Benjamin Leonhardt

Matrikel-Nr.: 7004553

benjaminleonhardt@gmx.de

KeyManager – Passwort-tresor für Raspberry Pi

Projektarbeit



Inhaltsverzeichnis

[1. Einleitung 2](#_Toc466230408)

[2. Verschlüsselungsalgorithmus 3](#_Toc466230409)

[2.1. Der AES-Algorithmus 3](#_Toc466230410)

[2.1.1. AddRoundKey 5](#_Toc466230411)

[2.1.2. SubBytes 6](#_Toc466230412)

[2.1.3. ShiftRows 7](#_Toc466230413)

[2.1.4. MixColumns 7](#_Toc466230414)

[2.1.5. Entschlüsseln 8](#_Toc466230415)

[2.1.6. InvShiftRows 9](#_Toc466230416)

[2.1.7. InvSubBytes 10](#_Toc466230417)

[2.1.8. invMixColumns 10](#_Toc466230418)

[2.1.9. ExpandKey 11](#_Toc466230419)

[2.2. Blockchiffre und Stromchiffre 12](#_Toc466230420)

[2.2.1. ECB – Electronic Code Book 12](#_Toc466230421)

[2.2.2. CBC – Cipher Block Chaining 13](#_Toc466230422)

[2.3. Initialisierungsvektor 14](#_Toc466230423)

[2.3.1. Null-Initialisierungsvektor 14](#_Toc466230424)

[2.3.2. Random-Initialisierungsvektor 14](#_Toc466230425)

[2.4. Paddingverfahren 15](#_Toc466230426)

[2.4.1. Null-Padding 15](#_Toc466230427)

[2.4.2. ANSI X.923 15](#_Toc466230428)

[2.4.3. PKCS7 16](#_Toc466230429)

[3. SHA256 16](#_Toc466230430)

[3.1. Algorithmus des SHA256 16](#_Toc466230431)

[3.2. SHA256 Paddingverfahren 19](#_Toc466230432)

[3.3. Passwort als Hash 20](#_Toc466230433)

[3.4. Hash zur Integrationssicherung 21](#_Toc466230434)

[4. Aufbau der Datenbank 21](#_Toc466230435)

[5. Aufbau der Datei 22](#_Toc466230436)

[5.1. Der Header 22](#_Toc466230437)

[5.2. Die Passwortsignatur 23](#_Toc466230438)

[5.3. Dateistruktur 23](#_Toc466230439)

[6. Fazit 24](#_Toc466230440)

[Abbildungsverzeichnis 25](#_Toc466230441)

[Listingsverzeichnis 26](#_Toc466230442)

[Quellenverzeichnis 27](#_Toc466230443)

# Einleitung

Im Rahmen der Lehrveranstaltung Projektarbeit, wurde ein Passwort-Tresor-Programm namens „KeyManager“ erstellt. Es handelt sich um ein Konsolen-Programm[[1]](#footnote-1), das sowohl für Windows als auch für Linux kompilierbar ist. Des Weitern läuft KeyManager auf ARM-Prozessoren so wie x86-/AMD64-Prozessoren.

Es ist ein Konsolen-Programm, da das Ziel des Projektes war, diesen Passwort-Tresor für den Raspberry Pi zu optimieren. Da dieser nicht viele Rechenpower und Speicher besitzt, wurde das Programm auf das Nötigste begrenzt und hat daher keine grafische Benutzeroberfläche.

Das Programm ermöglicht dem Benutzer, Passwörter mitsamt Titel (wofür das Passwort verwendet wird, wie z. B. E-Mail-Adresse, Webseite oder IP eines Rechners), Benutzername und dem Passwort zu speichern. Das Passwort kann dabei selbst bestimmt oder mit beliebiger Länge zufällig generiert werden. Dank der Kopierfunktion von KeyManager, mit der Benutzer beim Einloggen in die in der Datenbank hinterlegten Konten Einträge per Copy-and-paste-Prinzip aus der Zwischenablage nutzen können, wird eine effiziente Handhabung gewährleistet.

Verschlüsselt wird die Datenbank grundsätzlich mit dem AES-Algorithmus (siehe Kap. 2.1). Um die Sicherheit weiter zu erhöhen, kann die Datenbank neben dem ECB[[2]](#footnote-2)-Modus (siehe Kap. 2.2.1) auch im CBC[[3]](#footnote-3)-Modus (siehe Kap. 2.2.2) operieren und so die verschlüsselten Blöcke der AES-Blockchiffre miteinander verknüpfen. Der CBC-Modus kann mit einem Initialisierungsvektor (siehe Kap. 2.3) gestartet werden. Der IV kann manuell oder automatisch generiert werden. Der Benutzer hat beim Erstellen der Datenbank, darüber hinaus auch Einfluss, welche Padding-Methode (siehe Kap. 2.4) verwendet werden soll.

Da Verschlüsselung ein sehr komplexes und umfangreiches Thema ist und die programminternen Begrifflichkeiten für die meisten Benutzer nicht selbsterklärend sind, wird an allen Stellen des Programms, an denen sich der Benutzer für eine Eingabe entscheiden muss, eine Hilfe angeboten. Darüber hinaus wird stets eine Einstellungsempfehlung angezeigt, die die größtmögliche Sicherheit bietet, um für unerfahrene Benutzer die Bedienung von KeyManager so einfach wie möglich zu gestalten.

Die KeyManager Passwort-Datenbank wird mit einem sogenannten Master-Passwort gesichert. Dieses Master-Passwort, mit dem man sich in die Datenbank einloggen kann, wird mit dem SHA256-Algorithmus (siehe Kap. 3) gehasht. Dieser Algorithmus wird auch zur Fehlererkennung bei beschädigten Datenbanken sowie zur Erkennung von böswilligen Datenbankmanipulationen verwendet (siehe Kap. 3.4).

# Verschlüsselungsalgorithmus

Die Wahl des Verschlüsselungsalgorithmus ist keine triviale Frage. Aus den Unmengen an Algorithmen muss ein passender herausgesucht werden. Der AES[[4]](#footnote-4) ist eine Blockchiffre, die sehr wenige Ressourcen benötigt und sogar auf einem 8-Bit-Rechner ausgeführt werden kann (Müller, 2011). Darüber hinaus, ist dies der Algorithmus der vom NIST[[5]](#footnote-5) und dem BSI[[6]](#footnote-6) aktuell empfohlene und gilt als sicher (BSI, kein Datum). Der AES-Algorithmus ist ein offizieller Standard und ist im FIPS-197-Dokument zu finden (FIPS PUPS, 2001). FIPS steht für Federal Information Processing Standards und sind Dokumente in denen Standards erklärt werden. Diese werden vom NIST erstellt und sind die Gesetzlichen Vorgaben in Amerika zur Implementierung von Standards innerhalb der staatlichen Behörden. Die NIST ist dem der PTB[[7]](#footnote-7) in Deutschland ähnlich. Das FIPS-197-Dokument beinhaltet alle Vorgaben sowie Algorithmen, die zum Implementieren des AES-Algorithmus benötigt werden.

## Der AES-Algorithmus

Der AES ist eine Blockchiffre. Das heißt, AES verschlüsselt mit einem Schlüssel, 128-Bit lange Klartext-Blöcke, zu einem 128-Bit langen verschlüsselten Chiffre-Block. Da es sich um ein symmetrisches Verschlüsselungsverfahren handelt, wird der verschlüsselte Block mit demselben Schlüssel wieder entschlüsselt. Abbildungen 1 und 2 veranschaulichen dies.

**Verschlüsselung**

Verschlüsseln mit Schlüssel K

**Klartext Chiffre**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 |  | Byte 1‘ | Byte 2‘ | Byte 3‘ | Byte 4‘ |
| Byte 5 | Byte 6 | Byte 7 | Byte 8 |  | Byte 5‘ | Byte 6‘ | Byte 7‘ | Byte 8‘ |
| Byte 9 | Byte 10 | Byte 11 | Byte 12 |  | Byte 9‘ | Byte 10‘ | Byte 11‘ | Byte 12‘ |
| Byte 13 | Byte 14 | Byte 15 | Byte 16 |  | Byte 13‘ | Byte 14‘ | Byte 15‘ | Byte 16‘ |

Abbildung : Verschlüsselung mit AES

**Entschlüsselung**

Entschlüsseln mit Schlüssel K

**Klartext Chiffre**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 |  | Byte 1‘ | Byte 2‘ | Byte 3‘ | Byte 4‘ |
| Byte 5 | Byte 6 | Byte 7 | Byte 8 |  | Byte 5‘ | Byte 6‘ | Byte 7‘ | Byte 8‘ |
| Byte 9 | Byte 10 | Byte 11 | Byte 12 |  | Byte 9‘ | Byte 10‘ | Byte 11‘ | Byte 12‘ |
| Byte 13 | Byte 14 | Byte 15 | Byte 16 |  | Byte 13‘ | Byte 14‘ | Byte 15‘ | Byte 16‘ |

Abbildung : Entschlüsselung mit AES

Der AES besteht aus mehreren nacheinander ausgeführten Funktionen, die den 128-Bit-Block entsprechend verarbeiten. Bevor in die Hauptschleife gesprungen wird, wird der Rundenschlüssel mit dem Klartextblock verknüpft (siehe Kap. 2.1.1). Dafür gibt es die Funktion AddRoundKey(0). Diese wird zur Initialisierung vor dem Verschlüsseln ausgeführt. Dabei wird der Schlüssel mit dem zu verschlüsselnden Block ver-X-Odert. Die 0, die als Argument übergeben wird, entspricht hierbei der Rundennummer und gibt an, von welchem Index der Schlüssel aus dem RoundKey-Array (siehe Kap. 2.1.9) verwendet werden soll.

Ein Durchgang in der Hauptschleife des AES-Algorithmus entspricht einer Runde. In der implementierten Variante, die einen Schlüssel mit der Länge von 128-Bit hat, werden 10 Runden durchlaufen. Eine Runde ist wie folgt aufgebaut:

SubBytes();

ShiftRows();

mixColumns();

AddRoundKey(i);

Die letzte Runde unterscheidet sich von den anderen 9 Runden dadurch, dass MixColumns nicht aufgerufen wird. Die 10. Runde sieht anschließend wie folgt aus:

ShiftRows();

SubBytes();

AddRoundKey(Runde);

Listing 1 – Die Hauptschleife des AES-Algorithmus:

void AES::verschluesseln() {

AddRoundKey(0);

for (int i = 1; i <= 9; i++) {

SubBytes();

ShiftRows();

mixColumns();

AddRoundKey(i);

}

ShiftRows();

SubBytes();

AddRoundKey(nr);

}

### AddRoundKey

AddRoundKey verknüpft den Schlüssel mit dem Chiffrat. Dabei handelt es sich um eine einfache XOR-Verknüpfung. Die an die Funktion übergebe Rundenzahl, entscheidet dabei, welcher Teil des Schlüssels verwendet wird.

Der Schlüssel wird mit expandKey() (siehe Kap. 2.1.9) auf die 11-fache Größe erweitert. Dabei handelt es sich um ein Array mit 44 Spalten. In den ersten 4 Spalten steht der Original-Schlüssel. Die übrigen 40 Spalten werden durch das Expand-Verfahren der Methode expandKey gefüllt. In jeder durchlaufenen Runde wird der Bereich des zur XOR-Verknüpfung verwendeten Schlüssels um 4 Spalten verschoben. In der ersten Runde wird Spalte 4 bis 7 mit dem Chiffrat verknüpft; in Runde 2 wird 8 bis 11 verknüpft und so weiter. Da es sich bei den 4 verwendeten Spalten bei beiden Blöcken dann um 128-Bit große Blöcke handelt, können diese 1:1 verknüpft werden.

Listing 2 – AddRoundKey:

void AES::AddRoundKey(unsigned int round) {

s[0].dasWort ^= Roundkey[round \* 4].dasWort;

s[1].dasWort ^= Roundkey[round \* 4 + 1].dasWort;

s[2].dasWort ^= Roundkey[round \* 4 + 2].dasWort;

s[3].dasWort ^= Roundkey[round \* 4 + 3].dasWort;

}

Das Array s aus Listing 2 ist das aktuelle Chiffrat, auch State genannt, also der aktuelle Status des 128-Bit-Blocks, der verschlüsselt wird.

Das Union „Wort“ (siehe Listing 3) ist eine Datenstruktur die 4 Bytes in einer Zeile darstellen. Union ist eine Spezialität von C/C++. Hierbei überlappen sich alle darin befindlichen Variablen. Wenn eine Variable geändert wird, ändern sich alle anderen automatisch mit, da alle auf dieselbe Speicherstelle zeigen. Dabei überlappen sich der Integer „dasWort“ (wird in Listing 2 verwendet), der eine 32-Bit Integer-Variable ist, und die Struct mit ihren 4 darin befindlichen Bytes, die im Speicher hintereinander liegen und dadurch auch 32 Bit belegen. Jede darin befindliche Variable interpretiert nur die Bits, die sich in ihrem Speicherbereich befinden. Dies wird in Abbildung 3 verdeutlicht. Im oberen Rahmen wird die Union in einer Big-Endian-Umgebung dargestellt und unten in einer Little-Endian-Umgebung. Die Bytes und der Integer überlappen sich.

**Union Wort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Byte 0 | Byte 1 | Byte 2 | Byte 3 | 🡨 Wort Big Endian |
| 32-Bit integer dasWort | | | |  |
|  |  |  |  |  |
| Byte 3 | Byte 2 | Byte 1 | Byte 0 | 🡨Wort Little Endian |
| 32-Bit integer dasWort | | | |  |

Abbildung : Das Union „Wort“

Listing 3 – Union Wort:

union wort {

int dasWort;

struct {

#ifdef ARCG\_BIG\_ENDIAN

uint8 byte3;

uint8 byte2;

uint8 byte1;

uint8 byte0;

#else

uint8 byte0;

uint8 byte1;

uint8 byte2;

uint8 byte3;

#endif

};  
};

Der Vorteil hierbei liegt darin, dass man Big und Little Endian durch KeyManager selbst ausgleichen kann und somit eine flexiblere Datenstruktur besitzt. Ändert man das Byte 2 einer Union Wort, wird egal, ob die Hardware Big Endian (ARM / Raspberry Pi) oder Little Endian (Intel / PC) verwendet, das jeweils korrekte Byte in der Integer-Variable mitgeändert. Somit ist es möglich, eine Datenbank auf einem PC zu erstellen und auf einen Raspberry PI zu kopieren und dort damit zu arbeiten und vice versa. Dem Compiler werden durch Präprozessor-Direktiven mitgeteilt, welche der beiden Arten kompiliert werden sollen. Dazu wird die Flag ARCG\_BIG\_ENDIAN (siehe Listing 3) durch die ifdef abgefragt, somit weiß der Compiler, welche der beiden Byte-Anordnungen er nehmen soll. Uint8 ist ein Typedef von Char, welcher der 8-Bit-Datentyp in C/C++ ist.

### SubBytes

SubBytes() tauscht die Bytes des States gegen die in der SBox hinterlegten aus. Die SBox ist ein 256 Byte großes Array mit vordefinierten Werten. Dieses kann in der FIPS-197-Dokumentation (FIPS PUPS, 2001, S.16) nachgeschlagen werden und wird hier der Übersicht wegen nicht aufgeführt.

Dabei wird der Inhalt des Bytes, das getauscht werden soll, als unsigned Integer interpretiert und als Index für die SBox verwendet. Es wird also s[0].byte0 = sbox[s[0].byte0] durchgeführt. Unsigned Integer ist hier wichtig, da die Werte, die mit 1en im höchstwertigen Bit sind, sonst als negative Zahlen interpretiert würden, also die Zahlenrange von -128 bis 127 verlaufen würde. -128 ist allerdings kein gültiger Index für ein Array, wäre also „out of bounds“.

In einem konkreten Beispiel dargestellt, wird ein ASCII-Buchstabe ‚C‘, der in Byte 0, steht als Zahl interpretiert. „C“ entspricht laut ASCII-Tabelle der Zahl 67 oder 0x43. Es wird also Byte 0 gegen den in der SBox im Index 67(0x43) vermerkten Wert getauscht. Die Tabelle im FIPS-197-Dokument (FIPS PUPS, 2001, S.16) kann wie folgt gelesen werden:

Das Byte in Hexadezimal wird als xy-Koordinate interpretiert. Dabei gibt x die Zeile und y die Spalte an. Es wird also der ursprüngliche Wert 67(0x43) gegen das, was in Zeile 4(0x4) Spalte 3(0x3) steht, getauscht, also gegen den Wert 133(0x1a).

Listing 4 – SubBytes:

void AES::SubBytes() {

for (int i = 0; i<4; i++) {

s[i].byte0 = sbox[s[i].byte0];

s[i].byte1 = sbox[s[i].byte1];

s[i].byte2 = sbox[s[i].byte2];

s[i].byte3 = sbox[s[i].byte3];

}

}

### ShiftRows

ShiftRows verschiebt die Zeilen um n Bytes. n ist dabei abhängig von der Zeilenzahl. Die oberste Zeile ist Zeile 0. Diese bleibt unverändert. Die nächste Zeile wird um 1 Byte verschoben, die darauf folgende um 2 und die letzte um 3 Bytes. Bytes, die links aus dem Block hinausgeschoben werden, werden rechts wieder eingeschoben.

**ShiftRows als Darstellung**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 |  | Byte 1 | Byte 2 | Byte 3 | Byte 4 |
| Byte 5 | Byte 6 | Byte 7 | Byte 8 |  | Byte 6 | Byte 7 | Byte 8 | Byte 5 |
| Byte 9 | Byte 10 | Byte 11 | Byte 12 |  | Byte 11 | Byte 12 | Byte 9 | Byte 10 |
| Byte 13 | Byte 14 | Byte 15 | Byte 16 |  | Byte 16 | Byte 13 | Byte 14 | Byte 15 |

Abbildung : ShiftRows

Listing 5 – ShiftRows:

void AES::ShiftRows() {

char tmp, tmp2;

tmp = s[0].byte1; //zeile 0 muss nicht geshifted werden

s[0].byte1 = s[1].byte1; //zeile 1 wird um eins geschoben

s[1].byte1 = s[2].byte1; //(a0,a1,a2,a3)->(a1,a2,a3,a0)

s[2].byte1 = s[3].byte1;

s[3].byte1 = tmp;

tmp = s[1].byte2; //zeile 2 wird um 2 geschoben

tmp2 = s[0].byte2;

s[0].byte2 = s[2].byte2; //(a0,a1,a2,a3)->(a2,a3,a0,a1)

s[1].byte2 = s[3].byte2;

s[2].byte2 = tmp2;

s[3].byte2 = tmp;

tmp = s[0].byte3; //zeile 3 wird um 3 geschoben

s[0].byte3 = s[3].byte3; //(a0,a1,a2,a3)->(a3,a0,a1,a2)

s[3].byte3 = s[2].byte3;

s[2].byte3 = s[1].byte3;

s[1].byte3 = tmp;

}

### MixColumns

MixColumns ist die komplizierteste Funktionen des AES-Algorithmus. Hier werden die Zeilen in einer vorgegebenen Art verrechnet und verodert. Die Funktion „Gmulb“ spielt hier eine entscheidende Rolle, die zuerst erläutert werden sollte.

Gmulb multipliziert die Bytes, die ihr übergeben werden, mit dem Faktor, der ebenfalls übergeben wird. Zum Beispiel: gmulb(tmp[i].byte0, 2). Linkes Argument (tmp[i].byte0) ist dabei das Byte und das rechte Argument (2) der Faktor. Gmulb betrachtet das übergebene Byte als Polynom, welches mit dem Faktor multipliziert und Modulo 0x11B genommen wird, da es sonst überlaufen würde, also größer 256 werden würde. 0x11B ist ein Polynom 8ten Grades. Es wurde so gewählt, damit nach der Modulo Operation das Ergebnis wieder in ein Byte passt. 0x11B ist ein irreduzibles Polynom das vorgebeben wird. Diese Vorgabe ist im FIPS-197 Dokument (FIPS PUPS, 2001, S.10) zu finden.

Ein Polynom ist irreduzibel, wenn es nur durch sich selbst und 1 teilbar ist. Das heißt, es ist eine Primzahl als Polynom. MixColumns hat die in Abbildung 5 abgebildete Form.

**MixColumns als Darstellung**

|  |  |
| --- | --- |
| Byte 1,C | 🡨 Byte 1,C • 0x2 ^ Byte 2,C • 0x3 ^ Byte 3,C ^ Byte 4,C |
| Byte 2,C | 🡨 Byte 1,C ^ Byte 2,C • 0x2 ^ Byte 3,C • 0x3 ^ Byte 4,C |
| Byte 3,C | 🡨 Byte 1,C ^ Byte 2,C ^ Byte 3,C • 0x2 ^ Byte 4,C • 0x3 |
| Byte 4,C | 🡨 Byte 1,C • 0x3 ^ Byte 2,C ^ Byte 3,C ^ Byte 4,C • 0x2 |

Abbildung : MixColumns

In Abbildung 5 ist die Berechnung zu sehen, die der Algorithmus durchläuft. Dabei steht die Variable C für die Spalte im State-Block. C läuft von 1 bis 4 durch. Somit wird die oben dargestellte Berechnung für jede Spalte des State-Blocks durchgeführt. Der •-Operator steht für eine Multiplikation, welche die mit der Funktion Gmulb berechnet wird. Der ^-Operator ist eine Oder-Verknüpfung. Somit wird aus jeder Berechnung ein maximal 8 Bit größer Wert und passt folglich wieder in das Byte.

Listing 6 – mixColumns:

void AES::mixColumns() {

wort tmp[4];

for (int i = 0; i<4; i++) {

tmp[i] = s[i];   
 s[i].byte0=gmulb(tmp[i].byte0,2)^gmulb(tmp[i].byte1,3)^tmp[i].byte2^tmp[i].byte3;

s[i].byte1=tmp[i].byte0^gmulb(tmp[i].byte1,2)^gmulb(tmp[i].byte2,3)^tmp[i].byte3;

s[i].byte2=tmp[i].byte0^tmp[i].byte1^gmulb(tmp[i].byte2,2)^gmulb(tmp[i].byte3,3);

s[i].byte3=gmulb(tmp[i].byte0,3)^tmp[i].byte1^tmp[i].byte2^gmulb(tmp[i].byte3,2);

}

}

### Entschlüsseln

Zum Entschlüsseln wird folgendermaßen vorgegangen: Zuerst wird der Rundenschlüssel mit dem Chiffrat verknüpft. Dies wird in der Funktion AddRoundKey() durchgeführt. Der Funktion wird 10 als Argument übergeben, danach wird in die Hauptschleife zum Entschlüsseln gesprungen. Diese zählt ihre Laufvariable von 9 auf 1 runter. In ihr werden die Umkehrfunktionen (siehe Kap. 2.1.6 – 2.1.8) aufgerufen.

invShiftRows();

invSubBytes();

AddRoundKey(i);

invMixColumns();

Wie beim Verschlüsseln, wird in der letzten Runde wieder MixColumns ausgelassen. Die Funktionen werden also wie folgt aufgerufen:

invSubBytes();

invShiftRows();

AddRoundKey(0);

Listing 7 – Die Entschlüsselungshauptschleife des AES-Algorithmus:

void AES::entschluesseln() {

AddRoundKey(nr);

for (unsigned int i = 9; i >= 1; i--) {

invShiftRows();

invSubBytes();

AddRoundKey(i);

invMixColumns();

}

invSubBytes();

invShiftRows();

AddRoundKey(0);

}

Da eine XOR-Verknüpfung, wenn man sie 2 Mal ausgeführt, wieder zum ursprünglichen Wert führt, ist AddRoundKey beim Ver- sowie Entschlüsseln die gleiche Funktion. Durch das Dekrementieren der Rundenzahl, die der Funktion AddRoundKey als Argument übergeben wird, wird hier die Operation umgedreht.

### InvShiftRows

InvShiftRows schiebt die Zeilen des States in die umgekehrte Richtung wie ShiftRows und stellt somit den Originalzustand wieder her. Es ist also ein Rechts-Shift. Die Zeilenzahl ist auch hier für die Anzahl, wie viele Bytes geschoben werden sollen, ausschlaggebend. Die erste Zeile wird erneut als Zeile 0 betrachtet und wie zuvor 0 Bytes nach rechts geschoben. Zeile 1 wird 1 Byte geschoben, Zeile 2 wird 2 Bytes und Zeile 3 3 Bytes.

**InvShiftRows als Darstellung**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 |  | Byte 1 | Byte 2 | Byte 3 | Byte 4 |
| Byte 5 | Byte 6 | Byte 7 | Byte 8 |  | Byte 8 | Byte 5 | Byte 6 | Byte 7 |
| Byte 9 | Byte 10 | Byte 11 | Byte 12 |  | Byte 11 | Byte 12 | Byte 9 | Byte 10 |
| Byte 13 | Byte 14 | Byte 15 | Byte 16 |  | Byte 14 | Byte 15 | Byte 16 | Byte 13 |

Abbildung : InvShiftRows

Listing 8 – invShiftRows:

void AES::invShiftRows() {

char tmp, tmp2;

tmp = s[0].byte1; //zeile 0 muss nicht geshifted werden

s[0].byte1 = s[3].byte1; //zeile 1 wird um eins geschoben

s[3].byte1 = s[2].byte1;

s[2].byte1 = s[1].byte1; //(a0,a1,a2,a3)->(a3,a0,a1,a2)

s[1].byte1 = tmp;

tmp = s[1].byte2; //zeile 2 wird um 2 geschoben

tmp2 = s[0].byte2;

s[0].byte2 = s[2].byte2; //(a0,a1,a2,a3)->(a2,a3,a0,a1)

s[1].byte2 = s[3].byte2;

s[2].byte2 = tmp2;

s[3].byte2 = tmp;

tmp = s[0].byte3; //zeile 3 wird um 3 geschoben

s[0].byte3 = s[1].byte3; //(a0,a1,a2,a3)->(a1,a2,a3,a0)

s[1].byte3 = s[2].byte3;

s[2].byte3 = s[3].byte3;

s[3].byte3 = tmp;

}

### InvSubBytes

Als Gegenstück zur SBox, gibt es die SBoxInv. Die SBoxInv tauscht also die Bytes, welche von der Sbox substituiert wurden, wieder in ihren ursprünglichen Wert zurück. Wenn man das oben aufgeführte Beispiel aus Kap. 2.1.2 betrachtet, bei dem der ASCII-Buchstabe „C“ mit dem Wert 63(0x43) in den Hex-Wert (133)0x1a getauscht wurde, dann wird der Wert (133)0x1a in der SBoxInv wieder in den für den ASCII Buchstaben „C“ passenden Hex-Wert zurückgetauscht. Betrachtet man also wieder den hexadezimalen Wert als xy-Koordinate, steht in Zeile 1(0x1) Spalte 10(0xa) von SBoxInv also 67(0x43). Die Tabelle der SBoxInv ist im FIPS-197-Dokument (FIPS PUPS, 2001, S.22) aufgeführt.

Listing 9 – invSubBytes:

void AES::invSubBytes() {

for (int i = 0; i<4; i++) {

s[i].byte0 = sboxinv[s[i].byte0];

s[i].byte1 = sboxinv[s[i].byte1];

s[i].byte2 = sboxinv[s[i].byte2];

s[i].byte3 = sboxinv[s[i].byte3];

}

}

### invMixColumns

Die Inverse von MixColumns benutzt ebenfalls die Multiplikationsfunktion Gmulb. Diese wird allerdings mit anderen Multiplikationswerten aufgerufen. Durch die veränderten Werte können die ursprünglichen Spalten wiederhergestellt werden.

Die Berechnung lautet wie in Abbildung 7 dargestellt.

|  |  |
| --- | --- |
| Byte 1,C | 🡨 Byte 1,C • 0xE ^ Byte 2,C • 0xB ^ Byte 3,C • 0xD ^ Byte 4,C • 0x9 |
| Byte 2,C | 🡨 Byte 1,C • 0x9 ^ Byte 2,C • 0xE ^ Byte 3,C • 0xB ^ Byte 4,C • 0xD |
| Byte 3,C | 🡨 Byte 1,C • 0xD ^ Byte 2,C • 0x9 ^ Byte 3,C • 0xE ^ Byte 4,C • 0xB |
| Byte 4,C | 🡨 Byte 1,C • 0xB ^ Byte 2,C • 0xD ^ Byte 3,C • 0x9 ^ Byte 4,C • 0xE |

Abbildung : invMixColumns

Die Variable C läuft hier analog zu MixColumns von 1 bis 4, also über jede Spalte im State-Block.   
Der •-Operator wird in Kapitel 2.1.4 erläutert.

Listing 10 – invMixColumns:

void AES::invMixColumns() {

wort tmp[4];

for (int i = 0; i <= 3; i++) {

tmp[i] = s[i];

s[i].byte0 = gmulb(tmp[i].byte0, 0x0e) ^ gmulb(tmp[i].byte1, 0x0b) ^   
gmulb(tmp[i].byte2, 0x0d) ^ gmulb(tmp[i].byte3, 0x09);

s[i].byte1 = gmulb(tmp[i].byte0, 0x09) ^ gmulb(tmp[i].byte1, 0x0e) ^   
gmulb(tmp[i].byte2, 0x0b) ^ gmulb(tmp[i].byte3, 0x0d);

s[i].byte2 = gmulb(tmp[i].byte0, 0x0d) ^ gmulb(tmp[i].byte1, 0x09) ^   
gmulb(tmp[i].byte2, 0x0e) ^ gmulb(tmp[i].byte3, 0x0b);

s[i].byte3 = gmulb(tmp[i].byte0, 0x0b) ^ gmulb(tmp[i].byte1, 0x0d) ^   
gmulb(tmp[i].byte2, 0x09) ^ gmulb(tmp[i].byte3, 0x0e);

}

}

### ExpandKey

ExpandKey vergrößert künstlich durch einen Algorithmus den 128-Bit- bzw. 16-Byte-Key zu einem 110-Byte-RoundKey. RoundKey bedeutet, dass jede Runde ein anderer Teil des vergrößerten Schlüssels verwendet wird, um ihn mit dem Chiffrat zu verknüpfen.

ExpandKey kopiert erst den originalen Key in die ersten 16 Byte. Diese werden als Wörter à 4 Byte gesehen und mit der Wort Union (siehe Kap. 2.1.1) als Array implementiert. Der Algorithmus geht also Wort für Wort durch das Array. Dabei wird zuerst eine temporäre Kopie des zuletzt in RoundKey gefüllten Wortes gemacht. In jedem 4. Schleifendurchlauf wird das Wort um 1 Byte nach rechts rotiert. Danach werden alle 4 Bytes des Wortes durch jene der SBox ausgetauscht und das gesamte Wort mit dem Wert des rcon-Array aus dem Index i / nk-1 verodert. i ist dabei der Schleifenzähler und nk ist die Größe des Schlüssels, der in AES zum Verschlüsseln verwendet wird. Dieser kann laut FIPS-197 (FIPS PUPS, 2001) 128, 196 oder 256 Bit umfassen, welche den Werten 4, 5 und 6 für die Variable nk entsprechen. nk ist in KeyManager stets 4, da KeyManager mit 128 Bit Schlüsselgröße arbeitet. Die Werte, mit denen Rcon[[8]](#footnote-8) gefüllt wird, können in der FIPS-197-Dokumentation (FIPS PUPS, 2001, S.27) nachgeschlagen werden. Diese werden am Ende noch, und zwar in jedem Schleifendurchlauf, auf das temporär gespeicherte Wort, mit dem Wort des RoundKey an der Stelle i – nk verodert. i und nk entsprechen den oben erwähnten Werten. Da die Schleife bei 4 anfängt, ist diese Rechnung kein Problem, da beim ersten Schleifendurchlauf i = 4 ist und nk = 4. Es wird also bei i – nk auf das Element im Index 0 zugegriffen.

Listing 11 – expandKey:

void AES::expandKey() {

Roundkey[0].dasWort = key[0].dasWort;

Roundkey[1].dasWort = key[1].dasWort;

Roundkey[2].dasWort = key[2].dasWort;

Roundkey[3].dasWort = key[3].dasWort;

wort tmp;

for (int i = 4; i < 44; i++) {

tmp.dasWort = Roundkey[i - 1].dasWort;

if (i % nk == 0) { //immer für das erste Wort also alle 4 wörter bei 128Bit

tmp.dasWort = (tmp.dasWort << 24) | (tmp.dasWort >> 8); //wort rotieren (b0,b1,b2,b3)->(b1,b2,b3,b0)

tmp.byte0 = sbox[tmp.byte0]; //mit sbox sublementieren

tmp.byte1 = sbox[tmp.byte1];

tmp.byte2 = sbox[tmp.byte2];

tmp.byte3 = sbox[tmp.byte3];

tmp.dasWort ^= rcon[i / nk - 1]; //mit rcon verodern

}

Roundkey[i].dasWort = Roundkey[i - nk].dasWort ^ tmp.dasWort;

//jedes wort mit dem wort -4 verodert (schleife fängt bei 4 an)

}

}

## Blockchiffre und Stromchiffre

AES an sich verschlüsselt, wie oben bereits erwähnt, beim Durchlaufen des Algorithmus genau einen 128-Bit-Klartextblock zu einem 128-Bit-Chiffreblock. Diese Art des Verschlüsselns nennt man Blockchiffre. Da allerdings 128Bit einen sehr begrenzten Speicherraum darstellen, muss man das zu speichernde in mehrere Blöcke aufspalten und diese aneinander hängen. Dieses Aneinanderhängen nennt man Modus.

Es gibt verschiedene Arten dieser Modi, die die Sicherheit des Chiffrats stark beeinflussen. In KeyManager sind 2 Modi implementiert: ECB und CBC. Somit wird aus einer Blockchiffre eine Stromchiffre.

### ECB – Electronic Code Book

Der ECB-Modus ist das reine Aneinanderhängen der verschlüsselten Blöcke. Somit kann man sehr einfach aus einer Bockchiffre mit 128Bit Länge eine Stromchiffre beliebiger Länge schaffen.

Das Ganze sieht am Beispiel wie folgt aus:

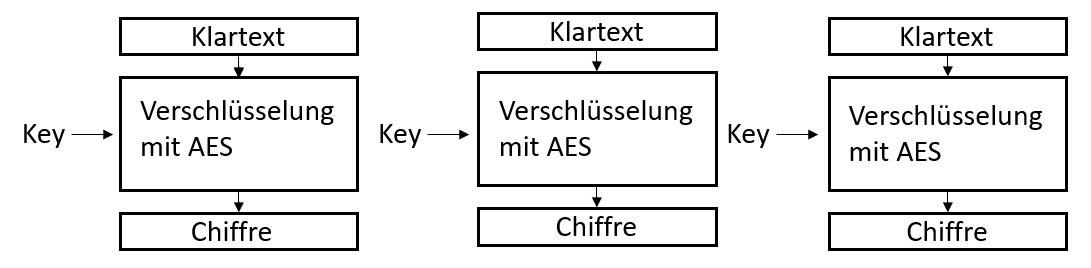


Abbildung : ECB-Modus

Dieser Modus hat eine große Problemstelle, da bei AES Klartextblöcke mit einem Schlüssel verschlüsselt immer zu dem gleichen Chiffrat-Block verschlüsselt werden. Damit würden im ungünstigsten Fall, dass in KeyManager 2 Mal derselbe Benutzername mit Passwort für verschiedene Accounts gespeichert und diese in einen zu verschlüsselnden AES-Block gelangen würden, diese Daten zu einem identischen Chiffrat-Block verschlüsselt. Dies würde einem Kryptoanalyst Hinweise auf die Verschlüsselung geben und könnte zum Dechiffrieren der Datenbank missbraucht werden.

In einem von der IJNSA aufgeführten Beispiel (IJNSA, 2013, S.18) zeigt in Abbildung 9 das Problem:

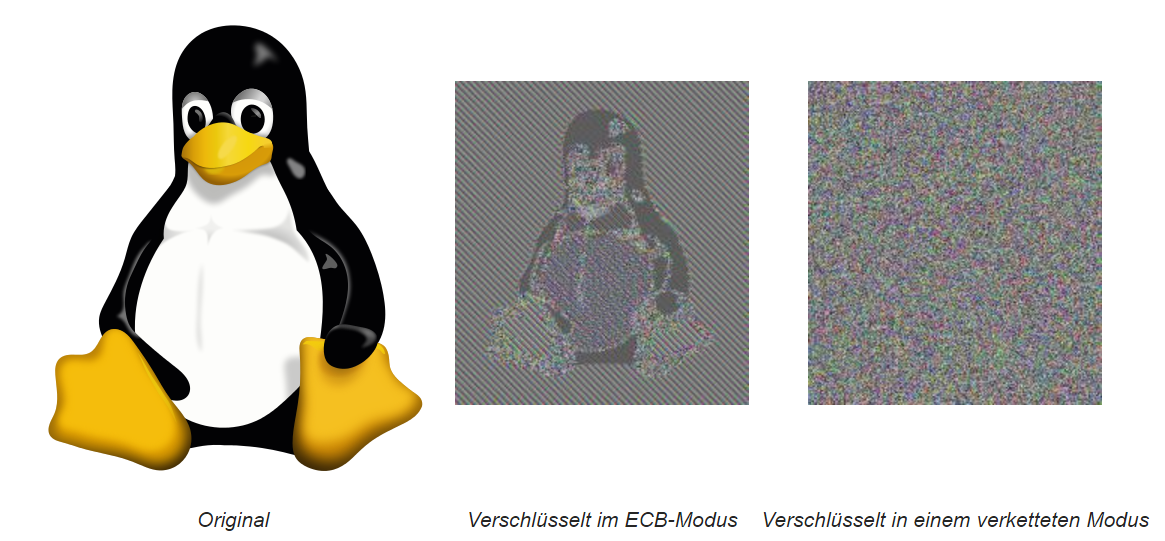


Abbildung : Vergleich von ECB mit CBC

Beim Verschlüsseln des Bildes des Pinguins können im ECB-Modus, trotz eigentlich sicherem Verschlüsselungsverfahren, noch deutlich die Konturen des Pinguins erkannt werden. Im Bild rechts daneben ist nur noch Rauschen zu erkennen; hier wurden die Blöcke mit dem CBC-Verfahren verknüpft, um mehr Entropie[[9]](#footnote-9) in die Verschlüsselung zu bekommen. Im Allgemeinen gilt, wenn man das Chiffrat betrachtet und man Muster erkennen kann, ist das unvorteilhaft. Dabei muss das Muster nicht einmal so deutlich zu erkennen sein wie im gegebenen Beispiel.

### CBC – Cipher Block Chaining

Der CBC-Modus verknüpft mit einer XOR-Verknüpfung den Klartext mit dem Chiffrat des Vorgängerblocks. Somit fließen in den Block, der verschlüsselt werden soll, alle vorher verschlüsselten Blöcke mit ein.

Der erste Block wird mit einem Initialisierungsvektor (siehe Kap. 2.3) verknüpft. Dabei handelt es sich um einen 128-Bit-Block, der mit einer XOR-Verknüpfung mit dem ersten Block der Stromchiffre verbunden wird. Bei geschickter Wahl des Initialisierungsvektors, kann das Chiffrat sogar mit immer gleichem Schlüssel und bei gleichem zu verschlüsselnden Klartext, komplett unterschiedliche Chiffrate erstellen (siehe Kap. 2.3.2). Dadurch wird die Sicherheit des Chiffrats erheblich erhöht. Für eine Kryptoanalyse sind Wiederholungen und Muster Angriffspunkte, mit dem CBC-Modus können diese vermieden werden.

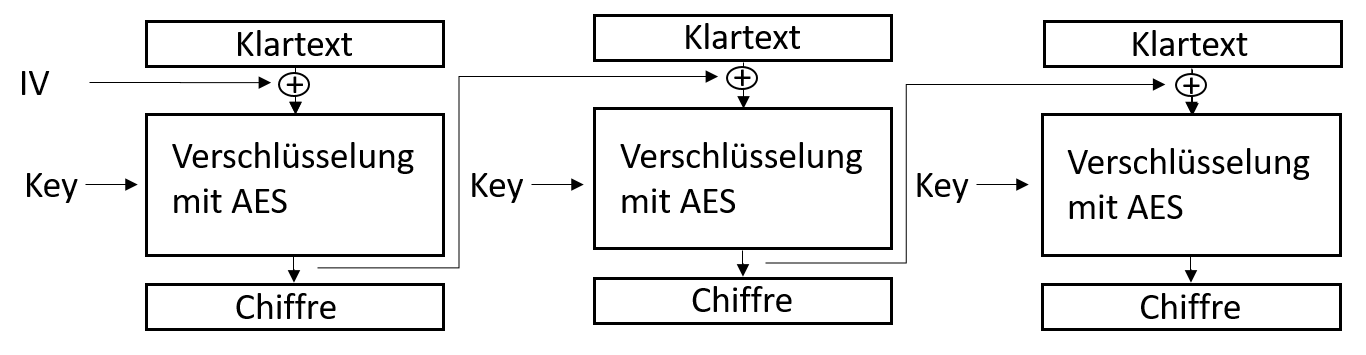


Abbildung : CBC-Modus

Durch das Verknüpfen mit den Vorgängerblöcken könnte man meinen, dass bei einem beschädigten Block alle dahinterliegenden Blöcke auch beschädigt werden. Dem ist allerdings nicht so. Bei einem defekten Block, ist nur dieser und der Nachfolgerblock beeinträchtigt und können nicht entschlüsselt werden. Da das Chiffrat des Nachfolgers noch vorhanden ist, kann ab hier weiter entschlüsselt werden.

## Initialisierungsvektor

KeyManager bietet die Möglichkeit einen Initalisierungsvektor in 3 Arten zu generieren. Einen Null-Initialisierungsvektor, einen selbst eingegebenen Initialisierungsvektor und einen automatisch generierten Initialisierungsvektor. Letztere 2 sind im Grunde dasselbe, nur dass man beim einen die Zeichen selbst eingibt und auf die Zeichen der Tastatur beschränkt ist und beim anderen der Computer die Bits des IV durch den Zufallszahlengenerator füllt. Dem PC die Erstellung des IVs zu überlassen stellt in jedem Fall die beste Wahl dar, da hier am meisten Entropie mit in die Stromchiffre einfließt.

### Null-Initialisierungsvektor

Bei einem Null-Initialisierungsvektor wird der Vektor mit 0en gefüllt, das heißt, dass die XOR-Verknüpfung mit dem Nachfolgerblock keine Veränderung verursacht. Betrachtet man die Datenbank von KeyManager sieht man, dass im ersten Block eine Signatur steht. Dieser bleibt mit einem Null-IV und gleichem Schlüssel immer gleich. Dies ist denkbar ungünstig und sollte verhindert werden.

Sollte jemand zufällig das gleiche Passwort verwenden, könnte er am ersten Block erkennen, dass dieser gleich aussieht und würde wissen, dass diese Datenbank auch mit demselben Passwort verschlüsselt wurde.

Der Null-IV bietet einzig den Vorteil, dass dieser einfach zu implementieren ist. Er sollte in der Praxis allerdings nicht verwendet werden. Als Null-Vektor könnte man auch verstehen, dass man keinen IV verwendet und die Verkettung der Stromchiffre mit dem ersten verschlüsselten Block beginnt.

### Random-Initialisierungsvektor

Durch die Implementierung eines zufälligen Initialisierungsvektors, kann bei jeder Neuverschlüsselung derselben Daten, ein komplett unterschiedliches Chiffrat erzeugt werden. In KeyManager wurde dies erreicht, indem ein Block mit zufällig gefüllten Werten, mit einem Zufallsschlüssel via AES verschlüsselt wurde. Dieser Block wurde dann mit dem ersten Datenblock mit XOR verknüpft.

Dass der IV mit einem zufälligen, dem Benutzer unbekannten, Schlüssel verschlüsselt wurde macht nichts. Dieser wird nicht entschlüsselt, sondern nur mit dem ersten Block mit XOR verknüpft. Damit dieser seinen Originalzustand wiedererhält. Der IV wird also verworfen und kann und sollte jedes Mal ein anderer sein.

## Paddingverfahren

AES ist eine Blockchiffre, die Blöcke nur in vordefinierten Größen verschlüsseln kann. Bei KeyManager wird eine Größe von 128 Bit verwendet. Beim Verschlüsseln stellt sich allerdings schnell die Frage: Was mache ich mit nicht vollen Blöcken? Diese müssen gefüllt werden!

Das Auffüllen der Blöcke nennt man Padding. Paddingverfahren sind in Standards definiert. Dabei gibt es zahlreiche Ansätze zum Füllen. In KeyManager wurde das Null-Padding, der ANSI X.923[[10]](#footnote-10) Standard und der PKCS7[[11]](#footnote-11) Standard (Kaliski, 1998) implementiert.

Die Schwierigkeit bei diesem Prozess, liegt darin, das Padding von der Datenstruktur unterscheiden zu können. Daher muss ein Trennzeichen am Ende der Datenstruktur und vor dem Padding eingefügt werden. KeyManager verwendet dazu ein Null-Byte. Der Vorteil dieser Wahl ist, dass die Datenstruktur ein langer String ist und das Null-Byte auch in C++ das terminierende Zeichen für den String ist. Bei einem genau abschließenden String am Ende eines Blockes, wird dieser allerdings weggelassen, da kein Padding nötig ist. Padding muss, falls nötig, ausschließlich im letzten Block vorgenommen werden, da alle anderen Blöcke komplett gefüllt sind.

### Null-Padding

Beim Null-Padding wird, der Block mit Bytes, die aus logischen Nullen bestehen, gefüllt. Dies ist das einfachste aller Paddingverfahren. Der Vorteil dabei ist, dass eine 0 das terminierende Zeichen eines Strings darstellt. Fügt man also von einem entschlüsselten Block, alle Bytes die sich darin befinden, mit der append()-Methode der <String>-Bibliothek ein, werden die Bytes, die eine 0 darstellen, weggelassen. Dadurch kann man sich komplizierte Algorithmen sparen, die den String nach terminierenden Zeichen durchsuchen. Eine Null am Ende eines Strings ist ausreichend. Die Methode append() erkennt dies und lässt überflüssige Nullen weg.

In Abbildung 11 ist das Null-Padding dargestellt.

0x52 0x30 0x30 0x47 **0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00**

Abbildung : Null-Padding

Es ist ein mit 4 Byte Klartext gefüllter 128-Bit-Block. Die restlichen 12 Bytes des 128-Bit-Blocks werden mit Nullen gefüllt.

### ANSI X.923

Der ANSI X.923 Standard unterscheidet sich an einer Stelle vom Null-Padding. Am Ende des Padding wird die Länge des Padding in das Byte als hexadezimale Zahl gespeichert. Das heißt, bei einem 5 Byte langen Padding wird das hinterste Byte mit 0x05 gefüllt. Diese Methode bietet den Vorteil, dass die Länge des Paddings hinterlegt wurde und dementsprechend nur in das letzte Byte geschaut werden muss und man die Länge kennt, die weggelassen werden kann.

Das bringt bei langem und häufig auftretendem Padding mehr Performance. Ist allerdings im Falle von KeyManager nicht ausschlaggebend. Da nur einmal am Ende der Datenbank gefüllt werden muss. Gespeichert wird nur sobald der Benutzer den Befehl dazu gibt. Der einzig wirkliche Vorteil für KeyManager ist, dass die Bytes nicht ausschließlich mit Nullen gefüllt werden. Er also mehr Entropie bringt, als das Null-Padding.

ANSI X.923 ist in Abbildung 12 dargestellt.

0x52 0x30 0x30 0x47 **0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x0C**

Abbildung : ANSI X.923 Padding

Da insgesamt 12 Byte angefügt wurden, wird das letzte Byte mit dem Wert 12 (0x0C) gefüllt.

### PKCS7

Der Public-Key Cryptography Standard 7 ist dem ANSI X.923 ähnlich. Hier werden allerdings alle Bytes mit der Länge des Paddings gefüllt. Wenn 4 Bytes aufgefüllt werden müssen, wird hier jedes Byte mit 0x04 gefüllt. Bei diesem Verfahren ist es zwingend notwendig ein Trennzeichen nach der Datenstruktur zu setzen, da die Padding-Bytes sonst nicht von jenen der Daten unterschieden werden können. Das Trennzeichen ist allerdings kein Bestandteil des PKCS7.

In Bezugnahme des oben aufgeführten Beispiels, sieht der zu speichernde Block dann wie in Abbildung 13 dargestellt aus.

0x52 0x30 0x30 0x47 0x00 **0x0B 0x0B 0x0B 0x0B 0x0B 0x0B 0x0B 0x0B 0x0B 0x0B 0x0B**

Abbildung : PKCS7 Padding

Nach den 4 Bytes Daten, wird das terminierende Null-Byte angehängt. Durch das Paddingverfahren werden hier dann nur noch 11 Bytes aufgefüllt. Jedes Byte wird demnach mit dem Wert 11 (0x0B) aufgefüllt.

# SHA256

SHA256 steht für Secure Hash Algorithm 256 und ist, wie der Name schon sagt, ein Hash-Verfahren. Es ist in der FIPS 180-4 (FIPS PUPS, 2015) definiert. Ein Hash-Verfahren ist eine Einwegfunktion. Das heißt, es kann in diese Funktion eine Nachricht/Text beliebiger Länge eingegeben werden und man erhält einen Hash[[12]](#footnote-12) fester Länge zurück. Dieser beträgt 256 Bit bei SHA256. Der Hash kann nicht mehr in die ursprüngliche Nachricht zurück gewandelt werden, da es sich um eine Einwegfunktion handelt. Ein Hash dient zum Abspeichern von Passwörtern oder zur Integritätsprüfung/Fehlererkennung von Daten.

## Algorithmus des SHA256

Der SHA256 arbeitet in 512-Bit-Stückchen, auch Chunks genannt. Dabei wird ein Text in beliebiger Länge auf diese Chunks aufgeteilt. Chunks, die keine 512 Bit füllen, werden durch ein vorgegebenes Paddingverfahren (siehe Kap. 2.4 und Kap. 3.2) aufgefüllt. Der Algorithmus verarbeitet nur einen Chunk gleichzeitig. Das heißt, die Anzahl der Chunks entscheidet, wie oft der Algorithmus durchlaufen werden muss.

Ähnlich dem AES-Algorithmus hat der SHA256 auch ein Array das den State beinhaltet. Dieses besteht aus 8 \* 32-Bit[[13]](#footnote-13), die zu Beginn mit vorgegebenen Werten initialisiert werden. Diese Werte können in der FIPS 180-4 (FIPS PUPS, 2015, S.15) nachgeschlagen werden.

Der in Chunks aufgeteilte Eingabetext in den Algorithmus gegeben und jeder Durchlauf wird auf das State Array aufaddiert. Der Chunk wird in einem Array w zwischengespeichert und vom Algorithmus auf 64 \* 32-Bit (2048 Bit) künstlich vergrößert.

Listing 12 – SHA256-Chunk-Initialisierung:

for (j = 0; j < 16; j++) {

SHA2\_PACK32(&chunks[i][j << 2], &w[j]);

}

for (j = 16; j < 64; j++) {

w[j] = SHA256\_SSIG1(w[j - 2]) + w[j - 7] + SHA256\_SSIG0(w[j - 15]) + w[j - 16];

}

SHA2\_PACK32 kopiert die Werte vom linken Argument (&chunks[i][j << 2]) in das rechte Argument (&w[j]), das ihm übergeben wird. Die Variable chunks ist ein Vektor aus Strings, der die einzelnen Chunks beinhaltet. In der zweiten For-Schleife aus Listing 12, wird das Array w mit den Werten, die ihm vorher übergeben wurden, künstlich erweitert. SHA256\_SSIG1 ist in Listing 13 dargestellt.

Listing 13 – SHA256\_SSIG1:

SHA256\_SSIG1(x) (SHA2\_ROTR(x, 17) ^ SHA2\_ROTR(x, 19) ^ SHA2\_SHFR(x, 10))

Das in Listing 13 übergebene Argument X von SHA256\_SSIG1 wird an die Funktionen SHA2\_ROTR und SHA2\_SHFR weitergegeben.

Listing 14 – SHA2\_ROTR und SHA2\_SHFR:

SHA2\_ROTR(x, n) ((x >> n) | (x << ((sizeof(x) << 3) - n)))  
SHA2\_SHFR(x, n) (x >> n)

SHA2\_ROTR im Listing 14 rotiert die Bits des Wertes, der übergeben wird, im ersten Aufruf um 17 und im zweiten um 19 Stellen nach rechts. Hier werden die Bits, die rechts herausgeschoben werden, links wieder hineingeschoben.

SHA2\_SHFR (ebenfalls im Listing 14) dagegen schiebt (shiftet) die Bits hinaus und füllt die dadurch leeren Stellen mit 0en auf. Der <<-Operator sowie der >>-Operator stellen einen logischen Bit-Shift in C++ dar. Die Rückgabewerte von SHA2\_ROTR und SHA2\_SHFR werden in Listing 13 dann ver-Odert. Der Rückgabewert von SHA\_SSIG1 ist demnach immer ein 32-Bit-Wert.

In SHA\_SSIG0 aus Listing 12, passiert das Gleiche, wie in SHA256\_SSIG1, nur mit anderen Werten für die Shift-Operationen. Dies ist in Listing 15 ersichtlich.

Listing 15 – SHA256\_SSIG0:

SHA256\_SSIG0(x) (SHA2\_ROTR(x, 7) ^ SHA2\_ROTR(x, 18) ^ SHA2\_SHFR(x, 3))

Als Nächstes wird der aktuelle State aus m\_h in das Integer-Array wv zwischengespeichert. Das ist in Listing 16 zu sehen.

Listing 16 – State wird zwischengespeichert:

for (j = 0; j < 8; j++) {

wv[j] = m\_h[j];

}

Nun folgt die Hauptschleife. In ihr werden die künstlich aufgeblasenen Werte auf die Initialwerte in einer komplexen Form aufaddiert. Diese Form wird in Abbildung 14 dargestellt.

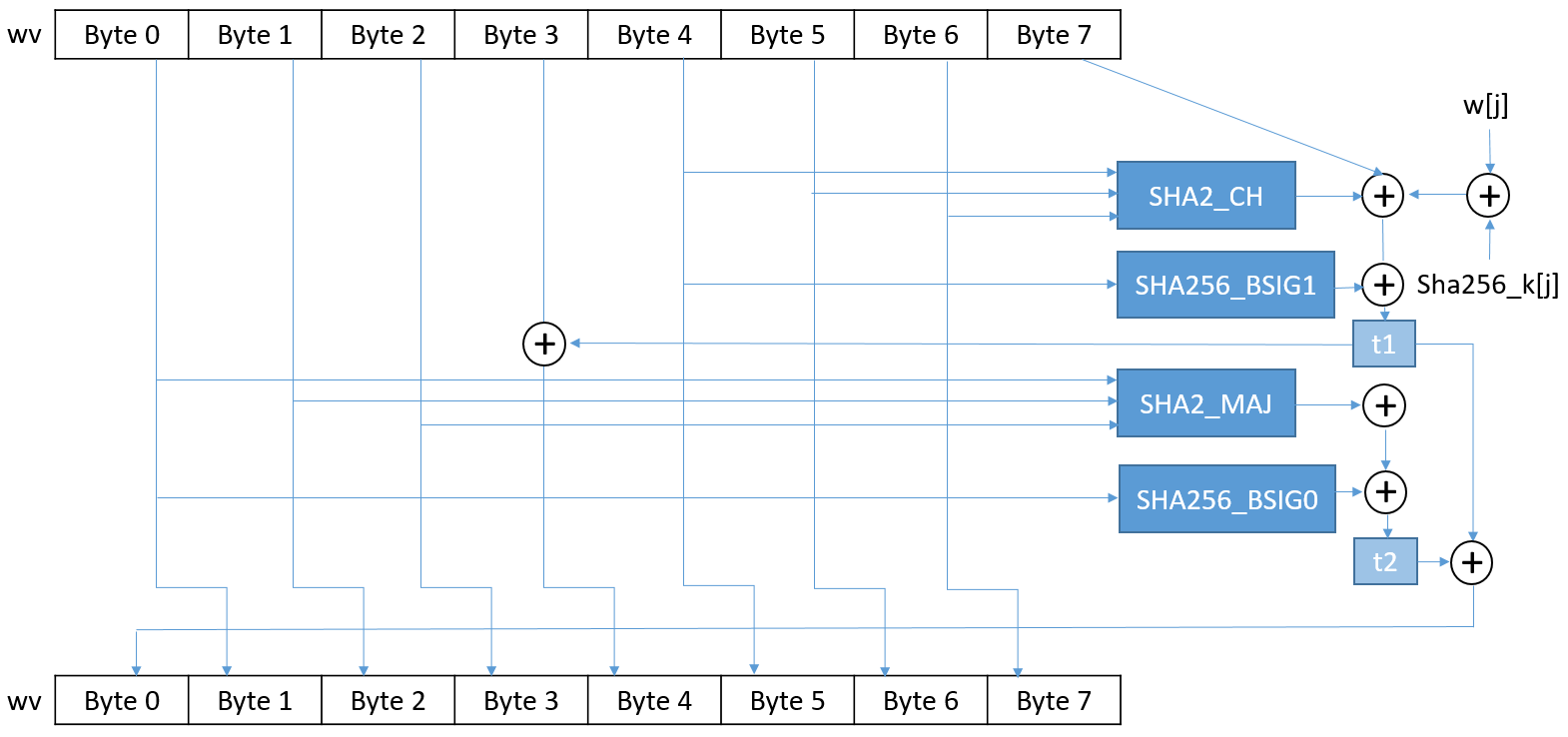


Abbildung : SHA256 Algorithmus

Der zu Abbildung 14 passende C++-Code wird in Listing 17 dargestellt.

Listing 17 – SHA256-Hauptschleife:

for (j = 0; j < 64; j++) {

t1 = wv[7] + SHA256\_BSIG1(wv[4]) + SHA2\_CH(wv[4],wv[5],wv[6]) + sha256\_k[j] + w[j];

t2 = SHA256\_BSIG0(wv[0]) + SHA2\_MAJ(wv[0], wv[1], wv[2]);

wv[7] = wv[6];

wv[6] = wv[5];

wv[5] = wv[4];

wv[4] = wv[3] + t1;

wv[3] = wv[2];

wv[2] = wv[1];

wv[1] = wv[0];

wv[0] = t1 + t2;

}

Die Definitionen der in Listing 17 verwendeten Funktionen SHA256\_BSIG1, SHA2\_CH und SHA2\_MAJ werden in Listing 18 dargestellt.

Listing 18 – SHA256\_BSIG1, SHA2\_CH und SHA2\_MAJ:

SHA256\_BSIG1(x) (SHA2\_ROTR(x, 6) ^ SHA2\_ROTR(x, 11) ^ SHA2\_ROTR(x, 25))   
SHA2\_CH(x, y, z) ((x & y) ^ (~x & z))

SHA2\_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))

Der &-Operator entspricht einer logischen Und-Verknüpfung. Der ~-Operator entspricht einer Negation, also dem Umdrehen aller Bits.

Sha256\_k aus Listing 17 ist ein vom Algorithmus vordefiniertes Array. Die Werte, mit denen es gefüllt ist, sind in der FIPS 180-4 (FIPS PUPS, 2015, S.11) zu finden.

Zum Abschluss wird das Zwischenergebnis, aus der Hauptschleife aus Listing 17, auf den State m\_h aufaddiert. Dies ist in Listing 19 zu sehen.

Listing 19 – SHA256-Zwischenergebnis auf State aufaddieren:

for (j = 0; j < 8; j++) {

m\_h[j] += wv[j];

}

Ab hier ist die Schleife des SHA256 beendet. Die Schleife wird so lange wiederholt, bis keine Chunks mehr übrig sind. Der komplette Quellcode wird in Listing 20 abgebildet.

Listing 20 – SHA-256-Algorithmus:

void sha256::hash() {

unsigned int w[64];

unsigned int wv[8];

unsigned int t1, t2;

int i;

int j;

for (i = 0; i < (int)chunks.size(); i++) {

for (j = 0; j < 16; j++) {

SHA2\_PACK32(&chunks[i][j << 2], &w[j]);

}

for (j = 16; j < 64; j++) {

w[j] = SHA256\_SSIG1(w[j - 2]) + w[j - 7] + SHA256\_SSIG0(w[j - 15]) +   
 w[j - 16];

}

for (j = 0; j < 8; j++) {

wv[j] = m\_h[j];

}

for (j = 0; j < 64; j++) {

t1 = wv[7] + SHA256\_BSIG1(wv[4]) + SHA2\_CH(wv[4], wv[5], wv[6]) +   
 sha256\_k[j] + w[j];

t2 = SHA256\_BSIG0(wv[0]) + SHA2\_MAJ(wv[0], wv[1], wv[2]);

wv[7] = wv[6];

wv[6] = wv[5];

wv[5] = wv[4];

wv[4] = wv[3] + t1;

wv[3] = wv[2];

wv[2] = wv[1];

wv[1] = wv[0];

wv[0] = t1 + t2;

}

for (j = 0; j < 8; j++) {

m\_h[j] += wv[j];

}

}

}

## SHA256 Paddingverfahren

Das Paddingverfahren (siehe Kap. 2.4) des SHA256 ist vorgegeben. Hier gibt es also keine Wahlmöglichkeit, wie man die nicht vollen Blöcke füllt. Gefüllt werden müssen beim SHA256 die Chunks. Dem nicht vollen Chunk, wird dabei hinter dem letzten Bit, das mit Daten gefüllt wird, eine 1 angehängt. Die Länge der Daten im Chunk ist l. Darauf folgen k Bits. k ist dabei die nicht negative Lösung der Formel l+1+k≡448 mod 512. Damit sind 448 Bits des 512-Bit-Blocks gefüllt. In den letzten 64 Bit wird die Länge l in Binärschreibweise angehängt.

Betrachtet man das Beispiel aus Kapitel 2.4.1 stellt sich das Schema aus Abbildung 15 dar.

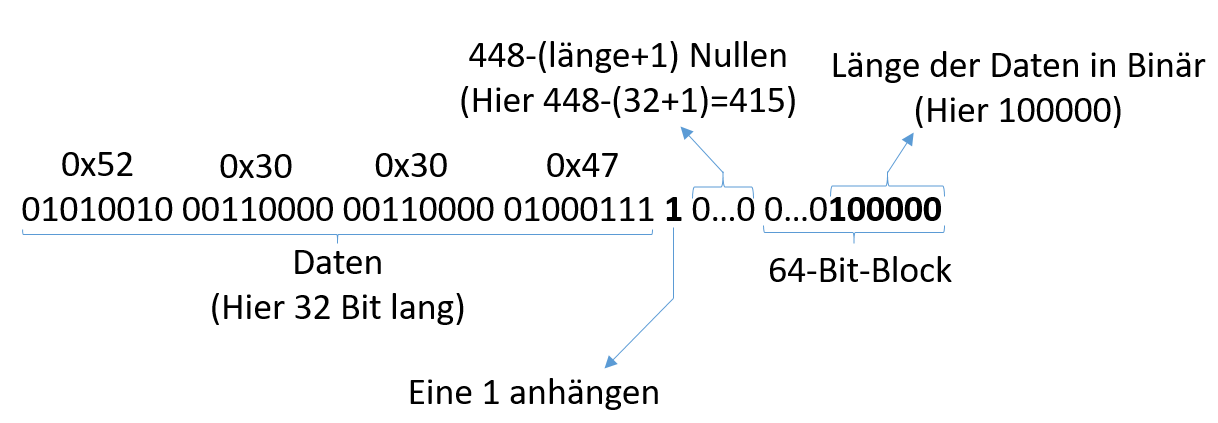


Abbildung : SHA256 Paddingverfahren

Der in Abbildung 15 dargestellte Chunk beinhaltet 4 Chars Daten, die je 8 Bit groß sind. 4 \* 8 Bit entspricht einer Datenlänge l von 32 Bit. Darauf folgt eine 1 und 448-(l+1) an Nullen. 448-(32+1) = 415. Es werden also 415 Nullen angehängt. Am Ende kommt ein 64 Bit großer Block, in dem in Binärdarstellung, die Länge des mit Daten gefüllten Bereichs steht. Die Daten nehmen 32 Bit ein, daher wird 100000 in die hintersten Bits geschrieben. Durch das vorgegebene Paddingverfahren wird gewährleistet, dass bei gleicher Eingabe auch identische Hashes berechnet werden.

Implementiert wurde dieses Verfahren folgendermaßen. Erst wird die Länge der Daten des letzten Chunks ermittelt und alle Chunks werden mit Nullen aufgefüllt, ab der Position, an der keine Daten mehr stehen plus 1. Dies ist in Listing 21 zu abgebildet.

Listing 21 – Chunks für Padding vorbereiten:

int geslaenge\_bits = gesamtlaenge \* 8;

for (int i = gesamtlaenge+1; i < 64; i++) {

chunks[chunks.size() - 1][i] = 0;

}

Danach wird im letzten Chunk das 1er Bit dazugehängt. Der Aufruf der Funktion SHA2\_UNPACK32 füllt die Bits im hinteren 64-Bit-Block mit der Länge des Datenblocks. In Listing 22 ist der Code dafür zu sehen.

Listing 22 – 1er Bit einfügen und Länge hinten anfügen:

chunks[chunks.size() - 1][gesamtlaenge % 64] = 0x80;

SHA2\_UNPACK32(geslaenge\_bits, chunks[chunks.size() - 1] + 64 - 4);

Das Paddingverfahren als C++ Code ist im Ganzen noch mals in Listing 23 dargestellt.

Listing 23 – SHA256 Paddingverfahren:

void sha256::padding() {

int geslaenge\_bits = gesamtlaenge \* 8;

for (int i = gesamtlaenge+1; i < 64; i++) {

chunks[chunks.size() - 1][i] = 0;

}

chunks[chunks.size() - 1][gesamtlaenge % 64] = 0x80;

SHA2\_UNPACK32(geslaenge\_bits, chunks[chunks.size() - 1] + 64 - 4);

}

## Passwort als Hash

Da eine gleiche Eingabe immer zum selben Hash führt, können damit Dinge, wie Passwörter oder Text, verglichen werden ohne das Original als Klartext zu besitzen. Dies wird beim Speichern von Passwörtern in Datenbanken praktiziert. Dabei wird statt dem Passwort dessen Hash gespeichert, damit das Passwort nicht im Klartext abgespeichert werden muss. In KeyManager wird beim Erstellen einer Datenbank der Hash des Masterpassworts zum Verschlüsseln der Datenbank verwendet.

Der Hash steht während der Benutzung der Datenbank im AES-Objekt und könnte mit einem Debugger[[14]](#footnote-14) oder einen Memory Dump[[15]](#footnote-15) einfach ausgelesen werden. Würde das Passwort im Klartext verwendet werden, könnte dies direkt als String aus dem Speicher ausgelesen werden. Durch die Verwendung des SHA256 ist dieses Problem beseitigt. Damit ist der Hash im Speicher nicht mehr so einfach erkennbar wie ein normaler String, da er keine sinnvollen ASCII- oder UTF-8 Zeichen darstellt.

## Hash zur Integrationssicherung

Zur Erkennung ob die Datenbank korrekt entschlüsselt wurde, wird sie gehasht und der Hash am Ende der Datei angehängt. Beim Einloggen in die Datenbank prüft KeyManager, ob der angehängte Hash, mit dem Hash des Klartextes der Datenbank die entschlüsselt wurde, übereinstimmt. Somit können Fehler oder mutwillige Manipulationen erkannt werden. Nur ein fehlerhaftes Bit führt zu einem völlig anderen Hash. Bei nicht übereinstimmen des nach der Entschlüsselung berechneten Hashs mit dem in der Datenbank hinterlegten Hashs bricht KeyManager mit einer Fehlermeldung ab.

# Aufbau der Datenbank

In KeyManager wird, um den Overhead so gering wie möglich zu halten, der Inhalt als langer String abgespeichert(somit werden keine Bibliotheken wie SQL[[16]](#footnote-16) oder ähnliches benötigt). Dabei werden die Einträge mit einem Trennzeichen abgetrennt. Ein Eintrag besteht aus dem Titel, dem Benutzernamen und dem Passwort, welche selbst nochmals mit einem anderen Trennzeichen abgetrennt sind.

Einträge haben als Trennsymbol das New-Line-Symbol ‚/n‘, welches dem Wert 10 (0x0A) in der ASCII-Tabelle entspricht. Als Trennzeichen zwischen Titel, Benutzername und Passwort, wird das Symbol mit dem Wert 3 (0x03) der ASCII-Tabelle verwendet. Dies ist das ASCII-Symbol mit dem Herzen. Beide Werte wurden so gewählt, dass die Symbole nicht direkt mit der Tastatur eingegeben werden können, um zu verhindern, dass KeyManager beim Interpretieren der Datenbank (des Strings) die Benutzereingaben falsch versteht.

Hier eine schematische Darstellung der Datenbankstruktur ist in Abbildung 16 dargestellt.

MeinEMail 0x03 Username 0x03 SicheresPasswort 0x0A MeineHP 0x03 Username2 0x03 AnderesPasswort 0x0A

Abbildung : Datenbankstruktur

Die Leerzeichen zwischen den Hexadezimalwerten wurden nur zur Übersichtlichkeit eingefügt und werden so nicht von KeyManager abgespeichert. Um die Anzahl an gespeicherten Einträgen ermitteln zu können, muss man nur die 0x0A Symbole zählen. Diese stehen am Ende jeden Eintrages und entsprechen exakt der Anzahl an Einträgen in der Datenbank.

Beim Öffnen der Datenbank zerlegt KeyManager den String in seine Teile und generiert daraus Objekte. Dies bietet den Vorteil, dass die Einträge flexibler und performanter zu bearbeiten sind. So kann man ganze Einträge an beliebiger Stelle löschen und Titel, Benutzername oder Passwort einzeln ändern. Beim Abspeichern werden diese Teile wieder zu einem langen String aneinander gekettet und verschlüsselt abgespeichert.

# Aufbau der Datei

Die Datei, in der KeyManager eine Datenbank abspeichert, ist in viele verschiedene Segmente aufgeteilt. Es kommt erst der Header, dann der optionale IV, dann eine Signatur, danach die Datenbank und am Ende wird ein Hash angehängt.

## Der Header

Der Header ist 16 Byte lang und besteht aus der Magic Number „KeyManager“ als normaler String, gefolgt von der Versionsnummer und den Flags für den Algorithmus. Die Flags zeigen an, welcher Stromchiffre-Modus, welches Paddingverfahren und welche Art des Initialisierungsvektors benutzt wurde.

Die ersten 10 Byte sind für die Symbole der Magic Number „KeyManager“. Jedes Zeichen benötigt 1 Byte. Dieser wird beim Öffnen der Datenbank geprüft, um sicherzustellen, dass keine falsche Datei geöffnet wird. Sollte eine Datei ohne die korrekte Magic Number geöffnet werden, wird eine Fehlermeldung ausgegeben und abgebrochen.

Darauf folgen 4 Byte für die Version, die getrennt durch ein Leerzeichen auch als String gespeichert werden. Die aktuelle Version 1.0 notiert den String „1.0“ dort. Durch diese Information soll beim Öffnen direkt erkennbar werden, ob die Version mit der Datenbank kompatibel ist.

Darauf folgt 1 Byte, welches mit mehreren Flags besetzt ist. Die ersten 2 Bits sind für den Algorithmus, der zum Verschlüsseln verwendet wird, reserviert. Diese tragen immer eine 1 im rechten Bit und das bedeutet, dass der AES-Algorithmus verwendet wurde. Danach kommen 3 Bits, die angeben, welcher Stromchiffre-Modus gewählt wurde. Wenn alle 3 Bits eine 0 sind wurde der ECB-Modus zur Verschlüsselung verwendet, wenn eine 1 im rechten Bit steht, wurde der CBC-Modus verwendet. Dahinter kommen weitere 3 Bits, die die Wahl des Paddingverfahrens angeben. Zero-Padding kann man daran erkennen, dass alle Bits 0 betragen. Wenn eine 1 im rechten Bit ist, wurde das ANSI X.923-Verfahren verwendet. Wenn eine 1 in der Mitte steht, wurde das PKCS7-Verfahren verwendet.

Das letzte Byte im Header ist für die Wahl des Initialisierungsvektors bestimmt. Wenn alle Bits 0 sind, wurde der Null-Initialisierungsvektor verwendet. Bei einer 1 im rechten Bit wurde ein automatisch generierter IV verwendet. Und bei einer 1 im mittleren Bit wurde ein selbst eingegebener IV verwendet.

Der gesamte Header ist in Abbildung 17 zu sehen.

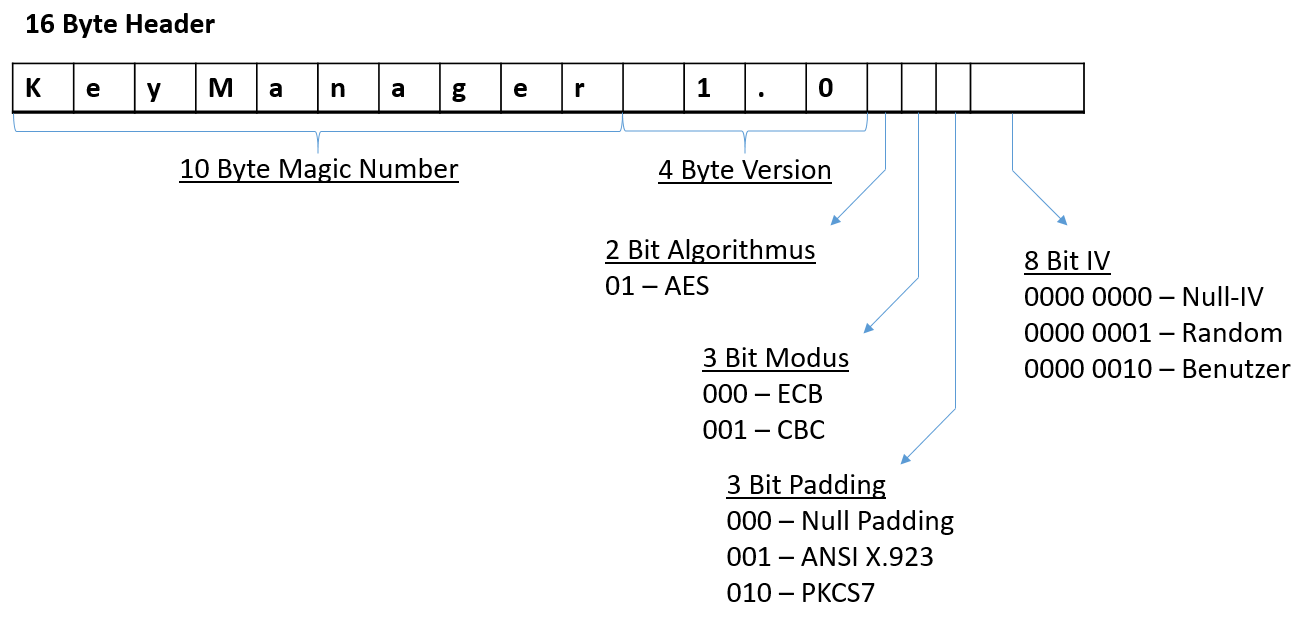


Abbildung : Header KeyManager-Datei

In diesem Header sind alle Informationen gespeichert, die KeyManager braucht, um die Datei korrekt bearbeiten zu können. Die einzelnen Bits werden beim Öffnen mit if-else-Abfragen bearbeitet, damit KeyManager in die richtige Schleife für den passenden Algorithmus springen kann. Würde ein falscher Algorithmus die Datei bearbeiten, würde ein für KeyManager sinnloser Datenbrei beim Entschlüsseln rauskommen. Daher sind alle im Header gespeicherten Daten essentiell für die Bearbeitung der Datenbank.

## Die Passwortsignatur

Um feststellen zu können, ob eine Datenbank mit dem richtigen Passwort entschlüsselt wurde, wird eine Signatur verwendet. Mit dem AES-Algorithmus können beliebige Daten mit beliebigen Schlüssel entschlüsselt werden. Dabei ist dem Algorithmus egal, ob es sich um die richtigen Daten mit dem korrekten Passwort handelt. Bei einem falschen Passwort kommen beim Entschlüsseln allerdings völlig sinnlose Daten raus.

Um festzustellen, ob nach dem Entschlüsselungsvorgang sinnlose oder sinnvolle Daten entstanden sind, wurde ein immer gleicher Block als Passwortsignatur eingefügt. Sollte also nach einem Entschlüsselungsversuch dieser Block nicht die korrekte Signatur enthalten, ist ein falsches Passwort zur Entschlüsselung eingegeben worden. Im Folgenden kann eine entsprechende Fehlermeldung ausgegeben und der Entschlüsselungsvorgangabgebrochen werden.

Da der AES in 16-Byte-Blöcken arbeitet, wurde in diesen Signatur-Block der String „Passwort korrekt“ gespeichert. Dieser besteht aus genau 16 Chars mit je 1 Byte Große. Es wird also Char für Char überprüft, ob das richtige Ergebnis erzielt wird. Wenn alles übereinstimmt, wird mit dem Entschlüsseln des nächsten Blocks fortgefahren.

Darstellung der Passwortsignatur in einem AES-Block ist in Abbildung 18 zu sehen.

|  |  |  |  |
| --- | --- | --- | --- |
| P | a | s | s |
| w | o | r | t |
|  | k | o | r |
| r | e | k | t |

Abbildung : Passwortsignatur

## Dateistruktur

Die Struktur der Datei hängt von den verwendeten Algorithmen ab. Der ECB-Modus hat zum Beispiel keinen Initialisierungsvektor. Dies erkennt KeyManager am Header und kann dementsprechend die Datei bearbeiten. Dabei ist allerdings die Reihenfolge der Teile, aus denen sich die Datei zusammensetzt, stets die Gleiche. Es kommt erst der 16 Byte lange Header, dann ein optionaler 16 Byte langer Initialisierungsvektor, danach die 16 Byte lange Signatur, dahinter kommen beliebig viele 16 Byte große Blöcke mit dem Inhalt der Datenbank und zum Schluss wird ein Hash mit 32 Byte angehängt. Der Hash wird über den Klartext des Datenbankinhaltes erstellt.

Der Aufbau ist in Abbildung 19 dargestellt.

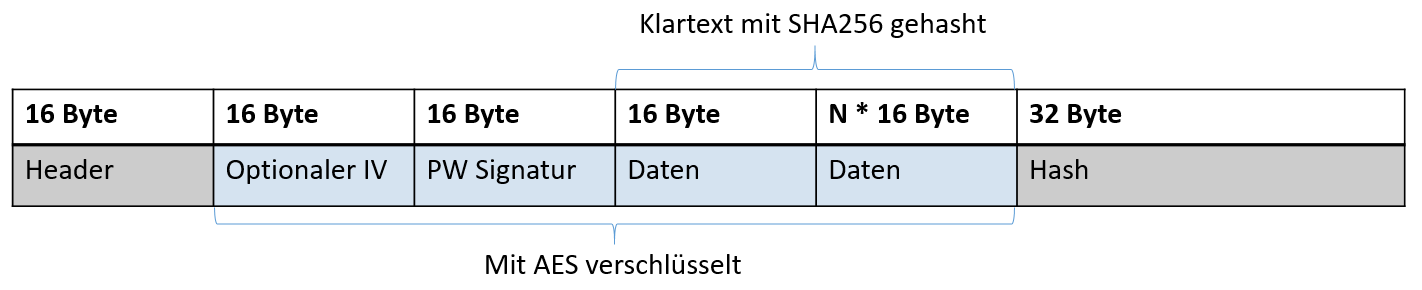


Abbildung : Struktur einer KeyManager-Datei

# Fazit

Verschlüsselung ist ein sehr umfangreiches Feld. Die Einarbeitung ins Thema hat die größte Zeit in Anspruch genommen, allerdings konnte durch die Implementierung der Algorithmen ein sehr umfassender Einblick in die Thematik erreicht werden. Die Algorithmen haben sich allesamt einfach Programmieren lassen. KeyManager erledigt seine Aufgabe hervorragend und läuft sehr stabil. Sehr interessant hat sich auch das Erstellen der Datenbank gestaltet. Da sie in verschiedenen Modi verschlüsselt werden kann, musste dafür ein Protokoll in den Header implementiert werden. Durch das Auslesen des Headers, kann KeyManager problemlos feststellen, mit welchen Einstellungen die Datenbank erstellt wurde. Durch einen geschickten Aufbau des Quellcodes konnte er höchstflexibel gestaltet werden. Es werden Big und Little Endian ausgeglichen und KeyManager kann unter Linux sowie Windows problemlos kompiliert und verwendet werden. Man kann ohne Probleme auf einem Windows-Rechner eine Datenbank für den Raspberry Pi generieren und sie auf dem Raspberry auslesen. Umgedreht funktioniert das Ganze natürlich auch, trotz anderem Betriebssystem und Prozessorarchitektur. Diese Flexibilität gibt dem Projekt ein gewisses Etwas und macht KeyManager zu einem sehr nützlichen Tool, mit dem man sicher Passwörter hinterlegen und verwalten kann.

# Abbildungsverzeichnis

[Abbildung 1: Verschlüsselung mit AES 3](#_Toc466205979)

[Abbildung 2: Entschlüsselung mit AES 4](#_Toc466205980)

[Abbildung 3: Das Union „Wort“ 5](#_Toc466205981)

[Abbildung 4: ShiftRows 7](#_Toc466205982)

[Abbildung 5: MixColumns 8](#_Toc466205983)

[Abbildung 6: InvShiftRows 9](#_Toc466205984)

[Abbildung 7: invMixColumns 10](#_Toc466205985)

[Abbildung 8: ECB-Modus 12](#_Toc466205986)

[Abbildung 9: Vergleich von ECB mit CBC 13](#_Toc466205987)

[Abbildung 10: CBC-Modus 14](#_Toc466205988)

[Abbildung 11: Null-Padding 15](#_Toc466205989)

[Abbildung 12: ANSI X.923 Padding 16](#_Toc466205990)

[Abbildung 13: PKCS7-Padding 16](#_Toc466205991)

[Abbildung 14: SHA256-Algorithmus 18](#_Toc466205992)

[Abbildung 15: SHA256-Paddingverfahren 20](#_Toc466205993)

[Abbildung 16: Datenbankstruktur 21](#_Toc466205994)

[Abbildung 17: Header-KeyManager-Datei 22](#_Toc466205995)

[Abbildung 18: Passwortsignatur 23](#_Toc466205996)

[Abbildung 19: Struktur einer KeyManager-Datei 24](#_Toc466205997)

# Listingsverzeichnis

Listing 1: Die Hauptschleife des AES-Algorithmus 4  
Listing 2: AddRoundKey 5  
Listing 3: Union Wort 6   
Listing 4: SubBytes 7   
Listing 5: ShiftRows 7  
Listing 6: mixColumns 8  
Listing 7: Die Entschlüsselungshauptschleife des AES-Algorithmus 9  
Listing 8: invShiftRows 9  
Listing 9: invSubBytes 10  
Listing 10: invMixColumns 11  
Listing 11: expandKey 12  
Listing 12: SHA256-Chunk-Initialisierung 17  
Listing 13: SHA256\_SSIG1 17  
Listing 14: SHA2\_ROTR und SHA2\_SHFR 17  
Listing 15: SHA256\_SSIG0 17  
Listing 16: State wird zwischengespeichert 17  
Listing 17: SHA256 Hauptschleife 18   
Listing 18: SHA256\_BSIG1, SHA2\_CH und SHA2\_MAJ 18  
Listing 19: SHA256 Zwischenergebnis auf State aufaddieren 19   
Listing 20: SHA-256 Algorithmus 19  
Listing 21: Chunks für Padding vorbereiten 20  
Listing 22: 1er Bit einfügen und Länge hinten anfügen 20   
Listing 23: SHA256 Paddingverfahren 20

# Quellenverzeichnis

[1] Müller, O. (29.09.2011). *Heise*. „Cryptography Engineering, Teil 1: Zur Theorie des Advanced Encryption Standard“. <https://www.heise.de/developer/artikel/Cryptography-Engineering-Teil-1-Zur-Theorie-des-Advanced-Encryption-Standard-1350362.html>

[2] BSI. (kein Datum). *BSI - Bundesministerium für Sicherheit in derInformationstechnik*. <https://www.bsi-fuer-buerger.de/BSIFB/DE/Empfehlungen/Verschluesselung/Datenverschluesselung/Grundlagen/Funktionsweise/funktionsweise_node.html>

[3] FIPS PUPS, (26.11.2001). *Federal Information Processing Standards Publication 197. „Announcing the ADVANCED ENCRYPTION STANDARD (AES)“.* <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

[4] Hunag, K., Chiu J., Shen, S. (Januar 2013). *IJNSA. International Journal of Network Security & Its Applications. “A NOVEL STRUCTURE WITH DYNAMIC OPERATION MODE FOR SYMMETRIC-KEY BLOCK CIPHERS”.* <http://airccse.org/journal/nsa/0113nsa02.pdf>

[5] Kaliski, B. (März 1998). RFC. *Request for Comments 2315*. “PKCS #7: Cryptographic Message Syntax Version 1.5”. <https://tools.ietf.org/html/rfc2315>

[6] FIPS PUPS, (August 2015). *Federal Information Processing Standards Publication 180-4* <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>

1. Ein Textbasiertes Programm in der Command-Shell, ohne graphisches Benutzerinterface. [↑](#footnote-ref-1)
2. Electronic Code Book [↑](#footnote-ref-2)
3. Cipher Block Chaining [↑](#footnote-ref-3)
4. Advanced Encryption Standard [↑](#footnote-ref-4)
5. National Institute of Standards and Technology [↑](#footnote-ref-5)
6. Bundesamt für Sicherheit in der Informationstechnik [↑](#footnote-ref-6)
7. Physikalisch-Technische Bundesanstalt [↑](#footnote-ref-7)
8. Rcon steht für Round Constant Word Array. [↑](#footnote-ref-8)
9. Entropie bedeutet durcheinander; je mehr durcheinander man in ein Chiffrat bekommt,  
   desto weniger Rückschlüsse lassen sich auf den Klartext ziehen. [↑](#footnote-ref-9)
10. Der Standard X.923 des American National Standard Institut [↑](#footnote-ref-10)
11. Public-Key Cryptography Standards 7 [↑](#footnote-ref-11)
12. Aneinander gekettete Zeichen ohne Sinn; Hash bedeutet auch Rauschen. [↑](#footnote-ref-12)
13. Ein 32-Bit Wert entspricht einem Integer in C/C++. [↑](#footnote-ref-13)
14. Programm zur Fehleridentifikation anderer Programme. [↑](#footnote-ref-14)
15. Inhalt des Arbeitsspeichers als Datei auf der Festplatte. [↑](#footnote-ref-15)
16. Structured Query Language [↑](#footnote-ref-16)