Programmieraufgabe zur Vorlesung Kryptologie

Michael Tempelmeier
michael.tempelmeier@tum.de

SS 2017

Technische Universität München
Fakultät für Elektro- und Informationstechnik

Lehrstuhl für Sicherheit
in der Informationstechnik

Inhaltsverzeichnis


Vorwort

Das Ziel der Programmieraufgabe ist, Ihnen ein besseres Verständnis der Funktionsweise und Verwendung aktueller Blockchiffren zu vermitteln. Die Anforderungen an die Implementierung beschränken sich dabei auf eine korrekte Funktion. Gegenmaßnahmen gegen (Seitenkanal-)Angriffe sind dann Thema der Vorlesung Sichere Implementierung kryptographischer Verfahren.

Rahmenbedingungen

Die Bearbeitung der Programmieraufgabe ist freiwillig. Für das Bestehen der Aufgabe sind die Aufgabenstellungen in den Abschnitten 4.1, 4.2 und 4.3 erfolgreich zu bearbeiten. Die Aufgaben gelten als erfolgreich bearbeitet, wenn der entsprechende Quellcode vom Studenten eingereicht wurde und das eingereichte Programm in Moodle korrekt ausgeführt wird.
Eine Bearbeitung der Aufgabe 4.1 ist in Teams bis zu drei Mitgliedern möglich. Die Lösung muss jedoch von **jedem Studenten eigenständig abgegeben** werden. Sofern Sie sich für Teamwork entscheiden, müssen Sie bei jeder Funktion eindeutig kennzeichnen, wer welchen Beitrag geleistet hat. Eine pauschale Aussage

> *Alle haben den gleichen Anteil geleistet!*

oder

> *An dieser Funktion haben Studen1, Student2 und Student3 gearbeitet!*

ist **nicht ausreichend** und führen zum Nichtbestehen!
Bei erfolgreicher Bearbeitung wird ein Notenbonus von **0,3** auf die Modulnote gewährt, falls die Modulprüfung mit 4,0 oder besser bewertet wurde. Die Abgabe erfolgt über Moodle. Letzte Möglichkeit zur Abgabe ist Donnerstag, der 27.7.2017, 13:00 Uhr, 15:00 Uhr.

Zielplattform

Zielplattform für die Implementierung ist die von Moodle bereitgestellte Sandboxumgebung. Auch wenn diese Umgebung prinzipiell ausreichend ist, wird dringend empfohlen den Code entweder auf dem eigenen Rechner oder im Eikon-Pool zu entwickeln und zu debuggen. Zum Testen, ob der zu hause entwickelte Code syntaxkonform zur Sandboxumgebung ist, steht ein “Precheck” zur Verfügung. Diesen können Sie ohne Nachteile der Bewertung beliebig oft ausführen. Beachten Sie, dass es sich hierbei um sehr einfache Testvektoren handelt. Vor der Abgabe steht ihnen zudem eine begrenzte Anzahl an “Checks” zur Verfügung.

3.1 Organisation des Arbeitsverzeichnisses

Nach dem Entpacken der Angabe in ein leeres Verzeichnis, finden Sie darin ein Verzeichnis `code`, das die vorbereiteten Code-Dateien und die Dokumentation enthält. Zusätzlich enthält das Archiv diese Angabe und ein Verzeichnis `testvektors` zum Testen der einzelnen Funktionen. `testvectors.txt` enthält die einzelnen Testvektoren, wie sie vom Programm aus `test.c` erzeugt wurden. Zum besseren Verständnis liegt auch der Quelltext der Testumgebung bei. Da Moodle leider keine Aufteilung in Header- und Implementierungsdateien unterstützt, müssen wir Ihnen leider raten bereits bei der Implementierung, alles in der Datei `abgabe.c` zu implementieren! Dies erleichtert Ihnen die spätere Abgabe ungemein!

3.2 Build

Die wichtigsten Befehle sind:

`gcc main.c -o kryptoS17` Übersetzt Ihr Programm. Damit wird im Verzeichnis `code` die Datei `kryptoS17` erzeugt.

`./kryptoS17` Führt Ihr Programm aus. Mit der vorgegebenen main()-Methode erhalten Sie direkt Rückmeldung, ob Ihr Programm funktioniert.

Aufgabenstellung

Nachstehend werden die einzelnen Aufgabenteile näher beschrieben. Eine Dokumentation der Skeleton-Funktionen liegt in Form von HTML-Dateien bei.

4.1 Teil 1: Die BKSQ-Blockchiffre

Der erste (und umfangreichste) Aufgabenteil besteht in der Implementierung der Blockchiffre *BKSQ* mit zugehörigem Betriebsmodus. BKSQ ist als Vorläufer des Rijndael-Algorithmus eng mit dem in der Vorlesung vorgestellten AES verwandt. Für die Implementierung ist es deswegen hilfreich, wenn Sie den entsprechenden Stoff vorher wiederholen und die einführenden Fragen beantworten.

1. Lesen Sie die Beschreibung von BKSQ unter `http://jda.noekeon.org/JDA_VRI_BKSQ_2000.pdf` Die Implementierung soll auf Basis dieser Spezifikation erfolgen. In welcher Relation stehen die Funktionen θ,γ, π und σ zu den Teilschritten des AES?

2. Im genannten Paper ist die Darstellung von $\mathrm{GF}(2^8)$ nicht näher spezifiziert. Der Einfachheit halber wählen wir die Rijndael-Darstellung, wie Sie im AES benutzt wird, d. h. $GF(2)[x]/x^8+x^4+x^3+x+1$ ($x^8+x^4+x^3+x+1 \mathrel{\hat{=}} 0\mathrm{x}11\mathrm{B}$ in Koeffizientendarstellung).

Berechnen Sie die Matrix der Abbildung θ^{-1}. Hinweis: Die Struktur von θ^{-1} ist ähnlich der von θ.[1]

Da die verwendete S-Box im Paper ebenfalls nicht exakt spezifiziert ist, benutzen Sie die AES-S-Box.

3. Nun haben Sie alle Spezifikationen zusammen um die Implementierung in Angriff zu nehmen.Implementieren sie die Verschlüsselung eines Datenblocks mit dem BKSQ-Algorithmus in der Routine

   uint8_t bksq_encrypt(**uint8_t const** *plain, **uint8_t** *cyphertext, **uint8_t const** *key)

   Mit dem Code in `main.c` können Sie Ihre Implementierung testen.

4. Wenn die Verschlüsselung eines einzelnen Blockes funktioniert, können Sie den Counter-Modus zur Verschlüsselung größerer Datenmengen implementieren. Der Counter ist dabei eine 96 bit Datenstruktur: **uint8_t** counter[12]; Die untere Hälfte (counter[0]...counter[5];) wird mit einer 48-bit-nonce initialisiert. Die obere Hälfte mit 0. Diese Hälfte wird als 48-bit-Zahl in big-endian-Darstellung interpretiert. Für jeden Block wird diese Zahl um 1 erhöht. Wie viele Blöcke können somit maximal (mit einer nonce) verschlüsselt werden, bis sich der Schlüsselstrom wiederholt?

   Implementieren Sie nun den Counter-Mode in der Funktion

   uint8_t ctr(CONTEXT **const** ctx)

   Die benötigten Parameter werden dabei in einer Datenstruktur CONTEXT übergeben, um Fehler bei der Übergabe der Argumente zu vermeiden. Diese Datenstruktur ist vordefiniert und dokumentiert.

```
/**
 * A data structure holding all information relevant for en/decryption
 */
typedef struct {
    uint8_t *data; ///< a pointer to the input data
    uint32_t data_length; ///< length of the input data in bits; must be a multiple of the blocklength
    uint8_t const *key; ///< a pointer to the key
    uint8_t const *nonce; ///< a pointer to the nonce to be used for encryption/decryption
    uint8_t nonce_length; ///< the length of the none in bits
} CONTEXT;
```

Ein Beispiel für die Anwendung der Struktur finden Sie in der Datei `main.c` mit der Sie auch wieder Ihre Implementierung testen können.

4.2 Teil 2: Die Davies-Meyer-Hash-Konstruktion und HMAC

4.2.1 Hash

In der Vorlesung haben Sie verschiedene Möglichkeiten kennengelernt, um aus einer Blockchiffre eine Hash-Funktion zu konstruieren. Eine mögliche Variante ist die Davies-

[1] Nämlich: $\begin{bmatrix} \alpha & \alpha+1 & \alpha+1 \\ \alpha+1 & \alpha & \alpha+1 \\ \alpha+1 & \alpha+1 & \alpha \end{bmatrix}$ und $\theta\theta^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Meyer-Konstruktion. In diesem Teil der Aufgabe sollen Sie diese Methode implementieren. Auch hierfür gibt es dann in `main.c` einen kleinen Test.
Bei der Berechnung des Hashes wird ein definierter Initialisierungsvektor benötigt. Dieser sei auf auf eine entsprechend lange Folge von Nullbits festgelegt.

4.2.2 MAC

Um die Integrität von Nachrichten abzusichern, muss auch ein Schlüssel in die Hashfunktion eingehen. Das ergibt dann einen Message-Authentication-Code (MAC). In der Vorlesung haben sie verschiedene Arten von MACs kennen gelernt. Für diese Aufgabe sollen sie die HMAC-Konstruktion gemäß RFC 2104[2] implementieren. Als Hashfunktion verwenden sie die Konstruktion aus der vorherigen Aufgabe. Natürlich führt dies auf den ersten Block die Grundidee des HMAC, schnelle Hashfunktionen statt langsamerer Blockchiffren zur Berechnung eines MAC zu verwenden, ad absurdum. Im Sinne einer kompakten Implementierung (nur eine Kernfunktion) ist dies aber durchaus zu rechtfertigen.
Etwas vereinfachend sei noch festgelegt, dass der Schlüssel bereits passende Länge hat. Diese Aufgabe sollen Sie ebenfalls implementieren. Auch hierfür gibt es dann in `main.c` einen kleinen Test.
Der vorgegebene Funktionsprototyp hat einen zusätzlichen Parameter `uint8_t * data_prefix`. Hat dieser den Wert `NULL` so wird er einfach ignoriert. Andernfalls wird angenommen, dass `data_prefix` auf einen weiteren Datenblock passender Größe zeigt, der den eigentlichen Daten beim Hashen vorangestellt wird. Dies vereinfacht die Implementierung der folgenden Aufgabe.

4.3 Teil 3: Authenticated Encryption

Mit der Möglichkeit, die Integrität von Nachrichten abzusichern, lässt sich der Counter-Mode aus dem ersten Teil der Aufgabe zu einem Authenticated-Encryption-Verfahren ausbauen. Unter den vielen Möglichkeiten beides zu kombinieren sticht Encrypt-then-MAC wegen der einfachen Konstruktion und der nachgewiesenen Sicherheit[3] besonders heraus. Es wird einfach ein HMAC über den gesamten Cyphertext (mit vorangestelltem IV!) erzeugt.[4] Im Rahmen dieser Aufgabe wird für Verschlüsselung und HMAC der gleiche Schlüssel verwendet. Die 6-byte-nonce aus Aufgabe 1 wird dabei durch angehängte Nullen zu einem vollen Block ergänzt.
Implementieren sie Ihre Lösung. Es ergibt sich ein Authenticated-Encryption-Betriebsmodus für die Blockchiffre, der mit (asymptotisch) zwei Blockchiffre-Aufrufen pro Datenblock auch akzeptable Geschwindigkeit liefert.
Sicher ist Ihnen aufgefallen, dass sich der HMAC parallel zur Verschlüsslung berechnen lässt und bei geeigneter Implementierung alle Berechnungen mit einem Durchlauf

[2] https://tools.ietf.org/html/rfc2104,
zugehöriges Paper: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.134.8430
[3] http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.106.5488,
https://eprint.iacr.org/2013/433 (Abschnitt 6)
[4] vgl. https://eprint.iacr.org/2014/206, Fig. 8 (Middle)

über die Daten möglich sind. Eine solche Implementierung ist hier zwar nicht gefordert, doch zeigt dies, dass es sich um eine Online-Authenticated-Encryption (OAE) handelt. Es müssen nicht alle Daten zwischengespeichert werden. Wie bei allen OAE-Verfahren muss hier darauf geachtet werden, dass nonces wirklich nur einmal verwendet werden.[5] Die Testwerte in der Datei `main.c` verdeutlichen dies noch einmal. Gleichzeitig zeigt dieses Beispiel, dass der MAC tatsächlich vor einer gezielten Manipulation des Cyphertextes schützt. Die Testdaten aus Teil 1 und Teil 3 unterscheiden sich nur in 3 Bytes. Die MACs unterscheiden sich aber deutlich.

4.4 Teil 4: Finden von Hashkollisionen (optional)

Für die letzte Teilaufgabe ist die Zielplattform nicht Moodle, sondern ein leistungsstärkerer Rechner, etwa Ihr PC. Ihre Ergebnisse geben Sie bitte base16-kodiert aus. Dazu können Sie die Routine

```
uint8_t printbase16(uint8_t const * data, size_t const len) {
    size_t i;
    for (i = 0; i < len; i++) {
        printf("%02x", data[i]);
    }
    printf("\n");
}
```

aus `printbase16.c` aufrufen oder in Ihr Programm übernehmen. Ein mögliches Ergebnis sieht dann so aus:

```
input 1:f544d1ebce408517b0a9646f hash:2c5cbc97
input 2:ee0c628133a2b390364218d5 hash:2c5cbc97
```

Die Lösung müssen Sie nicht abgeben!

1. Eine Hashfunktion hat eine Ausgabebreite von n bits. Wieviele verschiedene Inputs müssen Sie betrachten um mit hoher Wahrscheinlichkeit (Stichwort: Geburtstagsparadoxon) eine Kollision zu finden?

2. Der Hashwert aus der Davies-Meyer-Konstruktion wird mit der nachfolgenden Methode auf einen 32-bit Hashwert verkürzt. Finden Sie, unter Ausnutzung des Geburtstagsparadoxons, eine Input-Kollision, d. h. zwei beliebige, aber voneinander verschiedene Datenblöcke, die denselben Hashwert ergeben. (Bei halbwegs effizienter Programmierung dauert dies auf einem aktuellen Rechner ca. 1 Sekunde.)

   ```
   void myhash32(uint8_t const *in, uint8_t * hash) {
       uint8_t tmp[BLOCKLEN_BYTES];
       dmhash(in, BLOCKLEN_BYTES * 8, tmp);
       int i;
       for (i = 0; i < 4; i++) {
           hash[i] = tmp[i] ^ tmp[i + 4] ^ tmp[i + 8];
       }
   }
   ```

[5] https://eprint.iacr.org/2015/189

3. Nun wird lediglich auf einen 48-bit Hashwert verkürzt. Können Sie auch hier Kollisionen finden? (Dies funktioniert in wenigen Minuten.)

```
void myhash48(uint8_t const *in, uint8_t * hash) {
    uint8_t tmp[BLOCKLEN_BYTES];
    dmhash(in, BLOCKLEN_BYTES * 8, tmp);
    int i;
    for (i = 0; i < 6; i++) {
        hash[i] = tmp[i] ^ tmp[i + 6];
    }
}
```

   Wie verändert sich die Laufzeit Ihres Programms in Abhängigkeit von der Hashlänge?

Abgabe

Zur Abgabe der Aufgaben loggen Sie sich bitte auf der Webseite unter Moodle mit Ihrer TUM-Kennung und Ihrem Passwort ein. Anschließend können Sie Ihre Lösungen per copy and paste in die vorgegebenen Textfelder eingeben.

*Bitte beachten Sie, dass Moodle keine Abgabe von mehreren Dateien unterstützt! Sie müssen deshalb **alle** von Ihnen verwendeten (Sub-)Funktionen in das Abgabefeld kopieren!*