Übungen zur Vorlesung Systemsicherheit

Symmetrische Kryptographie

Tilo Müller, Reinhard Tartler, Michael Gernoth

Lehrstuhl Informatik 1 + 4

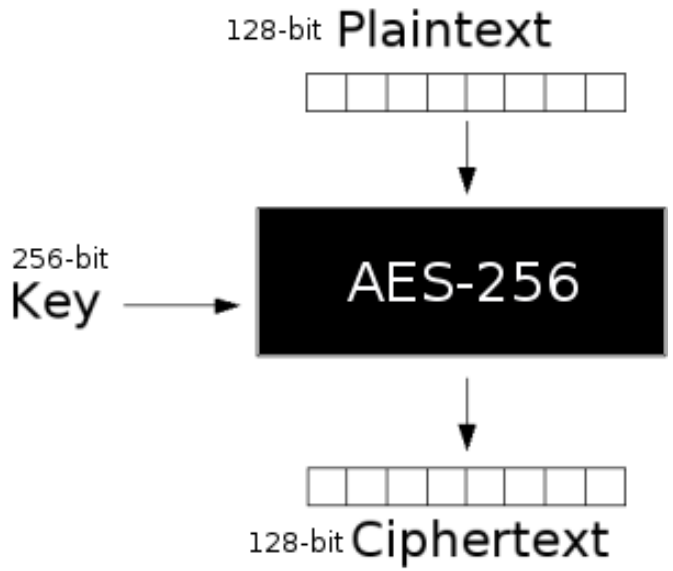
17. November 2010

Aufgabe 1.3

AES basierte Datei-Verschlüsselung

Aufgabe 1.3 | Block Ciphers

- Plain-Block $\rightarrow_{Key}$ Cipher-Block
- Block und Key haben feste Länge
- Beispiele:
  - DES: Blockgröße 64-Bit, Schlüssellänge 56-Bit
  - AES: Blockgröße 128-Bit, Schlüssellänge 128-, 192- oder 256-Bit



AES-Internals in Vorlesung; für die Übung kann AES als "Blackbox" betrachtet werden

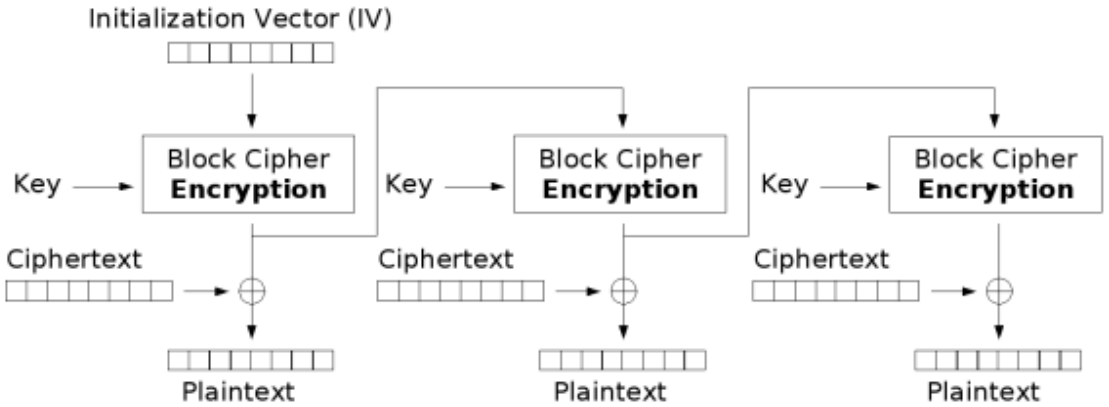
Aufgabe 1.3 | Block Cipher Modes

1. Problem: Wie werden Daten größer einem Block verschlüsselt?
→ Block Cipher Modes of Operation:

- Electronic Code Book (ECB):
  - Trivial: alle Blöcke unabhängig mit *demselben* Key verschlüsseln
  - ECB ist *unsicher*, da:
    - Gleiche Klartexte werden in gleiche Ciphertexte übersetzt
    - Cipherblöcke können beliebig vertauscht werden
- **Output Feedback Mode (OFB)**:
  - Aus dem Key wird ein *Schlüsselstrom* generiert und jeder Block mit einem anderen Teil des Stroms verschlüsselt
  - OFB wird in dieser Aufgabe verwendet
- Weitere: Cipher Block Chaining (CBC), Cipher Feedback (CFB), ...

2. Problem: Wie werden Daten verschlüsselt die nicht ein Vielfaches eines Blocks sind? → Padding

Aufgabe 1.3 | Output Feedback Mode



Output Feedback (OFB) mode decryption

- Verwandelt eine Block-Chiffre in eine *Strom-Chiffre*
- Die Block-Chiffre wird als *Schlüsselstrom-Generator* verwendet
- Die eigentliche Verschlüsselung geschieht durch *XOR*
- Eigenschaften:
  - Initialisierungs-Vektor (IV): **muss** zufällig sein.
  - Ver- und Entschlüsselung sind identisch; Schlüsselstrom kann unabhängig im Vorraus berechnet werden ($\rightarrow$ HW-Parallelität)
  - Übertragungsfehler breiten sich nicht aus; ...

Aufgabe 1.3 | Password-Based Key Derivation

Passwort: variable Länge, druckbare ASCII Zeichen
Schlüssel: feste Länge, alle Byte Werte

→ **Password-Based Key Derivation Function 2 (PBKDF2)**:

- Erstellt im Prinzip einen Hash über Passwort und Salt
- Iterationszahl einstellbar (hier: 2000)
- Salt und hohe Iterationszahl bremsen Bruteforce Attacken aus ("Key Strengthening")
- Teil der RSA Spezifikation; kann aber auch anderweitig verwendet werden (bspw. WPA/WPA2, OpenOffice, LUKS/dm-crypt, ...)
- Kann beliebige Schlüssellängen erzeugen (hier: 256-Bit)

Aufgabe 1.3 | HMAC

Keyed-Hash Message Authentication Code:

- angehängte Prüfsumme einer Nachricht;
- garantiert die **Integrität** einer Nachricht
- verhindert unbemerkte Manipulation; bei Festplattenverschlüsselung kann auch ein Defekt der Festplatte erkannt werden

$$\mathrm{HMAC}_{\mathrm{key}}(\mathrm{msg}) = \mathrm{HASH}((\mathrm{key} \oplus \mathrm{opad}) || \mathrm{HASH}((\mathrm{key} \oplus \mathrm{ipad}) || \mathrm{msg}))$$

hier:

- HASH = SHA1
- msg = (Data,Salt,IV)
- ipad/opad sind Konstanten

HMAC-SHA1 muss nicht selber implementiert werden, sondern wird von **OpenSSL** *zur Verfügung gestellt.*

Aufgabe 1.3 | OpenSSL

- Freie Implementierung des SSL/TLS Standards
- Kann als umfassende kryptographische Bibliothek genutzt werden:
  - Blockchiffren: AES, DES, IDEA, ...
  - Hash-Funktionen: MDx, SHA-x, HMAC, ...
  - Asymmetrisch: RSA, DSA, Diffie-Hellman, ...
- Wird eingesetzt von: OpenSSH, OpenVPN, ...

Aufgabe 1.3 | Programmieren mit OpenSSL

- Kryptographisch "starke" Zufallszahlen

```
#include ⟨openssl/rand.h⟩

int RAND_bytes(unsigned char *buf, int num);
```

- OpenSSL EVP: High-Level Schnittstelle kryptographischer Funktionen ("envelope")

```
#include ⟨openssl/evp.h⟩
```

- Linken:

```
gcc -lcrypto program.c
```

- Beispiel: Einlesen des Passworts von STDIN:

```
int EVP_read_pw_string(char *buf, int length, const char *prompt, int verify);
```

Aufgabe 1.3 | Programmieren mit OpenSSL

- OpenSSL und AES (`EVP_CIPHER` Schnittstelle):

```
EVP_CIPHER_CTX cctx;
EVP_CIPHER_CTX_init(&cctx);
EVP_CipherInit_ex(&cctx, EVP_aes_256_ofb(), NULL, key, iv, enc_dec);
EVP_CipherUpdate(...);
EVP_CIPHER_CTX_cleanup(&cctx);
```

- OpenSSL und PBKDF2 (nur SHA-1, undokumentiert)

```
#include ⟨openssl/evp.h⟩

int PKCS5_PBKDF2_HMAC_SHA1(char *pass, int passlen,
char *salt, int saltlen, int iter, int keylen, char *out);
```

- OpenSSL und HMAC

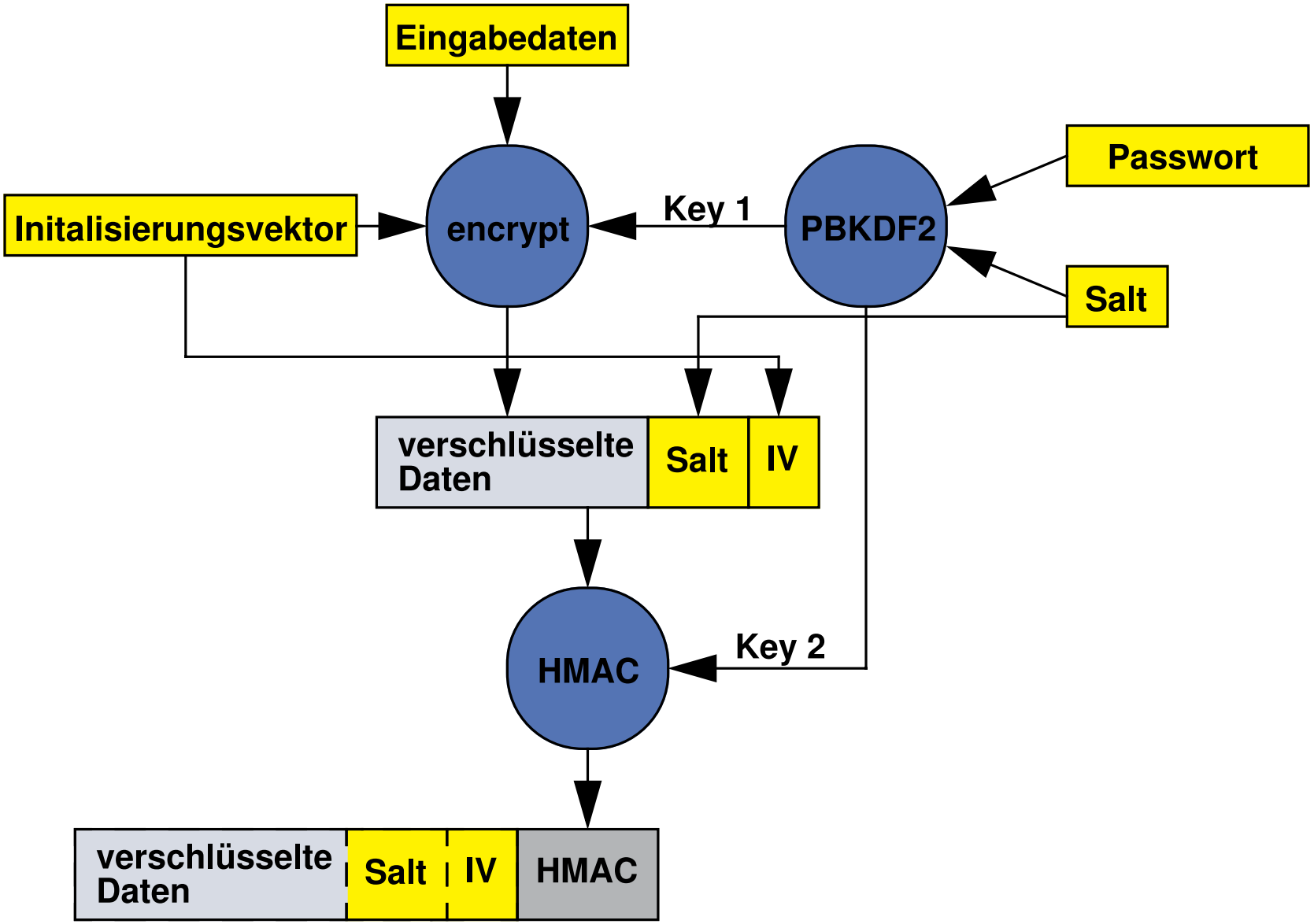
```
#include ⟨openssl/hmac.h⟩

unsigned char *HMAC(EVP_MD *evp_md, void *key, int key_len,
unsigned char *d, int n, char *md, int *md_len);
```

Aufgabe 1.3 | filecrypt

```
$./filecrypt enc test.txt test.enc.txt
Enter Password: geheim
$./filecrypt dec test.enc.txt text.txt
Enter Password: geheim
```

- AES, 256 Bit (32 Byte), Output-Feedback Mode (OFB)
- zufälliger IV, 128 Bit (16 Byte)
- SHA1 HMAC, 160 Bit (20 Byte)
- Salt, 160 Bit (20 Byte)
- PBKDF2, SHA1, 2000 Iterationen, 2 x 256 Bit Keys

Aufgabe 1.3 | filecrypt



Aufgabe 1.4

Fortgeschrittener Passwort Cracker

Aufgabe 1.4 | Advanced Password Cracker

Weiterentwicklung Ihrer Lösung aus Aufgabe 1.2

- Vorbild: *John the Ripper*
  - Benchmarking
  - Wörterbuch-Attacken
  - Muster-basierte Angriffe
  - ...
- Darüberhinaus: Parallelisierung
  - Mehrere Cores / Prozessoren
  - Verteilte Systeme (Cluster)
  - ...

Ende

- Nächste Woche: Aufgabenblatt 2, Asymmetrische Kryptographie
- Jetzt wieder Gruppenarbeit / Fragestunde