Karlsruhe Institute of Technology

Praktikum Kryptoanalyse

Sommersemester 2020

Willi Geiselmann
Thomas Agrikola
Simon Hanisch

Institut für Theoretische Informatik
Arbeitsgruppe Kryptographie
Fakultät für Informatik
Karlsruher Institut für Technologie

Organisatorisches

Vorbesprechung:	ca. alle 2 Wochen, nach Vereinbarung
Versuche:	Raum 258
System:	Linux
	ANSI-C

Ansprechpartner: **Willi Geiselmann**, Raum 271
Tel.: 0721/608-46297
E-Mail: `geiselmann@kit.edu`

Thomas Agrikola, Raum 256
Tel.: 0721/608-41863
E-Mail: `thomas.agrikola@kit.edu`

Simon Hanisch, Raum 241
E-Mail: `simon.hanisch@partner.kit.edu`

Inhaltsverzeichnis

Teil I.

Einleitung und (System-)Grundlagen

1. Ein Überblick über Unix

1.1. Das Dateisystem

Unix besitzt ein baumartiges Dateisystem. Auch Dateisysteme, die sich auf unterschiedlichen Platten oder Rechnern befinden werden in diesen Baum abgebildet. Bei Angabe von Verzeichnispfaden werden die einzelnen Verzeichnisse durch *Slashes* `"/"` voneinder getrennt. Das aktuelle Verzeichnis wird mit `"."` bezeichnet. Das übergeorgnete Verzeichnis mit `".."`. Das home-Verzeichnis eines Benutzers (das aktuelle Verzeichnis unmittelbar nach dem Anmelden) wird mit `"~"`, das home-Verzeichnis eines anderen Benutzers (z.B. user) mit `"~user"` bezeichnet. Das oberste (root-) Verzeichnis wird in dieser Notation mit `"/"` charakterisiert. Mit dem Befehl `cd` kann das aktuelle Verzeichnis geändert werden. Dieser wird in Abschnitt 1.3.2.1 genauer erläutert.

1.1.1. Datei- und Ordnerberechtigungen

Es gibt drei grundlegende Rechte die Nutzer auf Dateien und Ordnern haben können: Lesen (read, r), schreiben (write, w) und ausführen (execute, x). Diese Rechte können bei Dateien (und Ordnern) für den Besitzer der Datei, der Besitzergruppe der Datei, und allen anderen Nutzern vergeben werden. Der Befehl `stat` listet diese Rechte als eine Menge von Flags auf. `"drwxr-xr-x"` etwa bedeutet dass es sich um ein Verzeichnis handelt (`"d"`) auf das der Nutzer (die nächsten drei Zeichen, `"rwx"`) Lese- und Schreibrechte hat (`"rw"`) und sich in diesem Verzeichnis befinden darf (`"x"`). Andere Nutzer die der gleichen Gruppe angehören dürfen sich in dem Verzeichnis bewegen und die Inhalte ansehen, aber nicht schreiben (`"r-x"`). Die letzten drei Zeichen beschreiben die Rechte für alle anderen Nutzer (ebenfalls `"r-x"`).

Eine alternative Schreibweise der Dateirechte ist eine dreistellige Oktalzahl, wobei die Oktanzahlen jeweils die Rechte für Nutzer, Gruppe und Rest angeben. Die Berechtigungen sind dabei als Bits zu interpretieren, die Berechtigungen `"rwxr-xr-x"` resultieren zum Beispiel in der Oktalzahl 755.

Die Zugriffsrechte können von dem Dateibesitzer über `chmod` gesetzt werden. `"chmod 751 a.out"` etwa setzt die entsprechenden Rechte für die Datei a.out. Es können auch bestehende Rechte modifiziert werden: `"chmod u+x a.out"` gibt dem Nutzer (user, u) die Rechte zum Ausführen, hält aber alle anderen Berechtigungen bei.

1.2. Das Hilfesystem (man)

Eine der zentralen Einrichtungen unter Unix ist das Hilfesystem. Es besteht aus einer Sammlung von einzelnen Seiten, so daß jedes installierte Programpaket auch seine eigene Hilfeseite installieren kann. Mit dem Kommando `man thema` kann man sich die zu dem Thema gehörende Seite anzeigen lassen. Der Befehl `man` steht hier für `"Manual"`. So gibt etwa der Befehl `"man man"` eine ausführliche Einleitung in das Hilfesystem. Als Themen sind normalerweise Kommandos zugelassen. Wenn man das richtige Thema nicht kennt, so kann man versuchen das Thema über `apropos begriff` zu finden. Dabei werden die Titelzeilen aller Hilfeseiten nach dem jeweiligen Begriff abgesucht.

1.3. Die Shell

Unix besitzt (normalerweise) eine kommandoorientiere Schnittstelle. Den Kommandointerpereter nennt man *Shell*. Es gibt eine Reihe von verschiedenen Shells. Unter der X-Window-Oberfläche hat man ebenfalls Zugriff auf den Kommandointerpreter: Das Programm `xterm` erzeugt ein Fenster, in dem die Shell läuft.

1.3.1. Grundlegende Eigenschaften von Unix Shells

1.3.1.1. IO-Redirection

Jedem Prozess unter Unix ist eine Standardeingabe und eine Standardausgabe zugeordnet. Diese verhalten sich wie normale Dateien und sind normalerweise die Tastatur bzw. das dem Prozess zugeordnete Fenster. Es besteht nun die Möglichkeit Standardein- bzw. ausgabe auf Dateien umzulenken. Um die Standardeingabe umzulenken wird das Zeichen `"<"` verwendet. Für die Standardausgabe entsprechend das Zeichen `">"`. Um die Ausgabe eines Kommandos in eine Datei umzulenken verwendet man damit folgende Befehlszeile:

```
kommando > datei.
```

1.3.1.2. Pipes

Ebenso wie Standardein- und ausgabe in Dateien umgelekt werden können, kann man sie auch direkt an andere Prozesse weiterleiten. Hierfür werden sogenannte *pipes* eingesetzt. Das Pipesymbol ist `"|"`. Um die Ausgabe vom *Kommando1* als Eingabe für *Kommando2* zu verwenden und die Ausgabe von *Kommando2* als Eingabe für *Kommando3* wird folgende Befelszeile eingestezt:

```
kommando1 | kommando2 | kommando3
```

1.3.1.3. Hintergrund- und Vordergrundprozesse

Wird ein Kommando ohne weiter Zusätze gegeben, so wird das Kommando im Vordergrund abgearbeitet. Dies hat zur Folge, daß keine weiteren Kommandos über diese Shell gestartet werden können bevor das erste Kommando nicht beendet wurde. Wir das Kommando jedoch als

```
kommmando &
```

gegeben, so wird es im Hintergrund ausgeführt. Es ist daher möglich von der Shell aus noch weitere Kommandos zu starten bevor das erste Kommando beendet wurde. Siehe auch Abschnitt Prozessverwaltung 1.3.2.2.

1.3.1.4. Besondere Zeichen

In der Shell gibt es einige Zeiche mit besonderer Bedeutung:

<**Control C**> beendet das im Vordergrund laufende Kommando.

<**Control D**> ist die Kennung für das Dateiende. Es obliegt der jeweiligen Applikation geeignet darauf zu reagieren.

<**Control Z**> unterbricht das im Vordergrund laufende Kommando. Das Kommando kann dann mit *fg*, *bg* oder *kill* fortgeführt oder beendet werden.

1.3.1.5. Umgebungsvariablen

Die Shell selbst und Programme die daraus aufgerufen werden verwenden Umgebungsvariablen um ihren Ablauf zu beeinflusst. Die Variable `$HOME` etwa enthält das Heimverzeichnis des aktuellen Nutzers, `$PWD` das aktuelle Verzeichnis. Diese können als Befehlsparameter verwendet werden,

```
echo $PWD
```

gibt zum Beispiel das aktuelle Verzeichnis aus. Über den Befehl

```
env
```

können alle gesetzten Umgebungsvariablen und deren Werte aufgelistet werden. Variablen können für einzelne Befehle gesetzt werden indem man deren Definition dem Befehl voran hängt:

```
LC_ALL=en_GB make
```

führt den *make* Befehl mit englischer Ausgabe aus. Um eine Umgebungsvariable für die komplette Sitzung zu setzen, kann *export* verwendet werden:

```
export LC_ALL=en_GB
```

bewirkt dass alle nachfolgenden Befehle auf Englisch ausgeführt werden. Die Aufgaben des Praktikums verwenden die Umgebungsvariable `$PRAKTROOT`, auf den Computern des Lehrstuhles ist diese bereits entsprechend gesetzt.

1.3.2. Die wichtigsten Dienstprogramme und Kommandos

1.3.2.1. Verzeichnis- und Dateiverwaltung

Die wichtigsten Befehle für die Verzeichnisverwaltung sind:

pwd print working directory: gibt das aktuelle Verzeichnis aus.

cd change directory: ändert das aktuelle Verzeichnis.

- `"cd ~user"` wechselt zum Homeverzeichnis des Benutzers *User*.
- `"cd tmp"` wechselt in das Unterverzeichnis *tmp* des gegenwärtigen Verzeichnisses.
- `"cd /abc"` wechselt in das (absolut angegebene) Verzeichnis *abc*.
- `"cd .."` wechselt zum übergeordnetem Verzeichnis.

mkdir make directory: erstellt ein neues Verzeichnis.

rmdir remove directory: löscht ein Verzeichnis.

ls list: listet die Inhalte eines Verzeichnisses.

- `"ls"` gibt die Inhalte des aktuellen Verzeichnisses aus.
- `"ls /usr/bin"` gibt die Inhalte in /usr/bin aus.

cp copy: kopiert ein Verzeichnis oder eine Datei.

- `"cp a.txt ~"` kopiert die Datei a.txt in das Homeverzeichnis.

mv move: verschiebt eine Datei oder ein Verzeichnis.

stat status: gibt verschiedene Informationen zu Dateien aus, etwa deren Besitzer oder Zugriffsrechte.

1.3.2.2. Prozessverwaltung

Die wichtigsten Kommandos um Prozesse zu verwalten sind:

ps ohne Parameter listet alle Unterprozesse der gegenwärtigen login-Shell auf. Um alle Prozesse des Systems mit weiteren Angaben zu bekommen kann das Kommando `"ps -ef"` eingestezt werden. Die weiteren (sehr zahlreichen) Optionen zu *ps* sind in der man-page zu finden.

kill Um einen Prozess zu vernichten, der außer Kontrolle geraten ist wird das Kommando *kill* verwendet. Als Parameter wird die Process-ID (PID) angegeben. Sie kann über das Kommando *ps* ermittelt werden. Sollte `"kill PID"` keinen Erfolg haben, so gibt es als zweite Möglichkeit `"kill -9 PID"` was den Prozess auf jeden Fall beendet.

1.3.2.3. less

less ist ein hifreiches kleines Programm, das verwendet werden kann um Texte anzuzeigen, die sonst über den Bilschirm hinausrollen würden. Der Aufruf kann entweder über eine *pipe* (siehe auch Abschnitt 1.3) oder im Format `less dateiname` erfolgen. *less* stellt jeweils eine Bilschirmseite Text dar. Mit der <Leertaste> kann man um einen Bilschirm nach vorne rollen, mit <Return> um eine Zeile. <b> rollt eine Seite zurück,

`</Begriff>` sucht einen Begriff, `<n>` sucht das nächste Vorkommen des letzten Suchbegriffs. `<q>` beendet less. Weitere Dokumentation über die entsprechende man-page.

1.4. Übersetzen von Programmen

Um den Quellcode auszuführen, muss dieser erst übersetzt werden. Dies geschieht in zwei Schritten, erst werden die einzelnen Quelldateien übersetzt, dann werden sie zusammen gelinkt. Beide Aufgaben werden vom Compiler übernommen. Der systemeigene Compiler kann durch den Befehl *cc* aufgerufen werden. Die Praktikumsaufgaben verwenden den default Compiler der meisten Linux Distributionen, *gcc*.

Bei größeren Projekten werden zahlreiche Parameter und Befehle zum kompilieren benötigt. Um diese Arbeit zu erleichtern wurden so genannte Makefiles eingeführt. Diese enthalten eine Anleitung wie ein Projekt kompiliert werden muss.

Zum Übersetzen eines Programms `xyz.c` in ein ausführbares Programm `xyz` geben Sie den Befehl `make xyz` ein. `make` compiliert dann alle Programmteile, die `xyz` benötigt und die sich seit dem letzten `make`-Aufruf geändert haben, und linkt sie zusammen.

Um gezielt nur die Datei `xyz.c` zu compilieren, geben Sie `make xyz.o` ein. Ein manuelles Übersetzen kommt wegen der umfangreichen Menge an notwendigen Compiler-Optionen nicht in Frage.

Wird `make` ohne Parameter aufgerufen, so werden alle in dieser Aufgabe zu erstellenden Programme übersetzt. Dies ist aber nur dann sinnvoll, wenn Sie schon alle Rahmenprogramme ergänzt haben.

Das fertige Programm starten Sie mit `./xyz`.

Die Rahmenprogramme zu einer Aufgabe finden Sie in einer Sub-Directory in Ihrem Home-Verzeichnis und außerdem auch unter `$PRAKTROOT/versuche/VERSUCHSNAME`.

1.4.1. Makefiles

Zu jedem Versuch existiert ein sogenanntes `Makefile`, welches spezifiziert, aus welchen Quell-Dateien die zu einem Versuch gehörenden ausführbaren Programme zusammengesetzt sind. Beim Start überprüft `make`, welche Objekt-Dateien älter als die dazugehörenden Quell- oder Header-Dateien sind und übersetzt sie. Anschließend wird das ausführbare Programm gelinkt. Als Beispiel sei hier das `Makefile` der Vigenereversuchs aufgeführt:

```
 1 include $(PRAKTROOT)/include/Makefile.Settings
 2 SRC     = vigenere.c vigenere_attacke.c
 3 OBJ     = $(SRC:%.c=%.o)
 4 CFLAGS  = $(CFLAGS_DEBUG)
 5
 6 BINS    = vigenere vigenere_attacke
 7
 8 vigenere:       vigenere.o
 9         $(CC) -o $@ $@.o $(LDFLAGS)
10
11 vigenere_attacke:       vigenere_attacke.o
12         $(CC) -o $@ $@.o $(LDFLAGS)
13
14 all:    $(BINS)
15
16 #-----------------------------------------------
17
18 clean:
19         -rm -f *.o *~ *% $(BINS)
```

In Zeile 1 wird mit der Include-Anweisung eine globales Makefile gelesen, welches allgemeine Einstellungen

für das Praktikum enthält. In den Zeilen 2 bis 6 werden Variablen vereinbart. Zeile 4 zeigt das Auslesen einer Variable, nämlich `$(CFLAGS_DEBUG)`, welche in `Makefile.Settings` vereinbart wurde.

Zeile 8 und 9 zeigt eine Abhängigkeit und eine Erzeugungsregel. `vigenere` hängt von `vigenere.o` ab und wird durch das in Zeile 9 angegebene Kommando erzeugt. Analoges gilt für die Zeilen 11 und 12. Die Abhängigkeit von `vigenere.o` von `vigenere.c` wird durch eine eingebaute Regel beschrieben.

Zeile 18 und 19 zeigt schließlich die Verwendung eines „Pseudo-Ziels“. Wenn das Ziel „clean“ gemacht wird, werden alle Dateien, die durch Aufruf von `make` wieder erzeugt werden können, gelöscht.

Bei diesem Makefile können mit `make vigenere` bzw. `make vigenere_attacke` gezielt die beiden Versuchsteile übersetzt werden. `make all` erzeugt alle Dateien, `make clean` räumt auf.

2. Bibliotheken der Praktikumsumgebung

Zur Lösung der Praktikumsaufgaben stehen diverse Bibliotheken zur Verfügung, die die Programmierung der einzelnen Aufgaben vereinfachen. Die Bibliotheken werden durch Einbinden der entsprechenden Include-Dateien bei der Compilation und durch Angabe der Library beim Linken benutzt. Dies geschieht automatisch durch die Rahmenprogramme bzw. die Makefiles der einzelnen Versuche. Die Include-Dateien befinden sich unter `$PRAKTROOT/include`.

2.1. Grundlegende Datentypen

In `$PRAKTROOT/include/praktikum.h` werden zunächst die Standard-C Datentypen fester Größe aus `inttypes.h` eingebunden. Diese sind u.a.: vereinbart. Zudem wird ein String-Datentyp definiert.

```
uint8_t     /* vorzeichenlos,   8 Bit */
uint16_t    /* vorzeichenlos,  16 Bit */
uint32_t;   /* vorzeichenlos,  32 Bit */
int8_t;     /* mit Vorzeichen,  8 Bit */
int16_t;    /* mit Vorzeichen, 16 Bit */
int32_t;    /* mit Vorzeichen, 32 Bit */

#define STRINGLEN 256
typedef char String[STRINGLEN];  /* Typ für Zeichenketten  */
```

2.2. DES (Data Encryption Standard) Funktionen

Beim DES handelt es sich um eine Blockchiffre, die – parametrisiert durch einen 56 Bit großen Schlüssel – 64 Bit Daten ver- bzw. entschlüsselt.

Ein Schlüssel (bestehend aus 8 Bytes, bei denen das höchstwertigste Bit nicht mitbenutzt wird) muss zunächst mit der Funktion `DES_GenKeys` in eine interne Darstellung gewandelt werden. Außerdem wird beim Aufruf von `DES_GenKeys` angegeben, ob ver- oder entschlüsselt werden soll.

Mit diesem Schlüssel in interner Darstellung kann dann über die restlichen DES-Funktionen ein Datenblock ver- und entschlüsselt werden.

Die Deklarationen der DES-Funktionen finden sich in `$PRAKTROOT/include/praktikum.h`

2.2.1. Datentypen der DES-Routinen

Von den DES-Prozeduren werden die Datentypen `DES_key`, `DES_data` und `DES_ikey` benutzt, welche einen Schlüssel, einen zu bearbeitenden Datensatz oder einen internen Schlüssel darstellen. Sie haben folgende Form:

```
#define DES_DATA_WIDTH  8
typedef uint8_t DES_key[DES_DATA_WIDTH];
typedef uint8_t DES_data[DES_DATA_WIDTH];
typedef uint32_t DES_ikey[128/4];
```

2.2.2. DES_GenKey – Internen Schlüssel erzeugen

`DES_GenKey` wandelt den 8 Bytes langen Schlüssel `key` in die interne Darstellung um und legt ihn in `ikey` ab. Ist `decode` gleich 1, wird der Schlüssel zum Entschlüsseln aufbereitet, ansonsten zum Verschlüsseln,

```
void DES_GenKeys( const DES_key key, int decodeflg, DES_ikey ikey )
```

2.3. Zeichenkettenfunktionen

Die String-Funktionen der Praktikumsbibliothek arbeiten wie die Funktionen der C-Standard-Bibliothek auf 0-terminierten Zeichenketten. Zur einfachen Formulierung existiert der Datentyp `String` mit der Definition

```
#define STRINGLEN 256
typedef char String[STRINGLEN];
```

Diese „Strings“ sind bei Übergabe als Parameter an Funktionen wegen der Dualität von Arrays und Pointern in C äquivalent zu dem Datentyp `char *`.

Die Prozedur

```
void string_to_lower(char *s);
```

wandelt alle Zeichen in `s` in Kleinschrift um. Ihr Gegenstück ist die Funktion

```
void string_to_upper(char *s)
```

die alle Zeichen in Großbuchstaben wandelt.

Bei einigen Funktionen der Standard-C-Bibliothek (z.B. `fgets`) enthalten die zurückgelieferten Strings an ihrem Ende u.U. noch Linefeed-Zeichen. Mit der Funktion

```
void strip_crlf(char *s)
```

können alle „Carriage Return“ und „Line Feed“ Zeichen am Ende des Strings `S` entfernt werden.

Ein leidiges Problem bei der Handhabung von Zeichenketten in C ist das Konkatenieren. Die Funktion

```
char *concatstrings(const char *s1,...,NULL);
```

hängt die als Argument angegebenen Zeichenketten zu einer einzigen zusammen. Der dafür benötigte Speicher wird dynamisch alloziert. Kann der benötigte Speicher nicht belegt werden, wird das gesamte Programm mit einer Fehlermeldung terminiert. Damit `concatstrings` feststellen kann, wieviele Zeichenketten aneinandergehängt werden müssen, muss nach dem letzten String als abschließendes Argument `NULL` angegeben werden. Ein Beispiel:

```
{
  String datei;
  char *pfad;

  readline("Dateiname: ",datei,sizeof(datei));
  pfad=concatstrings("/usr/include/",datei,".h",NULL);
}
```

2.4. Ein- und Ausgabe von Daten

Die folgenden Funktionen vereinfachen das Einlesen diverser Datentypen von der Tastatur:

```
void readstring(const char *prompt, char *buffer, int size)
```

liest eine Zeile von maximal `size` Zeichen (das abschließende `'\0'` eingerechnet!) in `buffer` von der Tastatur ein. Zuvor wird `Prompt` ausgegeben.

Mit

```
int readint(const char *prompt);
```

wird nach der Ausgabe von `prompt` ein Integer im Bereich $-2^{31} \ldots 2^{31} - 1$ eingelesen. `Readint` verlangt automatisch eine neue Eingabe, wenn die vom Anwender eingegebene Zahl kein gültiger Integer-Wert ist.

Die Funktion

```
double readdouble(const char *prompt);
```

Liest analog zu `readint` eine Fließkommazahl ein und gibt sie zurück.

Der Aufruf von

```
char readchar(const char *prompt);
```

gibt zunächst `prompt` aus und liest dann genau ein Zeichen ein. Im Gegensatz zu den zuvor besprochenen Funktionen wartet `readchar` nicht auf ein abschließendes „Return“, sondern kehrt nach der Eingabe eines Zeichens sofort zum Aufrufer zurück.

2.5. Sonstige Funktionen

Die Funktion

```
uint32_t RandomNumber(void);
```

liefert eine Pseudo-Zufallszahl x im Bereich $0 \leq x < 2^{32}$ zurück.

Mit der Funktion

```
uint32_t GetCurrentTime(void);
```

kann die aktuelle Uhrzeit in Sekunden seit dem 1.1.1979, Mitternacht gelesen werden.

Eine andere Uhrzeitfunktion ist

```
const char *Now(void);
```

Sie liefert Datum und Uhrzeit in einem für Menschen lesbaren Format zurück.

2.6. Die GNU MP Bignum Library

GMP wird ab Versuch 6 benutzt. Einige der zur Verfügung stehenden Prozeduren sind der Dokumentation zur Arithmetik im Anhang B zu entnehmen. Eine vollständige Dokumentation ist online unter https://gmplib.org/manual/ verfügbar.

2.7. Die Network Support Library

Mit den Routinen dieser Bibliothek werden Verbindungen zwischen zwei Prozessen (über das Netzwerk) aufgebaut. Genaueres ist der Dokumentation im Anhang A zu entnehmen.

Die Deklarationen befinden sich in der Datei `$PRAKTROOT/include/network.h`.

3. Mathematische Grundlagen

3.1. Vorbemerkung

Im folgenden Abschnitt werden einige der mathematischen Grundlagen erklärt, die für die Versuche 6 und 7 notwendig sind. Dabei handelt es sich nicht um eine vollständige Darstellung des Stoffes. Vielmehr soll nur das Handwerkszeug vermittelt werden, das für die Versuche notwendig ist.

Die Beschreibung ist an einigen Stellen weniger formal als aus der Linearen Algebra bekannt. So werden zum Beispiel Elemente von Restklassenringen nicht als Äquivalenzklassen, sondern als ganze Zahlen dargestellt. Dies vereinfacht die Darstellung erheblich und kommt dem Gebrauch der Ringe in der Kryptographie wesentlich näher.

3.2. Gruppe, Ring und Körper

Definition 1 (**Gruppe**): Es sei $\mathbf{G}$ eine nichtleere Menge, auf der eine Verknüpfung '$\circ$' definiert ist. Das Paar $(\mathbf{G}; \circ)$ heißt Gruppe, wenn folgende drei Axiome erfüllt sind:

1. (Assoziativgesetz) $\forall x, y, z \in \mathbf{G} : (x \circ y) \circ z = x \circ (y \circ z)$.
2. (Neutrales Elements) $\exists n \in \mathbf{G}\ \forall x \in \mathbf{G} : n \circ x = x \circ n = x$.
3. (Inverses Element) $\forall x \in \mathbf{G}\ \exists y \in \mathbf{G} : x \circ y = y \circ x = n$.

Definition 2 (**abel'sche Gruppe**): Gilt in der Gruppe $(\mathbf{G}, \circ)$ zusätzlich das Axiom:

$$\forall x, y \in \mathbf{G} : x \circ y = y \circ x,$$

so heißt die Gruppe *kommutative* oder *abelsche Gruppe*.

Definition 3 (**Ring**): Es sein $\mathbb{R}$ eine Menge mit zwei Verknüpfungen $+$ und $\circ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Das Tripel $(\mathbb{R}; +, \circ)$ heißt *Ring*, wenn die folgenden drei Axiome gelten:

1. $(\mathbb{R}; +)$ ist eine abelsche Gruppe (ihr neutrales Element n wird mit 0 bezeichnet).
2. $\forall x, y, z \in \mathbb{R} : (x \circ y) \circ z = x \circ (y \circ z)$ (Assoziativität der Multiplikation),
3. Die Distributivgesetze gelten:
   - $\forall x, y, z \in \mathbb{R} : x \circ (y + z) = x \circ y + x \circ z$,
   - $\forall x, y, z \in \mathbb{R} : (x + y) \circ z = x \circ z + y \circ z$.

Definition 4 (Ring mit 1): Ein Ring $\mathbb{R}$ heißt *Ring mit 1*, falls ein Element $1 \in \mathbb{R}$ existiert, für das gilt:

$$\forall x \in \mathbb{R} : 1 \circ x = x \circ 1 = x.$$

Definition 5 (Kommutativer Ring): Ist die Multiplikation '$\circ$' kommutativ, so heißt der Ring ein *kommutativer Ring*.

Definition 6 (Körper): Eine Struktur $(\mathbb{K}; +, *)$ wird als *Körper* bezeichnet, falls gilt:

1. $(\mathbb{K}; +)$ ist eine kommutative Gruppe.

2. $(\mathbb{IK}\backslash\{0\}; *)$ ist eine kommutative Gruppe (0 ist das neutrale Element von $+$).
3. $\forall x, y, z \in \mathbb{IK} : x * (y + z) = x * y + x * z$

Man bezeichnet $\mathbb{IK}^* = (\mathbb{IK}; *)$ auch als die *multiplikative Gruppe* des Körpers $\mathbb{IK}$.

3.3. Ringe und Körper mit endlich vielen Elementen

Definition 7 $\mathbb{Z}_n := (\{0, \ldots, n-1\}; +, *)$ mit Addition bzw. Multiplikation modulo n heißt Restklassenring. Ist n eine Primzahl, so bildet $\mathbb{Z}_n$ einen Körper[1].

Definition 8 $\mathbb{Z}_n^* := (\{x \mid 1 \leq x < n \wedge ggT(x, n) = 1\}; *)$ mit der Multiplikation modulo n heißt die multiplikative Gruppe zum Ring $\mathbb{Z}_n$. $\mathbb{Z}_n^*$ ist kommutativ. Ist n prim, so gilt $\mathbb{Z}_n^* = \mathbb{Z}_n\backslash\{0\}$.

Definition 9 $\varphi(n) := |\mathbb{Z}_n^*|$ heißt Euler'sche Funktion.
Ist p prim, so gilt $\varphi(p) = p - 1$.

Satz 10 (Kleiner Fermat): Sei $a, n \in \mathbb{Z}\backslash\{0\}, ggT(a, n) = 1$. Es gilt: $a^{\varphi(n)} \equiv 1 \ mod \ n$.
Insbesondere gilt für p prim und $a, b \in \mathbb{N} : a^b \ mod \ p = a^{b \ mod \ p-1} \ mod \ p$

Für eine Primzahl p bildet $\mathbb{Z}_p$ also einen endlich Körper. $\mathbb{Z}_p$ wird auch als $GF(p)$ (Galois-Feld) bezeichnet. Allgemein sind nur endliche Körper mit p^n, p prim, vielen Elementen möglich. Da die Operationen in $GF(p^n)$ für $n > 1$ nicht leicht in Software zu realisieren sind, betrachten wir im Rahmen des Praktikums nur $GF(p)$.

In den folgenden Abschnitten bezeichnet p immer eine Primzahl. Somit ist $\mathbb{Z}_p$ ein endlicher Körper und $\mathbb{Z}_p^*$ die dazugehörende multiplikative Gruppe.

3.4. Primitives Element

Definition 11 $a \in \mathbb{Z}_n . \langle a \rangle := \{a^i \mid i \in \mathbb{N}, i > 1\}$ heißt Erzeugnis von a.

Definition 12 $ord(a) := |\langle a \rangle|$ heißt Ordnung von a.

Definition 13 Ein $\omega \in \mathbb{Z}_p^*$ heißt primitives Element (kurz p.E.) in $\mathbb{Z}_p^*$, wenn gilt: $ord(\omega) = p - 1$.

Satz 14 $\omega \in \mathbb{Z}_p^*$ ist genau dann primitives Element, wenn gilt:

$$\omega^i \neq 1 \quad \forall i : \ 1 \leq i < p - 1 \quad \wedge \quad \omega^{p-1} = 1$$

Satz 15 Aus $\omega \in \mathbb{Z}_p^*$ p. E. folgt: $\langle \omega \rangle = \mathbb{Z}_p^*$.

Satz 16 Sei $a \in \mathbb{Z}_p^*$, so gilt: $\langle a \rangle$ ist Untergruppe von $\mathbb{Z}_p^*$. $ord(a)$ teilt $p - 1$

Satz 17 Da die Ordnung eines Elements $ord(a)$ $p-1$ teilt, ist es bei bekannter Primzahlzerlegung von $p-1$ leicht möglich, ein Element auf Primitivität zu untersuchen:
Sei:

$$p - 1 = \prod_{i=1}^{k} {p_i}^{\alpha_i}$$

Dann gilt:

$$\omega \in \mathbb{Z}_p^* \text{ p. E. } \iff \omega^{\frac{p-1}{p_i}} \not\equiv 1 \ mod \ p \quad \forall i : 1 \leq i \leq k$$

[1] $\mathbb{Z}_6$ ist nicht nullteilerfrei $(2 * 3 = 0 \bmod 6)$ und somit kein Körper.

Bemerkung 18 Mit $a \in \mathbb{Z}_p^*$ ist auch $a^{-1} \in \mathbb{Z}_p^*$. Zur Berechnung von a^{-1} wird der erweiterte Euklid-Algrithmus zur Bestimmung des größten gemeinsamen Teiles benutzt. Dieser Algorithmus liefert zu zwei $u, v \in \mathbb{N}$ Zahlen $u', v' \in \mathbb{Z}, w \in \mathbb{N}$ mit $u * u' + v * v' = w = ggT(u, v)$.
Für $a < p$ liefert der Algorithmus also: $u * a + p * v = 1 = ggT(a, p)$. Betrachtet man diese Gleichung in $\mathbb{Z}_p$, so ist $p * v \equiv 0$ und es gilt $a * u \equiv 1$. Somit ist $u = a^{-1}\ mod\ p$.
Dieses Verfahren kann auch zur Bestimmung von multiplikativen Inversen modulo n benutzt werden. a ist nur dann invertierbar, wenn $ggT(a, n) = 1$ gilt. Dies ist keine Einschränkung des Algorithmus, sondern folgt aus der Definition von $\mathbb{Z}_p^*$.

3.5. Ein Beispiel

Wir betrachten nun die multiplikative Gruppe $\mathbb{Z}_7^*$. Wir wollen von allen Elementen die Ordnung bestimmen und außerdem primitive Elemente finden:

	1	2	3	4	5	6
x^1	$1^1 = 1$	$2^1 = 2$	$3^1 = 3$	$4^1 = 4$	$5^1 = 5$	$6^1 = 6$
x^2		$2^2 = 4$	$3^2 = 2$	$4^2 = 2$	$5^2 = 4$	$6^2 = 1$
x^3		$2^3 = 1$	$3^3 = 6$	$4^3 = 1$	$5^3 = 6$	
x^4			$3^4 = 4$		$5^4 = 2$	
x^5			$3^5 = 5$		$5^5 = 3$	
x^6			$3^6 = 1$		$5^6 = 1$	
$ord(x)$	1	3	6	3	6	2

Man sieht: 3 und 5 sind primitive Elemente.
$\langle 3 \rangle = \{3^i \mid 1 \leq i \leq 6\} = \mathbb{Z}_7^*$ und $\langle 5 \rangle = \mathbb{Z}_7^*$.
Man verifiziere: $\langle 1 \rangle = \{1\}$, $\langle 2 \rangle = \{1, 2, 4\}$ und $\langle 6 \rangle = \{1, 6\}$ bilden Untergruppen von $\mathbb{Z}_7^*$. Die Mächtigkeit der Untergruppen teilt 6.

3.6. Das Diskrete Logarithmus Problem

Seien $a, x \in \mathbb{Z}_p^*$. Betrachte:

$$\begin{array}{lll} f: & x \longrightarrow & a^x\ mod\ p \\ f^{-1}: & y \longrightarrow & dlog_a y \end{array}$$

Ist a primitives Element in $\mathbb{Z}_p^*$, so ist f bijektiv und f^{-1} existiert und ist ebenfalls bijektiv.

Die Funktion f heißt Modulo-Exponentation zur Basis a, f^{-1} ist der Diskrete Logarithmus. f ist leicht zu implementieren (vergl. Versuch 6). Zur Berechnung von f^{-1} für hinreichend große p existieren jedoch keine effizienten Algorithmen. Für eine Zahlenlänge von 512 Bit ist die Berechnung des diskreten Logarithmus in diesem Universum nicht mehr möglich. Derartige Funktionen werden als *Einwegfunktionen* oder auch englisch *oneway functions* genannt.

Beispiel: Für $4 \in \mathbb{Z}_7^*$ gilt $4^2 = 2 = 4^5$. Somit existiert f^{-1} nicht.
$5 \in \mathbb{Z}_7^*$ ist primitives Element. Es gilt:

x	1	2	3	4	5	6
$f(x) = 5^x$	5	4	6	2	3	1
$f^{-1}(x) = dlog_5(x)$	6	4	5	2	1	3

3.7. Faktorisierungsproblem

Seien p, q prim mit $p \neq q$ und $n := p * q$. Seien außerdem $x, e \in \mathbb{Z}_n^*$ mit $e, \phi(n)$ teilerfremd. Betrachte:

$$\begin{array}{lll} f: & x \longrightarrow & x^e\ mod\ n \\ f^{-1}: & y \longrightarrow & y^{e^{-1}\ mod\ \varphi(n)}\ mod\ n \end{array}$$

Ist nur n und nicht die Faktorisierung $p*q$ bekannt, so kann bei öffentlichem e die Funktion f leicht berechnet werden. Zur Bestimmung von $f^{-1}(y)$ ist jedoch die Kenntnis von e^{-1} d.h. die Faktorisierung von $n = p * q$ erforderlich[2]. Faktorisieren von großen Zahlen ist ein ähnlich schweres Problem, wie die Bestimmung des Diskreten Logarithmus.

Für Personen, die p und q kennen, ist die Berechnung von e^{-1} und somit $f^{-1}(y)$ leicht möglich.

Solche Funktionen, deren Umkehrfunktionen im Allgemeinen schwer und bei Kenntnis eines Geheimnisses leicht zu berechnen sind, nennt man *Einwegfunktionen mit Falltür* bzw. auf englisch *oneway functions with trapdoor*. Die Falltür ist hier die Kenntnis der Faktorisierung von n.

Die Funktion f wird bei dem bekannten Public-Key-Verfahren RSA benutzt.

3.8. Chinesischer Restsatz

Der Chinesische Restsatz ist ein sehr nützliches Werkzeug, um in endlichen Gruppen zu rechnen. Konkret sagt er aus:

Satz 19 Seien p und q teilerfremde Zahlen, und für ein $0 \leq x < p \cdot q$, $x_p = x \mod p$, $x_q = x \mod q$, dann ist x durch x_p und x_q eindeutig bestimmt.

Eine Folgerung daraus ist:

Sei nun o.B.d.A $p < q$, dann gilt für $p' = p^{-1} \mod q$:

$$x \equiv x_p + (x_q - x_p) \cdot p \cdot p' \mod p \cdot q$$

Da

$$x_p + (x_q - x_p) \cdot 0 \cdot p' = x_p = x \mod p$$

und

$$x_p + (x_q - x_p) \cdot p \cdot p^{-1} = x_p + (x_q - x_p) \cdot 1 = x_q = x \mod q$$

und x durch x_p und x_q eindeutig bestimmt ist, folgt, dass man mit der gegebenen Formel x rekonstruieren kann.

Die für die Implementierung einer konkreten Berechnung benötigten Funktionen wie Moduloberechnung, Moduloinvertierung etc. sind in GMPlib zu finden und in dem entsprechenden Anhang dokumentiert.

[2] Dies ist nicht ganz korrekt: Kann man n faktorisieren, so ist die Berechnung von e^{-1} trivial. Andererseits kann es andere Möglichkeiten zur Berechnung von f^{-1} geben!

4. Hinweise zur Einrichtung der Versuche

Die Versuche verwenden zwei Umgebunsvariablen. Die Variable `"$PRAKTIKUM_NAME"` wird als Gruppenname dem Server mitgeteilt. Die Variable `"$PRAKTROOT"` wird verwendet um nach libpraktikum in `"$(PRAKTROOT)/lib"` und diversen Headerdateien (sowie Makefile.Settings) in `"$(PRAKTROOT)/include"` zu suchen. Wie Umgebungsvariablen verwendet und gesetzt werden wird in Kapitel 1.3.1.5 genauer beschrieben.

Es wird außerdem die GMPLib verwendet. Auf Debian z.B. heißt das dafür benötigte Paket `libgmp-dev`.

Teil II.

Versuchsunterlagen

5. Klassische Chiffrierverfahren

Die wohl bekanntesten klassischen Chiffrierverfahren sind der *Caesar* und der *Vigenère*. Bei beiden Verfahren handelt es sich um Transpositionschiffren, d.h. die Zeichen des Alphabets (in diesem Versuch nur Großbuchstaben `'A'` bis `'Z'`) werden um einen bestimmten Offset verschoben. Überschreitet man dabei das `'Z'`, so geht es wieder bei `'A'` weiter. Dies ist mit einer Modulo-Arithmetik vergleichbar.

Bei der Caesar-Chiffre werden alle Zeichen um denselben Betrag verschoben, bei der Vigenère-Chiffre wird der Verschiebungsbetrag durch einen Schlüssel, bestehend aus mehreren Zeichen, festgelegt: Sei $m_1 \dots m_n$ der Klartext und $k_1 \dots k_l$ der Schlüsseltext, so werden die Zeichen $m_1, m_{l+1}, m_{2*l+1}, \dots$ um k_1, $m_2, m_{l+2}, m_{2*l+2} \dots$ um k_2 usw. wie bei einer Caesarchiffre verschoben.

Genaueres zu beiden Chiffren sowie zur Attacke mit Hilfe der Autokorrelation findet sich in den Kapiteln 5.3, 5.4 und 5.5.

5.1. Aufgabe: Implementierung der Vigenère-Chiffre

Bei unserer Version der Vigenère-Chiffre sollen nur Großbuchstaben verschlüsselt werden. Kleinbuchstaben werden zuvor in Großbuchstaben gewandelt, Satz-, Sonder- und Leerzeichen werden ignoriert. Sie werden auch bei der Weiterschaltung der Schlüssel nicht beachtet.

Der Schlüsseltext sollte nur aus Buchstaben bestehen. Kleinbuchstaben werden ebenfalls in Großschrift gewandelt. Ein `'A'` im Schlüsseltext soll eine Verschiebung der Klartextzeichen um eine Stelle bewirken.

Ergänzen Sie im Rahmenprogramm `vigenere.c` die Prozeduren `Encipher` und `Decipher`. Ergänzen Sie die Variablendeklaration um die von Ihnen benötigten Variablen und initialisieren Sie diese.

5.2. Attacke über Autokorrelation

In diesem Teilversuch soll mit Hilfe der Autokorrelation (vergl. 5.4 und 5.5) ein mit Vigenère verschlüsselter Text (Datei `testtext.ciph`) entschlüsselt werden.

Das Rahmenprogramm `vigenere_attacke.c` liest dazu die zu entschlüsselnde Datei in den Speicher (Prozedur `GetFile`) und lädt außerdem eine Tabelle, welche zu jedem Großbuchstaben dessen relative Häufigkeit in der englischen Sprache angibt (Array `PropTable`).

1. Berechnen Sie die Autokorrelationen für Perioden zwischen 1 und einem Maximalwert.
   Das Rahmenprogramm ruft dann `gnuplot` auf, um die berechneten Autokorrelationswerte über die Periode aufzuzeichnen.
2. Berechnen Sie die Periodenlänge (Entweder aus dem `gnuplot`-Output oder aus den berechneten Daten).
3. Ergänzen Sie die Prozedur `CountChars`, welche die absolute Häufigkeit der Zeichen im Text feststellt. Die Parameter der Prozedur lassen es auch zu, über einzelnen Caesar-Chiffren, in die der Vigenère bei bekannter Schlüssellänge zerfällt, getrennte Häufigkeitsanalysen zu fahren.
4. Bei einer angenommenen Schlüssellänge l zerfällt der Vigenère in l Caesarchiffren. Bestimmen Sie für ein festes l die Häufigkeiten der Zeichen der einzelnen Caesarchiffren und vergleichen Sie diese mit den Häufigkeiten in `PropTable`. Ist die Häufigkeitsverteilung der Caesarchiffre bis auf Verschiebung identisch mit `PropTable` (dazu soll die Korrelation zwischen der Häufigkeitsverteilung im Text und `PropTable` über die Verschiebung [Cäsar-Schlüssel] minimiert werden), so ist l wahrscheinlich die gesuchte Schlüssellänge und die Verschiebung ein Zeichen des Schlüssels.

5. Entschlüsseln Sie mit Hilfe des Vigenère-Programms und des herausgefundenen Schlüssels den Text und senden Sie ihn per Mail an Ihre Betreuer!

5.3. Auszug aus dem Skript Signale Codes und Chiffren II

5.3.1. Beispiel für eine Substitutionschiffre: Die Cäsar-Chiffre

Eine der ältesten Anwendungen von Substitutionschiffren geht auf Julius Cäsar zurück. Bei der sogenanten **Cäsar**-Chiffre wird jedes Zeichen des Klartextes um den Betrag des Schlüssels k im Alphabet verschoben. Für den Schlüssel $k = 3$ ergibt sich folgende Chiffrierabbildung E_k:

$$E_k : \begin{cases} A \to D \\ B \to E \\ C \to F \\ \quad \vdots \\ X \to A \\ Y \to B \\ Z \to C \end{cases}$$

Sei $n = |\mathbf{A}|$ die Anzahl der verschiedenen Zeichen des Alphabets $\mathbf{A}$. Dann kann ein Kryptoanalyst in maximal n Versuchen den Klartext zu einem Chiffretext bestimmen, d.h. auch diese Chiffre ist nicht sicher.

Eine weitere Möglichkeit die Cäsar-Chiffre zu knacken, besteht in der Betrachtung der Zeichenhäufigkeiten. In der deutschen Sprache ist z. B. das E der häufigste Buchstabe. Ist im Chiffrat K der häufigste Buchstabe, so wurde mit hoher Wahrscheinlichkeit der Schlüssel $k = 6$ verwendet.

Eine kleine Verbesserung ergibt sich, wenn man das Alphabet $\mathbf{A}$ auf den Ring $\mathbb{Z}_n = {}^{\mathbb{Z}}/_{n\mathbb{Z}}$ abbildet und neben der Addition eines Schlüssels die Multiplikation als zweite Ringoperation für die Chiffrierabbildung einführt:

$$E_{(\lambda,\alpha)} : I \to (\lambda \cdot I + \alpha) \bmod n$$

Dazu ein Beispiel für $k = (\lambda, \alpha) = (2, 1)$:

$$E_k : \begin{cases} A \to C \\ B \to E \\ C \to G \\ \quad \vdots \\ X \to W \\ Y \to Y \\ Z \to A \end{cases}$$

Für $\lambda = 1$ und den Schlüssel $k = (1, \alpha)$ ergibt sich die schon bekannte Cäsar-Chiffre. Leider bringt auch die Einführung des zweiten Schlüsselparameters keine wesentliche Erhöhung der Kryptokomplexität mit sich, da die oben beschriebene Häufigkeitsanalyse den Kryptoanalysten fast immer zum Ziel führt.

5.3.2. Beispiel für eine Substitutionschiffre: Die Vigenère-Chiffre

Eine Weiterentwicklung der Cäsar-Chiffre, die mehr Sicherheit bietet, ist die sogenannte **Vigenère-Chiffre**, benannt nach einem Franzosen des sechzehnten Jahrhunderts, Blaise de Vigenère. Der prinzipielle Unterschied zwischen diesen beiden Chiffren besteht darin, daß nicht ein konstanter Schlüssel zur Chiffrierung jedes einzelnen Zeichens verwendet wird, sondern eine — möglichst lange — Schlüsselfolge.

Ist $\underline{k} = (k_0, ..., k_{l-1})$ eine Schlüsselfolge der Länge l, so ist die Chiffrierabbildung der Vigenère-Chiffre gegeben durch:

$$E_{\underline{k}} \colon m_0...m_{r-1} \to t_{k_0}(m_0)...t_{k_{l-1}}(m_{l-1})t_{k_0}(m_l)...t_{k_{r-1} \ mod \ l}(m_{r-1})$$

$$\begin{aligned} \text{mit}: \quad & m_0...m_{r-1} \text{ Klartextfolge der Länge } r \\ & a_i \in \mathbf{A}: \ \forall i \in [1 \colon r-1] \text{ und} \\ & t_{k_j}(m_i) = m_i + k_j \ \ mod \ n \ (\text{Cäsar} - \text{Chiffrierung}) \end{aligned}$$

Der Weg über die Analyse der Häufigkeitsverteilung der Zeichen im Chiffretext (Aufstellen der Histogramme) führt hier nicht zum Ziel, da die Histogramme für große Werte von l verflachen, d.h. sich einander angleichen. Daher ist eine Vigenère-Chiffre wesentlich sicherer als eine Cäsar-Chiffre; sie wurde sogar bis Mitte des letzten Jahrhunderts für unbrechbar gehalten.

Als Beispiel, daß auch Vigenère-Chiffren relativ einfach zu brechen sind, diene folgender Klartext, der mit der Schlüsselfolge COVER chiffriert wird:

T H E M A T H E M A T I C A L I N T E L L I G E N C E R
C O V E R C O V E R C O V E R C O V E R C O V E R C O V

Chiffretext:

V V Z Q R V V Z Q R V W X E C K B O I C N W B I E E S M

Das sich wiederholende Muster VVZQR läßt ohne jegliche Information über den Klartext auf die – richtige! – Periode 5 schließen.

5.4. Bestimmung der Schlüssellänge mit Hilfe der Autokorrelationen

Eine Autokorrelation ist im allgemeinen wie folgt definiert:

$$K(d) = \int_x < f(x), f(x+d) > \mathrm{d}\, x \ .$$

Dabei ist $< \cdot, \cdot >$ eine Metrik auf dem Wertebereich der Funktion f.

Um die Periode einer Vigenère-Chiffre zu ermitteln, muß ein Extremum in der Autokorrelation des Chiffrats gefunden werden. Im Fall eines Textes kann als Metrik eine Funktion benutzt werden, die für gleiche Zeichen 0, sonst eine 1 ergibt.

Außerdem muß das Integral durch eine Summe ersetzt werden, da ein diskreter Definitionsbereich vorliegt (die Indizes der Zeichen).

Also ist die Autokorrelation für die Periode d definiert durch:

$$K(d) = \sum_{i=0}^{l-d} < a_i, a_{i+d} >$$

mit Länge des Textes l, der Text selber ist a_0 bis a_{l-1} und der oben definierten Metrik.

5.5. Bestimmung der einzelnen Schlüsselbuchstaben

Um die Schlüsselbuchstaben zu bestimmen, muß die im folgenden definierte Kreuzkorrelation zwischen der `PropTable` (gegebene Häufigkeiten) und den gemessenen Häufigkeiten der Buchstaben in der einzelnen Cäsar-Chiffre (verschoben um die Schlüssel) minimiert werden:

$$K(k) = \sum_{i=0}^{25} < P_i, H_{i+k \bmod 26} >$$

mit k als Schlüssel (1 entspricht `'A'`, ..., 25 entspricht `'Y'` und 0 entspricht `'Z'`). P_i sind die Häufigkeiten, die in `PropTable` vorgegeben sind, H_j sind die Häufigkeiten, die im Text (d.h. der einzelnen Cäsarchiffre) gemessen wurden.

Als Metrik kann zum Beispiel das Abstandsquadrat verwendet werden:

$$< a, b >= (a - b)^2 \ .$$

5.6. Rahmenprogramme

```
/************************************************************
**         Europäisches Institut für Systemsicherheit        *
**         Praktikum "Kryptoanalyse"                         *
**                                                           *
** Versuch 1: Klassische Chiffrierverfahren                  *
**                                                           *
************************************************************
**
** vigenere.c: Implementierung einer Vigenere-Chiffre
**/

#include <stdio.h>
#include <stdlib.h>
#include <praktikum.h>

/*********************  Globale Hilfsbvariabeln  ******************/
String Key;     /* Schlüssel */

/*
 * int Encipher(int c) : Interpretiert C als Zeichen, verschlüsselt es nach der
 *                       Methode von Vigenere und gibt das Ergebnis als Resultat
 *                       zurück.
 */

static int Encipher(int c)
  {
    /*>>>>                                  <<<<*
     *>>>> AUFGABE: Verschlüsseln von 'C' <<<<*
     *>>>>                                  <<<<*/
  }


/*
 * int Decipher(int c) : Interpretiert C als Zeichen, entschlüsselt es nach der
 *                       Methode von Vigenere und gibt das Ergebnis als Resultat
 *                       zurück.
 */

static int Decipher(int c)
  {
    /*>>>>                                  <<<<*
     *>>>> AUFGABE: Entschlüsseln von 'C' <<<<*
     *>>>>                                  <<<<*/
  }


/*
 * main(argc,argv) : Das Hauptprogramm, welches beim Aufruf von VIGENERE aufgerufen wird.
 *   ARGC ist die Anzahl der in der Kommandozeile angegebenen Argumente plus 1, ARGV ist
 *   ein Feld von Zeigern auf eben diese Argumente. ARGV[1] ist das erste usw.
 *   ARGV[0] enthält den Namen des Programms.
 */

int main(int argc, char **argv)
{
```

```
String infilename,outfilename,help,zeile;
int decipher;
/***** weitere (lokale) Hiflsbvariabeln *******/

FILE *infile,*outfile;

/* Wenn die Ein- bzw. Ausgabedatei oder der Schlüssel nicht in der
 * Kommandozeile angegeben wurden, fragen wir einfach nach .... */
if (argc<2) readline("Eingabefile : ",infilename,sizeof(infilename));
else strncpy(infilename,argv[1],sizeof(infilename));
if (argc<3) readline("Ausgabefile : ",outfilename,sizeof(outfilename));
else strncpy(outfilename,argv[2],sizeof(outfilename));
if (argc<4) readline("Schluessel  : ",Key,sizeof(Key));
else strncpy(Key,argv[3],sizeof(Key));

if (argc<5) {
  do {
    readline("V)er- oder E)ntschlüsseln : ",help,sizeof(help));
    string_to_upper(help);
  }
  while (strlen(help)!=1 && help[0]!='V' && help[0]!='E');
  decipher = help[0]=='E';
}
else {
  if (!strncmp(argv[4],"encipher",strlen(argv[4]))) decipher = 0;
  else if (!strncmp(argv[4],"decipher",strlen(argv[4]))) decipher = 1;
  else {
    fprintf(stderr,"FEHLER: Unbekannter Modus, 'encipher' oder 'decipher' erwartet.\n");
    exit(20);
  }
}
string_to_upper(Key);

/* Öffnen der Dateien:
 *  'fopen' gibt im Fehlerfall einen NULL-Pointer zurück. Kann die Datei
 *  geöffnet werden, so wird der von 'fopen' zurückgelieferte FILE-Pointer
 *  als Argument bei den Aufrufen 'fgets', 'fprintf', 'fclose' usw.
 *  zur Identifizierung der Datei angegeben.
 */
if (!(infile=fopen(infilename,"r"))) {
  fprintf(stderr,"FEHLER: Eingabefile %s kann nicht geöffnet werden: %s\n",infilename,strerror(errno)
  exit(20);
}
if (!(outfile=fopen(outfilename,"w"))) {
  fprintf(stderr,"FEHLER: Ausgabefile %s kann nicht geöffnet werden: %s\n",outfilename,strerror(errno
  exit(20);
}

/* Belegung der Variablen:
 *  infilename : Name der Eingabedatei
 * outfilename : Name der Ausgabedatei
 *      infile : 'Datei-Bezeichner', der die Eingabedatei repräsentiert.
 *     outfile : 'Datei-Bezeichner', der die Ausgabedatei repräsentiert.
 *         Key : Schlüssel, nach Großschrift gewandelt
 *    decipher : Flag, == 1 im Entschlüsselungsmodus, ansonsten 0.
 */
```

```
  do {
    fgets(zeile,sizeof(zeile),infile);
    if (!feof(infile)) {
      strip_crlf(zeile);
      string_to_upper(zeile);
      /*>>>>                                              <<<<*
       *>>>> AUFGABE: Vigenere-Verschlüsseln von ZEILE <<<<*
       *>>>>                                              <<<<*/
      fprintf(outfile,"%s\n",zeile);
    }
  }
  while (!feof(infile));

  /* Schließen der Ein- und Ausgabedateien */
  fclose(infile);
  fclose(outfile);

  return 0;
}
```

```
/************************************************************
**         Europäisches Institut für Systemsicherheit        *
**         Praktikum "Kryptoanalyse"                         *
**                                                           *
** Versuch 1: Klassische Chiffrierverfahren                  *
**                                                           *
************************************************************
**
** vigenere_attacke.c: Brechen der Vigenere-Chiffre
**/

#include <stdio.h>
#include <stdlib.h>
#include <praktikum.h>

#define GNUPLOT_CMD_FILENAME "gnuplot.in.cmd" /* Name fuer das erzeugte
            gnuplot-Kommandofile */
#define GNUPLOT_DATA_FILENAME "gnuplot.in.data" /* Name fuer Datenfile */

#define NUMCHARS    26       /* Anzahl der Zeichen, die betrachtet werden ('A' .. 'Z') */
#define MaxFileLen  32768    /* Maximale Größe des zu entschlüsselnden Textes */
#define MAXPERIOD   200      /* Maximale Periodenl"ange, f"ur die die
        Autokorrelation berechnet wird */

const char *StatisticFileName = "statistik.data";  /* Filename der Wahrscheinlichkeitstabelle */
const char *WorkFile          = "testtext.ciph";   /* Filename des verschlüsselten Textes */

double PropTable[NUMCHARS]; /* Tabellke mit den Zeichenwahrscheinlichkeiten.
           * ProbTable[0] == 'A', PropTable[1] == 'B' usw. */
char TextArray[MaxFileLen]; /* die eingelesene Datei */
int TextLength;             /* Anzahl der gültigen Zeichen in TextArray */

double AutoCor[MAXPERIOD+1]; /* Normierte Autokorrelationen */
int Period;                  /* berechnete Periodenlaenge */

/*---------------------------------------------------------------------------*/

/*
 * GetStatisticTable(): Liest die Statistik-Tabelle aus dem File
 * STATISTICFILENAME in das globale Array PROPTABLE ein.
 */

static void GetStatisticTable(void)
  {
    FILE *inp;
    int i;
    char line[64];

    if (!(inp=fopen(StatisticFileName,"r"))) {
      fprintf(stderr,"FEHLER: File %s kann nicht geöffnet werden: %s\n",
        StatisticFileName,strerror(errno));
      exit(20);
    }

    for (i=0; i<TABSIZE(PropTable); i++) {
      fgets(line,sizeof(line),inp);
      if (feof(inp)) {
```

```
        fprintf(stderr,"FEHLER: Unerwartetes Dateieine in %s nach %d Einträgen.\n",
    StatisticFileName,i);
  exit(20);
      }
      PropTable[i] = atof(line);
    }
    fclose(inp);
  }

/*--------------------------------------------------------------------------*/

/* GetFile(void) : Ließt den verschlüsselten Text aus dem File
 *   WORKFILE zeichenweise in das globale Array TEXTARRAY ein und zählt
 *   TEXTLENGTH für jedes Zeichen um 1 hoch.
 *   Eingelesen werden nur Buchstaben. Satz- und Sonderzeichen werden weggeworfen,
 *   Kleinbuchstaben werden beim Einlesen in Großbuchstaben gewandelt.
 */

static void GetFile(void)
  {
    FILE *inp;
    char c;

    if (!(inp=fopen(WorkFile,"r"))) {
      fprintf(stderr,"FEHLER: File %s kann nicht geöffnet werden: %s\n",
        WorkFile,strerror(errno));
      exit(20);
    }

    TextLength=0;
    while (!feof(inp)) {
      c = fgetc(inp);
      if (feof(inp)) break;
      if (c>='a' && c<='z') c -= 32;
      if (c>='A' && c<='Z') {
  if (TextLength >= sizeof(TextArray)) {
    fprintf(stderr,"FEHLER: Eingabepuffer nach %d Zeichen übergelaufen!\n",TextLength);
    exit(20);
  }
        TextArray[TextLength++] = c;
      }
    }
    fclose(inp);
  }


/*--------------------------------------------------------------------------*/

/*
 * CountChars( int start, int offset, int h[] )
 *
 * CountChars zählt die Zeichen (nur Buchstaben!) im globalen Feld
 * TEXTARRAY. START gibt an, bei welchen Zeichen (Offset vom Begin der
 * Tabelle) die Zählung beginnen soll und OFFSET ist die Anzahl der
 * Zeichen, die nach dem 'Zählen' eines Zeichens weitergeschaltet
 * werden soll. 'A' wird in h[0], 'B' in h[1] usw. gezählt.
 *
```

```
 *  Beispiel:  OFFSET==3, START==1 --> 1,  4,  7,  10, ....
 *             OFFSET==5, START==3 --> 3,  8, 13,  18, ....
 *
 * Man beachte, daß das erste Zeichen eines C-Strings den Offset 0 besitzt!
 */

static void CountChars( int start, int offset, int h[NUMCHARS])
  {
    int i;
    char c;

    for (i=0; i<NUMCHARS; i++) h[i] = 0;

    /*****************  Aufgabe Teil 3 *****************/
  }

/*
 * AutoCorrelation (int d)
 *
 * AutoCorrelation berechnet die Autokorrelation im Text mit der Verschiebung
 * (Periode) d.
 *
 * Als Metrik soll die Funktion eingesetzt werden, die bei gleichen Zeichen
 * 0, sonst 1 ergibt. Die Autokorrelation muss hier *nicht* normiert werden.
 * dies geschieht unten in main() im Rahmenprogramm.
 *
 * Der Text steht im Feld TextArray und enthaelt TextLength Zeichen.
 *
 * Das Ergebnis soll als Returnwert zur"uckgegeben werden.
 */

static double AutoCorrelation (int d)
{
  /*************** Aufgabe Teil 1 ******************/
}

/*
 * CalcPeriod ()
 *
 * Berechnet (oder liest vom Benutzer ein) die Periode der Chiffre.
 * Das Ergebnis soll in der globalen Variable Period gespeichert werden.
 * Zum Beispiel kann dazu das Array AutoCor, das die vorher berechneten
 * Autokorrelationen (normiert!) enth"alt.
 */

static void CalcPeriod (void)
{
  /*************** Aufgabe Teil 2 ******************/
}

/*--------------------------------------------------------------------------*/

int main(int argc, char **argv)
{

  GetStatisticTable();     /* Wahrscheinlichkeiten einlesen */
  GetFile();               /* zu bearbeitendes File einlesen */
```

```
  {
    int i;
    for (i = 0; i <= MAXPERIOD; i++) {
      AutoCor [i] = (double) AutoCorrelation (i) / (TextLength - i);
    }
  }

  /* Now prepare gnuplot */
  {
    FILE *f;
    int i;

    f = fopen (GNUPLOT_CMD_FILENAME, "w");
    if (! f) {
      perror ("Error creating file " GNUPLOT_CMD_FILENAME);
      exit (2);
    }
    fprintf (f, "set print \"-\"\n");             // make gnuplot print to stdout instead of stderr
    fprintf (f, "plot [1:%d] \"%s\" using 0:1 with lines\n", MAXPERIOD,
       GNUPLOT_DATA_FILENAME);
    fprintf (f, "print \"Bitte Return druecken...\"\npause -1\n");
    fclose (f);
    f = fopen (GNUPLOT_DATA_FILENAME, "w");
    if (! f) {
      perror ("Error creating file " GNUPLOT_DATA_FILENAME);
      exit (2);
    }
    for (i = 0; i <= MAXPERIOD; i++) {
      fprintf (f, "%f\n", AutoCor[i]);
    }
    fclose (f);
  }

  /* Now call it */

  system ("gnuplot " GNUPLOT_CMD_FILENAME);

  CalcPeriod ();

  /*****************  Aufgabe 4 *****************/

  return 0;
}
```

6. Padding Oracle

6.1. PKCS7 Padding

Um Nachrichten beliebiger Länge mit einem Block-Cipher zu verschlüsseln muss die Nachricht zuerst auf ein Vielfaches der Blockgröße gepaddet werden. In der Praxis verwendet man dazu oft PKCS7 Padding. Dabei wird die Nachricht um 1 bis Blockgröße Bytes verlängert die alle als Wert die Anzahl der Padding-Bytes haben. Ein paar Beispiele:

```
     .. .. .. .. 30
 ->  .. .. .. .. 30 01

     .. .. .. 30
 ->  .. .. .. 30 02 02

     .. .. 30
 ->  .. .. 30 03 03 03

     .. 30
 ->  .. 30 04 04 04 04
```

6.2. Padding Oracles

Beim Entfernen eines Paddings von nicht vertrauenswürdigen Daten muss ein Server vorsichtig sein, dass er nicht verrät, ob das Padding ungültig war oder nur die Nachricht. Im schlimmsten Fall antwortet der Server mit 2 verschiedenen Fehlermeldungen abhängig davon, ob das Padding ungültig war oder nicht. Ein Timing-basierter Angriff ist aber auch möglich, wenn der Server identische Fehlermeldungen ausgibt, aber unterschiedlich lange zur Bearbeitung der Anfrage braucht.

In diesem Versuch wollen wir den einfacheren Fall eines Padding Oracles betrachten, bei dem der Server direkt antwortet ob das Padding gültig war.

6.3. CBC

Für diesen Versuch ist wichtig, dass als Blockmodus für die Verschlüsselung CBC benutzt wird.

6.4. Aufgabenteil 1

Der Dämon, der in dieser Aufgabe verwendet werden soll, zieht bei Verbindungsaufbau einen zufälligen AES Schlüssel.

Im ersten Teil der Aufgabe stellt der Dämon ein AES-CBC Chiffrat (in `challenge`) zur Verfügung. Es besteht aus 3 Blöcken. Der erste Block ist dabei der zufällige IV. Es wird also ein String einer Länge 16 - 31 erst PKCS7 gepaddet auf 32 Bytes und diese dann mit AES-CBC mit einem zufälligen Schlüssel verschlüsselt.

Die erste Aufgabe besteht nun daraus, den zugehörigen Klartext zu bestimmen. Dafür steht ein Padding Oracle in der Funktion `padding_oracle` zur Verfügung. Dieser Funktion wird ein Chiffrat übergeben und

eine Chiffratlänge in Blöcken (2 oder 3) und sie antwortet mit 1, wenn das Padding gültig war und mit 0 wenn es ungültig war.

6.5. Aufgabenteil 2

Im zweiten Teil muss zu einem vorgegebenen Text (in Variable `sol_str`) ein gültiges Chiffrat erzeugt werden. Dieses Chiffrat wird dann an den Dämon gesendet, dieser entschüsselt es und überprüft den Inhalt. Wenn der Text nicht korrekt ist, meldet der Dämon den enschlüselten Text, um die Fehlersuche zu vereinfachen. Eine solche Lösung kann pro gezogenem AES Schlüssel nur genau einmal getestet werden. Danach muss das Programm neugestartet werden (oder eine neue Verbindung aufgebaut werden). Bevor die Lösung eingereicht wird kann das Padding Oracle beliebig oft benutzt werden.

6.6. Hinweise

- zu Aufgabenteil 1: Für diesen Versuch ist es wichtig, dass CBC als Betriebsmodus gewählt wird. Überlegen Sie sich, was passiert, wenn in einem CBC-verschlüsselten Ciphertext bits kippen.
- zu Aufgabenteil 2: Diese Aufgabe baut auf der ersten auf. Um einen selbst gewählten Plaintext verschlüsseln zu können hilft die Fähigkeit einzelne Blöcke entschlüsseln zu können. Wie können Sie mit einer solchen Fähigkeit CBC-Chiffrate bearbeiten oder erzeugen?

7. Meet in the middle-Attack

7.1. Einleitung

In diesem Versuch wird eine Chiffre mit 60 Bits großem Schlüssel gebrochen, und zwar durch vollständige Schlüsselsuche bei bekanntem Klartext. Dabei wird als Schwäche ausgenutzt, daß die Chiffre aus zwei (identischen) Teilchiffren mit jeweils 30 Bits langem Teilschlüssel (die beiden Teilschlüssel sind natürlich unabhängig voneinander gewählt) besteht. Dadurch ist ein Time-Memory-Tradeoff möglich. Es wird erheblich mehr Speicher eingesetzt, dafür benötigt die Attacke wesentlich weniger Zeit, nämlich in der Größenordnung 2^{30} statt 2^{60}.

7.2. Die Chiffre

Bei der Chiffre (d.h. den Teilchiffren) handelt es sich um eine für diesen Versuch entworfene Chiffre mit reduziertem Schlüsselraum, die dafür in Software schnell zu implementieren ist. Da wir nicht auf eine Attacke gegen innere Schwächen des Algorithmus hinaus wollen, wird die Chiffre nur im Objektcode zur Verfügung gestellt.

Die Chiffre operiert auf 64 Bits großen Blöcken und kann 32 Bits Schlüssel verarbeiten. Um den Rechenzeitaufwand in Grenzen zu halten, wird der Schlüsselraum dadurch auf 30 Bits eingeschränkt, daß die höchstwertigen 2 Bits des Schlüssels garantiert 0 sind.

7.3. Aufgabe

In der Versuchsumgebung werden die Dateien `plaintext` (ein 64-Bit Klartextblock in Hexadezimalschreibweise) und `ciphertext` (das zugehörige Chiffrat nach Verschlüsselung mit den zwei Teilschlüsseln) bereitgestellt. In dem Rahmenprogramm `attack.c` soll die meet-in-the-middle-Attacke implementiert werden (die Methode ist weiter unten beschrieben). Wenn mehrere Schlüsselkombinationen als Lösung möglich sind, sollen alle ausgegeben (und als Lösung an die Betreuer gemailt) werden.

7.4. Meet-in-the-middle-Attacke

Bei der Meet-in-the-middle-Attacke handelt es sich um eine Attacke, wo der Speicherbedarf erhöht wird, um dafür (erheblich) Rechenzeit einzusparen.

Die einfachste Variante ist, eine Tabelle aufzustellen, in der alle möglichen ersten (linken) Teilschlüssel, zusammen mit dem Block, der entsteht, wenn man den Klartextblock mit dem jeweils betrachteten Schlüssel verschlüsselt, enthalten sind.

Diese Tabelle wird dann nach den (halb) verschlüsselten Blöcken sortiert. Dann werden von rechts alle Schlüssel durchprobiert, indem man den Chiffretextblock mit dem Schlüssel entschlüsselt und den entstehenden Block in der vorher aufgestellten Tabelle sucht (da die Tabelle sortiert ist, geht dies recht schnell durch binäre Suche). Wenn man den Block findet, ist eine Lösung gefunden: der erste Teilschlüssel steht in der Tabelle bei dem gefundenen Block, der zweite in der Laufvariable, in der der zweite Teilschlüssel iteriert wird.

Nachteil dieser Methode ist der hohe Speicheraufwand: in unserem Beispiel wäre der Speicherbedarf $(8+4)\cdot 2^{30}$ Bytes, also etwa 12 Gigabyte.

Auf den Maschinen, auf denen der Versuch durchgeführt werden soll, steht aber nicht so viel Speicher zur Verfügung (nur etwa ein Viertel oder ein Sechstel). Also muß eine Lösung mit weniger Speicheraufwand gefunden werden.

Diese sieht wie folgt aus:

1. Zuerst wird über alle „linken“ Teilschlüssel iteriert. Der Klartextblock wird verschlüsselt. Von dem Ergebnis wird ein Hashwert berechnet und das Bit mit der entsprechenden Nummer in einem Bitarray gesetzt.
2. Jetzt wird über alle „rechten“ Teilschlüssel iteriert. Der Chiffretextblock wird mit dem Teilschlüssel entschlüsselt. Es wird wieder der Hashwert berechnet. Wenn das zugehörige Bit im ersten Schritt gesetzt wurde, ist es möglich, daß der betrachtete Teilschlüssel Bestandteil einer Lösung ist. In diesem Fall wird der betrachtete rechte Teilschlüssel zusammen mit dem damit (teilweise) entschlüsselten Chiffrat in eine Liste eingetragen. Wenn das Bit in dem Bitarray nicht gesetzt ist, kann der betrachtete rechte Teilschlüssel auf keinen Fall zu einer Lösung führen.
3. Nun wird die im zweiten Schritt entstandene Liste nach den gespeicherten Blöcken sortiert, um eine binäre Suche nach den Blöcken zu ermöglichen.
4. Als letztes wird noch einmal über alle möglichen linken Teilschlüssel iteriert. Der Klartextblock wird damit verschlüsselt und das Ergebnis in der Liste gesucht. Wenn der Block gefunden wird, ist der betrachtete linke Teilschlüssel, zusammen mit dem in der Liste enthaltenen rechten Teilschlüssel, eine Lösung.

Analyse des Speicherbedarfs dieser Methode:

Benötigt wird zum einen der Speicher für das Bitarray. Die Größe ist normalerweise eine Zweierpotenz, damit als Hashfunktion einfach eine Auswahl von Bits des Blocks benutzt werden kann.

Sei die Anzahl der Bits, die als Hashwert benutzt werden, h, und die Anzahl der Bits in einem Teilschlüssel k. Dann sind, wenn die Chiffre sich statistisch optimal verhält, am Ende 2^k Bits in dem Array gesetzt. (Wenn k kleiner als h ist... Wenn k größer/gleich h ist, sind wahrscheinlich fast alle Bits gesetzt und diese Methode bringt keinen Gewinn gegenüber der einfachen Methode oben.)

Im zweiten Schritt ist also die Wahrscheinlichkeit, daß bei einem Schlüssel ein gesetztes Bit getroffen wird, etwa $2^{-(h-k)}$. Damit ist der Erwartungswert für die Anzahl der Einträge in der im zweiten Schritt erzeugten Liste $2^k \cdot 2^{-(h-k)} = 2^{2k-h}$.

Damit ergibt sich bei $k = 30$ folgende Tabelle für den Speicherbedarf (für die Kollisions-Liste werden 12 Bytes pro Eintrag belegt):

h	Bitarray	Kollisions-Liste	Summe
34	2,0 GB	0,8 GB	2,8 GB
33	1,0 GB	1,5 GB	2,5 GB
32	0,5 GB	3,0 GB	3,5 GB
31	0,3 GB	6,0 GB	6,3 GB

Auf Kosten von noch etwas zusätzlicher Rechenzeit kann der Speicherbedarf noch weiter optimiert werden:

Das Bitarray wird in n (normalerweise eine Zweierpotenz) gleichgroße Teile zerlegt. Die Schritte 1 und 2 müssen für jeden Teil des Bitarrays getrennt durchlaufen werden (d.h. der Zeitaufwand für die Schritte 1 und 2 ver-n-facht sich, dafür verringert sich der Speicheraufwand für das Bitarray auf ein n-tel). Dabei wird nur dann das Bit gesetzt, wenn beim Durchlauf i der Hash-Wert im i-ten n-tel des Wertebereichs der Hashfunktion liegt. Genauso wird beim i-ten Durchlauf von Schritt 2 nur dann getestet, ob das entsprechende Bit gesetzt ist, wenn der Hash-Wert im i-ten n-tel des Wertebereichs liegt.

Sicherheitshalber sollte außerdem für die Liste der Matches, die im Schritt 2 erzeugt wird, mehr Speicherplatz reserviert werden, als der Erwartungswert angibt, um für den Fall statistischer Abweichungen vorzusorgen.

7.5. Versuchsumgebung

Dieser Versuch läuft (mit geschickter Wahl der Parameter) auf Maschinen mit 2 GB Hauptspeicher. Die Rechenzeit bewegt sich dann in der Größenordnung von 45 Minuten. Um nicht bei jedem Test diese lange Wartezeit zu haben, sollte man eigene Klartext/Chiffrat Paare erzeugen, bei denen nur 22 bis 26 Bit Schlüssel durchsuch werden müsst. Ein Paar mit 26 Bit wird mit der Endung ".26" zur Verfügung gestellt. Wenn der Algorithmus mit diesen Paaren zurecht kommt kann man die richtige Aufgabe angehen. Um zu überprüfen, ob der Speicherbedarf nicht zu groß ist oder ob ein anderer Benutzer auf demselben Rechner eine größere Berechnung durchführt ist das Kommando `top` recht hilfreich.

Zum Testen und Erzeugen von eigenen Chiffraten gibt es auch noch das Programm `ssc`, mit dem einfach ein Block mit einer Teilchiffre ver- oder entschlüsselt werden kann. (**Achtung:** Beim Verwenden von eigenen Klartext/Chiffrat-Paaren sollten die Original-Files `plaintext` und `ciphertext` kopiert werden, damit man sie am Ende wieder herstellen kann! Dann kann man die Daten für den eigenen Versuch unter `plaintext` und `ciphertext` speichern und die Attacke starten.)

8. Kocher Timing Attack

8.1. Einleitung

Viele Kryptosysteme benötigen unterschiedlich viel Zeit, um unterschiedliche Eingaben und/oder Schlüssel zu verarbeiten. Dies kann zu einer Attacke verwendet werden, die im Dezember 1995 öffentlich vorgestellt wurde. (Siehe auch das Papier von Paul C. Kocher, der diese Attacke vorgestellt hat, unter `http://www.cryptography.com/timingattack.html`, auch zu finden unter `file:/mnt/eden/users/felix/timingattack.html.gz`).

Unter anderem kann eine Exponentierung $z = x^y \pmod{n}$ angegriffen werden. Im Versuch ist n bekannt, y zufällig ausgewählt, x kann gewählt und z beobachtet werden. Die Zeitmessung wird durch simulierte Timings ersetzt, damit sich genaue und reproduzierbare Resultate ergeben.

Die Anzahl der Bits des Exponenten ist bekannt; der Exponent ist gleich groß wie der Modulus.

8.2. Versuchsumgebung

Es wird ein Dämon bereitgestellt, der die Funktion der angegriffenen Seite übernimmt.

In der Versuchsumgebung gibt es den Source `daem_acc.c`, der Routinen zum Zugriff auf diesen Dämonen anbietet.

Es können beliebig viele Werte mit einem zufälligen, aber festen Exponenten potenziert werden (`exp_daemon`). Danach soll der Exponent ermittelt und dem Dämonen mit `key_daemon` mitgeteilt werden. Diese Funktion gibt den tatsächlich verwendeten Exponenten zurück, sowie ein Flag, ob der Exponent richtig ermittelt wurde. Wenn danach noch Requests an den Dämonen geschickt werden, wird ein neuer Exponent generiert.

Die Funktion `exp_daemon` gibt außerdem noch als Return-Wert den Zeitverbrauch dieser Potenzoperation zurück.

Es stehen außerdem die Funktionen `LITTimeModExp`, `LITTimeModMult` und `LITTimeModSquare` zur Verfügung, um den Zeitverbrauch dieser einzelnen Operationen zu ermitteln. Es kann angenommen werden, daß `LITModExp` (die Potenzoperation, die auch von dem Dämonen verwendet wird) nur die Zeit für die Multiplikationen und die Quadrierungen verwendet, d.h. für die Schleifen, Zuweisungen und ähnliches wird keine Zeit berechnet.

Der Dämon benutzt für die Exponentiation die Funktion `LITModExp`, die in dem File `texp.c` definiert ist. Es handelt sich um eine Variante des einfachen square-and-multiply-Algorithmus.

8.3. Ablauf der Attacke

Der Algorithmus für die Exponentiation ist wie folgt:

Eingabe: x, y, m
Ausgabe: $z := x^y \pmod m$, Berechneter (simulierter) Zeitaufwand

$x_0 \leftarrow x$
$z_0 \leftarrow 1$
for $i \leftarrow 0$ **to** bits(y)-1
  if bit_gesetzt(i,y)
    $z_{i+1} \leftarrow z_i \cdot x_i$
  else
    $z_{i+1} \leftarrow z_i$
  $x_{i+1} \leftarrow x_i^2$
$z \leftarrow z_{\text{bits}(y)}$

Zuerst sollte das Hamming-Gewicht des Exponenten (d.h. die Anzahl der gesetzten Bits, damit also die Anzahl der Multiplikationen) ermittelt werden. Dazu kann als Basis die Zahl 1 gewählt werden. Dann können die Quadrierungen abgezogen werden, da dort keinerlei unbekannte Operanden Einfluß nehmen. Außerdem ist der Zeitverbrauch der Multiplikationen bekannt, da diese immer die Multiplikation $1 \cdot 1$ sind. Damit kann das Hamming-Gewicht berechnet werden, in dem die Zeit, die die Multiplikationen benötigen, durch die Zeit für eine Multiplikation geteilt wird.

Anschließend sollen die Bits des Exponenten, vom niederwertigsten Bit ausgehend, berechnet werden.

Dazu sollen einige Proben durch den Dämonen verschlüsselt werden. Anschließend kann man von den erhaltenen Timing-Werten wieder die Quadrierungen abziehen, da die Werte, die quadriert werden, bekannt sind. Übrig bleiben also Proben und die Timing-Werte für die Multiplikationen.

Jetzt wird für alle Proben bestimmt, wie lange die Multiplikation $z_0 \cdot x_0$ dauert, wenn sie ausgeführt wird (d.h. wenn das niederwertigste Bit des Exponenten 1 ist). Wenn diese Werte mit der Gesamtdauer der Exponentiation für die jeweiligen Proben korrelieren, ist es wahrscheinlich, daß das niederwertigste Bit des Exponenten 1 ist. Wenn keine Korrelation vorhanden ist, ist das Bit wahrscheinlich 0.

Eine Möglichkeit, diese Korrelation zu überprüfen, ist, von dem Gesamttiming der Multiplikationen (T_M) zwei Werte zu berechnen: $T_1 = T_M - (\text{TMult}(z_i, x_i) + (w-1) \cdot E\,\text{TMult})$ und $T_0 = T_M - (w \cdot E\,\text{TMult})$. Hierbei ist w das Hamming-Gewicht des Exponenten, i das betrachtete Bit des Exponenten (also zuerst 0), TMult die Zeit, die für eine Multiplikation (der angegebenen Werte) benötigt wird, E bedeute den Erwartungswert. T_0 gibt an, wie weit das gemessene Timing davon abweicht, was man erwarten kann, wenn das i-te Bit des Exponenten 0 ist, T_1 ist dasselbe, nur für ein gesetztes i-tes Bit im Exponenten. Wenn die Summe der T_1 über alle Stichproben kleiner ist als die Summe der T_0, ist es wahrscheinlicher, daß das i-te Bit des Exponenten 1 ist.

Die weiteren Bits findet man, indem man die obige Methode anwendet, allerdings vorher von T_M das tatsächliche Timing für die Multiplikation $z_{i-1} \cdot x_{i-1}$, und von w 1 abzieht, falls sich für das $i-1$-te Bit des Exponenten eine 1 ergeben hat. (Dies entspricht also dem Ermitteln des 0-ten Bit des Exponenten in der Exponentiation $x'^{y'}$ mit $x' = x^{2^i}$ und $y' = y \,\mathbf{div}\, 2^i$.)

Auf diese Weise kann sukzessive der vollständige Exponent y ermittelt werden. Wenn in irgendeinem Schritt dazwischen ein Bit des Exponenten falsch ermittelt wird, verwischen normalerweise alle folgenden Korrelationen, was eventuell für ein Backtracking verwendet werden kann. (Es ist allerdings auch eine Lösung möglich, die auch ohne Backtracking meistens die richtige Lösung ermittelt. Bei einer solchen Lösung genügt es, bei einem Fehlversuch noch eine weitere Attacke zu versuchen.)

Wenn mit `key_daemon` der richtige Exponent überprüft wird, wird dies geloggt. Daher ist es bei diesem Versuch nicht nötig, den Betreuern eine Lösung zu mailen.

9. Lineare Kryptoanalyse

9.1. Einführung

hbt In diesem Versuch soll die Blockchiffre FEAL (*Fast data Encipherment Algorithm*) mit 4 Runden gebrochen werden.

Dazu gibt es wieder einen Dämonen, der die angegriffene Seite darstellt. Im Rahmenprogramm können mit `feal_encrypt` bis zu 25 ausgewählte Klartexte vom Dämonen verschlüsselt werden. Die Chiffrate werden zurückgegeben. Anschließend sollten in `key[0]` bis `key[3]` die verwendeten Teilschlüssel der entsprechenden Runden berechnet werden. Das Rahmenprogramm meldet die berechneten Schlüssel dem Dämonen, der sie auf Richtigkeit überprüft.

Bei der Attacke soll die Methode der linearen Kryptoanalyse verwendet werden. Dabei werden einige Runden des Algorithmus (z.B. die ersten 3 Runden) linear approximiert. Dann wird für den Teilschlüssel der letzten Runde vollständig gesucht nach Kandidaten, bei denen die linearen Approximationen korrekt auftreten. Danach wird das Chiffrat mit den Kandidaten für den letzten Teilschlüssel teilweise (eine Runde) entschlüsselt. Dann kann auf ähnliche Weise der vorletzte Teilschlüssel gesucht werden, usw.

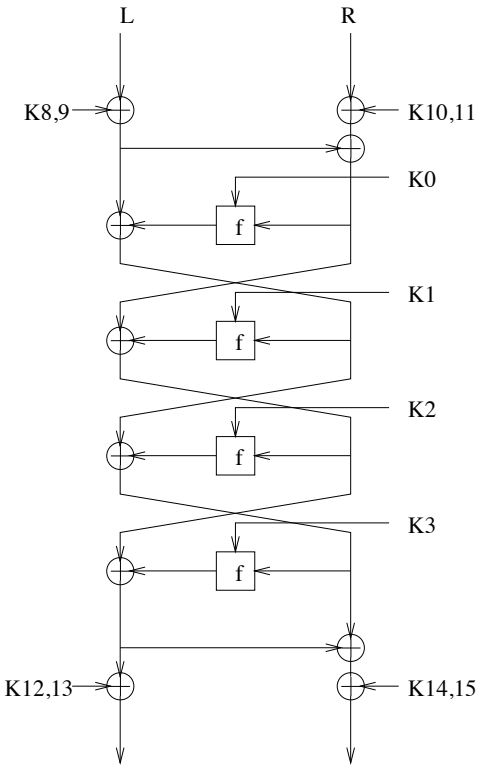

Abbildung 9.1.: Aufbau des FEAL-4

9.2. Der FEAL im Überblick

Die Blockchiffre FEAL (*Fast data Encipherment Algorithm*) ist leicht in Software realisierbar. Abbildung 9.1 zeigt den groben Aufbau des FEALs-4 („4“ steht dabei für die Anzahl der Runden, d.h. Verknüpfungen mit

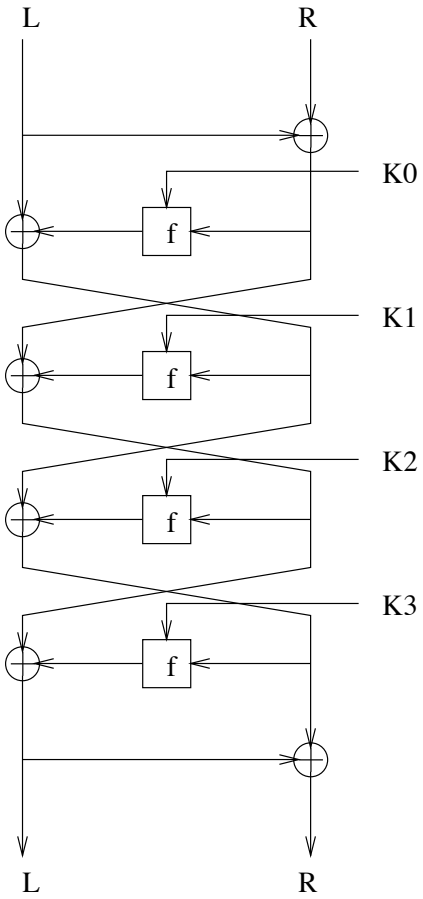

Abbildung 9.2.: Aufbau des vereinfachten Algorithmus'

der Funktion f). Hierbei werden 64 Bit Eingabedaten in 64 Bit Ausgabedaten verschlüsselt. K_i sind 16 Bit lange Teilschlüssel, die zuvor von der Schlüsselaufbereitung aus dem eigentlichen Schlüssel erzeugt wurden. (Die Schlüsselaufbereitung soll in diesem Versuch nicht interessieren.)

Für die Attacke wurden einige Teilschlüssel weggelassen. Allerdings können diese im Prinzip ähnlich gefunden werden, wie die in diesem Versuch zu bestimmenden Teilschlüssel.

Der vereinfachte Algorithmus ist in Abbildung 9.2 skizziert.

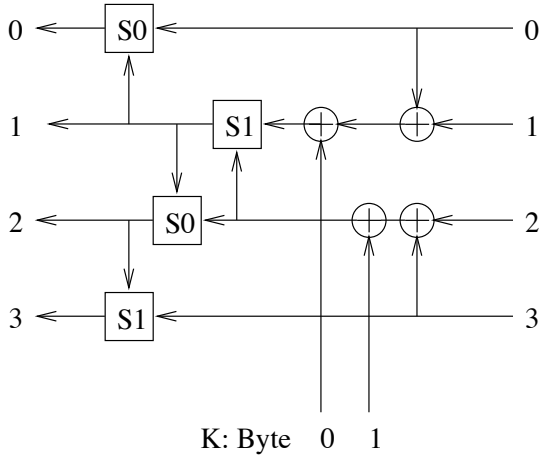

Abbildung 9.3.: Die F-Funktion

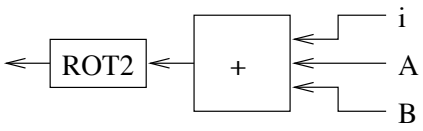

Abbildung 9.4.: Die Funktion $S_i(A, B)$

Die Funktion f ist aus den Bausteinen XOR-Verknüpfung und S_i zusammengesetzt (vergl. Abbildung 9.3 und 9.4). $S_i(a, b) := rot2((a + b + i) \ mod \ 256)$. $rot2(x)$ rotiert das Byte x um zwei Bits nach links. (Anmerkung: In Abb. 9.1, 9.2 und 9.3 bezeichnet $\bigoplus$ die bitweise exklusive Oder Verknüpfung.)

9.3. Lineare Analyse

Bei der linearen Kryptoanalyse werden Teile einer Chiffre mit Hilfe sogenannter linearer Charakteristiken angenähert. Eine lineare Charakteristik gibt an, daß die Parität eine bestimmten Menge von Eingabebits und Ausgabebits mit einer bestimmten Wahrscheinlichkeit p gerade ist.

In dieser Dokumentation werden Bits wie folgt numeriert: In einem Byte ist 0 das niederwertigste, 7 das höchstwertige Bit. Bei Worten, die aus mehr als einem Byte bestehen, werden die Bits im ersten Byte mit 0–7, im zweiten Byte mit 8–15 usw. numeriert.

Wenn wir z.B. die Funktion $S = S_0(A, B)$ (siehe Abbildung 9.4) betrachten, stellen wir fest, daß die Parität der Eingabebits A[0], B[0] und des Ausgabebits S[2] immer gerade ist.

Oder bei $S = S_1(A, B)$ ist die Parität der selben Bits (A[0], B[0], S[2]) immer ungerade.

Die genannten zwei Beziehungen können verwendet werden, um lineare Approximationen für die gesamte Funktion $F := f(B, k)$ (siehe Abbildung 9.3) anzugeben:

Für S_0 oben (Byteposition 0): $F[2] = B[0] + F[8]$ (d.h. gerade Parität zwischen den drei genannten Bits)
Für S_1 an Byteposition 1: $F[10] = 1 + B[0] + B[8] + K[0] + B[16] + B[24] + K[8]$ (d.h. ungerade Parität zwischen den genannten Bits. Hier geht auch der Schlüssel mit ein!)
Für S_0 an Byteposition 2: $F[18] = F[8] + B[16] + B[24] + K[8]$
Für S_1 an Byteposition 3: $F[26] = 1 + F[16] + B[24]$.

Diese 4 Charakteristiken für f können zu Charakteristiken für die ersten 3 Runden erweitert werden. Ein Beispiel (mit der zweiten oben angegebenen Charakteristik für f) ist in Abbildung 9.5 zu sehen.

Da alle beteiligten Charakteristiken deterministisch (d.h. Parität gerade entweder mit Wahrscheinlichkeit 0 oder 1) sind, ist die entstehende Charakteristik auch deterministisch.

D.h. die Parität der Bits 0,8,10,16,24 der linken Klartexthälfte, der Bits 0,8,16,24 der rechten Klartexthälfte, des Bit 10 der linken Hälfte der Ausgabe der 3. Runde und der Bits 0,8,16,24 der rechten Hälfte der Ausgabe der dritten Runde ist konstant.

Man sieht, daß bei einer Verzweigung im Datenfluß ein Bit nur in einer von beiden Richtungen weiterverfolgt wird, während bei einem XOR das Bit in beide Richtungen verfolgt werden muß.

Dies kann genutzt werden, um jetzt eine vollständige Suche nach dem Teilschlüssel K3 durchzuführen:

Für alle Möglichkeiten für K3 werden die gesammelten Chiffrate teilweise entschlüsselt. Dann wird überprüft, ob für alle Paare (Klartext, teilweise entschlüsseltes Chiffrat) die angegebene Parität gleich ist. Wenn nein, kann dieser Wert für K3 ausgeschlossen werden. Wenn ja, ist ein Kandidat für K3 gefunden (der mit Hilfe der anderen 3 deterministischen Charakteristiken genauer überprüft werden kann).

Wenn nach der Anwendung aller Charakteristiken für die ersten drei Runden einige Kandidaten für K3 übrig sind, kann eine ähnliche Attacke gegen K0 erfolgen: Nun werden die letzten drei Runden linear approximiert und für K0 eine vollständige Suche durchgeführt werden.

K1 und K2 können dann aus den teilweise verschlüsselten Klartexten und den teilweise entschlüsselten Chiffraten berechnet werden.

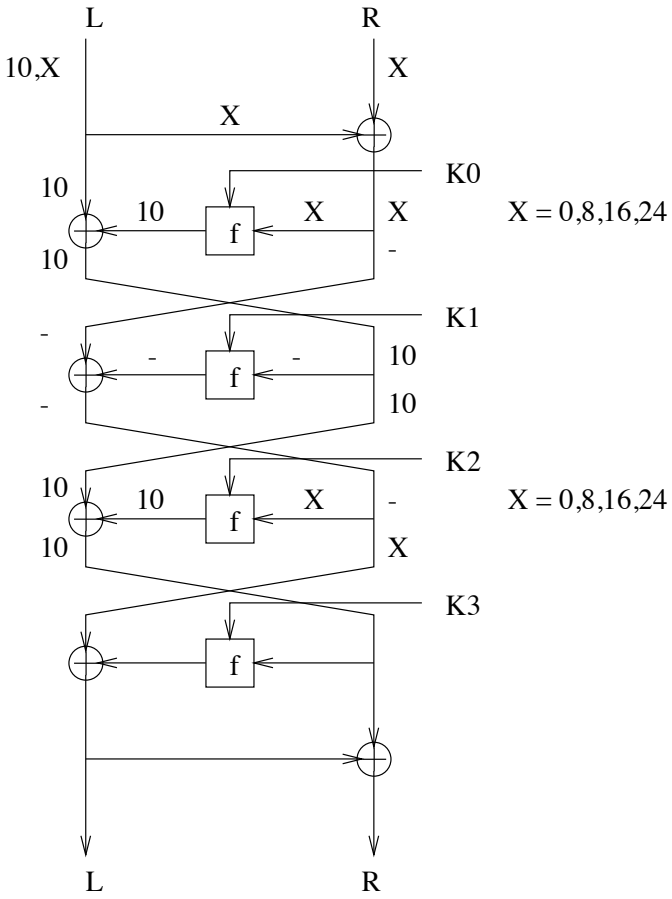

Abbildung 9.5.: Die Funktion $S_i(A, B)$

10. BREACH

10.1. BREACH im Überblick

Der BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) Exploit wurde 2013 vorgestellt. Er nutzt aus, dass mehrfache Vorkommen eines Teiltextes besser komprimiert werden können als verschiedene Teiltexte. Hat ein Angreifer Einfluss darauf was in Teilen eines komprimierten Dokumentes steht, so kann er aus der Dateigröße Schlüsse auf den andere Teile des Textes gewinnen. Der Angriff ist allgemein durchführbar falls ein Dokument erst komprimiert und dann verschlüsselt wird. Hier wird der konkrete Fall gegen die Deflate Kompression betrachtet.

10.2. Deflate

Die Kompression ist in RFC 1951[1] definiert. Es wird erst LZ77 verwendet um sich wiederholende Teiltexte zu komprimieren. Auf das Ergebnis wird dann eine Huffman-Codierung angewendet. Das Dokument wird in mehreren Blöcken verarbeitet, zur Vereinfachung wird im Rahmen dieser Aufgabe nur ein Block verwendet.

10.2.1. LZ77

LZ77 geht über den gesamten Text und speichert sich die gesehenen Zeichen in einem Puffer. Sobald ein Texteil gefunden wird der mindestens drei Zeichen mit einem vorhergehendem Textteil gemeinsam hat, wird dieser durch eine Referenz ersetzt. Die Referenz beschreibt wie lange das Duplikat ist und wie weit es zurück liegt. So wird etwa der Satz „Ein Spiel als Beispiel ist ein Spielerisches Beispiel.“ zu „Ein Spiel als Beis<5,13>ist e<8,28>erische<10,31>.“ komprimiert. Das Alphabet des Textes wird um die Referenzlängen erweitert, die Zeichen direkt nach einer Referenzlänge beschreiben immer den Abstand der Referenz und benötigt daher kein gesondertes Alphabet. Es werden außerdem, falls nötig, Extrabits eingefügt:

```
[...] the length is drawn from (3..258) and the distance is
drawn from (1..32,768).  In fact, the literal and length
alphabets are merged into a single alphabet (0..285), where
values 0..255 represent literal bytes, the value 256 indicates
end-of-block, and values 257..285 represent length codes
(possibly in conjunction with extra bits following the symbol
code) as follows:
```

[1] www.ietf.org/rfc/rfc1951.txt

```
      Extra               Extra               Extra
 Code Bits Length(s) Code Bits Lengths   Code Bits Length(s)
 ---- ---- ------   ---- ---- -------   ---- ---- -------
  257   0     3      267   1   15,16     277   4   67-82
  258   0     4      268   1   17,18     278   4   83-98
  259   0     5      269   2   19-22     279   4   99-114
  260   0     6      270   2   23-26     280   4  115-130
  261   0     7      271   2   27-30     281   5  131-162
  262   0     8      272   2   31-34     282   5  163-194
  263   0     9      273   3   35-42     283   5  195-226
  264   0    10      274   3   43-50     284   5  227-257
  265   1  11,12     275   3   51-58     285   0    258
  266   1  13,14     276   3   59-66
```

```
The extra bits should be interpreted as a machine integer
stored with the most-significant bit first, e.g., bits 1110
represent the value 14.
```

```
       Extra           Extra               Extra
 Code Bits Dist  Code Bits   Dist     Code Bits Distance
 ---- ---- ----  ---- ----  ------    ---- ---- --------
   0   0    1     10   4     33-48    20    9   1025-1536
   1   0    2     11   4     49-64    21    9   1537-2048
   2   0    3     12   5     65-96    22   10   2049-3072
   3   0    4     13   5     97-128   23   10   3073-4096
   4   1   5,6    14   6    129-192   24   11   4097-6144
   5   1   7,8    15   6    193-256   25   11   6145-8192
   6   2   9-12   16   7    257-384   26   12  8193-12288
   7   2  13-16   17   7    385-512   27   12 12289-16384
   8   3  17-24   18   8    513-768   28   13 16385-24576
   9   3  25-32   19   8   769-1024   29   13 24577-32768
```

Die Größe einer Referenz hängt also davon ab wie lange sie ist und wie weit sie zurück liegt (da eventuell zusätzliche Bits verwendet werden).

10.2.2. Huffman-Codierung

Nach der LZ77 Kompression wird eine Huffman-Codierung angewendet. Für hinreichend große Dokumente wird ein dynamischer Huffman-Code erzeugt und dem Dokument vorangehängt. Für kleine Dokumente (und im Rahmen dieses Versuches) wird ein fixer Huffman-Code verwendet der wie folgt definiert ist:

```
The Huffman codes for the two alphabets are fixed, and are not
represented explicitly in the data.  The Huffman code lengths
for the literal/length alphabet are:

          Lit Value    Bits        Codes
          ---------    ----        -----
            0 - 143     8          00110000 through
                                   10111111
          144 - 255     9          110010000 through
                                   111111111
          256 - 279     7          0000000 through
                                   0010111
          280 - 287     8          11000000 through
                                   11000111

[...]
```

```
Distance codes 0-31 are represented by (fixed-length) 5-bit
codes, with possible additional bits [...]
```

Man beachte, dass die Huffman-Codierung auf Zeichenebene und nicht auf Bytebene stattfindet. Insbesondere werden weder die Extrabits noch die Distanzcodes komprimiert.

10.3. Der Angriff

Bleibt die Dateigröße bei der Verschlüsselung erhalten, so kann ein Angreifer der Einfluss auf Teile des Dokumentes hat (etwa indem er ein Opfer veranlasst URLs mit GET Requests aufzurufen die im Dokument widergespiegelt werden) Dinge an anderer Stelle im Dokument erfahren (z.B. den Kontostand oder Suchpräferenzen). Dazu werden verschiedene Strings vom Angreifer geraten, ist das Dokument mit einem String kürzer als mit einem Anderem, so kommt dieser an anderer Stelle in dem Dokument vor. Weiß der Angreifer was vor oder nach dem Geheimnis steht, so kann er sicherstellen, dass der geratene String zumindest einem Teil des Geheimnisses entspricht, dazu muss nur das Prä- bzw. Postfix dem String beigefügt werden. Ist ein Teil des Geheimnisses bekannt, so können dann iterativ die nachfolgenden Zeichen geraten werden. Ein Angreifer kann also ein bekanntes Präfix „raten“ und dann nach und nach die folgenden Zeichen bestimmen.

10.4. Aufgabe: Erraten des Geheimnisses

Für diesen Versuch wurde ein Daemon eingerichtet das ein zufälliges Passwort zusammen mit ihrer Vermutung in ein zufälliges Dokument einsetzt und dieses dann wie beschrieben komprimiert und mit AES im CTR Modus verschlüsselt. Sie können ihre Vermutung im Feld `Guess` einer `Message` als `0`-terminierten String dem Deamon mitteilen, Ihnen wird dann eine Antwort in Form eines `Reply` zurück geschickt. Entspricht ihre Vermutung exakt dem Geheimnis (ohne Prä-/Postfix) so ist der `Type` `Correct` und der Versuch ist bestanden. Ansonsten ist `Type` `Wrong` und `Data` enthält das verschlüsselte Dokument der Länge `Len` in Bytes mit ihrer Vermutung. Um zu verhindern, dass einfach alle möglichen Geheimnisse geraten werden, wird das Geheimnis, sowie das Dokument, nach einer maximalen Anzahl von Rateversuchen neu gewählt.
Ergänzen sie die Funktion `attack()` in `attack.c` mit der Implementierung ihres Angriffes, ihnen steht `enc()` zur Verfügung um mit dem Daemon zu kommunizieren. Das Rahmenprogramm stellt sicher, dass vor dem Aufruf von `attack()` das Dokument sowie das Geheimnis zurückgesetzt werden, sodass ihnen die maximale Anzahl von Rateversuchen zur Verfügung stehen. Um den Versuch manuell zurück zu setzen können sie alternativ auch eine `Messsage` mit `Type` `Req_reset` senden, das Daemon antwortet dann mit einem `Reply` in dem `Type` auf `Correct` gesetzt ist (allerdings impliziert dies nicht, dass das Praktikum bestanden ist). Vor dem Geheimnis steht der String „`pw: `“ (inklusive Leerzeichen) und auf das Geheimnis folgt ein Leerzeichen. Das Geheimnis selbst besteht nur aus Kleinbuchstaben.

10.5. Hinweise zur Programmierung

- Die (verschlüsselte) Nachricht ist unwichtig für diese Aufgabe, es ist nur die Länge relevant.
- Nicht alle Zeichen haben nach der Huffman Codierung die gleiche Länge.
- AES-CTR rundet die Dateigröße auf das nächset Byte. Eventuell ist also eine längere oder weitere Referenz nicht von einer falschen Vermutung unterscheidbar, da die Nachricht in beiden Fällen um ein Byte größer wird.

11. El-Gamal Signatur

Ein weit verbreitetes Problem in der Krytographie ist das elektronische Unterschreiben von Dokumenten (z. B. Überweisungen im Bankbereich, E-Mail, ...). Ein Verfahren dazu ist die El-Gamal-Signatur, welche auf dem Diskreten Logarithmus Problem beruht.

11.1. Das Signaturverfahren

Setup: Eine Primzahl p und ein primitives Element $\omega \in \mathbb{Z}_p^*$ werden erzeugt und bekanntgegeben.

Initialisierung eines Teilnehmers A: A wählt eine Zufallszahl $x_A < p-1$, bildet $y_A := \omega^{x_A} \bmod p$ und publiziert y_A. Die Werte x_A und y_A können beliebig lange für Signaturen verwendet werden.

Signieren einer Nachricht m**:** A zieht eine Zufallszahl k mit $k < p-1$ und $ggT(k, p-1) = 1$ und berechnet $r := \omega^k \ mod\ p$ und $s := (m - r * x_A) * k^{-1} \bmod (p-1)$. r und s bilden die Signatur zur Nachricht m und werden zusammen mit dieser übertragen.

Die Signatur r, s einer Nachricht m ist korrekt gdw. ${y_A}^r * r^s \equiv \omega^m \bmod p$.

Folgende Probleme stellen sich bei der Verwendung von Signatur-Verfahren im Allgemeinen.

- Die Nachricht m muß genau so lang sein, wie die Zahlenlänge, bei uns 512 Bit. Ein lange Nachricht muß also entweder in einzelne zu signierende Blöcke zerlegt werden (rechenintensiv) oder es muß ein Verfahren zur Abbildung einer langen Nachricht auf eine kurze Zahl gefunden werden. Diese kurze Zahl wird als „MDC“ (*Message Digest Code*) bezeichnet. Zu seinen wichtigsten Eigenschaften zählt, daß zu einem gegebenen MDC nur schwer eine passende Nachricht konstruierbar sein sollte.
- Das geheime Datum von A, nämlich x_A muß auf irgend eine Weise vor fremdem Zugriff geschützt werden. Dies kann z. B. durch Chiffrierung und Speicherung an „geeigneter“ Stelle erreicht werden.
- Das für die Verifikation einer Signatur notwendige y_A muß (genau wie ω und p) von einer „vertrauenswürdigen“ Stelle auf einem authentischen Übertragungskanal zum Empfänger gelangen. Sonst könnte ein Angreifer die Signatur verändern und durch Modifikation von y_A, ω und p dafür sorgen, daß die Verfikation trotzdem erfolgreich ist.

11.2. Generierung der MDC

Zur Arbeitsminimierung wird im Versuch eine Nachricht m mit Hilfe des Algorithmus SHA256 auf eine MDC x abgebildet. Die Implementierung stellt hierfür die Funktion `Generate_MDC` zur Verfügung.

11.3. Versuchsaufbau

Um den Versuch möglichst realistisch zu gestalten, wird die Kommunikation zwischen den Praktikumsteilnehmern und einem Daemon-Prozeß[1] durch das El-Gamal-Signatur-Verfahren geschützt.

Jede Praktikumsgruppe kann dem Daemon-Prozeß eine der folgenden Nachrichten zusenden:

Der *ReportRequest* besteht aus dem signierten Accountnamen und fordert den Daemon zur Preisgabe des aktuellen Praktikum-Punktestandes auf. Dieser wird in einer *ReportResponse*-Nachricht zurückgeschickt, die

[1] Ein Daemon Prozeß läuft im Hintergrund und stellt irgendwelche Dienste, wie z. B. Drucken zur Verfügung

einen Text enthält, ob der betreffende Teilnehmer den Praktikumsschein erhalten wird. Die *ReportResponse* wird vom Daemon per El-Gamal signiert.

Mit der *VerifyRequest*-Nachricht schickt ein Teilnehmer eine vom Daemon unterschriebenen Report zu diesem zurück. Der Daemon überprüft seine eigene Unterschrift und teilt dem Teilnehmer mit einer *VerifyResponse*-Nachricht mit, ob er seine eigene Unterschrift akzeptiert hat. Diese Nachricht wird ebenfalls vom Daemon signiert.

Für jeden *Request* muss eine neue Verbindung zum Daemon mit `ConnectTo` aufgebaut werden. Über diese Verbindung wird die *Response* zurückgeliefert, danach schließt der Daemon sie wieder.

Die öffentlichen Daten eines jeden Benutzers sind in der Datei `$PRAKROOT/lib/public_keys.data` gespeichert und können mit der Prozedur `Get_Public_Key` gelesen werden. Diese Funktion erhält den Namen eines Benutzers A (durchgehend großgeschrieben) und liefert bei Erfolg dessen y_A zurück.

Die globalen Daten p und ω, sowie der private Schlüssel x werden mit der Funktion `Get_Privat_Data` aus dem File `privat_key.data` gelesen. Dieses File ist über den Protection-Mechanismus von Unix vor dem Lesen durch andere Personen geschützt.

Die Datenfiles für öffentliche und geheime Schlüssel sowie globale Daten werden *vor* Begin des eigentlichen Versuches durch die Praktikumsleiter erzeugt und an die richtigen Stellen im Dateibaum gebracht.

Die Funktionen `Get_Privat_Key` und `Get_Public_Key` sind in `SignSupport.c` vereinbart.

11.4. Versuch: Implementation der El-Gamal-Sigantur

Das C-Programm `getreport.c` soll vervollständigt werden. Es fehlen die Routinen zur Generierung einer El-Gamal-Signatur (für einen *ReportRequest*) und zur Überprüfung der vom Daemon erhaltenen *Report-Response*. Um die Aufgabe ein klein wenig interessanter zu machen, gibt der Daemon bei jeder zweiten Antwort eine **falsche** Unterschrift zurück. Dies ist auch aus dem gesendeten Text ersichtlich.

Die öffentlichen Daten aller Praktikumsgruppen und des Daemons stehen in `$PRAKTROOT/public_keys.data`. Die eigenen, geheimen Daten werden von der Software im Home-Directory der jeweiligen Gruppe unter `private_key.data` gesucht.

11.5. Versuch: Bestehen Sie das Praktikum

Der Daemon, teuflisch wie Prozesse seiner Art nun einmal sind, wird keinem Teilnehmer freiwillig ein Bestehen des Praktikums bescheinigen. Sie müssen sich also ihren Schein ergaunern, indem Sie eine Nachricht erzeugen, die Ihnen das Bestehen bescheinigt, und die die Unterschrift des Daemon trägt. Zur Überprüfung Ihrer gefälschten Nachricht können Sie die Nachrichtentypen *VerifyRequest* und *VerifyResponse* benutzten. Bei diesen Nachrichten arbeitet der Daemon einwandfrei und zuverlässig. Der Sourcecode zum Daemon befindet sich in der Datei `signdaemon.c` bei den restlichen Rahmenprogrammen.

Hinweis: Da auch im Kryptographie-Praktikum das Gute immer über das Böse siegt, ist diese Aufgabe tatsächlich in endlicher Zeit lösbar, weil wir einen kleinen Fehler in die Parameter der Signatur eingebaut haben!

Unwichtiger Hinweis: Bedenken Sie, daß der zuletzt an den Daemon gesendete Text, der Ihnen das Bestehen des Praktikums bescheinigt und eine korrekte Signatur trägt, auf dem Schein abgedruckt wird!

Wichtiger Hinweis: Diese Aufgabe ist **keine** Aufforderung, das Unix-System des Praktikums aufzuhacken, um den Code des Daemon zu modifizieren.

11.6. Anleitung

Eine Signatur nach El-Gamal und eine Berechnung der Hashfunktion mit SHA256 sind, korrekte Implementierungen vorausgesetzt, nicht trivial brechbar. Einer der kritischen Punkte ist die korrekte Wahl des Modulus

p. Wesentlich hierbei ist, daß $p-1$ über mindestens einen großen Faktor verfügt. Diese Bedingung wurde bei unserer Implementierung verlezt. Es ist daher möglich den diskreten Logarithmus mit vertretbarem Aufwand zu ziehen. Die Faktorisierung des Modulus ist in den Quellcodes angegeben.

11.6.1. Diskreter Logarithmus bei kleinen Faktoren von $p-1$

Zerfällt $p-1$ in kleine Primfaktoren $p_1, p_2, \ldots, p_k$, so kann der diskrete Logarithmus für jeden Primfaktor einzeln berechnet werden. Basis hierfür ist der kleine Satz von Fermat, der besagt, daß im Exponent modulo $p-1$ gerechnet werden kann:

$$p \text{ prim} \Rightarrow \forall a : a \equiv a^p \pmod{p}$$

Setzt man $p-1 = \prod_{i=1}^{k} p_i$ als Produkt unterschiedlicher Primzahlen und ω als erzeugendes Element der Gruppe an, so kann man das diskrete Logarithmus Problem modularisieren. Es sei (bei unbekanntem x):

$$a \equiv \omega^x \pmod{p}$$

Wir definieren für $i = 1, 2, \ldots, k$:

$$\begin{aligned} \omega_i &:\equiv \omega^{\frac{p-1}{p_i}} \pmod{p} \\ \text{und berechnen} \quad a_i :\equiv a^{\frac{p-1}{p_i}} &\equiv \omega^{x \cdot \frac{p-1}{p_i}} \equiv \omega_i^x \pmod{p}. \end{aligned}$$

Da für kleine p_i die Ordnung von ω_i sehr klein (nämlich p_i) ist, kann x modulo p_i durch vollständigen Suche gefunden werden. Bei größeren p_i versagt diese Methode jedoch.

11.6.2. Der Baby-Step–Giant-Step Algorithmus

Der hier vorgestellt Algorithmus zerlegt x_i in zwei Teile y_i und z_i, die jeweils kleiner als $\sqrt{p_i}$ sind. Dafür wird eine Hilfsgröße $q_i = \sqrt{p_i}$ eingeführt (q_i sollte der Wurzel zumindestens sehr nahe kommen). Es gilt nun:

$$\begin{aligned} x_i &= y_i + q_i z_i \\ \omega_i^{x_i} \equiv a_i &\equiv \omega_i^{y_i} (\omega_i^{q_i})^{z_i} \pmod{p} \\ a_i \omega_i^{-y_i} &\equiv (\omega_i^{q_i})^{z_i} \pmod{p} \end{aligned}$$

Die Werte y_i und z_i können nun mittels einer Meet in the middle Attacke gefunden werden, indem die Werte $a_i \omega_i^{-y_i}$ tabelliert werden.

11.6.3. Zusammensetzen des Exponenten

Sind die Werte x_i alle ermittelt, so kann der Wert x nach dem chinesischen Restsatz zusammengesetzt werden. Wie das geht, ist im Kapitel Mathematische Grundlagen in Abschnitt 3.8 beschrieben.

A. Die Network Support Library

A.1. Einleitung

Die Kommunikation mit den Daemon Prozessen ist als kleine Library extrahiert. Tatsächlich öffnet aber jeder Daemon ein TCP-Socket in einem festgelegten bereich und wartet dort auf Anfragen. Fast alle Daemon-Programme halten keinen Zustand zwischen Verbindungen. Das stellt sicher, dass jeder Praktikumsteilnehmer unabhänging von anderen Teilnehmern mit seiner eigenen Instanz des Daemon-Prozesses reden kann.

A.2. Verbindungsarten

A.2.1. Normale bidirektionale Verbindung

Eine *Connection* ist eine bidirektionale Verbindung zwischen genau zwei Partnern.

Stürzt einer der beiden Kommunikationspartner ab, oder schließt er die Verbindung, so ist für dessen Partner ab sofort kein Datentransport mehr möglich, da auch sein Ende der Verbindung geschlossen wird.

A.2.2. Ports

Ein *Port* ist eine global bekannt Anlaufstelle, zu der sich andere Prozesse verbinden können, um bestimmte Dienste, die der Portinhaber („Server“) anbietet, in Anspruch zu nehmen. Ein Port besitzt einen (eindeutigen) Namen. Es können mehrere Prozesse eine Verbindung zu einem Port aufbauen. Bei jeder Verbindung wird eine eigene Kopie des Deamon-Prozesses erzeugt die für die Abarbeitung dieser Verbindung zuständig ist.

Der „Server“ eröffnet ein Port. Anschließend muss er kund tun, daß er bereit ist, eine Verbindung auf diesem Port zu akzeptieren. Kommt diese *Port-Connection* zustande, entspricht sie der mit `ConnectTo` aufgebauten Kommunikationsbeziehung.

Nur der „Server“ ist in der Lage, ein Port zu schließen. Dies geschieht durch expliziten Aufruf einer Funktion oder durch Terminierung des Server-Programms.

A.3. Die Funktionen im Einzelnen

Um die Funktion verwenden zu können muss in C-Programmen das Include-File mit `#include "network.h"` eingelesen werden.

Die Typen `Connection` und `PortConnection` werden als Ergebniswerte der Funktionen benötig. Sie sind Zeiger auf `STRUCTs`. Ihr Aufbau ist zwar dokumentiert, es dürfen jedoch keine Komponenten verändert werden.

`NetName` ist der Typ der im Netzwerk verwendeten Namen. Dabei handelt es sich um Strings der Länge `MAXNAME_LEN`, z.Z. 80. Bei den Netzwerknamen spielt die Groß/Kleinschrift eine Rolle. Außer Buchstaben und Zahlen sind in Netzwerknamen nur die Sonderzeichen `_-` zugelassen.

Bei einem Fehler wird eine Fehlermeldung ausgegeben und das programm terminiert.

A.3.1. Verbindungsaufbau

A.3.1.1. ConnectTo

```
Connection ConnectTo(const char *name,const char *target)
```

Eine *Connection* wird mit der Funktion `ConnectTo(name,target)` aufgebaut. `name` ist der Netzwerkname „unseres“ Prozesses, `othersname` der des gewünschten Partners. `name` sollte der Praktikumsnutzernamen entsprechen. Standardmäßig wird hierfür der Name des aktuell angemeldeten Nutzers benutzt. Wenn dieser Name falsch ist, kann er durch setzen der Umgebungsvariable `PRAKTIKUM_NAME` überschrieben werden.

Beispiel: **Alice** will eine Verbindung zum Daemon **Bob** aufbauen. **Alice** ruft dann `ConnectTo("Alice","Bob")` auf. **Bob** muss vorher bereit sein Verbindungen anzunehmen. Das macht er mit `OpenPort("Bob")`. Anschließend ruft er `WaitAtPort(p)` auf. Diese Funktion beendet sich pro korrekt verbundenem Client einmal. **Bob** kann also direkt mit der Beantwortung der Anfrage beginnen.

Das Ergebnis von `ConnectTo` ist eine `Connection`. Sie muss beim Aufruf der Datentransportfunktionen und beim Abbau der Verbindung angegeben werden. Ihr Inhalt sollte nicht benutzt und schon gar nicht verändert werden.

A.3.1.2. OpenPort

```
PortConnection OpenPort(const char *portname)
```

`OpenPort` eröffnet ein globales Port, an dem die Dienste, die unser Prozeß anbietet, abgerufen werden können. Der Aufruf kehrt sofort zurück. Wenn schon ein Port oder eine andere Verbindung mit `portname` existiert, wird ein Fehler gemeldet und das Programm beendet.

Das Ergebnis von `OpenPort` ist eine `PortConnection`, die nur von der Funktion `WaitAtPort` verarbeitet werden kann (siehe Kapitel A.3.1.3, Seite II). Tritt ein Fehler auf, wird das Programm beendet.

A.3.1.3. WaitAtPort

```
Connection WaitAtPort(PortConnection *pc)
```

Mit `WaitAtPort` wartet ein „Server-Prozeß“ darauf, daß ein „Client“ Verbindung zu seinem Port aufnimmt. Die `PortConnection pc` ist der Rückgabewert eines vorhergehenden `OpenPort`-Aufrufs. Die von `WaitAtPort` zurückgelieferte `Connection` kann genau so wie eine von `ConnectTo` erzeugte zum bidirektionalen Datentransfer benutzt werden.

Wenn die Anfrage fertig bearbeitet ist, kann die Verbindung mit `DisConnect` (siehe Kapitel A.3.3.1, Seite III) geschlossen werden, oder einfach der aktuelle Prozess beendet werden, da ja ein Prozess pro einkommende Verbindung erzeugt wird.

A.3.2. Datentransferfunktionen

Allen Datentransferfunktionen sind die Parameter `buffer` und `length` gemeinsam. Beim Lesen gibt `buffer` die Adresse des Puffers an, der die empfangenen Daten aufnimmt. `length` gibt die Größe des Puffers in Bytes an. Es werden also maximal `length` viele Bytes gelesen.

Beim Schreiben ist `buffer` die Adresse des Puffers, aus dem `length` viele Bytes gesendet werden.

In C ist die Angabe eines Puffers mit dem Adressoperator und die Angabe der Größe des Puffers mit `sizeof` möglich.

Ein Beispiel:

```
typedef struct {
```

```
          x,y : integer;
            f : float;
        } data;
data x;
Connection con;
{
  ....
  Transmit(con,&x,sizeof(x));
  ....
}
```

A.3.2.1. Receive

```
int Receive(Connection con,void *buf,int length)
```

`Receive` liest von der Connection `con` genau `length` Bytes in den Puffer bzw. in das Objekt `buffer`. `con` ist das Ergebnis eines `ConnectTo`- oder `WaitAtPort`-Aufrufs.

Der Rückgabewert von `Receive` ist die Anzahl der tatsächlich gelesenen Bytes (also `length` im Erfolgsfall). 0 bedeutet, daß die Verbindung abgebaut wurde, kleiner 0 zeigt einen Fehler an.

A.3.2.2. ReceiveAll

```
void ReceiveAll(Connection con,void *buf,int length)
```

`ReceiveAll` verhält sich wie `Receive`, beendet aber das programm, wenn nicht `length` bytes gelesen werden konnten.

A.3.2.3. Transmit

```
void Transmit(Connection con,const void *buf,int length)
```

`Transmit` schreibt auf die Connection `con` `length` Bytes aus dem Puffer bzw. aus dem Objekt `buffer`. `con` ist das Ergebnis eines `ConnectTo`- oder `WaitAtPort`-Aufrufs.

Diese Funktion wird immer alle Daten übermitteln. Wenn dies fehlschlägt wird das Programm beendet.

A.3.3. Verbindungsabbau

Alle Verbindungen, die ein Programm geöffnet hat, werden beim Terminieren des Programms automatisch abgebaut. Eine Verbindung kann jedoch auch unter Programmkontrolle geschlossen werden.

Das Schließen einer Verbindung wird beim Partner durch Rückgabewert 0 der Datentransferfunktionen angezeigt. Alle in der Verbindung hängenden Tap- und Interconnections werden ebenfalls geschlossen. Das Abbauen einer Tap- oder Interconnection beeinflußt die abgehörte Verbindung nicht.

A.3.3.1. DisConnect

```
void DisConnect(Connection con)
```

`DisConnect` schließt die Connection `con`. `con` ist das Ergebnis eines `ConnectTo`- oder `WaitAtPort`-Aufrufs. Nach dem Schließen einer Port-Connection kann die Verbindung zum nächsten Clienten mit `WaitAtPort` aufgebaut werden.

A.3.4. Sonstige Routinen

A.3.4.1. MakeNetName

```
char *MakeNetName(const char *service);
```

`MakeNetName` erzeugt aus einem „Servicenamen“ und dem Benutzernamen einen String, der als innerhalb von Netzwerkanwendungen benutzt werden kann. Er ist besonders wichtig für die vom Dämon erzeugten Log-Dateien. Als Name wird standardmäßig der Unix-Benutzername benutzt. Wenn diesen nicht mit dem Name der Praktikumsgruppe übereinstimmt, sollte `PRAKTIKUM_NAME` dazu benutzt werden, um diesen namen zu überschreiben.

A.3.4.2. PeerName

```
const char *PeerName(PortConnection pc)
```

`PeerName` gibt den Namen des Kommunikationspartner, mit dem wir über die von `WaitAtPort` zurückgegebenen `PortConnection pc` verbunden sind. Dies ist der vom Partner bei `ConnectTo` angegebene eigene Name.

B. Die GNU MP Bignum Library

B.1. GMP im Überblick

Die GNU MP Bignum Library ist eine Bibliothek zum Umgang mit beliebig großen und genauen Zahlen. Sie wird vor allem in sicherheitsrelevanten Anwendungen verwendet. GMP wird seit 1991 aktiv entwickelt und regelmäßig aktualisiert.

Um GMP verwenden zu können muss gegebenenfalls das entsprechende Paket aus den Paketquellen installiert werden (zum Beispiel `gmp` in Debian, oder `math/gmp` in FreeBSD). Dem Linker muss `-lgmp` übergeben werden, dies wird bereits von den bereitgestellten Makefiles durchgeführt.

Wichtiger Hinweis: Diese Dokumentation beschreibt nicht alle Fähigkeiten von GMP. Aus Gründen der Übersichtlichkeit wurden nur Funktionen dokumentiert, die für die Lösung der Praktikumsaufgaben notwendig sind. Eine vollständige Anleitung ist unter https://gmplib.org/manual/ verfügbar.

B.2. Ganzzahlen

Ganzzahlen haben den Typ `mpz_t`. Bevor diese verwendet werden können, müssen sie mit `mpz_init()` initialisiert werden. Nach Verwendung müssen sie mit `mpz_clear()` wieder freigegeben werden. Funktionen die auf diesem Datentyp arbeiten beginnen mit dem Präfix `mpz_`, `mpz_add()` etwa addiert zwei Vorzeichen behaftete Ganzzahlen und speichert das Ergebnis in einer Dritten. Üblicherweise wird das Ergebnis einer Berechnung in dem erstem Parameter der Funktion gespeichert. Es können die gleichen Variablen zur Ein- und Ausgabe einer Funktion verwendet werden; der Funktionsaufruf `mpz_mul(x, x, x);` quadriert x.

B.2.1. Initialisieren, Kopieren, Löschen und Vergleichen

B.2.1.1. mpz_init — Initalisierung einer Ganzzahl

```
void mpz_init(op)
  mpz_t op                    write    Zu initialisierende Zahl
```

Ergebnis-Variablen: op
Return-Code: `void`

Beschreibung: `mpz_init` initialisiert op zur Verwendung mit GMP.

B.2.1.2. mpz_clear — Freigabe des durch eine Variable belegten Speichers

```
void mpz_clear(op)
  mpz_t op                    read     Zu befreiende Zahl
```

Ergebnis-Variablen:
Return-Code: `void`

Beschreibung: `mpz_clear` Gibt den Speicher einer Variable wieder frei.

B.2.1.3. mpz_set — Setzen des Wertes

```
void mpz_set(rop, op)
  mpz_t rop                    write    Ergebnis
  const mpz_t op               read     Wert
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_set` Kopiert den Wert von op nach rop.
Varianten:

- `mpz_set_si`: op ist ein signed long.
- `mpz_set_ui`: op ist ein unsigned long.
- `mpz_set_d`: op ist ein double.

B.2.1.4. mpz_init_set — Initialisierung mit existenter Ganzzahl

```
void mpz_init_set(rop, op)
  mpz_t rop                    write    Kopie
  const mpz_t op               read     Original
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_init_set` initialisiert rop und setzt den Wert auf den Wert von op.
Varianten:

- `mpz_init_set_si`: Initialisierung durch einen signed long.
- `mpz_init_set_ui`: Initialisierung durch einen unsigned long.
- `mpz_init_set_d`: Initialisierung durch einen double.

B.2.1.5. mpz_swap — Vertauschen von zwei Werten

```
void mpz_swap(rop1, rop2)
  mpz_t rop1                   modify   Erster Wert
  mpz_t rop2                   modify   Zweiter Wert
```

Ergebnis-Variablen: rop1, rop2
Return-Code: `void`

Beschreibung: `mpz_swap` tauscht rop1 und rop2.

B.2.1.6. mpz_cmp — Vergleich zweier Ganzzahlen

```
int mpz_cmp(op1, op2)
  const mpz_t op1                read    Erster Wert
  const mpz_t op2                read    Zweiter Wert
```

Ergebnis-Variablen: op1, op2
Return-Code: `int` int

Beschreibung: `mpz_cmp` gibt einen Wert größer Null zurück wenn op1 > op2, Null wenn op1 = op2 und kleiner null wenn op1 < op2.
Variante: `mpz_cmp_d`: op2 ist ein double.

B.2.2. Arithmetik

B.2.2.1. mpz_add — Addition zweier Ganzahlen

```
void mpz_add(rop, op1, op2)
  mpz_t rop                      write   Ergebnis
  const mpz_t op1                read    Erster Summand
  const mpz_t op2                read    Zweiter Summand
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_add` Addiert op1 mit op2 und speichert das Ergebnis in rop.
Variante: `mpz_add_ui`: op2 ist ein unsigned long.

B.2.2.2. mpz_sub — Subtraktion zweier Ganzzahlen

```
void mpz_sub(rop, op1, op2)
  mpz_t rop                      write   Ergebnis
  const mpz_t op1                read    Minuend
  const mpz_t op2                read    Subtrahend
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_sub` Subtrahiert op2 von op1 und speichert das Ergebnis in rop.
Varianten:

- `mpz_sub_ui`: op2 ist ein unsigned long.
- `mpz_ui_sub`: op1 ist ein unsigned long.

B.2.2.3. mpz_neg — Negation einer Ganzzahl

```
void mpz_neg(op)
  const mpz_t op                  modify    Zu negierende Zahl
```

Ergebnis-Variablen: op
Return-Code: `void`

Beschreibung: `mpz_neg` negiert op.

B.2.2.4. mpz_mul — Multiplikation zweier Ganzzahlen

```
void mpz_mul(rop, op1, op2)
  mpz_t rop                       write     Ergebnis
  const mpz_t op1                 read      Erster Faktor
  const mpz_t op2                 read      Zweiter Faktor
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_mul` multipliziert op1 mit op2 und speichert das Ergebnis in rop.
Varianten:

- `mpz_mul_ui`: Multiplikation mit einem unsigned long.
- `mpz_mul_si`: Multiplikation mit einem (signed) long.

B.2.2.5. mpz_mod — Berechnung des Modulo

```
void mpz_mod(r, n, d)
  mpz_t r                         write     Ergebnis
  const mpz_t n                   read      Divident
  const mpz_t d                   read      Divisor
```

Ergebnis-Variablen: r
Return-Code: `void`

Beschreibung: `mpz_mod` Berechnet n mod d und speichert das Ergebnis in r. Das Ergebnis ist immer positiv.

B.2.2.6. mpz_cdiv_q — Division zweier Ganzzahlen mit Abrunden

```
void mpz_cdiv_q(q, n, d)
  mpz_t q                         write     Quotient
  const mpz_t n                   read      Divident
  const mpz_t d                   read      Divisor
```

Ergebnis-Variablen: q
Return-Code: `void`

Beschreibung: `mpz_cdiv_q` Teilt n durch d und speichert den aufgerundeten Quotienten in q.
Anmerkung: Es handelt sich hier um eine Funktionsfamilie. Das “c“ in “cdiv“ steht für “ceil“ (also aufrunden), “fdiv“ rundet ab und “tdiv“ rundet zu Null (relevant bei Zahlen mit Vorzeichen). Das “q“ am Ende sagt aus dass der Quotient berechnet werden soll, ein “r“ Berechnet den Rest und “qr“ berechnet Beides.

B.2.2.7. mpz_tdiv_qr — Division zweier Ganzzahlen mit Rest und Runden zu Null

```
void mpz_tdiv_qr(q, r, n, d)
  mpz_t q                       write    Quotient
  mpz_t r                       write    Rest
  const mpz_t n                 read     Divident
  const mpz_t d                 read     Divisor
```

Ergebnis-Variablen: q, r
Return-Code: `void`

Beschreibung: `mpz_tdiv_qr` teilt n durch d, der Quotient wird zu Null hin gerundet und in q gespeichert, der Rest wird in r gespeichert.

B.2.2.8. mpz_powm — Exponentation mit Modulo

```
void mpz_powm(rop, base, exp, mod)
  mpz_t rop                     write    Ergebnis
  const mpz_t base              read     Basis
  const mpz_t exp               read     Exponent
  const mpz_t mod               read     Modulo
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_powm` Berechnet ($base^{exp}$) modulo mod und speichert das Ergebnis in rop. Negative Exponenten sind möglich wenn ein entsprechendes Inverses existiert.

B.2.3. Zahlentheoretische Funktionen

B.2.3.1. mpz_invert — Invertierung einer Ganzzahl

```
int mpz_invert(rop, op1, op2)
  mpz_t rop                     write    Inverses
  const mpz_t op1               read     Zu invertierende Zahl
  const mpz_t op2               read     Modulo
```

Ergebnis-Variablen: rop
Return-Code: `int` 0 falls kein Inverses existiert

Beschreibung: `mpz_invert` invertiert op1 in modulo op2. Falls ein Inverses existiert ist der Rückgabewert ungleich Null und es gilt 0 <= rop <= abs(op2).

B.2.3.2. mpz_probab_prime_p — Primzahltest

```
int mpz_probab_prime_p(n, reps)
  const mpz_t n                  read     Zu testende Zahl
  int reps                       read     Anzahl der Miller-Rabin Iterationen
```

Ergebnis-Variablen:
Return-Code: `int` >0 falls n eine Primzahl ist

Beschreibung: `mpz_probab_prime_p` testet ob die Zahl n eine Primzahl ist. Ist die Zahl definitiv eine Primzahl, wird 2 zurückgegeben, ist sie definitiv keine Primzahl, wird 0 zurückgegeben. Falls die Zahl wahrscheinlich eine Primzahl ist, wird 1 zurückgegeben. Es wird der probabilistische Primzahltest von Miller-Rabin verwendet. Mit dem Parameter reps kann die Anzahl der Iterationen festgelegt werden. Falls der Rückgabewert 1 ist, ist n mit höchstens der Wahrscheinlichkeit 4^{-reps} keine Primzahl. Angemessene Werte für reps sind zwischen 15 und 50.

B.2.3.3. mpz_gcd — Gröster gemeinsamer Teiler

```
void mpz_gcd(rop, op1, op2)
  mpz_t rop                      write    Kleinster gemeinsamer Teiler
  const mpz_t op1                read     Erster Parameter
  const mpz_t op2                read     Zweiter Parameter
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_gcd` berechnet den größten gemeinsamen Teiler von op1 und op2 und speichert diesen in rop. rop ist immer positiv.

B.2.4. Sonstige

B.2.4.1. gmp_randinit_default — Initalisierung des Zustandes für den Zufallszahlengenerator

```
void gmp_randinit_default(state)
  gmp_randstate_t state          write    Zustand
```

Ergebnis-Variablen: state
Return-Code: `void`

Beschreibung: `gmp_randinit_default` erzeugt einen neuen Zustand für den Zufallszahlengenerator.

B.2.4.2. mpz_urandomm — Zufallszahl

```
void mpz_urandomm(rop, state, n)
  mpz_t rop                      write    Zufällige Zahl
  gmp_randstate_t state          modify   Zustand des Zufallsgenerators
  const mpz_t n                  read     Obere Grenze
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_urandomm` erzeugt eine Zufallszahl zwischen 0 un n-1 (inklusive) und speichert sie in rop. Der Zustand muss vorher mit gmp_randinit_default initialisiert werden.

B.2.4.3. mpz_and — Bitweises Und

```
void mpz_and(rop, op1, op2)
  mpz_t rop                    write   Ergebnis
  const mpz_t op1              read    Erster Parameter
  const mpz_t op2              read    Zweiter Parameter
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_and` berechnet das bitweise Und von op1 und op2 und speichert das Ergebnis in rop.
Varianten:

- `mpz_ior`: Bitweises Oder
- `mpz_xor`: Bitweises exklusives Oder

B.2.4.4. mpz_com — Bitweises Komplement

```
void mpz_com(rop, op)
  mpz_t rop                    write   Bitweises Komplement
  const mpz_t op               read    Eingabe
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_com` berechnet das bitweise Komplement von op und speichert es in rop.

B.2.4.5. mpz_setbit — Bit setzen

```
void mpz_setbit(rop, bit_index)
  mpz_t rop                    modify  Zu modifizierende Zahl
  mp_bitcnt_t bit_index        read    Bitindex
```

Ergebnis-Variablen: rop
Return-Code: `void`

Beschreibung: `mpz_setbit` setzt das Bit an der Position bit_index.
Variante: `mpz_clrbit` setzt das Bit an der Position bit_index auf 0.

B.2.4.6. mpz_tstbit — Bit auslesen

```
int mpz_tstbit(op, bit_index)
  const mpz_t op               read    Eingabe
  mp_bitcnt_t bit_index        read    Bitindex
```

Ergebnis-Variablen: rop
Return-Code: `int` 1 wenn das Bit gesetzt ist, 0 sonst.

Beschreibung: `mpz_tstbit` Liest das Bit and der Position bit_index aus.

Index

mpz_urandomm, X
Zustandsinitialisierung
gmp_randinit_default, X