptotically, than is needed to determine the key.

For all systems save the general monoalphabet, the length of ciphertext needed is at worst a power of the length of the message. (Example: the Hill system has blocks of size $n \log_2 N$, and we use $n^2 \log_2 N$ bits of ciphertext.) Only general substitution requires a large amount of ciphertext. This suggests that a good cryptosystem will, in some sense, attempt to simulate a general substitution. An example of this is DES, which performs substitutions on a universe of size 2^{64}.

Known plaintext attacks

Here the cryptanalyst can, perhaps by observing traffic, obtain one or more plaintext-ciphertext pairs. To make this interesting we need a model for the plaintext source.

We will use a crude "monkeys and typewriters" idea. Let's assume that the plaintext symbols are chosen, uniformly and independently, from the set P. Thus, the plaintext stream is an infinite sequence

$$x_1, x_2, x_3, \ldots$$

of symbols, and we are assuming that for every n

$$\Pr[x_1 = a_1 \ldots x_n = a_n] = |P|^{-n}.$$

It is important to understand that better models are possible, but that this is often good enough to get a rough idea of what would occur in a real situation.

Analysis of known plaintext attacks

Again we are attempting to find the key. The number of plaintexts required until an attack succeeds is, in general, a random variable.

Keys for the shift cipher can be found as before. This attack is, in fact, deterministic, since any plaintext-ciphertext pair determines the key.

Consider the affine cipher. Suppose we have obtained two pairs (x_1, y_1) and (x_2, y_2) satisfying the relation

$$y_1 = ax_1 + b$$
$$y_2 = ax_2 + b$$

in $\mathbf{Z}_N$. We may rewrite this as

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}.$$

This can be solved for a and b provided that the determinant $x_1 - x_2$ is in $\mathbf{Z}_N^*$. We now introduce an idea that will be important in other contexts. Namely, if we have *any* x_1, if x_2 is chosen uniformly from $\mathbf{Z}_N$ in a way that does not depend on x_1, then

$$x_2 - x_1$$

is also distributed uniformly on $\mathbf{Z}_N$. That is, any shift of a random element is still random. To see this, we compute

$$\Pr[x_i - x_{i-1} = a] = \Pr[x_i = a + x_{i-1}] = 1/N.$$

By this argument, the chance that we can solve the system for unique a and b (which must be the key) is $\varphi(N)/N$. If this does not work we can try again with (x_2, y_2) and (x_3, y_3), and so on. Note that we have another variant on the urn model: each (x_i, y_i) gives us a new pair, and we are sampling with replacement,

with the chance of a good pair being $\varphi(N)/N$. Hence the expected number of pairs needed is

$$\frac{N}{\varphi(N)} = O(\log \log N),$$

where we have used Landau's estimate. (This estimate can probably be improved.)

Known plaintext analysis of a general substitution corresponds to a classic problem of probability theory called the coupon collection problem. To determine the key, we need to sample plaintext until each possible symbol has been seen. This corresponds to buying, say, baseball cards until one has at least one example of each possible card. We state without proof that the expected number of cards, if there are N possible ones, each equally likely, is

$$N(1 + 1/2 + \cdots + 1/N) \sim N \log N$$

[See e.g. W. Feller, An Introduction to Probability Theory and its Applications, p. 225.] It is crucial to note that we are considering random plaintext here. In real situations, the plaintext will have structure, and it is possible to determine the key much more quickly than this.

A key for the Vigenère system can be obtained from the first n plaintext-ciphertext pairs. This attack is also determinstic.

A known plaintext attack on Hill ciphers will be presented in the next lecture.

Notes

The plaintext models could be made a little more realistic. For coupon collecting with unequal probabilities, see Kullback, SMC [for mean number of coupons not seen by time t]; Rosén, Ann. Math. Stat. 1970 [asymptotics for waiting time]; von Schelling, Amer. Math. Monthly 1954 [mean waiting time]; Nath, Austral. J. Statist. 1973 [ditto].

Using dynamic programming, one can accurately compute the expected waiting time to collect n coupons (with unequal probabilities) in $O(n2^n)$ steps. For the 26 English letter frequencies appearing in Sinkov's book, the waiting time is 1266.00928... . Actually a substitution key is known as soon as we have ciphertext equivalents for $n-1$ letters, and for this, the waiting time is much smaller: 561.866... .

Lecture 9

This lecture finishes our discussion of known plaintext attacks and begins ciphertext-only cryptanalysis.

Known plaintext attack on the Hill cipher

We'll consider only the 2×2 case, as this contains the essential idea. As before the cleartexts $x_1, x_2, \ldots$ are randomly chosen from $\mathbf{Z}_N$.

Suppose the cryptanalyst observes

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

and

$$\begin{pmatrix} y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_3 \\ x_4 \end{pmatrix}$$

where a, b, c, d are unknown. This is equivalent to the following 4×4 system:

$$\begin{pmatrix} y_1 \\ y_3 \\ y_2 \\ y_4 \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & 0 & 0 \\ x_3 & x_4 & 0 & 0 \\ 0 & 0 & x_1 & x_2 \\ 0 & 0 & x_3 & x_4 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}.$$

The determinant of this system is

$$\begin{vmatrix} x_1 & x_2 \\ x_3 & x_4 \end{vmatrix}^2$$

which is in $\mathbf{Z}_N^*$ exactly when

$$\begin{vmatrix} x_1 & x_2 \\ x_3 & x_4 \end{vmatrix}$$

is. As we have observed, the chance of this is bounded below by a constant times $\varphi(N)/N$. Hence $O(\log \log N)$ plaintext-ciphertext pairs suffice on average to determine the key.

For $n \times n$ matrices, the corresponding estimate is $O(n^2 \log \log N)$.

Note: a more precise analysis of this problem appears in A. Konheim, Cryptography.

Ciphertext only attacks

The classic problem of cryptanalysis is to recover the message from a cryptogram. Such problems can be solved because the messages have structure. For cryptanalysis it is most fruitful to think of this structure as statistical in nature.

In natural languages such as English, letters occur with a definite frequency, which can be observed by sampling large texts. These frequencies are not uniform. That is to say, certain letters and symbol patterns are much more common than others.

In a long sample of English text, it is likely that about about 13% of the letters are `E`, about 9% `T`, and so on for the other letters. Sinkov (p. 177) reports on an experiment of this kind and gives frequencies for all letters.

It is important to realize that this is true of "typical" text, and that a given text may not be typical. An extreme case of this is the novel *Gadsby*, by E.V. Wright, which contains no instances of the letter `E`. (Such texts are called *lipograms*.)

There are also characteristic frequencies for digrams (pairs such as `XY`) and trigrams (triples like `XYZ`). Sinkov gives digram frequencies on p. 175.

Ciphertext-only analysis of shift ciphers

If N is small, we can compute all shifts of the cryptogram and see if any look like plaintext.

If we have a cryptogram that has been encrypted by a shift cipher, an obvious thing to do is to determine the most frequent letter. Assuming this represents `E` gives us a good guess for a plaintext-ciphertext pair, and hence a guess for the key. If this does not work, we can try again with the next most frequent letter, and so on.

For short cryptograms, we would like a more sensitive test. In particular, it makes sense that we should be using all of the frequency information, not just the top few letters. Here is a way to do this.

Let

$$P = (p_0, p_1, \ldots, p_{N-1})$$

be the expected frequencies of the plaintext language. We assume that we have sampled enough of the language to be confident that the first letter occurs with probability p_0, the second with probability p_1, and so on.

Let

$$Q = (q_0, q_1, \ldots, q_{N-1})$$

be the actual frequencies observed in our cryptogram. If we shift these frequencies down by k we get

$$Q^{(k)} = (q_k, q_{k+1}, \ldots, q_{N-1+k})$$

We want to select a k making $Q^{(k)}$ as close as possible, in some sense, to P. Since

$$e_k(x) = x + k$$

such a k will be a good guess for the encryption shift. (Draw the picture.)

Various measures of closeness are possible, but the easiest to work with is probably the Euclidean distance squared. By definition, this is

$$|P - Q^{(k)}|^2 = \sum_{i=0}^{N-1} (p_i - q_i^{(k)})^2$$
$$= \sum_{i=0}^{N-1} p_i^2 + \sum_{i=0}^{N-1} (q_i^{(k)})^2 - 2 \sum_{i=0}^{N-1} p_i q_i^{(k)}$$

We observe that the first sum is constant, and the second does not depend on k. Hence, minimizing this is the same as maximizing the *correlation*

$$\sum_{i=0}^{N-1} p_i q_i^{(k)}.$$

Note that this requires more work, about N^2 operations, than scanning the observed frequencies for a maximum. (Using the FFT it should be possible to compute the correlations with $O(N \log N)$ operations.) On the other hand, the test is more useful.

As a final note, we note that there is an expected value for the maximum correlation, when the text is large. Indeed, as the text size $n \to \infty$, the frequency of the i-th text letter will converge to p_i (assuming the text source obeys the weak law of large numbers). So the maximum possible correlation is

$$\sum p_i^2.$$

[John Gubner's e-mail to me contains a proof of this.] This number, called κ in statistical cryptanalysis, is characteristic of the plaintext language. Here are a few values for 26-letter alphabets (from Kullback, Statistical Methods in Cryptanalysis):

English $\kappa = 0.066$

Spanish $\kappa = 0.078$

Random $\kappa = 1/26 = 0.038$

Notes

Correlation cryptanalysis is extremely fast, but it is not as sensitive as another method, called *maximum likelihood.*

There is an English novel with no commas. (Source: Eats, Shoots, and Leaves.)

Georges Perec, famous for omitting the letter E in an entire novel, belonged to a literary movement whose mission was to play with language in this fashion. There is a fascinating description of these people in an article in New York Review of Books, mid-2006. (Or was it NYT Book Review?)

Frequency analysis is used by web browsers to solve the following problem. Different languages have different conventions for interpreting the "high" (≥ 128) characters in an 8-bit character set. Using frequency analysis on character strings, you can get a good idea of what the language is, so as to properly display the string. [Source: Joel Spolsky, The Absolute Minimum Every Software Developer Absolutely, Positively, Must Know about Unicode and Character Sets (No Excuses!), Manuscript, October 8, 2003.]

It is important to understand that a bald question such as "is shift encryption secure?" cannot be answered without knowing the way in which the system is being used. As a way to encrypt even short text messages, shifting is very insecure. However, it can be useful when the message is comparable in length to the key. A nice example of this is the Williamson key agreement protocol which will be discussed in upcoming lecture on discrete logarithms.

It would be good to have quantitative information about the effectiveness of correlation analysis for shifting.

The "FFT" algorithm is apparently the following. Form the two polynomials

$$f(Z) = \sum_{i=0}^{N-1} p_i Z^i,$$

$$g(Z) = \sum_{i=0}^{N-1} q_{N-1-i} Z^i.$$

(Note the coefficients of g have been reversed.) Using an $O(N \log N)$ algorithm for polynomial multiplication, compute

$$h = fg \bmod Z^N - 1.$$

The coefficients of h give the required correlations.

Lecture 10

Continuing with ciphertext-only cryptanalysis.

Expected versus observed frequencies.

Last time we noted the empirical fact that letter frequencies in a natural language tend to be stable. For example, English typically contains about 13% E, and so on.

A more subtle question is the following: how much text must we look at before the observed frequencies are close to the expected (long term) frequencies?

One way to think about this is to consider an individual letter x, and assume that each symbol of the text independently decides to be an x or not, with probabilities p and $1-p$, respectively. (This isn't really how it works, but it will get us started.) In this case, the actual number of occurrences of x will have a bell-shaped distribution.

Technical note: It will be a binomial distribution with parameters n and p. That is, in a text of length n, the probability that we see i occurrences of this letter is

$$P_i := \binom{n}{i} p^i (1-p)^{n-i}$$

Since each position has probability p of having an x or not, the mean number of times x occurs is given by

$$\mu = np$$

The fluctuation about this mean is measured by the *standard deviation*. For the binomial distribution, this is given by

$$\sigma = \sqrt{npq}$$

The rough idea is that we expect to see fluctuations comparable to σ.

More precise information can be obtained from the DeMoivre-Laplace limit theorem. This states that if C is the actual count of x's, then as $n \to \infty$.

$$\Pr[\frac{C-\mu}{\sigma} < a] \to \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{a} e^{-t^2/2} dt$$

Values of the integral (called the Gaussian or normal integral) are tabulated in books on probability and statistics.

More precisely, we expect to see

$$np - \sqrt{np(1-p)} \leq \text{ count of } x\text{'s } \leq np + \sqrt{np(1-p)}$$

about 68% of the time, and

$$np - 2\sqrt{np(1-p)} \leq \text{ count of } x\text{'s } \leq np + 2\sqrt{np(1-p)}$$

about 96% of the time.

These percentages come directly from numerical evalation of the Gaussian integral.

Let's take English E as an example, using $p = 0.13$. We then have

$$\mu = 0.13n$$
$$\sigma = \sqrt{(0.13)(0.87)n} = 0.34\sqrt{n}$$

With high probability, then, the actual fraction of letters that are E in English text will be

$$0.13\left(1 + O(n^{-1/2})\right).$$

How does this compare to real data? Kullback (Statistical Methods in Cryptanalysis, p. 95) counted the number of E's in five texts of 10,000 letters each. His values were

$$1367, 1294, 1292, 1275, 1270.$$

Our model predicts an average of 1300 occurrences (that's good, the mean of Kullback's data is 1299.6), with 95% of the observations within 68 of the mean (again reasonable, the largest deviation from the mean is 67).

This allows us to make a back of the envelope estimate for how much text is needed to separate E from the next most common letter T. We'll see, roughly, a fraction

$$0.13 \pm 0.34/\sqrt{n}$$

of E's, and

$$0.092 \pm 0.29/\sqrt{n}$$

of T's. The difference in means will equal the smaller standard deviation when

$$0.038 = 0.29/\sqrt{n}$$

that is when

$$n = \left(\frac{0.29}{0.038}\right)^2 \approx 58.$$

One could try to do this more precisely. One idea is to let X be the frequency of E's and Y the frequency of T's. Approximating these by normal variates and assuming they are independent (they are in fact negatively correlated, since each E is a position that cannot be occupied by a T), we can work out the probability that $X - Y > 0$ in a text of length n. For $n = 58$ it is about 75%.

Cryptanalysis of Affine Ciphers

If N is small, we could examine all keys and look for something resembling plaintext.

This could be automated by looking for the best match between the reference distribution

$$P = (p_0, p_1, \ldots, p_{N-1})$$

and the transformed distribution

$$Q_{a,b} = (q_{0a+b}, q_{a+b}, \ldots, q_{(N-1)a+b})$$

Correlation, as for the shift ciphers, provides a tool for choosing which values of a and b give the best match.

As with shift ciphers, the correct key should produce the best match. On the other hand, its correctness is now much less obvious on small texts, because the peaks and troughs of the standard frequency distribution are mixed up. Experiments with a 29 character plaintext gave, for example top five correlations of

0.060448	$a = 1, b = 0$
0.059517	$a = 5, b = 0$
0.057103	$a = 19, b = 22$
0.056034	$a = 11, b = 2$
0.055862	$a = 7, b = 18$

So it seems reasonable to use this technique to filter out likely candidates for further analysis.

[Could this be studied more quantitatively?]

Monoalphabets

For general substitutions, we cannot hope to consider all keys unless N is small, for there are $N!$ of them.

The easiest case to consider is the standard puzzle cryptogram, in which spaces are known.

This is not a bad assumption. In English text, the average word length is about 4.5 characters, so we expect about 22% of the symbols to be spaces. (To measure word lengths, you can use the Unix utility `wc`, which prints the number of lines, words, and bytes in a file.) Very likely, then the most frequent character is a space, and even if it is transformed the cryptanalyst will know what it is.

If word divisions are known, the strongest attack is the use of `pattern words`.

Consider a long word like `NINETEEN`. Various letters repeat. We can indicate the precise location of these repetitions by the pattern

```
A B A C D C C A
```

Apparently, `NINETEEN` is the only English word with this pattern. So if we see this pattern in a cryptogram, we know the encrypted values of `E`, `T`, `N`, `I`.

Traditional books on cryptanalysis provided lists of words matching various patterns. (See, e.g. Callimahos and Friedman, Military Cryptanalytics. I think Aegean Park Press has something too.)

Nowadays it is easier to write a computer program to search a word list. For example, the local AFS file system (for Unix systems) has a list in the file

`/usr/share/dict/words`,

and we can write a program that considers each word in turn and prints it if it matches the pattern. Since words are short, a simple $O(n^2)$ algorithm suffices for the matching.

The idea of the algorithm is to put each word into a standard form. We start by assigning the first letter the value `A`. Now, for each letter of the word, we consider it in turn. If it has already been given a value, we write down that letter. If it has not been given a value, we give it the next available value. For example, the first three characters of `NINETEEN` have the values `ABA`. The fourth character is `E`. Searching through the first three characters, we see it doesn't appear. Hence we give it the next available value `C`.

To solve an actual cryptogram, we look for large words with interesting patterns. Usually only a few of the possibilities will be common words, and consistent with each other. Trying these, we usually have enough text to determine the rest of the cryptogram.

It is an interesting question as to how this can be automated. One idea would be to make a graph of all the words matching the patterns in the text, and put an edge between two words if they follow each other in the cryptogram, and have consistent letter assignments. With luck there will not be too many complete paths from start to finish. Although I have not tried this out, you may find it instructive to do so.

Notes.

The web site `ftp://ftp.ox.ac.uk/pub/wordlists` has a bunch of word lists for various languages.

Words (or, more generally, character strings) with the same pattern are sometimes called *isomorphs*. For example, the binary sequences 100101 and 011010 are isomorphs.

Words that don't repeat any letters have been called *isograms*. The longest isogram in English is apparently UNCOPYRIGHTABLE.

Lecture 11

In this lecture, we finish up monoalphabetic cryptanalysis, and begin attacks on polyalphabets.

Space Elimination

As we have seen, a monoalphabetic cipher with word divisions exposed offers very little security against a dictionary attack based on pattern words.

For this reason, professional cryptographers eliminated spaces. This is a primitive form of data compression that makes a text harder to read even if it is not encrypted:

```
NOWIS THEWI NTERO FOURD ISCON TENT .
```

Five-letter blocks seem to date from the days of telegraphy. Among other things, using blocks of five made it easier for the sender to include a character count, which the receiver could check.

Monoalphabets with no word division

Solving these involves special skills and an appreciation for quirky features of the plaintext language. Accordingly, we will discuss only a few ideas, from Sinkov's book. Another classic is Helen Fourché Gaines, Cryptanalysis.

One might think that straight frequency analysis will solve any simple substitution cryptogram. Unfortunately, the letter frequencies are rather close in certain cases, and we cannot expect perfect matching of the expected frequencies even for rather long texts.

Example: consider the data on p. 95 of Kullback's Statistical Methods in Cryptanalysis. The top eight letters, with their ranks in five 10,000 character messages, are:

E	1	1	1	1	1
T	2	2	2	2	2
N	3	3	3	3	4
R	4	6	6	6	3
I	5	7	5	7	7
A	6	4	7	5	6
O	7	5	4	4	5
S	8	8	8	8	8

For this size of text, the top 3 letters and the last letter have fairly stable ranks, but we are unlikely to be able to distinguish the rest from frequency alone.

Sinkov recommends the following approach: 1) Use frequency counts to determine a set of high-frequency letters; 2) Separate vowels from consonants and find the encryptions of each high-frequency letter; 3) Finish off the cryptogram using pattern words.

High-frequency letters

In ordinary English, the top 8 letters `E T R I A O S` account for about 68% of the text. This is an example of the so-called 80-20 rule: the lion's share of the text is produced by relatively few letters. (Since $8/26 = 30.7\%$, perhaps we should call it the 68-31 rule.)

Usually, identification of these letters will expose enough text to allow the rest of the cryptogram to be guessed. Opening up The Portable Curmudgeon (this should be done frequently!) we find the following quote from Anatole France:

```
THE LAW IN ITS MAJESTIC EQUALITY
FORBIDS THE RICH AS WELL AS THE POOR
TO SLEEP UNDER BRIDGES TO BEG IN THE STREETS
AND TO STEAL BREAD
```

If we knock out the uncommon letters we are left with

```
THE*A *INIT S*A*E STI*E **A*I T**OR *I*STH
ERI** AS*E* *ASTH E*OO* TOS*E E**N* ER*RI*
*ESTO *E*IN THEST REETS AN*TO STEA* *REA*
```

Note that quite a bit of the plaintext is exposed here. A simple coin-flip model can explain this. Suppose we record the results of flipping a coin having $\Pr[1] = p$ and $\Pr[0] = 1 - p$. The longest run of 1's (which models the longest run of "common" letters) in a sequence of length n has mean length

$$\log_{1/p} n = -\log n / \log p.$$

[Need a reference for this. Is this exact?] In our example, we have the 14 character string

```
INTHESTREETSAN
```

On the other hand, our model predicts a longest run of

$$-\log 107 / \log 0.68 = 12.1$$

So we are in the right ballpark on this example.

Another reason to go after the common letters is that they make up many of the short prepositions, articles, etc. that glue longer words together. In our example, we would probably guess that the next letter after the string is `D` and then look for a verb to follow `TO`.

To extend the solution, one needs to be able to pattern match against a dictionary using wild cards. This is easy using the Unix utility `grep`. For example, on my workstation, the command

```
grep -i x.yz /usr/share/dict/words
```

finds all words with `X`, some other letter, and then `YZ`.

Separating vowels from consonants

Among the top 8 letters `A E I N O R S T`, half are vowels. In English text, vowels tend to avoid each other. (Some people say that vowels and consonants alternate, but that isn't quite right.)

The pattern can be modeled by a device with two states, which we will call V and C. The transition rules are:

$$V \to V \text{ with probability } 17\%$$
$$V \to C \text{ with probability } 83\%$$
$$C \to V \text{ with probability } 57\%$$
$$C \to C \text{ with probability } 43\%$$

[Draw the picture.]

(Note for the cognoscenti: this is a 2-state Markov chain.) The percentages came from sampling a 3000 character text with no distinguishing stylistic features (a proposal!). The letters `A E I O U Y` were considered vowels.

For data on vowel/consonant n-grams up to $n = 5$, see C. Meyer and S. Matyas, Cryptography, p. 718.

There are various classic criteria that give evidence of vowelhood, including: a) appears in many different digrams; b) appear in reversed digrams (`XY` and `YX`); c) don't appear together very often.

There are no sure-fire methods of vowel recognition in the open literature, but various ideas come to mind, including: i) try to make the best match to a finite-state model (this seems not to work very well); ii) put an edge between letters X and Y if they appear together in a digram, and then try to find a separation into two sets containing the fewest edges (I have not tried this. Optimal graph bisection is NP-complete, but we don't have many vertices here. Possible reference: John Carroll, Cryptologia, 1980's); iii) use singular value decomposition (see Forsythe/Malcolm/Moler, Computer Methods for Mathematical Computations, p. 238).

If nothing else works, it is well to remember that there are only $\binom{8}{4} = 70$ putative vowel sets to consider, and one could simply try them all.

Polyalphabetic ciphers

The general idea is to determine the period, and then use techniques special to the cryptosystem to reduce to a monoalphabetic substitution.

To make a polyalphabetic cryptogram, we choose a sequence of m encryption functions $e_1, e_2, \ldots, e_m$ and then encode the message

$$x_1 x_2 \cdots x_m x_{m+1} \cdots x_n$$

as

$$y_1 y_2 \cdots y_m y_{m+1} \cdots y_n$$

with $y_i = e_{i \bmod m}(x_i)$. (Here we agree that e_0 is the same as e_m.) The receiver knows the corresponding decryption functions $d_1, \ldots, d_m$ and can read the message.

The integer m is called the *period*.

Polyalphabetic ciphers resisted cryptanalysis until about the middle 1800's. The first good idea was published by F. W. Kasiski, a Prussian army officer, in 1863.

Kasiski observed that texts tend to repeat words, simply because these words often refer to the subject of the message. For example, in an American news broadcast we might expect the word `CLINTON` to appear several times. About 1 in m of these repetitions will occur at distances that are multiples of the period, and these will be encrypted identically, leading to repeated text in the cryptogram. Examining the factors of these distances gives us information about the period.

Consider the following example (Kahn, p. 208). Let us encrypt the message

`TOBEORNOTTOBETHATISTHEQUESTION`

using Vigenére with the keyword `RUN`. This produces

`KIOVIEEIGKIOVNURNVJNUVKHVMGZIA`

We notice that `KIOV` occurs at positions 1 and 10, which are 9 apart. Similarly, `NU` occurs at positions 14 and 20, which are 6 apart. Since

$$9 = 3^2$$
$$6 = 2 \cdot 3$$

we might guess that the period is 3.

In practice one must have some idea of how likely spurious repetitions are. The textbooks are not clear on this. We can, however, think about our example a little. In a random text of n characters from an N-letter alphabet, we expect to see $\sim n^2/(2N^4)$ four-letter repetitions, which for $n = 30$ and $N = 26$ works out to about 10^{-3}. Similarly, the expected number of two-letter repetitions is $\sim n^2/(2N^2)$, which for this example is much higher: about 66%. Therefore, we reject the idea that `KIOV` is an accidental repetition, but reserve judgement on the digram `NU`.

Taking the work of factoring as constant (the distances are no more than the length of the text), Kasiski's test can be done with an $O(n^2)$ algorithm. The idea

is simply to march two pointers i and j along the text, spitting out any pairs (i, j) with

$$y_i y_{i+1} \cdots y_{i+k} = y_j y_{j+1} \cdots y_{j+k}$$

for suitably large k. (The average value of k is $O(1)$.) The differences $j - i$ are then factored and displayed for the cryptanalyst's benefit.

[Maybe this deserves more careful analysis. We could use a sieve and precompute all possible factorizations then look these up as needed. The $O(1)$ value for the expected value of k comes from assuming random text. Real text would have different statistics.]

Notes

Monoalphabetically encrypted messages with spaces eliminated were used (briefly) by at least one South American intelligence network in the 1970's. The ciphertext was sent via telex. Eventually hand encryption was replaced by a US-supplied machine, presumably using stronger methods. Reference: John Dinges, The Condor Years: How Pinochet and His Allies Brought Terrorism to Three Continents, 1995, p. 122. See also Gerardo Irustra, Espionaje y servicios secretos en Bolivia, La Paz, 1995, p. 301.

Maximum-likelihood estimate for monoalphabetic substitutions: sort letters by frequency and match up to plaintext symbols. Mathematically elegant but not useful unless you have a lot of data. (Isn't there a recent IEEE-IT paper on this?)

Back of the envelope calculation for the number of ways to put spacings into a text. Assume n letters, if average letter is 5 chars long we will need $n/5$ spaces. Can assign them in $\binom{6n/5}{n/5}$ ways. [How large is this, asymptotically?] For $n = 50$ this is already pretty large, about 7×10^{10}.

Kasiski analysis can be reduced to sorting, as implemented on punched card machines by Snyder and Clark. (See Budiansky, Battle of Wits, p. 215.) In modern terms, here is the idea. Suppose we want to look for repeats of length up to m. Make a list of all m-character segments of the text, together with an index number indicating where the segment appeared. Sort the list on the first component.

Meyer and Matyas (p. 641) discuss a "local search" method which they ascribe to Don Coppersmith. The idea is to maintain a tentative decryption (i.e. a permutation of 26 letters), along with a digram frequency matrix. Start with a permutation given by frequency analysis. Choose a pair of letters at random and interchange. If they improve the digram matrix (making it closer to the digram matrix for plaintext), accept the interchange. Repeat this process.

For more details on singular value decomposition for vowel identification, see Moler and Morrison, Singular Value Analysis of Cryptograms, AMM v. 90, 1983, pp. 78-87. The idea is to use SVD to solve a variation of the maximum-cut problem. Empirically, this technique seems to become useful for texts of 1000 characters.

Jacques Guy has an interesting article on vowel recognition, Cryptologia, early 1990's. The algorithm is due to Sukhotin, and looks like a simple greedy heuristic for MAX CUT.

Lecture 12

This lecture: ciphertext-only cryptanalysis of polyalphabetic ciphers.

Recall that the sender chooses k encryption functions $e_1, e_2, \ldots, e_k$, and then encrypts the i-th text symbol x_i as $y_i = e_{i \bmod m}(x_i)$.

Main strategy: find the period, then reduce the problem to a simple substitution cryptogram.

Last time we observed that repeated strings in the cryptogram give information about the period. This idea is due to Kasiski.

The Index of Coincidence

The main weakness of Kasiski's analysis is that it relies on rare events: long repeated strings. William Friedman, in the 1920's, invented a statistical technique based on repetitions of individual symbols.

The main idea: repeated characters are more likely in text encrypted with a small period than in random text. This gives a way to estimate the period, and in particular, determine if it is greater than 1.

IC for monoalphabets

Consider the cryptogram $y_1 y_2 \cdots y_n$. Suppose there are N possible symbols, which we will call $0, 1, \ldots, N-1$. Let there be f_i occurrences of symbol i. The total number of $i < j$ with $y_i = y_j$ is

$$f_0(f_0 - 1)/2 + \cdots + f_{N-1}(f_{N-1} - 1)/2.$$

(To see this, we first observe that the number of pairs (i, j) with $1 \le i < j \le m$ is $\binom{m}{2}$. Now rearrange the cryptogram so that all the 0's come first, then all the 1's, and so on. This will not change the number of matching positions. So there are $\binom{f_0}{2}$ pairs of symbol 0, and similarly for the other symbols.) On the other hand, we have $n(n-1)/2$ pairs of symbols all told.

The ratio is called the *index of coincidence*:

$$\mathrm{IC} = \frac{\sum_i f_i(f_i - 1)/2}{n(n-1)/2} = \frac{\sum_i f_i(f_i - 1)}{n(n-1)}$$

Let's consider an example. Let the text be

A B R A C A D A B R A

We have $f_A = 5$, $f_B = 2$, $f_C = f_D = 1$, and $f_R = 2$. Also $n = 11$. So

$$\mathrm{IC} = \frac{20 + 2 + 0 + 0 + 2}{11 \cdot 10} = 0.218.$$

Note that this is a statistic: it depends on your data (the cryptogram).

(Terminology isn't standard here. We are following Sinkov, but Kullback uses $\phi = \sum_i f_i(f_i - 1)$.)

What is the expected value of IC? Let's assume that the i-th symbol occurs with probability p_i, and that as $n \to \infty$, we have $f_i \to np_i$. Then

$$E[\text{IC}] \to \frac{\sum np_i(np_i - 1)}{n(n-1)} \sim \sum_i p_i^2.$$

Previously, we called this κ. It is a characteristic of the source language. It can be shown that $\kappa \geq 1/N$, with equality iff the symbol frequencies are uniform (adapt Sinkov p. 66). For $N = 26$, we have for the uniform distribution

$$\kappa = 0.038$$

whereas for English text

$$\kappa = 0.066$$

IC for period 2

We'll assume that each encryption function is chosen so that the i-th position in the cryptogram, considered by itself, has a uniform distribution on symbols. This is true for most of the common ciphers (shift, affine, simple substitution, etc.)

Consider a long cryptogram of length n. There will be $n/2$ symbols encrypted with e_1, and $n/2$ symbols encrypted with e_2. $E[\text{IC}]$ is the probability that two randomly chosen positions in the cryptogram match. There are

$$\sim \frac{(n/2)^2}{2}$$

pairs encrypted by e_1, and for these the probability of a match is $\sim \sum p_i^2$. Call this κ_S (S for "source"). Similarly, we have

$$\sim \frac{(n/2)^2}{2}$$

pairs encrypted by e_2 with match probability κ_S. Finally, there are

$$\sim \left(\frac{n}{2}\right)^2$$

pairs encrypted differently. For each of these the probability of a match is $\kappa_R = 1/N$ (R for "random"). Putting all this together we get

$$E[\text{IC}] \sim \frac{(n/2)^2\kappa_S + (n/2)^2\kappa_R}{n^2/2} = \frac{\kappa_S + \kappa_R}{2}.$$

An estimator of the period

We can apply similar reasoning for a polyalphabetic cipher of period m. We get

$$E[\mathrm{IC}] \sim \frac{1}{m}\kappa_S + \frac{m-1}{m}\kappa_R.$$

Here are some values of this for small m, using $\kappa_S = 0.038$

$m = 1$	0.066
$m = 2$	0.052
$m = 3$	0.047
$m = 4$	0.045
$m = 5$	0.044
...	...
limit	0.038

These are asymptotic values, valid as $n \to \infty$.

If the source text symbols are pairwise independent, we can compute the expected value exactly. It is

$$E[\mathrm{IC}] = \frac{n-m}{m(n-1)}\kappa_S + \frac{(m-1)n}{m(n-1)}\kappa_R. \qquad (*)$$

Note that this is a convex combination of κ_S and κ_R.

An exact formula for the variance appears in the books by Kullback and Konheim. Roughly speaking, however, since we expect observed frequencies to be accurate to relative error $O(n^{1/2})$, the same will be true of the coincidence index.

Suppose we have an observed value $\hat{\mathrm{IC}}$ for the coincidence index. Setting this equal to the right hand side of (*) and solving for m, we get an estimator for the period:

$$\hat{m} = \frac{n(\kappa_S - \kappa_R)}{(n-1)\hat{\mathrm{IC}} - n\kappa_R + \kappa_S}$$

Note that as $n \to \infty$ we have

$$\hat{m} \sim \frac{\kappa_S - \kappa_R}{\mathrm{IC} - \kappa_R}.$$

Since IC is close to κ_R for large m, this tells us we are going to have problems for large m.

We have used a quick and dirty idea from statistics called the *method of moments*. You assume various things equal their expected values, and solve the resulting equations to obtain the parameter you want to estimate. Usually this leads to biased estimates. [It should be possible to figure out how bad (or good) the

estimator $\hat{m}$ is, and think about better alternatives. I have not seen any discussion of this.]

All of this suggests that $\hat{m}$ is a crude tool at best, and we should think about better ways to use the data. A more powerful technique will be presented next time.

Notes.

In the literature, the definition of IC can be modified depending on the author's favorite normalization. For example, one can subtract a null hypothesis (random text) value and divide by the standard deviation.

The exact expected value of IC can be calculated as follows. First, the number of occurrences of symbol i in a text of length n has a binomial distribution with parameters n and p. (This assumes text symbols are independent and identically distributed with the underlying plain language probabilities.) By a standard formula for this distribution, $E[f_i(f_i - 1)] = np_i^2$. Summing over all i, we get

$$E[\text{IC}] = \sum p_i^2.$$

The crude period estimator $\hat{m}$ has one virtue: it can be used for aperiodic ciphers as well. That is, if we select from a repertoire of m substitutions in a pseudo-random fashion, and there is no long-term bias in our selection process, $\hat{m}$ will converge to m.

IC is related to the GINI score used in machine learning. (Basically the same thing, when $n = 2$.)

For some information on machines that were built in the 1940's to do IC tests and other cryptanalytic tasks, see Colin Burke, Information and Secrecy, Scarecrow Press, 1994.

Lecture 13

This lecture: Accurate period tests, cryptanalysis of Vigenère and related systems.

Period Tests

The first step in polyalphabetic cryptanalysis is to learn the period. We have seen two ways to do this:

Kasiski: Factor distances between repeated substrings.

Coincidence index: Compute a statistic whose expected value depends on the period.

It is not hard to see what the problems with these methods are. The Kasiski method uses only a small part of the data, and fails utterly if there are no repeats. The coincidence index does use all the data, and has expected value close to

$$\frac{1}{m}\kappa_S + \frac{m-1}{m}\kappa_R$$

which we can think of as $1/m$ part "signal" coming from identically enciphered pairs, and $(m-1)/m$ parts "noise" coming from pairs whose encryptions have nothing to do with each other. Clearly what is needed is a way to listen only to the signal, and ignore the noise.

Friedman derived the following sharper test for period length. Let the cryptogram be

$$y_1 y_2 \cdots y_n.$$

To test the hypothesis that the period is m, compute the coincidence index separately on each group of letters that are distance m apart:

$$\begin{aligned} \mathrm{IC}_1 &= \mathrm{IC}(y_1, y_{m+1}, y_{2m+1}, \ldots) \\ \mathrm{IC}_2 &= \mathrm{IC}(y_2, y_{m+2}, y_{2m+2}, \ldots) \\ &\cdots \\ \mathrm{IC}_m &= \mathrm{IC}(y_m, y_{2m}, y_{3m}, \ldots) \end{aligned}$$

and then average these values. If we choose the m that maximizes this average, this is an estimate for the period.

Reference: W. Friedman, The Index of Coincidence and its Applications in Cryptanalysis, p. 10

It is worthwhile to think about how accurate this is. Since each group is encrypted the same way, we will have (roughly)

$$\mathrm{IC}_i = \sum p_i^2 + O(\sqrt{m/n}).$$

On the other hand, we are averaging m quantities, which reduces the error by the factor $1/\sqrt{m}$. (We are assuming here that the averaged quantities are independent.) So the fluctuation in the average is

$$\frac{1}{\sqrt{m}} O(\sqrt{m/n}) = O(n^{-1/2}).$$

Since this does not depend on m, we expect the results to be reliable over a much wider range of periods.

Vigenère Cryptanalysis

Let us consider a Vigenère-like system in which the message is encrypted monoalphabetically, then shifted using a key word.

For this system, the key is a permutation σ of $\mathbf{Z}_N$, plus m shifts $k_1, \ldots, k_m$. If the message is

$$x_1 x_2 \ldots x_n$$

the cryptogram

$$y_1 y_2 \ldots y_n$$

has $y_i = \sigma(x_i) + k_{i \bmod m}$.

Classic Vigenère takes σ to be the identity.

We note that there is some ambiguity in the key. The shifts are determined only up to a common additive constant.

Using previous methods, we can assume the period m is known.

Using correlation, we can determine the relative shifts $k_i - k_j$. Once this is done we can work with the simple substitution cryptogram

$$z_1 z_2 \ldots z_n$$

where $z_i = y_i + (k_1 - k_{i \bmod m})$. It should be noted that the substitution for this is not σ but σ followed by a shift of k_1.

To determine relative shifts, we put the cryptogram into m columns, and compute frequency distributions for each column. Suppose that for columns i and j we have the frequencies

$$P = (p_0, \ldots, p_{N-1})$$

and

$$Q = (q_0, \ldots, q_{N-1})$$

Find the value of d that maximizes

$$\sum_i p_i q_{i-d}.$$

This tells us that column j is shifted up d letters relative to column i, so that $k_j = k_i + d$. We can now combine the text from these two columns into one and continue.

Various strategies for the combining order seem possible. One idea would be to compute correlations for all pairs of columns, and combine the pair giving the strongest "signal" (a correlation closest to $\sum p_i^2$). Sinkov gives examples showing that this will work when simply taking the columns in order will not. [Is there any theory to tell us what is the optimal order?]

Notes

Some other polyalphabetic systems using shifts.

1. Vigenère followed by a substitution. That is, we have

$$y_i = \sigma(x_i + k_{i \bmod m}).$$

The classical solution method involves *symmetry of position*, an idea due to Kerckhoffs. A by-hand example of this is given in J. R. Wolfe, Secret Writing: The Craft of the Cryptographer, p. 130 ff. [There should be a statistical method for this one too. Can differential cryptanalysis be put to use?]

2. Choose two permutations τ and σ and encrypt by

$$y_i = \sigma(\tau(x_i) + k_{i \bmod m}).$$

Coincidence counts can be used to solve these, if all shifts are represented. [What if there are only a few shifts?] For an "aperiodic" version of this, see Deavours and Kruh's discussion of the Kryha machine.

3. Jefferson wheel cipher (see Beker and Piper section 2.2). Choose m permutations $\sigma_1, \sigma_2, \ldots, \sigma_m$. Then

$$y_i = \sigma_i(\sigma_i^{-1}(x_i) + k),$$

where the indices for σ are taken mod m. Mathematically, this is a fixed shift conjugated by σ_i.

4. Japanese RED machine (half rotor):

$$y_i = \sigma(x_i) + k_i,$$

where k_i is periodic. Deavours and Kruh explain this one.

5. From D. Crystal, The Cambridge Encyclopedia of Language, p. 67: "Yule's characteristic (K) measures the chance that two nouns chosen at random will be identical." This is basically an IC at the word level, and has been used in statistical studies of authorship questions. It is probably not coincidental that one of Friedman's first jobs involved trying to prove that someone other than Shakespeare wrote Shakespeare. (See D. Kahn, The Codebreakers, 1967, pp. 370 ff.)

Possible references: G. Udny Yule, The Statistical Study of Literary Vocabulary, Cambridge, 1944; G. Herdan, An Inequality between Yule's Characteristic K and Shannon's Entropy H, Z. f. Angewandte Math. u. Phys 9 (1958), 69-73.

6. The FFT can be put to use in detecting the period of a polyalphabetic encryption. For some discussion of this, see pp. 82-92 of F.-P. Heider, D. Kraus, and M. Welschenbach, Mathematische Methoden der Kryptoanalyse, Vieweg u. Sohn, Braunschweig, 1985.

Lecture 14

We will spend the next few lectures discussing some ideas of Claude Shannon.

Key reference: C. Shannon, Communication Theory of Secrecy Systems, Bell Systems Technical Journal, 1949. This paper summarizes Shannon's World War II work on cryptography.

Additional references: Chapter 2 of Stinson; Dominic Welsh, Codes and Cryptography.

Entropy

It is useful to have a way to measure the nonuniformity of a probability distribution. We will need a symmetric nonlinear function of the p_i's. (It can't be linear because the p_i must sum to 1.)

Definition: If $\{p_i\}$ is a probability distribution, its *entropy* is

$$H(\{p_i\}) = -\sum_i p_i \log p_i.$$

The name entropy comes from statistical mechanics, in which similar quantities appear.

We can think of the entropy as measuring the uncertainty in a random choice whose probabilities are given by p_i.

Usually $\log = \log_2$, and we speak of a choice from p_i as containing H bits of information.

Examples.

The uniform distribution on $\{1, \ldots, N\}$ has $p_i = 1/N$. This makes

$$H = \log N.$$

Considering our usual example of $N = 26$, we have $\log_2 26 = 4.700440$. We therefore expect to represent letters using 5 binary digits. This is indeed possible, one such code is obtained by assigning the k-th letter the binary representation of k. Thus

$$\texttt{A} = 00000$$
$$\texttt{B} = 00001$$
$$\ldots$$
$$\texttt{Z} = 11010$$

Consider the English letter frequencies. From Kullback's data we have

$$p_A = 3683/50000$$
$$p_B = 487/50000$$
$$\ldots$$
$$p_Z = 49/50000$$

This gives $H = 4.167$ bits per letter.

Degenerate distributions. Suppose that $p_0 = 1$ and all other $p_i = 0$. If we agree that

$$0 \log 0 = \lim_{x \to 0} x \log x = 0,$$

then $H = 0$. This agrees with our intuition that there is no uncertainty in a choice whose result is predetermined.

It can be shown that for distributions on $\{1, \ldots, N\}$, we have $0 \leq H \leq \log N$. The upper bound is an equality only for the uniform distribution, and the lower bound is an equality only when all the probability is concentrated on a single point.

The entropy function $H(p_1, \ldots, p_n)$ is convex.

The easiest way to show this is to use calculus. We consider H as a function of N positive variables (with no constraint on their sum). Since

$$H = -\sum p_i \log p_i$$

we have

$$\frac{\partial^2 H}{\partial p_i^2} = -1/p_i$$

whereas

$$\frac{\partial^2 H}{\partial p_i p_j} = 0$$

if $i \neq j$. This makes the quadratic function

$$\begin{pmatrix} x_1 & \cdots & x_N \end{pmatrix} \left(-\frac{\partial^2 H}{\partial p_i p_j} \right) \begin{pmatrix} x_1 \\ \cdots \\ x_N \end{pmatrix}$$

positive definite, and by a standard theorem (see W. Fleming, Functions of Several Variables) H is convex. Now, if we have two probability distributions $\{p_i\}$ and $\{q_i\}$ and we take a convex combination, say

$$r_i = \alpha p_i + \beta q_i$$

with $\alpha + \beta = 1$, $0 \leq \alpha, \beta \leq 1$, then we get a new probability distribution. So H is convex even when restricted to probability distributions.

As a consequence, we should expect the letter frequencies of a polyalphabetically encrypted text to be more random (or at least no less random!) than the frequencies from a monoalphabetic substitution.

Here's a justification for that. Suppose that $m = 2$ and that the symbol frequencies, after encryption, are $\{p_i\}$ and $\{q_i\}$. The combined frequencies will be $(p_i + q_i)/2$. By convexity,

$$H\left(\frac{p_i + q_i}{2}\right) \geq \frac{H(p_i) + H(q_i)}{2} \geq H(p_i)$$

since p_i and q_i are the same probabilities in different order.

As an experiment, I tried polyalphabetic substitutions on a 300-character text, with the following results:

$$\begin{aligned} m = 1 \quad & H = 4.172 \\ m = 2 \quad & H = 4.503 \\ m = 3 \quad & H = 4.554 \\ m = 4 \quad & H = 4.599 \\ m = 5 \quad & H = 4.602 \\ m = 10 \quad & H = 4.673 \end{aligned}$$

Entropy and data compression

We can also think of entropy as measuring the "compressibility" of a sequence of choices from a distribution.

Example. Suppose we have 5 letters, `A,B,C,D,E`. Let

$$p_A = 0.05, p_B = 0.05, p_C = 0.10, p_D = 0.30, p_E = 0.50.$$

We have

$$H(0.05, 0.05, 0.10, 0.30, 0.50) = 1.78454...$$

A straightforward encoding of these symbols as 000,001, etc. would use 3 bits per symbol. We can do much better on average if we adopt a variable length code, such as

$$\texttt{A} = 0000, \texttt{B} = 0001, \texttt{C} = 001, \texttt{D} = 01, \texttt{E} = 1.$$

(Note that no code word is a prefix of any other, so that it is clear how to parse any given string.) The expected length of a symbol encoded this way is

$$4 \times 0.05 + 4 \times 0.05 + 3 \times 0.10 + 2 \times 0.30 + 1 \times 0.50 = 1.8$$

which is close to H. If we were willing to use a more clever code and introduce strings not just for individual letters, but for commonly occuring blocks of letters, we could get as close as we like to H. [For a proof of this, see Cover and Thomas, Elements of Information Theory, pp. 53-54.]

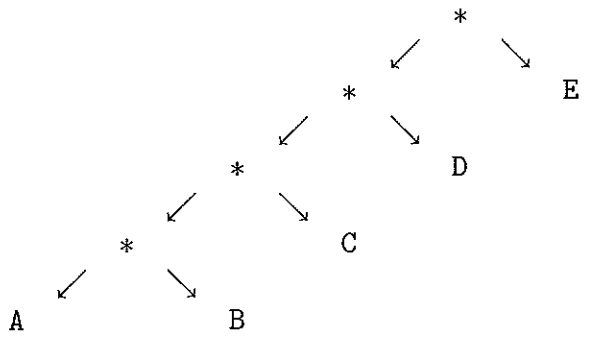
There is a clever method for constructing good codes for a set of symbols whose frequencies are known. This was invented by David Huffman in the 1950's.

Let the symbol probabilities be $p_1, p_2, \ldots, p_n$. Choose the smallest two probabilities, say p_i and p_j, and replace the i-th and j-th symbols by a new symbol, with probability $p_i + p_j$. Repeat this until only one symbol remains.

This process can be thought of as making a binary tree. The leaves are the original symbols, and the act of replacing symbols x and y with the new symbol

z gives x and y the parent z. Once the tree is built, the path from the root to a leaf determines the code for that leaf, if we write 0 for a left move and 1 for a right move.

To verify you understand this, you should try it on the five-letter alphabet we used above. You should obtain the second (more efficient) code that was described. The resulting tree is



It can be shown that Huffman encoding minimizes the expected encoding length, over all codes of the above type that are based on binary trees. Codes based on binary trees in this manner often called *prefix-free*, since in a tree, no path to a leaf goes through another leaf, thereby guaranteeing that no code word is a prefix of any other.

Entropy of languages

Suppose $X_1, X_2, \ldots$ are random variables, not necessarily independent. If the limit below exists, we assign this process the entropy rate

$$H = \lim_{n \to \infty} \frac{H(X_1, \ldots, X_n)}{n}$$

A sufficient condition for the limit to exist is that the process be stationary, which means (roughly) that its stochastic characteristics are invariant under time shifts.

Experiments with English text show that it can be compressed. Can we speak meaningfully of its entropy?

This is hard to measure (Welsh has some references to work in the area), but very roughly, we know that English text has an entropy rate around 2 bits per symbol.

Redundancy

Let there be N possible choices for the X_i. In some contexts it is useful to define the redundancy R to be

$$R = \log N - H.$$

We can also make this a fraction:

$$r = 1 - \frac{H}{\log N}.$$

Note that we always have $0 \leq r \leq 1$.

The data compression utility `gzip` compressed a 3700 character text file to 1700 characters. Since $1700/3500 = 0.46$, we should think of this text as being at least 54% redundant. This gives us an estimate for entropy. We have

$$1 - \frac{H}{\log N} \geq 0.54$$

so

$$H \leq 0.46 \log_2 26 = 2.1622...$$

Notes

Nice use of entropy calculations: you have a passphrase that is hashed down to m bits. How long should it be? Schneier (AC, p. 174) does an analysis of this.

(From a talk by J. Rosenthal, 2003) Suppose you are a cryptanalyst looking at some ciphertext. How would you distinguish this from random text? One way is to try to compress it. If it has lower entropy than $\log N$ bits per symbol, an algorithm like LZ would detect this.

Huffman encoding or some other data compression technique is often used as a precursor to encryption. Not only does this make the ciphertexts shorter, but it makes the cryptanalyst's job harder. (There should be some theory to explain this.)

Lecture 15

We are now ready to discuss Shannon's information-theory approach to cryptography. The distinguishing feature of this approach is that everything in sight is a random variable.

Cryptosystems

P is the message

K is the key

C is the cryptogram

We'll assume that i) P, K are independent; ii) C is determined by P and K; iii) P is determined by C and K.

Sometimes we will want to consider a family of systems (e.g. Vigenère with a fixed period and varying message length). When this is done we use P_n and C_n to indicate messages and cryptograms with n symbols.

Entropy

$$H(p_1, \ldots, p_n) = -\sum_i p_i \log p_i$$

Key Equivocation

Let

$$\begin{aligned} P_n = x_1 \ldots x_n &= \text{ message of length } n \\ K &= \text{ key} \\ C_n = y_1 \ldots y_n &= \text{ cryptogram of length } n \end{aligned}$$

The *key equivocation* is

$$E_n = H(P_n) + H(K) - H(C_n)$$

We will think of this as the average uncertainty about the key that remains after the cryptanalyst has seen n symbols of encrypted text.

Note that $n = 0$ corresponds to maximum uncertainty, we have $E_0 = H(K)$.

Technical note: Shannon's actual definition was $E_n = H(K|C_n)$. It is equivalent to ours under our assumptions i)–iii).

Intuitively, we believe that $E_n \to 0$ for any cipher with a fixed key length. That is, as the cryptogram becomes longer and longer, eventually there will be only one choice of key consistent with cryptogram. (It should be possible to prove this rigorously.)

In practice, the difficulty in using this formula is in computing $H(C_n)$. The others are easy to evaluate.

If keys are chosen uniformly, $H(K) = \log |K|$. See Lecture 8 where this is computed for various systems.

For "reasonable" plaintext models $H(P_n)$ will equal n times the entropy rate.

Despite the difficulty of computing $H(C_n)$, it is possible to get estimates.

Note that

$$H(C_n) \leq n \log N$$

and therefore if keys are chosen uniformly and $R = \log N - H$

$$\begin{aligned} E_n &= H(K) + H(P_n) - H(C_n) \\ &\geq H(K) + H(P_n) - n \log N \\ &= \log |K| + n(\log N - R) - n \log N \\ &= \log |K| - nR \end{aligned}$$

The lower bound becomes 0 at $n = \log |K|/R$. We will call this the *nominal unicity point.*

Technical quibble: Many authors define the unicity point to be "the value of n at which $E_n = 0$." This is not a good thing to do because E_n is never exactly 0, just asymptotic to 0. Shannon modeled ciphers using random graphs and concluded that after the nominal unicity point, E_n falls off exponentially. He concluded that for most ciphers, "we can speak, with little ambiguity, of a point after which decryption is unique."

One way to think of the nominal unicity point is that it gives a length n beyond which we have no reason to suppose the cipher secure.

Some computations

Biased random plaintext on a two-symbol alphabet.

Suppose $p_0 = p$ and $p_1 = q = 1 - p$. Let there be two substitutions, σ (which does nothing) and τ (which interchanges 0 and 1), chosen with equal probability.

This is one of the few situations in which we can compute everything exactly. Suppose that $y_1 \ldots y_n$ is a string with m 1's and $n - m$ 0's. The probability of obtaining this cryptogram is $(p^m q^{n-m} + p^{n-m} q^m)/2$, so

$$\begin{aligned} H(C_n) &= -\sum_{y \in \{0,1\}^n} \Pr[y] \log \Pr[y] \\ &= -\sum_{m=0}^{n} \binom{n}{m} \frac{p^m q^{n-m} + p^{n-m} q^m}{2} \log \left(\frac{p^m q^{n-m} + p^{n-m} q^m}{2} \right) \end{aligned}$$

We also have

$$H(K) = 1$$

and

$$H(P_n) = nH(p, q) = -n(p \log p + q \log q)$$

It is possible to plug this all into a computer and evaluate the key equivocation exactly. Here are some values for $p = 0.25$.

$$\begin{aligned} E_0 &= 1.0000 \\ E_1 &= 0.8113 \\ E_2 &= 0.6681 \\ E_5 &= 0.3855 \\ E_{10} &= 0.1632 \\ E_{20} &= 0.0320 \\ E_{50} &= 0.0003 \end{aligned}$$

The nominal unicity point is

$$U = \frac{H(K)}{R} = \frac{1}{1 + p \log p + q \log q} = 5.3.$$

Monoalphabetic substitution

Let's compute this for a monoalphabetic substitution on 26 symbols. Assume the plaintext is English, with $H = 2$ bits/symbol, and therefore $R = 4.7 - 2 = 2.7$. Therefore the nominal unicity point is

$$U = \frac{\log |K|}{R} = \frac{\log_2 26!}{2.7} = \frac{88.4}{2.7} = 32.7$$

Shannon noted that ciphers "with 30 letters almost always have unique solutions, whereas with 20 characters it is easy to find several."

Vigenère

Suppose $N = 26$ as usual, and the period is m. The key space has size 26^m, so for English (with $R = 2.7$)

$$U = \frac{m \log_2 26}{2.7} = 1.74m$$

[Is there a way to compute or estimate E_n?]

Lecture 16

Continuing with Shannon's theory of cryptography. As before:

Cryptosystems

P is the message

K is the key

C is the cryptogram

We assume: i) P, K are independent; ii) C is determined by P and K; iii) P is determined by C and K.

The subscript n indicates message length.

Entropy

$$H(p_1, \ldots, p_n) = -\sum_i p_i \log p_i$$

Perfect Secrecy

The system has *perfect secrecy* if P and C are independent.

Intuitively, this means that the cryptanalyst can learn nothing about the message by observing a cryptogram.

In more general situations where i)–iii) don't hold, this should be replaced by the condition $H(P|C) = H(P)$. See, e.g. Welsh's book for a definition and explanation of conditional entropy.

The following theorem is useful. If the system has perfect security, then there are least as many keys as possible plaintexts. (In this context "possible" means "has positive probability.")

Proof: We model the system as a labelled bipartite graph, with an edge labelled k connecting x to y if $e_k(x) = y$. Suppose key 1 maps P_1 to C_1. Some other key must map P_2 to C_1 (if not, we would know that P_2 is impossible upon observing C_1), and the same for $P_3, P_4, \ldots$. All of these keys must be distinct, since decryption is deterministic. This gives a 1-1 mapping from possible plaintexts to keys, and the result follows.

Some examples

Shift ciphers. Suppose the key distribution is uniform and the plaintext x is one letter. We compute

$$\begin{aligned}
\Pr[x = a, y = b] &= \Pr[x = a, k = b - a] \\
&= \Pr[x = a]\Pr[k = b - a] \\
&= \Pr[x = a] \cdot N^{-1} \\
&= \Pr[x = a]\Pr[y = b]
\end{aligned}$$

since the cryptanalyst observes a randomly chosen letter. On the other hand, this is not true if the message has more than one letter, since there are now more than N plaintexts but only N keys.

Affine ciphers and monoalphabetic substitutions have perfect security when $n = 1$ but not for $n > 1$. The reasoning is similar.

The Vigenère system has perfect security when $n \leq m$ but not otherwise.

It would be an interesting exercise to consider the Hill cipher.

Shannon also considered various ways in which to combine cryptosystems.

One way is to use one system with probability p and another with probability q. This adds one bit of key.

Another idea is composition. Suppose S and T are two systems with the same message space. We could imagine applying S then T. (Note that this composition is not commutative.) This corresponds to a classical idea called *superencryption*. Note that the key is now an ordered pair of keys, one for S and another for T.

Modern block ciphers like DES get their strength from composing many weaker systems. In fact, we can think of DES as a composition of 16 different systems, differing in how they use the (global) key.

It's important to note that composition need not increase security. For example, if we compose two shift ciphers the result is another shift cipher with the new key equal to the sum mod N of the old keys. Shannon called such systems S (i.e. with the property that S composed with itself is the same as S) *idempotent*.

Summary

Practical experience shows that cryptanalysis is easier when the cryptogram is long. Put another way, a cryptanalyst who can observe as much ciphertext as needed will eventually win. This is the primary reason why real systems require frequent changes of key.

We can think about Shannon's two ideas (perfect secrecy and unicity) as giving upper and lower bounds on the message length for which a given system is worth using.

Most systems will have perfect secrecy for $n \leq L$. If the system is well designed L is approximately the key length. If we send messages shorter than L, we have overbuilt the system, in the sense that they key (and therefore the complexity of using the system) is larger than needed.

The nominal unicity point U gives a value of n at which the system should no longer be considered secure. If we send message longer than U, we run the risk of successful cryptanalysis.

[*** What is L here? Key entropy? ***]

Notes

There is a subtle point which has to be emphasized here. Perfect secrecy is not just a property of the system but of the probability distributions on messages, keys, etc. To take an extreme example, if you have a monoalphabet but the only possible messages are those composed of one symbol, it is still perfectly secure. But not for more general distributions.

Lecture 17

Stream Ciphers and the One-Time Pad

So far we have considered cryptosystems that apply the same transformation to successive characters, or to successive blocks of characters.

Many practically important systems are *stream ciphers*. These take a message

$$x_1 x_2 x_3 \ldots$$

and a pseudo-random *key stream*

$$k_1 k_2 k_3 \ldots$$

and produce the encrypted message

$$y_1 y_2 y_3 \ldots$$

where $y_i = x_i + k_i \bmod N$.

For most practical systems N is 2 or 26.

Since we are allowed a new k_i at each time step, this is a fairly general method. It is possible, though, for k_i to depend on x_i or even on x_j for $j < i$. Systems of this kind can be described via the formalism of of finite automata with output.

Most of the stream ciphers we will study are *synchronous*, in the sense that they generate the same key stream no matter what the message.

Synchronous stream ciphers require a global clock.

The one-time pad

The ultimate in stream ciphers is to choose each k_i uniformly at random from $\mathbf{Z}_N$. This is called the *one-time pad*.

This was invented in the 1920's, independently, by Vernam and Mauborgne in the U.S. ($N = 2$) and by Kunze, Schauffer, and Langlotz in Germany ($N = 10$)

Some people call this the Vernam system. This is, strictly speaking, not historically accurate. Vernam's original device used a long loop of paper tape to specify the key bits, and this loop repeated. It was soon realized that adding the bits on two tapes (of lengths ℓ and $\ell + 1$ gave a much longer period. [This can be cryptanalyzed – see Konheim or Meyer/Matyas.] Mauborgne had the essential idea of making each key bit random.

The one-time pad offers perfect secrecy.

We note that

$$\begin{aligned}\Pr[y_i = b_i, i = 1, \ldots, n] &= \Pr[x_i + k_i = b_i, i = 1, \ldots, n] \\ &= \Pr[k_i = b_i - x_i, i = 1, \ldots, n] \\ &= \prod_{i=1}^{n} \Pr[k_i = b_i - x_i] \\ &= 1/N^n\end{aligned}$$

so the cryptanalyst sees uniformly distributed random characters.

If we observe that a one-time pad is equivalent to a Vigenère system whose key is longer than the message, we find that the one-time pad has perfect secrecy. Let us verify this. (We abbreviate x by $x_1 \ldots x_n$, and similarly for y, a, b, and k.) Using the previous result, and the independence of x and k, we have

$$\begin{aligned}\Pr[x = a, y = b] &= \Pr[x = a, k = b - a] \\ &= \Pr[x = a]\Pr[k = b - a] \\ &= \Pr[x = a]N^{-n} \\ &= \Pr[x = a]\Pr[y = b]\end{aligned}$$

Keys cannot be re-used.

Suppose that

$$y = y_1 y_2 y_3 \ldots$$

with $y_i = x_i + k_i$, and

$$z = z_1 z_2 z_3 \ldots$$

with $z_i = w_i + k_i$. Subtracting these, we cancel out the key and obtain

$$y - z = (x_1 - w_1)(x_2 - w_2)(x_3 - w_3)\ldots$$

This is equivalent to a "book cipher" where one text is used to encrypt another in a Vigenère-like manner. Methods are known to crack these. [This should be fleshed out. See W. Friedman, Riverbank Publication No. 16; Kahn, The Codebreakers, p. 375; Beker and Piper, p. 295.]

Despite the practical difficulties (the parties must meet and agree on key material beforehand) this system has been used for high-security applications since the 1930's.

Coupled with letter-to-number transcription schemes, the one-time pad was a favorite of the Soviets during the cold war. See D. Kahn, The Che Guevara Cipher, in Kahn on Codes, pp. 139-145.

One-time pads were reportedly used for the Moscow-Washington hot line [reference?] but they have now been replaced by a conventional cipher.

Pseudo-Random stream ciphers

The technical difficulties with one-time pads lead one to suspect that pseudo-random keys can be used in place of the random key stream. This is commonly done but difficult to get right. In particular, most of the common pseudo-random number generators lead to systems that are woefully insecure.

Suppose we have a system that takes a message

$$x_1 x_2 x_3 \ldots$$

and adds a pseudo-random key stream

$$k_1 k_2 k_3 \ldots$$

to obtain

$$y_1 y_2 y_3 \ldots$$

where $y_i = x_i + k_i \bmod N$.

In practice, large pieces of the message, particularly at the beginning, will be known or suspected. For example, one of my e-mail messages began with the line

```
From milewski@kleene.math.wisc.edu Mon Mar 22 15:32:09 1999
```

An opponent would know the date, the approximate time, and the names of people likely to send me this message.

Knowing a part of the message, the corresponding key can be obtained by subtraction.

Thus, any stream cipher for which the key stream can be predicted (forwards or backwards) from a small piece of key cannot be considered secure. In the next few lectures we will show how this prediction can be done for some common random number generators.

Notes

For $N \neq 2$, one can arrange things so that encryption and decryption are identical. Take

$$y_i = k_i - x_i,$$

and note that

$$k_i - y_i = k_i - (k_i - x_i) = x_i.$$

Traditionally, this system (or more precisely, its finite-period version) is called Beaufort. It is a good idea because it allows the same algorithm to be used in both directions.

Potential re-use of key material in an additive stream cipher led to a weakness in WEP encryption for wireless networks. [Reference?]

See F. Piper, Recent Developments in Cryptography, in J. K. Skwirzynski, ed., Performance Limits in Communication Theory and Practice, Kluwer 1988, pp. 207-223 for some interesting remarks on stream ciphers. He makes the point that stream ciphers that do not propagate errors are suitable for "noisy" channels, e.g. that carry digitized speech.

Stream Ciphers from Iterated Affine Maps

The commonest type of pseudo-random number for computer use is D. H. Lehmer's "multiplicative congruential" generator. For example, it is the algorithm behind the Unix system's `random()` routine, and many, many others. In this lecture, we'll study its suitability as a stream cipher.

The multiplicative congruential generator works as follows. Choose a modulus N and two parameters $a, b \in \mathbf{Z}_N$. The *seed* is an initial value k_0, and we generate further k_i, for $i \geq 1$, by the iteration

$$k_i = ak_{i-1} + b \pmod{N}.$$

As usual, once we have the key stream $k_0, k_1, k_2, \ldots$, we can encrypt the i-th ciphertext symbol x_i as

$$y_i = x_i + k_i,$$

and decrypt by solving this equation for x_i.

In a cryptographic setting, we can think of a, b, N as parameters of the system, and the seed k_0 as a session key.

An Example

Let's take $N = 26$, with parameters $a = 3$ and $b = 5$.

We took $a \in \mathbf{Z}_N^*$ to make the mapping reversible.

With this choice of parameters, there are four cycles of length 6, and one of length 2:

0	5	20	13	18	7	(repeats)
1	8	3	14	21	16	(repeats)
2	11	12	15	24	25	(repeats)
6	3	22	19	10	9	(repeats)
4	17	(repeats)				

Our system, as an additive stream cipher, is thus equivalent to a Vigenère system with period 6 or 2. Note that there are two seeds, 4 and 17, that give a short period.

As an exercise, you can use a computer to check that the plaintext message

```
ANYONE CONTEMPLATING ARITHMETICAL METHODS
                  OF GENERATING RANDOM DIGITS
               IS OF COURSE IN A STATE OF SIN
```

encrypts, with seed $k_0 = 0$, to

```
ASSBFL CTHGWTPQUGAUG FLVLOMJNVUHL RYGZVDX
                  IS YLNJLNLPNL LNFKOR XVYPTX
               CF GM CTOEKL IS U FLHTJ IS KPN
```

Maximal-period sequences

We can always choose parameters for the Lehmer generator so as to guarantee a period of N. This is clearly the longest possible, since the state is an element of $\mathbf{Z}_N$.

One such choice is $a = b = 1$. We'll see more interesting choices shortly.

We quote the following theorem without proof.

Let the prime factorization of N be

$$N = p_1^{e_1} \cdots p_r^{e_r}.$$

A sequence $\{k_i\}$ defined by the Lehmer multiplicative congruential method has period N if and only if, for every i, we have: i) b is relatively prime to p_i; and ii) p_i divides $a - 1$ (when p_i is odd), or 4 divides $a - 1$ (when $p_i = 2$).

This was proved by Hall and Dobell in the early 1960's.

Predicting multiplicative congruential sequences.

By knowing or guessing some plaintext, we can reduce the cryptanalysis of our system to the following question. Suppose that N is known, and we have a sequence

$$k_0, k_1, \ldots, k_{t-1}$$

generated by the recurrence $k_i = ak_{i-1} + b$. What are a and b?

To solve this, we define the *difference sequence*

$$d_i = k_i - k_{i-1}.$$

Then, from

$$k_i = ak_{i-1} + b$$
$$k_{i-1} = ak_{i-2} + b$$

we see that

$$d_i = a(k_{i-1} - k_{i-2}) = ad_{i-1}.$$

If $d_{i-1} \in \mathbf{Z}_N^*$ – and this seems likely – we can solve this to obtain

$$a = d_i d_{i-1}^{-1}.$$

Once a is known, we can determine the other parameter from the equation

$$b = k_i - ak_{i-1}.$$

In the "maximum period" case, the differences will always be invertible mod N. To see this, first prove by induction that

$$k_i = a^i k_0 + (a^{i-1} + \cdots + a^2 + a + 1)b,$$

and subtract the same equation for $i-1$ to obtain

$$\begin{aligned} d_i = k_i - k_{i-1} &= a^{i-1}((a-1)k_0 + b) \\ &= a^{i-1}((a-1)k_0 + b). \end{aligned}$$

However, for any p dividing N, we have $a-1 \equiv 0 \bmod p$, so $\gcd((a-1)k_0+b, N) = \gcd(b, N) = 1$ since $b \in \mathbf{Z}_N^*$. This proves that $d_i \in \mathbf{Z}_N^*$ for all i.

Notes.

The standard reference for pseudo-random number generation is vol. 2 or D. E. Knuth, The Art of Computer Programming.

There is also an efficient algorithm to predict this generator when the modulus N is unknown. For this, see J. Boyar, J. ACM 1989.

Lecture 19

Predicting the a/p generator

To further emphasize the idea that long periods do not guarantee security, we'll discuss a stream cipher based on decimal arithmetic. As far as I know, this was never used in practice, but it nicely illustrates the difference between apparent randomness and true unpredicability.

For the general class of system we are discussing, the plaintext is a sequence

$$x_1 x_2 x_3 \ldots$$

of elements from $\mathbf{Z}_N$. We make a pseudo-random key stream

$$k_1 k_2 k_3 \ldots$$

and let the ciphertext be

$$y_1 y_2 y_3 \ldots$$

where $y_i = x_i + k_i$. Decryption uses the reverse process: $x_i = y_i - k_i$.

In this lecture, we'll consider a method for generating the keystream based on the decimal expansion of a rational number. Therefore, we take $N = 10$.

What is the a/p generator?

We consider a simple example.

Let $N = 10$. Choose a fraction a/p with p prime and expand a/p into a decimal fraction. For example, $a = 10$ and $p = 29$ leads to

$$10/29 = 0.34482\ 75862\ 06896\ 55172\ 41379\ 310\ldots$$

where the 28 numeral pattern repeats.

Gauss [Disquistiones Arithmeticae, §312-8] proved that when $\gcd(a, p) = 1$ and $p \neq 2, 5$, the period is the least positive ν such that $10^\nu = 1$ in $\mathbf{Z}_p^*$.

This gives us a way to get very large periods, possibly as large as $p - 1$, from a seed (a, p) of only $2 \log p$ bits. Since

$$p - 1 \sim p = (1.414...)^{2 \log_2 p},$$

the period is potentially exponentially large in the key.

In practice, we can restrict the search to divisors of $p - 1$. To show this we need an auxiliary result.

Fermat's little theorem: If p is prime then $a^p \equiv a \bmod p$. This can be easily shown using the binomial theorem and induction on a. Indeed, it holds for $a = 0, 1$. Now consider

$$(a+1)^p = a^p + \sum_{i=1}^{p-1} \binom{p}{i} a^{p-i} + 1$$

Since $\binom{p}{i} = p!/(i!(p-i)!)$ and p is a prime, these binomial coefficients all vanish mod p. So

$$(a+1)^p \equiv a^p + 1$$
$$a+1$$

by induction.

Now, suppose that $p \neq 2, 5$. Then $10 \in \mathbf{Z}_p^*$, so

$$10^{p-1} \equiv 1 \bmod p$$

The minimum $\nu > 0$ making $10^\nu \equiv 1 \bmod p$ must be a divisor of $p-1$. To prove this, write $p - 1 = \nu q + r$ with $0 \leq r < p - 1$. Then

$$1 = 10^{p-1} = 10^{q\nu + r} \equiv 10^r,$$

which forces $r = 0$. So ν divides $p - 1$.

For our example, $p - 1 = 28$, so the period could be 1,2,4,7,14,28. We compute

$$10^{28/2} = 10^{14} \equiv 28 \bmod 29$$

$$10^{28/7} = 10^4 \equiv 24 \bmod 29$$

so the period cannot be any divisor of 28 or 24, so it must be $\mathrm{lcm}(4, 14) = 28$.

It is an interesting question as to whether there are infinitely many primes p for which this random number generator has period $p - 1$. This is a form of Artin's conjecture.

Using continued fractions to predict the a/p generator.

Recall that we considered using the digits of

$$10/29 = 0.34482\ 75862\ 06896\ 55172\ 41379\ 310\ldots$$

as an additive stream cipher with $N = 10$.

The continued fraction algorithm takes as input a positive real number x and produces a sequence of integers $a_0, a_1, a_2, \ldots$ making

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cdots}}.$$

The algorithm starts with $i = 0$ and then computes

$$a_i = \lfloor x_i \rfloor, x_{i+1} = \frac{1}{x_i - a_i}$$

for $i = 2, 3, 4, \ldots$.

For irrational x, the algorithm can go on forever. If x is rational, some x_i for $i \geq 1$ will vanish, and the algorithm terminates.

Let's try this on our example.

We must of course use a finite approximation to x. Displaying six figures, we have

$$\begin{aligned} x_0 &= 0.344828 & a_0 &= 0; \\ x_1 &= (0.344828)^{-1} = 2.900064 & a_1 &= 2; \\ x_1 &= (0.900064)^{-1} = 1.111032 & a_2 &= 1; \\ x_1 &= (0.111032)^{-1} = 9.006384 & a_3 &= 9. \end{aligned}$$

The successive finite continued fractions are called the *convergents*. Let's compute these.

$$\begin{aligned} 0 &= 0.000000; \\ 0 + 1/2 &= 1/2 = 0.500000; \\ 0 + \frac{1}{2 + 1/1} &= 1/3 = 0.333333; \\ 0 + \frac{1}{2 + \frac{1}{1+1/9}} &= 10/29 = 0.344827.... \end{aligned}$$

We note that very quickly, the decimal expansion of the fraction has told us the numerator and the denominator. It can be shown that about $2 \log_2 p$ bits of the key stream is sufficient for this purpose.

The next device we will discuss is the *linear feedback shift register* (LFSR). This was intensively studied in the 1950's as a way to easily generate long random-looking sequences. As we will see, these sequences have many nice pseudo-random properties but do not stand up to cryptanalysis. We will make two main points.

1. Cryptographers have different requirements for random number generators than do experimenters (who typically run Monte Carlo experiments).

2. LFSR's are still useful in the design of stream ciphers, but they must be thought of as a building block, not the whole building.

The standard reference for LFSR's is S. Golomb, Shift Register Sequences, Aegean Park Press.

Notes.

This generator comes from a paper by Blum, Blum, and Shub. It was never proposed seriously as a stream cipher, but makes a good way to illustrate that even quite random-looking sequences can be efficiently predicted.

Encodings with N equal to 10 or a power of 10 were commonly used, because of the convenience of decimal arithmetic. One system used base 100, and goes as follows.

Consider `I` and `J` to be the same plaintext symbol, so as to make 25 letters in all. Then, allow $0, 25, 50, 75$ to represent plaintext `A`, allow $1, 26, 51, 76$ to represent plaintext `B`, and so on. Systems of this type, in which a plaintext symbol can have more than one encryption, are called *homophonic* in old-fashioned literature.

Lecture 20

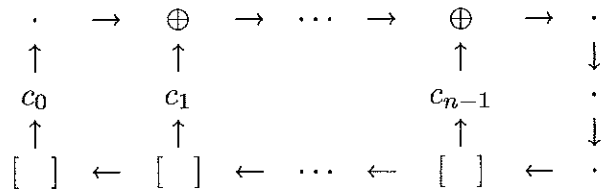
Today's lecture introduces linear feedback shift registers. These are pseudo-random number generators which, although not cryptographically secure by themselves, form a key ingredient in stream cipher design.

What are feedback shift registers?

A feedback shift register is composed of n *cells*, each of which can hold one bit. The device is clocked, and at each time step, the bit in a cell is copied into the cell immediately to its left. The rightmost cell is given a value that is some function of the previous cell contents.

For the moment we restrict attention to devices with linear feedback. These are specified by giving n feedback coefficients $c_0, \ldots, c_{n-1} \in \{0, 1\}$. A coefficient of 1 indicates that the value from that cell is used; all such values are added together mod 2 and the result is written in the rightmost cell.

Here's a picture:



In diagrams of this type, a feedback coefficient of 1 indicates a presence of a wire, and 0 the absence of a wire.

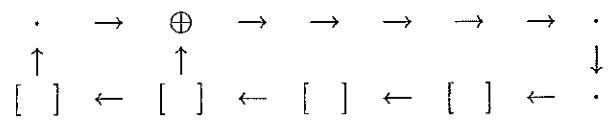
If the initial cell contents are $x_0, x_1, \ldots, x_{n-1}$, then we can define further x_i by

$$\begin{aligned} x_n &= c_0 x_0 + \cdots + c_{n-1} x_{n-1} \\ x_{n+1} &= c_0 x_1 + \cdots + c_{n-1} x_n \\ x_{n+2} &= c_0 x_2 + \cdots + c_{n-1} x_{n+1} \end{aligned}$$

and so on. Note that x_i is the bit in the leftmost cell at time i.

An example

Let's consider $n = 4$, with $c_0 = c_1 = 1$ and $c_2 = c_3 = 0$. The device is



If we load the cells initially with 1 1 1 1, the following behavior results:

1	1	1	1
1	1	1	0
1	1	0	0
1	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	1
0	0	1	1
0	1	1	0
1	1	0	1
1	0	1	0
0	1	0	1
1	0	1	1
0	1	1	1

This behavior is periodic with period $15 = 2^4 - 1$

For any shift register with linear feedback, 00...0 is a fixed point. Hence the maximum possible period is $2^n - 1$, which is attained in this case. We will give an algebraic criterion for this.

Polynomials with Coefficients in $\mathbf{Z}_2$

It is possible to define polynomials whose coefficients lie in $\mathbf{Z}_2$ and manipulate them more or less as in ordinary algebra.

Example:

$$(X^2 + X + 1)(X + 1) = X^3 + 2X^2 + 2X + 1 = X^3 + 1$$

Note that we have dropped any term whose coefficient is even, since $1 + 1 = 0$ in $\mathbf{Z}_2$.

Remainder theorem: if a, b are polynomials with coefficients in $\mathbf{Z}_2$, there are polynomials q, r such that

$$a = bq + r$$

and

$$\deg(r) < \deg q$$

The polynomials q and r can be computed by long division. For example, if

$$a = X^3 + X^2 + 1, b = X^2 + 1$$

then

$$X^3 + X^2 + 1 = X(X^2 + 1) + X^2 + X + 1$$
$$X^2 + X + 1 = 1(X^2 + 1) + X$$

So we have $q = X + 1$ and $r = X$.

Full-period LFSR's

If $c_0, \ldots, c_{n-1}$ are the feedback coefficients for a linear feedback shift register, then

$$f = c_0 + c_1 X + c_2 X^2 + \cdots + c_{n-1} X^{n-1} + X^n$$

is called the *characteristic polynomial.*

If $c_0 \neq 0$, there is some $\nu \geq 1$ for which $X^\nu - 1$ is a multiple of f. The least such ν is called the *order of X mod f.*

Let's prove this. We consider the remainders of X^i after division by f. They cannot all be distinct. Hence we have $1 \leq i < j$ making

$$X^i = q_i f + r$$
$$X^j = q_j f + r$$

Subtract these to get

$$(X^{j-i} - 1) X^i = X^j - X^i = (q_i - q_j) f$$

Now multiply both sides by $(X^{n-1} + \cdots + c_1)^i$ and obtain

$$(X^{j-i} - 1)(X^n + c_{n-1} X^{n-1} + \cdots + c_1 X)^i = sf,$$

for some s, which implies (since $c_0 = 1$)

$$(X^{j-i} - 1)(f - 1)^i = sf.$$

Expanding $(f-1)^i$ by the binomial theorem, and moving terms divisible by f to the right hand side, we find that

$$(X^{j-i} - 1) = s' f$$

for some polynomial s'. This proves the result.

Maximal period criterion: A LFSR with characteristic polynomial f, started in a state that is not all zeros, has maximum possible period iff the order of X mod f is $2^n - 1$.

We call f *primitive* if this happens. It can be shown that there are primitive polynomials of every degree, in fact, $\varphi(2^n - 1)/n$ of them.

Let's check this on our example. If $c_0 = c_1 = 1$ and $c_2 = c_3 = 0$, the characteristic polynomial is

$$X^4 + X + 1.$$

We have

$$X^{15} - 1 = (X^3 - 1)(X^8 + X^4 + X^2 + X + 1)(X^4 + X + 1)$$

so the order must divide 15. (Why must it divide 15? This should be explained.) We note that

$$X^3 - 1 \text{ can't be a multiple of } X^4 + X + 1$$

whereas

$$X^5 - 1 = X(X^4 + X + 1) + X^2 + X + 1$$

is not a multiple of $X^4 + X + 1$ either. So the order is exactly $15 = 2^4 - 1$, and as we have seen the sequence has maximum period.

Randomness properties

LFSR sequences were intensively studied in the 1950's. A good reference for this work is S. Golomb, Shift Register Sequences.

Consider the contents of a particular register over a full period, say (for our example)

1 1 1 1 0 0 0 1 0 0 1 1 0 1 0

Golomb proved

The number of 1's differs by 1 from the number of 0's. (Our example has 8 1's and 7 0's)

Among the runs (maximum consective subwords) of 1's, there are 2^{n-3} of length 1, 2^{n-2} of length 2, ..., 1 of length $n-2$ and 1 of length n. Note that we count runs that wrap around. (Our example had 2 length 1 runs, 1 of length 2, and 1 of length 4.)

The autocorrelation, which is by definition

$$C(t) = \sum (-1)^{x_i} (-1)^{x_{i+t}}$$

takes on the values $2^n - 1$ when $t = 0$ and -1 if $t \neq 0$. (Compare our example sequence with its shift one place to the left.

1 1 1 1 0 0 0 1 0 0 1 1 0 1 0
1 1 1 0 0 0 1 0 0 1 1 0 1 0 1

There are 7 agreements and 8 disagreements, giving $C = -1$ as advertised.)

LFSR's as stream ciphers

Because LFSR sequences do a good job of mimicking properties of random sequences, it is tempting to use them in place of a one-time pad.

As we will see, this is *not* secure.

Notes

LFSR stream cipher encryption has been proposed as a way to enhance performance of digital communication. See B. G. Lee and S. C. Kim, Scrambling Techniques for Digital Transmission, Springer-Verlag, 1994.

Rabin-Karp string matching algorithm (see CLR 34.2) is relevant here.

Nice implementation trick (from a 10/12/00 sci.crypt post by Terry Ritter): values themselves do not move; move a pointer around a queue to simulate shifting. Work per update is proportional to the number of taps. This is also in Knuth, vol. 2 somewhere.

It should be remarked that autocorrelation is essentially the same operation as coincidence counting, since we score $+1$ for an agreement and -1 for a disagreement.

The maximum length criterion for linear shift registers is easy to prove, using the following trick. Think of the state as a row vector, which evolves by the map $s \mapsto sA$. The transpose of A gives multiplication by X, mod the characteristic polynomial of the device. However, A and A^T are similar, so the period of $s \mapsto sA$ is the same is the period of $s \mapsto sA^T$.

For a nice (non-cryptographic) application of shift registers, see M. C. Valenti and J. Sun, Turbo Codes, Chapter 12 of F. Dowla, Handbook of RF and Wireless Technologies, Newnes.

For LFSR's applied to fault testing in logic circuits, see pp. 514-517 of M.L. Bushnell and V.D. Agrawal, Essentials of Electronic Testing for Digital, Memory and Mixed-signal VLSI Circuits, Kluwer. Possibly also R. David, Signature analysis of multi-output circuits, in Proc. of the International Fault-Tolerant Computing Symp. June 1984, pp. 366-371. (References suggested by DongHyun Baik.)

Lecture 21

Topic du jour: Cryptanalysis of an LFSR-generated stream cipher.

LFSR stream ciphers

We generate a key stream of elements in $\mathbf{Z}_2$ as follows. Choose n initial values $x_0, x_1, \ldots, x_{n-1}$, and then generate further values by the recurrence relation

$$x_m = c_{n-1}x_{m-1} + c_{n-2}x_{m-2} + \cdots + c_0 x_{m-n}.$$

These are added mod 2 to the message bits, in the usual way, to obtain a ciphertext.

The key stream has a long period (if the c_i are properly chosen it is $2^n - 1$) and passes various statistical tests for randomness. Nevertheless, the system is vulnerable to a known plaintext attack.

LFSR cryptanalysis via linear algebra

Suppose the cryptanalyst knows or can guess m bits of the message. Subtracting this mod 2 from the cryptogram provides m consecutive bits of the key stream. We may as well assume these are the first m bits.

Mathematically, we have the following problem: Given $x_0, \ldots, x_m$ coming from a linear feedback shift register, for what value of m can we predict the rest of the sequence?

Let's consider using linear equations. Considering the coefficients as unknowns, we have

$$\begin{pmatrix} x_0 & x_1 & \cdots & x_{n-1} \\ x_1 & x_2 & \cdots & x_n \\ x_2 & x_3 & \cdots & x_{n+1} \\ & \vdots & & \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} x_n \\ x_{n+1} \\ x_{n+2} \\ \vdots \end{pmatrix}$$

If we collect enough equations, we guess that there will be only one solution for the c_i's. How many equations are enough?

A good way to think about problems like this is to imagine that the matrix entries are random. (Of course they aren't but this will get us started.)

If there are n equations, then Jordan's formula for $p = 2$ tells us that

$$\begin{aligned} \Pr[\text{ unique solution }] &= (1 - 1/2^n)(1 - 1/2^{n-1}) \cdots (1 - 1/2) \\ &\geq \prod_{i \geq 1} (1 - 1/2^i) \\ &= 0.288788... \end{aligned}$$

(Jordan's formula applies because this is the probability that the matrix is invertible.)

We need $2n$ entries to make n equations, and our heuristic model suggests there is a good chance we can recover the c_i in this case.

Maximum length sequences are actually better than random for this purpose. This is because any n consecutive blocks

$$\begin{array}{ccc} x_i & \cdots & x_{i+n-1} \\ x_{i+1} & \cdots & x_{i+n} \\ & \vdots & \\ x_{i+n-1} & \cdots & x_{i+2n-2} \end{array}$$

are linearly independent.

Proof: We choose the orgin of the cycle so that at time 0 the register contents are all 0 except for a 1 at the right. The contents at times $0, 1, \ldots, n-1$ are

$$\begin{array}{cccccc} 0 & \ldots & 0 & 0 & 0 & 1 \\ 0 & \ldots & 0 & 0 & 1 & * \\ 0 & \ldots & 0 & 1 & * & * \\ 0 & \ldots & 1 & * & * & * \\ & \cdots & & & & \\ 1 & \ldots & * & * & * & * \end{array}$$

and these are evidently independent. Interpreting this array as the coefficients of a matrix M, we can multiply M on the left by

$$S = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ & & & \ddots & \\ 0 & 0 & 0 & \cdots & 1 \\ c_0 & c_1 & c_2 & \cdots & c_{n-1} \end{pmatrix}$$

and get the register contents at times $1, \ldots, n$. Since S is nonsingular these n blocks must form an invertible matrix. Continue in like fashion to get the same property for any group of n consecutive blocks.

Comments

This result is very nice from an information theory point of view. It takes $2n$ bits (n initial register contents, plus n feedback coefficients) to specify the device. So we don't expect to use fewer bits to predict it, and the theorem says $2n$ is exactly right.

We can solve for the c_i using any $2n$ consecutive bits of the LFSR sequence.

Lecture 22

Topic du jour: The Berlekamp-Massey algorithm

This is an extremely efficient method for determining the linear feedback shift register that produces a given sequence. It also plays a role in decoding of certain error-correcting codes.

Reference: J. Massey, IEEE Trans. Inform. Theory, 1969.

Connection polynomials

For this lecture we will change our point of view somewhat and consider a variation of the characteristic polynomial we have defined.

Suppose a sequence x_i satisfies the recurrence relation

$$x_n + c_1 x_{n-1} + \ldots + c_k x_{n-k} = 0$$

for all $n \geq k$. Then we call

$$f = 1 + c_1 Z + \cdots + c_{k-1} Z^{k-1} + c_k Z^k$$

a *connection polynomial* for the sequence. Note that if the sequence is generated by an LFSR with characteristic polynomial ϕ then the reverse of ϕ is a connection polynomial.

It is important to understand that a connection polynomial can have $c_k = 0$. We will refer to k as the *length* of f.

Suppose f is a polynomial as above. We say it is correct at position n if either

$$x_n + c_1 x_{n-1} + \ldots + c_k x_{n-k} = 0,$$

or if $n < k$. We will call f correct before n if it is correct at positions $0, 1, \ldots, n-1$.

The discrepancy of f at position n is

$$x_n + c_1 x_{n-1} + \ldots + c_k x_{n-k}.$$

When $n < k$ we will also say that the discrepancy is 0.

Splicing connection polynomials

Suppose we have a guess for the connection polynomial of some LFSR. We can use it to predict sequence values until it fails, at which time we should replace it by another polynomial. This is easy to do if we have remembered a previous polynomial that we have used for this purpose.

Splicing Theorem. Suppose f is correct before, but not at, position n, and g is correct before, but not at, position m, with $m < n$. Then the polynomial

$$f + Z^{n-m}g$$

is correct for all positions $\leq n$.

Proof: Consider the discrepancy of $f + Z^{n-m}g$. This is zero before position n (because f works before position n and g works before position m), and since the discrepancies for f and g cancel, it is also zero at position n.

We will now work with two connection polynomials f and g. The first represents our best guess for the connection polynomial of the sequence, and the second represents a previous guess.

We'll think of these as anchored at particular points of the sequence.

g is anchored at the last place it failed to make a correct prediction. To get started, we take $g = 1$, anchored at position -1 of the sequence.

f also starts out at position -1. The basic step of the algorithm slides f forward one position, and computes the discrepancy between f's prediction and the sequence value at that position. If the discrepancy is zero, we do nothing and continue. If the discrepancy is 1, we apply the splicing theorem to f and g to make a new f, which is good for predicting the sequence up to and including the current position.

The trick to the algorithm is to find a good strategy for updating the previous guess g.

The basic idea is to keep g's trailing edge as far along as possible. Intuitively, this will keep the degree of f small.

Suppose we have applied the theorem. There are two cases.

If the length of f has increased, we should reset g to be the old value of f.

If the length of f has not increased, we should keep g the same.

Suppose that f, before the splicing step, has length F. In his paper, Massey proves that the length of f increases iff $\max(F, n+1-F) > F$, that is, iff $n \geq 2F$. We can use this as a test, rather than keeping track of the lengths of f and g.

Massey also proves (we won't do it here) that this algorithm finds the minimal-degree connection polynomial possible at every stage.

Implementing the Berlekamp-Massey algorithm

Here is pseudocode for the procedure we outlined above.

To get started, let

$$f = 1, \qquad F = 0, \qquad n = -1.$$

This means that our first connection polynomial is 1, its length (formal degree) is 0, and it is anchored at position -1. Similarly, for the previous connection polynomial, we define

$$g = 1, \qquad G = 0, \qquad m = -1.$$

We also set $H = -1$ initially.

Now we make a loop that reads bits ad infinitum

repeat (forever)
    $n := n + 1$
    if f has nonzero discrepancy at at n:
        $h := f + Z^{n-m} \cdot g$
        $H := \max(H, G + n - m)$.
        if $(H > F)$ [length increased]:
            $g := f$
            $G := F$
            $m := n$
        $f := h$
        $F := H$

Notes

The Berlekamp-Massey algorithm can be thought of as a procedure that solves a special kind of linear system. For this reason, it has antecedents in the literature. See K. S. McCurley, Odds and Ends from Cryptology and Computational Number Theory, in Proc. Symp. Appl. Math., vol. 42, 1990, pp. 145-166.

The algorithm can be generalized to linear recurrences modulo composite numbers. See N. J. A. Sloane and J. A. Reeds, Shift-register synthesis (modulo m), SIAM J. Computing, v. 14, 1985, pp. 505-513.

Lecture 23

In this lecture, we'll survey the basic theory of nonlinear shift register devices.

References: S. Golomb, Shift Register Sequences (Aegean Park Press); article by R. Rueppel in Simmons, Contemporary Cryptology.

What we know about linear shift registers

Mathematically, a LFSR produces a sequence of bits

$$x_0, x_1, \ldots$$

where for all $i \geq n$, $x_i = \sum_{j=1}^{n} c_j x_{i-j} \bmod 2$.

Maximum period LFSR's have good pseudo-randomness properties

They are not cryptographically secure for use as additive stream ciphers. LFSR sequences can be predicted using linear algebra mod 2, or more quickly by the Berlekamp-Massey algorithm.

Linear complexity

Let $x_0, \ldots, x_{m-1}$ be a sequence of bits. The *linear complexity* of the sequence is the smallest n such that some linear feedback shift register with n cells generates the sequence.

If we can't do anything else, we can always take a LFSR with m cells and load it up with $x_0, \ldots, x_{m-1}$. So the linear complexity is always defined.

Linear complexity gives us a way to talk about the "complicatedness" of a finite sequence.

For an infinite sequence, we can define its *linear complexity profile* to be the function

$$C(m) = \text{linear complexity of } x_0, \ldots, x_{m-1}.$$

High linear complexity is a necessary (but not sufficient) condition for a pseudo-random stream to be secure. What can be achieved?

Always, $C(m) \leq m$.

Niederreiter proved that a random 0/1 sequence has, almost surely,

$$C(m) \sim m/2.$$

This is in some sense reasonable since there are $2n$ bits of information in the description of an n-cell LFSR (initial contents plus tap positions).

Devices with non-linear feedback

In some sense, the insecurity of LFSR sequences derives from the linearity of the function that derives the next bit from the previous ones. It stands to reason, then, we might get better sequences if we eliminate the linearity.

Mathematically, we choose initial bits $x_0, \ldots, x_{n-1}$ and then let

$$\begin{aligned} x_n &= F(x_0, \ldots, x_{n-1}) \\ x_{n+1} &= F(x_1, \ldots, x_n) \\ x_{n+2} &= F(x_2, \ldots, x_{n+1}) \end{aligned}$$

where F is some n-input 1-output Boolean function. This generalizes the usual LFSR, in which F is a linear function.

Examples.

Consider a machine with $n = 2$ and feedback function $F(x_0, x_1) = x_0 \vee x_1$. The digraph has two connected components. We see that the states 00 and 11 are fixed points. (More generally, a machine in which F is the Boolean or has the two fixed points 0...0 and 1...1. A fixed point will be reached in at most $n + 1$ steps.)

Another example is Stephen Wolfram's "Rule 30" device. This has $n = 3$ and $F(x_0, x_2, x_2) = x_0 + x_1 + x_2 + x_1 x_2$.

State graphs

A nonlinear shift register can be modeled by a directed graph which has vertices in $\mathbf{Z}_2{}^n$ and an edge $x \to y$ if the machine with x in the register will move to the state with y in the register.

The directed graph can be seen to be of the following form. There are one or more cycles, into which feed tails. For cryptographic purposes we would like these cycles to be long.

Cycle length heuristics

Nonlinear shift registers are very hard to analyze, so we resort to heuristic models, based on random functions.

Suppose that $F : \mathbf{Z}_2{}^n \to \mathbf{Z}_2{}^n$ is a random function. If we choose a random starting point, the expected cycle length is $O(2^{n/2})$. This surprising result is related to the "birthday problem" which we will discuss later.

You should observe that this is much less than the cycle length of a maximum-period LFSR, which is $2^n - 1$.

One problem with this model is that the "next state" function F is constrained to respect the shift register property. However, a more realistic result is known. Suppose the feedback function F is randomly chosen, uniformly from the 2^{2^n} possiblities. If the initial state is random, the mean time until a state repeats is $\sim \sqrt{\pi} 2^{n/2}$. [E.D. Karnin, J. Appl. Prob., v. 20, 413-418, 1983.]

Reversible shift registers

For the purpose of getting long periods, it would clearly be desirable for the state graph to be all cycles, that is, one in which each vertex has in-degree 1 and out-degree 1. Let us call such a shift register *reversible*. The next result gives a criterion for reversibility.

Golomb's Criterion: A Boolean function F produces a reversible shift register iff it is of the form

$$F = x_0 + G(x_1, \ldots, x_{n-1}).$$

Here + denotes addition mod 2 as usual.

Proof: Suppose

$$x_n = x_0 + G(x_1, \ldots, x_{n-1}).$$

Then we also have

$$x_0 = x_n + G(x_1, \ldots, x_{n-1}).$$

so the state $x_1 \ldots x_n$ has exactly one predecessor. Conversely, suppose this is true. Then $1x_1 \ldots x_{n-1}$ and $0x_1 \ldots x_{n-1}$ have different successors. THis implies that

$$F(1, x_1, \ldots, x_{n-1}) = 1 + F(0, x_1, \ldots, x_{n-1})$$

We consider this as a function of x_0, holding the other bits fixed. The only functions with this property are of the form $x_0 + c$, and we may take $c = F(0, x_1, \ldots, x_{n-1})$.

Cycle length heuristics for reversible machines

As we've seen, a nonlinear shift register with no special structure should be expected to repeat after a time equal to the square root of the number of states. On the other hand, if the device is reversible, the expected cycle length is 2^{n-1}, which is generally much larger.

We can model such a device by by a random permutation on the set of states. Suppose there are $N = 2^n$ states. The expected time until a state repeats can be found by the following "card dealing" argument. Take a pack of N cards, shuffle it, and then think of one card, say the ace of spades. Deal out the shuffled pack until this card appears. Probabilistically, this is the same as choosing a random state in our device and evolving it until that state appears again. It follows that the expected cycle length is uniformly distributed, and has, in particular, a mean length of $N/2 = 2^{n-1}$.

All of this suggests that choosing F to make a reversible shift register, which we can easily do by applying the theorem, is a good idea.

Notes

For a great paper on cycle lengths, see B. Harris, Probability distributions related to random mappings, Annals of Math. Stat., 1960. (Also in Selected Mathematical Papers, National Security Agency, Washington, D.C. 1964.)

More to the point may be A. Benczur, On the expected time of the first occurrence of every k bit long patterns in the symmetric Bernoulli process, Acta Math. Hung. 47 (1986), 233-238.

Floyd's cycle detection method will find the period of a sequence using a very small amount of memory.

H. Fredricksen, A Survey of Full Length Nonlinear Shift Register Cycle Algorithms, SIAM Review, v. 24, 1982.

E. S. Selmer, "From the Memoirs of a Norwegian Cryptologist," Proc. EUROCRYPT 93 [= LNCS 765].

S. W. Golomb, On the Cryptanalysis of Nonlinear Sequences, Proc. IMA – Cryptography and Coding 1999 [= LNCS 1746], pp. 236-242.

Lecture 24

Stream Ciphers Incorporating Nonlinearity

In this lecture, we'll study some examples of stream ciphers that combine linear with nonlinear techniques. One basic idea is to feed one or more linearly generated sequences into a nonlinear combining function, to make a key stream. Another is to use one sequence to control the rate at which others are generated.

Mechanically implemented shift register cryptography

Feedforward devices actually go back to the 1920's, but these were implemented mechanically rather than electrically. Here is one example.

Hagelin machines

Boris Hagelin was a Swedish inventor, who founded the company now called Crypto AG (Zurich). His designs were marvels of mechanical ingenuity, and were used by armies for field ciphers at least into the 1950's.

We'll give functional descriptions, imagining devices to be built out digital hardware. The goal of a Hagelin machine is to produce a pseudo-random stream of elements from $\mathbf{Z}_{26}$. This sequence is used as a key stream in a Beaufort-type system. That is, plaintext x encrypts as

$$y = k - 1 - x,$$

and decryption is given by the same function.

The basic element in a Hagelin system is the ring counter. This is an m-stage shift register with trivial feedback. For example, the states in a ring counter with $m = 4$ might be

$$0011, 0110, 1100, 1001,$$

and so on. Output is taken from a fixed position in the ring counter, so that the above example, if tapped at the leftmost bit, would make the output

$$0, 0, 1, 1, \cdots$$

Mechanically, ring counters were implemented by pinwheels, similar to those in music boxes.

Hagelin's initial design (the C-35) had five ring counters, with cycle lengths

$$17, 19, 21, 23, 25.$$

These numbers are relatively prime, so a period of

$$17 \cdot 19 \cdot 21 \cdot 23 \cdot 25 = 3.9 \times 10^6$$

is attainable. The initial load formed the key, for which there are

$$2^{17+19+21+23+25} = 2^{105} = 10^{35}$$

possible values. (It should not escape notice that the key length is comparable to today's block ciphers, such as 128-bit AES.) If the output of the i-th ring counter is x_i, then the key is

$$k = \sum_{i=1}^{5} a_i x_i \bmod 26,$$

with coefficients

$$a_1 = 1,\ a_2 = 2,\ a_3 = 4,\ a_4 = 8,\ a_5 = 10.$$

In this design, the key is a linear function of the x_i's which makes the cipher weak. For example, the parity of any seventeed successive key values gives away the settings for the ring of size 17. We can obtain more ring settings if we observe that

$$\begin{aligned} 0 \le k \le 9 &\Rightarrow \text{ only one choice for the } x_i\text{'s} \\ 10 \le k \le 15 &\Rightarrow \text{ two choices} \\ 16 \le k \le 25 &\Rightarrow \text{ one choice} \end{aligned}$$

That is, for a fraction $20/26 = 77\%$ of the keys, the value of k gives away the internal settings of the machine.

The next version (US Army M-209?) incorporated some nonlinearity. It has six ring counters, with lengths

$$17, 19, 21, 23, 25, 26.$$

In addition to the initial ring counter loads, the user can choose elements of a 27×6 array A of bits. (For mechanical reasons, A was constrained to have at most two 1 bits per row – it was implemented using a lug cage.) If we define, using Boolean matrix multiplication,

$$y = Ax,$$

then the key k is

$$k = [\text{number of nonzero bits in } y] \bmod 26.$$

That is, if A_i is the i-th column of A, then we have

$$k = \left(\text{number of 1's in } \bigvee_{i=1}^{6} A_i x_i \right) \bmod 26.$$

Note that the only non-linear part of this is the Boolean OR.

The combined period of the ring counters is now

$$26 \cdot 25 \cdot 23 \cdot 21 \cdot 19 \cdot 17 = 1.3 \times 10^8,$$

and the number of keys has been estimated (by Beker and Piper) to be about 6×10^{52}. [I don't know if this is a rigorous calculation.]

Because of the nonlinearity, cryptanalysis is more difficult, but still possible. The basic idea is to focus on one counter at a time. For example, to determine the contents of the size 17 ring, we observe the key stream at every 17th time unit. In the literature, success has been reported with a key stream sample of $10^2 - 10^3$ symbols. See Beker and Piper, Chapter 2, for more information on this.

Electronic feedforward machines

An obvious improvement to Hagelin-type machines is to replace the wheels by LFSR's of longer period. One can imagine this was done as soon as electrical shift register technology became available, although the details await a historian with access to the proper archives. Here are two ideas that appear in the literature.

Product sequences.

Suppose that u_i and v_i are two bit sequences generated by maximum-length LFSR's. Define a new bit sequence k_i by

$$k_i = u_i \cdot v_i.$$

The combining operation can be thought of as Boolean AND, so the key stream will be, roughly, 75 % 0's and 25% 1's. This violates our intuitive idea that the keystream look as if it were chosen randomly.

The periodic behavior and linear complexity of this generator has been analyzed by Selmer (Norway) and Zierler (US).

Can we use another combining function? Unfortunately, the only balanced 2-input 1-output Boolean functions are

$$u + v, \sim (u + v), u, \sim u, v, \sim v.$$

Thus, all we can do with 2 LFSR's is to make another linear device.

Geffe's generator.

A *multiplexer* is a Boolean function with n address inputs and 2^n data inputs. The address is used to select one of the data inputs which becomes the output.

In 1973, Geffe published a design based on using 3 LFSR's with relatively prime periods, together with a multiplexer with $n = 2$. The output of one of the LFSR's selects one of the other ones, whose output is then copied to the key stream.

If the LFSR's have lengths d_1, d_2, d_3, the linear complexity of Geffe's sequence is $d_1d_2 + d_3d_2 + d_2$. (Here d_2 is the length of the "selector", and the "selectees" have lengths d_1 and d_3.)

It is possible to analyze the Geffe generator (with known LFSR's) using the following idea. Let's call a 3-variable Boolean function *correlation immune* if, when the 3 inputs are chosen from the uniform distribution (on 2^3 choices), each output is statistically independent of the output. As it turns out, the multiplexer with $n = 1$ is not correlation immune. This means that we can focus on each shift register independently. When $d_i \leq n$, successful analysis using about $40n$ bits has been reported in the literature.

Many other variants on this idea, replacing multiplexers with other functions, have been published.

Clock-controlled devices

The Lorenz machine

This was a device used by the German army (?) in WWII. The See Ken Halton, The Tunny machine, in Hinsley and Stripp, Codebreakers: The Inside Story of Bletchley Park.

As with the Hagelin machines, the device contained a number of wheels, which we will think of as shift registers with trivial feedback.

Plaintext was encoded using the 5-bit Baudot code, which was commonly used for automatic telegraphy between the wars. The device produced a pseudo-random stream of 5-bit blocks, which was added mod 2 to the plaintext.

The machine has two banks of shift registers:

$$F_1 - F_5 \qquad \text{with 41, 31, 29, 26, 23 cells}$$

and

$$S_1 - S_5 \qquad \text{with 43, 47, 51, 53, 59 cells}$$

The i-th bit of an enciphered block is

$$m_i + f_i + s_i \bmod 2$$

where f_i is extracted from F_i and similarly for s_i.

From our point of view this machine is interesting because it introduced the idea of using one shift register to produce a signal that controls another. The F_i's are free and rotate once per clock cycle. The S_i's are controlled by a signal derived from two master registers M_1 and M_2. M_1 has 61 cells. A bit m_1 is extracted from this, and if it is 1, the 37-cell register M_2 advances one step. Another bit m_2 extracted from M_2 controls the "slaves" $S_1 - S_6$.

There should be an example or two of clock control via digital hardware.

Notes

Many of the Hagelin machines are described in C. A. Deavours and L. Kruh, Machine Cryptography and Modern Cryptanalysis, Artech House, 1985. A statistical solution method for the M-209 appears in H. Beker and F. Piper, Cipher Systems, Northwood, 1982.

For an early electromechanical device (built out of devices equivalent to today's "ring counters") see R. Doyle, The US Navy's first online crypto system, IEEE Annals of the History of Computing, January-March 2001, pp. 17–21.

For Geffe's design, see P. R. Geffe, How to Protect Data with Ciphers that are Really Hard to Break, Electronics, Jan. 4, 1973, pp. 99-101. (This article contains a number of nice ideas, and was far ahead of what was commonly known about cryptography in its time.)

Jennings (?) analyzed pseudo-random sequences generated by extracting $\leq n$ output bits from an n-cell LFSR, and $\leq 2^n$ output bits from a 2^n-cell LFSR, and feeding these into a multiplexer to produce an output.

For correlation immunity and its applications to cryptography, see the IEEE-IT papers by Sigenthaler (1984, 1985).

Lecture 25

This lecture begins the study of modern block ciphers.

Block ciphers came into public view with the US Government's call in the 1970's for a standard cipher to be used for "sensitive but unclassified data." At the time, very few groups had the requisite expertise, and the winner was a design by IBM Research, which later became known as DES.

The most important block cipher after DES is AES, which was selected as a replacement for DES. We will cover both algorithms in detail.

Block ciphers in general

Mathematically, a block cipher is just a particular kind of combinatorial function.

It takes as inputs m symbols and a key k, and produces m output symbols.

It must give a 1-1 map for each key k. Therefore there is an inverse function, and the cipher is usually designed to compute both using the same hardware.

Classical examples: Hill and Vigenère systems. You should note that the first system has a "mixing" property that the second does not, namely, each output symbol is potentially a function of all input symbols in the block. This type of mixing strengthens the cipher.

Nowadays, all block ciphers work at the bit level. So we can think of a block cipher as a Boolean function. If the key has length ℓ then this Boolean function has $m + \ell$ inputs and m output bits.

Good example to keep in mind: DES. Here $m = 64$ and $\ell = 56$.

What does classical cryptography tell us about block ciphers?

We want m to be large. If m is small, we have a substitution cipher on a small alphabet, and as we have seen, these are vulnerable to statistical analysis.

We also want the keyspace K to be large. If it is small, a cryptanalyst can try all keys.

The encryption functions should not be linear. If it is linear, we are basically using a Hill system, which is vulnerable to known plaintext attack.

For similar reasons, the encryption function should not be affine.

Defn: $f : {\mathbf{Z}_N}^m \rightarrow {\mathbf{Z}_N}^m$ is affine if there is a matrix A and vector b for which $f(x) = Ax + b$.

Affine systems can be attacked by *differencing*. Suppose we have two plaintext-ciphertext pairs, say

$$y_1 = Ax_1 + b$$
$$y_2 = Ax_2 + b$$

Subtracting, we obtain

$$y_1 - y_2 = A(x_1 - x_2),$$

which is a plaintext-ciphertext pair for the Hill system. Given enough such pairs, we can determine A and then use x_1, y_1 to get b.

Modes of operation

Block ciphers are flexible, which accounts for their popularity. For example, they can be combined with other devices to make stream ciphers or cryptographic systems with nice error-handling qualities. Here are some standard methods.

In all of these description, B_k is a block cipher with key k taking m bits to m bits.

Codebook mode.

Break the message up into blocks of m bits, say

$$x_1, x_2, x_3, \ldots$$

The corresponding ciphertext is

$$y_1, y_2, y_3, \ldots$$

with $y_i = B_k(x_i)$. Note that the same key is used for all blocks.

This mode has some weaknesses. First, if the source of the x_i's has low entropy, we are effectively dealing with a cipher on a small alphabet. Another, related problem is the presence of repetitive patterns in uncompressed text. For example, a computer program might have many blanks following each line of code, and each block of blanks will encrypt into the same cipher block. (Meyer/Matyas p. 64 have an example of this.)

Counter mode.

As we have seen, one way to beef up the LFSR stream cipher is to filter the output through a non-linear function. A block cipher can be used to provide the filtering, which has the advantage that it can be keyed.

Mathematically, let $\xi_1, \xi_2, \ldots$ be a maximum-length sequence produced by an m-bit shift register. Define blocks by

$$\begin{aligned} x_1 &= \xi_1 \xi_2 \cdots \xi_m \\ x_2 &= \xi_2 \xi_3 \cdots \xi_{m+1} \\ x_3 &= \xi_3 \xi_4 \cdots \xi_{m+2} \end{aligned}$$

and so on. Let f extract b bits of its m-bit input. The pseudo-random stream

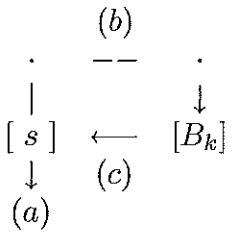
$$y_1, y_2, \ldots$$

with $y_i = f(B_k(x_i))$ can be used in a stream cipher.

The most convenient cases are probably $b = 1$ and $b = m$.

The next three modes share a common component: a loop using B_k to provide nonlinear feedback.

If you draw such a device, you will see that there three places to add in plaintext. We indicate these by (a), (b), and (c) in the figure below.



Think of s as the state.

Output feedback mode.

This is conceptually the simplest: construct a nonlinear feedback register using B_k, and use its output as a running key to be added to the plaintext.

In our picture above, this corresponds to adding in the plaintext at (a).

Mathematically, we choose an initial state s and then

$$\begin{aligned} &\texttt{for } i = 1, 2, 3, \ldots \\ &\quad s = B_k(s) \\ &\quad y_i = x_i + s \end{aligned}$$

transforms plaintext $x_1 x_2 \ldots$ into ciphertext $y_1 y_2 \cdots$.

Block chaining.

Here we work with m-bit blocks.

The message will be $x_1, x_2, \ldots$ (x_i is the i-th block)

Given an m-bit priming value y_0, the cryptogram $y_1, y_2, \ldots$ is made by doing

$$\begin{aligned} &\texttt{for } i = 1, 2, 3, \ldots \\ &\quad y_i = B_k(y_{i-1} + x_i) \end{aligned}$$

Referring to our picture, we are adding the plaintext in at (b).

As an exercise, you should verify that this is invertible, that is, one can recover plaintext given the ciphertext and the key.

Block chaining has a couple of nice properties. First, repeated blocks are enciphered into different values. Second, a transmission error is propogated into the entire following ciphertext. So errors are easy to detect.

Cipher feedback mode.

Mathematically, this corresponds to the scheme

$$\begin{aligned} &\texttt{for } i = 1, 2, 3, \ldots \\ &\quad y_i = x_i + B_k(y_{i-1}) \end{aligned}$$

Again, we need an initial vector y_0 to get started. This can be transmitted in the clear (think of it as a zero-th cipher block).

In terms of the picture, we are adding in plaintext at (c).

As an exercise you should prove that this is invertible as well.

A popular variant of this technique uses a shift register to store the state. Suppose the m is a multiple of b. We could take $b = 1$, for example, and encrypt one bit at a time. Let s denote an m-bit register, and let x_i and y_i be b-bit chunks of text. Encryption is done as follows:

$$\begin{aligned} &\texttt{for } i = 1, 2, 3, \ldots \\ &\quad y_i = x_i + f(B_k(s)) \\ &\quad s = (s \ll b) + y_i \end{aligned}$$

Here + denotes bitwise addition mod 2, and $\ll$ is a shift to the left. (The shift does not wrap around, so for example $10001 \ll 2$ is 00100.) The function f extracts b bits from the output of B_k.

Nice error recovery properties: If a block of the cryptogram is corrupted in transmission, this only affects a fixed number of message blocks. (How many?)

Notes

Sinkov (p. 131) suggests using Kasiski examination to determine block length, when codebook mode is used.

There are a few other operation modes (see, e.g. Schneier's book), but the five above are probably the most widely used.

Stream ciphers and other encryption gadgets with internal state can fruitfully be thought of as finite automata. The theory of finite automata is useful in several areas of computer science, for an introduction see M. Sipser, Introduction to the Theory of Computation.

A useful compendium of block cipher lore is C.H. Meyer and S.M. Matyas, Cryptography: A New Dimension in Computer Data Security, Wiley 1982. The accent is on DES and older IBM systems, but the book contains stuff you can't find anywhere else.

The Block Cipher Lounge is an interesting web site listing solution methods for various modern block ciphers. See

`http://www2.mat.dtu.dk/people/Lars.R.Knudsen/bc.html`

Lecture 26

In this lecture we finish up general block cipher techniques and study Feistel ciphers, an important family of block ciphers.

What do we do with short blocks?

In general, the length of a message will not be a multiple of the block length. There are a couple of techniques for dealing with this.

Padding.

Fill the remaining bits of a short block with pseudo-random bits. (We have seen this before in our discussion of transposition schemes.) These could be obtained from the message in a number of ways.

This requires the true length of the message to be communicated.

In general, this is OK for messages that are being transmitted.

Ciphertext stealing.

In certain applications (such as databases with fixed length records) the encryption transformation cannot increase the length of the message.

Suppose the last two blocks of the message are x_{n-1} and x_n. We assume that the block length is m, and that x_n has $m' < m$ bits. We compute

$$B_k(x_{n-1}) = \alpha \circ \beta$$

where β has $m - m'$ bits. The last two blocks are

$$y_{n-1} = \alpha \qquad \text{a short block}$$

and

$$y_n = B_k(\beta \circ x_n) \qquad \text{ordinary length.}$$

This is invertible when the length of the short block is known. This will be the case in a data storage application, for example, because the field length is fixed and known to the designer.

Feistel ciphers

These are named for Horst Feistel, who began his career at MITRE in the 1950's and later worked for IBM.

Suppose x, y represent elements of ${\mathbf{Z}_N}^m$, and f is *any* function from ${\mathbf{Z}_N}^m$ to itself. Then the mapping

$$x' = y$$
$$y' = x + f(y)$$

is invertible.

To prove this we work out the inverse mapping:

$$y = x'$$
$$x = y' - f(y) = y' - f(x')$$

This is a great result because we have made no assumptions whatsoever about the function f. To get keyed transformations we let f depend on a key k.

Feistel's idea was to repeat transformations of this type, possibly with varying f. Each repetition is called a *round*. The easiest way to understand how this works is to look at some examples.

Feistel ciphers with constant f.

Let's take f to be a constant function, with two rounds. Suppose the first constant is k_1 and the second is k_2. Then we have

$$x'' = y' = x + k_1$$
$$y'' = x' + k_2 = y + k_2$$

We recognize this as a Vigenère system, with period $2m$.

Because the composition of two Vigenère systems is another one, using more rounds will not increase security.

Feistel ciphers with linear f

Suppose $f(y) = Ay$ is a linear transformation, that is, $f(y_1 + y_2) = f(y_1) + f(y_2)$. We see that the one-round transformation is

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 0 & I \\ I & A \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

This is a linear transformation, so we have a Hill cipher.

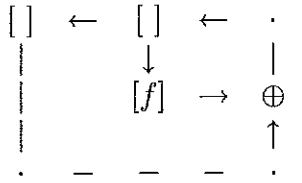
As before, using more rounds does not increase security.

Similarly, if we take $f(y) = Ay + b$, we get an affine cipher. This will still be affine no matter how many rounds are used.

Feistel designed many ciphers using $N = 2$.

Taking $N = 2$ means that the cipher algorithm is working on bits, and it is equivalent to a 2-cell shift register in which each cell holds m bits.

A picture for this is



Some Feistel designs:

NDS (New Data Seal) had a 128-bit block length, and 16 rounds. The same f was used for each round, which turned out to be a weakness. (See Beker and Piper, p. 264 and Grossman and Tuckerman, IBM Tech. Report 1977.) One can figure out the key using a few thousand chosen plaintexts.

Lucifer used a different f for each round, and had a 128 bit key. [Details??]

DES (Data Encryption Standard) was proposed by IBM in response to a National Bureau of Standards call for encryption standards. This cipher is very important, both because it is used and for its influence on future designs. It has a 56-bit key, 64-bit block length, and 16 rounds of encipherment. In addition, a special permutation is applied before and after encipherment.

Because of its small size, one might think of it as Lucifer Lite. On the other hand, some aspects of DES were strengthened as a result of consultation with the NSA.

Notes

The basic idea of a Feistel round (which makes nonlinear invertible functions) appears also in the Toffoli gate, a 3-input nonlinear function that is universal for Boolean logic. (See Preskill's notes on physics of information, Caltech.)

Lecture 27

In this lecture we (finally!) present the DES encryption algorithm in almost full detail.

Reference: Federal Information Processing Standards 46, January 15, 1997. This is reproduced in many cryptography books.

Why study DES?

DES is an extremely widely used cipher. Your ATM machine, for example, probably uses it. Knowledge of DES will allow you to read about many important applications of cryptography.

DES is an example of an extremely well-designed cipher. It was given final form as a result of a collaboration between IBM and the U.S. National Security Agency, full details of which have yet to appear. It is one of the few publicly available examples of modern governmental cipher design.

Cryptographic ingredients

Before discussing the details of DES, it is worthwhile to get a clear idea of the pieces from which it is assembled.

Boolean functions

We will call any $F : \mathbf{Z}_2{}^m \to \mathbf{Z}_2{}^n$ an m-in n-out Boolean function.

A function of the form

$$F(x_1 \dots x_m) = x_{i_1} x_{i_2} \cdots x_{i_n}$$

is called a *permuted choice*.

Two subcases are commonly used.

If $n = m$ and $i_1 \dots i_n$ is a permutation of $\{1, \dots, n\}$, then the Boolean function is itself called a permutation. This amounts to rearranging some wires.

If $n > m$ the Boolean function is called an expansion. Usually this is 1-1, which means that every input bit contributes to the output.

Shift registers: We have studied these in the context of stream ciphers.

Top level of DES

In a nutshell, DES works like this:

$$\text{64 bit plaintext} \to \sigma \to \text{16 rounds Feistel} \to \sigma^{-1} \to \text{64 bit ciphertext}$$

The initial permutation σ was described in Lecture 6. It is the same for every message, so it adds nothing to the real security of the device.

DES uses a 56 bit key, which determines 16 48-bit subkeys k_i. Each subkey bit is equal to one of the original key bits. The algorithm for determining the subkeys is called the *key schedule*.

The i-th Feistel round is

$$x' = y$$
$$y' = x + f_{k_i}(y)$$

where x and y are 32 bits.

How does f work?

[There should really be an illustration here. Stinson p. 72 has a good one. See also Trappe and Washington, p. 126.]

Certain bits of the 32-bit quantity y are duplicated to make a 48-bit quantity y^*. This is called *expansion*.

Expansion can be described by giving a table of 48 numbers, each in the range 1..32. The DES expansion table begins

32 1 2 3 4 5 4 5 6 7 8 9 8 9 ...

So bit 1 of y^* equals bit 32 of y, and so on. (See the standard for a complete table.)

Notice that bits 4 and 5 are duplicated, as are bits 8 and 9, and so on. This makes changes in y propogate quickly through the rounds. (Can we quantify this?)

The subkey k_i is added (bitwise addition mod 2) to y^* to make z.

The next stage of the algorithm involves eight 6-in 4-out Boolean functions $S_1, \ldots, S_8$.

These are called *S-boxes*. They are specified by tables (see the standard).

Let $z = z_1 z_2 \ldots z_8$, where z_i is a 6-bit block. We form

$$w = S_1(z_1)S_2(z_2)\cdots S_8(z_8)$$

which has 32 bits.

A transposition π is applied to w to make the output of the round. See the standard for a specification of π.

Note that everything in a round is fixed except for the subkey.

How are subkeys selected?

The key as presented to the algorithm is 64 bits long. Bits 8, 16, ... are used for parity checking, so that each 8-bit byte has odd parity.

Parity checking looks like an afterthought, and has led critics to conclude that the key length was intentionally shortened to make cryptanalysis feasible. Note that the data is not parity checked.

A permutation is applied to the 56 actual key bits (that is, bits $1-7$, $9-15$, and so on. (See the standard for its description.) This produces two 28 bit blocks which are loaded into shift registers.

The shift registers perform a cyclic shift, which means that they have the characteristic polynomial $X^{28} - 1$.

A register of this type is sometimes called a *ring counter.*

The registers are clocked once to make k_1, then twice to make k_2, and so on. The complete clocking schedule is

$$1\ 1\ 2\ 2\ 2\ 2\ 2\ 2\ 1\ 2\ 2\ 2\ 2\ 2\ 2\ 1$$

A 56-in 48-out permuted choice extracts the subkeys. This is the same for all subkeys, and specified in the standard.

Security

We note that the key schedule, that is, the mapping

$$k \mapsto k_1 k_2 \ldots k_{16}$$

is linear.

Therefore, the security resides in the S-boxes, which were carefully chosen. At present there is no good theory (at least in the open literature) that guarantees the security of a system of this type, so the designers used a number of selection criteria. It took a while for the criteria to become known. (See D. Coppersmith, IBM J. Res. Dev., 1994)

The number of n-in 1-out Boolean functions is 2^{2^n}. Therefore, there are

$$(2^{2^6})^4 = 2^{256} \approx 10^{77}$$

possible S-boxes. This is too many to examine individually, so one imagines a kind of heuristic search (perhaps guided by well selected pruning) was used to select interesting S-boxes for further study.

Here are some design criteria. The precise criteria are probably less useful than the reasons for their choice.

Each S-box is a 6-in 4-out Boolean function. Apparently, this would fit on a chip when DES was designed.

No output bit is too close to a linear or affine function of the inputs. This prevents cryptanalysis of DES as a Hill cipher.

For the next criterion we need the concept of *Hamming distance.* If x and y are n-bit quantities, then $\delta(x, y)$ denotes the number of positions in which x and y differ. For example

$$\delta(001, 111) = 2.$$

The S-boxes were chosen to make

$$\delta(x, x') = 1 \Rightarrow \delta(S(x), S(x')) \geq 2.$$

The goal here is to make any change in plaintext spread rapidly through the ciphertext (an avalance effect). Two other design criteria are similar to this.

Finally, there are criteria based on differencing. For any $d \in {\mathbf{Z}_2}^6$, there are 2^6 pairs x, x' with $x - x' = d$. Similarly, there are 2^4 possible output differences $S(x) - S(x')$. Suppose we restrict attention to pairs x, x' with a given d. If the outputs were chosen at random, the expected number of such pairs with a given output difference is $2^6/2^4 = 4$. The S-boxes were chosen so that, for each possible input difference, the actual number is no more than twice this, that is, 8.

This last criterion was chosen to slow down an attack that later became known as *differential cryptanalysis*. This was known to the designers, but not reproduced in the outside world until 1991.

Lecture 28

In this lecture we will discuss the security of multiple encryption

Reference: Merkle and Hellman, CACM v. 24, 1981, pp. 465-467.

The cryptanalytic methods we will discuss are called "time/space tradeoffs" or "birthday problem" or "meet in the middle" attacks.

Birthday problems

Suppose we have a universe of size N, say $\{1, \ldots, N\}$. We sample from this without replacement. We wish to answer the following type of question: how many duplications will there be?

The classic birthday problem

Choose $X_1, \ldots, X_n$ with replacement. A match is a pair $i < j$ with $X_i = X_j$. We will let $X_{ij} = 1$ if i, j is a match and 0 otherwise. Then the expected number of matches is

$$\begin{aligned} E\left[\sum_{1 \le i < j \le n} X_{ij}\right] &= \sum_{1 \le i < j \le n} E[X_{ij}] \\ &= \sum_{1 \le i < j \le n} 1/N \\ &= \binom{n}{2}/N \\ &\sim \frac{n^2}{2N} \end{aligned}$$

This is large when n is near $\sqrt{N}$. In particular, when $n = 1.414...\sqrt{N}$, the expected number of matches is 1.

Example: if birthdays are randomly distributed from 1 to 365 (no leap years), the expected number of matches among 27 people is 1. One can further show that for $n = 23$, the existence of a match is about as likely as not. (See Feller v. 1)

Matching between two samples

Let us choose two samples, say $X_1, \ldots, X_{n_1}$, and $Y_1, \ldots, Y_{n_2}$. By similar reasoning, the expected number of pairs i, j with $X_i = Y_j$ is

$$\frac{n_1 n_2}{N}$$

Taking $N = 365$ again, consider a singles bar with n_1 men and n_2 women. If $n_1 = n_2 = 19$, the expected number of couples with a matching birthday is close to 1. (It is exactly $361/365 = 0.989...$)

One cryptographic application of this involves messages encrypted with multiple keys. If we have K keys, then once the cryptanalyst has accumulated about $\sqrt{K}$ messages, there is a good chance two of them will be encrypted the same way. This is one reason why the random number generators for a stream cipher must have enormous periods.

Multiple Encryption

Suppose we have a block cipher like DES, with m-bit blocks and and ℓ-bit key. One natural idea for increasing the security of the cipher is to compose it with itself one or more times.

Formally, we define

$$e_{k_1,k_2}(x) = e_{k_2}(e_{k_1}(x))$$

and the corresponding decryption function is

$$d_{k_1,k_2}(y) = d_{k_1}(d_{k_2}(x))$$

Note that the order of the keys is reversed when decrypting.

One can similarly define triple encryption, quadruple encryption, and so on.

One popular variant of triple encryption uses two keys, and is defined by

$$e_{k_1,k_2}(x) = e_{k_1}(d_{k_2}(e_{k_1}(x)))$$

This was suggested by Tuchman of IBM, in part because taking $k_1 = k_2$ reduces it to single encryption. This provides upward compatibility with single encryption.

For some systems (affine, Hill, Vigènere), the set $\{e_k : k \in K\}$ is closed under composition. In such cases multiple encryption offers no more security than single encryption.

A set of 1-1 onto functions that is closed under composition is called a *group*. For this reason, it is of interest to know whether a particular block cipher's transformations form a group. It's known, for example, that DES is not a group. [Campbell and Weiner, CRYPTO 1992]

We will now discuss some attacks on these systems.

A Known Plaintext Attack on Double Encryption

Suppose our system is given by

$$e_{k_1,k_2}(x) = e_{k_2}(e_{k_1}(x))$$

and we have two matching plaintext-ciphertext pairs (x, y) and (x', y')

We'll assume that the underlying block cipher has m-bit blocks, and keys of ℓ bits.

To find the keys, we do the following:

For all $k \in K$

Add the tuple $(e_k(x), e_k(x'), k, L)$ to a list
Add the tuple $(d_k(y), d_k(y'), k, R)$ to a list
Sort the list on the first two entries
If there is exactly one pair
(z, z', k_1, L)
(z, z', k_2, R)
then output k_1, k_2.

It is clear that the correct pair of keys will produce a match. We can estimate the number of spurious matches using our birthday problem results.

Any incorrect key pair will be ascribed to bad luck, so we'll assume the encryptions are random samples, as are the decryptions. Each of these samples is two blocks long, so the universe of possible samples has size $N = 2^{2m}$. There are 2^ℓ encryptions and an equal number of decryptions. so $n_1 = n_2 = 2^\ell$. Hence we expect

$$\frac{2^{2\ell}}{2^{2n}} = 4^{\ell - n}$$

incorrect key pairs.

For DES, we have $\ell = 56$ and $n = 64$, so the expected number of spurious key pairs is

$$4^{-8} = 1/65536 = 1.5 \times 10^{-5}.$$

This is small enough to give us confidence in the algorithm.

This algorithm uses space for $2^{\ell+1}$ plaintexts and requires time $O(\ell 2^\ell)$ for the sorting. Note that

$$2^\ell \times \ell 2^\ell \sim 2^{2\ell+1},$$

which is close to the number of key pairs that must examined in a brute-force search.

Merkle and Hellman have pointed out that by testing one key pair and filtering out the others, one can reduce the space requirements somewhat.

Attack on Tuchman's Triple Encryption

We assume the system is of the form

$$e_{k_1,k_2}(x) = e_{k_1}(d_{k_2} e_{k_1}(x))$$

In block diagram form this is

$$x \rightarrow \overset{k_1}{[E]} \rightarrow z \rightarrow \overset{k_2}{[D]} \rightarrow w \rightarrow \overset{k_3}{[E]} \rightarrow y$$

This is a chosen plaintext attack, so we assume access to the underlying encryption device. To emphasize that the key is not yet known, we refer to this mapping as $EDE(x)$.

The basic idea is to work with plaintexts whose ciphertext z is fixed and known to us. Decrypting z with all keys will find these plaintexts. Then one can use ideas of the last method to try to find suitable k_2 and k_1 for the decryption and last encryption.

Let 0 and 1 denote two distinct blocks, for example the all 0's block and the all 1's block. The algorithm is

For all $k \in K$
    Let $x_0 = d_k(0)$ and $x_1 = d_k(1)$.
    Let $w_0 = d_k(EDE(x_0))$ and $w_1 = d_k(EDE(x_1))$
    Add the tuple $(w_0, w_1, k, \text{outer})$ to a list
    Let $w'_0 = w_0$ and $w'_1 = x_1$
    Add the tuple $(w'_0, w'_1, k, \text{inner})$ to a list
Sort the list on the first two entries
If there is exactly one pair
    $(w_0, w_1, k_1, \text{outer})$
    $(w_0, w_1, k_2, \text{inner})$
then output k_1, k_2.

By chasing 0 and 1 through the system we see that the correct keys will produce a pair. As before the expected number of incorrect key pairs is very small.

It's interesting to consider the precise assumption that underlies our assertion that spurious key pairs are unlikely. We are basically assuming that the sets

$$\left\{ d_k \begin{pmatrix} 0 \\ 1 \end{pmatrix} : k \in K \right\}$$

and

$$\left\{ d_k \left(EDE \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) : k \in K \right\}$$

are independent draws from the set of all possible block pairs. If the underlying cipher is good this is a reasonable assumption.

Practical Lessons

One can draw two morals from this lecture.

First, if a birthday problem can be exploited, the complexity of a search problem can often be reduced by the square root. Along with cryptanalysis, the baby-step/giant-step algorithm for solving $ax = b \bmod N$ is a another example of this. Algorithms relying on the birthday problem tend to have high space consumption.

Second, if a cipher is to be made by cascading together simpler ones, the designer should make sure that information from the key gets into all stages. The key schedule in DES is an example of this.

Lecture 29

This lecture: Finite Fields

The next block cipher we will cover is the Advanced Encryption Standard (AES). This block cipher is built around the mathematical idea of a finite field, to which this lecture is devoted.

Reference: Trappe and Washington, Section 3.11.

What is a Field?

Recall that the system $(\mathbf{Z}_N, +, \cdot)$ satisfies some familiar algebraic laws:

1. Addition is commutative and associative.
2. Adding any element to 0 gives you back that element.
3. Multiplication is commutative and associative.
4. Multiplying any element by 1 gives you back that element.
5. The distributive law: for any x, y, z we have

$$(x + y)z = xz + yz$$

In addition, when n is prime, there is an additional property:

6. When $x \neq 0$ there is a y such that $xy = 1$.

Any system $(F, +, \cdot)$ satisfying these six axioms is called a *field*. Not surprisingly, it is a *finite field* if $|F| < \infty$.

Roughly speaking, a field is a system in which you can do high-school algebra.

Examples include $\mathbf{Q}$ (the rational numbers), $\mathbf{R}$ (the real numbers), and $\mathbf{C}$ (the complex numbers), with the usual addition and multiplication laws.

Here is a more exotic one. Let $F = \{0, 1\}$, with addition and multiplication done mod 2. In our course, we have called this $\mathbf{Z}_2$.

The symbols $\mathbf{F}_q$ or $GF(q)$ stand for a finite field with q elements. Thus, the last example could also be called $GF(2)$. "GF" stands for "Galois field" which is another name for finite field. This term is quite common in engineering and coding theory literature.

Making Finite Fields

We will use the following result without proof. For every $n \geq 1$, there is a finite field F with 2^n elements. It can be defined as follows:

Choose a degree n polynomial $f(X)$ with coefficients in $GF(2)$ that is irreducible mod 2.

The elements of F are the polynomials of degree $< n$:

$$c_{n-1}X^{n-1} + \cdots + c_1 X + c_0, \qquad c_i \in GF(2).$$

Addition in F is just addition of polynomials – just add the coefficients.

To multiply, form the product of polynomials, and then take the remainder after long division by f.

It can be shown that different choices of f lead to a fields that are the same, except for relabeling the elements. Hence, we are entitled to call this field $GF(2^n)$.

Warning: Even if two systems are abstractly the same, the effort involved in using their concrete realizations may be vastly different. Consider, for example, the natural numbers in Arabic (base 10) notation, versus Roman numerals.

Example: let $f = X^2 + X + 1$. The field elements are

$$0, 1, X, X + 1,$$

so this is $GF(4)$. We will abbreviate $c_1X + c_0$ by c_1c_0.

Here is the addition table:

	00	01	10	11
$0 = 00$	00	01	10	11
$1 = 01$	01	00	11	10
$X = 10$	10	11	00	01
$1 + X = 11$	11	10	01	00

In computer science lingo, we add two field elements by XORing their bits. This table has the property that in a row (or column), each possible element appears exactly once. Such tables are called *Latin squares.*

In our field, we have

$$X^2 = X + 1,$$

$$X(X + 1) = X^2 + X = 1,$$

and

$$(X + 1)^2 = X^2 + 2X + 1 = X^2 + 1 = X.$$

So the multiplication table is

	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	11	01
11	00	11	01	10

This table is not as easy to describe, but we can observe some features. First, the elements in the first row and first column are all zero. Next, the second row and the second column just copy inputs from the top and left, respectively. (Ask

yourself what algebraic laws these properties reflect. Now check that these laws follow from the axioms for a field.)

The Field Used by AES

AES works with $GF(2^8)$, implemented as follows.

Let $f = X^8 + X^4 + X^3 + X + 1$.

You should verify that this is irreducible. The hard way, which you can do it you want some practice in polynomial arithmetic, is to take each polynomial of degree 4 or less, and divide it into f. You will get a nonzero remainder in every case.

Field elements are thus bit vectors of length 8:

$$c_7X^7 + c_6X^6 + c_5X^5 + c_4X^4 + c_3X^3 + c_2X^2 + c_1X + c_0 = c_7c_6c_5c_4c_3c_2c_1c_0$$

To add, we just add the bits mod 2. For example,

$$01010101 + 11110000 = 10100101.$$

Multiplication is best thought of as involving polynomials. For example,

$$00010000 = X^4, \qquad 00010001 = X^4 + 1.$$

The product is

$$X^8 + X^4,$$

but $X^8 = X^4 + X^3 + X + 1$, so the product is

$$2X^4 + X^3 + X + 1 = X^3 + X + 1 = 00001011.$$

Remember that in $GF(2)$, 2 times anything is 0.

It is recommended that you do a few more examples until you are very familiar with this.

For the AES algorithm, we'll need to know a^{-1} for each $a \neq 0$.

Here is a brute force approach. For each nonzero a, run through the b's until $ab = 1$. This requires you to compute about $2^8 \times 2^8 = 2^{16} = 65536$ field elements, which is certainly feasible on a computer.

Here another algorithm, which scales up a little better, and could actually be done by hand for a given a. (Try it!) Consider the equation

$$a(cX^4 + d) = 1, \qquad \deg c, \deg d < 4.$$

Rewrite this as

$$acX^4 = ad + 1.$$

Enumerate the 16 values of acX^4, and the 16 values of $ad+1$. Sort these 32 field elements and look for matches.

Another approach, which is even faster, is to use Euclid's algorithm to solve the equation

$$\alpha a + \beta f = 1$$

in polynomials. Then $\alpha a = 1$ in $GF(2^8)$. This is similar to using the usual Euclidean algorithm (for numbers) to find inverses in $\mathbf{Z}_N$.

Notes.

The theorems we have used can be found in any abstract algebra book. One particularly accessible text is G. Birkhoff and S. Maclane, A Survey of Modern Algebra. See Chapter XV, Section 6.

Lecture 30

This lecture: AES

In 2000, the US government (more precisely, its National Institute for Standards and Technology – NIST), announced a replacement for DES, to be the standard block cipher for general use.

The selection process involved a contest, to which several designs were submitted for evaluation. The winning design was by Joan Daemen and Vincent Rijmen. It was originally called Rijndael, but now is named the Advanced Encryption Standard. We'll just call it "AES."

AES improves upon DES in several ways:

Larger block size (128 bits).

Larger key size (can be 128, 192, or 256 bits).

Nonlinear algorithm to produce subkeys.

The S-boxes are based on algebraic formulas. (This is a "political" improvement, as it eliminates the possibility of a secret method for inserting trapdoors.)

We'll describe the 128-bit key version here. The other versions are very similar.

The Building Blocks of AES

We use the implementation of $F = GF(2^8)$ as given in the last lecture. Thus, field elements are bit vectors of length 8. There is a distinguished element X that satisfies

$$X^8 = X^4 + X^3 + X + 1.$$

The algorithm works with 4×4 blocks of field elements. Thus, a block has $16 \cdot 8 = 128$ bits.

Every nonzero element of F has a multiplicative inverse. We extend this to all of F, by agreeing that

$$0^{-1} = 0.$$

This makes $x \mapsto x^{-1}$ a permutation on F.

Warning: this is most definitely something extra, and is not part of the definition of a finite field.

The analog of the DES S-box is a function called ByteSub (BS). This function works by inverting elements, then applying an affine-Hill encryption:

$$\mathrm{BS}(u) = Au^{-1} + b.$$

To make this precise, we have to specify A and b. First, let's think of the elements of F as column vectors, that is,

$$u = u^7X^7 + u^6X^6 + \cdots + u_1X + u_0 = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_6 \\ u_7 \end{pmatrix}.$$

The offset b is the column vector

$$b = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

The matrix A looks like this:

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

A matrix like A in which each row is the previous one, rotated by one position, is called a *circulant*.

It can be shown that when $n = 2^k$, and M is a circulant with first row $(a_1, a_2, \ldots, a_n)$, we have

$$\det M \equiv a_1^n + a_2^n + \cdots + a_n^n \pmod 2.$$

Applying this result with $k = 3$, we see that

$$\det A = 5 \cdot 1 + 3 \cdot 0 \equiv 1 \pmod 2,$$

so A is invertible. This makes

$$\mathrm{BS}(u) = Au^{-1} + b$$

a permutation of F.

Round keys.

As with DES, encryption proceeds in a sequence of *rounds*. Each round uses a different subkey, which is added mod 2 to the state during the round.

Round keys are derived by a time-varying nonlinear shift register algorithm.

The initial key is a 4-tuple (c_0, c_1, c_2, c_3), where c_i is a column consisting of four field elements. As advertised, this consists of $4 \times 4 \times 8 = 128$ bits total.

For $i \geq 4$, we define new columns by

$$c_i = \begin{cases} c_{i-1} + c_{i-4}, & \text{if } i \not\equiv 0 \bmod 4; \\ N_i(c_{i-1}) + c_{i-4}, & \text{if } i \equiv 0 \bmod 4, \end{cases}$$

where

$$N_i \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \mathrm{BS}(\beta) \\ \mathrm{BS}(\gamma) \\ \mathrm{BS}(\delta) \\ \mathrm{BS}(\alpha) \end{pmatrix} + \begin{pmatrix} X^{(i-4)/4} \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Here, $\alpha, \ldots, \delta$ refer to field elements.

Note that N_i depends on i, and is only defined when i is a multiple of 4. You should check that N_i is invertible.

By Golomb's criterion for nonlinear shift registers, the state evolution mapping

$$(c_0, c_1, c_2, c_3) \to (c_1, c_2, c_3, c_4) \to (c_2, c_3, c_4, c_5) \to \cdots$$

is reversible. Thus, the subkey generation process does not destroy any of the information in the initial key.

The actual round keys are

$$\begin{aligned} \text{0th round key} &= (c_0, c_1, c_2, c_3), \\ \text{1st round key} &= (c_4, c_5, c_6, c_7), \\ \text{2nd round key} &= (c_8, c_9, c_{10}, c_{11}), \end{aligned}$$

etc. We will make 11 round keys in all.

Linear Ingredients

The remaining parts of the algorithm are linear, and act on 4×4 arrays of field elements.

SR (shift rows):

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \mapsto \begin{pmatrix} a & b & c & d \\ f & g & h & e \\ k & l & i & j \\ p & m & n & o \end{pmatrix}$$

MC (mix columns): multiply the state matrix by the 4×4 circulant

$$M = \begin{pmatrix} X & X+1 & 1 & 1 \\ 1 & X & X+1 & 1 \\ 1 & 1 & X & X+1 \\ X+1 & 1 & 1 & X \end{pmatrix}.$$

That is, if S is the state, $\mathrm{MC}(S) = MS$.

ARK: add a round key to the state. Since both states and round keys are 4×4 arrays of elements of F, this is addition in F, i.e. addition mod 2 of 128-bit blocks.

The AES Encryption Algorithm

To encrypt, make 11 round keys as indicated above. Then apply the following transformations:

ARK (using 10th round key).
for t from 9 down to 1:
    BS, SR, MC, ARK (using tth round key)
BS, SR, ARK (using 0-th round key)

In this algorithm, BS applied to a state means to apply BS to each of the 16 field elements comprising the state.

You should observe that MC is missing from the last round. This makes the number of basic steps (ARK, BS, etc.) a multiple of 8, which is good from a pipelining perspective. Another reason is the following. If we included MC in the last round, the last two operations would be, for a state s and round key k,

$$Ms + k = M(s + M^{-1}k).$$

From a cryptanalysis perspective, we could just imagine that the right-hand expression was used, and then the last M would not add any security, since it is the same for all messages.

Decryption just undoes this sequence of transformations. (Note that this will use round keys in the order in which they are generated.) There are some tricks to make the algorithm for decryption more closely resemble the algorithm for encryption.

Ideas behind the design of AES

AES retains the "layered" structure of DES, in that it alternates linear mixing transformations with small, locally applied nonlinear functions.

The analog of the DES S-box is the nonlinear byte sub (BS) transformation, which acts on field elements. If f is such a transformation from F to F, we can study the *components* f_i, defined by

$$f(\alpha) = \sum_{i=0}^{7} f_i(\alpha) X^i.$$

Here are three desirable properties for a nonlinear function f:

No sum of components is close to an affine function. Here distance means Hamming distance, the number of positions at which two bit vectors differ.

The algebraic form of each f_i has high degree.

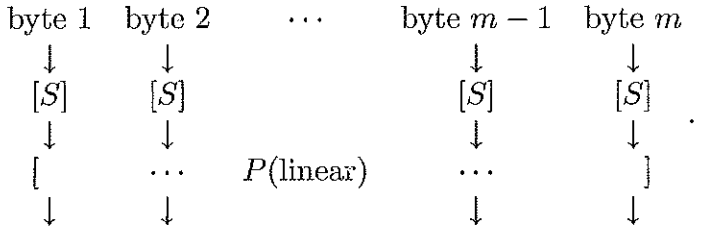
The differences of f are well distributed, in the sense that for all nonzero α and for all β,

$$\#\{z : f(x+\alpha) - f(z) = \beta\} \leq c,$$

where c is a small positive number.

Nyberg proved that $f(\alpha) = \alpha^{-1}$ performs well by all three criteria.

The design of AES involves a "bricklaying" structure, which alternates small S-boxes with a wide linear permutations:



The notion of optimal (MDS) error correcting codes underlies the choice of the permutation P.

Notes.

AES is described in a number of sources. The definitive work is by its designers: J. Daemen and V. Rijmen, The Design of Rijndael, Springer-Verlag 2002. For a very nice textbook treatment, see Chapter 11 of D. W. Hardy and C. L. Walker, Applied Algebra: Codes, Ciphers, and Discrete Algorithms, Prentice-Hall, 2003.

Lecture 31

This lecture: More block ciphers

There are many published block ciphers, many of them based on the same ideas as DES. In this lecture we'll look at two of them.

IDEA was designed by Lai and Massey. The original cipher, presented at EUROCRYPT 90, was souped up and published in final form in EUROCRYPT 91.

Skipjack was designed by the US government (presumably a group at NSA). It was the basis for the ill-fated escrowed encryption standard, and has recently been declassified. It is worthy of study as another example of the state of the cryptographic art.

IDEA

All cryptographic systems rely on some trick for making lots of invertible functions. Here's what IDEA uses.

For any function f, the transformation

$$s' = s + f(s+t)$$
$$t' = t - f(s+t)$$

is invertible. (The domain of s and t is anything over which the addition and subtraction make sense. IDEA uses $\mathbf{Z}_2{}^m$ with $m = 32$.) To see this, observe that $s' + t' = s + t$, so we can solve for s and t:

$$s = s' - f(s'+t')$$
$$t = t' + f(s'+t')$$

IDEA is a block cipher with 64-bit blocks and a 128-bit key. We think of a block as divided into 4 16-bit words.

IDEA uses 3 algebraic operations on 16-bit words

$\oplus$ denotes bitwise exclusive or, that is, addition in $\mathbf{Z}_2{}^{16}$.

$+$ denotes addition mod 2^{16}.

$\otimes$ denotes multiplication in $\mathbf{Z}^*_{2^{16}+1}$, in a non-standard representation. We note that $2^{16} + 1 = 65537$ is prime, so $\mathbf{Z}^*_{2^{16}+1}$ has 2^{16} elements. The numbers $1, \ldots, 2^{16} - 1$ stand for themselves, and 0 stands for 2^{16}, that is, -1.

It should be noted that all these operations are commutative, and have the further property that if one operand is fixed, the resulting one-variable function is 1-1.

There are eight rounds (identical except for the subkeys used) followed by an output transformation.

Each round consists of the following operations.

Selection of 6 16-bit subkeys $k_1, \ldots, k_6$. (The precise selection depends on the round number.)

A preliminary transformation

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} k_1 \otimes x_1 \\ k_2 + x_2 \\ k_2 + x_3 \\ k_3 \otimes x_4 \end{pmatrix} \qquad (*)$$

A Feistel-like transformation

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \oplus f\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \oplus \begin{pmatrix} x_3 \\ x_4 \end{pmatrix}\right)$$

$$\begin{pmatrix} x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \end{pmatrix} \oplus f\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \oplus \begin{pmatrix} x_3 \\ x_4 \end{pmatrix}\right)$$

(these two are performed simultaneously), followed by a swap of x_2 and x_3.

In this last transformation, f is key-dependent. Explicitly, if

$$f\begin{pmatrix} r \\ s \end{pmatrix} = \begin{pmatrix} r' \\ s' \end{pmatrix}$$

then

$$\begin{aligned} t &= r \otimes k_5 \\ s' &= (t + s) \otimes k_6 \\ r' &= t \otimes s' \end{aligned}$$

The output transformation is just (*) with different subkeys.

The subkey selection algorithm is given in Menezes, van Oorschot, and Vanstone, Handbook of Applied Cryptography, p. 263 ff.

Skipjack

Skipjack has an 80-bit key and 64 bit blocks.

Overview: a shift register machine, with nonlinear S-box like mixing functions.

Reference: See the web site

`http://csrc.nist.gov/encryption/skipjack-kea.htm`

[This link is stale – find a better one.]

Supposedly very memory-stingy, you need only a few bytes (!) of RAM.

Lecture 32

Big Number Arithmetic

Some references:

R. L. Swain, Understanding Arithmetic, Rinehart, 1957.

D. E. Knuth, The Art of Computer Programming, v. 2: Seminumerical Algorithms. 3rd Ed., Addison-Wesley, 1998.

A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.

Why Big Numbers?

To achieve the desired level of security, RSA and similar systems must operate with integers with hundreds or thousands of bits. Commercial processors, on the other hand, are designed to do arithmetic on numbers with tens of bits. To work with bigger numbers some programming is usually necessary.

With modular arithmetic in mind, we assume all integers nonnegative unless stated otherwise.

A number x can be represented in base b, i.e. in the form

$$x = x_{k-1}b^{k-1} + \cdots + x_2b^2 + x_1b + x_0$$

with $0 \le x_i < b$. There are k base-b digits, and we have $0 \le x < b^k$.

Usually b is a power of 2, but it can be any integer greater than 1.

The Classical Algorithms

Addition:

We want to add

$$x = x_{k-1}b^{k-1} + \cdots + x_1b_1 + x_0$$

to

$$y = y_{k-1}b^{k-1} + \cdots + y_1b_1 + y_0$$

The classic method is to set $c_0 = 0$ and then compute for $i = 0, \ldots, k-1$:

$$z_i = (x_i + y_i + c_i) \bmod b;$$
$$c_{i+1} = \lfloor \frac{x_i + y_i + c_i}{b} \rfloor.$$

This produces

$$z = c_kz^k + z_{k-1}b^{k-1} + \cdots + z_1b_1 + z_0$$

Note that the "carry" c_i is always either 0 or 1.

Subtraction:

Similar to addition except that we "borrow." If x and y are as above, then we compute the digits of $x - y$ by setting $c_0 = 0$, then finding for $i = 0, \ldots, k-1$

$$z_i = (x_i - y_i + c_i) \bmod b;$$
$$c_{i+1} = \lfloor \frac{x_i - y_i + c_i}{b} \rfloor.$$

So c_i will either be 0 or -1. If $y \leq x$ then the last borrow, c_k, is 0 and we get

$$x - y = z_{k-1}b^{k-1} + \cdots + z_1 b_1 + z_0$$

When $y > x$, there will be a borrow out, so $c_k = -1$. This actually gives us a way to do comparison, but it is much faster (on average) to compare the digits, most significant first.

Multiplication:

Suppose first that we want to multiply

$$x = x_{k-1}b^{k-1} + \cdots + x_2 b^2 + x_1 b + x_0$$

by a, with $0 \leq a < b$. To do this, we set $c_0 = 0$ and then, for $i = 0, \ldots, k-1$:

$$z_i = (ax_i + c_i) \bmod b;$$
$$c_{i+1} = \lfloor \frac{ax_i + c_i}{b} \rfloor.$$

Now we can use the relation

$$x(y_{k-1}b^{k-1} + \cdots + y_0) = b^{k-1}(xy_{k-1}) + \cdots + b(yx_1) + yx_0$$

to multiply x by y. (Note that multiplying by a power of b is easy, it is just a shift.)

Division:

Given $x, y > 0$ we want to determine q and r making $x = qy + r$ and $0 \leq r < x$. We can use the usual long division algorithm as long as we have a way of determining the quotient for

$$x = x_k b^k + x_{k-1}b_{k-1} + \cdots + x_1 b + x_0,$$
$$y = \qquad y_{k-1}b^{k-1} + \cdots + y_1 b + y_0, \qquad y_{k-1} \neq 0.$$

If b is a power of 2, we can shift both x and y to ensure that $y_{k-1} \geq b/2$. In this case, it can be shown that

$$\hat{q} = \min \left\{ \lfloor \frac{x_k b + x_{k-1}}{y_{k-1}} \rfloor, b - 1 \right\}$$

satisfies

$$\hat{q} - 2 \leq q \leq \hat{q}.$$

(Proof: Knuth, vol. 2, section 4.3.1) This gives 3 possible values of q. Trial subtraction will indicate which is correct.

Cost:

It is useful to have formulas in which the base b appears. Suppose that the operands are n-bit numbers. Then there will be about

$$\log_b 2^n = \frac{n}{\log_2 b}$$

base b digits.

Addition or Subtraction uses $O(n/\log b)$ operations

Multiplication or Division uses $O(n^2/(\log b)^2)$ operations

Lecture 33

Some Faster Algorithms

Karatsuba Multiplication

Suppose that

$$u = u_1 b^{k/2} + u_0,$$
$$v = v_1 b^{k/2} + v_0,$$

with $0 \leq u_i, v_i < b^{k/2}$. That is, we now think of these numbers in base $b^{k/2}$. If

$$uv = \alpha b^k + \beta b^{k/2} + \gamma,$$

then

$$\alpha = u_1 v_1$$
$$\gamma = u_0 v_0$$
$$\beta = (u_1 + u_0)(v_1 + v_0) - (\alpha + \beta)$$

This replaces 4 multiplications by 3 multiplications and some extra additions and subtractions. When k is a power of 2, the algorithm can be used recursively. If this is done an algorithm with complexity $O((n/\log b)^{1.59})$ results. It is practical for numbers of a few hundred digits.

FFT Multiplication

The Fast Fourier Transform (FFT) algorithm can be used to multiply two degree k polynomials using $O(k \log k)$ operations. Asssuming this result, we can multiply x by y as follows.

Replace x and y by polynomials in the variable B:

$$p_x = x_{k-1} B^{k-1} + \cdots + x_2 B^2 + x_1 B + x_0$$
$$p_y = y_{k-1} B^{k-1} + \cdots + y_2 B^2 + y_1 B + y_0$$

Find their product

$$p_x q_y = z_{2k-2} B^{2k-2} + \cdots + z_1 B + z_0$$

Release the carries by evaluating this polynomial at $B = b$.

The asymptotic complexity of this algorithm is close to linear but it is preferable only for numbers with tens of thousands of digits. Such numbers have yet to play a role in cryptography.

Division by Newton Iteration

We can find the quotient q and remainder r for $x = qy + r$ if we have a good enough approximation to $1/y$.

After dividing by a suitable power of 2, we can assume that we are to invert a, satisfying $1 \leq a < 2$.

It is sufficient to approximate the root of

$$f(w) = a - 1/w = 0.$$

The Newton iteration

$$w' = w - \frac{f}{f'}(w) = w(2 - aw)$$

converges quadratically to $1/a$ if the starting point is close enough. To see this, compare

$$\frac{1}{a} - w = \frac{1 - aw}{a}$$

to

$$\frac{1}{a} - w' = \frac{1}{a} - w(2 - aw) = \frac{(1 - aw)^2}{a}$$

So if the starting point w_0 satisfies $|aw_0 - 1/2| < 1/2$ (this is not hard to arrange), and $t \geq \log_2 n$, then after t iterations we will have

$$\left| \frac{1}{a} - w \right| \leq 2^{-n}.$$

Examination of the graph of f shows that if we start with a value less than $1/a$, all successive values will also be. (This can be strictly enforced by making sure that we round all values downward.)

Now we refer back to the division problem. Suppose that that x is length $2k$ and y is length k. Compute a $k \log b$ – bit approximation to $1/y$ using Newton's method, taking care that this underestimates the true value. Then

$$\hat{q} = x \times \text{ [approximate } 1/y \text{]}$$

is an overestimate of the true quotient, accurate to within $O(1)$. It can be tested (see if $0 \leq x - \hat{q}y < r$), and decremented until it is correct.

The algorithm uses only addition, subtraction, and multiplication. So it can be combined with one of the fast multiplication methods discussed above. The usual procedure is to double the precision at each step, making the asymptotic complexity of division the same as the underlying multiplication algorithm.

For exponentiation calculations such as the RSA system's $x^e \bmod N$, we will be dividing by the same number N. So we can precompute a high precision approximation to N^{-1} and use this repeatedly.

Lecture 34

Computing powers mod N

Many cryptographic systems use the powering operation

$$x^e \bmod N,$$

and rely on the fact that this can be implemented efficiently.

Consider the following recursive algorithm:

$$x^e = \begin{cases} x, & \text{if } e = 1; \\ (x^{e/2})^2, & \text{if } e \geq 2 \text{ is even;} \\ (x^{e-1})x, & \text{if } e \geq 3 \text{ is odd.} \end{cases}$$

Let $T(e)$ be the number of multiplications used by this algorithm. We have the recurrence relation

$$T(e) \leq T(\lfloor e/2 \rfloor) + 2; \qquad T(1) = 0.$$

It is easy to show by induction that $T(e) \leq 2 \log_2 e$.

If we reduce mod N after each operation, we get an algorithm for computing x^e mod N, using $O((\log e)n^2)$ single-word operations. (Recall our assumption that $1 \leq N < 2^n$.

Montgomery's representation

Peter Montgomery realized that one can save a lot of the work in exponentiation by using a special representation of the residue classes mod n.

For details, see his paper: Modular multiplication without trial division, Mathematics of Computation 44 (1985), pp. 519-521.

This is also described in Section 14.3.2 of A. J. Menezes et al., Handbook of Applied Cryptography, CRC Press, 1997.

Lecture 35

[***** We should include some basic facts about primes and unique factorization before this point. *****]

Public Key Cryptography and the RSA System

Reference: Rivest, Shamir, and Adleman, CACM 1978 (?). A great paper.

Public key systems in general

In a traditional cipher system, anyone who knows how to encrypt a message knows how to decrypt it as well. For example, the same key serves for encryption and decryption in DES.

In the early 1970's, computer scientists and mathematicians became intensely interested in the inherent difficulty of computational problems. This resulted both from the rapidly expanding availability of computing power (a trend that has not stopped to this day) and from the discovery that many combinatorial problems that had so far resisted efficient solution were related, in the sense that if one of them could be solved, then so could any of the others.

Viewed from the perspective of computational complexity theory, a cryptographic system becomes simply a family of transformations that are hard to invert, indexed by the key. There is no reason whatever to suppose that knowledge of the forward transformation provides knowledge of the inverse transformation, except in special cases.

Public-key cryptography was discovered independently, by by James Ellis of GCHQ (this was only declassified recently) and by Diffie and Hellman of the US (who did publish their work).

Rivest-Shamir-Adleman system

This is a block cipher in which messages are written as strings of elements from Z_N. (In practice, we would find the largest k for which $2^k \leq N$, and break the message up into k-bit blocks, interpreting these as binary numbers.) Therefore it is enough to describe the encryption and decryption of one element of Z_N.

A user chooses two (distinct) large primes p and q and forms $N = pq$. He also chooses two numbers $d, e \leq N$ with the property that $de \equiv 1 \bmod (p-1)(q-1)$.

The user's *public key* is the pair (N, e).

His *encryption function* is

$$x \mapsto E(x) = x^e \bmod N.$$

His *secret key* is the pair (N, d).

His *decryption function* is

$$y \mapsto D(y) = y^d \bmod N.$$

Encryption and decryption uses $O(\log N)$ multiplications. mod N.

The idea is for each user to put his pair (N, e) in a kind of "phone book." To send a message x to that user, any other user can send

$$y = x^e \bmod N$$

over an insecure channel. The "owner" of that public key computes

$$z = y^d \bmod N,$$

which (as we'll see in a minute) equals x.

Why does the system work?

Ignoring issues of security for the moment, the least we expect is that messages get correctly transmitted. Algebraically, this boils down to proving that

$$D(E(x)) = x^{de} \bmod N = x.$$

We need a lemma.

Fermat's little theorem: if p is prime, then x^p is congruent to x mod p.

This was proved (or the proof at least sketched) in Lecture 17.

Back to encryption.

Since $de - 1$ is a multiple of $(p-1)(q-1)$, it is also a multiple of $p-1$. There is an integer k for which

$$de = 1 + k(p-1).$$

Now, we compute

$$D(E(x)) = x^{ed} = x^{ed} = x^{1+kp-k}$$

By Fermat's little theorem, this is congruent to

$$x^{1+k-k} = x$$

mod p.

(If you are a stickler for detail, you might be worrying about negative exponents. We can always define $x^{-1} \bmod p$, except when $x \bmod p = 0$, but the result is evidently true in this case anyway.)

We just showed that $D(E(x)) - x$ is a multiple of p. By symmetry, it must also be the case that $D(E(x)) - x$ is a multiple of q. Since $N = pq$, and p and q are distinct primes, $D(E(x)) - x$ must also be a multiple of N, i.e.

$$D(E(x)) = x.$$

An example of RSA.

Parameters (chosen to make this easy to follow):

$N = 3 \cdot 17 = 51$.

$\varphi = (p-1)(q-1) = 2 \cdot 16 = 32$.

We are looking for solutions to $de \equiv 1 \bmod 32$, so let's choose $e = 3$, $d = 11$.

Encryption.

Let $m = 2$. Then

$$E(m) = 2^3 \bmod N = 8.$$

Decryption.

We use the binary algorithm to compute

$$8^{11} = ((8^2)^2 8)^2 8 \bmod 51.$$

From the inside:

$$\begin{aligned} 8^2 &= 64 \equiv 13 \\ 13^2 &= 169 \equiv 16 \\ 16 \cdot 8 &= 128 \equiv 26 \\ 26^2 &= 676 \equiv 13 \\ 13 \cdot 8 &= 104 \equiv 2 \end{aligned}$$

Notes

The RSA algorithm was invented independently by Clifford Cocks of GCHQ. This was done in 1973 but not made public until many years later.

Lecture 36

RSA Key Generation, Primality, Factoring

How do we generate keys?

Unlike most cryptographic systems, RSA key generation requires a nontrivial computation.

According to the prime number theorem, there are $\Theta(x/\log x)$ primes $\leq x$. Therefore, we can try large random numbers as possible p and q and test these for primality. There are efficient randomized prime tests, using $O(\log p)$ multiplications mod p to test p, for this purpose.

Once p and q are known, choose e relatively prime to $\varphi = (p-1)(q-1)$. It can be shown that $\Omega(N/\log\log N)$ of the possible $e \leq \varphi$ will have this property, so random guessing is a good strategy here as well.

Now, solve the equation $xe + y\varphi = 1$ in integers and set $d = x$. This can be done by the extended Euclidean algorithm. By construction $de \equiv 1 \bmod \varphi$.

Randomized primality testing

It is a remarkable fact that one can get reliable evidence of the primality of a large number without knowing anything at all about its prime factors.

We will describe a randomized primality test that uses computation equivalent to one exponentiation, and has the following properties. If the input p is prime, the algorithm always says "prime." If the input p is not prime, the algorithm, with probability at least 3/4, says "not prime."

Testing p for primality

We assume that p is odd and at least 3.

Choose a *base* a at random from $1 \leq a \leq p-1$, and let $p - 1 = m \cdot 2^k$, where m is odd.

Compute the sequence $x_0, x_1, \ldots, x_k$, where

$$x_k = a^{m2^k} \bmod n.$$

Note that each x_i can be obtained from the previous one by squaring.

If the sequence ends in a 1, and each 1 in the sequence is preceded by either a 1 or a -1, the algorithm says "prime."

Otherwise, the algorithm says "not prime."

This algorithm uses two necessary conditions for primality:

Fermat's theorem: If $\gcd(a, p) = 1$ then $a^{p-1} \equiv 1 \bmod p$. This was proved in Lecture 17.

No funny square roots of 1: If $x^2 \equiv 1 \bmod p$, then $x \equiv \pm 1$. This is easy to verify. If

$$p \mid x^2 - 1$$

then

$$p \mid (x-1)(x+1).$$

Since p is prime, it must divide either $x+1$ or $x-1$. That is, we must have $x \equiv \pm 1 \bmod p$.

It needs to be verified that the algorithm is unlikely to say "prime" if p is not prime. This is certainly plausible, but it has to be proved. Cormen, Leiserson, and Rivest give a nice proof that the error probability is bounded by 1/2. See Rabin's paper (J. Number Theory, 12, 1980, 128-138) for the improved estimate cited above.

The security of RSA

Good reference: Dan Boneh, Twenty Years of Attacks on the RSA Cryptosystem, Notices of the AMS, January 1999.

The system is definitely insecure if bad guys can factor N. How likely is this?

It is possible (with extraordinary effort) to factor numbers in the 130-140 decimal digit range. [Need a reference for RSA-129.]

The best known factoring algorithm, the *number field sieve*, is believed to use

$$\leq \exp(c(\log N)^{1/3}(\log\log N)^{2/3})$$

single-word operations to factor N. (The best guess is that c is around 2.)

Most cryptanalytic strategies have work equivalent to the factorization of N. (See the RSA paper.)

The existence of some other, faster, strategy for decrypting individual messages has never been ruled out.

Paul Kocher (recent CRYPTO proceedings) has published some interesting attacks on RSA based on physical observations of the encryption device.

There are also a number of ways in which RSA can be used badly. For example, one should not encrypt the same message with the same modulus. The exercises in Stinson (p. 158 ff.) list a bunch of these protocol failures.

Notes

Independently, and before RSA, the idea of using exponentiation mod N in public-key cryptography was invented by Clifford Cocks of GCHQ. As with Ellis's work, this was only declassified recently.

Lecture 37

This lecture: Discrete logarithms, Diffie-Hellman key exchange

Reference: Diffie and Hellman, IEEE-IT 1977. (A classic paper.)

Discrete logarithms

Let p be a prime. Recall

$\mathbf{Z}_p = \{0, 1, \ldots, p-1\}$ (residue class representatives mod p)

$\mathbf{Z}_p^* = \{1, \ldots, p-1\}$ (elements of $\mathbf{Z}_p$ that are prime to p)

It is a fact that all elements of $\mathbf{Z}_p^*$ are powers of some particular element g. This element g is called a *generator* or *primitive root.*

We will not prove this here. One way to do it is to carefully count the solutions to $a^{(p-1)/d} = 1$ for various d. The existence of generators goes back to Gauss.

Example: $g = 3$ is a primitive root for $p = 7$. To verify this, enumerate the powers of g:

x	1	2	3	4	5	6
g^x	3	2	6	4	5	1

The generator g is not unique. In fact, whenever $\gcd(x, p-1) = 1$ the element g^x is also a generator.

The power x making $g^x = a$ in $\mathbf{Z}_p^*$ is called the *discrete logarithm* of a. Just as with ordinary logarithms, its value depends on the base g.

Using the square-and-multiply algorithm, it is easy to find g^x if we are given x. Obtaining x from g^x is, at this point in time, a difficult problem.

The mapping $x \mapsto g^x$ is a permutation of $\{1, \ldots, p-1\}$. It is believed to be an example of a *one-way function*: something that is easy to compute whose inverse isn't.

Computing discrete logarithms

Suppose we have an element $a \in \mathbf{Z}_p^*$, as well as p and a generator g. We know that there is some x making $g^x = a$. How do we find it?

Try all powers of g until a appears

Cost: $O(p)$ multiplications in $\mathbf{Z}_p$.

Use the baby-step/giant-step algorithm

We have seen this as a method for solving $ax = b \bmod N$. The same idea works for solving $g^x = a$.

Let $m = \lceil p^{1/2} \rceil$

Look for $x = x_0 + x_1 m$ as follows. Generate all

$$g^{x_0}, \qquad x_0 = 0, \ldots, m-1.$$

Generate all

$$a(g^{-m})^{x_1}, \qquad x_1 = 0, \ldots, m-1.$$

Sort and look for a match.

Cost: about $O(\sqrt{p})$ operations.

By applying the Chinese remainder theorem to the exponent x, one can reduce the cost of this procedure to about $O(\sqrt{q})$, where q is the largest prime divisor of $p-1$. This is called the Pohlig-Hellman algorithm.

Use ideas from factoring algorithms

It's a remarkable fact that the ideas used to factor a number can also be put to use to compute discrete logarithms modulo a prime of the same length. Nobody knows why this is.

Best current algorithm is called the *number field sieve*. It uses about

$$\exp(c(\log p)^{1/3}(\log\log p)^{2/3})$$

operations, with c around 2.

Diffie-Hellman key exchange

We are now beginning the study of *protocols*. You can think of a protocol as a multi-party procedure (usually the parties involved are computers) to accomplish some goal.

A (Alice) and B (Bob) want to communicate secretly over a public network without prior arrangement. This can be done provided they have a protocol to produce a common piece of secret information, which can be used as a key for a conventional cryptographic system.

The management of the network makes public a prime p and a generator g.

A chooses a random x with $0 \le x < p-1$ and sends $g^x \bmod p$ to B.

B chooses a random y with $0 \le y < p-1$ and sends $g^y \bmod p$ to A.

A computes $k_A = (g^y)^x \bmod p$.

B computes $k_B = (g^x)^y \bmod p$.

Now, $k_A = g^{yx} = g^{xy} = k_B$, so this key is shared.

How secure is this?

A cryptanalyst listening to the messages between A and B could obtain x by computing the discrete logarithm of g^x, and then simulating A to obtain k_A.

Therefore, the system is insecure if computing discrete logarithms is easy.

Note, however, that the cryptanalyst's real job is to implement the mapping

$$g^x, g^y \mapsto g^{xy}.$$

This can be thought of as multiplication in a funny representation of the integers mod $p-1$. Nowadays this is called the *Diffie-Hellman* problem. The status of

this is similar to that of computing e-th roots in the RSA system: there is an obvious way to try to do it (discrete logs in one case, and factoring in the other), but nobody can prove that this is the only way.

Notes

The DH protocol was invented independently by Malcolm Williamson of the UK. (See Singh's book.) This was classified work and kept secret until quite recently. His protocol differs slightly from DH: B sends g^x to A, and A responds with $g^y, g^{xy}+m$. B can then subtract to get the key m. (Unlike standard DH, this has the property that any m-bit vector can be the key.) [Source: Talk at DIMACS Workshop on Unusual Applications of Number Theory, Jan 2000.]

Lecture 38

This lecture: More about Diffie-Hellman key exchange

Review of last lecture

Let p be a prime and g a generator of $\mathbf{Z}_p^*$. By this we mean that every element of $\mathbf{Z}_p^*$ is a power of g reduced mod p.

If $a = g^x$ then x is the *discrete logarithm* of a. This is believed to be hard to compute.

Key exchange algorithm: A chooses a random exponent x and sends g^x to B. B does the same with a random exponent y. A and B can now both compute

$$k = (g^x)^y = (g^y)^x$$

from the information they have.

The best known attack on this is to solve the discrete logarithm problem, say by computing x from g^x and then simulating A.

Cost of this attack: about $\sqrt{q}$ operations if q is the largest prime divisor of $p-1$ (Pohlig-Hellman) or about the cost of factoring a number of the same length as p (number field sieve).

How do we decide if an element is a primitive root?

The best known method uses the factorization of $p-1$.

Theorem: an element $g \in \mathbf{Z}_p^*$ is a generator iff

$$g^{(p-1)/q} \not\equiv 1 \bmod p$$

for each prime divisor q of $p-1$.

Proof: By Fermat's theorem, $g^{p-1} \equiv 1$. Hence there will be some e for which $g^e \equiv 1$. This is called the *order* of g mod p. We observe that this must be a divisor of $p-1$, for if it is not, we have

$$g^{p-1 \bmod e} = g^{p-1-me} = 1,$$

and since $p - 1 \bmod e < e$, the order e was not minimal. Furthermore, the elements

$$g, g^2, \ldots, g^{e-1}$$

are unique mod p, since if we had $g^x \equiv g^y$ with $x < y$, we would have $g^{y-x} \equiv 1$ with $y - x < e$. It is easy to see now that generators are exactly the elements of order $p-1$, and if the order of g is a proper divisor of $p-1$, there must be some q making $g^{(p-1)/q} \equiv 1$.

Example: Last time we enumerated the powers of 3 to show that this is a generator mod 7. We now use the theorem. We factor $p-1 = 6 = 2 \cdot 3$, and compute

$$3^{(7-1)/2} = 3^3 = 27 \equiv 6 \pmod 7,$$

$$3^{(7-1)/3} = 3^2 = 9 \equiv 2 \pmod 7.$$

This shows that 3 is a generator mod 7.

Choosing p and g

In practice, taking p and then factoring $p-1$ is not an appealing prospect. How do we get around this difficulty?

One way is to cook up p in such a way that the factorization of $p-1$ is known.

We could take $p = 2q+1$. These are called *Sophie Germain primes*, after a mathematician who worked on Fermat's last theorem in the 19th century. [Which is the SG prime exactly – is it p or q?] Although a good practical idea, it is not yet known if there are infinitely many of these.

We could choose $p-1$ in factored form and then test p for primality. This can be done in polynomial time. (See E. Bach, SIAM J. Computing, 1988.)

Another approach is to give up on the idea that g be a generator, and use instead an element of high order q.

To set up the system, we would first choose a prime q. Then test numbers of the form $qr+1$ (of the appropriate size) until one is prime.

To find a g of order q, we choose h randomly from $\mathbf{Z}_p^*$. Set $g = h^{(p-1)/q}$. If

$$h \not\equiv 1 \pmod p$$

We know that g has order q, since

$$g^q = \left(h^{(p-1)/q}\right)^q = h^{(p-1)} \equiv 1.$$

The chance of finding a suitable g is $1 - 1/q$ which is near 1 if q is large.

The key exchange protocol is the same, except that A and B must choose random exponents satisfying $0 \le x, y < q$.

KEA (Key Exchange Algorithm)

This is a protocol designed by the US government, to produce a key that can be used with the block cipher Skipjack. This was originally classfied (why? there isn't much new in it) but made public in 1998.

This uses an 160-bit prime q, a 1024-bit prime p, and an element g of order q.

Since $\sqrt{q} = 2^{80}$, the security against a discrete logarithm attack is comparable to exhaustive search on Skipjack's 80 bit key.

We note that $2^{1024} \doteq 10^{308}$, so discrete logarithm attacks based on ideas from factoring are unlikely to be successful. The current record for factoring is around 10^{140}.

The original Diffie-Hellman protocol allows a malicious user to force a weak key.

Simple example: suppose A chooses $x = 0$. Then $g^{xy} = 1^y = 1$.

More subtly, A can choose x to be zero modulo some divisor of $p - 1$. Then the key generated by the protocol is restricted to a much smaller set.

To get around this (I think – the documentation is not clear on this point), KEA uses a more complicated protocol.

The elements g, q, p are public.

A fixes a private key x_A and publicizes his public key g^{x_A}.

B fixes a private key x_B and publicizes his public key g^{x_B}.

When A and B wish to communicate, A generates a secret random number r_A and sends g^{r_A} to B. B responds in kind, using a secret random number r_B and sending g^{r_B} to A.

Now A has enough information to compute

$$w = g^{x_B r_A} + g^{r_B x_A} \bmod p,$$

which is the same as the value

$$w = g^{x_A r_B} + g^{r_A x_B} \bmod p$$

computed by B.

In practice, various checks are applied. The public keys should be checked to have order q. Also, if $w = 0$ the protocol fails and must be retried.

KEA specifies that q be 160 bits. This makes $\sqrt{q}$, which is the essentially the complexity of the Pohlig-Hellman discrete logarithm algorithm, 80 bits long. The prime p is 1024 bits, which makes p about 10^{308}.

In the final stage of the protocol, w is processed into an 80 bit key in a way that we will not go into here. The length of this key undoubtedly influenced the size of q. (Exhaustive search though 2^{80} keys has roughly the same complexity as a $\sqrt{q}$ discrete log algorithm.) One presumes the length of p was chosen to put attacks related to factoring out of reach.

Possible reference: Goss, US Patent 4 956 963. See also the MTI/A0 protocol on p. 518 of Menezes, van Oorschot, and Vanstone, Handbook of Applied Cryptography.

Lecture 39

This lecture: Password encryption, authentication

Up to now we have focused on cryptography as a technique for confidential message transmission. This is by no means the only application. Along with confidentiality, we may wish to have

Authenticity (who is the source of a piece of information)?

Integrity (is the data the same as when it was created)?

Commitment (could the creator of some information repudiate it?)

All of these come up in ordinary business dealings, for which there are long-standing customs to prevent malfeasance. Let's look at a few examples:

The user of a credit card must prove that he or she is the true owner of that account. (Honored more in the breach than the observance.)

Before a will is executed, it must be determined that the document has not been modified, say, to some beneficiary's advantage.

An order to buy cannot be retracted unilaterally.

Password encryption

A familiar problem is that of logging into a computer system.

Early on this was done by making a file of passwords. This presents an obvious target, since anyone who has access to the password file can impersonate any desired user.

Needham (and others?) suggested that the passwords be encrypted using a publicly known encryption algorithm.

What is needed here is not the full apparatus of cryptography, but simply a *one-way function.* This is a function e for which $e(x)$ can be easily computed for any x, but such that recovery of x from $e(x)$ is infeasible. Note that the concept of key does not appear.

Here are some examples of one-way functions:

Choose a prime p and a generator g of $\mathbf{Z}_p^*$. Let $e(x) = g^x \bmod p$. Inverting e is the same as computing a discrete logarithm.

[Purdy]. Using about $\log d$ arithmetic operations, one can evaluate polynomials of degree d. For example,

$$e(x) = (((x+1)^2 + 5)^2 + 3)^2 + 7$$

has degree 8, although only 3 multiplications are used. Since any function from $\mathbf{Z}_p$ to $\mathbf{Z}_p$ is a polynomial, it is possible to make a "random looking" function e simply by doing enough arithmetic. [Note: for this to be secure one needs to know that there are no good algorithms to find roots of polynomials defined using recursions. This should be looked into.]

Use a conventional block cipher algorithm for e. The original Unix operating system used a variant of the Enigma machine (1 rotor, 256 letters) for this purpose. Later the algorithm was changed to a variant of DES.

Needham's method is subject to a dictionary attack.

In practice, users tend to choose passwords that are easy to remember. (Therefore, in some sense, the process of password selection has extremely low entropy. Could this be somehow estimated?)

An attacker can simply make a list of all words, word pairs, short phrases, etc. and encrypt them using e. This list can be computed off-line and compared against the password file.

The next step in the arms race was *salting*.

Basic idea: Make each user's password encryption algorithm slightly different. The password entry for a user now contains

$$e(x, s), s$$

where s is a random string, called the *salt*. Note that $e(x, s)$ can now be thought of as a keyed encryption function, that is,

$$e(x, s) = e_s(x).$$

This does not defend against a determined attack on a single password but makes a dictionary attacker who hopes to find one insecure password in the file do a lot more work. If the salt has t bits, the attacker now has to compute 2^t encryptions of each item in the dictionary.

In practice, it is not good to use an off-the-shelf keyed encryption algorithm. (Why? Any popular encryption algorithm will have fast implementations available, which the attacker can use.) Some versions of Unix, for example, use a 12-bit salt to customize the 32 to 48 bit expansion function inside DES. (Details?) This prevents an attacker from using commercially available DES chips.

Challenge-response protocols

Password-based authentication schemes all share various weaknesses.

Passwords can be guessed.

Passwords can be captured and replayed.

For this reason, we might try to improve our authentication methods by making the claimant do some computation. We have now gone away from the naive idea that proof of identity is a piece of information, and now think of proof of identity as the ability to do something.

Here's a simple three-message protocol for this purpose, using public key encryption.

This means that each user A has a public encryption function e_A and a private decryption function d_A. They must satisfy $d_A(e_A(x)) = x$.

As an example of this, we could use RSA. Then A chooses $n = pq$ and $de \equiv 1 \pmod{\varphi(n)}$. The pair (e, n) is publicized, and A keeps d secret.

Suppose A wishes to gain access to the host computer B. Here is what happens.

1. A sends a message to B asking to gain access.
2. B chooses a random message r and sends it to A.
3. A sends $d_A(r)$ back to B, who checks that $e_A(d_A(r)) = r$.

There is also a private key version.

Suppose that A and B share a key k for a conventional cipher. Let the encryption and decryption functions be e_k and d_k, respectively.

In response to a request for access, B sends a random message r to A. A then computes $e_k(r)$ and sends it back to B. B compares this with his own computation of the same quantity.

Both of these protocols suffer from a weakness: someone impersonating the host has a way of getting random messages encrypted with A's private key. For a critique and further protocols, see Schneier, Applied Cryptography, pp. 54 ff.

Notes.

Details of DES-like password salting on Unix (from Mark Wooding, sci.crypt 6/12/00): "The salt is a 12-bit number, which is used to define a bit permutation applied immediately after the standard expansion permutation E in the DES round function. Each salt bit is associated with two bits from the 48-bit output of E: if the salt bit is set then the bits are swapped."

Challenge-response protocols are related to IFF (identify friend or foe) schemes. These should be looked into.

Reference for Purdy: G. P. Purdy, A high security log-in procedure, Comm. ACM, v. 17, 1974, pp. 442-445.

Lecture 40

This lecture: Authentication in networks, Kerberos

References: Needham and Schroeder, CACM Dec. 1978; ???, Usenix Winter 1988.

Technical background

In mainframe computing, the boundary between the computer and the outside world was well defined. Typically, a bus or small network linked the various parts of the computer (processor, disks, printers, ...) and users communicated with the computer through a limited number of ports. The ports were the obvious place to enforce password protection.

Networked computing, on the other hand, involves many separate units sharing a network. There is no obvious boundary between the computer and the user, indeed, it is not clear what exactly should be counted as the computer.

A useful division that has caught on:

Clients (workstations, user programs, etc.)

Servers (which provide file storage, printing, etc.)

Servers are a natural place to put authentication boundaries, as they get requests from many different users. So the problem we must solve is this: A client requests some action from a server. The server must examine the client's credentials and decide if the action should take place.

A naive solution to this is to provide a password for every possible client and every possible service. This rapidly becomes an administrative nightmare as the size of the system gets large.

Kerberos

Needham and Schroeder proposed various schemes whereby a trusted authority, the key distribution center (KDC) could be used to generate temporary keys for client-server interactions.

Based in large part upon their ideas, MIT's Athena project implemented Kerberos, a security and authentication service for networks.

The most basic protocol allows users U and V to generate a session key. This is done in cooperation with the trusted center T. The pattern of messages is:

$$U \rightarrow T \rightarrow U \rightarrow V \rightarrow U$$

After receiving a message, each party in the chain does some computation and constructs new messages.

We'll assume the system has been set up so that each user has a name (user ID) and a key for communicating with T. In practice this means that T has to be physically

secure. A common secret-key encryption like DES is used by all parties, and keys for communication with T are derived from user-generated passwords.

The basic protocol

Our goal is to make a session key for U and V.

U to T (in clear): ID's of U and V, together with a request for a session key.

T selects a session key S, a timestamp t, and a lifetime ℓ. These are encrypted to make two messages:

$$m_1 = e_U(S, \mathrm{ID}_V, t, \ell)$$

$$m_2 = e_V(S, \mathrm{ID}_U, t, \ell)$$

T now sends m_1 and m_2 to U.

U decrypts m_1 to get $S, \mathrm{ID}_V, t, \ell$. The message m_2 is passed on to V, along with

$$m_3 = e_S(t, \ell)$$

V decrypts m_2 and gets $S, \mathrm{ID}_U, t, \ell$. This provides the session key, so V can go on to decrypt m_3 and get another copy of t, ℓ. If these do not match those obtained from m_2, the protocol fails. V now makes

$$m_4 = e_S(t + 1)$$

and sends this to U.

U decrypts m_4 and checks that this equals the timestamp plus 1.

Kerberecito

One of the problems with a complicated setup like Kerberos is that it is rather hard to do examples. (DES encryption by hand is not a pleasant task.) To allow us to understand the mechanics (who sends what to whom) we'll modify the system so that it uses Vigenère encryption. This provides no real security but allows one to see what is going on.

Data format conventions

User ID's go from 00 to 99.

Keys are 2 decimal digits.

Time stamps are numbers from 0 to 99.

Lifetimes are numbers from 0 to 9.

The players

U is a user (client) who wishes to communicate with another user V (server).

T is a trusted authority

We will give U the user ID 05 and assume that the key 15 is used for communication with T. Similarly, V is user 06 and communicates with T using key 19.

Message from U to T:

"I am number 5. Give me a key to communicate with number 6."

T's computation

T generates the session key 34, the time stamp 20, and the lifetime 5.

T makes m_1 by encrypting (34 06 20 5) with U's key 15. We have

$$\begin{array}{rcccc} & 34 & 06 & 20 & 5 \\ + & 15 & 15 & 15 & 1 \\ = & 49 & 11 & 35 & 6 \end{array}$$

So m_1 is (49 11 35 6). Note that we don't do any carries, all the arithmetic is mod 10.

Similarly the encryption of (34 05 20 5) w/ V's key 19 gives

$$\begin{array}{rcccc} & 34 & 05 & 20 & 5 \\ + & 19 & 19 & 19 & 1 \\ = & 43 & 14 & 39 & 6 \end{array}$$

so m_2 is (43 14 39 6).

T now sends m_1 and m_2 to U.

U's computation.

Decryption of m_1 using U's key:

$$\begin{array}{rcccc} & 49 & 11 & 35 & 6 \\ - & 15 & 15 & 15 & 1 \\ = & 34 & 06 & 20 & 5 \end{array}$$

Note that U now has the session key 34.

U now uses the session key to make m_3:

$$\begin{array}{rcc} & 05 & 20 \\ + & 34 & 34 \\ = & 39 & 54 \end{array}$$

So m_3 is (39 54).

U now sends m_2 and m_3 to V.

V's computation

The message m_2 is decrypted using V's key:

$$\begin{array}{rcccc} & 43 & 14 & 39 & 6 \\ - & 19 & 19 & 19 & 1 \\ = & 34 & 05 & 20 & 5 \end{array}$$

Now V has the session key 34, and can use it to decrypt the second message:

$$\begin{array}{rrr} & 39 & 54 \\ - & 34 & 34 \\ = & 05 & 20 \end{array}$$

This matches the middle two fields of m_2, which came from T.

The time stamp is incremented, and this is encrypted using the session key to make m_4:

$$\begin{array}{rr} & 21 \\ + & 34 \\ = & 55 \end{array}$$

Thus, m_4 is 55.

The message m_4 now is sent to U.

U's computation:

Decrypt m_4 using the session key:

$$\begin{array}{rr} & 55 \\ - & 34 \\ = & 21 \end{array}$$

This is one more than the timestamp, so we are done.

Lecture 41

This lecture: Digital signatures a la RSA

Reference: Rivest, Shamir, Adleman, CACM v. 21, 1978, pp. 120-126. See also Stinson section 6.1.

Digital signatures in general

The goal is to make the digital analog of a signed document. What should the signature provide?

1) Assurance that only the alleged signer made the signature.

2) Assurance that the document was not changed after signing.

3) Assurance that the document is not a copy that is being reused.

Public key techniques can provide 1) but only for short messages. We can enforce 2) by the use of *cryptographic hash functions* (more about this later). Property 3) is enforced by including a time stamp or serial number in the document.

In a real situation, there are two parts to a signature scheme: compression and signing. Compression is necessary because most documents are much longer than the thousand or so bits that make up one block of an RSA-encrypted message. We will ignore compression for the moment, and assume that only short messages are to be signed.

Today's lecture focuses on signing and verification. These techniques usually use public key cryptography. There are mechanisms for using secret-key block ciphers (e.g. DES) to sign documents, but these require a trusted third party, who then becomes a bottleneck. (See Schneier p. 35 for an example.)

The basic RSA signature scheme

The basic idea is to use the RSA system in reverse. Recall that the RSA message transmission scheme involves encryption (using a public function) then decryption (using a private function). In digital signatures, the order of the public and private functions is reversed. Put another way, a message is signed using a private encryption function, then checked using the corresponding public decryption function.

RSA mechanics

The signer chooses $n = pq$ with p and q secret prime factors as before. He also chooses exponents d and e with $de \equiv 1 \bmod (p-1)(q-1)$. He makes public the pair (d, n).

To sign the message x, the signer computes $y = x^e \bmod n$. The pair (x, y) is a signed message.

To verify a signed message (x, y), a verifier checks that

$$y^d = x \bmod n.$$

This works because

$$y^d = (x^e)^d \equiv x \bmod n.$$

Potential problems

Anyone in possession of a signed message (x, y) can produce more message-signature pairs. Since the verifier only checks that one component is a power of the other mod n, any pair of the form (x^r, y^r) is also a valid signature. What saves us is that the first component x^r is unlikely to be a meaningful message. (Note the contrast with traditional cryptography: low entropy is good, even essential here.)

Reneging. The signer could publish the factors of n, say, in a newspaper advertisement, and then claim that someone else made the pair (x, y). To prevent this, the verifier must have proof that the message originated by a certain time. This requires timestamping.

How do we know that the signer's public key is really (n, e)? This could be done using a trusted phone book (we are now dragging in the third party again) or by meeting once and for all before any messages are signed. The second would work for two companies that plan to do a lot of business, but is probably too much to ask of individuals.

Encrypting signed messages.

So far we have seen RSA used as a cryptographic system for message transmission, and as a signature scheme. It is possible to combine both.

Suppose we have two users A and B.

A now has a modulus $n = pq$ as before, and two keys d, e satisfying $de \equiv 1 \bmod (p-1)(q-1)$. He publicizes his modulus n, and his public key e.

A' does the same thing, with $n' = p'q'$ and d', e'.

Suppose A wants to send x to A'. He computes

$$y = (x^{e'} \bmod n')^d \bmod n,$$

which we should think of as A's signature applied to a message encrypted with the public key belonging to A'.

To read and verify this message, A' would like to compute

$$z = y^e \bmod n = x^{e'} \bmod n' \qquad (*)$$

and then apply the private exponent d' to z to get

$$z^{d'} \bmod n' = x^{e'd'} \bmod n' = x.$$

There is a problem with this scheme. Namely, in the course of applying the exponent d, we will compute, in effect,

$$(x^{e'} \bmod n') \bmod n.$$

There is no guarantee that reduction mod n' and then mod n will preserve the message. This is called the *reblocking* problem.

Ideas for reblocking

Reference: Menezes et al, Handbook of Applied Cryptography. See also Davies and Price, Security for Computer Networks.

You might think of forcing everyone to use the same n. The problem with this scheme is obvious: all users have access to the private factors p and q.

A better idea is make sure that $n' < n$. If we only wanted to send messages in one direction, this would work. Most communication networks are symmetric, however.

For a symmetric network, there are a couple of choices.

Each user chooses a signing modulus n' and an encryption modulus n, in such a way that all the n' are smaller than any n. For example, we could make the n' 200-digit moduli, and the n 200-digit moduli. This is a good solution if one can afford to double the storage for keys.

One could also choose moduli of a special form (very close, say, to a power of 2). One can thereby make the probability of failure acceptably small. [Need more details here.]

Lecture 42

This lecture: The El-Gamal digital signature method

Reference: T. ElGamal, IEEE Transactions on Information Theory, v. 31, 1985, pp. 469-472.

This is also important because many ideas from it were incorporated into DSS, the US government's digital signature standard. Stinson (p. ??) discusses the differences.

The El-Gamal signature scheme

Again g and p are public. We want the ability to sign documents m, which are in the range $0 \leq m < p-1$. (In practice, m would be obtained from a larger document by hashing.)

Each user chooses a secret a, and publicizes $y = g^a$.

A signature is a pair (r, s) with $g^m \equiv y^r r^s \bmod p$.

Using the square-and-multiply method, we can check a signature using $O(\log p)$ multiplications mod p.

To sign m, the signer S executes the following protocol:

Choose random $k \in \mathbf{Z}_{p-1}^*$

$r = g^k \bmod p$

$s = k^{-1}(m - ar) \bmod p - 1$.

This generates valid signatures. We note that

$$ar + ks \equiv m \bmod p - 1,$$

so

$$g^m = g^{ar+ks} = (g^a)^r (g^k)^s = y^r r^s..$$

Clearly, one can forge signatures by solving the discrete log problem.

Why is this? The number r has been chosen to be a generator for $\mathbf{Z}_p^*$. So an attacker wishing to make a signature for another message m' need only solve

$$r^s = g^{m'} y^{-r}$$

mod p. Indeed, he doesn't need to use an r from a genuine signature, he can make his own.

Any $k \in \mathbf{Z}_{p-1}^*$ will produce a valid signature. It is important that k not be re-used, as this allows an attacker to find a.

See Stinson on this point.

An example

We'll take $p = 13$ and $g = 2$. Obviously numbers of this size offer no security, the main point is to understand everything.

Here are the powers of 2 mod 13.

$x =$	0	1	2	3	4	5	6	7	8	9	10	11
$2^x =$	1	2	4	8	3	6	12	11	9	5	10	7

The invertible elements in $\mathbf{Z}_{12}$ are 1,5,7,11. Suppose the signer chooses $a = 11$. This makes

$$y = 2^{11} = 7.$$

Signing the message $m = 10$: Let's take $k = 5$. Then

$$r = 2^5 = 6.$$

Also,

$$s = 5^{-1}(10 - 11 \cdot 6) = 8(-56) = 7.$$

So a signature of $m = 10$ is $(6, 7)$.

Let us check that the signature equation holds.

$$g^m = 2^{10} = 10.$$

We compare this to

$$y^r r^s = 7^6 6^7 = 10$$

which shows that the signature is valid.

It is interesting to note that a table of discrete logarithms (as we have above) can be used to aid calculations. In fact, discrete logs were tabulated in the pre-computer days just for this purpose. (See, e.g. C.J.J. Jacobi, Canon Arithmeticus) Suppose we wanted to check that

$$2^{10} \equiv 7^6 6^7 \pmod{13}$$

This holds iff

$$10 \equiv 6 \log 7 + 7 \log 6 \pmod{12},$$

where log now means discrete log base 2. Looking these up in our table, this becomes

$$10 \equiv 6 \cdot 11 + 7 \cdot 5 = 101,$$

which is true.

Lecture 43

This lecture: Cryptographic hash functions

Reference: Stinson, chapter 7. See especially 7.2 and 7.7.

What is a cryptographic hash function?

Digital signature schemes (RSA, ElGamal, etc.) apply relatively expensive operations, such as exponentiation, to messages. For this reason we'd like to compress the message before signing it. Compression algorithms for this purpose are called *secure* or *cryptographic* hash functions.

Suppose h is to be used as a cryptographic hash function. What properties should h have?

(1) The function h should be easy to compute.

(2) Non-invertibility: Given y, it should be hard to find an x for which $h(x) = y$.

(3) Collision avoidance: Given x, it should be hard to find an $x' \neq x$ such that $h(x') = h(x)$.

(4) It should be hard to find $x \neq x'$ with $h(x) = h(x')$.

We will refer to x as a *message* and $h(x)$ as a *message digest.*

Relations among our requirements

One naturally wonders how the requirements (1)–(4) are related. We will prove one result along these lines.

If inversion of h is easy, then it is easy to find two messages with the same message digest.

Proof. Let $h : X \to Y$. We can without loss of generality assume that h maps onto Y. For a given x, let C be the set of all elements x' with $h(x') = h(x)$. This gives a partition of X into equivalence classes. (Some people call the classes "fibers.")

We choose x uniformly from X and invert $h(x)$ to get x'. Then

$$\Pr[x \neq x' | x \in C] = (1 - 1/c),$$

where c is the size of C. So

$$\begin{aligned}
\Pr[x \neq x'] &= \sum_C \Pr[x \neq x' | x \in C] \Pr[x \in C] \\
&= \sum_C (1 - 1/c) \frac{c}{|X|} \\
&= \sum_C \frac{c}{|X|} + \sum_C 1 \\
&= 1 - |Y|/|X|.
\end{aligned}$$

This is $\geq 1/2$, provided that $|Y| \leq |X|/2$.

We can think of this result as applying to any compression function that decreases the message size by one bit.

[It should be possible to generalize this to infinite sets X – the natural case! Also there may be a way to say this in information theoretic terms, i.e. any invertible compression function that loses entropy is not collision-free.]

Finite automata

Most cryptographic hash functions are implemented as finite-state automata.

Crash course on automata theory:

A deterministic finite automaton has a finite set Q of states, a set Σ of input symbols, and an initial state q_0. The action of the machine is given by a transition function

$$\delta : Q \times \Sigma \to Q.$$

This means that if the machine is in state q and reads the input symbol x, it goes to the state $\delta(q, x)$.

Message compression via finite automata

We break the message x into blocks $x_0 x_1, x_n$. The automaton is started in state q_0. Then the input block x_0 is read, leaving the machine in $q_1 = \delta(q_0, m_0)$. Similarly, we let

$$q_i = \delta(q_{i-1}, m_{i-1}).$$

The hash value is the last state q_m.

How can this possibly be secure?

An opponent would like to be able to find a message x that produces a given message digest. That is, an input string x whose processing by the automaton leaves it in a specified state. If you have taken automata theory, you probably know an algorithm for this problem. The catch is that the running time of the algorithm depends on the number of states of the automaton. For a cryptographically interesting automaton, this state space is huge.

MD4: An example of cryptographic hashing.

MD4 (the MD stands for "message digest") was a key example in the evolution of cryptographic hash function design.

This was invented by Ron Rivest, and has since spawned off MD5, on which was based the US government's SHA or SHA-1 algorithms.

The state space Q is $(\mathbf{Z}_{2^{32}})^4$. That is, the state maintained by MD4 is a 4-tuple of 32-bit integers.

Message blocks are 512 bits long. That is, $\Sigma = \mathbf{Z}_2{}^{512}$.

The transition function is of the form

$$\delta(m, x) = F(m, n) + x$$

where the + indicates vector addition on elements of $\mathbf{Z}_{32}^4$ The function F is itself a cascade:

$$F(m, x) = F_3(m, F_2(m, F_1(m, x))).$$

Define three Boolean functions by

$$f_1 = xy + \bar{x}z$$
$$f_2 = xy + yz + xz$$
$$f_3 = x \oplus y \oplus z$$

(We're using logic design notation here: + is OR, $\times$ is AND, the bar indicates negation, and $\oplus$ is XOR.) Inspection of the truth tables reveals that each f_i maps half of its inputs to 0 and half to 1. We will extend f_i to a function on bit strings as follows:

$$f_i(x_1 \cdots x_m) = f_i(x_1) \cdots f_i(x_m).$$

[Is there a name for this construction?] For this extended function, all outputs are equally likely.

Finally, we let the binary operator <<< denote circular shift to the left.

The function F_1 is defined by a sequence of 16 instructions. Let the state be A, B, C, D. Let the message be divided into 32-bit blocks $m_0 \ldots m_{31}$. The first four instructions are

$$A \leftarrow (A + f_1(B, C, D) + m_0) <<< 3$$
$$D \leftarrow (D + f_1(A, B, C) + m_1) <<< 7$$
$$D \leftarrow (C + f_1(D, A, B) + m_2) <<< 11$$
$$B \leftarrow (B + f_1(C, D, A) + m_3) <<< 17$$

(Here + denotes addition mod 2^{32}, that is, 2's complement addition as provided by an arithmetic processor.) This is followed by the same thing with $m_4 - m_7$, with $m_8 - m_{11}$, and with $m_{12} - m_{15}$.

The other functions F_2 and F_3 are similar, using f_2 and f_3. There is, however, a different schedule for extraction of the message blocks, and there are constants added in before the circular shifts. Complete descriptions of these functions can be be found in Stinson, section 7.7.