Rechnernetze und -Organisation
Assignment A2

Horst Possegger

Technische Universität Graz,
Institut für Angewandte Informationsverarbeitung
und Kommunikationstechnologie (IAIK)
http://www.iaik.tugraz.at/

Sommersemester 2010

Zusammenfassung

Das Assignment A2 der Konstruktionsübung RNO [1] beschäftigt sich im Sommersemester 2010 mit dem Zusammenspiel von Softwarekomponenten, die in unterschiedlichen Programmiersprachen erstellt wurden. Aufbauend auf das Assignment A1 sollen Daten mit Hilfe eines 32-Bit linear rückgekoppelten Schieberegisters verschlüsselt werden. Um die verschlüsselten Daten anschließend über beliebige Kanäle transportieren zu können, werden diese Base 64 kodiert. Ziel des Assignments ist es, einen Einblick in das Zusammenspiel unterschiedlicher Softwarekomponenten zu erhalten.

1 Einleitung

Ausgehend von Ihren Erfahrungen mit dem 16-Bit-TOY-Simulator werden Sie für Assignment A2 ein einfaches Stromverschlüsselungsverfahren auf einem 32-Bit-System implementieren.

Bei den meisten Softwareprojekten entwickelt man in einer Hochsprache wie z.B. C. Dies bringt den Vorteil, dass man sich nicht mit den niedrigsten Implementierungsdetails beschäftigen muss, der Code leicht

(a) Auszug einer Base-64-Inline-Kodierung

(b) Anzeige im Browser

Abbildung 1: Inline-kodiertes Bild

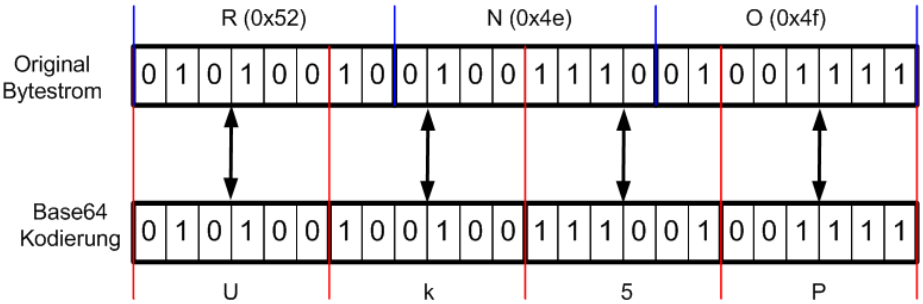

Abbildung 2: Base-64-(De-)Kodierung der Zeichenkette `RNO`

verständlich ist und das Programm weniger fehleranfällig ist. Trotzdem gibt es den Bedarf, einzelne Module in anderen Programmiersprachen zu entwickeln, z.B. aus Performancegründen, bei sehr maschinenabhängigen Codeteilen, etc.

In Assignment A2 werden Sie einzelne Module in C und Assembler umsetzen und so das Zusammenspiel unterschiedlicher Softwarekomponenten kennenlernen.

1.1 Base-64 Data Encoding

Base-64 dient zur Kodierung von Binärdaten in druckbare ASCII-Zeichenfolgen und ermöglicht so den einfachen Transport von beliebigen Binärdaten. Verwendung findet die Base-64-Datenkodierung hauptsächlich beim Versenden von Email-Anhängen oder um mit Hilfe des URI-Schemas `data`, Daten inline in Webseiten zu inkludieren. Abbildung 1 zeigt ein inline-kodiertes Bild.

Zur Kodierung werden die Binärdaten in 3 Byte große Blöcke unterteilt. Diese insgesamt 24 Bit werden in vier 6-Bit-Blöcke aufgespalten. Nun wird jedem 6-Bit-Block entsprechend seinem Wert ein druckbares Zeichen aus der 64-teiligen Lookup-Table zugewiesen, siehe Tabelle 1. Abbildung 2 zeigt die Base-64-Kodierung der Zeichenkette `"RNO"`.

Durch diese Vorgehensweise wird ein Nachteil der Base-64-Kodierung ersichtlich: der erhöhte Speicherbedarf. Die kodierte Zeichenfolge benötigt rund 33% mehr Speicherplatz als die originalen Binärdaten.

Wert	Zeichen	Wert	Zeichen	Wert	Zeichen	Wert	Zeichen
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

Tabelle 1: Base-64-Kodierungstabelle

Ist die Anzahl der zu kodierenden Bytes nicht durch 3 teilbar, müssen die Binärdaten mit Füllbytes (Padding) aufgefüllt werden, um die Kodierung durchführen zu können. Je nach Länge der Daten müssen bis zu zwei Füllbytes berücksichtigt werden. Als Füllbyte wird `0x00` verwendet. Um die Originaldaten wiederherstellen zu können, müssen diese Paddings gekennzeichnet werden. Daher werden kodierte Zeichen, die nur aus Füllbytes erzeugt wurden, mit dem Sonderzeichen '=' abgespeichert. So kann beim Dekodieren festgestellt werden, dass diese Bytes nicht Teil der originalen Daten sind.

Beim Dekodieren werden aus je 4 Zeichen der kodierten Zeichenfolge wieder die entsprechenden 3 Byte der originalen Binärdaten extrahiert. Dazu muss zuerst über das Base-64-Alphabet der zu dem Zeichen zugehörige Wert ermittelt werden. Anschließend werden die 4 Werte entsprechend kombiniert, um den Bytestrom wiederherzustellen.

In Umgebungen, in denen die Zeichen '+' und/oder '/' eine besondere Bedeutung haben (z.B. Dateipfade), werden die beiden Zeichen durch '-'/'_' ersetzt. Dieses geänderte Base-64-Alphabet wird auch als *URL and filename safe* bezeichnet. Weitere Informationen zur Base-64-Kodierung finden Sie unter [2].

1.2 Linear Rückgekoppeltes Schieberegister

Die Grundlagen von linear rückgekoppelten Schieberegistern (Linear Feedback Shift Register – LFSR) wurden bereits im Rahmen von Assignment A1 erläutert. Hier sollen nur mehr die wichtigsten Punkte kurz zusammengefasst werden. Die genaue Spezifikation des LFSR für Assignment A2 finden Sie in Abschnitt 2.2.

LFSR binär	Feedback-Bit	Output-Bit
001101	1	1
100110	1	0
110011	0	1
011001	1	1
101100	0	0
⋮	⋮	⋮

Tabelle 2: 6-Bit-LFSR mit Polynom $x^6 + x^5 + 1$

Ein linear rückgekoppeltes Schieberegister kann zur Erzeugung von Pseudo-Zufallszahlen verwendet werden. Zuerst wird das Schieberegister mit einem Startwert (*seed*) initialisiert. Der Startwert 0 muss vermieden werden, da das Schieberegister in diesem Fall nur 0-Bits liefern würde. Ein für das Schieberegister charakteristisches Polynom (zB $x^6 + x^5 + 1$) definiert einzelne Bits (sogenannte *taps*), die zur Erzeugung eines Feedback-Bits verwendet werden. Als Output-Bit wird das *Least Significant Bit* (*LSB*) des Schieberegisters verwendet. Weiters wird das Feedback-Bit über eine XOR-Verknüpfung aller Tap-Bits ermittelt. Anschließend werden die Bits im Schieberegister um eine Stelle nach rechts verschoben und das Feedback-Bit an der Stelle des *Most Significant Bit* (*MSB*) eingefügt. Tabelle 2 zeigt ein 6-Bit Schieberegister mit 2 *taps*. Weitere Informationen zu Linear Feedback Shift Register finden Sie auch unter [3], Kapitel 6.2.

Mit einem n-Bit-Schieberegister können bis zu $2^n - 1$ Bitfolgen erzeugt werden, bis wieder die gleiche Bitfolge ausgehend vom gewählten Startwert erreicht wird.

2 Aufgabenstellung

Das Assignment A2 besteht aus einem C-Teil und einem Assembler-Teil. Die Base-64-Kodierung eines Bytestroms soll durch ein C-Modul erfolgen. Die Verschlüsselung mit Hilfe eines 32-Bit-LFSR wird in Assembler realisiert. Aus den unterschiedlichen Modulen werden Objektdateien erzeugt, die mit Hilfe eines Linkers zu einem Executable zusammengefügt werden, siehe Abbildung 3.

2.1 Base-64-Kodierung

Der erste Teil des Assignment A2 besteht aus der Implementierung eines Base-64-En- bzw Dekoders. Dazu müssen Sie in der Datei `base64.c` die beiden Funktionen `encodeBase64(...)` und `decodeBase64(...)` implementieren. Die zu verwendende Lookup-Table ist ebenfalls in dieser Datei bereitgestellt. Im Rahmen dieses Assignments muss die Kodierung mit dem Base-64-Standard- Alphabet umgesetzt werden, siehe auch Tabelle 1.

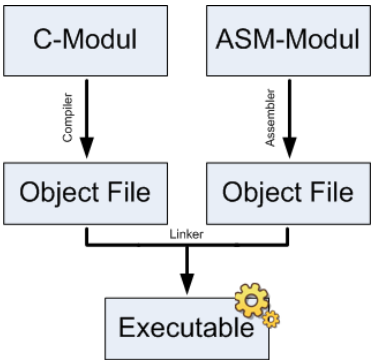

Abbildung 3: Erzeugung der Object-Files und Linken des Executables

2.2 Verschlüsselung

Für die (sehr vereinfachte) Verschlüsselung der Daten müssen Sie in der Datei `crypto.s` ein 32-Bit-LFSR implementieren. Als Startwert (Seed) des Schieberegisters dient Ihre Matrikelnummer. Also wird z.B. für die Matrikelnummer `0912345` das LFSR mit `0x0912345` initialisiert. Das Rückkopplungspolynom besitzt 6 *taps* und ist abhängig von Ihrem Startwert:

$$\text{Polynom} = \begin{cases} x^{32}+x^{31}+x^{30}+x^{29}+x^{28}+x^{22}+1 & \text{wenn } \mathit{seed} \bmod 4 = 0 \\ x^{32}+x^{31}+x^{30}+x^{27}+x^{25}+x^{18}+1 & \text{wenn } \mathit{seed} \bmod 4 = 1 \\ x^{32}+x^{31}+x^{30}+x^{23}+x^{17}+x^{7}+1 & \text{wenn } \mathit{seed} \bmod 4 = 2 \\ x^{32}+x^{31}+x^{30}+x^{22}+x^{14}+x^{12}+1 & \text{sonst.} \end{cases}$$

In unserem Beispiel würden wir das Polynom $x^{32}+x^{31}+x^{30}+x^{27}+x^{25}+x^{18}+1$ verwenden, da $0\text{x}0912345 \bmod 4 = 1$.

Die Daten müssen byteweise verschlüsselt werden. Zur Verschlüsselung eines Bytes benötigen wir 8 Output-Bits des Schieberegisters. Durch eine XOR-Verknüpfung erhalten wir das verschlüsselte Byte. Das LSB des Bytes wird mit dem ersten Output-Bit XOR-verknüpft, usw. Durch XOR-Verknüpfung der verschlüsselten Daten mit demselben Schlüssel erhalten wir wiederum die originalen Daten. Das folgende Beispiel zeigt die Verschlüsselung eines Bytes.

$$\begin{aligned} \text{Output-Bits} &= [1,0,1,0,1,1,1,0,\dots] \\ \text{Byte} &= 0\text{x}10 = 00010000_b \\ \text{Verschlüsseltes Byte} &= 00010000_b \oplus 01110101_b \\ &= 01100101_b = 0\text{x}65 \end{aligned}$$

Die Funktion `encrypt(...)` ist in Assemblersprache abzugeben. Eine Einführung zur Assembler-Programmierung unter Linux finden Sie in *Introduction to Linux Intel Assembly Language* von Norman Matloff [4].

Achtung: Wir empfehlen Ihnen, das LFSR zuerst in der Datei `crypo.c` zu implementieren und zu testen. Anschließend können Sie mit dem Befehl

`> gcc -S crypto.c`

den zugehörigen Assembler-Code generieren. Der Assembler-Code wird in der Datei `crypto.s` gespeichert (**Achtung: vorhandene Änderungen an `crypto.s` werden dabei überschrieben!!**).

Ihre Aufgabe ist es nun, den generierten Code zu verstehen und händisch zu "verfeinern". Sie sollen also den Code kommentieren und bei Bedarf Änderungen vornehmen, sofern diese für das bessere Verständnis notwendig sind (Labels umbenennen, ...). Wichtig ist, dass Sie den Assembler-Teil verstehen und beim Abgabegespräch erklären können.

Wenn Sie diese Variante wählen, hier noch ein paar Ratschläge, die Ihnen beim Verstehen helfen können:

- Denken Sie beim Entwickeln in C schon an die Lesbarkeit des assemblierten Codes (keine unnötig komplexen Konstrukte...).
- Schreiben Sie einfache Testprogramme/-funktionen (wie zB `add(x,y)`), erzeugen Sie daraus Assembler-Code und versuchen Sie, diesen zu verstehen.
- Wenn Ihnen die Adressierung der lokalen Funktionsvariablen Verständnisprobleme bereitet, versuchen Sie stattdessen globale Variablen zu verwenden (obwohl globale Variablen nicht immer empfehlenswert sind ...).

Der Vorteil dieser Variante ist, dass Sie bei der Implementation weniger Programmierfehler einbauen und sich intensiver mit dem Verstehen des Assembler-Teils auseinandersetzen können.

Anmerkung: Standardmäßig wird die Datei `crypto.c` bei einem `make`-Aufruf nicht berücksichtigt. Wenn Sie das LFSR zuerst in dieser Datei implementieren, müssen Sie das entsprechende Target im Makefile adaptieren. Die notwendigen Änderungen sind im Makefile beschrieben.

Es steht Ihnen natürlich frei, diesen Teil der Aufgabenstellung direkt in der Datei `crypto.s` (ohne Umweg über `crypto.c`) zu implementieren, sofern Sie dies bevorzugen.

Um im Assembler-Teil auf die Funktionsparameter zugreifen zu können, ist es notwendig den Stackaufbau bei Funktionsaufrufen zu verstehen. Die Übergabe von Parameter an Subroutinen erfolgt über den Stack. Zu diesem Zweck müssen diese von der aufrufenden Funktion in umgekehrter Reihenfolge auf den Stack "gepusht" werden. Anschließend wird die Adresse des nächsten Befehls (*Return Instruction Pointer*) "gepusht", um nach erfolgtem Funktionsaufruf bei der nächsten Anweisung fortführen zu können. Weiters wird der aktuelle Wert des Basepointers (*EBP*-Register) "gepusht" — der Grund dafür wird im nächsten Absatz erläutert. Abschließend wird noch Speicherplatz für lokale Variablen reserviert (sofern notwendig).

Da sich der Stackpointer (*ESP*-Register) im Laufe der Funktionsabarbeitung häufig ändern kann, erfolgt die Adressierung der Parameter und lokalen Variablen über den Basepointer (*EBP*). Das EBP-Register markiert den sogenannten *Stackframe* der Funktion. Um bei Beenden der Funktion den Stackframe des Aufrufers wiederherstellen zu können, muss der vorherige Wert des EBP-Registers zwischengespeichert werden (natürlich am Stack). Somit ergibt sich ein Stacklayout wie aus Abbildung 4 ersichtlich.

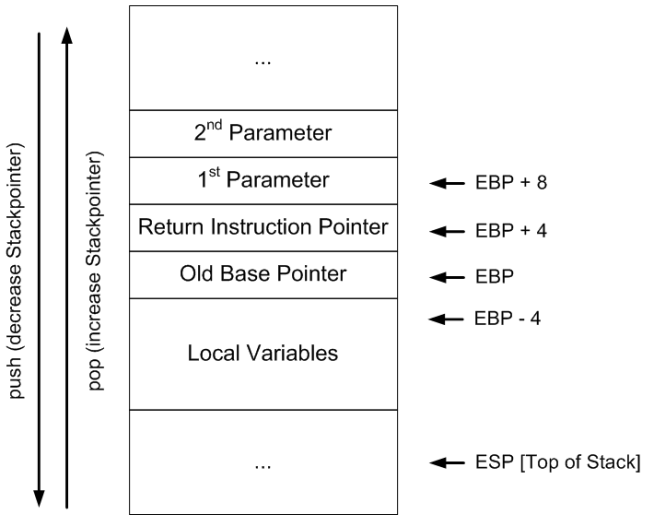

Abbildung 4: Stacklayout bei Aufruf einer Subroutine

Eine detailliertere Beschreibung zu Subroutinen finden Sie in Matloffs *Subroutines on Intel CPUs* [5].

Achtung: Um eine korrekte Verwendung des Stacks zu garantieren, müssen Sie **alle** Werte, die Sie auf den Stack pushen auch wieder entfernen (pop). Ansonsten können Programmabstürze oder Programmfehler (falsche Rücksprungadressen, Stack wächst immer weiter...) auftreten.

3 Framework

Das bereitgestellte Framework ist verpflichtend zu verwenden. Bei der Entwicklung wurden keine fortgeschrittenen Programmiertechniken verwendet, damit für weniger Erfahrene der Code leichter verständlich ist.

3.1 Erlaubte und verbotene Änderungen

Die Dateien `base64.c`, `base64.h`, `crypto.h`, `crypto.c` und `crypto.s` dürfen geändert werden. Alle anderen Dateien werden nach Ihrer Abgabe durch die Originale ersetzt. Testen Sie deshalb **vor** der Abgabe, ob Ihre Lösung auch mit den originalen Framework-Dateien funktioniert (sofern Sie die Framework-Dateien geändert haben...).

Wenn Sie Debug-Ausgaben einbauen, stellen Sie sicher, dass diese beim automatischen Testen Ihrer Abgabe nicht aufscheinen (z.B. mittels Commandline-Parameter, Flag im Source, ...).

Achtung: Damit Ihre Implementierung richtig getestet werden kann, müssen Sie in der Datei `test.c` Ihre Matrikelnummer eintragen (siehe entsprechendes `TODO` im Code).

3.2 Testcases

Es stehen Testcases zur Verfügung, die Sie beim Überprüfen der Funktionalität unterstützen sollen. Der Aufruf erfolgt dabei entweder über entsprechenden Parameter beim Executable oder als `make`-Target.

Testcase pad0

Base-64-Kodierung und Verschlüsselung einer Zeichenkette, bei deren Base-64-Kodierung keine Füllbytes notwendig sind.
`> make test_pad0` (oder `./rno_a2 pad0`)

Testcase pad1

Base-64-Kodierung und Verschlüsselung einer Zeichenkette, bei deren Base-64-Kodierung ein Füllbyte angehängt werden muss.
`> make test_pad1` (oder `./rno_a2 pad1`)

Testcase pad2

Base-64-Kodierung und Verschlüsselung einer Zeichenkette, bei deren Base-64-Kodierung zwei Füllbytes benötigt werden.
`> make test_pad2` (oder `./rno_a2 pad2`)

Testcase txt

Base-64-Kodierung und Verschlüsselung einer längeren Zeichenkette.
`> make test_txt` (oder `./rno_a2 txt`)

Testcase binary

Base-64-Kodierung einer Bilddatei.
`> make test_binary` (oder `./rno_a2 binary`)

Ausführung aller Testcases

Um alle Testcases hintereinander ausführen zu können, steht Ihnen ein Script zur Verfügung.
`> make test_all` (oder `./runalltests.sh`)

3.3 Arbeitsumgebung

Wir empfehlen Ihnen die Entwicklung unter einer Unix-basierten Plattform, z.B. Ubuntu, Debian o.ä. Sollten Sie mit einer anderen Plattform arbeiten, können Sie sich Ihre Arbeitsumgebung mit VMware [7] oder VirtualBox [8] einrichten. Alternativ kann das Assignment auch unter einer Umgebung wie MinGW oder Cygwin entwickelt werden. Wir empfehlen Ihnen jedoch eine der oben angeführten Varianten.

Für die Implementierung muss eine aktuelle Version der GNU Compiler Collection (`gcc`) sowie GNU Make (`make`) installiert sein. Da in der

GCC bereits der GNU Assembler (`gas`) enthalten ist, wird kein zusätzlicher Assembler benötigt. Benutzer eines 64-Bit-Systems benötigen das Paket `gcc-multilib`, um 32-Bit Executables erstellen zu können.

Für die Implementierung des Assignment A2 empfehlen wir Ihnen die Verwendung Ihres bevorzugten Texteditors (`kate`, `gedit`, `vi`...). Die Teilaufgaben sind in sich abgeschlossen und Sie werden IDE-Funktionalitäten wie Code-Completion etc. nur äußerst selten (wenn überhaupt) benötigen. Es steht Ihnen jedoch frei, das Projekt in eine IDE zu importieren und dort zu entwickeln.

4 Allfälliges

Allgemeines zu den Übungen ist auf der LV-Website zusammengefasst [1]. Dort finden Sie Informationen zu

- Ingenieurstagebuch
- Beurteilungsmodus
- Punkteschlüssel
- Punkteabzüge bei verspäteter Abgabe
- ...

Nutzen Sie auch die Newsgroup `news.tugraz.at/tu-graz.lv.rno`. Dort werden Neuigkeiten zur Konstruktionsübung bekannt gegeben. Die Newsgroup dient auch als Diskussionsplattform für Studierende. Stellen Sie dort Fragen, von denen Sie denken, dass diese auch für andere relevant sind.

4.1 Abgabe

Die Abgabe von Assignment A2 muss bis spätestens Freitag, 7. Mai 2010 um 23:59:59 erfolgen. Spätere Abgaben erhalten Punkteabzüge. Für die Abgabe ist die Webmaske am RNO-Web `http://www.iaik.tugraz.at/content/teaching/bachelor_courses/rechnernetze_und_organisation/practicals/#assign2` zu verwenden. Packen Sie die Abgabedateien (`base64.h`, `base64.c`, `crypto.h`, `crypto.c`, `crypto.s`, `test.h`, `test.c`, das Makefile und ein README.txt) mit

```
> make submission
```

zu einem `.zip`-Archiv zusammen. Wie bereits erwähnt, werden einige Dateien des Frameworks beim Testen Ihrer Implementierung ersetzt. Es ist deshalb nicht notwendig, diese Dateien mit abzugeben.

Als Referenzplattform dient `pluto.tugraz.at` [6]. Vergewissern Sie sich also **bevor Sie abgeben**, dass Ihre Implementierung dort funktioniert!

4.2 Dokumentation und Coding-Style

Der abgegebene Code soll eine Inline-Dokumentation (Englisch) enthalten, welche die grundsätzliche Funktionsweise erklärt. Es muss nicht jede

einzelne Codezeile dokumentiert sein. Stellen Sie sich nur immer die Frage, ob ein Außenstehender Ihren Code auch verstehen würde.

Sollten Dinge dennoch wichtig sein, die nicht in die Inline-Dokumentation passen, schreiben Sie diese in der Datei `README.txt` zusammen (Deutsch oder Englisch). Jedenfalls **muss** die Readme-Datei eine Liste aller implementierten Aufgabenteile (Base-64-Enkodierung, Base-64-Dekodierung, LFSR-Verschlüsselung) enthalten. Auch Fehler, die Sie bereits zum Zeitpunkt der Abgabe kennen, müssen in der Readme-Datei angegeben werden.

Achtung: Für die Konstruktionsübung gibt es keinen eigenen Coding-Standard oder ähnliche Vorgaben. Trotzdem wird von Ihnen erwartet, dass der Code leserlich und verständlich ist.

4.3 Abgabegespräche

Das Abgabegespräch für Assignment A2 findet zusammen mit dem Abgabegespräch für Assignment A1 statt. Termine dafür werden in der Newsgroup bekanntgegeben.

Achtung: Das Journal, in dem Sie Überlegungen zu den Assignments A1 und A2 (handschriftlich) aufgezeichnet haben, ist dabei mitzubringen und vorzuweisen.

4.4 Gruppenbildung für Assignment A3

Das Assignment A3 soll in Gruppen zu je 3 Studenten bearbeitet werden. Die Anmeldung der Gruppen erfolgt über eine entsprechende Maske auf der RNO Webseite [1]. Sie werden durch eine Ankündigung in der Newsgroup informiert, sobald die Gruppenmeldung freigeschaltet ist. Die Meldung Ihrer Gruppe muss bis spätestens Freitag, 30. April 2010 erfolgen.

Literatur

[1] TU Graz IAIK: Rechnernetze und -Organisation, Website, http://www.iaik.tugraz.at/content/teaching/bachelor_courses/rechnernetze_und_organisation/, SS 2010.

[2] RFC 4648: The Base16, Base32, and Base 64 Data Encodings, Website, http://tools.ietf.org/html/rfc4648, October 2006.

[3] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone: Handbook of Applied Cryptography, Chapter 6 - Stream Ciphers, PDF, http://www.cacr.math.uwaterloo.ca/hac/about/chap6.pdf, 1996.

[4] Norman Matloff: Introduction to Linux Assembly Language, PDF, http://www.iaik.tugraz.at/content/teaching/bachelor_courses/rechnernetze_und_organisation/downloads/02_LinuxAssembly.pdf, March 2007.

[5] Norman Matloff: Subroutines on Intel CPUs, PDF, http://www.iaik.tugraz.at/content/teaching/bachelor_courses/

rechnernetze_und_organisation/downloads/07_Subroutines.pdf, March 2007.

[6] Zentraler Informatikdienst der Technischen Universität Graz: Linux-Server pluto.TUGraz.at, http://www.zid.tugraz.at/allgemein/pluto.html.

[7] VMware: Virtualization Software, Website, http://www.vmware.com/.

[8] VirtualBox, Website, http://www.virtualbox.org/.