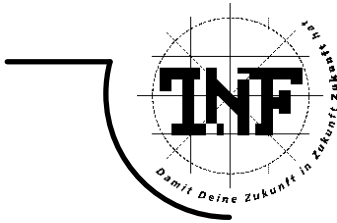




JOHANNES KEPLER
UNIVERSITÄT LINZ

Netzwerk für Forschung, Lehre und Praxis



Entwicklung und Implementierung eines SMTP/POP Clients unter konzeptioneller Integration symmetrischer Kryptographie

BAKKALAUREATSARBEIT
(Angewandte Kryptographie)

zur Erlangung des akademischen Grades

BAKKALAUREUS DER TECHNISCHEN WISSENSCHAFTEN

in der Studienrichtung

TECHNISCHE INFORMATIK

Angefertigt am *Institut für Computational Perception*

Betreuung:

a. Univ.-Prof. Dr. Josef Scharinger

Eingereicht von:

Sonnleitner Erik

Linz, Mai 2007

Entwicklung und Implementierung eines
SMTP/POP Clients unter konzeptionaler
Integration symmetrischer Kryptographie

Erik Sonnleitner

25. April 2007

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 1 | Einleitendes | 4 |
| 1.1 | Motivation | 4 |
| 1.2 | Danksagung | 5 |
| 1.3 | Terminologie | 6 |
| 1.4 | Abkürzungsverzeichnis | 7 |
| 2 | Kryptographische Grundlagen | 8 |
| 2.1 | Warum Kryptographie? | 8 |
| 2.2 | Grundgedanken der symmetrischen Kryptographie | 9 |
| 2.3 | Symmetrische versus asymmetrische Algorithmen | 14 |
| 2.4 | Symmetrische Blockchiffren | 15 |
| 2.4.1 | Rijndael - der <i>Advanced Encryption Standard</i> | 16 |
| 2.4.2 | Serpent | 17 |
| 2.4.3 | Twofish | 18 |
| 2.5 | Erzeugung großer Primzahlen | 18 |
| 2.6 | Sicherer Schlüsseltausch nach Diffie Hellman | 19 |
| 2.7 | Längen symmetrischer Schlüssel | 23 |
| 2.8 | Einweg-Hashfunktionen | 24 |
| 3 | Kommunikationsstrukturen via POP und SMTP | 26 |
| 3.1 | Postfix als SMTP Server | 26 |
| 3.2 | Protokollkoordination durch SMTP | 27 |

| | | |
|----------|--|-----------|
| 3.3 | Protokollkoordination durch POP3 | 28 |
| 3.4 | Einbindung von öffentlichen Servern in SMC | 29 |
| 4 | Implementierung von SMC | 31 |
| 4.1 | Systemumgebung | 31 |
| 4.2 | Elementare Protokollvorgaben | 32 |
| 4.3 | Das SMC Protokoll | 32 |
| 4.4 | Realisierung des Schlüsseltausches | 35 |
| 4.5 | Regeneration von Schlüsseln | 36 |
| 4.6 | Sicherstellung der Authentizität | 37 |
| 4.7 | Clientseitige Sicherheit | 37 |
| 4.8 | Algorithmische Modularität | 38 |
| 4.9 | Obligate CPAN Bibliotheken | 39 |
| 5 | Anwendung von SMC | 40 |
| 5.1 | Basiskonzepte | 40 |
| 5.2 | Das Hauptmenü | 41 |
| 5.3 | Senden, Empfangen und Zurückschicken von Einladungen | 42 |
| 5.4 | Senden verschlüsselter Nachrichten | 43 |
| 5.5 | Entschlüsselung von empfangenen Nachrichten | 44 |
| 5.6 | Sonstige Funktionalitäten | 45 |
| 6 | Ausblick | 46 |
| A | Die Module von SMC | 47 |
| B | Konfigurationsdateien von SMC | 49 |
| C | Auszüge aus dem Quellcode von SMC | 51 |
| C.1 | Der Schlüsseltausch - Generierung der Einladung | 51 |
| C.2 | Dekodierung einer SMC Einladung | 52 |

D Literaturverzeichnis

53

Kapitel 1

Einleitendes

1.1 Motivation

Trotz dem massiven Einsatz von E-Mails und einer stetig wachsenden Anzahl von Internetzugängen, bleibt die Sicherheit und somit die Privatsphäre der Benutzer oftmals auf der Strecke – nicht zuletzt aufgrund der für den Großteil der Anwender inakzeptablen Komplexität von asymmetrischer Kryptographie (welche z. B. durch Werkzeuge wie *PGP* realisiert wird).

Als Kompromisslösung zwischen Simplizität und der Aufrechterhaltung der Privatsphäre, wurde das Werkzeug SMC (*secure mail client*) konzipiert und implementiert, welches die von einem E-Mail-Client zu erwartende Grundfunktionalität modular um symmetrische Kryptographie als integralen Bestandteil erweitert und für den Benutzer dabei vollkommen transparent bleibt.

Anforderungsdefinition:

- Sicherer E-Mailverkehr durch symmetrische Verschlüsselungsalgorithmen
- Programmatisch autonomer Schlüsseltausch über *Diffie-Hellmann*
- Aufrechterhaltung der Authentizität
- Sicheres Schlüsselmanagement
- Sichere Schlüsselgenerierung, sowie -regenerierung

- Einfache Bedienbarkeit völlig ohne kryptographisches und/oder mathematisches Vorwissen

1.2 Danksagung

Dank gebührt insbesondere a. Univ. Prof. Dr. Josef Scharinger der mir sowohl inhaltlich als auch zeitlich genügend Freiraum bot um diese Arbeit zu verwirklichen, und dessen vorzügliche Lehrveranstaltung über Angewandte Kryptographie die Basis für das vorliegende Dokument darstellt.

Darüber hinaus danke ich meinen Eltern, ohne deren Unterstützung mein Studium nicht möglich wäre.

Speziellen Dank gebührt auch Magdalena, meiner treuen Begleiterin.

1.3 Terminologie

Um eine einheitliche terminologische Notation zu gewährleisten sind an dieser Stelle die wichtigsten im Dokument verwendeten Variablen sowie kryptografischen Funktionen kurz erläutert.

| Symbol | Semantik |
|-----------|---|
| k_i | Schlüssel i (<i>key</i>) |
| $ \phi $ | Länge des Objekts ϕ (z.B. Schlüssellänge $ k $) |
| M | Nachricht (<i>message</i>) |
| b | Block |
| E | Chiffrierungsfunktion (<i>to encrypt</i>) |
| D | Dechiffrierungsfunktion (<i>to decrypt</i>) |
| r_i | Runde i |
| w_i | Wort i |
| A, B | i.d.R. Sender und Empfänger, auch <i>Alice</i> und <i>Bob</i> |
| Σ | Verwendetes Alphabet über einer Funktion f |
| I | Eingehende Daten (<i>input data</i>) |
| O | Ausgehende Daten (<i>output data</i>) |
| $P(f(x))$ | Wahrscheinlichkeit des Eintretens von $f(x)$ |

1.4 Abkürzungsverzeichnis

Die Informationstechnik bedient sich seit jeher einer geraumen Fülle an Akronymen. Folgende Tabelle soll Aufschluss über die verwendeten Abkürzungen geben.

| Akronym | |
|---------|---|
| SMC | Symmetric mail client |
| RFC | Request for comments, techn. Dokumente bzgl. Rechnernetze |
| POP | Post office protocol |
| SMTP | Simple mail transfer protocol |
| IMAP | Internet message access protocol |
| IP | Internet protocol |
| TCP | Transmission control protocol |
| FTP | File transfer protocol |
| SSL | Secure socket layer |
| SASL | Simple authentication and security layer |
| ISO | International standards organisation |
| NIST | National institute of standards and technology |
| OSI | Open systems interconnection |
| ARP | Address resolution protocol |
| DNS | Domain name service |
| RIP | Routing information protocol |
| OSPF | Open shortest path first |
| CBC | Chipher block chaining |
| EBC | Electronic codebook (mode) |
| CFB | Cipher feedback (mode) |
| OFB | Output feedback (mode) |
| XOR | Exclusive Or, auch \oplus |
| DES | Data encryption standard |
| AES | Advanced encryption standard |
| WEP | Wireless equivalent privacy |
| HTTP | Hyper text transfer protocol |
| HTTPS | Secured hyper text transfer protocol |

Kapitel 2

Kryptographische Grundlagen

2.1 Warum Kryptographie?

Seit Anbeginn der Zeit ist es der Menschheit ein Anliegen ihre Daten zu verschlüsseln. Um ca. 50 v.Chr. verwendete Julius Cäsar zur Geheimhaltung militärischer Korrespondenz eine einfache Verschiebechiffre, und dabei war er keineswegs der Erste welcher diesen Gedankengang hegte.

Beginnend mit einfachen Transpositionschiffren, über mono- und polyalphabetische Kryptosysteme, entwickelte sich die Kunst der Verschlüsselung über Jahrtausende und erreichte ein lokales Maximum während des zweiten Weltkriegs unter dem Einsatz der legendären *Enigma*.

Die darauffolgende kryptographische Revolution wurde wesentlich durch den Einbruch des Computerzeitalters geprägt. Dabei koexistiert das Wissensgebiet der Kryptographie mit der rechnergestützten Umwelt und deren daraus resultierenden globalen Vernetzung von Information in einer Symbiose.

Nicht dass das Informationskonglomerat (landläufig als Internet bezeichnet) im Allgemeinen, und dessen Datentransfer im Speziellen, Kryptographie zwingend benötigen würde – die tatsächlichen Möglichkeiten würden jedoch großer Einschränkungen unterliegen.

Popularisierte Systeme wie eCommerce und eBanking würden auch ohne Einsatz von Verschlüsselung funktionieren, aber aufgrund von massiven Sicherheitsbedenken keine Anwendung finden. Auch Cäsars Botschafter hätten Mitteilungen aus höchster

Instanz im Klartext überbringen können – einzig und allein die *Möglichkeit*, dass die Information an falsche Adressaten kommen oder möglicherweise sogar manipuliert werden könnte, veranlasste Cäsar zu seiner Entscheidung.

So befinden wir uns heute auf einem semi-kryptographischen Pfad der Vernetzung. Das Internet ist jung (jedenfalls im Vergleich zur zeitgeschichtlichen Entfaltung der Chiffretechniken), jedoch alt genug um unter der kryptographischen Sorglosigkeit der 70er-Jahre immer noch leiden zu müssen.

Dabei ist der heutzutage i.A. verwendete Protokollstapel des Internets trotz seines Alters von monumentaler Eleganz, zeigt aber erste Schwächen, wenn man Informationssicherheit als Kriterium heranzieht – grundsätzlich findet von jeher sämtlicher Datentransfer unverschlüsselt statt.

Zahlreiche Protokolle der ISO OSI Schicht 7 haben seither Neuerungen erfahren, wie beispielsweise HTTPS als SSL-Erweiterung des *Hypertext transfer protocols* welches zugleich eines der am meisten verwendeten ist. Andere Protokolle wurden redefiniert und/oder erweitert, finden jedoch kaum Anwendung - darunter sind beispielsweise sFTP und IMAP.

Die meisten Protokolle niederer Schichten müssen bis dato vollkommen ohne Verschlüsselung auskommen, wie beispielsweise IP, X.25, ARP, DNS oder diverse Routingprotokolle wie RIP oder OSPF.

Durch die undurchschaubare Größe, dem stetigen Wachstum und der Unkontrollierbarkeit des Internets ist in dieser Hinsicht auch in nächster Zeit nur mit punktuellen Änderungen zu rechnen. Aus diesem Grund wollen wir uns hierbei auf die Sicherung von den für diese Arbeit relevanten Userland-Protokollen beschränken - POP und SMTP.

2.2 Grundgedanken der symmetrischen Kryptographie

Betriebsmodi und Funktionsweise von Blockchiffren

Blockchiffren Als *Blockchiffre* bezeichnen wir Verschlüsselungsalgorithmen, welche den eingehenden Klartext in immer gleich langen vordefinierten Blöcken von Daten (meinst Byteblöcken) verarbeiten und daraus den Chiffretext erzeugen. Die

meisten Vertreter moderner symmetrischen Verschlüsselungsalgorithmen nutzen die Funktionsweise der Blockchiffrierung; bekannte Beispiele hierfür sind der DES (der *Data encryption standard*) sowie Rijndael (der *Advanced encryption standard*).

Padding Nehmen wir an, der Klartext M besteht aus 125 Zeichen des erweiterten ASCII Zeichensatzes, also bestehend aus 8 Bit langen Zeichen, so besteht dieser Datenstrom aus 1000 Bits, welche es zu verschlüsseln gilt. Bei der Chiffrierung unter Verwendung einer Blockgröße von 64 Bit ergeben sich nun 15 ganze 64 Bit Blöcke und ein Rest von 40 Bit. Da ein Blockchiffrierungsalgorithmus jedoch immer Daten derselben einheitlichen Blockgröße erwartet, müssen Regeln definiert werden, welche das letzten Datenpaket (welches in unserem Fall lediglich 40 Bit aufweist) auf die benötigten 64 Bit expandiert, was wir als *Padding* oder *Auffüllen* bezeichnen.

Betriebsmodi Aus der Verarbeitungsweise von Blockchiffren lassen sich nun mehrere sogenannte *Betriebsmodi* ableiten, welche sich in der Verarbeitung einzelner Blöcke und der oftmals notwendigen Expandierung des letzten Blocks bei Klartexten, welche kein Vielfaches der Blockgröße aufweisen, unterscheiden [Sch05].

- Der *Electronic Codebook Mode* (ECB) ist die einfachste Realisierung einer Blockchiffre. Dabei wird ein bestimmter Klartextblock immer zu dem genau selben Chiffreblock überführt. Die Einfachheit dieses Modus' lässt es zu, dass ein vorgegebener Datenstrom nicht zwingend linear von Anfang bis Ende verschlüsselt werden muss – es ist möglich, die Blöcke des Klartextes in beliebiger Reihenfolge zu kodieren und dekodieren. Im speziellen lässt sich der Einsatz von parallelen Systemen sehr effizient realisieren.

Dabei ist ECB äußerst fehlertolerant, zumal ein fehlerhaft übertragenes Bit lediglich den aktuell dechiffrierten Block beschädigt, jedoch keinen Einfluss auf die anderen Blöcke nimmt.

Geht hingegen ein Bit bei der Übertragung verloren (oder es wird ein zusätzliches Bit eingefügt), zieht sich diese Fehlerfortpflanzung bis zum Ende hin, da immer Blöcke derselben Größe verarbeitet werden.

Das Padding funktioniert bei ECB mittels simplem Auffüllen von zumeist Eins- oder Nullfolgen bis zum nächsten Vielfachen der Blockgröße. Um bei der Entschlüsselung sicherzustellen, wieviel Bits nun zum eigentlichen Klartext gehören und welche Füllbits sind, wird im letzten künstlich angefügten Byte die Anzahl der notwendigen Füllbits gespeichert.

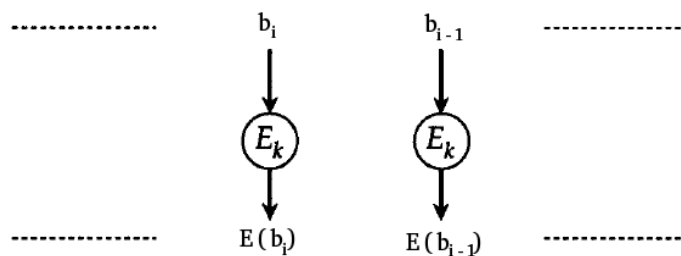


Abbildung 2.1: Der Electronic Codebook Mode

- Ein weiterer als *Cipher Block Chaining* (CBC) bezeichneter Betriebsmodus arbeitet mit einer iterativen Fortplantungsstrategie, wobei ein Chiffreblock nicht nur aufgrund des gegebenen Basisschlüssels ermittelt wird, sondern auch das Resultat der Verschlüsselung des vorangegangenen Blocks enthält.

Dabei wird im CBC Modus der Klartext eines Blocks *vor* der eigentlichen Verschlüsselung mit dem vorhergehenden Chiffreblock mittels XOR verknüpft¹, es gilt also die rekursive Definition

$$E(b_0) = IV \wedge \forall(i \in \mathbb{N}^+) : E(b_i) = E_k(b_i \oplus E(b_{i-1}))$$

Der Vorteil dieser Methode liegt darin, dass zwei identische Klartextblöcke verschiedene Chiffreblöcke ergeben. Da dies jedoch nur der Fall ist, wenn zwei Blöcke unterschiedliche Vorgänger haben (identische Nachrichten würden ansonsten identischen Chiffretext erzeugen), verwendet CBC einen sogenannten *Initialisierungsvektor*. Dieser besteht nach Möglichkeit aus reinen Zufallsdaten der Länge der Blockgröße $|b|$ und wird nur zur Verschlüsselung des ersten Klartextblocks einer Nachricht verwendet (da dieser keinen Vorgänger besitzt).

Im Gegensatz zu ECB leidet CBC unter einer stetigen Fehlerfortpflanzung. Sollte ein solcher einmal im Strom auftauchen, bleibt der Rest der Nachricht unbrauchbar, da jeder kommende Block direkt oder indirekt vom aktuellen Chiffreblock abhängt.

¹welcher aus diesem Grund nach jeder Iteration eines Blocks in ein Rückkopplungsregister zwischengespeichert wird

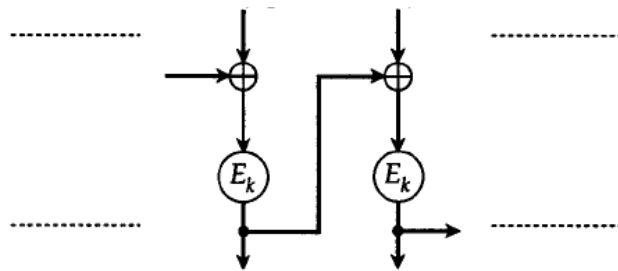


Abbildung 2.2: Cipher Block Chaining

Es gibt noch zahlreiche weitere Blockmodi, wie beispielsweise den *Cipher Feedback Mode* oder den *Output Feedback Mode* welche vorrangig bei Stromchiffren eingesetzt werden, auf welche wir im weiteren nicht näher eingehen möchten.

Stärken und Schwächen der XOR-Verknüpfung

Vom Grundgedanken zur Verschlüsselung Ein fundamentales Grundprinzip jeder modernen Blockchiffre stellt die binäre XOR-Operation dar. Ein wesentliches Merkmal bei der Bearbeitung von digitalen Daten ist die Tatsache, dass sich jegliche digital repräsentierbare Information in geordneten Mengen von Nullen und Einsen darstellen lässt (genauer gesagt, darstellen lassen muss). Diese Diskretisierung der vorhandenen Information erlaubt uns, sie in kleinste Einheiten – den Bits – aufzuteilen und zu verarbeiten.

Die Bool'sche Algebra definiert eine gewisse Grundmenge an binären Operationen, wie beispielsweise Und- und Oder-Verknüpfungen. Ein Spezialfall unter diesen bildet die so genannte XOR-Verknüpfung, auch als *exclusive or* bezeichnet. Sie stellt, wie alle übrigen binären Bool'schen Operationen auch, eine Funktion von zwei Bits dar, welche ein einziges Bit als Resultat liefert. Versuchen wir nun, zwei beliebige Bits mittels XOR zu verknüpfen, so liefert diese Funktion *false* beziehungsweise 0, falls beide zu verarbeitenden Eingabebits den gleichen Wert aufweisen, andernfalls jedoch *true* beziehungsweise 1.

Der große Vorteil der XOR-Operation beruht auf der natürlichen Symmetrie: Nehmen wir an, wir üben auf Bit a und Bit b eine XOR-Operation aus, mit dem Resultat von Bit c , so kann Bit a leicht durch die abermalige XOR-Verknüpfung von Bit b mit Bit c (also unserem vorherigen Ergebnis) wiedererrechnet werden. Folgt man diesem elementaren Gedanken, ist es leicht erkennbar, dass auch das Bit b durch eine XOR-Verknüpfung von Bit a mit Bit c errechnet werden kann.



| A | B | Out |
|---|---|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Abbildung 2.3: Bool'sche Wahrheitstabelle und Schaltsymbol der XOR-Verknüpfung

Genau diese triviale wie elegante Funktionalität binärer Operationen nutzt die symmetrische Kryptographie zu ihren Gunsten. Wenn wir nun davon ausgehen dass es sich beim besagten Bit a um einen Teil des zu verschlüsselnden Klartextes handelt und bei Bit b um den eigentlichen Schlüssel, so kann uns die XOR-Operation dieser beiden Elemente auf einfache Weise eine mögliche Verschlüsselung des eingehenden Klartextes liefern (korrespondierend zum obigen Beispiel also das Bit c).

Schwächen der XOR-Operation Unglücklicherweise genügt es jedoch nicht, den Klartext bitweise mit einer Endlosfolge eines vordefinierten Schlüssels durch XOR-Verknüpfungen in Chiffretext zu verwandeln. Wäre dies der Weisheit letzter Schluss, so wären sämtliche Ver- wie auch Entschlüsselungsvorgänge ungemein schnell und effizient – und nebenbei hätten wir somit den möglicherweise besten wie auch trivialsten Verschlüsselungsalgorithmus entdeckt.

Diese Vorgehensweise würde mit relativ einfachen Mitteln unbrauchbar gemacht werden können, in dem man den erzeugten Chiffretext bitweise mit sich selbst einer XOR-Verknüpfung unterzieht, wobei man die beiden Bitfolgen gegeneinander byteweise verschiebt. Man unterzieht das Ergebnis dieser Operation nun einem Vergleich, welche Bytes sich im Gegensatz zu Ihrem Ursprung verändert haben und welche gleich geblieben sind.

Dieser Vorgang wird mit jeweils unterschiedlichen Längen der Verschiebung sooft wiederholt bis zu erkennen ist, dass der prozentuelle Anteil der gleich gebliebenen Bytes bei einer bestimmten Längenverschiebung ein (bei weitem eindeutiges) Maximum erreicht – die Chancen, nun die Schlüssellänge gefunden zu haben stehen also äußerst gut. Hat man erst die Länge des Schlüssels herausgefunden, so Bedarf es nur mehr einer XOR-Verknüpfung der verschlüsselten Bitfolge mit sich selbst un-

ter Bitverschiebung der Schlüssellänge. Diese Vorgehensweise hebt sich laut [Sch05] aufgrund der Gesetze der mathematischen Logik den Schlüssel wieder auf und das Resultat der Operation zeigt uns den Klartext.

Lösungsansätze Um Verschlüsselungsalgorithmen als sicher gelten zu lassen, wird in den meisten Fällen mit Permutationen, Transformationen und Substitutionen auf Basis von n -Tupel großen Klartextteilblöcken gearbeitet welche auf den Klartext unter Einbezug des Schlüssels angewandt werden. Oftmals ziehen sich diese Prozesse über mehrere Iterationen (*Runden*) hinweg, was die Sicherheit des Chiffrats aufgrund der auftretenden Konfusion und Diffusion wesentlich erhöht.

2.3 Symmetrische versus asymmetrische Algorithmen

Im Gegensatz zur symmetrischen Kryptographie, zeichnen sich asymmetrische Algorithmen durch die Existenz eines *Schlüsselpaars* aus, wobei i.d.R. ein Schlüssel zur Ver- und ein Schlüssel zur Entschlüsselung verwendet wird.

Asymmetrische (oder auch Public-Key-) Kryptosysteme beruhen im Allgemeinen auf der *Annahme*, dass sich sehr große Zahlen nur mit enormen Aufwand in ihre Primfaktoren zerlegen lassen ², was auch eine der größten vermuteten Schwächen dieser Variante ist, zumal es bis dato keinen mathematischen Beweis gibt, welcher ein Minimum des zu erbringenden Aufwandes zur Zerlegung einer beliebigen Zahl in ihre Primfaktoren definieren würde. Anders ausgedrückt, wäre es durchaus denkbar, dass fähige Mathematiker in absehbarer Zeit einen wesentlich effizienteren Weg finden, Primfaktoren zu ermitteln.

Die heutzutage bestmöglichen Algorithmen zur Primfaktorzerlegung stellen das *Quadratische Sieb* von Carl Pomerance und das *Zahlenkörpersieb* von Hendrik W. Lenstra dar. Des weiteren beschäftigt sich die *RSA Factoring Challenge* intensiv mit der Forschung nach effizienten Faktorisierungsverfahren, was im Jahre 2005 zur Faktorisierung der bisher größten Semiprimzahl, bestehend aus 200 Dezimalstellen, durch das *Gittersieb* führte (zahlreiche weitere Faktorisierungsmethoden werden in [Rie94] erläutert).

²Die Rede ist hier vom Faktorisierungsproblem; neben dem diskreten Logarithmus-Problem eine der großen mathematischen Herausforderungen der modernen Kryptographie

Neben den rein mathematischen Bedenken gibt es Probleme, welche asymmetrische Kryptosysteme in der Praxis als schwierig einsetzbar klassifizieren. Beispielsweise würde jeder Sender sowie Empfänger ein eigenes unikates Schlüsselpaar benötigen. Darüber hinaus müssen sämtliche öffentliche Schlüssel der Teilnehmer jederzeit abrufbar sein, um Nachrichten verschlüsselt versenden zu können.

Einige Anwendungsgebiete der Informatik würden asymmetrische Verschlüsselung bereits aufgrund Ihrer elementaren Struktur nicht erlauben. Man bedenke hierbei die Implementierung eines Multicast-Services innerhalb eines heterogenes Netzes: der Service müsste in nahe-Echtzeit jedes Paket aller Adressaten mit deren öffentlichen Schlüsseln chiffrieren. Eine Annahme, welche allein durch die Definition von Multicastnetzen bereits keinen Halt finden kann.

Darüber hinaus verlangt die Verschlüsselung mit asymmetrischen Systemen einen sehr viel größeren Aufwand (und damit mehr Zeit) als jene mit symmetrischen Systemen. Aus diesem Grund werden in der Praxis keine reinen Public-Key-Systeme verwendet, sondern sogenannte *hybride Kryptosysteme* welche i.d.R. lediglich symmetrische Schlüssel asymmetrisch chiffrieren und die eigentliche Nachricht M wiederum symmetrisch verschlüsseln.

Ogleich die Erfindung der asymmetrischen Kryptographie ohne Zweifel als eine Revolution auf diesem Gebiet angesehen werden kann, verliert die symmetrische Kryptographie keineswegs ihre Daseinsberechtigung.

2.4 Symmetrische Blockchiffren

Wir wollen nun auf einige wenige essentielle in SMC verwendete symmetrische Blockchiffren näher eingehen. Der nach DES, welcher heute als hinreichend unsicher angesehen werden kann, meistverwendete Chiffrierer ist *Rijndael* der zum *Advanced Encryption Standard* erhoben wurde. Aus diesem Grund werden wir die genaue Spezifikation sowie Funktionsweise dieses Verfahrens näher betrachten, und erläutern in Folge die Blockchiffren *Serpent* (Anderson, Biham, Knutsen; [ABK98]) und *Twofish* (Schneier; [SKW⁺98]) nur am Rande.

2.4.1 Rijndael - der *Advanced Encryption Standard*

Grundsätzliche Merkmale von Rijndael Rijndael [DR01] zeichnet sich dadurch aus, dass er Information mit verschiedenen Schlüssel- sowie Blocklängen verarbeiten kann, wobei Schlüssel und Block jeweils 128, 196 oder 256 Bit aufweisen können. Auch Rijndael nutzt das bereits von DES bekannte *Rundenprinzip* (Feistelchiffre), iteriert jeden zu verarbeitenden Bitblock also rundenweise bis er fertig verschlüsselt ist. Die Anzahl der Runden, welcher ein Block durchläuft, ist jedoch von Schlüssel- und Blocklänge wie folgt abhängig:

| Runden | $ b = 128$ | $ b = 192$ | $ b = 256$ |
|-------------|-------------|-------------|-------------|
| $ k = 128$ | 10 | 12 | 14 |
| $ k = 192$ | 12 | 12 | 14 |
| $ k = 256$ | 14 | 14 | 14 |

Der Algorithmus Ist die Blocklänge $|b|$ erst spezifiziert, konstruiert Rijndael für jeden Block eine Byte-Matrix als Funktion von $|b|$. Bei einer Blocklänge von 128 Bit wird eine 4 x 4 Matrix erzeugt (4 x 4 Bytes = 128 Bit), bei 192 Bit Blocklänge eine 4 x 6 Matrix, und bei Blocklängen von 256 Bit eine 4 x 8 Matrix. Diese Matrix wird während des gesamten Algorithmus als *Zustand* oder auch *Zustandsmatrix* bezeichnet, da sämtliche Operationen auf dieser basieren (beziehungsweise diese Matrix modifizieren).

Ähnlich dem DES nutzt Rijndael ein Verfahren, welches für jede Runde eigene (unikate) Teilschlüssel verwendet. Diese werden vor Beginn der eigentlichen Rundeniterationen durch die *Schlüsselexpansion* erzeugt, der Originalschlüssel wird also erweitert. Da Rijndael bereits *vor* der ersten Runde den Klartextblock mittels XOR mit einem Teilschlüssel verknüpft, werden jedoch

$$\sum |r| + 1$$

Teilschlüssel benötigt, also einer mehr als Runden pro Block iteriert werden.

Der ursprüngliche Klartextblock wird nun vor Beginn der ersten Runde mit dem ersten Teilschlüssel k_0 (welcher dem Originalschlüssel entspricht) durch XOR verknüpft und durchläuft erst im Anschluss die eigentliche erste Runde.

Darüber hinaus basiert Rijndael auf drei Operationen, welche in jeder Runde auf den Block ausgeführt werden, jeweils mit einem anderen Rundenschlüssel. Diese

Operationen sind i.A. unter *ByteSub-Transformation*, *ShiftRow-Transformation* und *MixColumn-Transformation* bekannt und vollziehen die unter Rijndael verwendeten Substitutionen und Permutationen.

Hat ein Block alle drei Operationen durchlaufen, so folgt die XOR-Verknüpfung mit dem nächsten Rundenschlüssel k_{i+1} . Nach der Finalrunde wird dies nochmals mit dem letzten Teilschlüssel ausgeführt, bevor der Block als fertig verschlüsselt gilt.

2.4.2 Serpent

Serpent ist eine symmetrische Blockchiffre, entwickelt von Ross Anderson, Eli Biham und Lars Knudsen. Die Originalspezifikation zu Serpent ist unter [ABK98] nachzulesen.

Serpent arbeitet ebenfalls rundenbasiert und verwendet dabei 32 Iterationen. Als Schlüssellänge $|k|$ werden stets 256 Bit verwendet, wobei grundsätzlich auch Längen $|k| \leq 256$ Akzeptanz finden, welche jedoch durch Padding auf 256 Bit expandiert werden. Die Größe der jeweils einzeln zu chiffrierenden Blöcke beträgt 128 Bit.

Der Algorithmus Nach einer initialen Eingangspermutation ähnlich des DES iteriert Serpent 32 Runden $r_i, 0 < i < 31$, wobei die letzte Runde r_{31} leicht abgewandelt ist. In jeder Runde wird der 128-Bit Textblock mit einem 128-Bit Rundenschlüssel k_i mittels XOR verknüpft und im Anschluss in 32 4-Bit Worte aufgeteilt.

Die vorliegenden 32 separierten Worte werden nun parallel in acht S-Boxen (jeweils in 32-facher Ausführung implementiert) geschickt wo die ursprünglichen Werte eine Substitution durchlaufen. Der Algorithmus sieht vor, dass jede S-Box im Laufe der 32 Runden genau vier mal verwendet wird. Nach der Substitution werden alle 4-Bit Rückgabewerte der S-Boxen wieder zu einem Textblock zusammengefügt und in den Runden $r_0 - r_{30}$ einer linearen Transformation unterzogen.

In Runde r_{31} wird die lineare Transformation durch eine weitere XOR Operation mit dem Rundenschlüssel k_{32} ersetzt (k_{31} wurde bereits vor den Substitutionen der Runde r_{31} verwendet).

Die Konstruktion der S-Boxen ist dabei sehr an den von Ron Rivest entwickelten Blockchiffrierer RC4 angelehnt [Riv94], welcher u.a. im WEP-Protokoll [PF02] für Wireless LANs Verwendung findet.

2.4.3 Twofish

Twofish bildet einen symmetrischen Blockchiffrierer vom Kryptographen Bruce Schneier und wurde 1998 als Nachfolger von Blowfish ([Sch94]) veröffentlicht. Die Originalspezifikation ist unter [SKW⁺98] nachzulesen.

Twofish verwendet Schlüssellängen $|k| \in \{128, 192, 256\}$ Bit und iteriert dabei 16 Runden bei einer fixen Blocklänge von 128 Bit. Ähnlich wie Serpent akzeptiert auch Blowfish Längen $|k| \leq 256 \wedge |k| \neq 128 \wedge |k| \neq 192 \wedge |k| \neq 256$ Bit, diese werden jedoch bei Bedarf wiederum über Paddingalgorithmen auf die nächstgrößere legale Blocklänge erweitert.

2.5 Erzeugung großer Primzahlen

Primzahlen gelten als die Essenz der asymmetrischen Kryptographie und bilden somit auch die Grundlagen für den Diffie-Hellman-Schlüsseltausch, zumal dieser ein ähnliches Konzept verfolgt.

Der Stand der Mathematik zum Beginn des 21. Jahrhunderts erlaubt es uns nicht n -stellige Primzahlen direkt zu generieren. Um trotzdem Primzahlen erzeugen zu können, werden im Normalfall Formeln verwendet welche es erlauben aufgrund der spezifischen Eigenschaften von Primzahlen eine Zahl pseudo-zufällig zu generieren welche mit einer gewissen prozentuellen Wahrscheinlichkeit als prim angesehen werden darf. Bereits der große Mathematiker *Euler* nannte die Formeln

$$n^2 + n + 17, 0 < n < 16 \wedge n \in \mathbb{P}$$

sowie

$$n^2 - n + 41, 0 < n < 41 \wedge n \in \mathbb{P}$$

.

Heutzutage stehen zur Generierung von möglichen Primzahlen beispielsweise der

- *Lehmann*-Algorithmus mit $P(p \in \mathbb{P}) \geq 50\%$, der
- *Rabin-Miller*-Algorithmus mit $P(p \in \mathbb{P}) = 75\%$, und der
- *Solovay-Strassen*-Algorithmus mit $P(p \in \mathbb{P}) \geq 50\%$

zur Verfügung, wie in [Sch98] ausführlich diskutiert.

Um zu verifizieren dass diese Zahl wirklich eine Zahl aus der Menge der Primzahlen ist, muss sie nach der Generierung trotzdem auf ihre möglicherweise existierende Primalität hin überprüft werden. Dies kann mittels einfachen Algorithmen überprüft werden. Der AKS-Primzahltest [AKS04] beweist darüber hinaus, dass sich die Primalität einer Zahl in polynomialer Laufzeit beweisen lässt.

2.6 Sicherer Schlüsseltausch nach Diffie Hellman

Grenzen der Symmetrie

Die Notwendigkeit der Verschlüsselung von Information im Zweiten Weltkrieg veranlasste das deutsche Militär letztendlich zum Einsatz der weltbekannten *Enigma*-Maschine. Sie wurde von *Arthur Scherbius* entwickelt und realisierte ein elegantes Konzept zur Verschlüsselung von Texten mittels polyalphabetischer Chiffrierung. Das Prinzip der Enigma beruhte auf der Voreinstellung mehrere Walzen auf welchen sich Buchstaben in alphabetischer Reihenfolge befanden.

Ogleich auch diese Verschlüsselung letztendlich gebrochen werden konnte³, war das größte Problem die Verteilung der Schlüssel.

Dies ist keine spezielle Eigenschaft der Enigma, sondern betrifft alle symmetrischen Verschlüsselungsvarianten, da zur Ver- und Entschlüsselung derselbe Schlüssel benötigt wird. In den Zeiten des Zweiten Weltkrieges war man von jegliche weltumspannender Vernetzung weit entfernt und da ein verschlüsselter Funkspruch von zahlreichen Empfangsstationen empfangen und entschlüsselt werden konnte, mussten all diese Stationen den korrekten Schlüssel besitzen.

Aus diesem Grund wurden *Schlüsselbücher* angefertigt welche bei feindlichen Truppen hoch im Kurs standen und eine Reihe von Schlüsseln enthielten die in gewisser Regelmäßigkeit (meist täglich) gewechselt wurden um die Arbeit von Kryptanalytikern zu erschweren.

Das Konzept der Schlüsselbücher erfüllte zwar seinen Zweck, beseitigte das Problem des Schlüsseltausches aber nicht. Eine Problematik mit der sich der Mathematiker

³Was jedoch nicht ausschließlich durch die Konzeption der Enigma möglich war, sondern durch das Verwenden häufiger kurzer Nachrichten der Deutschen

Whitfield Diffie in den frühen Siebzigern auseinandersetzte. Über einen von IBM initiierten Vortrag im Jahr 1974 kam Diffie zu *Martin Hellman*, ein Professor der kalifornischen Stanford-Universität.

Gemeinsam mit *Ralph Merkle* entwickelten sie das später als *Diffie-Hellman-Schlüsseltausch* bekannt gewordene Verfahren zum sicheren Tausch von Schlüssel zwischen Kommunikationspartnern. Aufgabenstellung dabei ist jene, dass zwei Kommunikationspartner *Alice* und *Bob* auf unverschlüsseltem Wege einen geheimen Schlüssel austauschen wollen, welchen eine Dritter Person *Eve* trotz Abhören des gesamten Verkehrs nicht rekonstruieren kann.

Elementare Prinzipien

Die anfänglichen Überlegungen gleichen dem *Kiste-Schloss-Prinzip*, welches wie folgt aufgebaut ist. Angenommen Alice möchte Bob über den normalen Postweg eine Kisten mit geheimen Inhalt senden, wobei sichergestellt werden soll dass keine Person welche die Kiste während des Weges in Händen hält sie öffnen kann. Darüber hinaus besitzen Alice wie auch Bob keinen gemeinsamen Schlüssel.

Alice versieht die Kiste mit einem Schloss zu welchem nur sie den Schlüssel besitzt und schickt sie zu Bob. Dieser versieht die Kiste mit einem zweiten Schloss, zu welchem wiederum nur er den Schlüssel besitzt und schickt die gesamte, nun doppelt gesicherte Kiste wieder zurück an Alice. Diese entfernt ihr eigenes Schloss und schickt die Kiste ein letztes mal zurück an Bob, welcher lediglich sein eigenes Schloss vorfindet und dieses problemlos öffnen kann.

Dass diese Variante auf dem Postweg relativ lang dauert wird natürlich durch die Geschwindigkeit in heutigen Netzen entkräftet. Darüber hinaus hat dieses Konzept den Vorteil, dass ein Schlüssel gar nicht erst ausgetauscht werden *muss*, da Alice wie auch Bob ja jeweils ihre eigenen Schlüssel verwenden und keiner den Schlüssel des anderen benötigen.

Unglücklicherweise lässt sich das Verfahren in dieser Form nicht algorithmisch umsetzen. Kodieren wir eine Nachricht M zuerst mit einem Schlüssel k_0 und anschließend mit einem Schlüssel k_1 , so muss die Entschlüsselung ebenfalls in derselben Reihen-

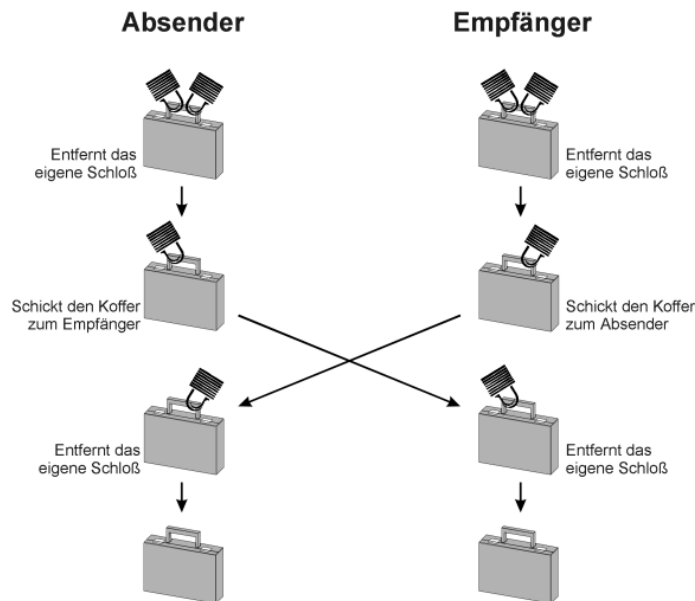


Abbildung 2.4: Veranschaulichung des Kiste-Schloss-Prinzips

folge stattfinden, da

$$M = D_{k_0}(D_{k_1}(E_{k_0}(E_{k_1}(M)))) \neq D_{k_1}(D_{k_0}(E_{k_0}(E_{k_1}(M))))$$

. Die Nachricht würde also falsch entschlüsselt werden und wäre somit unbrauchbar.

Der Algorithmus

Dies veranlasste die Herren Diffie, Hellman und Merkle nach mathematischen Operationen zu suchen mit welchen eine andere Variante des Schlüsseltausches realisiert werden kann, eine Variante womit der eigentliche Schlüssel wirklich über eine ungesicherte Leitung übertragbar ist.

Fündig wurden sie letztlich in der *Modul-Arithmetik*. Die dabei im Vordergrund stehende mathematische Operation ist die *modulo*-Funktion. Modulo teilt eine gegebene Zahl durch eine andere und liefert den ganzzahligen Rest der Division zurück: $13 \bmod 5 = 3$, da $13/5 = 2, 3 \text{ Rest}$.

Was diese Funktion auszeichnet ist das Prinzip die *Nicht-Umkehrbarkeit*. Haben sie das Ergebnis einer Modulo-Funktion sowie den Divisor gegeben ist es nahezu unmöglich den korrespondierenden Dividenden des Modus' ausfindig zu machen, da es unendlich viele davon gibt. Somit ist nicht nur $13 \bmod 5 = 3$ sondern beispielsweise

auch $23 \pmod 5$ und $24 \pmod 7$.

Zur Erläuterung des Algorithmus nehmen wir wieder Alice und Bob als Kommunikationspartner und setzen voraus dass Eve die gesamte Verbindung abhört. Alice und Bob einigen sich auf zwei unterschiedliche Zahlen G und M , welche ungesichert übertragen werden und folglich weder geheim sind noch mit der Sicherheit des Verfahrens in Zusammenhang stehen.

Alice errechnet nun eine Zahl Alpha, in dem sie

$$\alpha = G^a \pmod M$$

berechnet, wobei a eine von Alice geheimehaltene Zahl ist. Bob errechnet analog dazu eine Zahl Beta, mittels

$$\beta = G^b \pmod M$$

– auch hier definiert b eine von Bob geheimehaltene Zahl. Die Resultate α und β werden nun über den unsicheren Kanal ausgetauscht.

Nun berechnet Alice

$$K = \beta^a \pmod M$$

und Bob

$$K = \alpha^b \pmod M$$

. Wie bereits aus dem letzten formalen Schritt herausgeht, resultiert bei beiden Berechnungen erstaunlicherweise dieselbe Zahl, welche als Schlüssel verwendet werden kann. Darüber hinaus ist es Eve nicht ohne die Kenntnis von a und b möglich den vereinbarten Schlüssel zu rekonstruieren, da diese beiden Zahlen von Alice und Bob geheimhalten und nicht ausgetauscht wurden.

Der Schriftsteller und Physiker *Simon Singh* veranschaulichte in seinem grandiosen Werk [Sin01] die mathematische Vorgehensweise welche hier stattfindet gänzlich unmathematisch wie folgt. Alice, Bob und Eve haben je einen Dreiliter-Kanister Farbe, gefüllt mit einem Liter gelber Farbe (dies entspricht den öffentlich ausgetauschten Zahlen M sowie G . Alice rührt nun einen weiteren Liter beliebiger Farbe in ihren Kanister, ebenso Bob. Dieser zweiter Liter zugeführter Farbe muss dem jeweils anderen Kommunikationsteilnehmer nicht bekannt sein (was wir in unserer Rechnung als a und b bezeichnen), wir nehmen aber an dass Alice einen Liter rote, und Bob einen Liter grüne Farbe zumischt. Nun werden die beiden Farbkanister ausgetauscht und Alice wie auch Bob führen dem ihnen fremden Farbkanistern wiederum einen

Liter der von ihnen geheimgehaltenen Farbe hinzu. Schlussendlich erhalten beide exakt denselben Farbton.

Verfolgt Eve sämtliche Übertragungen, so weiß sie zwar dass Gelb die Grundfarbe war und kennt die beiden Gemische von Alice und Bob, nicht jedoch die einzelnen Farben mit denen gemischt wurde – ihr ist es unmöglich die endgültige Farbmischung exakt nachzuahmen, da sich gemischte Farben schließlich nicht ohne weiteres wieder heterogenisieren lassen.

Dieses Verfahren wird bis heute zum sicheren Schlüsseltausch verwendet.

2.7 Längen symmetrischer Schlüssel

Bei der Frage der Schlüssellänge spalten sich heutzutage die mathematischen und kryptologischen Geister. Die von DES effektiv umgesetzte Schlüssellänge von 56 Bit (was in etwa $7 \cdot 10^{18}$ Kombinationen entspricht) ist mittlerweile als zu kurz anzusehen und von effizient arbeitenden Rechnern in relativ kurzer Zeit zu brechen. Zumal die Berechnungskomplexität einer Schlüsselfunktion progressiv steigt, wäre es jedoch falsch zu behaupten dass eine verdoppelte Schlüssellänge die Sicherheit lediglich verdoppelt (man vergleiche $2^{56} \approx 7 \cdot 10^{18}$ mögliche Schlüsselwerte, $2^{112} \approx 5 \cdot 10^{35} \neq 2 \cdot 7 \cdot 10^{18}$).

Hier ein kleiner Auszug aus der Tabelle für empfohlene symmetrische Schlüssellängen von Lenstra/Verheul [LV00]:

| Jahr | Empfohlene Schlüssellänge (Bits) |
|------|----------------------------------|
| 2000 | 70 |
| 2005 | 74 |
| 2010 | 78 |
| 2015 | 82 |
| 2020 | 86 |
| 2025 | 89 |
| 2030 | 93 |
| 2040 | 101 |

Diese Werte beziehen sich auf die Approximation des Moor'schen Gesetzes welche besagt dass sich die Prozessorleistung in etwa alle 18 Monate verdoppelt (Bei genauerer Betrachtung stellen diese Werten sogar einen etwas pessimistischeren Ansatz

dar, da ausgehend von einer Leistung von 100%, sich diese nach fünf Jahren auf ca. 1064% steigert – dem Schlüssel hingegen werden i.d.R. alle fünf Jahre vier Bits hinzugefügt, was einer Steigerung der Rechenleistung auf 1600% entsprechen würde).

Zumal Rijndael minimal mit einer Schlüssellänge von 128 Bit arbeitet (und bis zu 256 Bit erlaubt) können wir uns wohl die nächsten Dekaden in Sicherheit wiegen, sofern man davon ausgeht keine bahnbrechende technologische Revolution auf dem Gebiet der Mikroelektronik bzw. Hardwaretechnik zu erfahren.

Einen sehr vielversprechenden wie auch mathematisch komplexen Ansatz liefert uns das Gebiet der Kryptographie über elliptischen Kurven (*ECC*), welche Asymmetrie durch wesentlich geringere Schlüssellängen bei gleichem Maß an Sicherheit bietet, worauf wir hier jedoch nicht näher eingehen wollen.

2.8 Einweg-Hashfunktionen

Hashfunktionen folgen im Allgemeinen einer Projektion von $h : \Sigma^* \rightarrow \Sigma^m$ wobei wir Σ als Eingabealphabet und Σ' als Ausgabealphabet definieren.

Dabei realisiert eine Hashfunktion i.A. folgende Prinzipien:

- **Datenreduktion (Datenexpansion):** Unabhängig von Größe und Umfang der eingehenden Daten I soll der dazugehörige Hashwert $h(I)$ eine fest definierte Größe aufweisen. Sind die Eingabedaten länger als die definierte Ausgabegröße des Hashes so sprechen wir von Datenreduktion, andernfalls von Datenexpansion (was selten der Fall ist).
- **Konfusion:** Ähnliche Strukturen der eingehenden Daten sollen zu völlig unterschiedlichen Ausgangsdaten führen.
- **Kollisionsfreiheit:** Die Möglichkeit zwei unterschiedliche Blöcke I_1 und I_2 von Eingangsdaten zu ermitteln welche denselben Hashwert $h(I_1) = h(I_2)$ erzielen soll minimiert werden. Aus mathematischer Sicht ist eine vollkommene Kollisionsfreiheit nur dann zu erreichen wenn der Ausgabewert $O = h(I)$ des Hashes mindestens solange ist wie die Größe der Eingangsdaten $|I|$. In der Praxis ist es meist sehr schwer Tupel (I_1, I_2) zu finden für welche $h(I_1) = h(I_2)$ gilt.

- **Inexistenz von inversen Funktionen:** Die Funktion h soll in sich unumkehrbar sein, also die Möglichkeit die ursprünglichen Eingangsdaten I über eine inverse Funktion h^{-1} aufgrund der Ausgangsdaten O zu rekonstruieren ausschließen ($I \neq h^{-1}(h(I))$).
- **Surjektivität:** Die Hashfunktion h soll es aus mathematischer Sicht ermöglichen, jedes existente Element der Zielmenge Σ^n (wobei $n = |h(I)|$) durch adäquate Eingangsdaten zu erreichen. Könnte man bestimmte Elemente der Zielmenge als Resultat von h ausschließen, würde die Sicherheit maßgeblich gefährdet.
- **(Effizienz):** Aus mathematischer Sicht kein zwingendes Kriterium für eine Hashfunktion, soll diese in der Praxis effizient in Bezug auf Dauer und Speicherverbrauch errechenbar sein (man spricht in diesem Zusammenhang auch von Performanz).

Kapitel 3

Kommunikationsstrukturen via POP und SMTP

3.1 Postfix als SMTP Server

Das Programm *Postfix* ¹ erlaubt uns die Verwendung eines zu *Sendmail* ² kompatiblen SMTP Servers der hohen Sicherheitsanforderungen genügt. Darüber hinaus bietet Postfix eine sehr anwenderfreundliche Konfiguration, welche über die globale Konifugationsdatei `/etc/postfix/main.cf` zu erreichen ist.

Es genügen hierbei die Optionsvariablen `mydomain` sowie `mydestination` korrekt zu initialisieren um Postfix mitzuteilen lokale E-Mails auf die Mail an den für die Empfänger zuständigen SMTP Server weiterzuleiten, sofern diese außerhalb des eigenen Netzes liegen welche nicht unter die Administration von Postfix fallen.

Welcher SMTP Server bei SMC verwendet wird ist nicht von Bedeutung, da SMC sich lediglich mit dem in `~/ .smc/smc.conf` definierten SMTP Server verbindet und die Nachricht versendet. Der Einfachheit halber kommt in der Proof-Of-Concept Variante jedoch ein eigener Server zum Einsatz.

Anmerkung: SMC ist bis dato nicht fähig, SASL-verschlüsselte Verbindungen zu POP- wie auch SMTP Servern aufzubauen wodurch die Verwendung bestimmter öffentlicher Server nur eingeschränkt möglich ist.

¹www.postfix.org

²www.sendmail.org

3.2 Protokollkoordination durch SMTP

SMTP (*Simple mail transfer protocol*) entstammt dem RFC 821 ([Pos82]) aus dem Jahre 1982 und stellt ein textbasiertes Netzwerkprotokoll zum Versand von E-Mail Nachrichten über entsprechende SMTP-Services dar. SMTP sieht eine Identifikation aber keine Authentifikation vor.

Die SMTP zugrundeliegende Protokollstruktur ist denkbar einfach – der Client benötigt lediglich fünf Kommandos um eine E-Mail über einen SMTP-Server zu versenden:

- **HELO** Senderdomain
Liefert dem Server die Domain des Absenders, also dem Benutzer welcher sich am SMTP Server einloggt. Obwohl das **HELO**-Kommando noch von (fast) allen SMTP-Services erwartet wird, existiert es i.A. nur mehr aus Kompatibilitätsgründen. Die dabei übergebene Domain des Absenders wird in vielen Fällen gar nicht erst ausgewertet, sondern vom SMTP-Service selbst über einen Reverse Lookup in Erfahrung gebracht.
- **MAIL FROM:** [Absender]
Über dieses Kommando wird dem Server die Quelladresse des Absenders übermittelt. In der ursprünglichen Definition erfolgt keine Sicherheitsprüfung ob die angegebene Domain überhaupt existent ist.
- **RCPT TO:** [Adressat]
Analog zu **MAIL FROM** wird der Adressat festgelegt. Auch hier enthält die Originalspezifikation von SMTP keine Überprüfung auf Existenz von Domänen.
- **DATA** <CR><LF> [Nutzdaten] <CR><LF> .<CR><LF>
Nach dem **DATA** Kommando und einem darauf folgendem Zeilenumbruch folgt die Eingabe des eigentlichen Nachricht (Nutzdaten, *payload*). Ab diesem Zeitpunkt können beliebige Zeichen zum Server übermittelt werden. Als Abbruchzeichen (*escape character*) dient ein Zeilenumbruch, gefolgt von einem Punkt und einem abermaligen Zeilenumbruch. Sämtliche Header-Daten einer E-Mail werden ebenfalls in diesem Datensegment definiert, von einem E-Mail Client i.d.R. über die spezielle Zeichenfolge <Type>:<Value> erkannt und nicht als eigentlicher Nachrichtentext ausgegeben.
- **QUIT**
Beendet die Verbindung zum SMTP Server

Nach jedem abgesetztem Kommando reagiert der Server mit einer Nachricht welche entweder die korrekte Abarbeitung des abgesetzten Kommandos oder eine Fehlermeldung enthält. Diese Rückgabewerte werden über dreistellige Nummernwerte definiert, jedoch trotzdem als reiner Text versendet.

Die Nachlässigkeit bei der Überprüfung von Absender- und Zieladressen wurde erstmals beim Aufkommen von Spam-Mails in der Post-Usenet-Ära um 1990 schwer in Frage gestellt. Seither überprüfen die meisten SMTP-Services ob ihnen die genannte Absender Adresse bekannt ist und verweigern ggf. den Versand.

3.3 Protokollkoordination durch POP3

POP3 ist eine 1988 publizierte Redefinition von seinen Vorgängern POP2 [BPC+85] und POP [Rey84]. Von Version zu Version wurde im wesentlichen die Komplexität geringer um die Implementierung von POP sowie die Abwicklung des *Fetch*-Prozesses zu vereinfachen.

Wie auch bei SMTP wurde POP als textbasiertes Protokoll definiert und kennt in der Version 3 folgende für uns relevante Befehle:

Verbindungsaufbau:

- **USER** [Benutzername]
Übernimmt die Identifikation des Benutzers bzw. des POP-Clientprogramms gegenüber dem Service. Dieser überprüft die Identität des übergebenen Benutzernamens und meldet eine Fehlermeldung falls dieser unbekannt ist.
- **PASS** [Passwort]
Übernimmt die Authentifikation des Benutzers bzw. des POP-Clientprogramms gegenüber dem Service. Man beachte dass auch das Passwort laut POP-Spezifikation *unverschlüsselt* über das Netzwerk transferiert wird.

Mailverwaltung:

- **STAT**
Liefert die Gesamtanzahl sowie die Anzahl der ungelesenen (d.h. noch nicht abgerufenen) E-Mails.

- **LIST**
Bei der Ausführung des **LIST**-Kommandos ohne Argument wird eine komplette Liste der verfügbaren Nachrichten und deren Größe in *Octets* zurückgegeben. Optional kann eine Nachrichtennummer angegeben werden, wodurch lediglich die Größe der Nachricht mit der übergebenen Nummer geliefert wird.
- **RETR [Nummer]**
Liefert die eigentliche Nachricht (das Datensegment) mit der angegebenen Nummer.
- **DELE [Nummer]**
Löscht die Nachricht mit der angegebenen Nummer vom Server

Servertest und Verbindungsabbau:

- **NOOP**
Das **NOOP**-Kommando dient lediglich zu Analyse- und Testzwecken ob der Server noch reagiert. Es wird keine Aktion ausgeführt.
- **RSET**
Verwirft alle bisher getätigten **DELE**-Kommandos, sofern diese in der aktuellen POP-Sitzung stattgefunden haben.
- **QUIT**
Beendet die Verbindung zum POP-Server.

Einige Implementierungen von POP-Services realisieren noch einige weitere Kommandos, welche aber optional sind und für uns nicht von Relevanz.

3.4 Einbindung von öffentlichen Servern in SMC

SMC legt in der Konfigurationsdatei `smc.conf` die benötigten Felder fest um den Verbindungsauf- und -abbau zu SMTP und POP Servern korrekt abwickeln zu können.

Folgend der Beispielaufbau einer unverschlüsselten `smc.conf`:

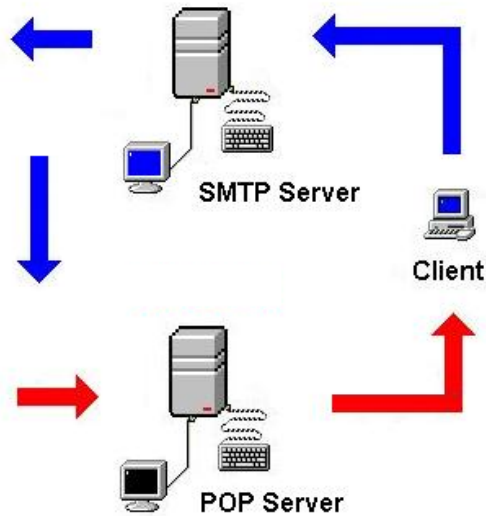


Abbildung 3.1: Datentransfer zwischen Clients, SMTP- und POP-Servern

```

1  smtp_server      localhost
2  smtp_account    my-domain.tld
3  smtp_sender     example_address@my-domain.tld
4  pop_server      pop.my-domain.tld
5  pop_user        testuser
6  pop_passwd      testpasswd
7  home            /home/testuser/

```

Die Syntax ist selbstbeschreibend: Zwischen einem Parameternamen-Werte-Tupel liegen beliebig viele Whitespace-Zeichen und enden in jedem Fall mit einem Zeilenbruch. Der adäquate Perl-kompatible reguläre Ausdruck hierfür ist $\text{~}(w)(s)^*(w)\$$.

Kapitel 4

Implementierung von SMC

4.1 Systemumgebung

Als Systemumgebung für SMC dient eine 32-Bit Linux-Plattform. Aufgrund der verwendeten Bibliotheken sowie Systemaufrufe ist die Implementierung jedoch auf jedem POSIX-kompatiblen System lauffähig (i.A. also alle bekannten freien wie auch proprietären Unix-Derivate).

Eine native Unterstützung für Microsoft Windows ist nicht gegeben, sehr wohl jedoch lässt sich das Programm über den renomierten POSIX-Emulator Cygwin¹ auch unter Microsoft Betriebssystemen mit installiertem Perl-Interpreter und den notwendigen Perl-Modulen ausführen.

Als Programmiersprache kommt Perl in der Version 5.8 zum Einsatz. Die Sprache Perl an sich weist Vorzüge auf, welche mir zur Umsetzung des Projekts ausschlaggebend erschienen. Perl ist äußerst mächtiges Werkzeug und daher in der Unix-Welt sehr beliebt zur Realisierung komplexer algorithmischer Kontrollstrukturen.

Obgleich Perl eine reine *interpretierende Skriptsprache* ist, also zur Laufzeit in Maschinencode umgesetzt wird, werden trotzdem signifikante Geschwindigkeiten gegenüber nicht-interpretierenden Hochsprachen erzielt [Con02]. Durch die von Perl überaus elegante Art und Weise der Verarbeitung von Zeichenketten, zeichnet sie sich als Implementierungssprache für text-basierende Protokolle wie in SMC verwendet, aus.

¹www.cygwin.com

4.2 Elementare Protokollvorgaben

Ein Hauptkriterium des Projekts zielt darauf ab, sichere Kommunikation über einen unsicheren Kanal zu ermöglichen. POP [Ros88] und SMTP [Pos82] sind Protokolle aus den Urzeiten des Internets und sehen wie zu erwarten keine protokollinterne Verschlüsselung vor.

Beide Protokolle sind textbasiert, d.h. ist in beiden Fällen kein eigener Client zwingend erforderlich um ohne große Mühen mit POP- sowie SMTP-Servern kommunizieren zu können. Ein einfaches bidirektionales 8-Bit-orientiertes netzwerkfähiges Kommunikationsprogramm wie *Telnet* (RFCs 854 und 855) reicht hierfür aus.

SMC verwendet zur Unterstützung von diesen Protokollen die offiziellen CPAN-Module *Net::SMTP* und *Net::POP3*, welche die Schnittstelle zur Kommunikation mit den korrespondierenden Servern, auf von SMC instanziierte Objekte abstrahiert und dadurch die Fehlerbehandlung (*exception management*) vereinfacht.

4.3 Das SMC Protokoll

Um den Austausch von Schlüsseln und Daten sowie die Koordination zwischen einzelnen SMC Clientprogrammen zu gewährleisten, definiert SMC ein eigenes textbasiertes Protokoll.

Sämtliche SMC-Protokolldaten werden hierfür in den standardisierten Datensegment (*Body*) einer E-Mail Nachricht eingebettet, und folgen folgender allgemeiner Konvention:

```
smc!<MESSAGE-TYPE>!smc-end  
smc!<KEY=VALUE>!smc-end
```

wobei pro Nachricht *M* genau 1 Protokollelement **MESSAGE-TYPE** definiert werden muss, und mehrere Protokollattribute **KEY=VALUE** definiert werden können.

Die von SMC festgelegten Standard-Typen von Messages sind

- INVITATION
- INVITATION-REPLY

- MESSAGE
- MESSAGE-BEGIN
- MESSAGE-END

INVITATION ist ein Protokollobjekt welches als Initialnachricht (*initial message*) zur Schlüsselgenerierung mit einem bestimmten Kommunikationspartner dient. Es enthält dabei alle relevanten Informationen welche für den Adressaten notwendig sind um den endgültigen Kommunikationsschlüssel (*final key*) zu erzeugen. Aufgrund des Prinzips des Diffie-Hellman-Schlüsseltausches ist diese Nachricht nicht an nur einen Adressaten gebunden sondern kann auch zeitgleich an mehrere Empfänger gesendet werden, da jeder einzelne die Form des resultierenden finalen Schlüssels durch Zufallsgeneratoren abermals beeinflusst.

INVITATION verfügt überdies über folgende obligate Objektattribute:

1. `request-from=<SENDER-ADDRESS>`
2. `p=<NUMBER>`
3. `g=<NUMBER>`
4. `pk=<NUMBER>`

Die genauere Spezifikation der einzelnen Werten werden wir weiter unten näher betrachten.

INVITATION-REPLY :

Kennzeichnet ein Protokollobjekt welches der Empfänger nach erfolgreichem Erhalt der initialen INVITATION-Nachricht dem ursprünglichen Sender zurückliefert. Mit Hilfe der hier mitgesendeten Informationen ist nun auch der ursprüngliche Sender in der Lage den finalen Schlüssel vollständig zu generieren.

INVITATION-REPLY verfügt über folgende obligate Objektattribute:

1. `reply-from=<SENDER-ADDRESS>`
2. `pk=<NUMBER>`

MESSAGE :

Kennzeichnet eine mittels SMC verschlüsselte Nachricht. Dabei werden die Subtypen `MESSAGE-BEGIN` und `MESSAGE-END` in folge definiert um die eigentliche Nachricht von den in einer `MESSAGE`-Nachricht enthaltenen Metainformationen über die eigentliche Nachricht zu trennen.

Dabei sind folgende Objektattribute der Nachricht angegeben:

1. `smc!sender=<SENDER>!smc-end` Legt die Senderadresse fest. Diese wird nach Dechiffrierung der Nachricht über den Schlüssel des angegebenen Empfängers verifiziert.
2. `smc!cipher=<CIPHER>!smc-end` Definiert den Verschlüsselungsalgorithmus der bei der Chiffrierung der Nachricht verwendet wurde um sicherzustellen dass die Nachricht korrekt rekonstruiert werden kann (mögliche Werte sind `rijndael`, `serpent` sowie `twofish`).
3. `smc!hmac=<HMAC>!smc-end` Liefert den vom Sender errechneten Hashwert der Nachricht, konkatinert mit dem verwendeten Schlüssel. Dieser Hashwert wird nach der Entschlüsselung über den Empfänger neu errechnet und mit dem im Klartext mitgelieferten `hmac`-Attribut verglichen. Sollten diese Werte nicht gleich sein, besteht die Gefahr einer Manipulation durch Dritte (neben anderen nicht-fremdartigen Fehlerquellen wie z.B. Datenübertragungsfehler).

MESSAGE-BEGIN :

Kennzeichnet den Beginn einer Nachricht für welche der symmetrische Kommunikationsschlüssel bereits im Vorfeld auf den Clients beider Kommunikationspartner definiert werden. Auch dieses Element beinhaltet ein zwingendes Objektattribut, nämlich die Wahl des verwendeten Verschlüsselungsalgorithmus `smc!cipher=<CIPHER>!smc-end`.

MESSAGE-END :

Kennzeichnet das Ende einer verschlüsselten SMC Nachricht.

Standardmäßig wird in Abhängigkeit des verwendeten Chiffreverfahrens der jeweils längste mögliche Schlüssel als Untermenge des Errechneten finalen Schlüssels verwendet, i.d.R. also 256 Bit.

4.4 Realisierung des Schlüsseltausches

Der Schlüsseltausch via SMC basiert auf den *Diffie-Hellman*-Algorithmus und findet autonom über das SMC-Clientprogramm statt. Eines der gravierendsten Argumente gegen die Verwendung von symmetrisch verschlüsselter E-Mail-Kommunikation stellt die zwingende Synchronität von symmetrischen Algorithmen dar, obgleich der zugrundeliegende E-Mail-Protokollstapel grundsätzlich von einer asynchronen Anwendung seiner Benutzer ausgeht.

In der Praxis heißt dies, dass eine zum Zeitpunkt t_0 versandte E-Mail i.A. erst zu einem Zeitpunkt t_1 von seinem Adressaten gelesen werden wird. Die dabei entstehende chronologischer Differenz ist für die korrekte Abarbeitung des Protokolls nicht von Relevanz, zumal eine über SMTP versandte E-Mail i.d.R. solange auf dem verwendeten Server gespeichert bleibt bis sie über ein Übernahmeprotokoll wie POP oder IMAP abgerufen wird (dieses entscheidet wiederum ob die E-Mails am Server gespeichert bleiben sollen oder nicht, sofern der Server dies zulässt).

Der *Diffie-Hellman*-Algorithmus (sowie alle anderen bis heute verfügbaren vergleichbaren Schlüsseltauschalgorithmen) basiert darauf dass zwischen Sender und Empfänger eine zeitlich synchronisierte Verbindung besteht da zum eigentlichen Schlüsseltausch der Versandt *einer* Nachricht von Kommunikationspartner A zum Kommunikationspartner B hierfür nicht ausreicht.

SMC löst dieses Problem mit dem *Invitation - Invitation Reply*-Prinzip. Wir gehen im folgenden Beispiel davon aus dass A eine verschlüsselte Nachricht zu B schicken möchte. SMC definiert das Protokoll wie folgt:

1. A sendet zum Zeitpunkt t_0 eine Initialnachricht über einen existierenden unsicheren Kanal (wir nehmen SMTP als Grundlage) zu B . SMC nennt diese Nachricht *Invitation*, also Einladung. B wird somit eingeladen einen gemeinsamen Schlüssel mit A zu generieren. Dabei enthält die Invitation-Nachricht folgende Attribute:
 - **request-from** Hier wird die Absenderadresse unabhängig von der in SMTP angegebenen Adresse nochmals spezifiziert. Es ist also theoretisch möglich von beliebigen E-Mail-Adressen (bspw. Sekundäradressen) Einladungen zur Benutzung anderer Adressen zu verwenden. Dieses Prinzip könnte aufgrund der Spamlage der heutigen Zeit durchaus von Nutzen sein.

- **p** Definiert die (meist sehr große) Primzahl p wie sie im Algorithmus von Diffie-Hellman definiert wird.
- **g** Definiert die Basis zum geheimen Exponenten welche der Diffie-Hellman-Schlüsseltausch zur weiteren Modulo-Operation benötigt und relativ prim zu p sein sollte. Die Größe von g ist nicht entscheidend.
- **pk** Definiert den sogenannten *Public key* des Senders A . Dieser darf ohne Bedenken über die unsichere Verbindung gesendet werden und wird von B zur Kalkulation des Schlüssels benötigt.

Man beachte dass zum Zeitpunkt t_0 der Absender A den finalen Kommunikationsschlüssel noch nicht errechnen kann.

2. B antwortet zu einem beliebigen Zeitpunkt $t_1 > t_0$ auf die Invitation-Nachricht mit einem *Invitation reply*. B hat zu diesem Zeitpunkt den finalen Schlüssel bereits errechnet, da zu den von A gegebenen Werten p , g und pk_A lediglich sein eigenes Schlüsselpaar (pk_B sowie der private Schlüssel $priv_B$) von Nöten ist welches lokal errechnet werden kann. Der Invitation Reply enthält:

- **reply-from** Die E-Mail Adresse des ursprünglichen Adressaten (zum Zeitpunkt t_0) und jetzigem Sender (Zeitpunkt t_1).
- **pk** Den öffentlichen Schlüssel von B , also aus mathematischer Sicht pk_B . Dieses Element ist das einzige welches A benötigt um den finalen Schlüssel ebenfalls errechnen zu können.

Da nun beide Kommunikationspartner denselben Schlüssel errechnet haben werden ihre E-Mail-Adressen gegenseitig in das jeweilige Adressbuch des SMC-Clientprogramms aufgenommen und ein gesicherter E-Mail-Verkehr kann ab nun stattfinden (Zeitpunkt t_2).

4.5 Regeneration von Schlüsseln

Die Sicherheit einer chiffrierten Nachricht hängt nicht nur von der Wahl des Schlüssels, dessen Länge sowie Geheimhaltung ab sondern auch wesentlich von der Nutzungsdauer. Wird im arithmetischen Mittel in etwa

$$\frac{t_{decrypt}}{2} = f(C, l_K)$$

Zeit benötigt um einen Schlüssel K der Länge $|k|$ über den Algorithmus C zu berechnen, der Schlüssel jedoch nur für eine Periode von

$$t_{usage} = \frac{t_{decrypt}}{\mu > 2}$$

Verwendung findet, so erhöht sich die Gesamtsicherheit unweigerlich.

Diese Schlüsselregeneration (*key regeneration*) implementiert SMC standardmäßig über eine vordefinierte maximale Periodendauer t_{usage} von 15 E-Mails oder 365 Tagen, was immer zuerst eintrifft.

4.6 Sicherstellung der Authentizität

Um zu garantieren dass eine empfangene Nachricht auch wirklich vom suggerierten Absender stammt, verwendet SMC das in [KBC97] definierte HMAC-Verfahren.

Dabei wird der Klartext der zu versendenden Nachricht mit dem vollständigen Schlüssel der Nachricht konkateniert, und das dabei resultierende Bytearray als Eingabedatum für eine Hashfunktion zur Verfügung gestellt. Der 160-Bit lange Funktionswert des Hashes wird als eigenes SMC Objektattribut mit der E-Mail im Klartext (d.h. außerhalb der `BEGIN-MESSAGE-END-MESSAGE`-Umgebung) mitversandt.

Beim Empfang entschlüsselt SMC die Nachricht zuerst und führt im Anschluss exakt denselben Algorithmus auf Basis der erhaltenen Daten durch. Sollte sich die eigens kalkulierte HMAC von der im Klartext über die E-Mail erhaltene HMAC unterscheiden, so liegt die Vermutung einer Korruption des Textes, des Hashwertes oder ein Übertragungsfehler nahe. In diesem Fall ersetzt SMC die Ausgabe des ursprünglich bereits errechneten Klartextes der Nachricht mit einer Warnmeldung.

4.7 Clientseitige Sicherheit

Clientseitige Sicherheit funktioniert, wie in [RC00] ausführlich beschrieben, nur eingeschränkt. Besser ausgedrückt, clientseitige Sicherheit funktioniert in gewissen Anwendungsgebieten kaum bis gar nicht, in anderen Gebieten sehr gut. Versucht man drohende Buffer-Overflows eines Netzwerkprotokolls clientseitig zu unterbinden ist der korrespondierende Service auf der Gegenseite trotzdem nicht gefeit gegen Angrif-

fe über kompromittierte Clientprogramme welche trotzdem falsche Pakete senden.

Die lokale clientseitige Sicherheit von SMC bezieht sich jedoch lediglich auf die Sicherheit der lokalen Daten. Sämtliche adressspezifischen Schlüssel werden in einer lokalen Datei verwaltet welche wiederum symmetrisch chiffriert wird. Den hierfür erforderlichen Schlüssel muss der Benutzer in Form einer Passphrase am Programmstart eingeben. Dieser wird nicht im Klartext am System gespeichert, sondern lediglich ein Hashwert davon – wir nennen diesen Schlüssel auch den Aktivierungsschlüssel (*activation key*).

Das heisst dass SMC selbst den eigentlich Aktivierungsschlüssel gar nicht kennt, sondern bei jedem Programmstart die eingegebene Passphrase durch dasselbe Hashverfahren kodieren muss und anschließend diesen Wert mit dem lokal gespeicherten Wert vergleicht. Unix verfolgt eine ähnliche Strategie zur Verwaltung der Passwörter von Benutzeraccounts am System (i.d.R. in der Datei `/etc/shadow`) [Bau05].

Diese Datei (`smc.passwd`) sowie die Datei zu Verwaltung der Adressschlüssel (`smc.addresses`) sollten keinesfalls andere Berechtigungen besitzen als `-rw-----` bzw. oktal `0600`, um zu gewährleisten dass kein unbefugter Benutzer lesenden und/oder schreibenden Zugriff auf die Dateien findet. Im Idealfall sollte `smc.passwd` gar den Berechtigungen `0400` unterlegen sein und manuell geändert werden falls man den Aktivierungsschlüssel ändern möchte.

4.8 Algorithmische Modularität

SMC wurde bewusst einer modularen Programmierung unterzogen welche es erlaubt, mit wenigen Änderungen alternative Verschlüsselungsalgorithmen zu verwenden. Dabei ist die Arbeitsweise einer Chiffre irrelevant, sie muss nur eine (meist implizit existente) adäquate Programmierschnittstelle aufweisen welche (zumeist als Funktion realisiert) den Klartext als Array von Bytes übergeben werden kann und wiederum ein Array von Bytes mit verschlüsseltem Inhalt zurückgibt.

Aktuell verfügt SMC über die Möglichkeit Nachrichten mittels Rijndael, Serpent und Twofish zu kodieren, jeder beliebige andere Algorithmus wäre jedoch auch denkbar. Diese Modularität findet speziell Nutzen bei der Integration in bestehende Systemkomplexe mit existenten alternativen Verschlüsselungsverfahren sowie bei der Einbindung künftig entwickelter symmetrischer Kryptosysteme.

4.9 Obligate CPAN Bibliotheken

CPAN² (*Comprehensive Perl Archive Network*) stellt ein zentrales Bibliothekenarchiv für Perl zur Verfügung. Zahlreiche kryptographische Routinen werden von SMC über CPAN-Module realisiert, zumal diese bereits als ausreichend getestet gelten und SMC somit selbst modular über Aktualisierungen des CPAN erweiterbar ist, ohne den eigentlichen Quellcode ändern zu müssen.

- *Class::Loader*
Notwendig zur korrekten Instanzierung, Allokierung sowie Deallokierung von Objekten welche über objektorientierte Module von Perl definiert wurden.
- *Math::Pari*
Dieses Modul realisiert eine Schnittstelle zur, von zahlreichen anderen Sprachen bekannten, PARI-Bibliothek. Diese bietet Funktionalitäten zur Errechnung von numerischen, analytischen sowie zahlentheoretischen Kalkulationen und wird bei SMC primär zur Generierung von großen Primzahlen benötigt.
- *Crypt::Random*
Wird benötigt um pseudozufällige Zahlenfolgen zu generieren. Dabei wird hauptsächlich auf das von allen POSIX-Unices standardmäßig zur Verfügung gestellten Pseudodevice `/dev/random` zugegriffen. Falls es das verwendete Betriebssystem erlaubt, nutzt *Crypt::Random* auch einen *entropy gathering daemon* um die oftmals nur mäßig verwendbare Entropie von Spezialgerätedateien wie `/dev/random` zu erhöhen.
- *Crypt::Primes*
Realisiert die Erzeugung großer Primzahlen.
- *Crypt::Rijndael*, *Crypt::Serpent*, *Crypt::Twofish*
Implementieren die Blockchiffren Rijndael, Serpent und Twofish. Um die Performanz zu erhöhen, wurden diese Module lediglich als Dummy-Schnittstelle in Perl integriert, welche darunterliegende unter C kompilierte Funktionen zur Errechnung der korrekten Werte aufrufen und wiederum über PerlFunktionen zurückliefern.
- *Crypt::SHA1*
SHA1 ist der von SMC standardmäßig verwendete Hashalgorithmus und wird zur Konstruktion sowie Verifikation von HMAC-Strings benötigt.

²www.cpan.org

Kapitel 5

Anwendung von SMC

5.1 Basiskonzepte

SMC ist als Mail-User-Agent gedacht, und setzt kein Wissen über die dahinterliegenden Protokolle oder Mail-Server voraus. Um eine kryptographisch sichere symmetrische Kommunikation zu gewährleisten, werden einige Grundprinzipien erläutert:

- Die erste Nachricht an einen beliebigen Adressaten wird aufgrund der hinter Diffie-Hellmann liegenden Mathematik nicht verschlüsselt gesendet, da hierfür zuerst die symmetrischen Schlüssel untereinander ausgetauscht werden müssen.
- Die Realisierung des Tausches erfolgt bei SMC über das *Invitation/Reply*-Prinzip. Die initiale Nachricht an einen bis dato SMC unbekanntem Benutzer ist immer eine Einladung um einen Schlüsselaustausch vorzunehmen.
- In dieser Nachricht befinden sich bereits alle relevanten Daten um den Sender die Möglichkeit zu bieten den Schlüssel zu errechnen. Dieser muss zur korrekten Abarbeitung des Protokolls jedoch zwingend eine valide Antwort zur empfangenen Einladung (*Reply*) zurücksenden.
- Ab diesem Zeitpunkt kann der verschlüsselte Verkehr stattfinden.

5.2 Das Hauptmenü

SMC stellt die Umsetzung einer *Proof of concept*-Idee dar und wurde somit nicht grundsätzlich zur breiten Verwendung konzipiert. Aus diesem Grund schien es mir angebracht die Benutzerschnittstelle (*user interface*) nicht über eine grafische Programmumgebung zu implementieren.

Da auch heutzutage ein Großteil der getätigten Arbeiten unter Unix auf der Kommandozeile stattfindet, wurde diese als Ein- und Ausgabemedium zur Interaktion mit dem Benutzer gewählt.

Nach dem Start von SMC durch Aufruf von `./smc`¹ leitet dieses den Benutzer automatisch in das Hauptmenü, welches wie folgt strukturiert ist:

```
1 Definition module initialized
2 Configuration module initialized
3 SMTP module initialized
4 POP3 module initialized
5 Crpto module initialized
6 IO module initialized
7
8 Hello smc_test1@yahoo.de
9 Welcome to SMC v0.3
10
11 Press Ctrl+C or q to quit.
12 m: Send new email
13 g: Fetch emails from POP server (and delete them from server)
14 c: Change configuration values
15 r: Reply on received SMC invitations
16 a: Process received SMC invitation replies
17 d: Decrypt SMC-encrypted fetched E-Mails
18 q: Quit SMC
```

Dabei kontrolliert SMC von welchem Systembenutzer das Programm gestartet wurde und liefert in Abhängigkeit davon die dazugehörige Mailbox der gespeicherten E-Mail Adresse und lädt sämtliche Keys der Kommunikationspartner.

¹Die Datei `smc` muss dabei das *execute*-Flag gesetzt haben – sollte dies nicht der Fall sein, sollte die Eingabe von `chmod +x smc` Abhilfe schaffen

5.3 Senden, Empfangen und Zurückschicken von Einladungen

Möchte der Benutzer *A* nun eine E-Mail an einen bisher unbekanntem Adressaten *B* senden, erkennt SMC den Sachverhalt indem überprüft wird ob der Schlüssel von *B* im Adressbuch definiert ist. Ist dies nicht der Fall, wird *A* gefragt ob er/sie der angegebenen Adresse eine Einladung zusenden möchte.

```
1 Enter recipient mail address: smc_test2@yahoo.de
2 The recipient you have chosen does not seem
3   to be in your addressbook.
4 Do you want to send an initial invitation e-mail? (y/n)
5
6 Generating 512-bit prime... (this may take a while)
7 Generating base...
8 Generating DH keypair...
9 Generating initial invitation message ...done
10 Send invitation? (y/n)
11   Sender OK
12   Recipient OK
13
14 Message successfully delivered to smc_test2@yahoo.de
15 Everything OK - Press any key.
```

Nachdem der zweite Benutzer *B* zu einem beliebigen späterem Zeitpunkt seine E-Mails über SMC abrufen, stellt dieses fest dass eine Einladung zum Schlüsseltausch gefunden wurde und informiert *B* darüber.

```
1 You have 1 new messages
2   (0 SMC messages, 1 SMC Invitations, 0 SMC Invitation replies,
3    0 others)
4 Fetched new mails to local Inbox and deleted from server.
```

Möchte der *B* die Einladung erwidern, errechnet SMC die hierfür notwendigen Werte und sendet eine Nachricht zurück.

```
1 Processing mail 1163947188-1...
2 Invitation reply successfully sent to smc_test1@yahoo.de.
```

Beim nächsten E-Mail-Abruf von *A*, wird dieser informiert dass seine Einladung erwidert wurde.

```

1 You have 1 new messages
2   (0 SMC messages, 0 SMC Invitations, 1 SMC Invitation replies,
3     0 others)
4 Fetched new mails to local Inbox and deleted from server.
```

Nun kann der finale Schlüssel auch von A errechnet werden und B wird ins Adressbuch aufgenommen.

```

1 Processing invitation reply 1163947773-1...
2
3 Found pk value.
4 Found sender address: smc_test2@yahoo.de
5 Successfully calculated final key: 2399[...]87.
6
7 Your key finger print is:
8   305e f29e c76e 9404 9b0d f829 4082 23b6 08ce 3eb7
9
10 Invitation reply successfully processed.
11 Added user smc_test2@yahoo.de in addressbook.
12 Press any key.
```

5.4 Senden verschlüsselter Nachrichten

Wechselt der Benutzer in den Sendemodus und gibt eine SMC bereits bekannte Adresse als Adressaten ein, so wird der dazugehörige Schlüssel aus dem Adressbuch extrahiert und im Anschluss zur Chiffrierung verwendet.

Der eigentliche Inhalt der E-Mail Nachricht kann beliebig lang sein und wird zeilenweise direkt im Kommandozeilenfenster eingegeben. Wie bei SMTP üblich, beendet ein Newlinezeichen gefolgt von einem Punkt und einem abermaligem Newlinezeichen den Datenblock.

Abschließend steht es dem Benutzer frei den gewünschten Verschlüsselungsalgorithmus zu wählen bevor die Nachricht verschickt wird.

```

1 Enter recipient mail address: smc_test2@yahoo.de
2 Recipient known, getting corresponding encryption key.
3 Key found for smc_test2@yahoo.de (7539[...]246)
4   Sender      OK
5   Recipient   OK
```

```

6
7 You may enter your desired mail text now (quit with \n.\n):
8 Hello,
9 this is a test E-Mail.
10 .
11
12 Ok, message successfully read (2 lines).
13 Which cipher do you want to use for encryption?
14 (1) Rijndael
15 (2) Serpent
16 (3) Twofish
17
18 Using: rijndael. Everything seems to be fine - Really send (y/n)?

```

5.5 Entschlüsselung von empfangenen Nachrichten

Bei der erfolgreichen Entschlüsselung empfangener Nachrichten werden jegliche via SMC chiffrierte empfangene Nachrichten welche sich aktuell in der Inbox befinden in chronologischer Reihenfolge interiert und dechiffriert.

```

1 Decrypted message 1164819919-2, sent by smc_test1@yahoo.de (encrypted via
2 Hello,
3 this is a test E-Mail!
4
5 HMAC successfully verified, message authenticated:
6 Mail was seriously sent from smc_test1@yahoo.de

```

Dabei bezeichnet 1164819919-2 eine systemweit eindeutige Message-ID. SMC erkennt den zur Verschlüsselung der Nachricht verwendeten Algorithmus (in diesem Fall Rijndael) und gibt den entschlüsselten Text aus. Im Anschluss wird dem Benutzer nochmals explizit mitgeteilt, dass das mit der Nachricht versandte HMAC-Bytefeld erfolgreich verifiziert, und somit der Absender erfolgreich authentifiziert werden konnte.

Ist dies aus unbestimmten Gründen nicht der Fall, so meldet SMC folgenden Fehler und entschlüsselt die Nachricht nicht, auch wenn der Schlüssel korrekt wäre.

```

1 HMAC couldnt be verified! Warning: Possible attack.

```

5.6 Sonstige Funktionalitäten

Adressbuch Im Adressbuch können alle bis dato gespeicherten Adressen eingesehen werden. Dabei beschränkt sich SMC auf das Speichern von E-Mail-Adressen, für welche ein Schlüsseltausch vollzogen wurde und somit Adressat, Schlüssel sowie Zeitstempel der Vollendung des Tausches feststehen.

Eine Erweiterung, um normale Adressen ohne erzeugten Schlüssel speichern zu können, wäre nicht sinnvoll da SMC lediglich E-Mails an Empfänger versendet, für welche ein Schlüssel existiert. Wird kein passender Schlüssel zu einer gegebenen Adresse gefunden, so verweigert SMC den Versand.

Ändern der Konfiguration Im Konfigurationsmenü lassen sich die wichtigsten Einstellung bezüglich SMTP- sowie POP-Server ändern ohne direkt auf die darunterliegenden Konfigurationsdateien zugreifen zu müssen. Mögliche Optionen sind hier:

- Ändern der Adresse des SMTP-Servers,
- ändern der eigenen Domain (meist nicht benötigt),
- ändern der eigenen E-Mail-Adresse,
- ändern der Adresse des POP-Servers,
- ändern des Benutzernamens am POP-Service,
- ändern des Passworts am POP-Service.

Bei der Änderung der eigenen E-Mail-Adresse sei anzumerken dass SMTP die Senderadresse lediglich als Text in den Header einer SMTP-Nachricht einbettet, nicht jedoch validiert oder auf Gültigkeit überprüft. Die meisten öffentlichen SMTP Server lehnen den Versand von Ihnen unbekanntem Sender-Adressen grundsätzlich ab.

Verwendet man einen eigenen, lokalen SMTP Server (wie bspw. Postfix, Exim, Sendmail), so kann die Senderadresse i.d.R. beliebig gewählt werden, zumal die Konfiguration des eigenen Servers beliebig erfolgen kann. Der Vollständigkeit halber sei jedoch erwähnt, dass der Großteil der bekannten E-Mail-Provider die Domain der Senderadresse, mit der Domain des SMTP-Servers, über den der Versand erfolgte, gegenprüft. Sollten diese beiden Domains nicht übereinstimmen, wird die Nachricht i.A. als Spam klassifiziert.

Kapitel 6

Ausblick

SMC stellt lediglich die *Proof of Concept*-Implementierung einer symmetrischen E-Mail-Verschlüsselung dar, und die dabei entstandenen Möglichkeiten sind weitem nicht bis zur Vollendung ausgereizt.

So wäre es durchaus denkbar das SMC-Protokoll weiter zu verfeinern und zu standardisieren und als Plugin für gängige E-Mail-Clients wie *Outlook*, *Thunderbird*, *Lotus Notes* oder *Mutt* zu implementieren.

Auch der Erhalt von INVITATION-REPLY-Nachrichten um sicherzustellen ob die Anfrage nach einem Schlüsseltausch korrekt beendet worden ist wird bei SMC nicht überprüft. Im Gegensatz zum TCP-Protokoll, welches über die *Sliding Window*-Technik nach einer gewissen Anzahl an erhaltenen Paketen in Abhängigkeit von der gewählten Fenstergröße ein Acknowledge-Paket zum Sender zurückschickt, ist eine solche Überprüfung vom Erhalt der Nachrichten weder bei SMTP noch bei POP3 vorgesehen (jedoch ursprünglich in der Definition von POP1 [Rey84]).

Technisch wäre es problemlos realisierbar von Sender zu Empfänger eine weitere SMC-Nachricht zu übermitteln welche verifiziert dass der Schlüssel von beiden Kommunikationspartnern korrekt errechnet worden ist, jedoch würde dies den gesamten Invitation-Prozess um einen weiteren Zyklus verlängern.

Zur Umsetzung von verteilten $1 : n$ E-Mail-Kommunikationen wie sie bspw. bei Mailing-Listen genutzt wird, wäre eine Reimplementierung des Diffie-Hellman-Algorithmus wünschenswert wie sie in [Sch05] beschrieben wird.

Anhang A

Die Module von SMC

Die von SMC implementierten Module in alphabetischer Reihenfolge:

`smc_config`

Definiert die zum erfolgreichen Start von SMC notwendigen Konfigurationswerte. Hierzu zählen die aus den jeweiligen Konfigurationsdateien entnommenen Werte für SMTP Server, SMTP Account, E-Mail Adresse, POP Server, POP Passwort sowie das Heimatverzeichnis des ausführenden Benutzers.

`smc_crypt`

Beinhaltet sämtliche notwendigen kryptographischen Subroutinen. Insbesondere die Generierung von den erforderlichen Primzahlen, die Errechnung der Schlüssel, die Wrapper-Funktionen für die einzelnen Ver- und Entschlüsselungsprozeduren sowie die Auffüllalgorithmen für Datenwerte ungleich eines Vielfachen der Schlüssellänge.

`smc_defines`

Beinhaltet die globalen Variablen- sowie Konstantendefinitionen für sämtliche Module. Diese beinhalten (in Abhängigkeit vom Heimatverzeichnis des Benutzers) die Pfade Dateien in denen ein- und ausgehende Nachrichten gespeichert werden. Darüber hinaus werden hier die Standard-Textausgaben definiert, was eine Internationalisierung denkbar vereinfacht.

`smc_io`

Implementiert die für SMC notwendigen Ein-/Ausgabe-Methoden zur Benutzerinteraktion und Dateiverarbeitung.

`smc_pop`

Realisiert den Zugriff und die Kommunikation über POP-Services, sowie die lokale

Abbildung der geholten Nachrichten auf dem Dateisystem.

smc_smtp

Realisiert den Zugriff und die Kommunikation über SMTP-Services. Auch die Erstellung von Einladungen und deren Antworten ist hier implementiert.

Anhang B

Konfigurationsdateien von SMC

SMC verwendet zur Schlüssel-, Adress-, und Programmverwaltung folgende Konfigurationsdateien und -verzeichnisse:

- `~/ .smc/`

Das `.smc` Verzeichnis bildet das Repositorium für sämtliche Konfigurationsdateien sowie die Mailbox für alle ein- und ausgehende E-Mails. Die Struktur der Unterverzeichnisse lautet wie folgt:

- `./sent/`
Gesendete Nachrichten.
- `./incoming/smc_messages/`
Empfangene SMC-verschlüsselte Nachrichten.
- `./incoming/smc_invitations/`
Empfangene SMC-Einladungen.
- `./incoming/smc_invitation_replies/`
Empfangene SMC-Einladungs-Antworten.
- `./incoming/none_smc/`
Empfangene Nachrichten welche nicht über SMC verschlüsselt wurden.

- `smc.conf`

Hier werden alle zum Zugriff auf POP- und SMTP-Services notwendigen Informationen. Diese können auch innerhalb des SMC-Clientprogramms geändert werden (Benutzerkonten, Passwörter, Server-Adressen, etc.). Symmetrisch verschlüsselt.

- **smc.addresses**

Bildet das Adressbuch von SMC und beinhaltet für jeden Adressaten mit dem bereits der Schlüsseltausch vollzogen wurde ein Tupel mit E-Mail Adresse und Schlüssel. Symmetrisch verschlüsselt.

- **smc.inv_sent**

Wird eine Invitation-Message an einen Kommunikationspartner gesendet, so dient diese Datei als Zwischenspeicher für die gesendeten p , g , pk_A sowie $privkey_A$ Werte um den Schlüssel nach Erhalt von pk_B errechnen zu können. Einträge in dieser Datei werden nach erfolgreicher Kalkulation des finalen Schlüssels gelöscht. Symmetrisch verschlüsselt.

- **smc.passwd**

Beinhaltet den Hashwert des Aktivierungsschlüssels. Unverschlüsselt, jedoch i.d.R. unlesbar.

Anhang C

Auszüge aus dem Quellcode von SMC

C.1 Der Schlüsseltausch - Generierung der Einladung

In den Zeilen 1 bis 4 werden die für die Mathematik hinter Diffie-Hellman benötigten Werte wie bereits o.a. generiert. Es handelt sich hier zumeist um Primzahlen, welche vom SMC Crypto-Modul erzeugt werden. Hierfür werden zuerst die Werte für g sowie p erstellt, welche im Anschluss über die Funktion `gen_keys()` unser Schlüsselpaar liefern sollen. Das zuletzt instanzierte Objekt `savedata` dient lediglich zur lokalen Speicherung der Werte, welche an den Adressaten gesendet werden.

Die Zeilen 7 bis 10 erstellen aus den errechneten Werten eine für die SMC(-kompatible) Applikation beim Empfänger, einen String welcher valid zum SMC Protokoll über SMTP verschickt werden kann.

Der Rest des Codes beschäftigt sich mit dem eigentlichen Versenden der Nachricht. Über die Instanz eines SMTP-Objekts wird diese aufgebaut und zum Server versandt.

```
1 @msg = ();
2 my $p = &smc_crypt::gen_p,$g = &smc_crypt::gen_g;
3 my @keypair = &smc_crypt::gen_keys($p, $g);
4 my $savedata = "$rec:$g:$p:$keypair[0]:$keypair[1]";
5
```

```

6 print "Generating_initial_invitation_message...";
7 push @msg, "smc!INVITATION!smc-end\n",
8   "smc!request-from:$config{'smtp_sender'}!smc-end\n",
9   "smc!p=$p!smc-end\n", "smc!g=$g!smc-end\n",
10  "smc!pk=$keypair[1]!smc-end\n";
11
12 if($ok = &smc_io::get_char eq 'y') {
13   my $smtp = Net::SMTP->new($config{'smtp_server'},
14     hello=>$config{'smtp_account'}, Debug=>0);
15   die "Error_connecting_to_server" unless defined $smtp;
16
17   $smtp->mail($config{'smtp_sender'}) ? print "Sender_OK\n"
18     : die "Sender_address_not_accepted.\n";
19   $smtp->recipient($rec) ? print "Recipient_OK\n"
20     : die "Recipient_address_not_accepted.\n";
21   $smtp->data(@msg) ? print "Message_delivered_to_$rec\n"
22     : die "Error_delivering_message.\n";
23   $smtp->quit();

```

C.2 Dekodierung einer SMC Einladung

Die Dekodierung der SMC Einladung beim Empfänger verläuft größtenteils über reguläre Ausdrücke. Die Zeilen 1 bis 3 versuchen die zuvor vom POP-Server empfangenen Nachrichten zu öffnen. Die Zeile 6 iteriert jede Zeile innerhalb der empfangenen E-Mail nach Attributen welche SMC-Werte beinhalten können. Wurden diese gefunden, werden sie in lokale Variablen gespeichert und am Ende auf Gültigkeit verifiziert. Die letzte Zeile errechnet den finalen Schlüssel zwischen den beiden Parteien.

```

1 open FILE, "$smc_defines::CFG_DIR_INVMAILS/$file"
2   or die "Cannot_open_file:_$!"; my @filedata = <FILE>;
3 close FILE;
4
5 my ($p, $g, $pk, $from);
6 foreach my $fileline (@filedata){
7   if($fileline =~ /^smc!p=(.*)!smc-end/ and $p = $1){
8     print "Found_value:_$1\n" if $smc_defines::DEBUG;
9   } elsif ($fileline =~ /^smc!g=(.*)!smc-end/ and $g = $1){

```

```
10     print "Found_g_value:_$1\n" if $smc_defines::DEBUG;
11 } elsif ($fileline =~ /^smc!pk=(.*)!smc-end/ and $pk = $1){
12     print "Found_public_key:_$1\n" if $smc_defines::DEBUG;
13 } elsif($fileline=~/^smc!request-from:(.*)!smc-end/ and $from=$1){
14     print "Found_sender_address:_$1\n" if $smc_defines::DEBUG;
15 }
16 }
17
18 [...]
19 die "Invitation_seems_to_be_corrupted,_skipping.\n" if (!defined($g)
20 || !defined($p) || !defined($pk) || !defined($from));
21 my @pair = &smc_crypt::calc_finalkey($p, $g, $pk);
```

Literaturverzeichnis

- [ABK98] Ross Anderson, Eli Biham, and Lars Knudsen. *Serpent: A proposal for the advanced encryption standard*. Cambridge University, Technion Haifa Israel, University of Bergen Norway, 1998.
- [AKS04] M. Agrawal, N. Kayal, and N. Saxena. *Primes is in p*. Department of Computer Science and Engineering, Institute of Technology Kanpur, India, 2004. (URL: <http://www.cse.iitk.ac.in/users/manindra/algebra/primality.pdf>).
- [Bau05] Michael Bauer. *Sichere Server mit Linux*. O'Reilly, 2005.
- [BPC⁺85] M. Butler, J. Postel, D. Chase, J. Goldberger, and J. K. Reynolds. *Post office protocol - version 2*, 1985. (URL: <http://www.ietf.org/rfc/rfc937.txt>).
- [Con02] Michael Connell. *Python vs. Perl vs. Java vs. C++ runtimes*, 2002. (URL: <http://furryland.org/mikec/bench/>).
- [DR01] John Daemen and Vincent Rijmen. *The design of Rijndael*. Springer, 2001.
- [Ert03] Wolfgang Ertel. *Angewandte Kryptographie*. Hanser Fachbuchverlag, 2003.
- [GC04] John Graham-Cumming. *Die Tricks der Spammer*. Hakin9 Magazine, 2004. Ausgabe 3/2004.
- [HL02] Brian Hatch and James Lee. *Hacking exposed. Linux security secrets and solutions*. McGraw-Hill Professional, 2002.
- [HW05] Robert Harris and Christel Wiemken. *Enigma*. Weltbuch Verlag, 2005.

- [KBC97] H. Krawczyk, M. Bellare, and R. Canetti. Rfc 2104: Hmac: Keyhashing for message authentication, 1997. (URL: <http://www.ietf.org/rfc/rfc2104.txt>).
- [LV00] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. Citibank N.A., Techn. Universiteit Eindhoven, 2000.
- [MSK02] Stuart McClure, Joel Scambray, and George Kurtz. *Hacking exposed*. Mcgraw-Hill Professional, 2002.
- [Pac05] Lars Packschies. *Praktische Kryptographie unter Linux*. Open source press, 2005.
- [PC04] Cyrus Peikari and Anton Chuvakin. *Security Warrior*. O'Reilly, 2004.
- [PF02] Bruce Potter and Bob Fleck. *802.11 Security*. O'Reilly, 2002.
- [Pos82] Jonathan B. Postel. Rfc 821: Simple mail transfer protocol, 1982. (URL: <http://www.ietf.org/rfc/rfc821.txt>).
- [PW06] Johannes Plötner and Steffen Wendzel. *Netzwerksicherheit*. Galileo press, 2006.
- [RC00] Ryan Russel and Stace Cunningham. *Hacking exposed*. Osborn Media, 2000.
- [Rey84] J. K. Reynolds. Post office protocol, 1984. (URL: <http://www.ietf.org/rfc/rfc918.txt>).
- [Rie94] H. Riesel. *Prime numbers and computer methods for facorization*. Birkhaeuser, 1994.
- [Riv94] Ron Rivest. Rc4 algorithm revealed, 1994. (URL: <http://groups.google.com/group/sci.crypt/msg/10a300c9d21afca9>).
- [Ros88] M. Rose. Post office protocol - version 3, 1988. (URL: <http://www.ietf.org/rfc/rfc1081.txt>).
- [Sch94] Bruce Schneier. *Fast software encryption, Cambrindge Security Workshop Proceedings (Dec. 1993)*. Springer, 1994. pp. 191-204.
- [Sch98] Reinhard Schmidt. Skriptum zur Vorlesung Kryptologie, Hochschule Esslingen, 1998. (URL: <http://www.it.fht-esslingen.de/schmidt/vorlesungen/kryptologie/seminar/es9798/html/prim/>).

- [Sch04] Bruce Schneier. *Secrets and Lies. IT-Sicherheit in einer vernetzten Welt*. Dpunkt Verlag, 2004.
- [Sch05] Bruce Schneier. *Angewandte Kryptographie. Algorithmen, Protokolle und Sourcecode in C*. Pearson Studium, 2005.
- [Sin01] Simon Singh. *Geheime Botschaften*. Dtv, 2001.
- [SKW⁺98] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-bit block cipher, 1998.
- [Wae03] Dietmar Waetjen. *Kryptographie. Grundlagen, Algorithmen, Protokolle*. Spektrum Akademischer Verlag, 2003.