Cryptography

Notes for Computer Security
Also part of exam revision

Contents

A wrap up of all the reading assigned from Lecture 7 - Cryptography onwards, discussing cryptographic schemes.

Symmetric Cryptography

(Source)

The concept of cryptography began as a way for two parties (typically *Alice* and *Bob*) to communicate securely even if their messages may be read by an eavesdropper (*Eve*).

In Symmetric Cryptography the same *key* is used for both encryption and decryption. The reccomended symmetric encryption algorithm is AES.

Cryptosystem attacks

(Source)

When discussing attacks on cryptosystems we assume the attacker *knows* the algorithms for Encryption and Decryption, but *does not know* **anything** about the keys being used in it. This follows the open design principle, also referred to as *Kerckhoff's Principle*.

Types of cryptosystem attacks

There are *four primary types* of attacks (listed in order by the amount of information an attacker can access when performing them) an adversary can attempt

Ciphertext-only attack

In this attack, the attacker has access to the ciphertext of one or more messages, all encrypted using the same *key **K***. Their goal is to determine the plaintext for one (or more) of these ciphertexts (or better yet, discover ***K***).

Known-plaintext attack

In this attack, the attacker has access to one or more *plaintext-ciphertext pairs* such that each plaintext was encrypted using the same *key **K***. Their goal is to determine this *key **K***.

Chosen-plaintext attack

In this attack, the attacker can choose one or more *plaintext messages* and *get the ciphertext associated* with each one (based on use of the same *key **K***). There are two variations of this attack:

- In the ***offline chosen-plaintext attack***, the attacker must choose all plaintexts *in advance of recieving any ciphertexts*

- In the ***adaptive chosen-plaintext attack***, the attacker can choose plaintexts *in an interative manner, where the subsequent plaintext can be based on the information gained from previous ciphertexts*

Chosen-ciphertext attack

This attack is similar to the Chosen-plaintext attack but simply using *ciphertext* and the *decryption algorithm* instead. It also has an ***offline*** and ***adaptive*** variant.

Feasibility of these attacks

These attacks are feasible for a few reasons:

- It's usually quite easy to recognise a message is a valid plaintext. If you tried an attack and got the plaintext *NGGNPXNGQNJABAVEIVARORNPU*, you can immediately dismiss it. If you try another and get the plaintext *ATTACKATDAWNONIRVINEBEACH*, you can be confident you've found the *decryption key **K***.
- The ***unicity distance*** of a cryptosystem - *the minimum number of characters of ciphertext that are needed so that there is a single intelligble plaintext associated with it* - is usually much less (in characters) than the *key lengths* (in **bits**) of a cryptosystem translating to natural language.

Substitution Ciphers

(Source)

A generic substitution cipher, such as a *Ceasar cipher*, can be generalised so that each letter has an arbitrary substitution, which greatly increases the security of the cryptosystem. In English plaintexts there are $26!$ possible substitution ciphers (more than $4.03 * 10^{26}$ ciphers!)

A weakness of these ciphers (all $4.03 * 10^{26}$ of them) is ***frequency analysis*** - the fact that certain letters appear much more commonly than other letters (e.g *'e'* appears on average ~12% of the time in English).

Polygraphic Substitution Ciphers

In a *polygraphic substitution cipher*, groups of letters are encrypted together: e.g, a plaintext could be partitioned into strings of two - *digrams* - and each *digram* is substituted with a different one to create the ciphertext.
For example, in English there are 26^2 (*676*) possible *digrams*.
The problem with this is that we must note all the substitutions. One way to solve this is with a *substitution box* (or *S-box*); a two dimensional table where the first letter in a pair (for example) would specify a row, the second a column, and each entry would be the unique two-letter substitution.

Substitution box

If we take a b-bit word (x) and divide it into two words, y and z, consisting of the first c bits and the last d bits (of x), such that $b = c + d$, we could specify substitution to use for such a word x by using an S-box of dimensions $2^c * 2^d$.
As long as the substitutions specified in an S-box S are unique, there exists and inverse S-box S^{-1} that can be used to reverse the substitutions specified by S.

A 4-bit S-box (*in binary*)

	00	**01**	**10**	**11**
00	0011	0100	1111	0001
01	1010	0110	0101	1011
10	1110	1101	0100	0010
11	0111	0000	1001	1100

One-Time Pads

(Source)

This substitution concept can be applied to larger *blocks* of letters at a time. For example, the *Vignère cipher* is an example of a polygraphic substitution cipher applied to blocks of length m (since it amounts to repeatedly using m shift ciphers in parallel). A key to this cryptosystem is a sequence of m shift amounts $(k_1, k_2, \ldots, k_m)$ *modulo* the alphabet size (for English, 26).

- Given a block of m characters, we can encrypt by shifting the first character by k_1, the second by k_2, third by k_3, etc etc.
- Decryption is done by performing the reverse shifts on each block of m characters.
  This cipher is unfortunately broken using the same statistical techniques (such as frequency analysis) **as long as the cipher text is long enough relative to *m***.

One-Time Pads are a type of cipher that is unbreakable. We using a block of keys $(k_1, k_2, \ldots, k_m)$ to encrypt a plaintext M of length n (like with the Vignère cipher) but there are 2 key differences that make it immune to statistical analysis.

1. The length m of the block of keys has to be the same as n, the length of the plaintext.
2. Each shift amount k_i must be chosen completely at random
   As long as the *pads* are not revealed this encryption is uncrackable (in a *feasible* and *normal* amount of time). The term *"One-Time"* comes from the fact these pads cannot be reused as this allows for statistical analysis.

Binary One-Time Pads

There exists a binary version of the one-time pad using the *XOR* ($\oplus$) operation.
If we have our plaintext message M as a binary string of length n, and a pad P to be a *completely random binary string* of length n, we can produce the ciphertext C using the formula:

$$C = M \oplus P$$

Likewise, we can recover the plaintext from the ciphertext using the formula:

$$M = C \oplus P$$

Since *XOR* is associative, we have:

$$C \oplus P = (M \oplus P) \oplus P = M \oplus (P \oplus P) = M \oplus \vec{0} = M$$

where $\vec{0}$ is a vector of all zero bits.

Pseudo-Random Number Generators

Much of Cryptography relies on *randomness*. In computers, everything is determined by code so getting a *random* value is quite tricky. This is why many *pseudo-random* (as in, not really random, but quite unpredictable so it might as well be random) generators exist.

Linear Congruential Generator

Desirably, a random generator will generate numbers that are uniformly distributed. In an *LCG*, we start from a random number x_0 (the *seed*), and generate the next number in a sequence (x_{i+1}) from the previous number x_i according to:

$$x_{i+1} = (ax_i + b)\, mod\, n$$

We assume $a > 0, b \geq 0$ are chosen at random from the range $[0, n-1]$. If a and n are relatively prime$^{[1]}$ the sequence is *uniformly distributed*

$[1]$ - Two integers are relatively prime (or *coprime*) **when there are no common factors of them other than 1**.

Security of PRNG's

- It should be hard to predict x_{i+1} from previous numbers
    - For the LRG we can determine the values of a and b as soon as we have seen three consecutive numbers
- We are also concerned about a generator's ***period*** - the number of values output by the sequence before it starts to repeat
    - E.g if a is relatively prime to n, the period of the *LRG* is n.

AES can be used in a PRNG to encrypt, using a common random key, each number in a deterministic sequence that starts from a random seed. Breaking the predictability of such a sequence amounts to a type of ciphertext-only attack, where the adversary knows the associated plaintexts are taken from a known sequence. With a block size n, the *period* of this PRNG is 2^n.

AES

The **Advanced Encryption Standard (AES)** is a *block-cipher*[2] that operates on 128-bit blocks. It's designed to be used with keys that are 128, 192, or 256 bits long; yielding the ciphers *AES-128*, *AES-192*, and *AES-256* respectively. The block size **is always 128 bits, however.**

AES Rounds

Using the 128-bit version of AES as example, the AES algorithm proceeds in *ten rounds*. Each *round* performs an *invertible* transformation on a 128-bit array (called a *state*). The inital state X_0 is the *XOR* of the plaintext P with the key (128-bits in this example) K:

$$X_0 = P \oplus K$$

Round i $(i = 1, \ldots, 10)$ receives state X_{i-1} as input and produces state X_i.
The ciphertext C is the output of the final round: $C = X_{10}$.
Each round is built on four basic steps:

1. ***SubBytes step***: an S-box substitution step
2. ***ShiftRows step***: a permutation step
3. ***MixColumns step:*** a matrix multiplication (*Hill cipher*[3]) step
4. ***AddRoundKey step***: an *XOR* step with a *round key*, derived from the 128-bit encryption key
   (These steps are described in more detail in Details of AES)

Typical AES implementation

Software tends to optimise AES for speed of execution and uses *lookup tables* to implement the basic steps of each round. Using the 128-bit version of AES, this can be implemented in 8 tables, each mapping an input *byte* (8-bits) onto an output *int* (32-bits). Each of these lookup tables stores 256, 32-bit ints. These tables are precomputed and accessed using encryption/decryption.
A round of AES encryption/decryption is implemented by a combination of only three types of operations

- ***XOR* of two ints**: $y = x_1 \oplus x_2$, where x_1, x_2, and y are all *ints*.
- **Split of an int into 4 bytes**: $(y_1, y_2, y_3, y_4) = x$, where y_{1-4} are all *bytes* and x is an *int*.

- **Table lookup of an int, indexed by a byte**: $y = T[x]$, where y is an *int* and x is a *byte*.

Attacks on AES

Variations of a *timing attack* can be used on AES. This is due to the fact accessing certain table entries takes a certain amount of time; with enough known plaintext attacks, an attacker can learn the key.

Details of AES

We will first provide a structure to the 128-bits blocks AES operates on, starting with the 128-bit block of plaintext as 16 bytes of 8 bits, each

$$(a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1}, a_{1,1}, a_{2,1}, a_{3,1}, a_{0,2}, a_{1,2}, a_{2,2}, a_{3,2}, a_{0,3}, a_{1,3}, a_{2,3}, a_{3,3})$$

arranged in a 4×4 matrix as follows

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

SubBytes Step

In this step each byte in the matrix is substituted with a replacement byte according to this S-box, known as $GF(2^8)$:

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

This step substitutes each byte in the matrix with one in the massive table

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \to \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

ShiftRows Step

This step is a simple permutation which mixes up the bytes in each row of the 4×4 matrix output from the SubBytes Step. It's a cyclical shift; the first row is shifted by 0, second by 1, third 2, fourth 3, as follows:

$$\begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} \to \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{bmatrix}$$

$$= \begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix}$$

MixColumns Step

This step mixes up the information in each column in each column of the 4×4 matrix output from the ShiftRows Step. It does this by applying what turns out to be a Hill Cipher matrix-multiplication transformation applied to each column, using the number system $GF(2^8)$

I doubt we have to go into as much detail as the book does, but it's described here in the book.

Effectively, the MixColumns step is performed as follows

$$\begin{bmatrix} 00000010 & 00000011 & 00000001 & 00000001 \\ 00000001 & 00000010 & 00000011 & 00000001 \\ 00000001 & 00000001 & 00000010 & 00000011 \\ 00000011 & 00000001 & 00000001 & 00000010 \end{bmatrix} \cdot \begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \$\$\$\$ = \begin{bmatrix} d_{0,} \\ d_{1,} \\ d_{2,} \\ d_{3,} \end{bmatrix}$$

AddRoundKey Step

Finally, we *XOR* the result from the MixColumns Step with a set of keys derived from the 128-bit secret key (explained below).

$$\begin{bmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{bmatrix} \oplus \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

$$= \begin{bmatrix} e_{0,0} & e_{0,1} & e_{0,2} & e_{0,3} \\ e_{1,0} & e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,0} & e_{2,1} & e_{2,2} & e_{2,3} \\ e_{3,0} & e_{3,1} & e_{3,2} & e_{3,3} \end{bmatrix}$$

AES Key Schedule

This is really the critical part of the AddRoundKey Step.

First, the secret key K is divided into 16-bytes and arranged into a 4×4 matrix in column major ordering. We refer to these columns as $W[0], W[1], W[2]$ and $W[3]$, such that the *round 0 key matrix* (the one referred to before) can be viewed as

$$\begin{bmatrix} W[0] & W[1] & W[2] & W[3] \end{bmatrix}$$

We can now determine the columns $W[4i], W[4i+1], W[4i+2]$, and $W[4i+3]$ for the round i key matrix from the columns $W[4i-4], W[4i-3], W[4i-2]$, and $W[4i-1]$ of the round $i-1$ key matrix.
The first column, $W[4i]$, is computed slightly differently than the rest

$$W[4i] = W[4i-4] \oplus T_i(W[4i-1])$$

where T_i is a special transformation. The other 3 columns are computed as follows, in this order:

$$W[4i+1] = W[4i-3] \oplus W[4i]$$

$$W[4i+2] = W[4i-2] \oplus W[4i+1]$$

$$W[4i+3] = W[4i-1] \oplus W[4i+2]$$

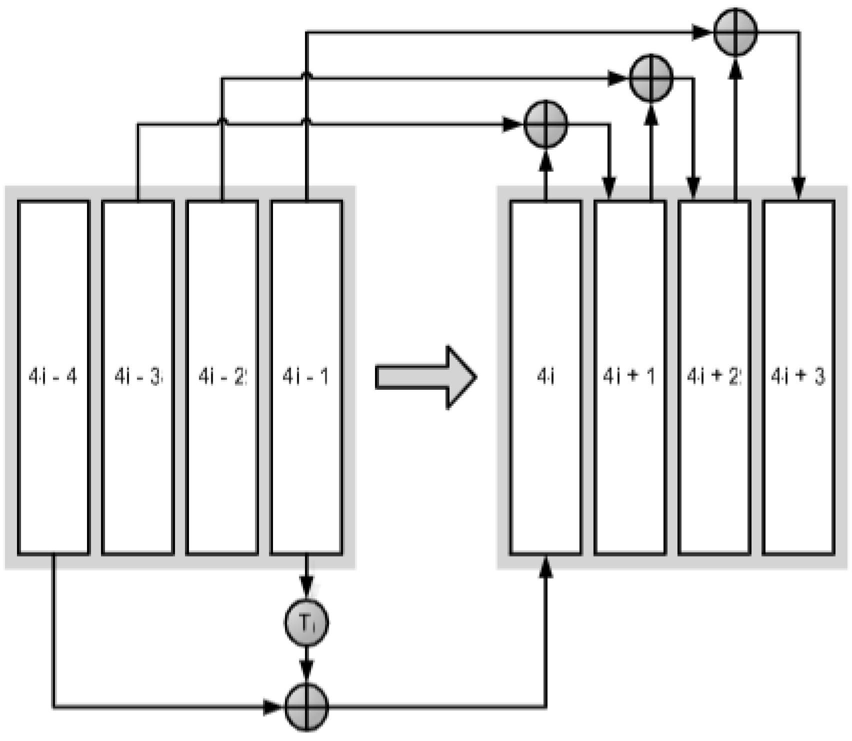


Figure 15: The key schedule for AES encryption.

The special Transformation

I give up typing this, here's the book

References/clarifications

$[2]$ - basically what we've covered breaking data into blocks
$[3]$ - Wikipedia link, but I don't think it's implementation will be tested

Modes of Operation

(Source)

There are many ways to use a *block cipher* like AES, known as its *modes of operation*. The general scenario for these modes is that we have a sequence of blocks $B_1, B_2, B_3, \ldots, B_i$
to encrypt all with the same key K using a block cipher algorithm (like AES).

Electronic Codebook (ECB) Mode

Notes from the lecture on ECB

> ## Electronic Code Book (ECB) mode
>
> To encrypt M under K using ECB
>
> - M is padded
>   - -> M'
> - M' is broken into *m* blocks of length *l*
> - Each block M_i is encrypted under the key K using the block cipher
>   - Creates a bunch of corresponding C_i's
> - Ciphertext corresponding to *M* is concatenation of all the C_i's
>
> ### Weaknesses of ECB
>
> If the blocks are the same, the ciphertext is the same
> You can manipulate it with frequency analysis
> Example:

> ECB has been proven to be a bad way of using AES

ECB involves encrypting the block B_i according to the following formula:

$$C_i = E_K(B_i)$$

(where E_K denotes the encryption algorithm (such as AES using *key* K)

Similarly, decryption is achieved using the following formula:

$$B_i = D_K(C_i)$$

The advantage of ECB is simplicity. It can also tolerate the loss of a block (in case they're being sent over a network, for example. This is because decrypting the block B_i does not depend on having the block B_{i-1} in any way.

The disadvantage is that if our encryption algorithm is *deteministic*[4] (like AES *is*), ECB mode may reveal patterns that appear in the steam of blocks. For example, in a large image file many blocks are the same colour and will be encrypted in the same way.

Cipher-Block Chaining (CBC) Mode

An alteration of ECB is CBC mode. The concept is that we *XOR* a block with it's previous *encrypted ciphertext block* before doing stuff with it. For the first plaintext block B_1, we *XOR* it with an *initialisation vector* (sent with the message) instead (as there is no B_0).
We set C_0 to the initialisation vector and encryption is handled by the following formula:

$$C_i = E_K(B_i \oplus C_{i-1})$$

Decryption is handled by reversing this formula:

$$B_i = D_k(C_i) \oplus C_{i-1}$$

If identical blocks appear at different places in the input sequence they are very likely to have different encryptions in the *ciphertext* output sequence.

Decryption can be done in parallel if all *ciphertext* blocks are available, and is not as volatile as it may seem. If block C_i is lost, the decryption of blocks i and $i+1$ are lost; but C_{i+2} can still be done since we have C_{i+1} and C_{i+2}.

Cipher Feedback (CFB) Mode

Similar to CBC. Also has an *initialisation vector* C_0.
To compute the encryption of the ith block we use the formula:

$$C_i = E_K(C_{i-1}) \oplus B_i$$

The ith block is encrypted by *XOR*ing the encrypted previous block with the i th *plaintext* block. Decryption is done similarly:

$$B_i = E_k(C_{i-1}) \oplus C_i$$

Output Feedback (OFB) Mode

In *OFB* a sequence of blocks is encrypted much as in one-time pads, but with a sequence of blocks generated with the block cipher. Beginning with an *initialisation vector* V_0, it generates a sequence of vectors:

$$V_i = E_K(V_{i-1})$$

Given this sequence of pad vectors, we perform block encryptions:

$$C_i = V_i \oplus B_i$$

and block decryptions:

$$B_i = V_i \oplus C_i$$

This mode of operation *can tolerate block losses* and can be *performed in parallel, for both **encryption** and **decryption*** (provided the vectors V_i have been pre-computed)

Counter (CTR) Mode

Similar to OFB and every step of CTR can be *done in parallel*.
We start with a random seed s and compute the ith offset vector according to the formula:

$$V_i = E_K(s + i - 1)$$

The first pad is an encryption of the seed, the second pad an encryption of $s + 1$, the third $s + 2$, and so on.
Encryption of performed identically to OFB mode but using these vectors:

$$C_i = V_i \oplus B_i$$

Decryption performed predictably the same:

$$B_i = V_i \oplus C_i$$

The generation of the pad vectors V_i as well as the encryption/decryption can all be done in parallel. CTR can also recover from dropped blocks

Public-Key Cryptography

In modern cryptography messages are usually encoded in binary, which allows us to do some cool stuff with them

Modular Arithmetic

We need to make sure any operations on blocks of bits result in outputs of the same size. We can do this with *modular artithmetic*. After an operation on a block of size n we return the remainder of a division of the result with n.
Technically this means we are performing arithmetic in Z_n:

$$Z_n = 0, 1, 2, \ldots, n - 1$$

Any operations we perform are the same with standard integers with this added step of *reducing* the result to a value in Z_n.

Modulo Operator

$x \, mod \, n$ (or $x \, modulo \, n$) takes an arbitrary integer x and a positive integer n and returns a value in Z_n, defined using the following rules:

- If $0 \leq x \leq n-1$ (that is $x \in Z_n$), $x \, mod \, n = x$
- If $x \geq n$, $x \, mod \, n$ is the remainder when dividing x by n
- If $x < 0$, we add a large multiple of n to x (denoted by kn) to get a non-negative number $y = x + kn$. We have that $x \, mod \, n = y \, mod \, n$, and can use previous rules to determine an actual value

Modular Inverses

We can consider the inverse x^{-1} of a number x in Z_n since we can write a/b as ab^{-1}.
We say y is the $modular \; inverse \; of \; x$, modulo n, if:

$$xy \, mod \, n = 1$$

For example, 4 is the inverse of 3 in Z_{11} since

$$4 * 3 \, mod \, 11 = 12 \, mod \, 11 = 1$$

If n is prime, every element $\neq 0$ in Z_n admits a modular inverse.

Modular Exponentiation

Modular Exponentiation, that is to say:

$$x^y \, mod \, n$$

can also be used. For the following figure

$$x^1 \, mod \, n, x^2 \, mod \, n, \ldots, x^{n-1} \, mod \, n$$

we can see the following patterns:

- If n is *not* prime, there are modular powers equal to 1 only for the elements of Z_n. These are the same elements x such that the *greatest common divisor (GDC)* of x and n is equal to 1.

- If n is prime every nonzero element of Z_n has a power equal to 1; we always have

  $$x^{n-1} \, mod \, n = 1$$

  We can generalise these patterns by considering the subset Z_n^* of Z_n defined as:

  $$Z_n^* = \{x \in Z_n \text{ such that } GCD(x, n) = 1\}$$

  (We could say $Z_n^$ is the subset where the elements are coprime with n, I think)

Let $\phi(n)$ be the number of elements in Z_n^*:

$$\phi(n) = |Z_n^*|$$

Euler's Theorem holds for each element x of Z_n^*:

$$x^{\phi(n)} \, mod \, n = 1$$

This is the main takeaway here. For an element x of Z_n^*, the modular inverse is $x^{\phi(n)-1}$, since:

$$x * x^{\phi(n)-1} \, mod \, n = x^{\phi(n)} \, mod \, n = 1$$

RSA

The setup RSA allows a potential receiver *Bob* to create his public and private keys. Bob generates two large random prime number p and q, and calculates $n = pq$. He picks a large number e that is relatively prime to $\phi(n)$ and computes $d = e^{-1} \, mod \, \phi(n)$. Bob does no longer need p, q or $\phi(n)$ (and it is probably in his best security interest to throw these away).
Bob's ***public key*** is the pair (e, n).
Bob's ***private key*** is d.
Bob keeps his private key secret but can publish his public key to anyone he wants to send him a secret message

Given Bob's public key, *Alice* can *encrypt* a message M for him by computing

$$C = M^e \, mod \, n$$

To *decrypt* the sent *ciphertext* C, Bob can compute:

$$C^d \, mod \, n$$

This is because

$$C^d \, mod \, n = (M^e)^d \, mod \, n$$
$$= M^{ed} \, mod \, n$$
$$= M^{ed \, mod \, \phi(n)} \, mod \, n$$
$$= M^1 \, mod \, n$$
$$= M$$

(when M is relatively prime to n)

Security of RSA

RSA's security mainly stems from finding d given e and n; if we knew $\phi(n) = (p-1)(q-1)$ it would be easy to compute d (Bob's private key). It is best Bob destroy all knowledge of p and q. RSA can be subject to *side channel attacks* (measuring CPU time/power consumption of known p and q 's).

RSA is also deterministic; if $C_1 = C_2$, then $M_1 = M_2$.

Efficiently implementing RSA

To efficiently implement RSA, efficient algorithms in the following tasks are needed:

- **Primality testing** - usually this is done by checking if a number is prime and just generating random numbers until a prime one is found. Primes are still one of mathematics' greatest mysteries and appear in whack-ass places
- **Computing the GCD** - used in the setup phase to pick the encryption exponent
- **Modular power** - used basically everywhere computing the power than applying the modulo operator isn't efficient with large numbers - RSA is reccomended to use 2048-bit moduli since 2010

Cryptographic Hash Functions

A *cryptographic hash function* provides a mapping from one set of things to another (in our interest, *message-space* to *ciphertext-space*) that is deterministic, *one-way*, and *collision-resistant*.

Properties

One-way

This says given a message M, it should be *easy* to compute the hash value $H(M)$, but given a value x it should be *difficult* to find a message M such that $x = H(M)$. Moreover, the hash value should be significantly smaller than a typical message (for example, *SHA-256* produces hash values with 256 bits).

Collision Resistant

We say a hash function H has *strong collision resistance* if is is computationally difficult to compute two distinct messages M_1 and M_2 such that

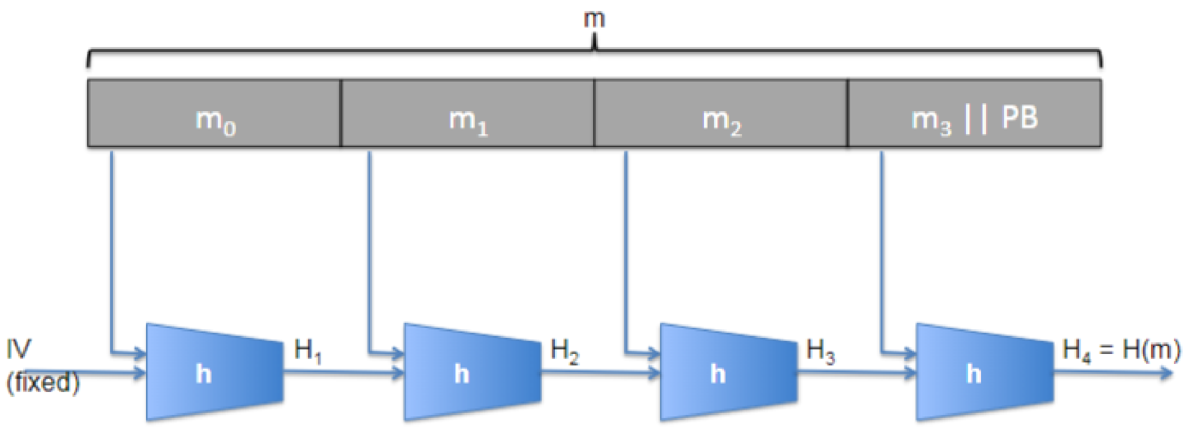
$$H(M_1) = H(M_2)$$

The Merkle-Damgård Construction

A common method of building a hash function is to use a *cryptographic compression function* $C(X, Y)$ - a cryptographic hash function $C, that takes two strings \$X$ and Y with fixed length m and n respectively, and produces a hash value of length n.

Given a message M, we divide this into multiple blocks $M_1, M_2, \ldots, M_k$ each of length m. M_k will be passed in an unambiguous way to ensure it is of length m. We take an *initialisation vector* v of length n and apply C to the first block. Denoting the resulting hash value as $d_1 = C(M_1, v)$ we compute d_2 using M_2 & d_1; $d_2 = C(M_2, d_1)$. We finally define the resulting hash value H of M as $H = d_k$.

The Merkle-Damgard construction



- Compression function: $h: \mathcal{T} \times \mathcal{X} \rightarrow \mathcal{T}$
- PB: $1000\ldots0||$mes-len (add extra block if needed)

If using the Merkle-Damgård Construction high collision resistance is important. If an attacker finds a collision between two different messages M_1 and M_2 (such that $H(M_1) = H(M_2)$) then other arbitrary collisions can be performed; for any message P:

$$H(M_1||P) = H(M_2||P)$$

Birthday Attacks

If there are more than 23 people in a room, there is a >50% chance two of them share the same birthday, this is *almost certain* if there are >60 people in the room. Tis is because if there are 23 people in a room we have

$$23 * 22/2 = 253$$

possible pairs of people, and all would have to be different for this fact to not be true.

Suppose a hash function H has a b-bit output. The number of possible hash values is 2^b; we might at first think Eve, our attacker, has to generate a number of inputs proportional to 2^b. By using this above factoid, this is not the case.
Eve generates a large number of random messages, computing the *hash value* of each one and hoping to find two messages hashing to the same value. With a sufficiently large number of generated messages, there is a high likelihood she will find a collision.
She will, in fact, only really have to generate a little bit more than $2^{b/2}$ inputs

Analysis of the Birthday Attack

Consider the b-bit hash function and let $m = 2^b$ denote the number of possible hash values. The probability the i-th message generated by the attacker does *not* collide with any of the previous $i - 1$ messages is

$$1 - \frac{i-1}{m}$$

Therefore, the *failure probability* at round k - the probability the attacker has not found any collisions after generating k messages, is

$$F_k = (1 - \frac{1}{m}) + (1 - \frac{2}{m}) + (1 - \frac{3}{m}) + ... + (1 - \frac{k-1}{m})$$

which can be approximated (using $1 - x \approx e^{-x}$) to say

$$F_k \approx e^{(1-\frac{1}{m})+(1-\frac{2}{m})+(1-\frac{3}{m})+...+(1-\frac{k-1}{m})} = e^{-\frac{k(k-1)}{m}}$$

Solving $e^{-\frac{k(k-1)}{m}} = \frac{1}{2}$ we get $k \approx 1.17\sqrt{m}$. Since $\sqrt(m) = \frac{b}{2}$, we can see the justification of the birthday attack

Digital Signatures

While we've covered a lot of ways to ensure the integrity of data, we now also need to ensure it's authenticity (message sender authentication). A *digital signature* is a way for an entity to demonstrate the authenticity of a message. These signature methods should ideally have two important properties:

- ***Nonforgeability*** - It should be difficult for Eve to forge a signature $S_{Alice}(M)$ for a message M as if it is coming from Alice
- ***Nonmutability*** - It should be difficult for Eve to take a signature $S_{Alice}(M)$ for a message M and convert the signature into a valid signature on a different message N.
  If these properties are met then a third one - ***Nonrepudiation*** - is also met by accident
- It should be difficult for *Alice* to claim she didn't sign a document M once she has produced a digital signature $S_{Alice}(M)$ for that document.

The RSA Signature Scheme

In the RSA Signature Scheme, Bob encrypts a message M using his secret key d:

$$S = M^d \, mod \, n$$

Any third part can verify this signature, using Bob's public key (e, n) by checking

$$\text{Is it true that } M = S^e \, mod \, n?$$

This is true since $de \, mod \, \phi(n) = 1$ and therefore

$$S^e \, mod \, n = M^{de} \, mod \, n = M^{de \, mod \, \phi(n)} \, mod \, n = M^1 \, mod \, n = M$$

This is nonforgeable since the RSA algorithm is very difficult to break. However, the RSA signature scheme does not achieve nonmutability. Suppose Eve has two valid signatures

$$S_1 = M_1^d \, mod \, n \text{ and } S_2 = M_2^d \, mod \, n$$

from Bob on messages M_1 and M_2. Eve could produce a new signature

$$S_1 * S_2 \, mod \, n = (M_1 * M_2)^d \, mod \, n$$

which would be a valid signature from Bob on the message $M_1 * M_2$. Thankfully in practise signatures are almost always used with Cryptographic Hash Functions which fixes this problem.

ElGamal Signature Scheme

In the *ElGamal signature scheme*, signatures are done through randomisation.
Like in standard ElGamal Encryption, Alice chooses a large random number p, finds a generator g for Z_p, picks a (secret) random number x, computes $y = g^x \, mod \, p$, and publishes (y, p) as her public key. To sign a message M, Alice generates a fresh one-time-use random number k and computes the following two numbers from it

- $a = g^k \, mod \, p$
- $b = k^{-1}(M - xa) \, mod \, (p - 1)$
  The pair (a, b) is Alice's signature on the message M: $S_{Alice}(M)$

To verify the signature (a, b) on M, Bob performs the following test:

$$\text{Is it true that } y^a a^b \; mod \, p = g^M \; mod \, p?$$

This is true because of the following:

$$\begin{aligned} y^a a^b \; mod \, p &= (g^x \; mod \, p)((g^k \; mod \, p)^{k^{-1}(M-xa) \; mod \; (p-1)} \; mod \, p) \\ &= g^{xa} g^{kk^{-1}(M-xa) \; mod \; (p-1)} \; mod \, p \\ &= g^{xa+M-xa} \; mod \, p \\ &= g^M \; mod \, p \end{aligned}$$

It is important Alice *never* reuse a random number k for two different signatures. Suppose she produces

$$b_1 = k^{-1}(M_1 - ax) \; mod \, (p-1) \;\text{ and }\; b_2 = k^{-1}(M_2 - ax) \; mod \, (p-1)$$

with the same $a = g^k \; mod \, p$ for two different messages M_1 and M_2. Then

$$(b_1 - b_2)k \; mod \, (p-1) = (M_1 - M_2) \; mod \, (p-1)$$

and since both $b_1 - b_2$ and $M_1 - M_2$ are easily computed values, Eve can easy compute k; and with k, can compute x from either b_1 or b_2 - from this point on, Eve knows Alice's secret key!

Using Hash Functions with Digital Signatures

In practise we would not just use these two encryption schemes as is. For one, if the signed message M is very long, both schemes are inefficient. Additionally, one can construct valid RSA signatures on combined messages from existing RSA signatures.
IRL, digital signatures are usually applied to cryptographic hashes of messages and not the real message. This significantly reduces the mutability risk for RSA signatures for example, since the chance of two hash values $H(M)$ and $H(N)$ would be equal to the hash of the product $H(M * N)$ is ridiculously small.
Additionally, signing a hash value is more efficient than signing a full message.

However, if Eve has found a collision between inputs M and N (such that $H(M) = H(N)$) and gets Alice to sign the hash $H(M)$ of message M, the risks of the birthday attack are heightened