s-DES By Examples

Spring 2022

How DES Works in Detail?

The DES Algorithm Illustrated (tu-berlin.de)

Cryptography | DES implementation in C | Techie Delight

Introduction

Simplified DES is an algorithm that has many features of the DES, but is much simpler then DES.

Like DES, this algorithm is also a bock cipher.

Block Size:

- In Simplified DES, encryption/decryption is done on blocks of 12 bits.
- The plaintext/ciphertext is divided into blocks of 12 bits and
- the algorithm is applied to each block.

Key:

- The key has 9 bits.
- The key, K_i, for the ith round of encryption is obtained by using 8 bits of K, starting with the ith bit.

Example:

- If K = 111000111, then K_1 = 11100011 and K_3 = 10001111 and K_{10} = K_1 = 11100011

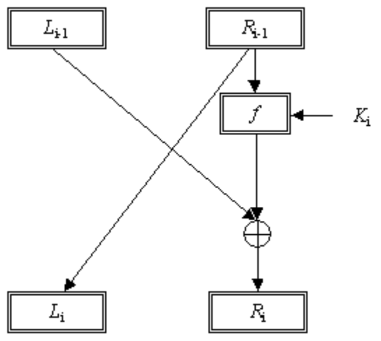
https://www.cs.uri.edu/cryptography/dessimplified.htm

Algorithm:

1. The block of 12 bits is written in the form L_0R_0, where:
   - L_0 consists of the first 6 bits, and
   - R_0 consists of the last 6 bits.
2. The ith round of the algorithm transforms an input $L_{i-1}R_{i-1}$ to the output L_iR_i using an 8-bit K_i derived from K.
3. The output for the ith round is found as follows:

   $$L_i = R_{i-1}$$

   $$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$
4. This operation is performed for a certain number of rounds, say n, and produces L_nR_n.
5. The ciphertext will be R_nL_n.
6. Encryption and decryption are done the same way except the keys are selected in the reverse order:
   - The keys for encryption will be K_1, K_2 …… K_n and
   - for decryption will be K_n, K_{n-1} …… K_1.



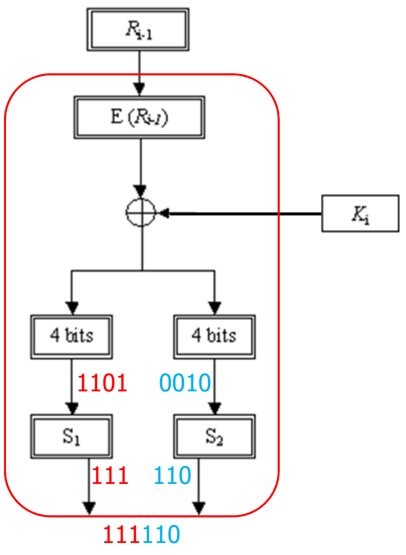
One Round of a Feistel System

Function f(R_{i-1},K_i)

As shown in the red box:

1. The expansion function E(R_{i-1}) takes 6-bit input and produces an 8-bit output.
2. The 8-bit output from the Expansion is Exclusive-ORed with the key K_i
3. The 8-bit output is divided into two blocks.
   - The first block consists of the first 4 bits and the last 4 bits make the second block.
   - The first block is the input for S1 and the second block is the input for S2.
4. The S-boxes take 4 bits as input and produce 3bits of output.
5. The output from the S-boxes is combined to form a single block of 6 bits. These 6 bits will be the output of the function f(R_{i-1}, K_i).

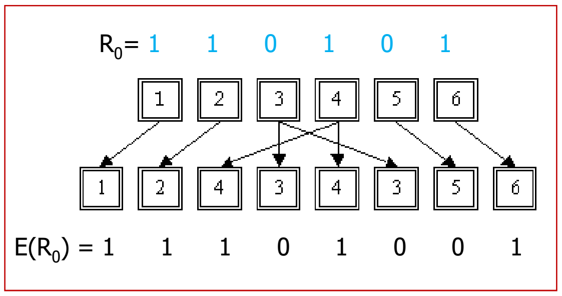
In the example, the S1 output 111 and S2 output 110. So the output for the function f(R_{i-1},K_i) will be 111110.



The Function f(R_{i-1}, Ki)

The Expansion Function

- The Expansion Function **E(R_{i-1})**:
  - The 6-bits are expanded using the expansion function.
  - The expansion function takes 6-bit input and produces an 8-bit output.
  - This output will be XOR-ed with Key and become the input to the two S-boxes



- Breaking R (half data block, 32 bits) into two 3-bit chunks
- and expanding each chunk into 4 bits
  - by stealing the rightmost bit of the left chunk to the right chunk, next to the leftmost bit.
  - Similarly for the right chunk.

The S Boxes

S1	0	1	2	3	4	5	6	7
0	101	010	001	110	011	100	111	000
1	001	100	110	010	000	**111**	101	011

S2	0	1	2	3	4	5	6	7
0	100	000	**110**	101	111	001	011	010
1	101	011	000	111	110	010	001	100

- Selection Rule: The first bit of the input is used to select the row from the S-box, 0 for the first row and 1 for the second row. The last 3 bits are used to select the column.

Example: Let the output from the previous step be 11010010.

- 1101 will be the input for the S1 box and 0010 will be the input for the S2 box.
- For S1, the first of the input is 1 so select the second row and 101 will select the 6th column. The output from the S1 box will be 111,.
- Similarly the output from the S2 box will be 110.

Example: Encryption

Let Input message be 100010110101 and the key be 111000111.
The encryption is explained for two rounds:

Round 1 (i = 0)

1. L_0= 100010 and R_0= 110101; K_1 = 11100011.
2. $E(R_0)$ = 11101001. [see the expansion diagram]
3. $E(R_0) \oplus K_1$ = 11101001 ⊕ 11100011 = 00001010.
4. S1(0000) = 101, S2(1010) = 000,
   ➔ $f(R_0,K_1)$ = 101000
5. $f(R_0,K_1) \oplus L_0$ = 101000 ⊕ 100010 = 001010.
6. Now using the formulas
   $L_i = R_{i-1}$
   $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$
   we get L_1 = 110101 and R_1 = 001010.

Round 2 (i = 1)

1. L_1 = 110101 and R_1 = 001010; K_2 = 11000111.
2. $E(R_1)$ = 00010110.
3. $E(R_1) \oplus K_2$ = 00010110 ⊕ 11000111= 11010001.
4. S1(1101) = 111; S2(0001) = 000;
   ➔ $f(R_1,K_2)$ = 111000
5. $f(R_1,K_2) \oplus L_1$ = 111000 ⊕ 110101 = 001101.
6. Now using the formulas
   $L_i = R_{i-1}$
   $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$
   we get L_2 = 001010 and R_2 = 001101.

Finally, the encrypted message: R_2 L_2 = 001101001010

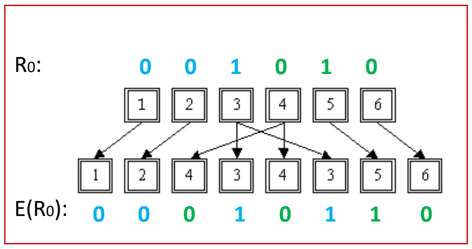
Example: Decryption

Round 1(i = 0)

1. L_0= 001101 and R_0= 001010; K_2 = 11000111.
2. $E(R_0)$ = 00010110.
3. $E(R_0) \oplus K_1$ = 00010110 ⊕ 11000101 = 11010001.
4. S1(1101) = 121; S2(0001) = 000; ➔ $f(R_0,K_2)$ = 111000
5. $f(R_0,K_2) \oplus L_0$ = 111000 ⊕ 001101 = 110101.
6. Now using the formulas $L_i = R_{i-1}$ and $R_i = L_{i-1} \oplus f(R_{i-1}, K_{n-i})$ we get L_1 = 001010 and R_1 = 111000.

Round 2 (i = 1)

1. L_1 = 001010 and R_1 = 111000; K_1 = 11100011.
2. $E(R_1)$ = 00010110.
3. $E(R_1) \oplus K_1$ = 11101001 ⊕ 11100011= 00001010.
4. S1(0000) = 101; S2(1010) = 000; ➔ $f(R_1,K_1)$ = 101000
5. $f(R_1,K_1) \oplus L_1$ = 101000 ⊕ 001010 = 100010.
6. Now using the formulas $L_i = R_{i-1}$ and $R_i = L_{i-1} \oplus f(R_{i-1}, K_{n-i})$ we get L_2 = 110101 and R_2 = 100010.



So decrypted message,
R_2L_2 =100010110101,
which is the original plaintext message.

9

Example: DES CBC

Online DES Encryption and Decryption:
https://encode-decode.com/des-encrypt-online/
Encryption: DES-CBC
Plaintext: This is a fun class. We should spend more effort.
Secret: EEL4347
Ciphertext: 3GDUGPxsa0JC4NAaYle2tjXZZC9gQdMlYhwOVEwoboeu4Zhdy5uEokEie2XY9acEjqwCY43p8Bc=

Online DES Encryption and Decryption:
https://www.tools4noobs.com/online_tools/decrypt
Key: EEL4347
Ciphertext: 3GDUGPxsa0JC4NAaYle2tjXZZC9gQdMlYhwOVEwoboeu4Zhdy5uEokEie2XY9acEjqwCY43p8Bc=
Algorithm: Des
Mode: CBC
Decode the input using: Base64
Plaintext: dXYCYCa fun class. We should spend more effort.???????

10

3DES

Triple DES or DESede , a symmetric-key algorithm for the encryption of electronic data, is the successor of DES (Data Encryption Standard) and provides more secure encryption then DES. The Triple DES breaks the user-provided key into three subkeys as k1, k2, and k3. A message is encrypted with k1 first, then decrypted with k2 and encrypted again with k3. The DESede key size is 128 or 192 bit and blocks size 64 bit. There are 2 modes of operation - Triple ECB (Electronic Code Book) and Triple CBC (Cipher Block Chaining).

Encryption: 3DES
Mode: CBC
IV (optional):
Plaintext: `The quick brown fox jumps over the lazy dog`
Secret key: `This is a fun class and for test only.`
Output text format: Base64
Cyphertext:
`0h+YhyB6cDSYblSOPT4P9J4ITk3vcac24A0RTqFAdwenan4XSUv33OCGJFpHcX9l`

3DES decryption in Base64
Mode: CBC
Secret key: the same
Decrypt: `VGhlIHF1aWNrIGJyb3duIGZveCBqdW1wcyBvdmVyIHRoZSBsYXp5wqBkb2c=`
Plaintext: `The quick brown fox jumps over the lazyÂ dog`

Online Tool for Triple DES Encryption and Decryption (devglan.com)

11

OpenSSL: DES3

To create an encrypted version of the file "dog1.tex", enter the following command:

Password: This is a fun class and for test only.

An encrypted version of your file was created in the same directory and named "dog1enc.tex.des3." If you try to open this file without first entering the password, you will get incoherent output.

To decrypt your file, enter the following command:

OpenSSL will once again prompt you to enter your password. Upon successful authentication, OpenSSL will create a new, decrypted version of your file "dog1new.tex" in the same directory.

In OpenSSL, the salt will be prepended to the front of the encrypted data, which will allow it to be decrypted. The purpose of the salt is to prevent dictionary attacks, rainbow tables, etc.

encryption - OpenSSL - Password vs Salt Purpose - Stack Overflow

```
mingyu@LAPTOP-CGO8MRKC ~/Example4
$ ls
dog1.tex
mingyu@LAPTOP-CGO8MRKC ~/Example4
$ cat dog1.tex
The quick brown fox jumps over the lazy dog

mingyu@LAPTOP-CGO8MRKC ~/Example4
$ openssl des3 -salt -in dog1.tex -out dog1enc.tex.des3
enter des-ede3-cbc encryption password:
Verifying - enter des-ede3-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

mingyu@LAPTOP-CGO8MRKC ~/Example4
$ ls
dog1.tex  dog1enc.tex.des3
mingyu@LAPTOP-CGO8MRKC ~/Example4
$ cat dog1enc.tex.des3
Salted__'▒▒09A▒▒81#▒▒%▒Y▒Я▒▒X▒▒▒
Q▒!▒N0xx{▒▒▒g@▒▒▒Iз▒y#
mingyu@LAPTOP-CGO8MRKC ~/Example4
$ openssl des3 -d -salt -in dog1enc.tex.des3 -out dog1new.tex
enter des-ede3-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
mingyu@LAPTOP-CGO8MRKC ~/Example4
$ ls
dog1.tex  dog1enc.tex.des3  dog1new.tex
mingyu@LAPTOP-CGO8MRKC ~/Example4
$ cat dog1new.tex
The quick brown fox jumps over the lazy dog
```

12

OpenSSL: AES

Basic Usage

The most basic way to encrypt a file is this

```
$ openssl enc -aes256 -base64 -in some.secret -out some.secret.enc
enter aes-256-cbc encryption password :
Verifying - enter aes-256-cbc encryption password :
```

It will encrypt the file *some.secret* using the AES-cipher in CBC-mode. The result will be Base64 encoded and written to *some.secret.enc*. OpenSSL will ask for password which is used to derive a key as well the initialization vector. Since encryption is the default, it is not necessary to use the **-e** option.

Use a given Key

It also possible to specify the key directly. For most modes of operations (i.e. all non-ECB modes) it is then necessary to specify an initialization vector. Usually it is derived together with the key from a password. And as there is no password, also all salting options are obsolete.

The key and the IV are given in hex. Their length depending on the cipher and key size in question.

```
$ openssl enc -des-ecb -K e0e0e0e0f1f1f1f1 -in mesg.plain -out mesg.enc
```

The key above is one of 16 weak DES keys. It should not be used in practice.

Decryption

```
$ openssl enc -aes-128-cbc -d -in some.secret.enc -out some.secret
```

Enc - OpenSSLWiki

13

Differences between DES and AES

	DES	AES
Developed	1977	2000
Key Length	56 bits	128, 192, or 256 bits
Cipher Type	Symmetric block cipher	Symmetric block cipher
Block Size	64 bits	128 bits
Security	Proven inadequate	Considered secure

14

Differences between DES and AES

DES Encryption Algorithm	AES Encryption Algorithm
Established as a standard in 1977.	Standardized in 2001.
Has the key length of 56 bits.	It offers key lengths of 128, 192, and 256 bits.
The block size is 64 bits.	The block size can be of 128, 192, or 256 bits – depending upon the key length.
The encryption process is time-consuming.	It offers almost six times faster performance compared to 3DES.
The encryption process involves 16 rounds.	The encryption process involves 10, 12, and 14 rounds in the case of 128, 192, and 256 bits, respectively.
The original version of DES has been found to be insecure and was deprecated from use in 2005. 3DES, the upgraded version of DES, is currently in use in some applications, but it's due to be deprecated in 2023.	AES is a much secure symmetric encryption algorithm with no considerable weakness found in it. It's used worldwide in applications such as hardware, software, SSL/TLS protocols, etc. It's the current standard of symmetric encryption.
It is based on Feistel Cipher Structure.	AES works substitution & permutation principle.
An encryption round involves Expansions permutation Xor, S-box, P-box, Xor, and Swap.	Encryption round involves Subtypes, Shiftonce, Mix columns, and Addroundkeys.

Base64 Encoding

Base64 Encoding

To encode a file *text.plain* you can use

```
$ openssl enc -base64 -in text.plain -out text.base64
```

To decode a file the the decrypt option (**-d**) has to be used

```
$ openssl enc -d -base64 -in text.base64 -out text.plain
```

Convert binary to base64

```
mingyu@LAPTOP-CGO8MRKC ~/Example4
$ ls
dog1.tex  dog1enc.tex.des3  dog1new.tex

mingyu@LAPTOP-CGO8MRKC ~/Example4
$ file dog1enc.tex.des3
dog1enc.tex.des3: openssl enc'd data with salted password

mingyu@LAPTOP-CGO8MRKC ~/Example4
$ openssl enc -base64 -in dog1enc.tex.des3 -out dog1enc.tex.des3.base64

mingyu@LAPTOP-CGO8MRKC ~/Example4
$ ls
dog1.tex  dog1enc.tex.des3  dog1enc.tex.des3.base64  dog1new.tex

mingyu@LAPTOP-CGO8MRKC ~/Example4
$ cat dog1enc.tex.des3.base64
U2FsdGVkX18RJxaD1TA5Qf8Q3RM4bCOPmRol6Fm40K/M7Vjz9/cKURGQIfhOMHh4
e7aqkGdAiqbUSRLTn955Iw==
```

17

base64 encoding and decoding

```
mingyu@LAPTOP-CGO8MRKC ~/Example4
$ cat dog1.tex
The quick brown fox jumps over the lazy dog

mingyu@LAPTOP-CGO8MRKC ~/Example4
$ openssl enc -base64 -in dog1.tex -out dog1.tex.base64

mingyu@LAPTOP-CGO8MRKC ~/Example4
$ cat dog1.tex.base64
VGhlIHF1aWNrIGJyb3duIGZveCBqdW1wcyBvdmVyIHRoZSBsYXp5wqBkb2cK
mingyu@LAPTOP-CGO8MRKC ~/Example4
$ openssl -d -base64 -in dog1.tex.base64 -out dog1.tex.plain
Invalid command '-d'; type "help" for a list.

mingyu@LAPTOP-CGO8MRKC ~/Example4
$ openssl base64 -d -in dog1.tex.base64 -out junk2.tex

mingyu@LAPTOP-CGO8MRKC ~/Example4
$ cat junk2.tex
The quick brown fox jumps over the lazy dog
```

18

Why DES is no longer effective?

To show that the DES was inadequate and should not be used in important systems anymore, a series of challenges were sponsored to see how long it would take to decrypt a message. Two organizations played key roles in breaking DES: distributed.net and the Electronic Frontier Foundation (EFF).

- The **DES I contest** (1997) took 84 days to break the encrypted message using a brute force attack.
- In 1998, there were two **DES II challenges** issued. The first challenge took just over a month and the decrypted text was *"The unknown message is: Many hands make light work".* The second challenge took less than three days, with the plaintext message *"It's time for those 128-, 192-, and 256-bit keys".*
- The final **DES III challenge** in early 1999 only took 22 hours and 15 minutes. Electronic Frontier Foundation's Deep Crack computer (built for less than $250,000) and distributed.net's computing network found the 56-bit DES key, deciphered the message, and they (EFF & distributed.net) won the contest. The decrypted message read *"See you in Rome (Second AES Candidate Conference, March 22-23, 1999)",* and was found after checking about 30 percent of the key space – finally proving that DES belonged to the past.

19

openssl genrsa – Generate RSA keys with OpenSSL

Use the openssl genrsa command to generate an RSA private key. The generated RSA private key can be customized by specifying the cipher algorithm and key size.

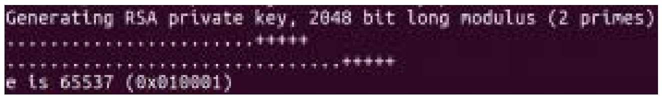
Prime numbers are used in generating the RSA private key. During generation the following symbols will be outputted demonstrating the progress of key generation:

. represents each number which has passed an initial sieve test

+ represents a number that has passed a round of the Miller-Rabin primality test

* represents that the current prime starts regenerating progress due to failed tests.

A newline represents the number has passed all the prime tests.



```
Generating RSA private key, 2048 bit long modulus (2 primes)
.......................+++++
...............................+++++
e is 65537 (0x010001)
```

openssl genrsa - Generate RSA keys with OpenSSL - Mister PKI

20

openssl genpkey vs genrsa

The openssl **genpkey** utility has superseded the **genrsa** utility. While the **genrsa** command is still valid and in use today, it is recommended to start using **genpkey**.

openssl genrsa 2048 example without passphrase

```
openssl genrsa -out key.pem 2048
```

Where `-out key.pem` is the file containing the plain text private key, and `2048` is the numbits or keysize in bits. Completion of running this command will result in a 2048 key generated by openssl genrsa.

OpenSSL: AES/DES to encrypt private key

openssl genrsa password example

```
openssl genrsa -out key.pem -aes256
```

Where `-out key.pem` is the file containing the AES encrypted private key, and `-aes256` is the chosen cipher. With this cipher, AES CBC 256 encryption is the type of encryption.
Note that other ciphers are also supported, including aria, camellia, des, des3, and idea.
Completion of running the above command will result in an aes256 key generated by openssl genrsa. How to determine and verify the private key is encrypted? List the private key file which shows the following:

OpenSSL: AES/DES to encrypt private key

```
cat key.pem
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-256-CBC,B26F7007EB3543EF0EEF4EBB0F508F6E
1SUT8Dv40Ay78zl26xJJrpkP1cDPsLk1V4eVoRQuGceS7hIo8gmKxbAVWdn15lPY
Q1fcifRaTINXg6tDH4WNhxBLbgF+Yqd3ouP7kS978LlFUZq3cAC8UAekvIcnJwzx
. . .
 LBlhkupf+owXTlqwulo6igP/esdaoNQ+2o13Cz5E3Ex+ShpJvByRLjP+5bVQSsj
-----END RSA PRIVATE KEY-----
```

`Proc-Type: 4,ENCRYPTED` indicates the key is encrypted
`DEK-Info: AES-256-CBC` indicates the AES cipher used for encryption
If the private key is not encrypted, the previous two lines will not be included in the file. If security is important, note that a private key should almost always be encrypted AND kept in a secure place.

OpenSSL: AES/DES to encrypt private key

openssl genrsa -passout

The **-passout** flag looks for an argument containing the file or variable holding the password. The following are options for the **-passout** argument:

- **pass**:password – password will be the actual password. This should only be used when security is not critical because the password will be available in history and other OS utilities.
- **env**:var – var will be the name of the environment variable.
- **file**:pathname – pathname of the file containing the password. The first line of the file should be the password.
- **fd**:number – This can be used to send the password with a pipe.
- **stdin** – Read the password from standard input.

An example of openssl genrsa -passout with a 2048 bit key size reading the password from a file:

```
openssl genrsa -passout file:pass.txt -out key.pem 2048
```

OpenSSL: AES/DES to encrypt private key

How to remove a private key password using openssl.
If it is necessary to store the decrypted version of your private key, run this openssl rsa command to decrypt your private key. Removing the encryption from your private key makes it more vulnerable to theft and is not recommend if the security of the key is important. In any instance, if the private key is stolen while encrypted or not, it should immediately be replaced, any associated public key or certificate should be revoked, and a security review of your system should be conducted to ensure the new private key cannot be stolen again.

```
openssl rsa -in key.pem -out decrypted-key.pem
```

Where `rsa` is the RSA algorithm, `-in key.pem` is the encrypted RSA private key file, and `-out decrypted-key.pem` is the file that will contain the decrypted private key.

OpenSSL and DES/AES

The encryption param of openssl genrsa command is used to specify which algorithm to use for encrypting your private key (using the password you specify).

CSR (Certificate Signing Request) includes your public key and some additional public information to be included into certificate. CSR never includes a private key.

So, choice of algorithm for encrypting the private key is completely unrelated to CSR. Choose whatever you prefer. AES variants and Triple-DES (-des3) should be preferred; plain DES is usually considered not secure these days. Also see why AES is more secure than DES. But I think algorithm choice in this particular case is not as important as using a strong password and protecting it.

OpenSSL both commandline and library uses the bad PBKDF (EVP_BytesToKey with one iteration) for **traditional** (i.e. not PKCS8) privatekey files, which genrsa writes, but (since 1.0.0 in 2010) genpkey writes PKCS8 using by default PBKDF2 with 2048 iterations, and (since 1.1.0 in 2016) piping to pkcs8 -topk8 -iter N can increase that. OpenSSL supports only CBC encryption of the keybag in PKCS12, but the *entire* PKCS12 is protected by PBMAC *usually* with the same password (you can change that with -twopass but the result is not very interoperable).

About the primality test:

cryptography - How does OpenSSL generate a big prime number so fast? - Information Security Stack Exchange

Testing for primality is much easier than performing integer factorization.

There are several ways to test for primality, such as the deterministic Sieve of Eratosthenes and the probabilistic Miller–Rabin primality tests. OpenSSL uses several tests to check for primality. First they subject the number to the deterministic checks, attempting division of the candidate with a number of small primes, then a series of Miller–Rabin primality tests. The vast majority of candidate primes are discarded with the very first primality test. Any candidates which pass them are subject to further rounds of testing, each of which increase the certainty that it is a prime.

When using the Miller–Rabin tests, a composite number has a 75% chance of being detected as such at each round, so after a mere 64 rounds of testing, the probability that a composite number goes undetected is a staggering 2^{-128}. In other words, the test has a 4^{-n} chance of a false negative, where *n* is the number of testing rounds. There are also a number of much slower ways to test if a number is prime with complete certainty, such as the Agrawal–Kayal–Saxena primality test, but for cryptographic purposes, being *really, really sure* is sufficient, so they tend not to be used.

By default, OpenSSL tends to be extra paranoid and does some other tests, specifically for a safe prime. So when a prime, *p*, is found, it also checks if *(p - 1) / 2* is prime. This is important for specific applications of primes such as Diffie–Hellman where safe primes prevent certain attacks.

This is possible at a high speed because verifying that an integer is a prime with an extremely low margin of error is significantly easier than factoring it, and because primes are not *that* uncommon (it is easy to find a large number of primes by incrementing a number and testing for primality). The Miller–Rabin primality test is very efficient. The test proves compositeness if $x^{n-1} \not\equiv 1 \pmod{n}$ (the Fermat test), and by testing whether or not $x^{(n-1)/2e} \pmod{n}$ is a nontrivial square root of *1 mod n* where *n* is the integer being tested and *x* is a random witness satisfying the interval $1 < x < n$.

The pseudocode implementation of the test taken from Wikipedia is:

Input: n > 3, an odd integer to be tested for primality; k, a parameter that determines the accuracy of the test Output: composite if n is composite, otherwise probably prime write $n - 1$ as $2^r \cdot d$ with d odd by factoring powers of 2 from $n - 1$ WitnessLoop: repeat k times: pick random integer a in the range $[2, n - 2]$ $x \leftarrow a^d \bmod n$ if $x = 1$ or $x = n - 1$ then continue WitnessLoop repeat $r - 1$ times: $x \leftarrow x^2 \bmod n$ if $x = 1$ then return composite if $x = n - 1$ then continue WitnessLoop return composite return probably prime

See this Crypto.SE answer on RSA key generation and the FIPS 186-4 standard, section 5.1.

Sieve testing of prime numbers:

In mathematics, the **sieve of Eratosthenes** is an ancient algorithm for finding all prime numbers up to any given limit.

It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2. The multiples of a given prime are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime. This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime. Once all the multiples of each discovered prime have been marked as composites, the remaining unmarked numbers are primes.

Sieve of Eratosthenes - Wikipedia

Miller–Rabin primality test

The **Miller–Rabin primality test** or **Rabin–Miller primality test** is a probabilistic primality test: an algorithm which determines whether a given number is likely to be prime, similar to the Fermat primality test and the Solovay–Strassen primality test.

It is of historical significance in the search for a polynomial-time deterministic primality test. Its probabilistic variant remains widely used in practice, as one of the simplest and fastest tests known.

Gary L. Miller discovered the test in 1976; Miller's version of the test is deterministic, but its correctness relies on the unproven extended Riemann hypothesis. Michael O. Rabin modified it to obtain an unconditional probabilistic algorithm in 1980.

Homework

- Investigate the online DES sites.
- Demo the encryption and decryption with an encryption, a key, and a plaintext. Show the resulted ciphertext.

Blowfish (cipher)

Blowfish is a symmetric-key block cipher, designed in 1993 by Bruce Schneier and included in many cipher suites and encryption products. Blowfish provides a good encryption rate in software, and no effective cryptanalysis of it has been found to date. However, the Advanced Encryption Standard (AES) now receives more attention, and Schneier recommends Twofish for modern applications.

Schneier designed Blowfish as a general-purpose algorithm, intended as an alternative to the aging DES and free of the problems and constraints associated with other algorithms. At the time Blowfish was released, many other designs were proprietary, encumbered by patents or were commercial or government secrets. Schneier has stated that "Blowfish is unpatented, and will remain so in all countries. The algorithm is hereby placed in the public domain, and can be freely used by anyone."

Notable features of the design include key-dependent S-boxes and a highly complex key schedule.

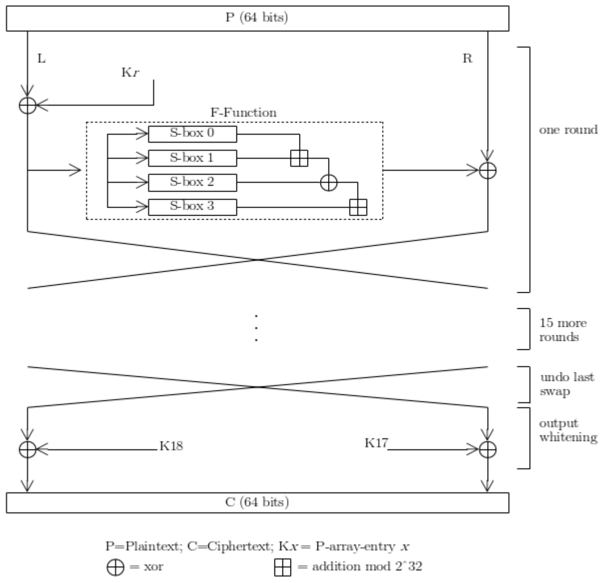
The algorithm in Blowfish

Blowfish has a 64-bit block size and a variable key length from 32 bits up to 448 bits. It is a 16-round Feistel cipher and uses large key-dependent S-boxes. In structure it resembles CAST-128, which uses fixed S-boxes.

The adjacent diagram shows Blowfish's encryption routine. Each line represents 32 bits. There are five subkey-arrays: one 18-entry P-array (denoted as K in the diagram, to avoid confusion with the Plaintext) and four 256-entry S-boxes (S0, S1, S2 and S3).

Every round *r* consists of 4 actions:

Action 1	XOR the left half (L) of the data with the *r* th P-array entry
Action 2	Use the XORed data as input for Blowfish's F-function
Action 3	XOR the F-function's output with the right half (R) of the data
Action 4	Swap L and R



The Feistel structure of Blowfish

The keys are P1, P2, ..., P17, P18, respectively, which are the same as K1, K2, …, K17, K18.

The algorithm

The F-function splits the 32-bit input into four 8-bit quarters and uses the quarters as input to the S-boxes. The S-boxes accept 8-bit input and produce 32-bit output. The outputs are added modulo 2^{32} and XORed to produce the final 32-bit output (see image in the upper right corner).

After the 16th round, undo the last swap, and XOR L with K18 and R with K17 (output whitening).

Decryption is exactly the same as encryption, except that P1, P2, ..., P18 are used in the reverse order. This is not so obvious because xor is commutative and associative. A common misconception is to use inverse order of encryption as decryption algorithm (i.e. first XORing P17 and P18 to the ciphertext block, then using the P-entries in reverse order).

Blowfish's key schedule starts by initializing the P-array and S-boxes with values derived from the hexadecimal digits of pi, which contain no obvious pattern. The secret key is then, byte by byte, cycling the key if necessary, XORed with all the P-entries in order. A 64-bit all-zero block is then encrypted with the algorithm as it stands. The resultant ciphertext replaces P_1 and P_2. The same ciphertext is then encrypted again with the new subkeys, and the new ciphertext replaces P_3 and P_4. This continues, replacing the entire P-array and all the S-box entries. In all, the Blowfish encryption algorithm will run 521 times to generate all the subkeys – about 4 KB of data is processed.

35

Some comments on the algorithm

Because the P-array is 576 bits long, and the key bytes are XORed through all these 576 bits during the initialization, many implementations support key sizes up to 576 bits. The reason for that is a discrepancy between the original Blowfish description, which uses 448-bit keys, and its reference implementation, which uses 576-bit keys. The test vectors for verifying third-party implementations were also produced with 576-bit keys. When asked which Blowfish version is the correct one, Bruce Schneier answered: "The test vectors should be used to determine the one true Blowfish".

Another opinion is that the 448 bits limit is present to ensure that every bit of every subkey depends on every bit of the key, as the last four values of the P-array don't affect every bit of the ciphertext. This point should be taken in consideration for implementations with a different number of rounds, as even though it increases security against an exhaustive attack, it weakens the security guaranteed by the algorithm. And given the slow initialization of the cipher with each change of key, it is granted a natural protection against brute-force attacks, which doesn't really justify key sizes longer than 448 bits.

36

Blowfish in pseudocode

```
uint32_t P[18];
uint32_t S[4][256];
uint32_t f (uint32_t x) {
        uint32_t h = S[0][x >> 24] + S[1][x >> 16 & 0xff];
        return ( h ^ S[2][x >> 8 & 0xff] ) + S[3][x & 0xff];
}
void blowfish_encrypt(uint32_t *L, uint32_t *R) {
        for (short r = 0; r < 16; r++) {
                *L = *L ^ P[r];
                *R = f(*L) ^ *R;
                swap(L, R);
        }
        swap(L, R);
        *R = *R ^ P[16];
        *L = *L ^ P[17];
}
void blowfish_decrypt(uint32_t *L, uint32_t *R) {
        for (short r = 17; r > 1; r--) {
                *L = *L ^ P[r];
                *R = f(*L) ^ *R;
                swap(L, R);
        }
        swap(L, R);
        *R = *R ^ P[1];
        *L = *L ^ P[0];
}
```

```
// initializing the P-array and S-boxes with values derived from pi; omitted in the
example (you can find them below)
// ...
{
        /* initialize P box w/ key*/
        uint32_t k;
        for (short i = 0, p = 0; i < 18; i++) {
                k = 0x00;
                for (short j = 0; j < 4; j++) {
                        k = (k << 8) | (uint8_t) key[p];
                        p = (p + 1) % key_len;
                }
                P[i] ^= k;
        } /* blowfish key expansion (521 iterations) */
        uint32_t l = 0x00, r = 0x00;
        for (short i = 0; i < 18; i+=2) {
                blowfish_encrypt(&l, &r);
                P[i] = l;
                P[i+1] = r;
        }
        for (short i = 0; i < 4; i++) {
                for (short j = 0; j < 256; j+=2) {
                        blowfish_encrypt(&l, &r);
                        S[i][j] = l;
                        S[i][j+1] = r;
                }
        }
}
```

Blowfish in practice

Blowfish is a fast block cipher, except when changing keys. Each new key requires the pre-processing equivalent of encrypting about 4 kilobytes of text, which is very slow compared to other block ciphers. This prevents its use in certain applications, but is not a problem in others.

In one application Blowfish's slow key changing is actually a benefit: the password-hashing method (crypt $2, i.e. bcrypt) used in OpenBSD uses an algorithm derived from Blowfish that makes use of the slow key schedule; the idea is that the extra computational effort required gives protection against dictionary attacks.

Blowfish has a memory footprint of just over 4 kilobytes of RAM. This constraint is not a problem even for older desktop and laptop computers, though it does prevent use in the smallest embedded systems such as early smartcards.

Blowfish was one of the first secure block ciphers not subject to any patents and therefore freely available for anyone to use. This benefit has contributed to its popularity in cryptographic software.

bcrypt is a password hashing function which, combined with a variable number of iterations (work "cost"), exploits the expensive key setup phase of Blowfish to increase the workload and duration of hash calculations, further reducing threats from brute force attacks.

bcrypt is also the name of a cross-platform file encryption utility developed in 2002 that implements Blowfish.

38

Weakness and successors

Blowfish's use of a 64-bit block size (as opposed to e.g. AES's 128-bit block size) makes it vulnerable to birthday attacks, particularly in contexts like HTTPS. In 2016, the SWEET32 attack demonstrated how to leverage birthday attacks to perform plaintext recovery (i.e. decrypting ciphertext) against ciphers with a 64-bit block size. The GnuPG project recommends that Blowfish not be used to encrypt files larger than 4 GB due to its small block size.

A reduced-round variant of Blowfish is known to be susceptible to known-plaintext attacks on reflectively weak keys. Blowfish implementations use 16 rounds of encryption, and are not susceptible to this attack.

Bruce Schneier has recommended migrating to his Blowfish successor, Twofish.

Birthday Attack

A **birthday attack** is a type of cryptographic attack that exploits the mathematics behind the birthday problem in probability theory. This attack can be used to abuse communication between two or more parties. The attack depends on the higher likelihood of collisions found between random attack attempts and a fixed degree of permutations (pigeonholes). With a birthday attack, it is possible to find a collision of a hash function in 2^{n/2}, with 2^n being the classical preimage resistance security. There is a general (though disputed) result that quantum computers can perform birthday attacks, thus breaking collision resistance, in

Understanding the problem

As an example, consider the scenario in which a teacher with a class of 30 students (n = 30) asks for everybody's birthday (for simplicity, ignore leap years) to determine whether any two students have the same birthday (corresponding to a hash collision as described further). Intuitively, this chance may seem small. Counter-intuitively, the probability that at least one student has the same birthday as *any* other student on any day is around 70% (for n = 30), from the formula:

$$1 - \frac{365!}{(365-n)! \cdot 365^n}$$

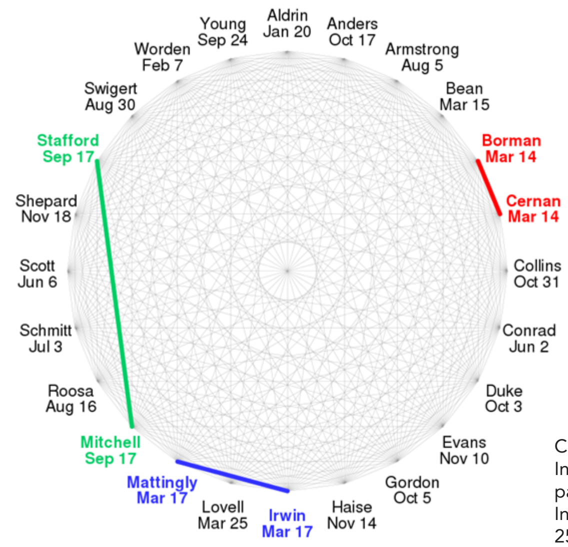
If the teacher had picked a *specific* day (say, 16 September), then the chance that at least one student was born on that specific day is about 7.9%:

$$1 - (364/365)^{30}$$

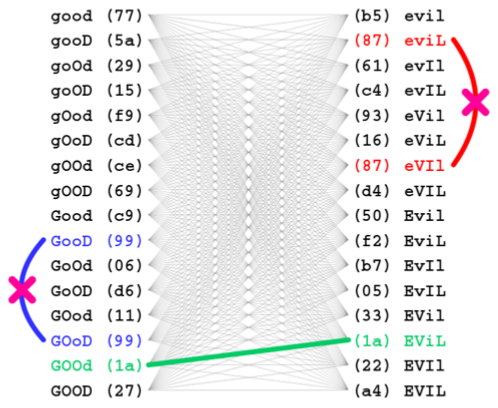
In a birthday attack, the attacker prepares many different variants of benign and malicious contracts, each having a digital signature. A pair of benign and malicious contracts with the same signature is sought. In this fictional example, suppose that the digital signature of a string is the first byte of its SHA-256 hash. The pair found is indicated in green – note that finding a pair of benign contracts (blue) or a pair of malicious contracts (red) is useless. After the victim accepts the benign contract, the attacker substitutes it with the malicious one and claims the victim signed it, as proven by the digital signature.

41

(1)



(2)



Comparison of the birthday problem (1) and birthday attack (2):
In (1), collisions are found within one set, in this case, 3 out of 276 pairings of the 24 lunar astronauts.
In (2), collisions are found between two sets, in this case, 1 out of 256 pairings of only the first bytes of SHA-256 hashes of 16 variants each of benign and malicious contracts.

42

Preimage attack

In cryptography, a **preimage attack** on cryptographic hash functions tries to find a message that has a specific hash value. A cryptographic hash function should resist attacks on its preimage (set of possible inputs).

In the context of attack, there are two types of preimage resistance:

- *preimage resistance*: for essentially all pre-specified outputs, it is computationally infeasible to find any input that hashes to that output; i.e., given y, it is difficult to find an x such that $h(x) = y$.
- *second-preimage resistance*: for a specified input, it is computationally infeasible to find another input which produces the same output; i.e., given x, it is difficult to find a second preimage $x' \neq x$ such that $h(x) = h(x')$.

These can be compared with a collision resistance, in which it is computationally infeasible to find any two distinct inputs x, x' that hash to the same output; i.e., such that $h(x) = h(x')$.

Collision resistance implies second-preimage resistance, but does not guarantee preimage resistance. Conversely, a second-preimage attack implies a collision attack (trivially, since, in addition to x', x is already known right from the start).

Applied preimage attacks

By definition, an ideal hash function is such that the fastest way to compute a first or second preimage is through a brute-force attack. For an n-bit hash, this attack has a time complexity 2^n, which is considered too high for a typical output size of $n = 128$ bits. If such complexity is the best that can be achieved by an adversary, then the hash function is considered preimage-resistant. However, there is a general result that quantum computers perform a structured preimage attack in $\sqrt{2^n} = 2^{n/2}$, which also implies second preimage and thus a collision attack.

Faster preimage attacks can be found by cryptanalysing certain hash functions, and are specific to that function. Some significant preimage attacks have already been discovered, but they are not yet practical. If a practical preimage attack is discovered, it would drastically affect many Internet protocols. In this case, "practical" means that it could be executed by an attacker with a reasonable amount of resources. For example, a preimaging attack that costs trillions of dollars and takes decades to preimage one desired hash value or one message is not practical; one that costs a few thousand dollars and takes a few weeks might be very practical.

All currently known practical or almost-practical attacks on MD5 and SHA-1 are collision attacks. In general, a collision attack is easier to mount than a preimage attack, as it is not restricted by any set value (any two values can be used to collide). The time complexity of a brute-force collision attack, in contrast to the preimage attack, is only $2^{n/2}$.

Restricted preimage space attacks

The computational infeasibility of a first preimage attack on an ideal hash function assumes that the set of possible hash inputs is too large for a brute force search. However if a given hash value is known to have been produced from a set of inputs that is relatively small or is ordered by likelihood in some way, then a brute force search may be effective. Practicality depends on the input set size and the speed or cost of computing the hash function.

A common example is the use of hashes to store password validation data for authentication. Rather than store the plaintext of user passwords, an access control system stores a hash of the password. When a user requests access, the password they submit is hashed and compared with the stored value. If the stored validation data is stolen, the thief will only have the hash values, not the passwords. However most users choose passwords in predictable ways and many passwords are short enough that all possible combinations can be tested if fast hashes are used, even if the hash is rated secure against preimage attacks. Special hashes called key derivation functions have been created to slow searches.

45

Birthday attack is a type of cryptographic attack that belongs to a class of brute force attacks. It exploits the mathematics behind the birthday problem in probability theory. The success of this attack largely depends upon the higher likelihood of collisions found between random attack attempts and a fixed degree of permutations, as described in the **birthday paradox problem**.

Birthday paradox problem –
Let us consider the example of a classroom of 30 students and a teacher. The teacher wishes to find pairs of students that have the same birthday. Hence the teacher asks for everyone's birthday to find such pairs. Intuitively this value may seem small. For example, if the teacher fixes a particular date say ***October 10***, then the probability that at least one student is born on that day is $1 - (364/365)^{30}$ which is about ***7.9%***. However, the probability that at least one student has the same birthday as any other student is around ***70%*** using the following formula:

$1 - 365!/((365 - n!) * (365^n))$ (substituting $n = 30$ here)

Birthday attack in Cryptography - GeeksforGeeks

46

Derivation of the above term:
Assumptions –
1. Assuming a non leap year(hence 365 days).
2. Assuming that a person has an equally likely chance of being born on any day of the year.

Let us consider n = 2.
P(Two people have the same birthday) = 1 – P(Two people having different birthday)
= 1 – (365/365)*(364/365)
= 1 – 1*(364/365)
= 1 – 364/365
= 1/365.

So for n people, the probability that all of them have different birthdays is:
P(N people having different birthdays) = (365/365)*(365-1/365)*(365-2/365)*….(365-n+1)/365.
= 365!/((365-n)! * 365^n)

Hash function –
A hash function H is a transformation that takes a **variable sized input m** and returns a **fixed size string** called a **hash value(h = H(m))**. Hash functions chosen in cryptography must satisfy the following requirements:

•The input is of variable length,
•The output has a fixed length,
•H(x) is relatively easy to compute for any given x,
•H(x) is one-way,
•H(x) is collision-free.

A hash function H is said to be one-way if it is hard to invert, where "hard to invert" means that given a hash value h, it is computationally infeasible to find some input x such that **H(x) = h**.
If, given a message x, it is computationally infeasible to find a message y not equal to x such that **H(x) = H(y)** then H is said to be a weakly collision-free hash function.
A strongly collision-free hash function H is one for which it is computationally infeasible to find any two messages x and y such that **H(x) = H(y)**.
Let **H: M => $\{0, 1\}^n$** be a hash function (**|M|** $>> 2^n$)
Following is a generic algorithm to find a collision in time **$O(2^{n/2})$** hashes.

Algorithm:

1. Choose $2^{n/2}$ random messages in M: m_1, m_2, …., $m_{n/2}$
2. For i = 1, 2, …, $2^{n/2}$ compute $t_i = H(m_i)$ => $\{0, 1\}^n$
3. Look for a collision ($t_i = t_j$). If not found, go back to step 1

We consider the following experiment. From a set of H values, we choose n values uniformly at random thereby allowing repetitions.
Let *p(n; H)* be the probability that during this experiment at least one value is chosen more than once. This probability can be approximated as:

```
p(n; H) = 1 - ( (365-1)/365) * (365-2)/365) * ...(365-n+1/365))
```
$p(n; H) = e^{-n(n-1)/(2H)} = e^{-n2/(2H)}$

Digital signature susceptibility –
Digital signatures can be susceptible to birthday attacks. A message ***m*** is typically signed by first computing ***H(m)***, where ***H*** is a cryptographic hash function, and then using some secret key to sign ***H(m)***. Suppose Alice wants to trick Bob into signing a fraudulent contract. Alice prepares a fair contract ***m*** and fraudulent one ***m'***. She then finds a number of positions where ***m*** can be changed without changing the meaning, such as inserting commas, empty lines, one versus two spaces after a sentence, replacing synonyms, etc. By combining these changes she can create a huge number of variations on m which are all fair contracts.

Similarly, Alice can also make some of these changes on ***m'*** to take it, even more, closer towards ***m***, that is ***H(m) = H(m')***. Hence, Alice can now present the fair version ***m*** to Bob for signing. After Bob has signed, Alice takes the signature and attaches to it the fraudulent contract. This signature proves that Bob has signed the fraudulent contract.

To avoid such an attack the output of the hash function should be a very long sequence of bits such that the birthday attack now becomes computationally infeasible.