Inhaltsverzeichnis

1. Kryptographische Algorithmen verwenden

1.1. Ziele

☑ Wir wollen

- ☐ kryptographische Algorithmen kennen lernen und verwenden können
- ☐ Die openSSL Bibliothek kennen lernen und verwenden.

☑ Quellen:

- ☐ S. Spitz, et.al.(2011): „Kryptographie und IT-Sicherheit“ , Vieweg+Teubner, 2011

1.2. Installation von openssl

Zur Verwendung in eigenen Programmen besprechen wir:

1. die **openSSL** – Bibliothek: http://www.openssl.org/
2. installation:

```
sudo apt-get install libssl-dev
```

1.3. +Klassische/einfache symmetrische Verschlüsselung

Die symmetrischen Verfahren Skytale, Cäsar (monoalphabetisch) und Vigenère (polyalphabetisch) werden in der Datei krypt-skytale-caesar-vigenere.odt (DS-ALGO) besprochen.

1.3.1. Skytale

Matrizenverfahren: siehe krypt-skytale-caesar-vigenere.odt (DS-ALGO)

1.3.2. Caesar

Monoalphabetisch: siehe: krypt-skytale-caesar-vigenere.odt (DS-ALGO)

1.3.3. Vigenere

Polyalphabetisch: siehe: krypt-skytale-caesar-vigenere.odt (DS-ALGO)

1.4. Moderne symmetrische Verschlüsselungsverfahren

http://verplant.org/facharbeit/html/node6.html

Bei der symmetrischen Verschlüsselung besitzen Sender und Empfänger exakt den selben Schlüssel zum Kodieren bzw. Dekodieren einer Nachricht.

Strom- und Block-chiffre
Symmetrische Verschlüsselungsverfahren lassen sich noch weiter unterteilen. Wird der Klartext Zeichen für Zeichen verschlüsselt spricht man von **Stromchiffren**.
Teilt man den Klartext in Teile mit definierter Größe auf und kodiert/dekodiert somit mehr als ein Zeichen auf einmal spricht man von **Blockchiffren**.

Moderne symmetrische Algorithmen sind komplex und eine genaue Betrachtung der Funktionsweise würde den Rahmen hier sprengen. Deshalb seien hier nur die wichtigsten Verfahren genannt:

Zur Verwendung in eigenen Programmen siehe u.a.:

1. die **openSSL** – Bibliothek: http://http://www.openssl.org/
2. Kryptographie mit **Java** http://www.torsten-horn.de/techdocs/java-crypto.htm

1.4.1. DES – data encryption standard

http://de.wikipedia.org/wiki/Data_Encryption_Standard

☑ steht für „**D**ata **E**ncryption **S**tandard“ und ist ein Block-chiffre Verfahren zur Verschlüsselung großer Datenmengen.

☑ Basiert auf Bit-Permutations und Bit-Substitutionsverfahren.

☑ von IBM in den 70er Jahren entwickelt und vom NIST (http://www.nist.gov/index.html) zum Standard erhoben.

☑ Seine 56-Bit Schlüssel (7 Zeichen) sind für den heutigen Stand der Technik zu kurz, weswegen häufig der **3DES** (112 Bit) eingesetzt wird.

☑ 3DES-Anwendungen sind Verschlüsselungen von:

- ☐ Multimedia Datenströmen
- ☐ Unix-Kennwörter, Passphrases, ...
- ☐ **PIN für EC-Karte**.
  Seit 1998 wird 3DES verwendet. Die PIN selbst ist nicht im Klartext auf dem Magnetstreifen.
- ☐ **MAC** (message authentication code) auf Basis von DES (s.u.)
- ☐ **Einweg-Hash-Funktion** auf Basis von DES

DES mit 56 Bit

Die relativ kleine Schlüssellänge von 56 Bit kann mittels Brute Force Angriffen geknackt werden. Es wird vermutet, dass die NSA (bei der Entwicklung von DES beteiligt) absichtlich eine Schlüssellänge auswählte, die von ihren Rechnern in den 70er Jahren problemlos geknackt werden konnte. Durch die rasante Entwicklung ist es mit jedem herkömmlichen Computer möglich eine Brute Force Attacke mit Erfolg zu starten.

3DES mit 112 Bit

Aus diesem Grund erarbeiteten die DES – Entwickler den 3DES – Algorithmus.
Hierbei wird mit dem DES – Algorithmus ein Datenblock zuerst mit einem Schlüssel codiert, mit einem anderen decodiert und zuletzt wieder mit dem ersten Schlüssel chiffriert. Der Algorithmus wird dadurch deutlich sicherer jedoch auch wesentlich langsamer. Der Schlüsselraum vergrößert sich von 2^{56} auf 2^{112}.

1.4.2. DES-Verschlüsselung mit openSSL

OpenSSL bietet neben anderen Funktionalitäten (s. später) eine Menge von C-Funktionen aus dem Bereich der Kryptographie. Wir wollen hier die DES-Verschlüsselung in C kennenlernen.

http://www.codealias.info/technotes/des_encryption_using_openssl_a_simple_example

Datei: test-DES.c

```
/*
file: test-DES.c
http://www.codealias.info/technotes/
des_encryption_using_openssl_a_simple_example

Es werden 2 Funktionen erstellt und getestet:
- Encrypt()
- Decrypt()

Sie benutzen den DES-Algorithmus. Eine symmetrische Block-Chiffre.

gcc test-DES.c -o test-DES.exe -lcrypto

*/

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <openssl/des.h>

char* Encrypt( char *Key, char *Msg, int size){
     static char*    Res;
     int             n=0;
     DES_cblock      Key2;
     DES_key_schedule schedule;

     Res = ( char * ) malloc( size );

     /* Prepare the key for use with DES_cfb64_encrypt */
     memcpy( Key2, Key, 8);
     DES_set_odd_parity( &Key2 );
     DES_set_key_checked( &Key2, &schedule );

     /* Encryption occurs here */
     DES_cfb64_encrypt(
               ( unsigned char * ) Msg,
               ( unsigned char * ) Res,
               size, &schedule, &Key2, &n, DES_ENCRYPT );

      return (Res);
}


char* Decrypt( char *Key, char *Msg, int size){
     static char*    Res;
     int             n=0;
```

```
     DES_cblock      Key2;
     DES_key_schedule schedule;

     Res = ( char * ) malloc( size );

     /* Prepare the key for use with DES_cfb64_encrypt */
     memcpy( Key2, Key,8);
     DES_set_odd_parity( &Key2 );
     DES_set_key_checked( &Key2, &schedule );

     /* Decryption occurs here */
     DES_cfb64_encrypt(
               ( unsigned char * ) Msg,
               (unsigned char * )  Res,
               size, &schedule, &Key2, &n, DES_DECRYPT );

     return (Res);
}


int main() {
     char key[]="password";
     char clear[]="This is a secret message";
     char *decrypted;
     char *encrypted;

     encrypted=malloc(sizeof(clear));
     decrypted=malloc(sizeof(clear));

     printf("Clear text\t : %s \n", clear);

     memcpy(encrypted, Encrypt(key,clear,sizeof(clear)), sizeof(clear));

     printf("Encrypted text\t : %s \n",encrypted);



     memcpy(decrypted,Decrypt(key,encrypted,sizeof(clear)),
                          sizeof(clear));
     printf("Decrypted text\t : %s \n",decrypted);

     return (0);
}
```

Übersetzen mit

```
gcc test-DES.c -o test-DES.exe -lcrypto
```

openSSL installieren

```
sudo apt-get install libssl-dev
```

Aufruf und Ausgabe:

```
$./test-DES.exe
Clear text  : This is a secret message
Encrypted text   : 8��#0t�E�#E� \@�Z9T�)��a�
Decrypted text   : This is a secret message
```

1.4.3. AES (Advenced Encryption Standard)

http://de.wikipedia.org/wiki/Advanced_Encryption_Standard

Bereits 1997 war abzusehen, dass die bisherigen Verfahren für die Zukunft nicht ausreichen würden. So hat das NIST (National Institute of Standards and Technology) eine öffentliche Ausschreibung nach neuen Verfahren „Advanced Encryption Standard“ und deren weltweite Prüfung durchgeführt.

Im Herbst **2000** fiel die Entscheidung auf den Rijndael-Algorithmus, der von Daemen u. Rijmen aus Belgien vorgeschlagen worden war.

Merkmale:

☑ symmetrischer Block-Chiffre

☑ der gleiche Schlüssel k wird zum Ver- und Entschlüsseln verwendet.

☑ Die Blocklänge ist 128

☑ der **Schlüssel kann 128, 192 oder 256** Bits (eher in der Zukunft) haben.

☑ AES ist ein schneller Algorithmus.

☑ Anwendung:

- ☐ **WLAN**
- ☐ IP-Telefonie, Skype
- ☐ **7zip**, RAR, gpg
- ☐ Verschlüsseln von **Platten**
- ☐ Es gibt auch HW-Unterstützung (Intel,AMD,...); wird von openSSL automatisch erkannt und genutzt.

AES-Verschlüsselung selbst erfolgt in Runden. Bei einem 128 Bit Schlüssel werden 10 Runden durchlaufen. Dabei werden jeweils 16 Bytes (=128 Bits) der Nachricht in diesen 16 Runden durch Substitution und Permutation verändert und mit einem rundenspezifischen Teil-Schlüssel mit XOR verknüpft.

1.4.4. AES-Verschlüsselung mit openSSL command line tool

http://users.dcc.uchile.cl/~pcamacho/tutorial/crypto/openssl/openssl_intro.html

openssl bietet nicht nur eine C-API an, sondern bietet auch ein hervorragendes command-line-tool an.

OpenSSL bietet viele Verschlüsselungsverfahren. Mit dem folgenden Befehl können Sie eine Liste anzeigen lassen:

```
openssl list-cipher-commands
aes-128-cbc
aes-128-ecb
aes-192-cbc
aes-192-ecb
aes-256-cbc
aes-256-ecb
base64
...
```

Wir wollen nun den Text "I love OpenSSL!" mit AES verschlüsseln. Dabei wollen wir als Betriebsart für die Blockverschlüsselung den CBC (Cipher Block Chaining) Modus einsetzen. Der Schlüssel soll 256 Bits haben.

Zunächst erzeugen wir eine Datei, die den Klartext enthält:

```
touch plain.txt
echo 'I love OpenSSL!' > plain.txt
```

Nun erfolgt die Verschlüsselung:

```
openssl enc -aes-256-cbc -in plain.txt -out encrypted.bin

enter aes-256-cbc encryption password: hello
Verifying - enter aes-256-cbc encryption password: hello
```

Der eigentliche 256 Bit Schlüssel wird von dem hier sehr schlechten Passwort „hello" abgeleitet.

Hier noch die Entschlüsselung:

```
openssl enc -aes-256-cbc -d -in encrypted.bin -pass pass:hello
I love OpenSSL!
```

Ein besseres Passwort wäre zB:

DgFs100J@W&B [übersetzt: Der gefangene Floh sitzt 100 Jahre bei Wasser und Brot]

Anmerkung zur Betriebsart: CBC

Bei der Block-Chiffre ist der einfachste Betriebsmodus ECB (Electronic Code Book). Dabei werden die einzelnen Datenblöcke absolut unabhängig voneinander verschlüsselt. Dadurch werden identische Klartextblöcke zu identischen Chiffre-Blöcken. Dies ist sicherlich ein Nachteil. Aus diesem Grund hat man andere Betriebsarten entwickelt, die hier Abhilfe geben, indem die Blöcke zueinander in Anhängigkeit gebracht werden.

Bei CBC (Cipher Block Chaining) hängt zum Beispiel ein Cipher-Block von seinem Vorgänger-Cipher-Block ab.

1.4.5. +AES-Verschlüsselung: C Beispiel

http://www.codeplanet.eu/tutorials/cpp/51-advanced-encryption-standard.html

1.4.6. Vor/Nachteile der symmetrischen Verfahren

☑ Der Vorteil bei der symmetrischen Verschlüsselung ist
- ☐ die wesentlich höhere Sicherheit gegenüber klassischen Verschlüsselungsverfahren sowie
- ☐ die bessere **Performance** gegenüber asymmetrischen Verfahren.
- ☐ Des Weiteren kann mit kürzeren Schlüsseln die selbe Sicherheit gewährleistet werden als mit vergleichsweise längeren asymmetrischen Schlüsseln.

☑ Ein großer Nachteil der symmetrischen Verschlüsselung ist, dass
- ☐ jedes Paar von Personen, die separat von anderen Personen, geheime Nachrichten tauschen wollen jeweils einen eigenen Schlüssel benötigen.

  Für die geschützte Kommunikation von N Personen werden **N*(N-1)/2 Schlüssel** benötigt (4950 Schlüssel bei einer Gruppe von 100 Personen).

- ☐ Ein weiterer Nachteil ist, dass Schlüssel geheim bleiben müssen. Um absolute Sicherheit zu gewährleisten, ist es nicht möglich den **Schlüssel** seinem Kommunikationspartner über eine ungesicherte Leitung zu **übermitteln**.

1.5. Übung: Block- und Stromchiffre (C++)

Block- und Stromchiffre werden auch noch Mono- bzw. Polyalphabetische Verfahren genannt. Um die beiden Verfahren zu vertiefen, wollen wir anhand der Programmiersprache C++ eine kleine Klassenbibliothek dazu erstellen.

(s. 02-AB-KRYPTO-ALGOS-ciper-diffie-hellman)

1.6. Übung: Schlüsselaustausch nach Diffie-Hellman

Diffie und Hellman entwickelten ein Verfahren, um das Schlüsselaustauschproblem (ein Dritter hört mit) zu umgehen. Mit Hilfe des Verfahrens einigt man sich über einen unsicheren Kanal auf einen Schlüssel, ohne dass Dritte ebenfalls an den Schlüssel kommen können.

Wir wollen nun diesen Algorithmus genauer kennen lernen und dazu ein Programm erstellen.
(s. 02-AB-KRYPTO-ALGOS-ciper-diffie-hellman)

1.7. Kryptographische Hash-Funktionen

Hash-Funktionen können auf der Basis von Block-Chiffren (DES,AES) arbeiten. Diese sind allerdings aufwendig, sodass man eigenständige kryptographische Hash-Funktionen untersucht/nutzt.

Diese sind:

- MD5
- SHA-1 Hash-Funktionen
- SHA-2 Hash-Funktionen
  - SHA-224, SHA-256, SHA-384, SHA-512

Verwendung von Hash-Funktionen für MAC (Message authentication code):

Eine kleine Änderung der Nachricht erzeugt einen komplett anderen Hash. Diese Eigenschaft wird in der Kryptographie auch als Lawineneffekt bezeichnet.

```
SHA224("Franz jagt im komplett verwahrlosten Taxi quer durch Bayern") =
49b08defa65e644cbf8a2dd9270bdededabc741997d1dadd42026d7b
```

```
SHA224("Frank jagt im komplett verwahrlosten Taxi quer durch Bayern") =
58911e7fccf2971a7d07f93162d8bd13568e71aa8fc86fc1fe9043d1
```

Um Dateien zu vergleichen, verwendet man zB. Auch. Kopieren Sie in einem Verzeichnis eine Datei und verwenden Sie dann den Befehl

```
sha512sum *
```

1.7.1. MD5

http://de.wikipedia.org/wiki/Message-Digest_Algorithm_5

MD5 war lange Zeit sehr in Verwendung.

MD5 ist jetzt aber **nicht mehr** zu empfehlen. Siehe dazu: http://www.md5decrypter.co.uk/

Um die Integrität von Dateien zu ermitteln verwendet man oft noch md5sum:
https://wiki.archlinux.de/title/Md5sum

1.7.2. SHA-2

http://de.wikipedia.org/wiki/SHA-2

SHA-2 (von englisch secure hash algorithm, *sicherer Hash-Algorithmus*) ist die Bezeichnung für die vier kryptologischen Hashfunktionen **SHA-224**, **SHA-256**, **SHA-384** und **SHA-512**, die 2001 vom US-amerikanischen NIST als Nachfolger von SHA-1 standardisiert wurden.

Siehe auch: man sha256sum

1.8. MAC (message authentication code) auf Basis von DES

MAC kann als

1. kryptographischer Fingerabdruck aber auch als
2. kryptographische Prüfinformation einer langen Nachricht im Klartext verwendet werden.

☑ Der MAC wird vom Sender erzeugt und kann vom Empfänger überprüft werden.

☑ Sender und Empfänger kennen den gleichen geheimen Schlüssel.

☑ Ein MAC gewährleistet Authentizität und Integrität.

☑ ISO9797, ISO8730:
Ein MAC auf der Basis von DES ist 64 Bit lang. Die Nachricht m wird also in 64 Bit lange Blöcke m_i unterteilt und mit DES verschlüsselt (k ... Schlüssel). Wie die folgende Formel zeigt, werden die verschlüsselten Blöcke in die Verschlüsselung selbst einbezogen.

MAC (k, m) := $DES_k(m_1 \text{ XOR } DES_k(m_2 \text{ XOR } \ldots \text{ XOR } DES_k(m_i)))$

☑ Der Klartext m wird zusammen mit dem MAC versendet:
[m, MAC(k,m)]

☑ Der MAC wird anschließend vom Empfänger geprüft.
Dies ist möglich, weil er den geheimen Schlüssel kennt und somit selbst den MAC berechnen und diesen mit dem gesendeten MAC vergleichen kann.

☑ Der MAC kann statt DES auch andere Algorithmen verwenden.
ZB: IDEA oder AES. Im Falle des AES wäre dann der MAC 128 Bit lang. (s.u.)

1.8.1. Fragen: MAC

☑ Die Änderung einer einzigen Binärstelle in der Nachricht ändert den MAC nicht|*dramatisch.

☑ Änderung in der Reihenfolge der Blöcke ….m_i,m_{i+1}, … ändert den MAC nicht|*dramatisch.

☑ Der MAC ist *leicht|schwer zu berechnen, wenn der Schlüssel k bekannt ist.

☑ Ein Angreifer kann aus Kenntnis von m und MAC(k,m) den Schlüssel k praktisch|*praktisch nicht finden.

☑ Ein Angreifer kann aus Kenntnis von m und MAC(k,m) , aber ohne Kenntnis des Schlüssels k, *keine | eine manipulierte Nachricht m' mit gültigem MAC(k, m') konstruieren.

☑ Eine Nachricht m mit MAC(k,m) gewährleistet *Integrität und *Authentizität beim Empfänger, jedoch keine *Verbindlichkeit.

☑ Warum kann bei der Verwendung von MAC(k,m) der Sicherheits-Dienst Verbindlichkeit nicht

gewährleistet?
A: Weil MAC eine symmetrische Verschlüsselung (gemeinsame Verwendung des geheimen Schlüssel) verwendet.

☑ Ein Empfänger erhält eine Nachricht m zusammen mit ihrem Message Authentication Code MAC [m, MAC(k,m)]. Warum ist sich der Empfänger sicher, von wem die Nachricht stammt und dass sie unverändert ist?

A: geheimer symmetrischer Schlüssel. Nur A und B können die richtige MAC bilden. Falls die Nachricht verändert worden wäre, sind die von beiden berechneten MAC auch unterschiedlich.

1.9. Implementierungen

1.9.1. Aufgabe: Implementieren in C++ (AES und RSA)

Aufgabe:

Laden Sie crypto.h , crypto.cpp, ... von https://github.com/shanet/Crypto-Example und bringen Sie die Testprogramme crypto-example*.cpp zum Laufen.

Beschreibung: siehe http://shanetully.com/2012/06/openssl-rsa-aes-and-c/

1.9.2. Implementieren in Java

http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html