Karlsruhe Institute of Technology

Skript zur Stammvorlesung

Sicherheit

Karlsruher Institut für Technologie

Fakultät für Informatik

Institut für Theoretische Informatik
Arbeitsgruppe für Kryptographie und Sicherheit

Die aktuelle Version des Skriptes befindet sich noch im Aufbau, daher kann weder für Vollständigkeit noch Korrektheit garantiert werden. Hinweise zu Fehlern, Kritik und Verbesserungsvorschläge nehmen wir gerne per Mail an skript-sicherheit@ira.uka.de entgegen.

Letzte Änderung: 7. August 2014

Karlsruher Institut für Technologie
Institut für Theoretische Informatik
Arbeitsgruppe für Kryptographie und Sicherheit
Am Fasanengarten 5
76131 Karlsruhe

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Was ist Sicherheit?

Sicherheit bedeutet, dass Schutz geboten wird. Was wird geschützt? Vor wem? Wie wird es geschützt? Wer schützt? Es gibt zwei verschiedene Ansätze, die beide unter den deutschen Begriff *Sicherheit* fallen: *Betriebssicherheit* (engl. safety) und *Angriffsicherheit* (engl. security).

Betriebssicherheit: Unter *Betriebssicherheit* versteht man die Sicherheit einer Situation, die von einem System geschaffen wird: Ist der Betrieb eines Systems sicher? Das bedeutet vor allem, dass keine externen Akteure betrachten werden: Niemand manipuliert das System! Diese Art von Sicherheit wird mit Methoden erreicht, die wahrscheinliche Fehlerszenarien abdecken und verhindern. Beispiele für Systeme, die uns Betriebssicherheit gewähren, sind Arbeitsschutzkleidung zur Vermeidung von Arbeitsunfälle, Backup-Systeme zur Vorbeugung gegen den Ausfall von Komponenten oder elektrische Sicherungen, um uns vor gefährlichen Kurzschlussströmen zu schützen.

Angriffssicherheit: Unter *Angriffssicherheit* versteht man die Sicherheit eines Systems in Bezug auf das externe Hinzufügen von Schäden: Ist es möglich, das System von Außen zu manipulieren? Anders als bei *Betriebssicherheit* betrachten wir keine Schäden, die durch den aktuellen Zustand des Systems entstehen können. Wir betrachten Schäden die von einem externen Akteur, im folgenden *Angreifer* genannt, ausgehen. Dabei gehen wir davon aus, dass ein Angreifer Schwachstellen des Systems gezielt sucht und verwendet. Aus diesem Grund genügt es nicht, wahrscheinliche Fehlerszenarien zu betrachten. Es ist vielmehr nötig, alle Angriffsmöglichkeiten zu unterbinden. Beispiele für diese Art von Sicherheit sind gepanzerte Fahrzeuge, Türschlösser gegen Einbrecher und Wasserzeichen, um das Fälschen von Banknoten zu erschweren.

In dieser Vorlesung beschäftigen wir uns ausschließlich mit dem Konzept der Angriffssicherheit. Darüber hinaus beschäftigen wir uns nur mit dem Schutz informationstechnischer Systeme.

1.2 Grundlagen

Betrachten wir ein informationstechnisches System. Es existierten zahlreiche Arten von Attacken, vor denen wir uns durch verschiedene Techniken schützen müssen. Es ist selten hilfreich, das Gesamtsystem als Einheit zu betrachten. Dafür ist es einfach zu komplex. Stattdessen zerlegen wir es in kleinere „Bausteine“, für deren Sicherheit wir einzeln garantieren können. Diese Vorlesung stellt die wichtigsten Bausteine vor.

In diesem Abschnitt geben wir einen Überblick über die wichtigsten Grundbegriffe. Im Anschluss werden wir stets auf die weiterführenden Kapitel verweisen.

1.2.1 Verschlüsselung

Ziel der Verschlüsselung ist es, Informationen auf einen bestimmten Personengruppe zu begrenzen. Stellen wir uns vor, ein Sender Bob möchte eine Nachricht an eine Empfängerin Alice übermitteln. Die Nachricht ist privat, doch Eve lauscht. Können Alice und Bob kommunizieren, ohne das Eve sinnvolle Informationen erhält? Wie?

Ein Verschlüsselungsverfahren (*Chiffre*) besteht aus einer oder mehreren mathematischen Funktionen, die zur Ver- und Entschlüsselung einer Nachricht eingesetzt werden. Bei der Verschlüsselung wird ein Klartext (eine *Nachricht*) in einen Geheimtext (ein *Chiffrat*) umgewandelt. Das Chiffrat soll einem Dritten keine Informationen über die Nachricht offenbaren. Das Chiffrat kann dann durch Entschlüsselung wieder in den Klartext umgewandelt werden. Verschlüsselung wird auch als *Chiffrierung*, Entschlüsselung als *Dechiffrierung* bezeichnet.
In heutigen Algorithmen wird zur Chiffrierung und Dechiffrierung noch eine weitere Information, der *Schlüssel*, benutzt. Diese Situation ist in Abbildung 1.1 dargestellt. Ist der Schlüssel für Ver- und Entschlüsselung gleich, so spricht man von einem *symmetrischen* Verfahren. Sind die Schlüssel verschieden, handelt es sich um ein *asymmetrisches* Verfahren. Symmetrische Verfahren werden in Kapitel 2 vorgestellt, asymmetrische Verfahren in Kapitel 5.
Klartext und Chiffrat können aus beliebigen Zeichen bestehen. Im Kontext computergestützter Kryptographie sind beide normalerweise binär kodiert.

Für den Fall, das Bob seine Nachricht an Alice vor dem Senden verschlüsselt, können die beiden ihre Kommunikation vor Eve verbergen. Im Gegensatz zu Eve sollte Alice die Nachricht natürlich entschlüsseln können. Ein Chiffrat muss jedoch nicht immer versendet werden. Es kann auch zu Speicherung auf einem Datenträger vorgesehen sein.

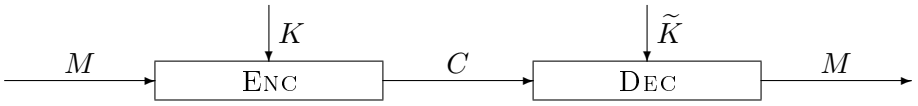


Abbildung 1.1: $\mathrm{DEC}_{\widetilde{\mathrm{K}}}(\mathrm{ENC}_{\mathrm{K}}(\mathrm{M})) = \mathrm{M}$. Falls $K = \tilde{K}$ handelt es sich um ein symmetrisches Verschlüsselungsverfahren, ist $K \neq \tilde{K}$ so ist es ein asymmetrisches Verfahren.

Üblicherweise benutzen wir folgende Abkürzungen:

Nachricht:	M	(engl. *message*)
Chiffrat:	C	(engl. *ciphertext*)
Schlüssel:	K	(engl. *key*)
Chiffrierung:	ENC	(engl. *encryption*)
Dechiffrierung:	DEC	(engl. *decryption*)

1.2.1.1 Geheime Verfahren

Zwar gibt es eine ganze Reihe von Verschlüsselungsverfahren ohne Schlüssel, allerdings hängt deren Sicherheit allein davon ab, dass der Algorithmus geheim bleibt. Im Kontext von Algorithmen, deren Sicherheit auf der Geheimhaltung des Verfahren beruht, spricht man auch von *security by obscurity*. Solche Algorithmen sind unflexibel und aus heutiger

Sicht unsicher. Sie sind daher eher von historischem Interesse und werden im Folgenden nicht näher betrachtet. Stattdessen hat sich Kerckhoffs' Prinzip etabliert.

1.2.1.2 Kerckhoffs' Prinzip

Kerckhoffs' Prinzip ist ein Grundsatz moderner Kryptographe. Er wurde im 19. Jahrhundert von Auguste Kerckhoff formuliert [5].

> The cipher method must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience.

Anders ausgedrückt darf die Sicherheit eines Verschlüsselungsverfahren nur von der Geheimhaltung des Schlüssels und nicht von der Geheimhaltung des Algorithmus abhängen. Kerckhoffs' Prinzip findet in den meisten heutigen Verschlüsselungsverfahren Anwendung. Gründe dafür sind:

- Es ist einfacher, einen Schlüssel als einen Algorithmus geheim zu halten.
- Es ist einfacher, einen kompromittieren Schlüssel zu ersetzen statt einen ganzen Algorithmus zu tauschen. Tatsächlich ist es gängige Sicherheitspraxis, den Schlüssel regelmäßig zu wechseln, selbst wenn dieser nicht bekannt geworden ist.
- Bei vielen Teilnehmerpaaren (z.B. innerhalb einer Firma) ist es um einiges einfacher, unterschiedliche Schlüssel zu verwenden, statt unterschiedlicher Algorithmen für jede Kombination zu erfinden.
- Veröffentlichte Verfahren können von vielen Fachleuten untersucht werden, wodurch eventuelle Fehler wahrscheinlicher auffind- und behebbar sind.
- Da der Schlüssel kein Teil des Algorithmus (bzw. seiner Implementierung) darstellt, ist er im Gegensatz zum Algorithmus nicht anfällig gegen Reverse-Engineering.
- Öffentliche Entwürfe ermöglichen die Etablierung von Standards.

Diese Gründ mögen einleuchtend sein. Trotzdem wurde Kerckhoffs' Prinzip immer wieder zugunsten geheimer Verfahren ignoriert, was zu fatalen Ergebnissen führte. Es sollten nur standardisierte und öffentlich getestete Verfahren verwendet werden.

Kapitel 2

Symmetrische Verschlüsselung

Ein symmetrisches Verschlüsselungsverfahren sichert eine Kommunikation zwischen (typischerweise zwei) Parteien durch einen geheimen Schlüssel, den alle Parteien kennen. Der Schlüssel dient sowohl der Chiffrierung als auch der Dechiffrierung. Er wird keiner bestimmten Partei, sondern einer bestimmten Kommunikationsverbindung zugeordnet. Alle klassischen Verschlüsselungsverfahren sind symmetrisch.

Um eine sichere Kommunikation zu beginnen, müssen sich beide Parteien zuvor auf einen gemeinsamen Schlüssel einigen. Diesen Vorgang nennen wir *Schlüsselaustausch*. Bei *offenen* digitalen Systemen wie dem Internet können wir nicht davon ausgehen, dass die Kommunikationspartner schon vorher in Kontakt standen: Prinzipiell kann jeder an einem offenen System teilnehmen und hat Zugriff auf die im System angebotenen Dienste. Daher muss der Schlüsselaustausch innerhalb des Systems selbst erfolgen. Schlüsselaustauschverfahren betrachten wir erst in Kapitel 8. Der Einfachheit halber gehen wir zunächst davon aus, dass beide Kommunikationspartner bereits über einen gemeinsamen geheimen Schlüssel verfügen.

Eine Verschlüsselungsfunktion erwartet in der Regel eine Eingabe fester Länge. Daher wird ein Klartext beliebiger Länge vor der Verarbeitung in eine Folge von Blöcken oder Zeichen fester Länge aufgeteilt, die dann einzeln chiffriert werden. Wird für jeden Block die Verschlüsselungsoperation mit dem selben Schlüssel verwendet, so spricht man von *Blockchiffren*. Diese werden in Kapitel 2.2 ausführlich behandelt. Als *sequentielle Chiffren* oder *Stromchiffren* bezeichnet man Verschlüsselungsverfahren, bei denen die Klartextzeichen nacheinander mit einem in jedem Schritt variierendem Element eines Schlüsselstrom kombiniert werden.

2.1 Stromchiffren

Wir können einen Klartext M als eine endliche Folge $M = (M_i) = (M_1, M_2, \ldots, M_n)$ von Zeichen M_i aus einem Klartextalphabet auffassen. Eine Stromchiffre verschlüsselt einen Klartext, indem sie jedes Klartextzeichen M_i durch ein Chiffratzeichen C_i aus einem Chiffratalphabet in geeigneten Weise ersetzt. Dabei wird dasselbe Klartextzeichen an verschiedenen Positionen nicht notwendigerweise durch das gleiche Chiffratzeichen codiert: Im Allgemeinen folgt für $i \neq j$ aus $M_i = M_j$ nicht $C_i = C_j$. Eine derartige Zeichenersetzung nennt man auch *polyalphabetische Substitution*.

An dieser Stelle sei erwähnt, dass eine Stromchiffre nicht auf dem ursprünglichen Alphabet des Klartextes arbeiten muss. Sie verwendet jedoch elementare Einheiten „kleiner“ Länge, aus denen der Klartext durch Konkatenation aufgebaut werden kann. Solche Einheiten

nennen wir im folgenden Zeichen.

In den meisten Fällen wird die Verschlüsselungsfunktion ENC mit einer einfachen Funktion realisiert, die unabhängig vom Schlüssel ist. Bei binären Klartextströmen findet häufig die XOR-Funktion Anwendung, der Schlüsselstrom wird also bitweise modulo 2 zum entsprechenden Teil des Klartextes hinzuaddiert ($C_i = M_i \oplus K'_i$). Der Schlüsselstrom $K' = (K'_i) = (K'_1, K'_2, \ldots, K'_n)$ wird dabei durch einen Generator G aus den Schlüssel K erzeugt. Das Verfahren ist in Abbildung 2.1 dargestellt.

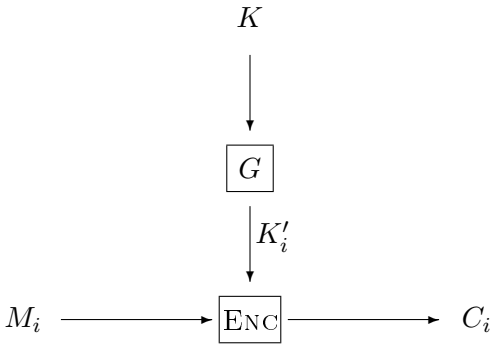


Abbildung 2.1: Prinzip einer Stromchiffre. Der Klartextstrom $(M_1, M_2, \ldots, M_n)$ wird zeichenweise mit einem, aus dem Schlüssel K mit Generator G erzeugten, Schlüsselstrom $(K'_1, K'_2, \ldots, K'_n)$ durch ENC verschlüsselt.

Das klassische Beispiel einer Stromchiffre ist die in Abschnitt 2.1.2 vorgestellte *Vigenère-Chiffre*. Im Gegensatz zur Vigenère-Chiffre bietet eine Stromchiffre, die auf einer wirklich zufälligen Schlüsselfolge basiert, perfekte Geheimhaltung der verschlüsselten Nachricht. Dieses Verfahren heißt *One-Time-Pad* und wird im Abschnitt 2.1.3 vorgestellt.

2.1.1 Caesar-Chiffre

Eine der ersten schriftlich belegten Chiffren ist die Caesar-Chiffre. Der Name stammt vom römischen Feldherrn *Julius Caesar*, der nach Aufzeichnungen des römischen Schriftstellers *Sueton* seine militärische Korrespondenz verschlüsselte indem er jeden Buchstaben des Klartextalphabets um 3 nach rechts verschob.

Beispiel 2.1. *Caesar-Chiffre:*

Klartextalphabet:	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*	*N*	*O*	*P*	*Q*	*R*	*S*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*
Geheimtextalphabet:	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*	*N*	*O*	*P*	*Q*	*R*	*S*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*	*A*	*B*	*C*

Aus dem Klartext „CHIFFRE“ wird das Chiffrat „FKLIIUH“. Zur Entschlüsselung werden die Buchstaben im Geheimtextalphabet entsprechend um 3 nach links verschoben. Das Problem bei dieser Art von Verschlüsselung ist unmittelbar ersichtlich, die Methode verändert sich nicht. Daher kann jeder der einmal erkannt hat, wie Caesar seine Nachrichten verschlüsselte, diese ohne Probleme entschlüsseln. Es gibt keinen Schlüssel und die Sicherheit des Verfahrens hängt allein von der Geheimhaltung der Chiffre ab.

Manchmal wird auch die allgemeine *Verschiebe-Chiffre* als Caesar-Chiffrierung bezeichnet. Bei dieser Chiffre gibt es einen Schlüssel, der die Anzahl der Stellen angibt um die verschoben wird. Für die Verschlüsselung gilt dann: $\text{ENC}_K(M_i) = (M_i + K)\ mod\ 26$ und für die Entschlüsselung: $\text{DEC}_K(C_i) = (C_i - K)\ mod\ 26$. Da es allerdings nur 26 mögliche Schlüssel gibt, ist es selbst ohne Computerunterstützung möglichen jeden Schlüssel auszuprobieren.

Ein solcher Angriff wird als *exhaustive search* oder *Brute-Force-Angriff* bezeichnet.

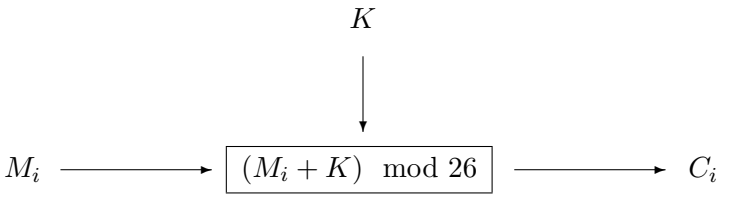


Abbildung 2.2: Prinzip einer Verschiebechiffre. Der Klartextstrom $(M_1, M_2, \ldots, M_n)$ wird zeichenweise um den Schlüssel $K \in \{1, \ldots, 26\}$ verschoben.

Dies führt zu dem wichtigen Prinzip, dass jedes sichere Verschlüsselungsverfahren einen Schlüsselraum besitzen muss, der nicht durch exhaustive search angreifbar ist. Im heutigen Zeitalter, in dem für einen Brute-Force-Angriff mehrere tausende Computer benutzt werden können, muss die Anzahl der möglichen Schlüssel sehr groß sein (mindestens 2^{90} [1]). Es ist jedoch wichtig zu verstehen, dass das obige Prinzip lediglich eine notwendige und keine hinreichende Bedingung für ein sicheres Verschlüsselungsverfahren darstellt.

Interessanterweise ist eine Variante der Caesar-Verschlüsselung heute weit verbreitet. Sie wird *ROT-13* genannt und führt eine Verschiebung um 13 anstatt um 3 Stellen durch. Es ist bekannt das diese Art von Verschlüsselung keine kryptographische Sicherheit bietet. *ROT-13* wird lediglich dazu verwendet, um Spoiler oder Pointen bis zu einer bewussten Entschlüsselung zu verschleiern. Der Vorteil von *ROT-13* besteht darin, dass Ver- und Entschlüsselung exakt die selbe Funktion verwendet, was für eine einfach Implementierung sorgt.

2.1.2 Vigenère-Chiffre

Eine Weiterentwicklung der Caesar-Chiffre, die mehr Sicherheit bietet, ist die sogenannte *Vigenère-Chiffre*, benannt nach einem Franzosen des sechzehnten Jahrhunderts, Blaise de Vigenère. Der Unterschied zur Caesar-Chiffre besteht darin, dass nicht ein konstanter Schlüssel (und damit ein einzelnes Alphabet) zur Chiffrierung jedes einzelnen Zeichens verwendet wird, sondern eine (möglichst lange) Folge von Schlüsseln. Der Zeichenvorrat ist das lateinische Alphabet mit seinen 26 Buchstaben. Die Verknüpfung der Schlüsselfolge mit der Klartextfolge geschieht durch die zeichenweise Addition modulo 26. Als Schlüsselwort dient eine periodisch wiederkehrende Zeichenfolge. Das Wiederholen einer im Verhältnis zum Klartext kurzen Schlüsselfolge ermöglicht allerdings erst die Kryptoanalyse des Vigenère-Systems.

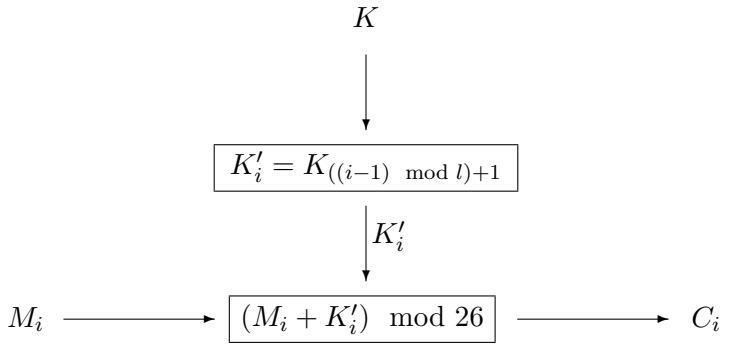


Abbildung 2.3: Prinzip einer Vigenère-Chiffre. Der Schlüsselstrom K'_i entsteht durch Wiederholung des Schlüssels K: $K'_i = K_{((i-1) \bmod l)+1}$ (l ist die Länge des Schlüssels K). Der Klartextstrom $(M_1, M_2, \ldots, M_n)$ wird zeichenweise entsprechend der Verschiebe-Chiffre mit dem entsprechenden Buchstaben aus dem Schlüsselstrom verschlüsselt.

Ist $\underline{k} = (k_1, \ldots, k_l)$ eine Schlüsselfolge der Länge l, so ist die Chiffrierabbildung der Vigenère-Chiffre gegeben durch:

$$E_{\underline{k}} \colon m_1 \ldots m_r \mapsto t_{k_1}(m_1) \ldots t_{k_l}(m_l) t_{k_1}(m_{l+1}) \ldots t_{k_{r \bmod l}}(m_r) \tag{2.1}$$

mit: $m_1 \ldots m_r$ Klartextfolge der Länge r und
$t_{k_j}(m_i) := m_i + k_j \bmod n$, wobei die Indices von k als die
Repräsentanten $1, \ldots, l$ der Restklassen modulo l zu verstehen sind. (2.2)

Beispiel 2.2. *Vigenère-Chiffre:*
Schlüssel: SICHER

Klartext:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Schlüsselfolge:	S	I	C	H	E	R	S	I	C	H	E	R	S	I	C	H	E	R	S	I	C	H	E	R	S	I
Geheimtext:	T	K	F	L	J	X	Z	Q	L	R	P	D	F	W	R	X	V	J	L	C	X	D	B	P	R	I

Der Weg über die Analyse der Häufigkeitsverteilung der Zeichen im Chiffretext (Aufstellen der Histogramme) führt hier nicht zum Ziel, da die Histogramme für lange Schlüssel verflachen, d.h. sich einander angleichen. Daher ist eine Vigenère-Chiffre wesentlich sicherer als eine einfache Substitution von Buchstaben; sie wurde sogar bis Mitte des vorletzten Jahrhunderts für unbrechbar gehalten und als *Le Chiffre indéchiffrable* bezeichnet.

Allerdings ist das Brechen der Vigenère-Chiffre relativ einfach, sobald man die Länge l des Schlüssels kennt. Diese kann man z.B. durch die Kasiski-Friedmann-Methode ermitteln. Nun kann das Chiffrat in l unterschiedliche Teile aufgespalten werden ($t_{k_j}(m_j)$, $t_{k_j}(m_{l+j})$, $t_{k_j}(m_{2 \cdot l+j}), \ldots$ entspricht einem Teil), wobei die Verschlüsselung der einzelnen Teile einer Verschiebe-Chiffre entspricht (jeder enthaltene Chiffretextbuchstabe wurde mit dem selben k_j verschlüsselt), die leicht mit Hilfe von Histogrammen gebrochen werden kann. Die Länge der Schlüsselfoge kann auf folgende Weise bestimmt werden. Man betrachte für $\tau = 1, 2, \ldots$ die entsprechenden Geheimtextbuchstaben $t_{k_j}(m_j), t_{k_j}(m_{\tau+j}), t_{k_j}(m_{2 \cdot \tau+j}), \ldots$ und die folgende Gleichung:

$$S_\tau = \sum_{i=0}^{25} q_i^2 \tag{2.3}$$

Wobei q_i die Anzahl der Vorkommen des i-ten Buchstaben des Alphabets in der Sequenz geteilt durch die Summe aller Buchstaben der Sequenz ist. Sollte für die Schlüsselänge l, $l = \tau$ gelten, so wäre zu erwarten, dass S_τ ungefähr den gleichen Wert hat wie unter den Wahrscheinlichkeiten eines natürlichsprachlichen Textes (da eine Verschiebe-Chiffre die Häufigkeitsverteilung nicht verschleiert), was ungefähr 0.075 entspricht. Für $l \neq \tau$ ist dagegen zu erwarten, dass alle Buchstaben mit ungefähr gleicher Wahrscheinlichkeit in der Folge $t_{k_j}(m_j)$, $t_{k_j}(m_{\tau+j})$, $t_{k_j}(m_{2\cdot\tau+j}), \ldots$ auftreten. Das bedeutet für alle i gilt: $q_i \approx 1/26$.

$$S_\tau \approx \sum_{i=0}^{25} (1/26)^2 \approx 0.038 \tag{2.4}$$

Was einen ausreichenden Unterschied darstellt damit diese Methode funktioniert. Abschließend bleibt noch zu erwähnen, dass für einen solchen Angriff das Chiffrat selbstverständlich eine gewisse Länge aufweisen muss.

2.1.3 One-Time-Pad

Das One-Time-Pad ist wie bereits gesagt eine spezielle Form einer Stromchiffre und wie im Folgenden definiert:

- Der zu Verschlüsselung verwendete Schlüssel K besitzt die gleich Länge l wie der Klartext M.
- Der Schlüssel wird gleichverteilt aus dem Schlüsselraum $K = \{0,1\}^l$ ausgewählt. Jeder beliebige Schlüssel wird also mit einer Wahrscheinlichkeit von $\frac{1}{2^l}$ ausgewählt.
- Zur Verschlüsselung wird der Klartext und der Schlüssel bitweise mit XOR verknüpft: $\forall i : C_i = M_i \oplus K_i$.
- Zur Entschlüsselung wird das Chiffrat und der Schlüssel bitweise mit XOR verknüpft: $\forall i : M_i = C_i \oplus K_i$.
- Der Schlüssel darf weder vollständig noch teilweise wiederverwendet werden.

Ist die obige Definition erfüllt, bietet das One-Time-Pad bewiesenermaßen perfekte Geheimhaltung.

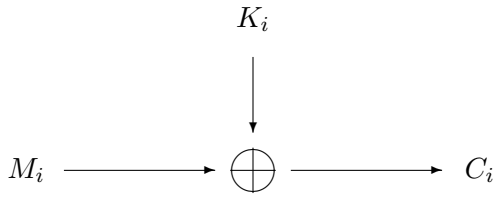


Abbildung 2.4: Prinzip eines One-Time-Pads. Der Klartextstrom $(M_1, M_2, \ldots, M_n)$ wird zeichenweise mit den Schlüsselstrom $(K_1, K_2, \ldots, K_n)$ binär addiert (auch XOR genannt).

Allerdings hat das One-Time-Pad auch etliche Nachteile. Der Hauptnachteil besteht darin, dass der Schlüssel echt zufällig gewählt und genauso lang sein muss, wie das Chiffrat, was das Problem der sicheren Übertragung der Nachricht in das Problem der sicheren Übertragung des Schlüssels umwandelt. Weiterhin ist es zeitlich und/oder technisch aufwändig „echten Zufall“ zu erzeugen, da dieser aus physikalischen Phänomenen erhalten wird. Außerdem bietet das Verfahren zwar perfekte Geheimhaltung allerdings absolut keinen Schutz gegen eine Veränderung der Nachricht.

Die obigen Gründe machen die Verwendung des One-Time-Pad sehr aufwändig, weswegen es auch nur sehr selten eingesetzt wird. Moderne Stromschiffren funktionieren prinzipiell wie ein One-Time-Pad, benutzen jedoch einen Pseudozufallszahlengenerator, der aus einem kurzen Seed, dem Schlüssel, eine Folge aus Pseudozufallszahlen erzeugt. Diese Art von Chiffren wird kurz im nächsten Abschnitt vorgestellt.

2.1.4 Stromchiffren mit Pseudozufallszahlen

Wir wissen bereits, dass die Zufallsfolge, die beim One-Time-Pad als Schlüssel dient, mindestens so lang sein muss wie die zu verschlüsselnde Nachricht und nur ein einziges Mal verwendet werden darf. Hieraus folgt, dass dieses Verfahren einen extrem hohen Aufwand für die sichere Schlüsselverteilung erfordert und aus diesem Grund für die meisten Anwendungen unpraktikabel ist. Es liegt nun nahe, die genannte Schwierigkeit zu umgehen, indem man nach dem Vorbild des One-Time-Pad Stromchiffren konstruiert, die statt einer wirklichen Zufallsfolge sogenannte Pseudozufallsfolgen verwenden. Unter einer *Pseudozufallsfolge* versteht man dabei eine Folge von Zeichen, die mittels eines deterministischen Prozesses aus einem relativ kurzen Initialisierungswert erzeugt wird und gewisse Eigenschaften einer echt zufälligen Folge aufweist. Wie im letzten Abschnitt angesprochen wird dieser Initialisierungswert *Seed* genannt. Wenn beide Kommunikationspartner über identische Generatoren verfügen, muss nur noch der Initialwert und die gewählte Parametrisierung des Generators als Schlüssel verteilt werden. Die eigentliche Schlüsselfolge kann dann an beiden Enden des Kanals erzeugt werden. Eine Voraussetzung der Konstruktion ist offensichtlich, dass der Pseudozufallsgenerator effizient berechenbar ist. Außerdem soll hier noch auf den Umstand hingewiesen werden, dass es sich bei der Schlüsselfolge nicht um den Schlüssel des Verfahrens handelt, da die Folge ein Menge von internen Werten des Algorithmus ist. In Abbildung 2.5 ist der prinzipielle Aufbau einer derartigen Stromchiffre gezeigt.

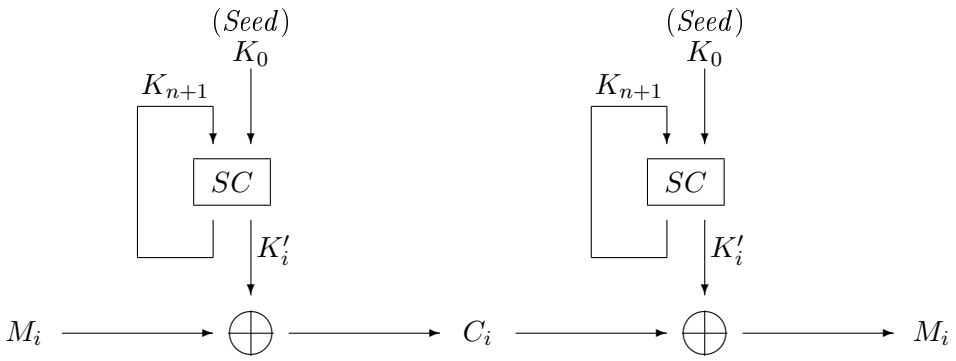


Abbildung 2.5: Prinzip einer Stromchiffre mit Pseudozufall. Der Klartextstrom wird zeichenweise mit einem aus dem *Seed* K_0 generierten pseudozufälligen Schlüsselstrom ver- und entschlüsselt. Wichtig ist dass sowohl bei Ver- als auch Entschlüsselung der selbe *Seed* und die selbe Funktion SC verwendet wird. Nennenswert ist außerdem, dass die Funktion SC nicht in jedem Iterationsschritt ein Schlüsselbit erzeugen muss, weshalb die Zählvariablen i und n nicht synchron sein müssen.

Eine einfache und beliebte Möglichkeit zur Implementation bieten *Linear Feedback Shift Register* (*LFSR*). Die Arbeitsweise ist in Abbilding 2.6 dargestellt und funktioniert wie folgt: Der Mechanismus besitzt einen Zustand, welcher in k Registern gespeichert ist. In einem Arbeitsschritt wird die Summe der Produkte der Register und einem je Register konstantem Koeffizienten α gebildet. Dieses Ergebnis wird nun im höchstwertigsten Register gespeichert. Der Inhalt dieses Registers wird dabei im nächsten Register gespeichert

usw. Damit werden also alle Register „weitergeschoben“, wobei das niederwertigste Register ausgegeben wird, d.h. der Inhalt des Registers wird nun als Teil des Schlüsselstroms zur Verschlüsselung der Nachricht verwendet.

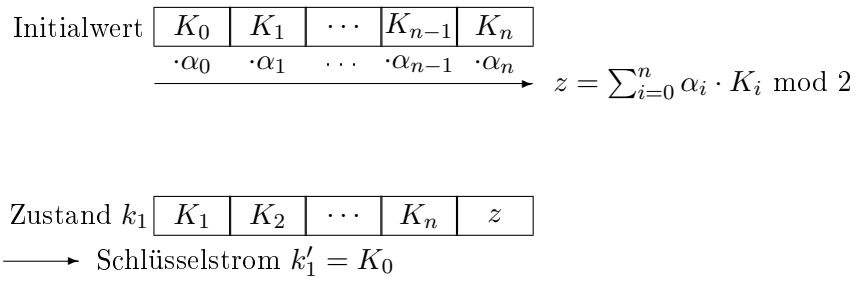


Abbildung 2.6: Prinzip von *Linear Feedback Shift Registern*. In jedem Schritt wird z aus den in den Registern vorhandenen Schlüsselbits und den Konstanten α_i berechnet. z ersetzt dann das Schlüsselbit K_n, welches an Stelle von K_{n-1} rückt. K_{n-1} selbst rückt wiederum an Stelle von K_{n-2}, usw. Schließlich wird K_0 durch K_1 ersetzt. K_0 wird ausgegeben und als nächstes Bit im Schlüsselstrom verwendet.

Wenn man für die Zustände der *LFSR* nun die Gestalt $(K_1, K_2, \cdots, K_n)^T$ wählt, lassen sich alle Zustände wie folgt darstellen:

$$k_{i+1} = A \cdot k_i, \quad A := \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & & 0 \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & 0 & & 1 \\ \alpha_1 & \alpha_2 & \alpha_3 & \cdots & \alpha_n \end{pmatrix}$$

Daraus ergibt sich für den Schlüsselstrom:

$$\begin{aligned} k'_{i+1} &= (1, 0, \cdots, 0) \cdot k_i \\ &= (1, 0, \cdots, 0) \cdot (A^i \cdot k_0) \\ &= ((1, 0, \cdots, 0) \cdot A^i) \cdot k_0 \end{aligned}$$

Die Beziehung des *LFSR* zu einem Klartext-Chiffrat-Paar lässt sich nun als Gleichungssystem darstellen:

$$\begin{array}{lllllllllll} C_1 & = & M_1 & \oplus (& (1,0,\cdots,0) \cdot A^0 &) \cdot k_0 & = & M_1 & \oplus & (1,0,\cdots, & 0) & \cdot k_0 \\ C_2 & = & M_2 & \oplus (& (1,0,\cdots,0) \cdot A^1 &) \cdot k_0 & = & M_2 & \oplus & (0,1,0,\cdots, & 0) & \cdot k_0 \\ & \vdots & & & & & \vdots & & & & & \\ C_n & = & M_n & \oplus (& (1,0,\cdots,0) \cdot A^{n-1} &) \cdot k_0 & = & M_n & \oplus & (0,\cdots, & 0,1) & \cdot k_0 \\ C_{n+1} & = & M_{n+1} & \oplus (& (1,0,\cdots,0) \cdot A^n &) \cdot k_0 & = & M_{n+1} & \oplus & (\alpha_1,\alpha_2,\cdots, & \alpha_n) & \cdot k_0 \\ C_{n+2} & = & M_{n+2} & \oplus (& (1,0,\cdots,0) \cdot A^{n+2} &) \cdot k_0 & = & M_{n+2} & \oplus & (\alpha_1,\alpha_2,\cdots, & \alpha_n) \cdot A^1 & \cdot k_0 \\ & \vdots & & & & & \vdots & & & & & \end{array}$$

Besitzt der Angreifer also ein Klartext-Chiffrat-Paar mindestens der Länge n (die Anzahl der Register des *LFSR*s), kann er dadurch direkt k_0 berechnen. Entsprechend ist ein solches Schieberegister alleine eher unsicher. Hilfe bietet hier eine möglichst strukturzerstörende Verbindung mehrerer Schieberegister. Beispielsweise könnte man zwei *LFSR* verwenden, wobei das zweite *LFSR* nur dann arbeitet, wenn die Ausgabe des ersten *LFSR* 1 ist.

Das Thema wird in der Vorlesung „symmetrische Verschlüsselungsverfahren“ tiefer behandelt.

2.2 Blockchiffren

2.2.1 Funktionsweise

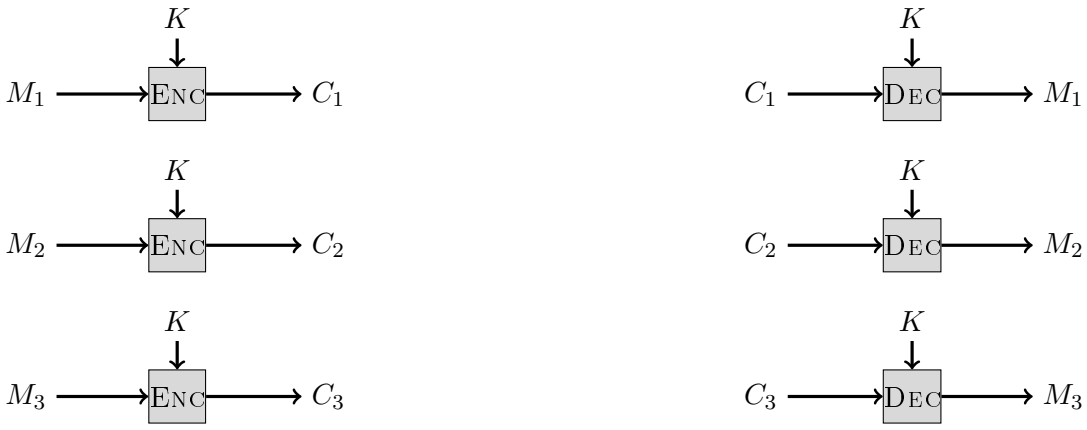
Im Gegensatz zu Stromchiffren, werden bei Blockchiffren in jedem Schritt nicht ein Zeichen sondern eine feste Anzahl, ein Block, an Zeichen verschlüsselt. Schematisch ergibt sich nahezu das selbe Bild wie bei Stromchiffren (siehe Abbildung 2.1), allerdings ergeben sich gerade in der Implementierung große Unterschiede.

Einerseits kann die tatsächliche Verschlüsselungsfunktion nun komplexer sein als ein einfaches XOR, da es bei Blöcken mehr Möglichkeiten für Strukturänderungen gibt. Andererseits benötigen diese Verfahren natürlich mehr Rechenleistung als die Schieberegister und XOR-Netze von Stromchiffren, wodurch der Datendurchsatz sinkt. Außerdem bieten komplexere Verfahren mehr Möglichkeiten für Angreifer, weshalb es auch schwieriger ist, eine beweisbare Sicherheit festzustellen.

2.2.2 Betriebsmodi

Da die Verschlüsselungsfunktion stets Blöcke einer festen Länge erwartet, die Nachrichten allerdings variierende Länge besitzen, gibt es zum Verarbeiten der Nachrichten mehrere Möglichkeiten, genannt Betriebsmodi.

2.2.2.1 ECB - Electronic Codebook



(a) Verschlüsselung im ECB

(b) Entschlüsselung im ECB

Abbildung 2.7: Skizze der Verschlüsselung einer Nachricht $M = M_1M_2M_3$ sowie der Entschlüsselung des zugehörigen Chiffrats $C = C_1C_2C_3$ im Electronic Codebook Mode. Grafik erstellt von Martin Thoma [12].

Beim ECB wird der Klartext in Blöcke mit einer festen Länge von n *bits* aufgeteilt und jeder Block unabhängig von den anderen einzeln verschlüsselt. In den letzten Block wird dabei zusätzlich die Gesamtzahl von Blöcken codiert und falls notwendig, wird dieser Block, um die Blockgröße zu erhalten, z.B. mit Nullen oder besser noch mit einer Zufallsfolge aufgefüllt. Identische Klartextblöcke liefern damit auch identische Chiffretextblöcke; daher wird diese Betriebsart in Analogie zu einem Code-Buch als Electronic Codebook Mode bezeichnet.

Diese Betriebsart hat im Hinblick auf die Sicherheit wenigstens zwei Nachteile:

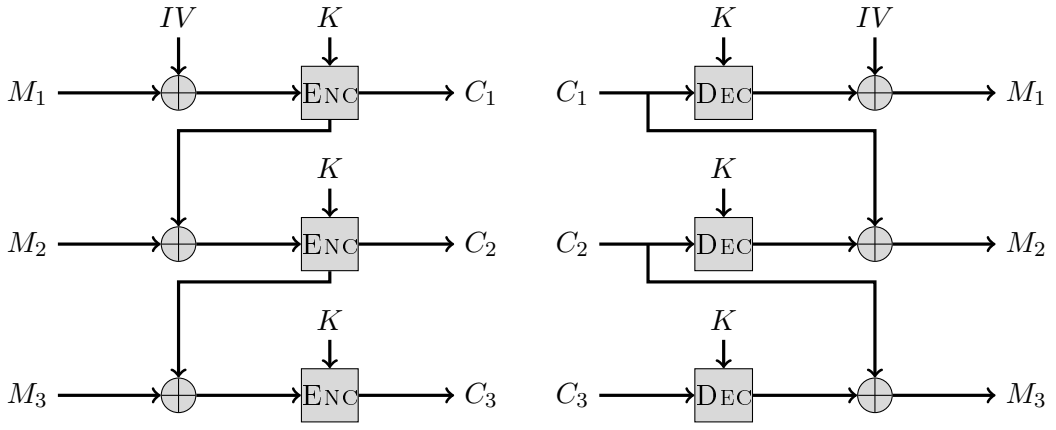
– Da gleiche Klartextblöcke, verschlüsselt mit dem gleichen Schlüssel, zu gleichen Chiffretextblöcken führen, kann ein passiver Angreifer Information über den Klartext folgern, obwohl der Angreifer die Blöcke selbst nicht entschlüsseln kann.

– Der zweite, schwerwiegendere Nachteil ist darin zu sehen, dass ein Angreifer den Chiffretext selbst ändern kann, ohne dass der Empfänger der Nachricht dies bemerkt. Chiffretextblöcke, die mit dem gleichen Schlüssel chiffriert und bei vorausgegangenen Übertragungen aufgezeichnet wurden, könnten z.B. eingefügt werden, um den Sinn einer Nachricht zu ändern.

Aufgrund dieser Nachteile ist der ECB-Modus ungeeignet, um lange Nachrichten zu verschlüsseln. Tritt ein Bitfehler bei der Übertragung in Block C_i auf, so ist wegen der Unabhängigkeit der Chiffretextblöcke untereinander nur der Block C_i gestört, d.h. bei der Dechiffrierung erhält man i. allg. einen total gestörten Klartextblock. Alle folgenden Blöcke werden wieder korrekt dechiffriert. Es gibt also keine Fehlerfortpflanzung.

2.2.2.2 CBC - Cipher Block Chaining

Im CBC-Modus wird eine Nachricht genau wie im ECB-Modus zuerst in Blöcke gleicher Länge zerlegt. Wie in Abbildung 2.8 gezeigt, benutzt das CBC-Verfahren die Ausgabe eines jeden Chiffrierschrittes, um den folgenden Block „vorzuchiffrieren“. Für Anwendungen, wie die Festplattenverschlüsselung, ist es daher problematisch, auf CBC zu setzen: Zwar ist ein wahlfreier Lesezugriff - also das Entschlüsseln - auf den Chiffratblock C_i mit Kenntnis von C_{i-1} möglich, jedoch müssen für das Schreiben eines Blocks M_i alle Klartextblöcke neuverschlüsselt werden müssen. In der Praxis gibt es dennoch Varianten der Festplattenverschlüsselung, die CBC nutzen. Beispielsweise löst der *Linux Unified Key Setup (LUKS)* das Problem, indem 512 Byte große Datenblöcke jeweils einzeln verschlüsselt werden.



(a) Verschlüsselung im CBC (b) Entschlüsselung im CBC

Abbildung 2.8: Skizze der Verschlüsselung einer Nachricht $M = M_1M_2M_3$ sowie der Entschlüsselung des zugehörigen Chiffrats $C = C_1C_2C_3$ im Cipher Block Chaining Mode. Grafik erstellt von Martin Thoma [12].

Man erhält im CBC Mode die Chiffretextblöcke $C_0, C_1, C_2, \ldots$ durch:

$$\begin{aligned} C_0 &:= IV \text{(Initialisierungsvektor)} \\ C_i &:= \text{ENC}_K(M_i \oplus C_{i-1}). \end{aligned}$$

Der erste Block M_1 und ein Initialisierungsvektor IV (Es gibt ja zu diesem Zeitpunkt noch keinen verschlüsselten Chiffretextblock) werden bitweise modulo 2 addiert, das Ergebnis wird dann wie im ECB Mode verschlüsselt und ergibt den ersten Chiffretextblock C_1. Dieser und der zweite Nachrichtenblock M_2 werden bitweise modulo 2 addiert, das Ergebnis wird verschlüsselt, usw. Dieser Vorgang wiederholt sich bis zum Ende der Nachricht.

Die Entschlüsselung geschieht folgendermaßen:

$$\begin{aligned} C_0 &:= IV \\ M_i &:= \text{DEC}_K(C_i) \oplus C_{i-1} \end{aligned}$$

Es gilt:

$$\begin{aligned} M_i &= \text{DEC}_K(C_i) \oplus C_{i-1} \\ &= \text{DEC}_K\big(\text{ENC}_K(M_i \oplus C_{i-1})\big) \oplus C_{i-1} \\ &= M_i \oplus C_{i-1} \oplus C_{i-1}. \end{aligned}$$

Es hängt also jeder Chiffretextblock C_i von den vorausgegangenen Blöcken C_j, $1 \leq j < i$, und vom Initialisierungsvektor IV ab. Damit liefern gleiche Klartextblöcke M_i und M_j $(i \neq j)$ i. allg. verschiedene Chiffretextblöcke C_i und C_j.

Die Wahl des Initialisierungsvektors IV ist wichtig für die Sicherheit dieser Verschlüsselung, denn durch Änderung einzelner Bits des IV können gezielt bestimmte Bits des ersten Blockes verändert werden, der dadurch anfällig für sinnvolle Veränderungen ist.

Aufgrund der Verkettung der Chiffretextblöcke im CBC Mode muss untersucht werden, welche Auswirkungen ein Fehler bei der Übertragung eines Chiffretextblockes hat.

- Tritt ein Bitfehler im Block C_i auf, so zeigt sich der in Abbildung 2.9 dargestellte Effekt. Die Ausgabe bei der Dechiffrierung des Blockes C_i ist rein zufällig, da ein gefälschtes Bit in der Eingabe die Ausgabe völlig verändert. Der ganze Klartextblock M_i ist also zerstört. Durch die Verkettung der Blöcke wird auch noch Block M_{i+1} in Mitleidenschaft gezogen. Im Speicher steht jetzt der bitfehlerbehaftete Chiffretextblock C_i. Durch die Addition modulo 2 wird bewirkt, dass an der Stelle des Bitfehlers im Block C_i im Klartextblock M_{i+1} nun auch ein Bitfehler entsteht. Nachfolgende Blöcke werden aber nicht mehr beeinflusst.

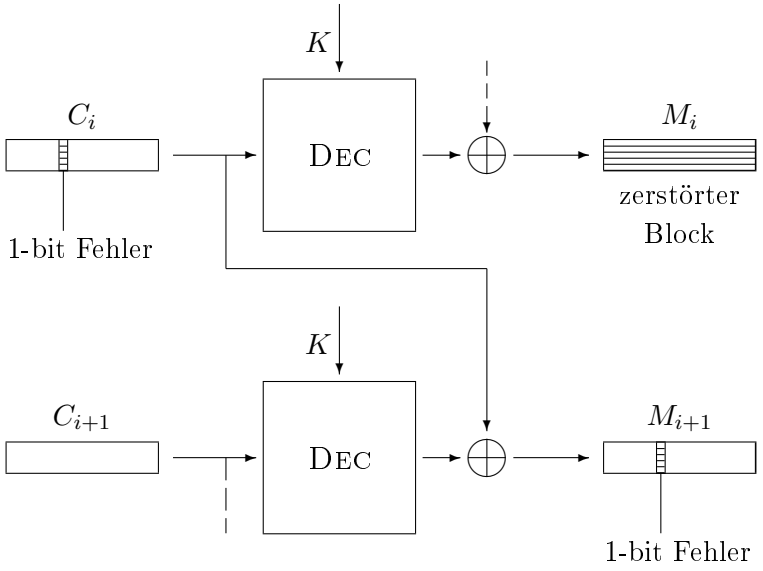


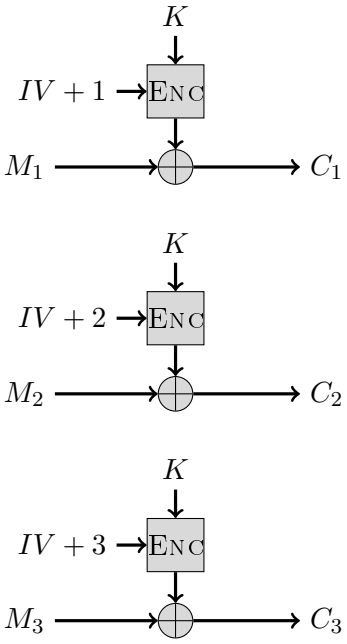
Abbildung 2.9: Fehlererweiterung beim CBC Mode

Der CBC Mode ist selbstkorrigierend, Übertragungsfehler innerhalb eines Blockes wirken sich bei der Entschlüsselung nur auf diesen und den direkt nachfolgenden Block aus. Daraus folgt, dass der Initialisierungsvektor zum Start des Systems zwischen Sender und Empfänger nicht vereinbart sein muss. Wählen Sender und Empfänger je einen zufälligen IV, so kann nur Block M_1 vom Empfänger nicht korrekt wiedergewonnen werden.

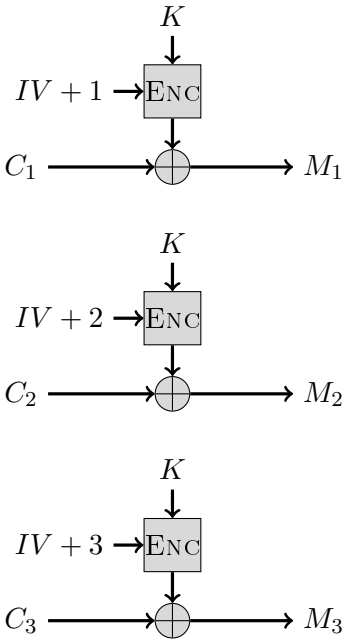
Die eben erläuterte Art der Fehlererweiterung des CBC Mode beinhaltet ein Sicherheitsrisiko. Durch das gezielte Verändern eines Bits im Chiffretext wird zwar der zugehörige Klartextblock völlig zerstört, aber im nächsten Klartextblock wird genau dieses Bit negiert, was von entscheidender Bedeutung sein kann[1]. Nach Auftreten eines Fehlers sollte also die ganze Übertragungsfolge wiederholt und nur dann akzeptiert werden, wenn alle Blöcke korrekt übertragen wurden. Um dies zu prüfen, können auf den Klartext bezogene Prüfnummern verwendet werden.

2.2.2.3 CTR - Counter Mode

Betrachten wir nun einen Betriebsmodus, der die Vorteile von CBC beinhaltet und gleichzeitig das Parallelisieren der Verschlüsselung und Entschlüsselung ermöglicht. Dieser Modus ist der *Counter Mode* (CTR). Er funktioniert wie folgt:



(a) Verschlüsselung im CTR



(b) Entschlüsselung im CTR

Zu einer gegebenen Nachricht $M = M_1 M_2 \dots M_n$ berechnet sich das dazugehörige Chiffrat $C = C_1 C_2 \dots Cn$ durch:

$$C_i \quad := \quad \text{ENC}_K(IV + i) \oplus M_i$$

Analog zu CBC verwendet CTR einen Initialisierungsvektor IV, der zufällig und gleichverteilt vor jedem Verschlüsselungsvorgang gewählt werden muss. Der Unterschied zu CBC liegt in der Verschlüsselung: Zum Verschlüsseln eines Klartext-Blocks wird kein vorher berechneter Chiffrat-Block benötigt. Stattdessen wird für jedes C_i der IV um 1 erhöht; für keine zwei Chiffrat-Blöcke C_i, C_j $(i \neq j)$ wird die gleiche Eingabe an die ENC-Funktion übergeben. Damit stellen wir sicher, dass gleiche Nachrichtenblöcke auf unterschiedliche Chiffrat-Blöcke abgebildet werden. Um einen im CTR-Modus verschlüsselten Text $C = C_1 C_2 \dots Cn$ zu entschlüsseln, gehen wir folgendermaßen blockweise vor:

$$M_i \quad := \quad \text{ENC}_K(IV + i) \oplus C_i$$

Wir sehen, dass das Entschlüsseln, wie bei CBC, parallelisierbar ist und dass der IV bekannt sein muss. Ein wesentlicher Unterschied zu den vorangegangenen Modi ist, dass

[1] Ein erfolgreich durchgeführter Angriff auf diese Schwachstelle ist hier beschrieben.

wir zum Ver- und Entschlüsseln dieselbe Funktion ENC benutzen. ENC muss folglich nicht invertierbar sein.

Betrachten wir die Fehlerfortpflanzung des CTR-Modus, stellen wir fest, dass wir gezielt Bits im Block M_i manipulieren können, indem wir sie in dem entsprechenden Chiffratblock C_i verändern. Mittels CTR verschlüsselte Nachrichten sind somit homomorph veränderbar. Wird hingegen der gewählte IV verändert, erfolgt eine komplette Zerstörung der ursprünglichen Nachricht.

2.2.2.4 Galois/Counter Mode

Der oben vorgestellte CTR-Modus besitzt bereits einige wünschenswerte Eigenschaften, die eine Blockchiffre haben sollte. Ein Nachteil von CTR ist jedoch, dass er homomorph veränderbar ist: Ein Angreifer kann gezielt Bits verändern und dabei unbemerkt bleiben. Wir sind nun auf der Suche nach einem Modus, bei dem keine unbemerkten Veränderungen an der Nachricht M durchgeführt werden können.

Im *Galois/Counter Mode* (GCM) finden wir einen Modus, der die obige Eigenschaft mit den Vorteilen von CTR vereint. Zusätzlich zum Verschlüsseln einer Nachricht M, welches analog zu CTR erfolgt, wird zeitgleich eine Signatur[2] der Nachricht generiert. Das Verschlüsseln von M im GCM liefert uns ein Chiffrat-Signatur-Paar (C, σ), wobei sich die Signatur σ aus C ergibt[3]. σ wird verwendet, um Manipulationen an C zu erkennen. Vereinfacht dargestellt erhalten wir folgende Operationen:

$$EncryptAndAuthenticate(M) := (Encrypt((M), Authenticate(C)) = (C, \sigma)$$

$$DecryptAndAuthenticate(C, \sigma) := \begin{cases} M & Authenticate(C) == \sigma \\ error & \text{sonst} \end{cases}$$

Wir sehen, dass wir nur dann aus einem Chiffrat-Signatur-Paar (C, σ) die dazugehörige Nachricht M erhalten, wenn die Signatur von C identisch mit der übergebenen Signatur σ ist. Wurde also ein Chiffrat C mit Signatur σ zu einem Chiffrat C' mit Signatur σ' verändert, schlägt die Entschlüsselung $DecryptAndAuthenticate(C', \sigma)$ fehl, da die übergebene Signatur nicht mit der Signatur des Chiffrats übereinstimmt.

2.2.2.5 Zusammenfassung

Betrachten wir abschließend eine Gegenüberstellung der hier vorgestellten Betriebsmodi.

[2] Signaturen werden ausführlicher in Kapitel 6 behandelt.
[3] Für genauere Informationen verweisen wir auf die Veröffentlichung der Entwickler.

	ECB	CBC	CTR	GCM
Hauptsächliche Verwendung	Für Nachrichten, die kürzer als ein Block sind	Für Nachrichten, die länger als ein Block sind	Für Nachrichten, die länger als ein Block sind	Für Nachrichten, die länger als ein Block sind und vor Manipulationen geschützt werden müssen
IND-CPA* sicher	Nein	Ja**	Ja**	Ja**
Parallelisierbar	Ja	Nur Entschlüsselung	Ja	Ja, das Signieren selbst aber nicht
Bit-Fehler im Block X	Block X zerstört	Block X zerstört und 1 Bit im Block (X - 1) geändert	1 Bit verändert	1 Bit verändert und geänderte Signatur

* IND-CPA ist ein Sicherheitsbegriff und wird in Abschnitt 3.2 definiert
** Hierfür muss der IV vor jeder Verschlüsselung zufällig gleichverteilt gewählt werden

Die hier vorgestellten Modi sind nur ein Teil einer Vielzahl an Betriebsmodi. Zusammenfassend betrachtet hat jeder Modus mehr oder weniger wünschenswerte Eigenschaften, weshalb man sich gut überlegen sollte, ob der gewählte Modus im Kontext der jeweiligen Anwendung die nötige Sicherheit garantieren kann. In einem Szenario, in dem auch vor aktiven Angriffen Schutz geboten werden soll, kann nur der *Galois/Counter Mode* Sicherheit bieten. Von denen hier vorgestellten Modi ist er der einzige, der aufgrund des Chiffrat-Signatur-Paars neben Vertraulichkeit auch Datenintegrität sicherstellt. Jedoch gibt es noch immer Anwendungen, die den Modus nicht unterstützen.

2.2.3 Verfahren

Nachdem wir nun mehrere Betriebsmodi vorgestellt haben, betrachten wir jetzt Verschlüsselungsalgorithmen (Blockchiffren), die von diesen Modi verwendet werden. Blockchiffren werden von den Betriebsmodi dazu verwendet, die einzelnen Blöcke einer Nachricht zu ver- bzw. entschlüsseln. Formal dargestellt ist eine Blockchiffre eine Funktion

$$\textsc{Enc}\colon \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^l,$$

wobei k die Schlüssellänge und l die Blocklänge ist. Blockchiffren stellen also Permutationen der Menge $\{0,1\}^l$ dar. Bevor wir eine erste Blockchiffre anschauen, müssen wir uns überlegen, welche Eigenschaften eine Blockchiffre aufweisen muss.

Das übergeordnete Design-Kriterium, welches Blockchiffren unterliegen sollen, ist die Nichtunterscheidbarkeit von einer echt zufälligen Funktion. Präziser gesagt darf sich die Permutation einer Blockchiffre nicht von einer echt zufälligen Permutation derselben Menge unterscheiden. Daraus folgt sofort, dass kleine Änderungen in der Eingabe zu großen Änderungen in der Ausgabe einer Blockchiffre führen müssen. Betrachten wir kurz ein Szenario, in dem eine Blockchiffre verwendet wird, die diese Eigenschaft nicht erfüllt: Ein Angreifer ist im Besitz zweier Chiffratblöcke, die sich nur in einem Bit unterscheiden. Er kann nun davon ausgehen, dass sich auch die beiden dazugehörigen Nachrichten nur geringfügig voneinander unterscheiden und kann dieses Wissen für einen Angriff auf das Chiffrat bzw. das Verschlüsselungsverfahren ausnutzen. Wie aber können wir eine Blockchiffre konstruieren, die obige Eigenschaft erfüllt?

Zum Beantworten dieser Frage fordern wir zunächst zwei Eigenschaften[4], die unsere Blockchiffre erfüllen soll: Die erste Eigenschaft besagt, dass jedes Zeichen des Chiffrats von mehreren Teilen des verwendeten Schlüssels abhängig sein soll. Im Englischen wird diese als *confusion* bezeichnet. Sie erschwert es einem Angreifer, Zusammenhänge zwischen einem Schlüssel und eines damit generierten Chiffrates zu erkennen. Die zweite Eigenschaft fordert, dass das Ändern eines einzelnen Zeichens in der Nachricht bzw. dem Chiffrat zu großen Änderungen im Chiffrat bzw. der Nachricht führt. Diese Eigenschaft wird als *diffusion* bezeichnet.

Eine Umsetzung dieser Eigenschaften in eine Blockchiffre führt uns zu dem Konzept der *Feistel networks*. Die Grundidee hinter so einem Netzwerk ist, dass wir unsere Blockchiffre Enc aus mehreren Rundenfunktionen $F_1, F_2, \ldots, F_n$ zusammenbauen, die nacheinander ausgeführt werden. Die einzelnen Funktionen müssen dabei nicht notwendigerweise verschieden sein, wie wir im Abschnitt zu *DES* sehen werden. Die Funktion F_i wird in der i-ten Runde des Algorithmus ausgeführt und ihre Ausgabe dient als Eingabe für die Funktion F_{i+1}. Die Rundenfunktionen sind dabei so konstruiert, dass sich kleine Änderungen in der Eingabe exponentiell über die Runden ausbreiten.

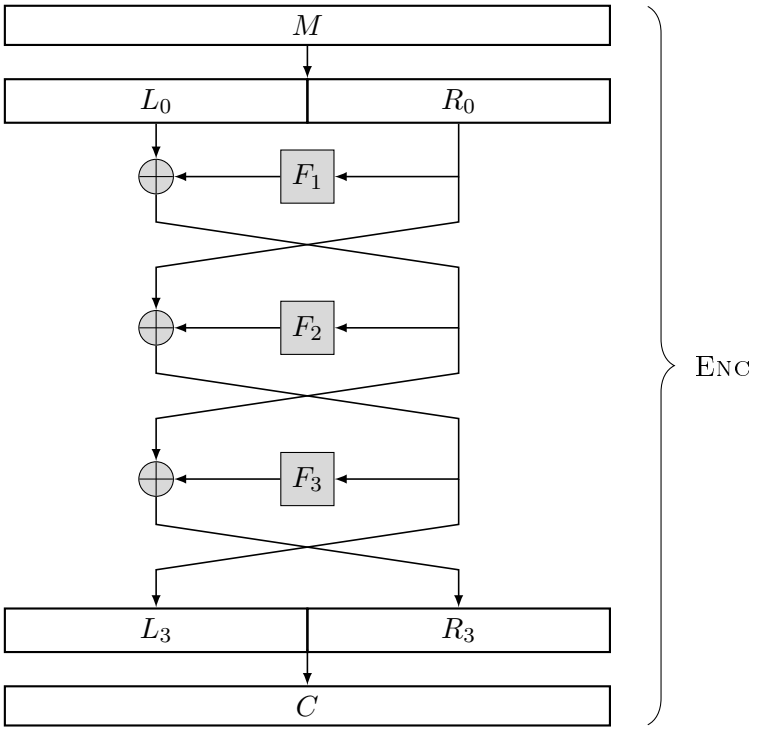


Abbildung 2.10: 3-ründige *Feistel*-Struktur

Eine Rundenfunktion F_i besteht typischerweise aus Permutationen und mehreren Funktionen, auf denen die Eingabe aufgeteilt wird. Diese Funktionen werden als *S-Boxen* (substitution boxes) bezeichnet und sind der Grundbaustein der *Feistel*-Struktur. Die hier betrachteten *S-Boxen* realisieren eine Funktion der Form

$$S\colon \{0,1\}^m \to \{0,1\}^n$$

mit $m > n$. Dabei werden alle m-Bit langen Wörter (2^m viele) auf n-Bit lange Wörter (2^n viele) abgebildet und wir erkennen, dass diese *S-Boxen* nicht invertierbar sind. Als

[4] Beide Eigenschaften wurden zuerst von Claude Shannon in Communication Theory of Secrecy Systems definiert und stellen eine wichtige Grundlage der heutigen Kryptographie dar.

Folgerung ergibt sich, dass die Rundenfunktionen nicht invertierbar sein können. Diese Eigenschaft ist signifikant für die Sicherheit von *Feistel*-Netzwerken und der sie verwendenden Blockchiffren. Zusätzlich zur Nichtinvertierbarkeit haben die *S-Boxen* und die Rundenfunktionen weitere folgende Eigenschaften:

1. Wird in der Eingabe für eine *S-Box* ein Bit verändert, so ändern sich mindestens zwei Bit in der Ausgabe.

2. Die Ausgabe-Bits einer Rundenfunktion F_i werden so permutiert, dass alle Ausgabe-Bits einer *S-Box* auf unterschiedliche *S-Boxen* der nächsten Runde verteilt werden.

Beide Merkmale stellen die *confusion*-Eigenschaft der *Feistel*-Struktur sicher. Betrachten wir kurz folgendes vereinfachendes Beispiel: Gegeben seien zwei Eingaben X und X', die sich in genau einem Bit unterscheiden. In wie vielen Bits unterscheiden sich $\textsc{Enc}(X)$ und $\textsc{Enc}(X')$? In der ersten Runde unterscheidet sich die Eingaben $X_1 = X$ und $X'_1 = X'$ in genau einem Bit, die Ausgaben $X_2 = F_1(X_1)$ und $X'_2 = F_1(X'_1)$ unterscheiden sich also mindestens in 2 Bits. In der zweiten Runde unterscheiden sich die beiden Eingaben X_2, X'_2 in mindestens 2 Bit, die Ausgaben $X_3 = F_2(X_2)$ und $X'_3 = F_2(X'_2)$ sind in mindestens 4 Bits unterschiedlich. Führen wir das für weitere Runden fort, stellen wir fest, dass sich die Anzahl der Bits, die von der ursprünglichen 1-Bit Änderung betroffen sind, exponentiell erhöht (nach n Runden mindestens 2^n viele Bits). Wichtig ist, dass sich die Ausgaben $F_n(X_n)$ und $F_n(X'_n)$ dabei nicht in allen Bits unterscheiden müssen. Wäre dies immer der Fall, dann könnte ein Angreifer unsere Blockchiffre von einer echten Zufallsfunktion unterscheiden[5]. Wir erhalten also lediglich eine untere Schranke für die Anzahl der durchzuführenden Runden. Bei einer Eingabelänge von n Bit sind mindestens $\lceil \log n \rceil$ viele Runden nötig, damit sich eine Änderung der Eingabe auf alle Bits der Ausgabe auswirkt. Führen wir weniger Runden aus, enthält die neue Ausgabe unveränderte Bits und die Blockchiffre kann von einer echten Zufalls-Funktion unterschieden werden.

Das besondere Merkmal einer *Feistel*-Struktur ist, dass obwohl ihre Komponenten (Rundenfunktionen, *S-Boxen*) nicht invertierbar sein müssen, die *Feistel*-Struktur selber es aber sehr wohl ist.

2.2.3.1 DES - Data Encryption Standard

Im Jahr 1973 gab das *National Bureau of Standards* (NBS) der USA, das heutige *National Institute of Standards and Technology* (NIST), eine öffentliche Anfrage nach einem Verschlüsselungsalgorithmus ab, der sensitive Regierungsinformationen sicher verschlüsseln kann. 1974 entwarf IBM einen Kandidaten, der auf dem *Lucifer*-Algorithmus basiert und ein *Feistel*-Netzwerk verwendet. Das NBS kontaktierte daraufhin die *National Security Agency* (NSA), um die Sicherheit dieses Algorithmus zu überprüfen. Nachdem die NSA einige Änderungen an diesem durchgeführt hatte, wurde der überarbeitete Algorithmus 1977 als *Data Encryption Standard* (DES) standardisiert und für die öffentliche Verwendung freigegeben. Der DES ist ein symmetrischer Verschlüsselungsalgorithmus, der ein wie oben beschriebenes Feistel-Netzwerk verwendet, einen 64-Bit langen Schlüssel benutzt und Daten in je 64-Bit Blöcken verschlüsselt. Die öffentliche Standardisierung des DES durch eine US-Regierungsbehörde trug maßgeblich zur schnellen weltweiten Verbreitung des Algorithmus bei. Gleichzeitig führte die Beteiligung der NSA am Entwurf des DES dazu, dass die Sicherheit des DES kontrovers diskutiert wurde. Die durchgeführten Änderungen der NSA umfassten die Verkürzung des Schlüssels von ursprünglich 128 Bit auf 56 zufällig gewählte Bit, sowie eine unkommentierte Veränderung der verwendeten S-Boxen. In Anbetracht der zentralen Bedeutung der S-Boxen für die Sicherheit eines *Feistel*-Netzwerkes

[5] Für eine echte Zufallsfunktion wird erwartet, dass sich bei einer 1-Bit Änderung der Eingabe nur die Hälfte der Ausgabe-Bits verändert.

wurde somit befürchtet, dass die NSA eine Hintertür in den DES eingebaut hatte. Daraufhin wurden 1994 die Design-Kriterien für die verwendeten S-Boxen von IBM veröffentlicht. Die Veröffentlichung ergab, dass die S-Boxen besonders resistent gegenüber der erst kurz vorher (1990) öffentlich-bekannt gewordenen *differentiellen Kryptoanalyse* sind.[6]

Betrachten wir nun etwas genauer den Aufbau des DES, wie er in Abbildung 2.11 dargestellt wird. Der verwendete Schlüssel besteht aus 64 Bit, von denen 56 Bit den eigentlichen Schlüssel darstellen und die restlichen 8 Bit lediglich Paritätsbits sind, die die acht 7-Bit-Zeichen eines Nachrichtenblockes auf ungerade Parität ergänzen. Somit umfasst der Schlüsselraum insgesamt $2^{56} \approx 7,2 \cdot 10^{16}$ mögliche Schlüsselkandidaten.

Der Algorithmus läuft folgendermaßen ab: Zuerst wird jeder neue Nachrichtenblock einer Eingangspermutation IP unterzogen und durchläuft anschließend 16 Schlüssel-abhängige Runden, wobei die verwendete Rundenfunktion immer die selbe ist. Jede der Runden benutzt unterschiedliche 48 Bit als Schlüssel; diese werden als Teilschlüssel aus den 56 Bit ausgewählt. Auf das Ergebnis der letzten Runde wird zum Schluss die zu IP inverse Permutation IP^{-1} angewandt. Ver- und Entschlüsselung sind aufgrund einer Konstruktionseigenschaft des Verschlüsselungsalgorithmus (s.u.) bis auf die Reihenfolge der pro Runde verwendeten Teilschlüssel identisch.

- Chiffrierung: $K_1, K_2, \ldots, K_{15}, K_{16}$
- Dechiffrierung: $K_{16}, K_{15}, \ldots, K_2, K_1$

[6] Untersuchungen haben ergeben, dass eine zufällige Wahl der S-Boxen zu einer deutlich höheren Anfälligkeit gegenüber der *differentiellen Kryptoanalyse* geführt hätte. Dies impliziert, dass IBM und die NSA bereits Jahre vor der Öffentlichkeit über diese Angriffsmethode Bescheid wussten.

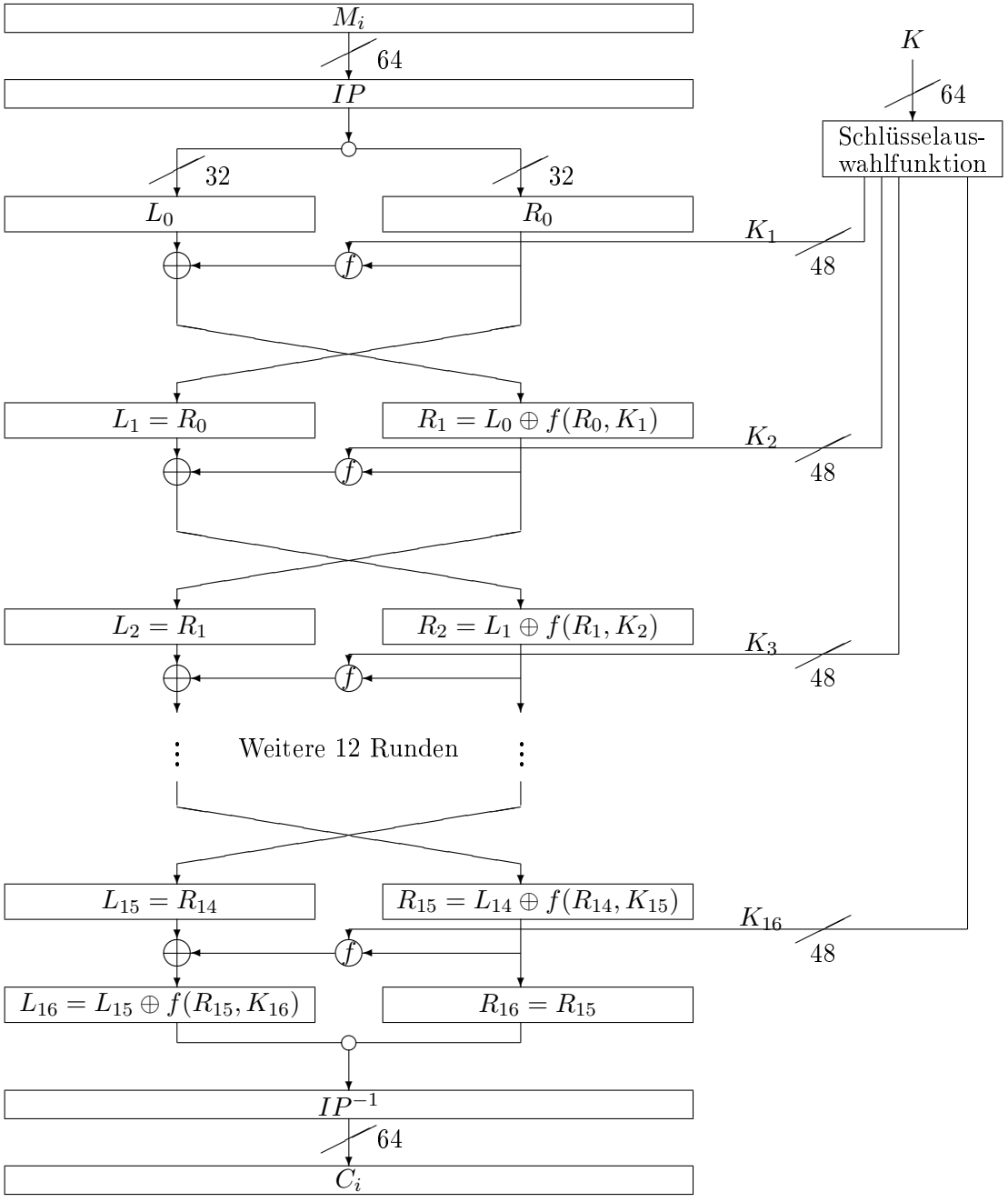


Abbildung 2.11: Struktur des DES

Der genaue Ablauf einer Verschlüsselung mit dem DES läuft wie folgt ab:

Initialpermuation Der 64-Bit Eingabeblock wird zunächst einer Initialpermutation IP unterworfen. Dabei werden die 64 Bit lediglich umgeordnet. Dieser Vorgang bietet keine nennenswerte kryptographische Sicherheit und dient lediglich einer effizienteren Nutzung der Hardware.

Aufteilung Der Eingabeblock wird nun in 2 Hälften geteilt: Eine linke und eine rechte Hälfte die jeweils 32 Bit enthalten.

Runden Nun wird die rechte Hälfte mit einem Teilschlüssel durch die Rundenfunktion f verschlüsselt und durch XOR mit der linke Hälfte verbunden. Hieraus ergibt sich die neue rechte Hälfte. Die Iteration der linke Hälfte ist schlicht die rechte Hälfte der vorherigen Runde. Die Rundenfunktion f arbeitet wie in Abbildung 2.12 dargestellt: Der 32-Bit-Datenblock wird zunächst auf 48 Bit expandiert, indem einige Bits verdoppelt werden. Danach wird der entstandene Block durch bitweises XOR mit dem Teilschlüssel verbunden. Dieser 48-Bit-Block wird nun auf verschiedene S-Boxen aufgeteilt, die jeweils aus einer 6-Bit-Eingabe eine 4-Bit-Ausgabe produzieren. In Tabelle 2.2 ist auszugsweise eine verwendete *S-Box* dargestellt. Konkateniert man nun die Ergebnisse der S-Boxen erhält man einen 32-Bit-Block, der das Ergebnis der Rundenfunktion darstellt. Dieser Vorgang wird 16 mal wiederholt, wobei in der letzten Runde das Ergebnis der Berechnungen nun die linke Hälfte einnimmt und die rechte Hälfte die rechte Hälfte der zweitletzten Runde übernimmt.

Zusammenführen Nun werden die beiden Hälften durch einfache Konkatenation wieder zu einem 64-Bit-Block zusammengeführt.

Ausgabepermutation Schließlich werden die Bits erneut permutiert, um die ursprüngliche Ordnung wieder herzustellen.

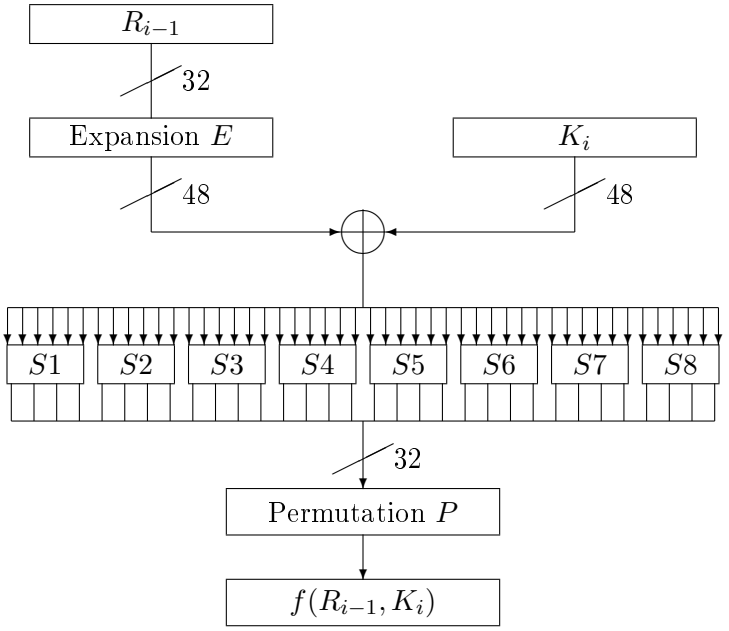


Abbildung 2.12: Die Rundenfunktion f des DES

Wie schon erwähnt, kann der DES-Algorithmus sowohl zum Chiffrieren als auch zum Dechiffrieren verwendet werden. Zum Dechiffrieren wird der chiffrierte Textblock genau derselben Prozedur wie beim Chiffrieren unterzogen, wobei die K_i in umgekehrter Reihenfolge

Input-Bits 4-5 \ Input-Bits 0-3	···	**0100**	**0101**	**0110**	**0111**	**1000**	···
00	···	0111	1010	1011	0110	1000	···
01	···	0100	0111	1101	0001	0101	···
10	···	1010	1101	0111	1000	1111	···
11	···	0001	1110	0010	1101	0110	···

Tabelle 2.2: Ein Auszug der 5. *S-Box* des DES

angewendet werden. Der erste Dechiffrierschritt ist dann IP, der den letzten Chiffrierschritt IP^{-1} aufhebt. Danach dient $R_{16}L_{16}$ als permutierter Eingabeblock, und die einzelnen R_i bzw. L_i erhält man durch:

$$\begin{aligned} R_{i-1} &= L_i \\ L_{i-1} &= R_i \oplus f(L_i, K_i) \end{aligned}$$

Wendet man auf L_0R_0 dann noch IP^{-1} an, so hebt man den Chiffrierschritt IP auf; der Klartextblock ist somit zurückgewonnen.

Der DES ist strukturell nahezu ungebrochen: Es gibt Angriffe durch *lineare Kryptoanalyse*, die besser sind als die vollständige Suche des Schlüsselraums, allerdings sind diese nicht praktikabel. Problematisch ist allerdings der, für heutige Verhältnisse, mit 56 Bit kleine Schlüsselraum, der Brute-Force Attacken in akzeptabler Zeit zulässt. Schon in den 90er-Jahren gelang es, Maschinen zu konstruieren, die eine Brute-Force Attacke erfolgreich innerhalb eines Tages durchführten[7]. Konsequenterweise zog die NIST den DES Standard daraufhin 2005 zurück und empfiehlt nur noch die Verwendung von 3-DES für die Verschlüsselung von sensitiven Informationen[8].

2.2.3.2 2DES

Die naive Lösung des Schlüsselproblems beim DES ist der 2DES. Hierbei wird der Klartextblock zwei mal mit verschiedenen Schlüsseln per DES verschlüsselt, wodurch man sich die doppelte Schlüsselbitanzahl erhofft.

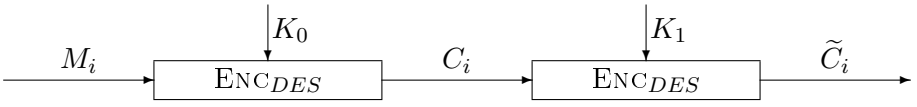


Abbildung 2.13: Prinzip des 2DES

Leider ist der 2DES nicht so effektiv wie erwartet, da es sogenannte Meet-in-the-Middle-Angriffe gibt. Unter der Voraussetzung, dass man ein Klartext-Chiffrat-Paar besitzt, ist die Vorgehensweise wie folgt:

1. Erstelle eine Liste aller möglichen im ersten Schritt erzeugbaren Chiffrate $C_{K_0} = \text{ENC}_{DES_{K0}}$, d.h. verwende alle möglichen K_0.

[7] In den letzten Jahren entwickelte Maschinen haben nicht nur die erforderliche Zeit für Brute-Force Attacke weiter reduziert, sondern auch die Produktionskosten gesenkt. So wurde 2006 von den Universitäten Bonn und Kiel der Computer *COPACOBANA* gebaut, der insgesamt nur noch knapp 9000 € in der Produktion kostete.

[8] Diese Empfehlung gilt aktuell nur bis zum Jahr 2030 und soll den Übergang zum *AES* erleichtern, der der eigentliche Nachfolger des DES ist.

2. Sortiere diese Liste lexikographisch um binäre Suche zu ermöglichen.

3. Berechne alle möglichen Chiffrate $C_{K_1} = \text{DEC}_{DES_{K1}}$

4. Falls nun ein Paar $C_{K_0} = C_{K_1}$ gibt, gebe jeweils den Schlüssel K_0 und K_1 aus.

Ein solcher Angriff besitzt einen Platzbedarf von $64 \cdot 2^{56} + \epsilon$ und hat eine Laufzeit in $O(56 \cdot 2^{56})$. 2DES bietet also einen sehr geringen Vorteil gegenüber einfachem DES.

2.2.3.3 3DES

Die direkte Erweiterung des 2DES ist der 3DES. Wie der Name bereits verrät werden hier 3 DES-Verschlüsselungen verwendet. Allerdings wird die mittlere DES-Verschlüsselung umgekehrt verwendet, d.h. bei der Verschlüsselung ist diese im Entschlüsselungsmodus und umgekehrt.

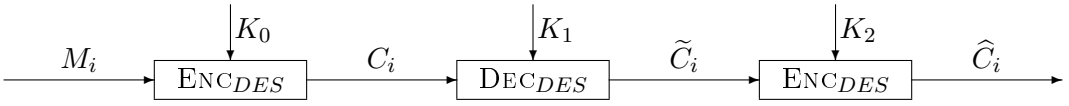


Abbildung 2.14: Prinzip des 3DES

Ein Meet-in-the-Middle-Angriff ist hier zwar noch möglich, aber bereits weit weniger praktikabel: Die Laufzeit befindet sich in $O(2^{112})$.

2.2.3.4 AES - Advanced Encryption Standard

Im Jahr 2000 stellte das *National Institute of Standards and Technology* (NIST) den *Advanced Encryption Standard* (AES) als Nachfolger des DES vor, nachdem drei Jahre zuvor ein offener Wettbewerb, um die alte Blockchiffre zu ersetzen, ausgeschrieben worden war. Den eigentliche Sieger des Wettbewerbs, der *Rijndael-Algorithmus*, hatte man dabei nur in einigen wenigen unwesentlichen Punkten angepasst.

Im Gegensatz zu DES verschlüsselt AES jeweils 128 Bit große Datenblöcke, wohingegen die Schlüssellänge aus 128 Bit, 192 Bit und 256 Bit gewählt werden kann. Dementsprechend bezeichnet AES-256 genau die Variante mit der größten Schlüssellänge. Der 128 Bit große Datenblock wird zu Beginn sequenziell in eine zweidimensionale 4x4-Tabelle, den sogenannten *state*, geschrieben. Jede Zelle des *states* repräsentiert dabei genau ein Byte des Klartextblocks. Ähnlich zu DES, läuft die Verschlüsselung bei AES rundenbasiert ab, wobei eine Runde jeweils aus den vier folgenden Schritten besteht:

1. **AddRoundKey** Der aus dem Hauptschlüssel abgeleitete, ebenfalls 128 Bit lange Rundenschlüssel wird byteweise mit dem *state* XOR-verknüpft

2. **SubBytes** Benutze die *S-Box*, um jedes Byte der zweidimensionalen Tabelle durch ein anderes Byte zu ersetzen

3. **ShiftRows** Rotiere die zweite Zeile zyklisch um ein Byte, die dritte Zeile zyklisch um zwei Byte und die vierte Zeile zyklisch um drei Byte nach links

4. **MixColumns** Wende auf jede Spalte eine invertierbare lineare Transformation[9] an

[9] Vereinfacht kann man sich unter der invertierbaren linearen Transformation eine Matrixmultiplikation auf einer speziellen Struktur vorstellen. Wichtig ist insbesondere die Invertierbarkeit. Genauere Details möchten wir an dieser Stelle jedoch nicht besprechen. Bei weiterführendem Interesse bietet der Standard der NIST einen formalen, aber aufschlussreichen Einblick.

Für die Verschlüsselung greift AES also auf Operationen zurück, die man in ihrer Ganzheit treffend als Substitutions- und Rotationsnetzwerk beschreiben kann. Wir sehen, dass S-Boxen nicht nur in der Feistel-Struktur Verwendung finden und sehr wohl auch bijektiv seien können. Die Anzahl der Verschlüsselungsrunden ist von der gewählten Schlüssellänge abhängig: Bei 128 Bit werden 10, bei 192 Bit 12 und bei 256 Bit 14 Verschlüsselungsrunden durchlaufen. Um zu verhindern, dass ein Angreifer die letzten drei Schritte der Schlussrunde zurückrechnen kann, wird anstelle von MixColumns ein zusätzliches AddRoundKey ausgeführt.

Da die XOR-Operation, das byteweise Ersetzen mittels einer S-Box und das zyklische Rotieren jeweils invertierbare Funktionen sind und wir die Invertierbarkeit bei MixColumns explizit fordern, funktioniert das Entschlüsseln eines Chiffrats problemlos und effizient.

Nach heutigem Kenntnisstand garantiert AES ein hohes Maß an Sicherheit. Zwar gibt es theoretische Kryptoanalysen, die aber aufgrund ihrer hohen Laufzeit (für AES-256 werden 2^{254} Schritte benötigt) in der Praxis keine Relevanz haben. Nicht zuletzt deswegen ist AES ab der Schlüssellänge von 192 Bit in den USA noch immer zur Verschlüsselung staatlicher Dokumente der höchsten Geheimhaltungsstufe zugelassen.

2.2.4 Angriffe auf Blockchiffren

2.2.4.1 Lineare Kryptoanalyse

Die *lineare Kryptoanalyse* stellt einen Angriff auf Blockchiffren dar, der meist besser als die vollständige Suche des Schlüsselraums ist. Das Ziel dieser Angriffsmethode ist eine lineare Abhängigkeit für die Verschlüsselung zu bestimmen:

$$P\left[i_1, i_2, \ldots, i_a\right] \oplus C\left[j_1, j_2, \ldots, j_b\right] = K\left[k_1, k_2, \ldots, k_c\right], \tag{2.5}$$

wobei $i_1, i_2, \ldots, i_a, j_1, j_2, \ldots, j_b, k_1, k_2, \ldots, k_c$ feste Bitpositionen bezeichnen und die Gleichung mit einer Wahrscheinlichkeit von $p \neq \frac{1}{2}$ für einen zufälligen Klartext M und den zugehörigen Chiffretext C gilt. Die Größe $\left|p - \frac{1}{2}\right|$ gibt die Effektivität der Gleichung 2.5 an. Wenn eine effektive lineare Approximation bekannt ist, dann kann man mit der naiven Maximum-Likelihood-Methode ein Schlüsselbit $K\left[K_1, K_2, \ldots, K_c\right]$ erraten.

Bei Verschlüsselungssystemen, welche die *Feistel*-Struktur verwenden, sehen entsprechende Angriffe wie folgt aus:

1. Finde lineare Abhängigkeiten zwischen Ein- und Ausgabe
2. Erweitere Abhängigkeiten auf die ersten n - 1 *Feistel*-Runden
3. Vollständige Suche über letzten Rundenschlüssel K_n
4. Gefundene Kandidaten durch die bekannten linearen Abhängigkeiten überprüfen
5. Wenn K_n, fahren mit K_{n-1} fort usw.

Für den gewöhnlichen DES mit 16 Runden ist dieses Vorgehen allerdings schon wegen der immensen Anzahl an benötigten Klartext-Chiffrat-Paaren nicht praktikabel (es werden bis zu 2^{43} solcher Paare benötigt). Für andere Blockchiffren hingegen, die ebenfalls eine *Feistel*-Struktur verwenden, beispielsweise FEAL, ist ein effizienter Angriff mit dieser Methode möglich.

Beispiel: 3-Runden DES Bei einem wie in Abbildung 2.15 dargestellten DES beginnt man zunächst damit die bekannten, aber nicht linearen S-Boxen linear zu approximieren, d.h. einen linearen Zusammenhang zwischen den Eingangs- und Ausgangsbits einer S-Box zu finden. Dies wurde bereits in [8] getan und übersteigt den Umfang dieser Vorlesung, jedoch erhält man hierdurch wichtige Zusammenhänge wie:

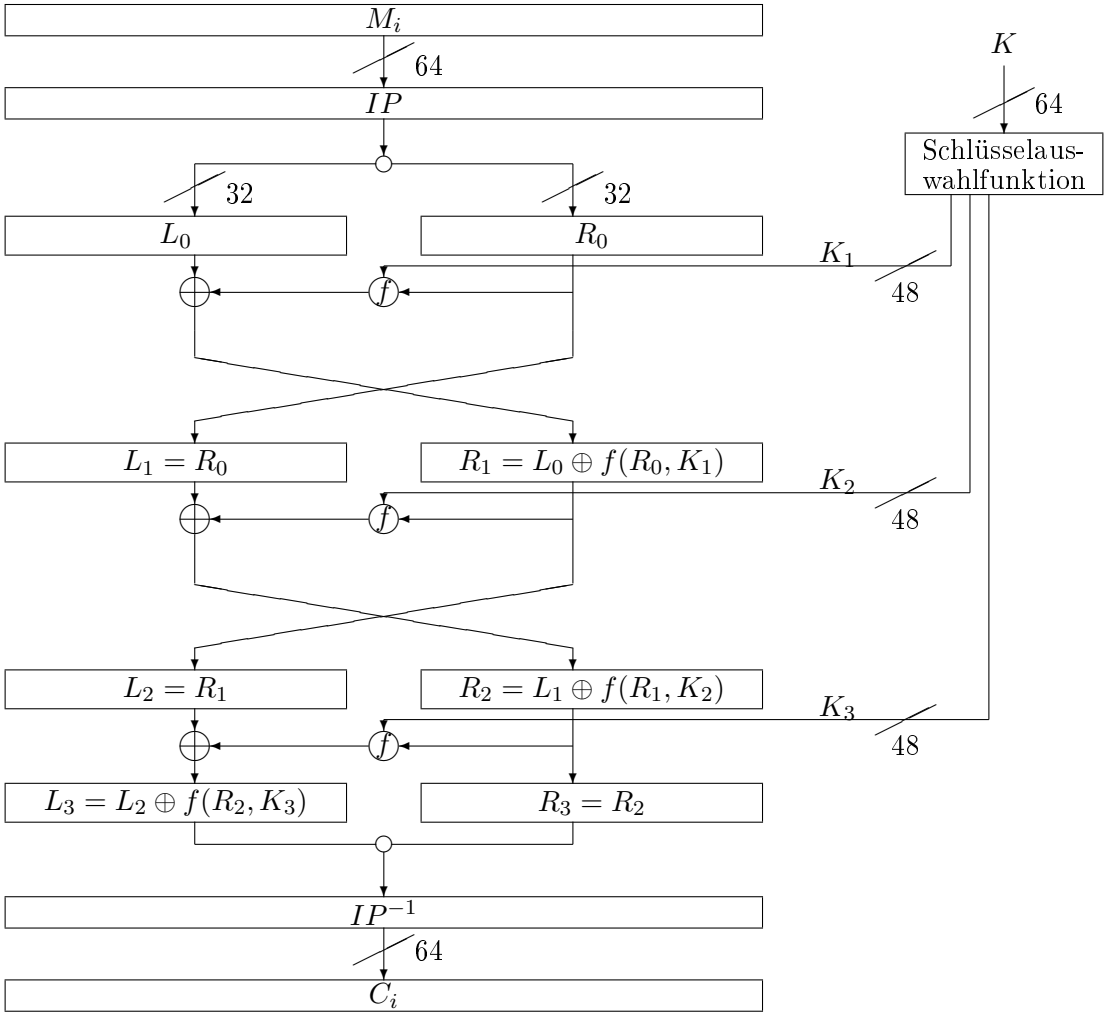


Abbildung 2.15: Darstellung eines DES mit 3 Runden

Bei S-Box S_5 gilt, dass das Eingabebit vier (von rechts mit null beginnend gezählt) in 12 der 64 Fälle (Eingaben) mit dem XOR der vier Ausgabebits überein stimmt.

Falls es Gleichungen gibt, welche für ungleich 32 der 64 Fälle gelten, so gibt es eine Korrelation der Eingabe- und Ausgabebits der S-Box. Das obige Beispiel ist die größte bekannte Abweichung, da sie nur in 12 der 64 Fällen gilt. Berücksichtigt man nun die Permutation P und die Expansion E in der Rundenfunktion f, ergibt sich daraus die Gleichung:

$$R_i\,[15] \oplus f(R_i, K_i)\,[7, 18, 24, 29] = K_i\,[22] \tag{2.6}$$

Die Gleichung 2.6 gilt also in 12 der 64 Fälle und hat damit eine Wahrscheinlichkeit von $\approx 0,19$. Wenn man diese Gleichung 2.6 auf die erste Runde anwendet, dann sieht man, dass

$$R_1\,[7, 18, 24, 29] \oplus L_0\,[7, 18, 24, 29] \oplus R_0\,[15] = K_1\,[22] \tag{2.7}$$

mit der Wahrscheinlichkeit 12/64 gilt. Dasselbe gilt für die letzte Runde, also

$$R_1\,[7, 18, 24, 29] \oplus L_3\,[7, 18, 24, 29] \oplus R_3\,[15] = K_3\,[22]\,. \tag{2.8}$$

Aus den beiden vorhergehenden Gleichungen erhält man folgende lineare Approximation des 3-Runden-DES, indem man die beiden Gleichungen addiert und dabei die gemein-

samen Terme weglässt:

$$L_0[7,18,24,29] \oplus L_3[7,18,24,29] \oplus R_0[15] \oplus R_3[15] = K_1[22] \oplus K_3[22]. \tag{2.9}$$

Die Wahrscheinlichkeit, dass die obige Gleichung für einen zufälligen Klartext M und den zugehörigen Chiffretext C gilt, beträgt $(12/64)^2 + (1 - 12/64)^2 \approx 0,70$. Da die Gleichung 2.6 die beste Approximation der F-Funktion ist, ist damit die Gleichung 2.9 die beste Approximation für den 3-Runden-DES. Man kann jetzt statistisch (Maximum-Likelyhood-Methode) Gleichung 2.9 lösen, um $K_1[22] \oplus K_3[22]$ zu erhalten.

Das bedeutet, dass man nun Schlüsselinformationen effizienter als mit vollständige Suche erhalten hat. Man kann mit diesen Informationen nun eine effizientere vollständige Suche für K_1 und K_3 durchführen, um Kandidaten zu finden und mit diesen K_2 finden.

2.2.4.2 Differentielle Kryptoanalyse

Anders als bei der *linearen Kryptoanalyse* werden hier nicht direkte Zusammenhänge zwischen einzelnen Klartextblöcken und deren Chiffrate gesucht, sondern indirekt durch Vergleiche zweier Blöcke miteinander: Es gilt die Auswirkungen der Differenz zweier Klartextblöcke $M \oplus M'$ auf die Differenz ihrer Chiffrate $\text{DEC}_K(M) \oplus \text{DEC}_K(M')$ zu finden.

Für Feistel-Strukturen ist die Vorgehensweise ähnlich der *linearen Kryptoanalyse*:

1. Finde die wahrscheinlichsten Zusammenhänge zwischen Eingabe- und Ausgabedifferenzen der vorletzten Runde.
2. Führe vollständige Suche für K_n durch.
3. Überprüfe Kandidaten durch Testen der Konsistenz bezüglich den Ein- und Ausgabedifferenzen.

Wie bereits bei der *linearen Kryptoanalyse* ist DES selbst sehr resistent gegenüber der *differentiellen Kryptoanalyse*, während andere auf *Feistel*-Struktur basierende Systeme anfällig sind. Dies ist vor allem darauf zurückzuführen, dass diese Resistenz ein Entwicklungsziel des DES war, obwohl dieser Angriff erst ein Jahrzehnt später veröffentlicht wurde.

Kapitel 3

Sicherheitsbegriff

3.1 Semantische Sicherheit

Nachdem wir uns bereits mit Verschlüsselungssystemen auseinandergesetzt haben, stellt sich natürlich die Frage was wir genau erreichen wollen. Primäres Ziel ist es bisher, dass ein Angreifer durch das Chiffrat keine Informationen zum Klartext erhält. Dies entspricht dem Begriff der *semantischen Sicherheit*, welcher 1982 von Goldwasser und Micali definiert wurde. Diese besagt konkret:

> *Alles was mit mit C (effizient) über M berechnet werden kann, kann auch ohne das Chiffrat berechnet werden.*

Dabei ist zu beachten, dass diese Form von Sicherheit lediglich passive Angriffe abdeckt, d.h. ausschließliche Angriffe ohne Veränderung des Chiffrats.

Definition 3.1 (Semantische Sicherheit)**.** Ein symmetrischer Verschlüsselungsalgorithmus ist semantisch sicher, wenn es für jede M-Verteilung von Nachrichten gleicher Länge, jede Funktion f und jeden *effizienten* Algorithmus $\mathcal{A}$ einen *effizienten* Algorithmus $\mathcal{B}$ gibt, so dass

$$\Pr\left[\mathcal{A}^{\textsc{Enc}(K,\cdot)}\left(\textsc{Enc}\left(K,M\right)\right)=f(M)\right]-\Pr\left[\mathcal{B}(\epsilon)=f(M)\right]$$

klein ist.

Allerdings impliziert die Existenz von mehrfach benutzbaren, semantisch sicheren Verfahren damit $P \neq NP$. Das bedeutet, falls $P = NP$ gelten sollte, kann es kein solches Verfahren geben. Außerdem ist diese Definition technisch schwer zu handhaben, da sie viele Quantoren enthält. Hierfür wurden handlichere aber äquivalente Begriffe eingeführt wie beispielsweise *IND-CPA*.

3.2 Der IND-CPA-Sicherheitsbegriff

IND-CPA steht für *indistinguishability under chosen-plaintext attacks*. Bei einem Verfahren, welches diese Sicherheit besitzt, kann ein Angreifer $\mathcal{A}$ die Chiffrate von selbstgewählten Klartexten nicht unterscheiden. Es gelten folgende Bedingungen:

- Der Angreifer $\mathcal{A}$ besitzt Zugriff auf ein $\textsc{Enc}(K,\cdot)$-Orakel[1]
- Der Angreifer $\mathcal{A}$ wählt zwei Nachrichten M_1 und M_2 gleicher Länge
- Der Angreifer erhält $C_* := \textsc{Enc}(K, M_b)$ für ein zufällig gleichverteiltes $b \in \{1, 2\}$.

[1] Ein Orakel kann man sich als *black box* vorstellen, bei der der Fragende zwar das Ergebnis, jedoch nichts über dessen Berechnung in Erfahrung bringt.

- Der Angreifer $\mathcal{A}$ gewinnt, wenn er b korrekt errät.

Ein Verfahren ist nun IND-CPA-sicher, wenn der Vorteil des Angreifers gegenüber dem Raten einer Lösung, also $\left(\Pr\left[\mathcal{A} \text{ gewinnt}\right] - \frac{1}{2}\right)$, für alle Angreifer $\mathcal{A}$ vernachlässigbar klein ist. Eine solche Abfolge von Bedingungen nennen wir auch Experiment.

Theorem 3.2. *Ein Verfahren ist genau dann semantisch sicher, wenn es IND-CPA-sicher ist.*

3.2.1 Beispiele

3.2.1.1 ECB-Modus

Behauptung: Keine Blockchiffre ist im ECB-Modus IND-CPA-sicher.

Beweis: Betrachte folgenden Angreifer $\mathcal{A}$

Schritt 1: Der Angreifer $\mathcal{A}$ wählt zwei Klartextblöcke $M_1 \neq M_2$ beliebig.

Schritt 2: Der Angreifer $\mathcal{A}$ erhält $C_* := \textsc{Enc}(K, M_b)$.

Schritt 3: Der Angreifer $\mathcal{A}$ erfragt $C_1 = \textsc{Enc}(K, M_1)$ durch sein Orakel.

Schritt 4: Der Angreifer $\mathcal{A}$ gibt 1 aus, genau dann wenn $C_1 = C_*$, sonst gibt er 2 aus.

$\Pr[\mathcal{A} \text{ gewinnt}] = 1$, also ist das Schema nicht IND-CPA-sicher.

Bei diesem Beispiel nutzt der Angreifer die Schwäche des ECB-Modus, dass gleiche Klartextblöcke immer zu gleichen Chiffrat-Blöcken werden, aus.

3.2.1.2 CBC-Modus

Wie wir bereits im Abschnitt 2.2.2.2 gelernt haben gilt bei der Verwendung des CBC-Modus: $C_i = \textsc{Enc}(K, M_i \oplus C_{i-1})$ mit $C_0 := IV$.

Im Folgenden nehmen wir an, dass der Initialisierungsvektor IV bei jeder Verschlüsselung erneut gleichverteilt gewählt und dem Chiffrat beigefügt wird. Dies wird getan, um zu verhindern, dass sich der Modus bei einem einzelnen Datenblock gleich wie der ECB-Modus verhält: Bei einem festen IV ergeben gleiche einzelne Datenblöcke gleiche Chiffratblöcke und wir haben die selbe Problematik wie bereits beim ECB-Modus. Im CBC-Modus existiert diese Problematik generell nur für den ersten Datenblock, selbst bei festem und öffentlichen IV, da darauffolgende Datenblöcke mit den vorausgehenden Chiffraten verkettet wird und somit kein direkter Zusammenhang mehr zwischen Daten- und Chiffratblock besteht. Allerdings werden bei der IND-CPA-Sicherheit zwei einzelne Blöcke verwendet, weshalb wir hier auf die Variante mit dem zufälligen IV ausweichen.

Das zufällige Wählen eines IV löst zwar ein Problem, erzeugt jedoch ein anderes: Um eine korrekte Entschlüsselung zu gewährleisten muss der gewählte IV mitgesendet werden. Das resultierende Problem ist nun, dass ein Angreifer eben diesen IV verändern kann und somit „direkte Änderungen“, d.h. beliebige Manipulation am ersten Block einer bekannten Nachrichtoder eine Unlesbarkeit aller Blöcke erzeugen kann. Diese Art von Angriffen, bei denen die Nachricht nicht nur abgehört sondern auch verändert wird, werden auch *aktive Angriffe* genannt und sind von der IND-CPA-Sicherheit, welche sich mit *passiven Angriffen* beschäftigt, nicht abgedeckt.

Behauptung: Eine Blockchiffre ist im CBC-Modus genau dann IND-CPA-sicher, wenn die Verschlüsselungsfunktion $\textsc{Enc}(K, \cdot) : \{0,1\}^l \to \{0,1\}^l$ nicht von einer Zufallsfunktion $R : \{0,1\}^l \to \{0,1\}^l$ unterscheidbar ist.

Beweisidee:

IND-CPA-sicher $\Rightarrow$ Ununterscheidbarkeit
$\Leftrightarrow$ IND-CPA-unsicher $\Leftarrow$ Unterscheidbarkeit
Wenn $\textsc{Enc}(K, \cdot)$ als Angreifer von einer Zufallsfolge unterscheidbar ist, bedeutet das, dass zwischen mindestens zwei Verschlüsselungsergebnissen eine Zusammenhang erkennbar ist. Dies bedeutet es gibt mindestens einen Fall, bei dem der Angreifer zusätzliche Informationen für das Zuordnen des Chiffrats besitzt. Daher gilt für zufällig gewählte Nachrichten im IND-CPA-Szenarium: $\Pr[\mathcal{A} \text{ gewinnt}] > \frac{1}{2} \Rightarrow$ IND-CPA-unsicher.

IND-CPA-sicher $\Leftarrow$ Ununterscheidbarkeit
Wenn die Verschlüsselungsfunktion aus Sicht des Angreifers eine Zufallsfunktion sein könnte, gibt es keine bekannten Zusammenhänge der Verschlüsselungen, d.h. Die Wahrscheinlichkeit dass der Angreifer ein Chiffrat korrekt zuordnet ist genau $\frac{1}{2}$.

3.3 Der IND-CCA-Sicherheitsbegriff

Der CPA-Angreifer ist mit Zugriff auf ein Verschlüsselungsorakel ausgestattet. Er kann sich jedmöglichen Klartext verschlüsseln lassen und versuchen, Muster in den Ausgaben des Orakels zu erkennen. Eingeschränkt ist er dennoch, da ihm die Möglichkeit fehlt, zu beliebigen Ciphertexten den Klartext zu berechnen. Ein stärkerer Sicherheitsbgeriff ist daher IND-CCA. Ähnlich wie IND-CPA, steht IND-CCA abkürzend für *indistinguishability under chosen-ciperthext attacks*. Dabei suggeriert das Akronym CCA bereits einen mächtigeren Angreifer. Das in 3.2 vorgestellte Experiment können wir problemlos auf einen IND-CCA-Angreifer anpassen. Modifiziert ergibt sich:

- Der Angreifer $\mathcal{A}$ besitzt Zugriff auf ein $\textsc{Enc}(K, \cdot)$-Orakel und ein $\textsc{Dec}(K, \cdot)$-Orakel
- Der Angreifer $\mathcal{A}$ wählt zwei Nachrichten M_1 und M_2 gleicher Länge
- Der Angreifer erhält $C_* := \textsc{Enc}(K, M_b)$ für ein zufällig gleichverteiltes $b \in \{1, 2\}$.
- Der Angreifer $\mathcal{A}$ gewinnt, wenn er b korrekt errät.

Natürlich darf der Angreifer den Klartext zum Chiffrat C_* nicht bei dem Entschlüsselungsorakel erfragen. Ein Angriff wäre ansonsten trivial und es gäbe kein Verfahren, dass dem IND-CCA-Sicherheitsbegriff genügt.

Analog heisst ein Verfahren nun IND-CCA-sicher, wenn der Vorteil des Angreifers gegenüber dem Raten einer Lösung, also $\left(\Pr\left[\mathcal{A} \text{ gewinnt}\right] - \frac{1}{2}\right)$, für alle Angreifer vernachlässigbar klein ist.

Etwas granularer kann bei IND-CCA zwischen einer adaptiven und einer nichtadaptiven Variante unterschieden werden. Erstere erlaubt dem Angreifer weitere Berechnungen, auch nachdem C_* bereits erhalten wurde, wohingegen die nichtadaptive Variante solche Berechnungen explizit verbietet. In der Theorie kann sich ein adaptiver Angreifer also auch von C_* abhängige Ciphertexte entschlüsseln lassen. Sprechen wir in diesem Skript von IND-CCA, so gehen wir von der adaptiven Variante aus.

Kapitel 4

Hashfunktionen

4.1 Grundlagen

Hashfunktionen sind Funktionen, die von einer großen, potentiell unbeschränkten Menge in eine kleinere Menge abbilden:

$$H\colon \{0,1\}^* \to \{0,1\}^k$$

Diese Funktionen werden dazu verwendet, größere Datenmengen effizient zu kennzeichnen (ihnen sozusagen einen Fingerabdruck zuzuordnen). Anwendungsgebiete sind z.B. das Verifizieren der Datenintegrität eines Downloads oder das Signieren von Daten durch einen Abgleich der entsprechenden Hashwerte.

4.2 Sicherheitseigenschaften

Um eine Hashfunktion im kryptographischen Sinne verwenden zu können, reicht eine Funktion, die von einer großen Menge in eine kleine Menge abbildet, nicht aus. Sie muss zusätzlich einige weitere Anforderungen erfüllen.

4.2.1 Kollisionsresistenz

Die wichtigste Eigenschaft einer Hashfunktion H ist die Kollisionsresistenz (*collision resistance*). Das bedeutet, es soll schwierig sein, zwei Urbilder X, X' zu finden, für die gilt:

$$X \neq X' \text{ und } H(X) = H(X')$$

Da wir von einer großen in eine kleine Menge abbilden, kann H nicht injektiv sein. Es ist uns also nicht möglich, Kollisionen komplett zu verhindern. Trotzdem können wir fordern, dass diese möglichst selten auftreten. Präziser formuliert verlangen wir, dass bei jeder kollisionsresistenten Hashfunktion ein *effizienter* Algorithmus eine Kollision nur mit *kleiner* Wahrscheinlichkeit findet.

Im Folgenden bezeichnen wir einen Algorithmus als *effizient*, der in Polynomialzeit zur Eingabe läuft, d.h. seine Laufzeit höchstens polynomiell mit der Eingabelänge wächst, und wenn nötig zufällige Entscheidungen treffen darf. Wir nennen einen solchen Algorithmus *probabilistic polynomial time*, kurz PPT.

Um auch *kleine* Kollisionswahrscheinlichkeiten genauer zu definieren, beginnen wir zunächst damit die Bildmenge der Hashfunktion zu parametrisieren:

$$H_k\colon \{0,1\}^* \to \{0,1\}^k$$

Wir nennen k den Sicherheitsparameter von H_k. Ein höheres k bedeutet eine größere Bildmenge und damit für H_k intuitiv auch eine geringere Kollisionswahrscheinlichkeit.

Desweiteren fordern wir, dass die Funktion $f\colon \mathbb{N} \to \mathbb{R}$, die allen Parametern eine Kollisionswahrscheinlichkeit zuordnet, in k *vernachlässigbar* ist. Das bedeutet, dass $|f|$ asymptotisch schneller verschwindet, als der Kehrwert jedes vorgegebenen Polynoms c:

$$\forall c \exists k_0 \forall k > k_0 : \ |f(k)| \leq k^{-c}$$

Beispiel 4.1. *$f = \frac{1}{2^k}$ ist vernachlässigbar, $f = \frac{1}{k^2}$ jedoch nicht.*

Definition 4.2 (Kollisionsresistenz)**.** Eine über k parametrisierte Funktion H ist kollisionsresistent, wenn jeder PPT-Algorithmus nur mit höchstens vernachlässigbarer Wahrscheinlichkeit eine Kollision findet.
Noch präziser formuliert ist der Vorteil für jeden PPT-Angreifer $\mathcal{A}$

$$Adv^{cr}_{H,\mathcal{A}}(k) := \Pr\left[(X, X') \leftarrow \mathcal{A}(1^k) : X \neq X' \wedge H_k(X) = H_k(X')\right]$$

vernachlässigbar.

4.2.2 Einwegeigenschaft

Die zweite kryptographisch wichtige Eigenschaft von Hashfunktionen ist die Einwegeigenschaft (*pre-image resistance*), die sicherstellt, dass eine Hashfunktion nur in eine Richtung berechenbar ist. Genauer gesagt fordern wir, dass es bei einem gegebenen Wert $H(X)$ *schwierig* ist, ein passendes X zu finden.

Angewendet wird diese Eigenschaft beispielsweise beim Speichern von Passwörtern auf einem Server. Der Server speichert nur $H(X)$ ab und vergleicht bei einem Anmeldungsversuch lediglich $H(X)$ mit dem ihm vom Client zugesendeten $H(X')$. Dadurch muss das Passwort nicht im Klartext auf dem Server liegen. Ebenfalls nützlich ist die Einwegeigenschaft bei der Integritätssicherung von Daten. Wenn die verwendete Hashfunktion die Einwegeigenschaft erfüllt, ist es schwierig, einen Datensatz so zu verändern, dass der Hashwert des Datensatzes gleich bleibt und die Veränderung sich nicht bemerkbar macht.

Es stellt sich nun die Frage, wie eine Hashfunktion beschaffen sein muss, damit sie die Einwegeigenschaft erfüllen kann. Ist z.B. die Urbildmenge zu klein, kann durch Raten einfach auf ein passendes X' geschlossen werden. Außerdem sollte es intuitiv keinen Kandidaten X' als Urbild für $H(X)$ geben, der wahrscheinlicher ist als andere Kandidaten. Um das zu erreichen, wird für die Elemente der Urbildmenge üblicherweise eine Gleichverteilung angestrebt.

Definition 4.3 (Einwegfunktion)**.** Eine über k parametrisierte Funktion H ist eine Einwegfunktion bezüglich der Urbildverteilung χ_k, wenn jeder PPT-Algorithmus nur mit höchstens vernachlässigbarer Wahrscheinlichkeit ein Urbild eines gegebenen, aus χ_k bezogenen Bildes findet. Genauer ist der Vorteil für jeden PPT-Angreifer $\mathcal{A}$

$$Adv^{ow}_{H,\mathcal{A}}(k) := \Pr\left[X' \leftarrow \mathcal{A}(1^k, H(X)) : H(X) = H(X')\right]$$

vernachlässigbar, wobei $X \leftarrow \chi_k$ gewählt wurde.

Dabei muss $\mathcal{A}$ nicht zwingend $X' = X$ zurückgeben.

Die Forderungen nach Kollisionsresistenz und Einwegeigenschaft, die wir bisher für eine kryptographische Hashfunktion aufgestellt haben, hängen bei näherer Betrachtung sehr eng miteinander zusammen. Das führt uns zu folgender Feststellung:

Theorem 4.4. *Jede kollisionsresistente Hashfunktion $H_k\colon \{0,1\}^* \to \{0,1\}^k$ ist eine Einwegfunktion bzgl. der Gleichverteilung auf $\{0,1\}^{2k}$.*

Beweisidee. *Bei $X \in \{0,1\}^{2k}$ hat fast jedes Urbild X viele „Nachbarn" X' mit $H(X) = H(X')$. Also gilt für die Wahrscheinlichkeit, dass ein Element $H(X)$ der Bildmenge nur ein einziges Urbild X besitzt:*

$$\Pr\left[|H^{-1}(H(X))| = 1\right] \leq \frac{2^k}{2^{2k}} = \frac{1}{2^k}$$

Beweis. *Zu jedem H-Invertierer $\mathcal{A}$ geben wir nun einen H-Kollisionsfinder $\mathcal{B}$ an mit*

$$Adv^{cr}_{H,\mathcal{B}}(k) \geq \frac{1}{2} \cdot Adv^{ow}_{H,\mathcal{A}}(k) - \frac{1}{2^{k+1}}$$

Nun wählt $\mathcal{B}$ ein $X \leftarrow \{0,1\}^{2k}$ gleichverteilt zufällig und gibt $H(X)$ als Eingabe an $\mathcal{A}$. $\mathcal{B}$ setzt nun $X' \leftarrow \mathcal{A}(1^k, H(X))$ und gibt (X, X') aus.
Dann gilt für $\mathcal{B}$s Erfolgswahrscheinlichkeit:

$$\begin{aligned}
&\Pr[\mathcal{B} \textit{ gewinnt}] \\
&= \Pr\left[H(X) = H(X') \wedge X \neq X'\right] \\
&= \Pr\left[\mathcal{A} \textit{ invertiert} \wedge X \neq X'\right] \\
&\geq \Pr\left[\mathcal{A} \textit{ invertiert} \wedge X \neq X' \wedge |H^{-1}(H(X))| > 1\right] \\
&= \underbrace{\Pr\left[X \neq X' | \mathcal{A} \textit{ invertiert} \wedge |H^{-1}(H(X))| > 1\right]}_{\geq \frac{1}{2}} \cdot \underbrace{\Pr\left[\mathcal{A} \textit{ invertiert} \wedge |H^{-1}(H(X))| > 1\right]}_{\geq \Pr[\mathcal{A} \textit{ invertiert}] - \frac{1}{2^k}} \\
&\geq \frac{1}{2} \cdot Adv^{ow}_{H,\mathcal{A}}(k) - \frac{1}{2^{k+1}}
\end{aligned}$$

□

4.2.3 Target Collision Resistance

Die *Target Collision Resistance* (auch *Second Pre-image Resistance* oder *Universal One-Way*) ist eine weitere Eigenschaft, die zur Bewertung von Hashfunktionen herangezogen wird. Genügt eine Hashfunktion H der Target Collision Resistance, ist es *schwierig*, für ein gegebenes Urbild X ein $X' \neq X$ zu finden, für das gilt: $H(X') = H(X)$.

Die Target Collision Resistance stellt einen Zwischenschritt zwischen der Kollisionsresistenz und der Einwegeigenschaft dar: Kollisionsresistenz impliziert die Target Collision Resistance. Target Collision Resistance impliziert wiederum die Einwegeigenschaft.

Beispiel 4.5. *Gegeben sei ein Zertifikat für einen gehashten Public Key. Für eine Hashfunktion, die keine Target Collision Resistance garantiert, ist es einfach, einen zweiten Public Key zu finden, der den gleichen Hashwert hat. Da das Zertifikat an den Hashwert des Schlüssels und nicht an den Schlüssel selbst gekoppelt ist, ist es auch für den zweiten Schlüssel gültig.*

4.3 Merkle-Damgård-Konstruktion

In der Praxis werden Hashfunktionen benötigt, die nicht nur die Eigenschaften aus den obigen Abschnitten berücksichtigen, sondern auch flexibel in ihrer Eingabelänge und konstant in ihrer Ausgabelänge sind. Typischerweise werden für diesen Zweck Merkle-Damgård-Konstruktionen eingesetzt.

4.3.1 Struktur von Merkle-Damgård

Die Eingabenachricht wird bei einer Merkle-Damgård-Konstruktion H_{MD} zunächst in Blöcke $X_0, \ldots, X_n$ mit fester Länge m aufgeteilt (z.B. 512 Bit). Auf diese Blöcke wird dann

nacheinander eine Kompressionsfunktion F angewendet, die die Blöcke mithilfe eines Eingabeparameters Z auf eine festgelegte Länge $k < m$ verkürzt.

Aus dem ersten Nachrichtenblock X_0 und dem Initialisierungsvektor IV der Länge k wird durch die Kompressionsfunktion ein neuer Wert Z_0 der Länge k erzeugt. Z_0 wird daraufhin gemeinsam mit dem zweiten Block X_1 zur Berechnung von Z_1 benutzt und so fort. Wenn alle Blöcke $X_0 \dots X_n$ der Nachricht abgearbeitet sind, ergibt sich aus Z_n der Hashwert. Der Ablauf ist in Abbildung 4.1 gezeigt.

Der Initialisierungsvektor IV wird dabei für jede Hashfunktion fest gewählt. Ist der letzte Block X_n zu kurz, wird er auf die benötigten m Bits ergänzt. Das Padding enthält dabei die Nachrichtenlänge, um zu verhindern, dass Verlängerungen der Nachricht für einen Angriff genutzt werden können.

4.3.2 Sicherheit von Merkle-Damgård

Die Sicherheit einer Merkle-Damgård-Konstruktion H_{MD} hängt stark von der verwendeten Kompressionsfunktion F ab:

Theorem 4.6. *Ist F kollisionsresistent, so ist auch H_{MD} kollisionsresistent.*

Beweis. *Seien X, X' zwei Urbilder von H_{MD} mit $X \neq X'$ und*

$$H_{MD}(X) = Z_n = Z'_n = H_{MD}(X')$$

Wir suchen nun eine Kollision in F.
Falls $(Z_{n-1}, X_n) \neq (Z'_{n-1}, X'_n)$, wurde eine Kollision in F gefunden, da $X_n \neq X'_n$, aber $Z_n = F(Z_{n-1}, X_n) = F(Z'_{n-1}, X'_n) = Z'_n$.
Falls nicht, prüfe, ob $(Z_{n-2}, X_{n-1}) \neq (Z'_{n-2}, X'_{n-1})$ und damit eine Kollision in F gefunden ist.
Falls nicht, prüfe, ob $(Z_{n-3}, X_{n-2}) \neq (Z'_{n-3}, X'_{n-2})$.
...
Da nach Voraussetzung $X \neq X'$ ist, muss irgendwann $(Z_{i-1}, X_i) \neq (Z'_{i-1}, X'_i)$ gelten. Damit ist die Kollision von H_{MD} auf eine Kollision in F zurückführbar.

□

4.3.3 Bedeutung von Merkle-Damgård

Es gibt einige bekannte Hashfunktionen, die auf dem Prinzip von Merkle-Damgård basieren. Darunter sind:

- MD5 (vorgeschlagen 1992)
- SHA-1 (vorgeschlagen 1995)
- SHA-2 (vorgeschlagen 2001)

MD5 und SHA-1 sind inzwischen gebrochen. Der aktuelle Hash-Standard SHA-3 („Keccak“) nutzt keine Merkle-Damgård-Konstruktion.

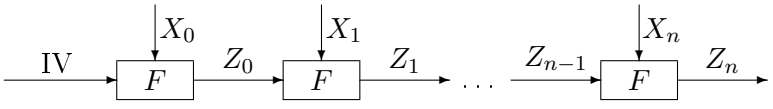


Abbildung 4.1: *Merkle-Damgård*-Konstruktion H_{MD}

4.3.3.1 SHA-1

SHA-1 ist gilt inzwischen zwar als gebrochen, war jedoch lange Zeit die wichtigste kryptographische Hashfunktion. Sie teilt die ursprüngliche Nachricht in Blöcke zu 512 Bits ein, wobei der letzte Block bei Bedarf auf 512 Bits aufgefüllt wird. Jeder Block durchläuft mit einem Initialisierungsvektor von 160 Bit Länge jeweils 20 Runden mit vier verschiedenen Funktionen. Das Ergebnis eines Durchgangs hat eine Länge von 160 Bit und wird rückgekoppelt als Initialwert für den nächsten Block genutzt. Das Ergebnis des letzten Durchgangs ist der Hashwert der Nachricht.

Eine Runde von SHA-1 ist schematisch in Abbildung „“ dargestellt. Abbildung „“ zeigt die gesamte Kompressionsfunktion.

4.4 Angriffe

4.4.1 Angriffe auf SHA-1

Die Angriffe auf SHA-1 nutzen die Möglichkeit aus, für eine Runde Kollisionen zu finden, und versuchen diese auf mehrere Runden auszuweiten. Dabei sind auch Ausgaben hilfreich, die nur ähnlich und nicht exakt gleich sind (also $H(X) \approx H(X')$).

2005 wurden dann in kurzer Abfolge zwei Angriffe entdeckt, die Kollisionen über 53 bzw. alle 80 Runden erzeugen können. Zwar sind diese Angriffe noch immer theoretisch und erfordern einen hohen Rechenbedarf (etwa 2^{61} Schritte), sind den typischen Angriffen jedoch um einige Größenordnungen überlegen.

4.4.2 Birthday-Attack

Für diesen Angriff berechnen wir möglichst viele $Y_i = H(X_i)$. Danach suchen wir unter diesen Hashwerten nach Gleichheit (und finden so $X \neq X'$ mit $H(X) = Y = Y' = H(X')$).

Vorgehen:

1. Schreibe (X_i, Y_i) in Liste. Dabei ist $X_i \in \{0,1\}^{2k}$ gleichverteilt und $Y_i = H(X_i)$.
2. Sortiere die Liste nach Y_i.
3. Untersuche die Liste auf Y_i-Kollisionen.

Theorem 4.7. *Sei $n \leq 2^{\frac{k}{2}}$ und $Y_1, \ldots, Y_n \in \{0,1\}^k$ unabhängig gleichverteilt. Dann gibt es $i \neq j$ mit $Y_i = Y_j$ mit Wahrscheinlichkeit $p > \frac{1}{11} \cdot \frac{n^2}{2^k}$.*

Wir haben also schon für $n = 2^{\frac{k}{2}}$ zufällige, verschiedene X_i mit einer Wahrscheinlichkeit von $p > \frac{1}{11}$ Kollisionen unter den den dazugehörigen Y_i. Für die Berechnung brauchen wir $\Theta(k \cdot 2^{\frac{k}{2}})$ Schritte und haben einen Speicherbedarf von $\Theta(k \cdot 2^{\frac{k}{2}})$ Bits.

Beispiel 4.8. *Alice möchte ein paar Tage Urlaub buchen und holt dafür Angebote ein. Mallory würde gern eine Weltreise machen, möchte die Kosten aber nicht tragen und nutzt jetzt aus, dass Alice gerade einen Urlaub buchen möchte. Er berechnet die Hashwerte von unterschiedlichen Kurztrips für Alice und einige Buchungen über eine Weltreise für sich selbst auf Kosten von Alice. Wenn er eine Kollision gefunden hat, also einen Hashwert, der sowohl einen Kurztrip als auch eine Weltreise abdeckt, lässt er von Alice den Hashwert der harmlosen Kurztripbuchung signieren. Bevor er diese jedoch an das Reisebüro weiterleitet, tauscht er die Buchungen aus. Das Reisebüro prüft die Buchung der Weltreise samt Hashwert und Signatur und befindet sie für gültig.*

4.4.3 Weitere Angriffe

Auch ein Meet-in-the-Middle-Angriff kann die Zeit zum Auffinden einer Kollision verkürzen. Allerdings setzt dieser Angriff voraus, dass die Hashfunktion eine „Rückwärtsberechnung“ zulässt.

Vorgehen:

1. Gehe aus von $M = M_1 M_2$.
2. Verändere M_1 möglichst oft, erzeuge eine Liste aller Z.
3. Sortiere die Liste aller Z.
4. Rechne von $Y = H(M)$ zurück zu Z für viele Kandidaten für M_2.

$$IV \xrightarrow{M_1} Z \xrightarrow{M_2} Y = H(M_1 M_2)$$

Abbildung 4.2: Hilfsskizze für Meet-in-the-Middle-Angriff auf eine Hashfunktion H

Als Ergebnis aus diesem Angriff erhalten wir M_1' und M_2' mit $H(M_1' M_2') = H(M_1 M_2)$. Der Aufwand für diesen Angriff nähert sich asymptotisch dem für die Geburtstagsattacke an.

4.4.4 Fazit

Die vorgestellten Angriffe zeigen, dass sich der Aufwand zum Finden einer Kollision gegenüber einer Brute-Force-Attacke stark verringern lässt. Bei einer Hash-Ausgabe mit einer Länge $\geq k$ Bits kann man nur mit einer „Sicherheit“ von $\frac{k}{2}$ Bits rechnen.

Kapitel 5

Asymmetrische Verschlüsselung

Symmetrische Verschlüsselung, wie wir sie in den letzten Kapiteln behandelt haben, funktioniert über ein gemeinsames Geheimnis K (siehe Abbildung 5.1). Das verursacht uns einige Unannehmlichkeiten:

- das gemeinsame Geheimnis K muss auf einem sicheren Kanal übertragen werden
- bei n Benutzern werden im System $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ Schlüssel verwendet (für jedes Teilnehmerpaar einen)

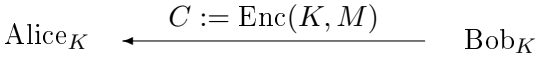


Abbildung 5.1: schematischer Ablauf einer symmetrisch verschlüsselten Kommunikation

5.1 Idee

Public-Key-Kryptographie basiert auf der Grundidee, für die Verschlüsselung (öffentlich) einen anderen Schlüssel zu verwenden als für die Entschlüsselung (privat). Abbildung 5.2 zeigt den Ablauf einer asymmetrisch verschlüsselten Kommunikation.

Die Vorteile eines Public-Key-Verfahrens sind offensichtlich. Wir benötigen für den Schlüsselaustausch keinen sicheren Kanal mehr, sondern könnten sogar ähnlich einem Telefonbuch ein öffentliches Verzeichnis mit den öffentlichen Schlüsseln anlegen. Außerdem müssen nicht mehr so viele Schlüssel gespeichert werden: Bei n Benutzern gibt es nur noch n öffentliche (und n geheime) Schlüssel.

Die Sicherheit eines solchen Verfahrens hängt davon ab, wie schwierig es für einen Angreifer ist, vom (allgemein bekannten) öffentlichen Schlüssel pk auf den (geheim gehaltenen) privaten Schlüssel sk zu schließen. Um das praktisch unmöglich zu machen, werden Probleme aus der Mathematik verwendet, die anerkannt schwierig zu lösen sind.

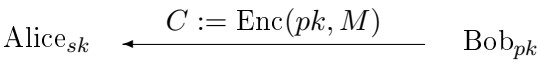


Abbildung 5.2: schematischer Ablauf einer asymmetrisch verschlüsselten Kommunikation

5.2 RSA

Das bekannteste Public-Key-Verfahren ist RSA (1977). Benannt nach seinen Erfindern (Rivest, Shamir, Adleman) macht es sich den enormen Aufwand zunutze, eine Zahl in ihre Primfaktoren zu zerlegen.

5.2.1 Vorgehen

Für die Erstellung eines Schlüsselpaares werden zwei große Primzahlen benötigt. Die Berechnung von öffentlichem und privatem Schlüssel funktioniert folgendermaßen:

- wähle zwei große Primzahlen P, Q mit $P \neq Q$ und vorgegebener Bitlänge k
- berechne $N = P \cdot Q$
- setze $\varphi(N) = (P-1)(Q-1)$
- wähle e mit $\text{ggT}(e, (P-1)(Q-1)) = 1$
- wähle d, sodass d und e zueinander invers, also $e \cdot d = 1 \mod \varphi(N)$
- setze den geheimen Schlüssel $sk = (N, d)$ und den öffentlichen Schlüssel $pk = (N, e)$

Üblicherweise werden P und Q zufällig gleichverteilt aus den ungeraden Zahlen der Länge k gezogen, bis P und Q prim sind. Aus $e \in \{3, \ldots, \varphi(N) - 1\}$ wird dann mit dem erweiterten euklidischen Algorithmus das Inverse berechnt: $d = e^{-1} \mod \varphi(N)$. Der Nachrichtenraum ist $\mathcal{M} := \mathbb{Z}_N$. Für die Ver- und Entschlüsselungsfunktionen gilt:

$$\begin{aligned} \textsc{Enc}(pk, M) &= M^e \mod N \\ \textsc{Dec}(sk, C) &= C^d \mod N \end{aligned}$$

Wie immer muss $\textsc{Dec}(\textsc{Enc}(M)) = M$ gelten. Für die Korrektheit von RSA bedeutet das, dass $(M^e)^d = M^{ed} = M \mod N$ erfüllt sein muss. Um das zu beweisen, verwenden wir den Kleinen Satz von Fermat und den Chinesischen Restsatz.

Theorem 5.1 (Kleiner Satz von Fermat)**.** *Für primes P und $M \in \{1, \ldots, P-1\}$ gilt: $M^{P-1} \equiv 1 \mod P$.*

Daraus folgt auch: $\forall M \in \mathbb{Z}_P, \alpha \in \mathbb{Z} : (M^{P-1})^\alpha \cdot M \equiv M \mod P$.

Theorem 5.2 (Chinesischer Restsatz)**.** *Sei $N = P \cdot Q$ mit P, Q teilerfremd. Dann ist die Abbildung $\mu : \mathbb{Z}_N \to \mathbb{Z}_P \times \mathbb{Z}_Q$ mit $\mu(M) \equiv (M \mod P, M \mod Q)$ bijektiv.*

Daraus folgt auch: $(X \equiv Y \mod P) \wedge (X \equiv Y \mod Q) \Rightarrow X \equiv Y \mod N$.

Theorem 5.3 (Korrektheit von RSA)**.** *Sei $N = P \cdot Q$ mit P, Q teilerfremd und prim. Seien weiter e, d teilerfremd wie oben. Dann ist $M^{ed} \equiv M \mod N$ für alle $M \in \mathbb{Z}_N$.*

Beweis. *Nach Definition gilt $e \cdot d \equiv 1 \mod (P-1)(Q-1)$. Daraus folgt:*

$$\begin{aligned} (P-1)(Q-1) \mid ed-1 \quad &\Rightarrow \quad P-1 \mid ed-1 \\ &\Rightarrow \quad ed = \alpha(P-1)+1 \quad (\textit{für } \alpha \in \mathbb{Z}) \\ &\Rightarrow \quad M^{ed} = (M^{(P-1)})^\alpha \cdot M \overset{Fermat}{\equiv} M \mod P \end{aligned}$$

Analog ist $M^{ed} \equiv M \mod Q$.
Da $N = P \cdot Q$ *ergibt sich mithilfe des Chinesischen Restsatzes:*

$$(M^{ed} \equiv M \mod P) \wedge (M^{ed} \equiv M \mod Q) \Rightarrow M^{ed} \equiv M \mod N$$

□

Das bisher behandelte Verfahren nennt sich *Textbook-RSA* und umfasst das grundlegende Prinzip von RSA. Textbook-RSA weist einige Schwächen auf und sollte daher in der Praxis nicht verwendet werden.

5.2.2 Sicherheit von RSA

Bevor wir die Sicherheit von RSA betrachten, benötigen wir einen Sicherheitsbegriff, an dem wir uns bei der Beurteilung von asymmetrischen Verschlüsselungsverfahren orientieren können. Wir definieren semantische Sicherheit, vergleichbar mit der Definition für symmetrische Chiffren in Kapitel 3.1 und äquivalent zu IND-CPA.

Definition 5.4 (Semantische Sicherheit für Public-Key-Verfahren)**.** Ein Public-Key-Verschlüsselungsschema ist *semantisch sicher*, wenn es für jede M-Verteilung von Nachrichten gleicher Länge, jede Funktion f und jeden PPT-Algorithmus $\mathcal{A}$ einen PPT-Algorithmus $\mathcal{B}$ gibt, so dass

$$\Pr\left[\mathcal{A}(1^k, pk, \textsc{Enc}(pk, M)) = f(M)\right] - \Pr\left[\mathcal{B}(1^k) = f(M)\right]$$

vernachlässigbar (als Funktion im Sicherheitsparameter) ist.

Umgangssprachlich formuliert bedeutet semantische Sicherheit, dass jeder Angreifer über ein Chiffrat C nur die Länge der Eingabe lernt.

RSA ist deterministisch, d.h. eine Nachricht M wird unter Verwendung desselben Schlüssels pk immer zu C_M verschlüsselt. Ein Angreifer kann zwei Chiffrate effizient voneinander unterscheiden (z.B. $\textsc{Enc}(pk, \text{annehmen})$ und $\textsc{Enc}(pk, \text{ablehnen})$). RSA ist also nicht semantisch sicher.

Es gibt noch einige andere Angriffspunkte, die im Folgenden umrissen werden.

Wahl von e: Aus Effizienzgründen liegt auf den ersten Blick es nahe, den Parameter e aus dem öffentlichen Schlüssel nicht für jeden Benutzer neu zu berechnen, sondern für alle gleich zu wählen. Da diese Wahl nur den öffentlichen Schlüssel betrifft, scheint diese Einschränkung nicht kritisch zu sein, führt jedoch zu Problemen, wenn dieselbe Nachricht M an mindestens drei unterschiedliche Benutzer verschlüsselt wird. Setzen wir für dieses Beispiel $e = 3$. Ein Angreifer, der die drei öffentlichen Schlüssel pk_1, pk_2, pk_3 kennt, mit denen M verschlüsselt wurde, kann sich die Nachricht M berechnen:

$$\begin{aligned} & M^3 \mod N_i && \text{für } 1 \leq i \leq 3 \\ \equiv\ & M^3 \mod N_1 N_2 N_3 && \text{(Chinesischer Restsatz)} \\ \equiv\ & M^3 && \text{(wegen } 0 \leq M \leq N_1, N_2, N_3\text{)} \end{aligned}$$

Wurzelziehen über $\mathbb{Z}$ liefert die Nachricht M.

Wahl von N: Auch N für alle Benutzer gleich zu vergeben schwächt das Verschlüsselungssystem. Wird wieder dieselbe Nachricht M mit zwei öffentlichen Schlüsseln (e_1, N)

und (e_2, N) chiffriert und gilt weiterhin $\text{ggT}(e_1, e_2) = 1$ in $\mathbb{Z}$, kann ein Angreifer aus den Chiffraten M berechnen:

$$\begin{aligned} & re_1 + se_2 = 1 \\ \Longrightarrow C_1^r C_2^s \mod N &\equiv M^{re_1} M^{se_2} \mod N \\ &\equiv M^{re_1 + se_2} \mod N \\ &\equiv M \end{aligned}$$

Es besteht in diesem Szenario die noch gravierendere Gefahr, dass ein Teilnehmer A aus seinem eigenen öffentlichen Schlüssel $pk_A = (e_A, N)$ und dem eines Teilnehmers B $pk_B = (e_B, N)$ ein d'_B berechnen kann, das äquivalent zu d_B aus Bs privatem Schlüssel ist. A ist also durch einfache Anwendung des Erweiterten Euklidischen Algorithmus in der Lage, sich ein $sk'_B = (d'_B, N)$ zu erstellen, mit dem sie alle an B verschlüsselten Nachrichten dechiffrieren kann.

Homomorphie: Wir betrachten eine Auktion mit dem Auktionsleiter A und zwei Bietern B_1 und B_2. Damit keiner der Interessenten einen anderen knapp überbietet oder sich von den Geboten anderer in seiner eigenen Abgabe beeinflussen lässt, nimmt der Auktionator die Gebote verschlüsselt entgegen. Dafür hat er seinen öffentlichen Schlüssel pk_A zur Verfügung gestellt. Das Gebot eines Bieters wird chiffriert und zur Aufbewahrung an den Auktionator geschickt, wie in Abbildung 5.3 dargestellt. Wenn die Zeit abgelaufen ist, werden keine neuen Preisvorschläge mehr angenommen, die eingegangenen Gebote entschlüsselt und der Höchstbietende ermittelt.

Der unehrliche Bieter B_2 kann nun seinen Preisvorschlag mithilfe des verschlüsselten Gebots von B_1 zu seinen Gunsten wählen. Dafür setzt er z.B. $C_2 = C_1 \cdot \text{Enc}(pk_A, 2)$ oder, wenn er besonders sparsam ist, $C_2 = C_1 \cdot \text{Enc}(pk_A, 1001/1000 \mod N)$. Damit kann er das Gebot von B_1 verdoppeln bzw. knapp überbieten, ohne dass der Auktionator und der ehrliche Bieter B_1 ihm Betrug nachweisen können.

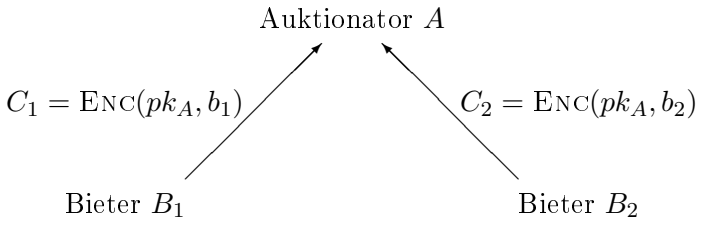


Abbildung 5.3: schematischer Ablauf einer Auktion mit verschlüsselten Geboten

5.2.3 Sicheres RSA

Wir haben festgestellt, dass RSA deterministisch und damit nicht semantisch sicher ist. Die gepaddete RSA-Variante RSA-OAEP dagegen ist IND-CCA-sicher. Wir verwenden dabei eine Zufallszahl R, mit deren Hilfe wir die Nachricht M vor dem Verschlüsseln abwandeln. Zu diesem Zweck wird die in Abbildung 5.4 dargestellte Konstruktion von Hashfunktionen G, H verwendet. Wir können R nach dem Entschlüsseln gut wieder entfernen, aber $\text{Enc}_R(M)$ lässt sich nun nicht mehr so einfach mit anderen Chiffraten abgleichen.

Diese Konstruktion ist heuristisch genau so sicher, wie N zu faktorisieren.

5.2.4 Bedeutung von RSA

Im Gegensatz zu den meisten symmetrischen Chiffren basiert RSA als Beispiel einer asymmetrischen Verschlüsselungstechnik nicht auf einfachen, bit-orientierten sondern auf einer

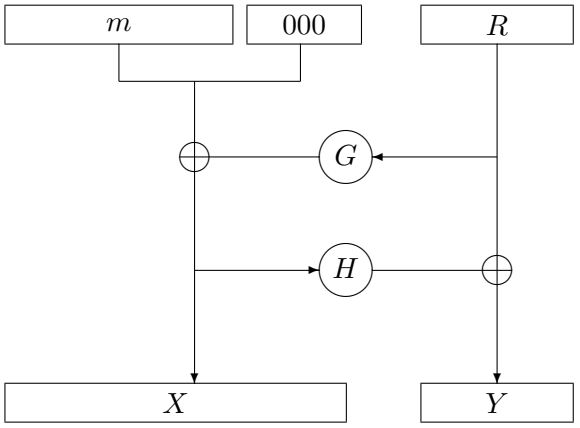


Abbildung 5.4: pad-Funktion von RSA-OAEP (G, H sind Hashfunktionen)

mathematischen Funktion. Der für Ver- und Entschlüsselung sowie für die Schlüsselerzeugung nötige Rechenaufwand steigt dadurch ungemein: ein naiver Exponentiationsalgorithmus benötigt für die Berechnung einer modulo l-Bit-Zahl $\omega(l)$ Bitoperationen.

Nichtsdestotrotz wird RSA in der Praxis häufig eingesetzt. Es macht sich relativ relativ einfache Arithmetik zunutze und die Ähnlichkeit zwischen Ver- und Entschlüsselungsfunktion vereinfachen die Implementierung zusätzlich. Mit einfachen Anpassungen ($e = 3$ bei Verschlüsselung, Chinesischer Restsatz nutzen bei Entschlüsselung) kann RSA so weit beschleunigt werden, dass es die Laufzeit betreffend gegenüber anderen Verschlüsselungsverfahren konkurrenzfähiger wird.

5.3 ElGamal

Das ElGamal-Verfahren (1985) macht sich die Schwierigkeit zunutze, das Diffie-Hellman-Problem (Berechnung von diskreten Logarithmen in zyklischen Gruppen) zu lösen.

5.3.1 Vorgehen

Für die Erzeugung der Schlüssel benötigen wir eine angemessen große, zyklische Gruppe $\mathbb{G}$ mit dem Erzeuger g: $\mathbb{G} = \langle g \rangle$. Geeignete Kandidaten für eine solche Gruppe sind (echte) Untergruppen von $\mathbb{Z}_P^*$ mit P prim oder allgemeiner Untergruppen von $\mathbb{F}_q^*$ mit q Primpotenz mit einer Gruppengröße von $|\mathbb{G}| \approx 2^{2048}$. Effizienter sind Untergruppen von elliptischen Kurven $\mathbf{E}(\mathbb{F}_q^*)$ mit einer Gruppengröße von $|\mathbb{G}| \approx 2^{200}$.

Wir wählen außerdem eine Zufallszahl x und berechnen $h = g^x$. Für unser Schlüsselpaar gilt jetzt:

- $pk = (\mathbb{G}, g, h)$
- $sk = (\mathbb{G}, g, x)$

Wenn Alice uns eine Nachricht M schicken möchte, wählt sie ein y zufällig gleichverteilt, berechnet damit $Y = g^y$ und $Z = h^y M$ und sendet das Tupel (Y, Z). Wir können die Nachricht entschlüsseln, indem wir auflösen:

$$\begin{aligned} & Z = h^y M \\ \Leftrightarrow \quad & M = \frac{Z}{h^y} = \frac{Z}{g^{xy}} = \frac{Z}{(g^y)^x} = \frac{Z}{Y^x} \end{aligned}$$

Für Ver- und Entschlüsselung gilt also:

- $\textsc{Enc}(pk, M) = (g^y, h^y \cdot M)$ (mit y zufällig)
- $\textsc{Dec}(sk, (Y, Z)) = \frac{Z}{Y^x}$

Durch die zufällige Wahl von y ist das Chiffrat $\textsc{Enc}(pk, M)$ randomisiert. ElGamal ist somit semantisch sicher. Allerdings ist ElGamal wie RSA homomorph:

$$\begin{aligned}\textsc{Enc}(pk, M) \cdot \textsc{Enc}(pk, M') &= (g^y, g^{xy} \cdot M) \cdot (g^{y'}, g^{xy'} \cdot M') \\ &= (g^{y+y'}, g^{x(y+y')} \cdot M \cdot M') \\ &= \textsc{Enc}(pk, M \cdot M')\end{aligned}$$

Es existieren allerdings bereits nicht-homomorphe Varianten von ElGamal.

Kapitel 6

Symmetrische Authentifikation von Nachrichten

Bisher haben wir uns nur mit der Frage beschäftigt, wie ein Kommunikationsteilnehmer Bob eine Nachricht an Alice für einen unbefugten Lauscher unverständlich machen, also verschlüsseln kann. Wir haben uns noch nicht der Frage nach der Authentifikation einer Nachricht gewidmet. Der Angreifer könnte mit dem entsprechenden Zugriff auf den Übertragungskanal sogar eine verschlüsselte Kommunikation beeinflussen, deren Inhalt er nicht versteht. Er kann Nachrichten abfangen, verändern und wieder auf den Weg bringen, ohne dass Alice oder Bob etwas von dem Zwischenstopp der Nachricht bemerken.

Falls ein Angreifer trotz der Verschlüsselungsmaßnahmen außerdem in der Lage ist, die Kommunikation zu verstehen, könnte er sogar *gezielt* den Inhalt von Nachrichten verändern. Gezieltes, aber auch ungezieltes Verändern von Nachrichten bezeichnen wir als *aktive Angriffe*. Es kann jedoch auch ohne Angreifer geschehen, dass der Kommunikationskanal gestört und Bobs Nachricht durch technische Einwirkungen abgewandelt wird.

Im besten Fall erhält Alice dann eine unbrauchbare Nachricht und kann bei Bob eine Wiederholung anfordern. Im schlechtesten Fall ist die Veränderung zufällig (oder vom Angreifer gewollt) sinnvoll und beeinflusst damit das weitere Vorgehen der beiden Kommunikationsteilnehmer.

6.1 Ziel

Angesichts dessen, dass wir uns unseren Kommunikationskanal nicht immer aussuchen können, hätten wir gern einen Mechanismus, der uns ermöglicht, eine erhaltene Nachricht auf Fehler und Veränderungen zu überprüfen (Integrität) und den Absender zu bestimmen (Authentizität). Dafür erstellt Bob für seine Nachricht M zusätzlich eine „Unterschrift“ σ und überträgt diese gemeinsam mit M. Alice erhält das Paar (M, σ) und überprüft, ob die Unterschrift auf die erhaltene Nachricht passt.

Um ein funktionierendes und gegen einen Angreifer möglichst sicheres Unterschriftensystem zu erhalten, müssen einige Anforderungen erfüllt sein:

- Bob muss σ aus der bzw. für die Nachricht M berechnen können
- Alice muss σ zusammen mit M verifizieren können
- ein Angreifer soll kein gültiges σ für ein selbst gewähltes M berechnen können

6.2 MACs

Die Idee von *Message Authentication Codes* (MACs) basiert auf der Grundannahme, dass Alice und Bob bereits ein gemeinsames Geheimnis K besitzen. Es ist also ein symmetrisches

Verfahren. Signatur und Verifikation der Unterschrift funktionieren dann wie folgt:

- **Signieren:** $\sigma \leftarrow \textsc{Sig}(K, M)$
- **Verifizieren:** $\textsc{Ver}(K, M, \sigma) \in \{0, 1\}$

Es muss immer gelten: $\textsc{Ver}(K, M, \textsc{Sig}(K, M)) = 1$. Eine mit Sig erstellte Signatur muss also unter demselben Schlüssel K von Ver als gültig ausgezeichnet werden.

6.3 Der EUF-CMA-Sicherheitsbegriff

Damit ein MAC uns nicht nur vor Übertragungsfehlern sondern auch vor einem Angreifer schützt, verlangen wir, dass kein Angreifer selbstständig ein Nachrichten-Signatur-Paar finden kann, das gültig ist.

Er bekommt dafür ein Signaturorakel mit vor ihm verborgenem Schlüssel K, mit dem er Nachrichten seiner Wahl signieren kann. Er gewinnt, wenn er die Signatur einer Nachricht M korrekt vorhersagen kann, ohne M vorher an das Orakel gegeben zu haben. Etwas strukturierter sieht der Angriff für einen PPT-Angreifer $\mathcal{A}$ so aus:

1. $\mathcal{A}$ erhält Zugriff auf ein Signaturorakel $\textsc{Sig}(K, \cdot)$
2. $\mathcal{A}$ wählt ein Nachrichten-Signatur-Paar (M^*, σ^*)
3. $\mathcal{A}$ gewinnt, wenn das Orakel zu M^* σ^* ausgibt, wenn also $\textsc{Ver}(K, M^*, \sigma^*) = 1$, und das Signaturorakel vorher M^* noch nicht von $\mathcal{A}$ erhalten hat

Dieses Schema bildet passive Angriffe ab, bei denen der Angreifer keinen Zugriff auf die Ver-Funktion hat, sondern „blind“ signiert. Wenn es für ein M nur ein einziges, gültiges σ gibt, ist dieser Angriff dagegen äquivalent zu einer Attacke mit Zugriff auf die Ver-Routine.

Einen Signaturalgorithmus, der sicherstellt, dass $\mathcal{A}$ dieses Spiel nur vernachlässigbar oft gewinnt, nennen wir *EUF-CMA-sicher* (EUF: Existential Unforgeability, CMA: Chosen Message Attack).

6.4 Konstruktionen

6.4.1 Hash-then-Sign Paradigma

Eine Signatur, die so lang ist, wie das ursprüngliche Dokument, erfüllt zwar ihren Zweck, was Datenintegrität und Authentizität angeht, ist jedoch unpraktikabel. Wir suchen also einen Mechanismus, der uns bei variabler Nachrichtenlänge eine kurze Signatur einer Nachricht M ermöglicht. Dafür bieten sich sofort Hashfunktionen an.

Die Idee des Hash-then-Sign Paradigmas ist also, nicht die vollständige Nachricht $H \in \{0,1\}^*$ zu signieren, sondern den aus dieser Nachricht berechneten Hashwert $H(M) \in \{0,1\}^k$. Die Sicherheit des MACs ist dabei sowohl von der verwendeten Hashfunktion als auch vom Signaturalgorithmus abhängig.

Theorem 6.1. *Sei* $(\textsc{Sig}, \textsc{Ver})$ *EUF-CMA-sicher und* H *eine kollisionsresistente Hashfunktion. Dann ist der durch*

$$\textsc{Sig}(K, M) = \textsc{Sig}(K, H(M))$$
$$\textsc{Ver}(K, M, \sigma) = \textsc{Ver}(K, H(M), \sigma)$$

erklärte MAC EUF-CMA-sicher.

Beweis (Entwurf). *Der EUF-CMA-Angreifer $\mathcal{A}$ muss entweder eine Kollision in der Hashfunktion H finden und kann damit auf dieselbe Signatur von zwei Nachrichten schließen oder er muss einen einzelnen Hashwert $H(M)$ finden, für den er eine Signatur errechnen kann.*

6.4.2 Pseudorandomisierte Funktionen

Wenn man sich die Berechnung eines MACs als eine einfache Funktion im mathematischen Sinne vorstellt und damit die Errechnung eines „frischen“ MACs zum Finden eines unbekannten Funktionswertes wird, erkennt man schnell, dass Regelmäßigkeit in einer solchen Funktion zu Sicherheitslücken führt. Zielführender ist es, die Funktionswerte möglichst zufällig auf ihre Urbilder zu verteilen.

Definition 6.2 (Pseudorandomisierte Funktion (PRF)). Sei
$PRF\colon \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ eine über $k \in \mathbb{N}$ parametrisierte Funktion. PRF heißt Pseudorandom Function (PRF), falls für jeden PPT-Algorithmus $\mathcal{A}$ die Funktion

$$\mathrm{Adv}^{prf}_{PRF,\mathcal{A}}(k) := \Pr[\mathcal{A}^{PRF(K,\cdot)}(1^k) = 1] - \Pr[\mathcal{A}^{R(\cdot)}(1^k) = 1]$$

vernachlässigbar ist, wobei $R : \{0,1\}^k \to \{0,1\}^k$ eine echt zufällige Funktion ist.

Ein Kandidat für eine solche PRF ist eine Hash-Konstruktion: $PRF(K,X) = H(K,X)$. Allerdings lässt sich eine solche Konstruktion manchmal, wie bereits in Abschnitt 4.3 bei Merkle-Damgård ausgenutzt, nach der Berechnung von $H(K,X)$ auch ohne Zugriff auf das Geheimnis K noch auf $H(K,X,X')$ erweitern. Das führt dazu, dass die PRF-Eigenschaft für Eingaben unterschiedlicher Länge nicht mehr hält. Abbildung 6.1 verdeutlicht das.

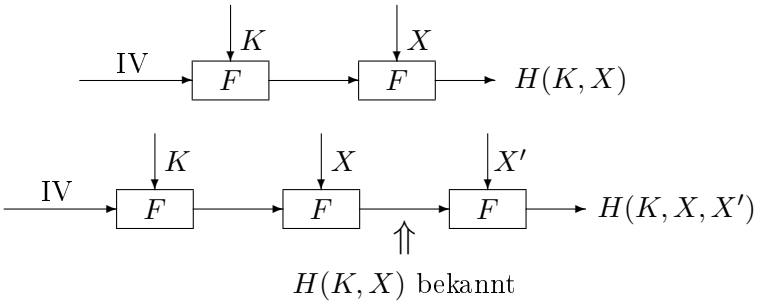


Abbildung 6.1: *Merkle-Damgård*-Konstruktion H_{MD}. Es ist möglich, an einen bereits bekannten Hashwert $H(K,X)$ einen Wert X' anzuhängen und trotzdem einen korrekten Hashwert zu erzeugen.

Theorem 6.3. *Sei $PRF\colon \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ eine PRF und $H\colon \{0,1\}^* \to \{0,1\}^k$ eine kollisionsresistente Hashfunktion. Dann ist der durch $\mathrm{Sig}(K,M) = PRF(K,H(M))$ gegebene MAC EUF-CMA-sicher.*

Beweis (Entwurf). *Sei $\mathcal{A}$ ein erfolgreicher EUF-CMA-Angreifer auf ein durch $\mathrm{Sig}(K,M) = PRF(K,H(M))$ gegebenen MAC. Dann können wir annehmen, dass $\mathcal{A}$ eine Fälschung (M^*,σ^*) mit einer noch nicht signierten Nachricht M^* berechnen kann. Wir können also $\mathcal{A}$ als PRF-Unterscheider auffassen, der mit nicht-vernachlässigbarer Wahrscheinlichkeit $PRF(K,H(M^*))$ vorhersagt. Eine Vorhersage ist jedoch nur dann möglich, wenn PRF keinen Zufall ausgibt. Da PRF jedoch nach Definition nur mit vernachlässigbarer Wahrscheinlichkeit von echtem Zufall unterscheidbar ist, kann es einen solchen PRF-Unterscheider nicht geben.*

6.4.3 HMAC

Im Abschnitt zu Pseudorandomisierten Funktionen haben wir gesehen, dass Signaturverfahren, die *Merkle-Dåmgard* nutzen, dem EUF-CMA-Sicherheitsbegriff im Allgemeinen nicht genügen. Ein Angreifer, dem $\sigma = H(K, M)$ bekannt ist, erhält durch Anfügen eines Blockes X problemlos den korrekten Hashwert $H(K, M, X)$ und somit die Signatur der Nachricht M, X. Dennoch ist es möglich, EUF-CMA-sichere MACs zu konstruieren, die mittels einer *Merkle-Dåmgard*-Konstruktion signieren. Eines der verbreitetsten Verfahren ist der *Keyed-Hash Message Authentication Code*, der HMAC. Das Signieren einer Nachricht funktioniert dabei wie folgt:

$$\textsc{Sig}(K, M) = H(K \oplus \mathit{opad}, H(K \oplus \mathit{ipad}, M))$$

Dabei sind *opad*, das *outer padding* und *ipad*, das *inner padding* zwei Konstanten der Blocklänge m der Hashfunktion, die bei jedem Signaturvorgang gleich bleiben. Üblich[1] ist es, $\mathit{opad} = \{0x5C\}^{m/8}$ und $\mathit{ipad} = \{0x36\}^{m/8}$ zu wählen. Das Verifizieren funktioniert analog zu der in 6.2 gegebenen Definition.

Immun gegen den in Abbildung 6.1 vorgestellten Angriff ist HMAC aufgrund seiner verschachtelten Struktur. Die Nachricht M, die es zu Signieren gilt, wird in jeweils zwei Hashvorgängen verarbeitet. Für eine Nachricht M, X ist

$$H(K \oplus \mathit{opad}, H(K \oplus \mathit{ipad}, M), X)$$

aber offensichtlich keine gültige Signatur. Der Angreifer müsste einen Nachrichtenblock X bereits im inneren Hashvorgang unterbringen. Da er dafür allerdings H invertieren, oder das Geheimnis K kennen müsste, schlägt der Angriff fehl.

Selbstverständlich muss, obwohl HMAC das eben angesprochene Problem löst, keine *Merkle-Dåmgard*-Funktion verwendet werden. Für jede pseudorandomisierte Hashfunktion genügt HMAC dem EUF-CMA-Sicherheitsbegriff.

[1] Sowohl im RFC 2104, sowie in einer Veröffentlichung des NIST und in diverser Fachliteratur werden diese Werte (als Standard) vorgeschlagen. Siehe:

http://tools.ietf.org/html/rfc2104
http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf

Kapitel 7

Asymmetrische Authentifikation von Nachrichten

Wie wir bereits bei den Verschlüsselungsverfahren festgestellt haben, weisen symmetrische Verfahren einige Unbequemlichkeiten auf. Allen voran stellt sich das Problem der Schlüsselverteilung, wenn für die Kommunikation zwischen zwei Partnern bei beiden derselbe Schlüssel vorhanden sein muss. Dieses Problem stellt sich natürlich umso mehr, wenn wir über einen nicht vertrauenswürdigen Kanal kommunizieren. Selbst für die Authentifikation unserer Nachrichten, die wir im Zweifelsfall nur betreiben, weil wir dem Kanal nicht vertrauen, müssen wir unter einigem Aufwand Schlüssel mit unseren Kommunikationspartnern festlegen.

Symmetrische Authentifikation verhindert außerdem, dass wir von uns veröffentlichte Dokumente oder Nachrichten unterschreiben und damit die Urheberschaft für jeden nachprüfbar machen können. Zur Authentifikation des Dokuments sollte schließlich jeder befähigt sein, der sich dafür interessiert. Wenn wir mit symmetrischen Verfahren arbeiten, bedeutet das, dass wir zur Prüfung den Schlüssel herausgeben müssen, mit dem wir das Dokument signiert haben. Das bedeutet aber auch, dass jeder Interessierte nun nicht nur zur Prüfung der bereits bestehenden Signatur in der Lage ist, sondern auch eigene Signaturen erstellen kann. Damit ist die Urheberschaft einer Unterschrift nicht mehr gesichert.

Es bietet sich ein Verfahren an, bei dem die Prüfung einer Signatur nicht mit einem privaten Schlüssel erfolgt. Dieses System kennen wir bereits aus dem Kapitel 5. Bei der Verwendung von asymmetrischen Verfahren zur Authentifikation verwenden wir die folgenden Algorithmen:

- $(pk, sk) \leftarrow \textsc{KeyGen}(1^k)$ zur Schlüsselgenerierung
  - pk : öffentlicher Schlüssel
  - sk : privater Schlüssel
  - k : Sicherheitsparameter
- $\sigma \leftarrow \textsc{Sig}(sk, M)$ zum Signieren
- $\textsc{Ver}(pk, M, \sigma) \in \{0, 1\}$ zum Verifizieren

Sig und Ver müssen korrekt sein, d.h. es muss wie bei MACs gelten:

$$\forall (pk, sk) \leftarrow \textsc{KeyGen}(1^k), \forall M, \forall \sigma \leftarrow \textsc{Sig}(sk, M) : \textsc{Ver}(pk, M, \sigma) = 1$$

Wir passen außerdem die Definition der EUF-CMA-Sicherheit aus Abschnitt 6.3 formlos an asymmetrische Verfahren an.

1. $\mathcal{A}$ erhält Zugriff auf ein Signatur-Orakel $\textsc{Sig}(sk, \cdot)$

2. $\mathcal{A}$ gibt dem Signaturorakel beliebig oft (M^*, σ^*)
3. $\mathcal{A}$ gewinnt, wenn er ein „frisches“ M^* findet, sodass $\textsc{Ver}(pk, M^*, \sigma^*) = 1$

Ein asymmetrisches Signaturverfahren ist EUF-CMA-sicher, wenn der Angreifer $\mathcal{A}$ das oben genannten Spiel nur vernachlässigbar oft gewinnt.

7.1 RSA

Wir betrachten zuerst RSA als Kandidaten für ein asymmetrisches Signaturverfahren. RSA besteht, wie in Kapitel 5.2.1 entwickelt, aus den folgenden Algorithmen:

$$\begin{aligned}\textsc{Enc}(pk, M) &= M^e \mod N\\ \textsc{Dec}(sk, C) &= C^d \mod N\end{aligned}$$

Unser privater Schlüssel ist $sk = (N, d)$ und der öffentlichen Schlüssel $pk = (N, e)$. In ein Signaturverfahren umgewandelt, sollte RSA also folgendermaßen funktionieren:

$$\begin{aligned}\textsc{Sig}(sk, M) &= M^d \mod N\\ \textsc{Ver}(pk, M, \sigma) = 1 &:\Leftrightarrow M = \sigma^e \mod N\end{aligned}$$

Um das zu erreichen, vertauschen wir beim Signieren im Gegensatz zum Verschlüsseln einfach die Dec- und die Enc-Routinen.

Allerdings stoßen wir hier erneut auf ein Problem, das wir im Abschnitt 5.2.2 bereits untersucht haben. Der Determinismus von Enc stellt auch beim Signieren ein Sicherheitsrisiko dar. Allerdings ergeben sich mit dieser simplen Lösung noch einige weitere Angriffsmöglichkeiten.

Signatur abhängiger Nachrichten: Ein Angreifer wählt zunächst ein beliebiges $\sigma \in \mathbb{Z}$. Dann kann er mithilfe des öffentlichen Schlüssels pk zu dieser Signatur einfach ein M generieren, zu dem die Signatur σ passt: $M := \sigma^e \mod N$.
Zwar ist für diesen Angriff im ersten Moment keine sinnvollen Nutzung ersichtlich, die Problematik eines Missbrauchs besteht jedoch prinzipiell. Dieser Angriff bricht also die für ein Signaturverfahren geforderte EUF-CMA-Sicherheit.

Homomorphie von RSA: Angenommen, ein Angreifer ist im Besitz dreier Nachrichten M_1, M_2, M_3 mit $M_1 \cdot M_2 = M_3$. Dann könnte er den Besitzer eines privaten Schlüssels dazu bringen, die beiden (womöglich harmlosen) Nachrichten M_1 und M_2 zu signieren und damit eine gültige Signatur für M_3 erhalten, da aufgrund der Homomorphie und der Beziehung zwischen den Nachrichten gilt: $\textsc{Sig}(M_3) = \textsc{Sig}(M_1) \cdot \textsc{Sig}(M_2) \mod N$. Neue Signaturen lassen sich so aus bereits bekannten Signaturen errechnen.
Für diesen Angriff sind intuitiv bereits einige Anwendungsmöglichkeiten denkbar. Auch hier hält die geforderte EUF-CMA-Sicherheit nicht.

Wie bereits bei der RSA-Verschlüsselung in Kapitel 5.2.2 können wir diese Probleme lösen, indem wir die Nachricht vor der Verarbeitung padden:

$$\begin{aligned}\textsc{Sig}(sk, M) &= (\text{pad}(M))^d \mod N\\ \textsc{Ver}(pk, \sigma, M) = 1 &:\Leftrightarrow \sigma^e \mod N \text{ ist gültiges Padding für } M\end{aligned}$$

Das so entstehende Signaturverfahren nennt sich (RSA-)PSS (*Probabilistic Signature Scheme*) und ist wie RSA-OAEP (als Teil der *PKCS*) kryptographischer Standard.

Unter Verwendung idealer Hashfunktionen und mit der Annahme, dass die RSA-Funktion schwierig zu invertieren ist, ist RSA-PSS heuristisch EUF-CMA sicher. Ein Angreifer ist

gezwungen, die RSA-Funktion direkt anzugreifen. Der beste bekannte Angriff basiert auf der Faktorisierung von N (unter Verwendung des Zahlkörpersiebs). Die Parameter werden ähnlich wie bei RSA-OAEP gewählt und haben so eine Länge von meistens 2048 Bit. Um eine effiziente Verifikation der Signaturen zu gewährleisten, ist es außerdem ohne Schwierigkeiten möglich, den Parameter e klein zu wählen.

7.2 ElGamal

Auch das ElGamal-Verfahren aus Kapitel 5.3 lässt sich zu einem funktionierenden Signaturverfahren ausbauen. Sei für unseren ersten Versuch der geheime Schlüssel $sk = (\mathbb{G}, g, x)$ und der öffentliche Schlüssel $pk = (\mathbb{G}, g, g^x)$. Dann bietet sich die Verwendung von ElGamal zur Erzeugung einer Signatur auf diese Art an:

$$\begin{aligned} \textsc{Sig}(sk, M) &= a \text{ mit } a \cdot x = M \mod |\mathbb{G}| \\ \textsc{Ver}(pk, \sigma, M) = 1 &:\Leftrightarrow (g^x)^a = g^M \end{aligned}$$

Allerdings lässt sich diese Konstruktion auf einfache Art brechen, indem mit $x = Ma^{-1} \mod \mathbb{G}$ der geheime Schlüssel berechnet wird.

Auch hier führt uns also nicht sofort der intuitive Ansatz zu einem sicheren Signaturverfahren. Stattdessen wählt Alice, die eine Nachricht signieren will, eine Zahl e zufällig und berechnet damit:

$$\begin{aligned} a &:= g^e \\ a \cdot x + e \cdot b &= M \mod |\mathbb{G}| \\ &\Leftrightarrow b := (M - a \cdot x) \cdot e^{-1} \mod |\mathbb{G}| \end{aligned}$$

Daraus ergibt sich dann als Signatur- und Verifikationsalgorithmen:

$$\begin{aligned} \textsc{Sig}(sk, M) &= (a, b) \\ \textsc{Ver}(pk, \sigma, M) = 1 &:\Leftrightarrow (g^x)^a \cdot a^b = g^{ax} \cdot g^{be} = g^M \end{aligned}$$

Doch auch bei dieser Variante gibt es noch einige offene Angriffspunkte:

Doppelte Verwendung von e: Wird der zufällige Parameter e mehrmals zur Erzeugung von Signaturen verwendet, kann der geheime Schlüssel x aus den beiden Signaturen errechnet werden. Seien die Signaturen $(a = g^e, b, M)$ und $(a' = g^{e'} = a, b', M')$. Dann ergibt sich durch Aufaddieren und Umformen der Gleichungen

$$\begin{aligned} & a \cdot x + e \cdot b = M \mod |\mathbb{G}| \\ \wedge \quad & a \cdot x + e \cdot b' = M' \mod |\mathbb{G}| \\ \Rightarrow \quad & e = \frac{M - M'}{b - b'} \mod |\mathbb{G}| \end{aligned}$$

Mit bekanntem e kann also wiederum auf den geheimen Schlüssel x geschlossen werden. Bei zufälliger Wahl geschieht es nur vernachlässigbar oft, dass zwei Mal dasselbe e ausgewählt wird und infolgedessen ausgenutzt werden kann.

Erzeugung „unsinniger“ Signaturen: Durch günstige Wahl der Parameter ist es möglich, auch ohne Kenntnis des Schlüssels x gültige Signaturen zu erzeugen. Wähle zunächst c zufällig. Setze außerdem:

$$\begin{aligned} a &:= g^c g^x = g^{c+x} = g^e \\ b &:= -a \mod |\mathbb{G}| \end{aligned}$$

Dann ist (a, b) eine gültige Signatur zu M mit

$$\begin{aligned} M &:= a \cdot x + b \cdot e \\ &= a \cdot x - a \cdot (c + x) \\ &= -ac \mod |\mathbb{G}| \end{aligned}$$

ElGamal ist also ebenfalls nicht EUF-CMA sicher. Wie bei RSA muss hier noch Zusatzarbeit geleistet werden. Dafür verwenden wir im Folgenden Hashfunktionen, mit denen wir die Nachricht bearbeiten, bevor wir sie signieren. Das bietet uns zusätzlich den Vorteil, dass nicht mehr vollständige Nachrichten durch den Signaturalgorithmus geschickt werden, sondern nur noch vergleichsweise kurze Hashwerte. Die Sicherheit eines solchen Schemas sichert das folgende Theorem.

Theorem 7.1 (Hash-Then-Sign-Paradigma). *Sei* (KeyGen, Sig, Ver) *EUF-CMA-sicher und H eine kollisionsresistente Hashfunktion. Dann ist das durch*

$$\begin{aligned} \text{KeyGen}'(1^k) &= \text{KeyGen}(1^k) \\ \text{Sig}'(sk, M) &= \text{Sig}(sk, H(M)) \\ \text{Ver}'(pk, M, \sigma) &= \text{Ver}(pk, H(M), \sigma) \end{aligned}$$

erklärte Signaturverfahren EUF-CMA-sicher.

Der Beweis dieses Theorems verläuft analog zu 6.4.1.

Die Verwendung einer kollisionsresistenten Hashfunktion ermöglicht eine Abwehr gegen die Erzeugung „unsinniger“ Signaturen, denn die errechneten „unsinnigen“ Klartexte müssen nun zusätzlich denselben Hashwert erzeugen wie die Originalnachricht.

7.3 Digital Signature Algorithm (DSA)

Aus der Anwendung des Hash-Then-Sign-Paradigmas auf das ElGamal-Signaturverfahren entsteht unter Verwendung einer kollisionsresistenten Hashfunktion H der *Digital Signature Algorithm* (DSA):

$$\begin{aligned} a &:= g^e \\ b &:= (H(M) - a \cdot x) \cdot e^{-1} \mod |\mathbb{G}| \end{aligned}$$

mit der Signatur $\sigma = (a, b)$.

Der DSA ist nach RSA der zweitwichtigste Signaturalgorithmus und wurde 1994 standardisiert. Für die Bewertung von DSA wirkt sich nachteilig aus, dass für jede neue Signatur eine frische Zufallszahl gewählt werden muss (ein guter Zufallsgenerator wird also vorausgesetzt) und dass die Verifikation einer DSA-Signatur im Vergleich mit einer RSA-Signatur mit kleinem Modulus deutlich langsamer ist.

Ob DSA EUF-CMA-sicher ist, steht noch nicht eindeutig fest.

Kapitel 8

Schlüsselaustauschprotokolle

In diesem Kapitel widmen wir uns der offenen Frage nach dem Schlüsselaustauschproblem, das insbesondere bei der Besprechung von symmetrischen Verschlüsselungs- und Signaturverfahren einige Male aufgekommen ist. Zwei Kommunikationspartner Alice und Bob können ohne vorherigen Schlüsselaustausch keine sichere Verbindung einrichten. Allerdings werden sie nicht jedes Mal die Möglichkeit haben, sich vor ihrer eigentlichen Kommunikation privat zu treffen, um einen gemeinsamen Sitzungsschlüssel auszuhandeln. Vielleicht kennen sie einander nicht einmal persönlich, auf jeden Fall aber wäre ein solches Vorgehen sichtlich nicht praktikabel.

Alice und Bob müssen also die unsichere Leitung zum Schlüsselaustausch verwenden. Den Schlüssel im Klartext darüber zu senden, würde einen Mithörer trivial in die Situation bringen, auch den verschlüsselten Teil der darauf folgenden Kommunikation mitzulesen. Der neue Sitzungsschlüssel K von Alice und Bob muss also bereits so über die Leitung gesendet werden, dass ein Lauscher nicht in der Lage ist, den Schlüssel zu rekonstruieren. Dabei sind folgende grundlegende Szenarien denkbar:

- Alice und Bob besitzen bereits einen alten Schlüssel K' aus einem früheren Austausch und möchten ein frisches K erzeugen
- es existierte eine Secret-Key-Infrastruktur mit einer Schlüsselzentrale (Alice besitzt einen Schlüssel K_A, Bob K_B und die Schlüsselzentrale beide)
- es existiert eine Public-Key-Infrastruktur (pk_A, pk_B sind öffentlich, Alice besitzt sk_A, Bob besitzt sk_B)
- Alice und Bob besitzen ein gemeinsames Passwort
- Alice und Bob besitzen keine gemeinsamen Informationen

8.1 Symmetrische Verfahren

Als Grundszenario für symmetrische Verfahren wird hier ein System mit einer Secret-Key-Infrastruktur gewählt. Das bedeutet, dass jeder Teilnehmer einen geheimen, symmetrischen Schlüssel mit der Schlüsselzentrale hat. Jeder Verbindungsaufbau mit einem anderen Teilnehmer beginnt deshalb mit einer Anfrage an die Zentrale. Da die Zentrale die Anlaufstelle für viele Teilnehmer ist, sollte die Kommunikation mit dieser Stelle möglichst minimiert werden, was die vollständige Kommunikation der beiden Teilnehmer Alice und Bob über die Zentrale ausschließt. Gleichzeitig sind jedoch die Leitungen nicht vertrauenswürdig, sodass die Kommunikation über große Strecken verschlüsselt stattfinden sollte.

8.1.1 Kerberos

Eine Lösung für dieses Szenario bietet das Protokoll *Kerberos* an, das in Abbildung 8.1 in seiner ursprünglichen Form dargestellt ist. Alice sendet dabei der Schlüsselzentrale eine Anfrage, die ihren Namen und den ihres gewünschten Gesprächspartners erhält und bekommt dafür von der Zentrale zwei Pakete zurück, von denen eines mit ihrem und eins mit Bobs Schlüssel verschlüsselt ist. Beide Pakete erhalten den gemeinsamen Sitzungsschlüssel K, sowie die Lebensdauer L des Schlüssels und einen Zeitstempel T_{KC} der Schlüsselzentrale, der Replay-Attacken erschwert. Alice entpackt das an sie adressierte Paket, erhält den Sitzungsschlüssel und leitet nach Prüfung von L und T das für Bob vorbereitete Paket weiter. Sie fügt außerdem eine mit K verschlüsselte Nachricht bei, in der sie ihre Identität und einen von ihr erstellten Zeitstempel T_A einfügt.

Bob überprüft seinerseits den Zeitstempel der Zentrale und die Lebensdauer des Sitzungsschlüssels und dechiffriert dann Alices Nachricht mit dem neuen Sitzungsschlüssel. Er kann nun sowohl den Zeitstempel überprüfen als auch, ob die Anfrage an die Schlüsselzentrale vom selben Teilnehmer stammt wie die mit dem Sitzungsschlüssel chiffrierte Nachricht. Außerdem kann er bei erfolgreicher Entschlüsselung sicher sein, dass Alice K besitzt. Er sendet nun seinerseits eine mit K verschlüsselte Nachricht an Alice, mit der er nachweist, dass er den Sitzungsschlüssel besitzt. Mit der Erhöhung des Zeitstempels kann er außerdem beweisen, dass er die korrekte Nachricht erhalten und dechiffriert hat.

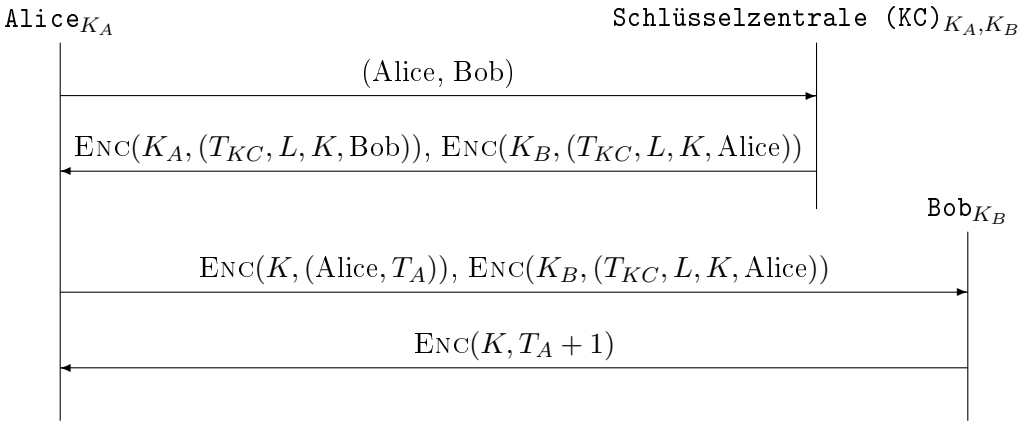


Abbildung 8.1: Ursprüngliches Schlüsselaustauschprotokoll Kerberos. T_X bezeichnet einen von X ausgestellten Zeitstempel, K den erzeugten Sitzungsschlüssel für Alice und Bob und L seine Lebensdauer.

Die verschachtelte Konstruktion von Kerberos verhindert Man-in-the-Middle-Angriffe. Die Kodierung der Absender- und Empfängernamen durch die Schlüsselzentrale ermöglicht eine Authentifizierung der Kommunikationsteilnehmer und der Einsatz von Zeitstempeln sowie die Zuordnung einer Lebensdauer zu einem Schlüssel erschwert zudem Replay-Attacken. Nichtsdestotrotz ist für das Protokoll ein aktiv sicheres Verschlüsselungsverfahren nötig. Über die Sicherheit von Kerberos lässt sich also formal keine Aussage treffen.

8.2 Asymmetrische Verfahren

Als Grundlage für die folgenden Schlüsselaustauschprotokolle nutzen wir eine Public-Key Infrastruktur. Die Schlüssel werden wie in Kapitel 5 von den Teilnehmern selbst erzeugt. Jeder hält also seinen privaten Schlüssel geheim. Die öffentlichen Schlüssel hinterliegen an einem allgemein zugänglichen Ort und sind von einer vertrauenswürdigen Stelle zertifiziert.

8.2.1 Public-Key Transport

Das einfachste Verfahren, das sich zum Schlüsselaustausch in Public-Key-Infrastruktur anbietet, nennt sich *Public-Key Transport*. Alice erzeugt einen Sitzungsschlüssel, den sie für die Kommunikation mit Bob verwenden will. Die bereits bestehende Infrastruktur wird nun dafür genutzt, den Sitzungsschlüssel mit Bobs öffentlichem Schlüssel zu chiffrieren und an Bob zu senden (siehe Abb. 8.2).

$$\text{Alice}_{sk_A} \xrightarrow{C := \text{Enc}(pk_B, K)} \text{Bob}_{sk_B}$$

Abbildung 8.2: Während des Protokolls Public-Key Transport wählt Alice einen Sitzungsschlüssel K und sendet ihn unter Ausnutzung der zur Verfügung stehenden Public-Key-Infrastruktur an Bob.

Vorausgesetzt, das verwendete Public-Key-Verfahren ist IND-CPA-sicher, kann der Angreifer C nicht von Zufall unterscheiden oder den darin enthaltenen Sitzungsschlüssel extrahieren. Public-Key Transport ermöglicht also passive Sicherheit gegenüber einem Angreifer, der C auf der Leitung mithören kann.

Allerdings bietet das Verfahren in dieser Form keine Möglichkeit zur Authentifizierung der Kommunikationsteilnehmer an. Das lässt sich durch das Hinzufügen von Signaturen wie in Abbildung 8.3 lösen. Trotzdem ist es dann noch immer möglich, einen Replay-Angriff durchzuführen und C zu einem späteren Zeitpunkt noch einmal zu senden, ohne dass Bob der Fehler sofort auffällt.

$$\text{Alice}_{sk_{\text{PKE},A}, sk_{\text{Sig},A}} \xrightarrow{(C := \text{Enc}(pk_{\text{PKE},B}, K), \sigma := (sk_{\text{Sig},A}, C))} \text{Bob}_{sk_{\text{PKE},B}, sk_{\text{Sig},B}}$$

Abbildung 8.3: Digitale Signaturen ermöglichen den Ausbau des Protokolls Public-Key Transport auf die Authentifikation der Teilnehmer.

8.2.2 Diffie-Hellman-Schlüsselaustausch

Der Diffie-Hellman-Schlüsselaustausch (1976) hat auf den ersten Blick Ähnlichkeit mit dem asymmetrischen Verschlüsselungsverfahren von ElGamal (1985). Auch hier benötigen wir eine ausreichend große, zyklische Gruppe $\mathbb{G}$ mit dem Erzeuger g. Alice und Bob wählen sich jeweils eine Zufallszahl x bzw. y und schicken g^x bzw. g^y an den jeweils anderen. Jeder von beiden ist nun in der Lage, g^{xy} zu berechnen. Das Verfahren ist noch einmal in Abbildung 8.4 dargestellt.

Unter der Diffie-Hellman-Annahme, die besagt, dass das Ziehen diskreter Logarithmen in bestimmten, zyklischen Gruppen schwierig ist, ist dieser Schlüsselaustausch passiv sicher. Allerdings ist er weiterhin anfällig für Replay- oder Man-in-the-Middle-Attacken.

8.3 Transport Layer Security (TLS)

Dieses Kapitel befasst sich mit einem Protokoll zum Schlüsselaustausch zweier einander unbekannter Kommunikationspartner. Die klassische Motivation hierfür sind Einkäufe mit der Kreditkarte. Dabei ist es nicht ausschließlich Alices Sorge, dass die Daten unterwegs abgefangen und für andere Käufe verwendet werden könnten. Ein Angreifer könnte außerdem die Kaufsumme ihres Auftrags manipulieren oder sich für den Server ausgeben, mit dem Alice kommunizieren möchte und dem sie ihre Kreditkartendaten überträgt.

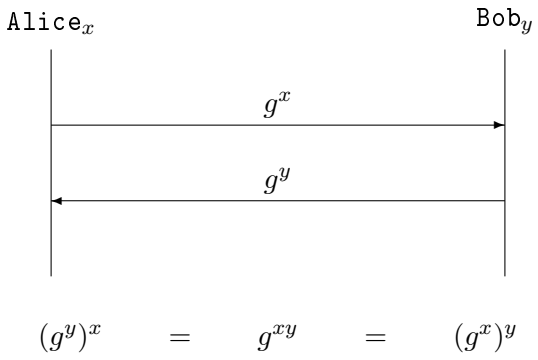


Abbildung 8.4: Beim Diffie-Hellman-Schlüsselaustausch wählen beide Kommunikationspartner einen zufälligen Wert x bzw. y und tauschen die entsprechenden Potenzen des Erzeugers g aus. Daraufhin können beide denselben Schlüssel berechnen.

Dieses Problem, das gleichzeitig den Schlüsselaustausch wie auch die Authentifikation der Kommunikationspartner umfasst, beschränkt sich allerdings nicht auf den Interneteinkauf über *http* sondern auch auf andere Anwendungsprotokolle wie *ftp* zur Übertragung von Dateien und *imap* und *smtp*, denen Alice ihre E-Mail-Passwörter anvertraut.

Kurz gefasst benötigt Alice also ein Protokoll, das die Integrität der übertragenen Daten sowie die Authentifikation des Senders bzw. Empfängers implementiert und einen sicheren Schlüsselaustausch zur Verfügung stellt. Gleichzeitig sollte es möglichst viele Anwendungsprotokolle abdecken, damit nicht jedes einzeln abgesichert werden muss.

Zu diesem Zweck wurde SSL (*Secure Socket Layer*) entwickelt und in 1999 mit einigen Änderungen als TLS (*Transport Layer Security*) standardisiert. TLS ist ein hybrides Protokoll zum Aufbau und Betrieb sicherer Kanäle über ein eigentlich unsicheres Medium, einschließlich eines Schlüsselaustauschs. Dafür wird erst ein authentifizierter asymmetrischer Schlüsselaustausch durchgeführt und danach mit diesem ausgehandelten Schlüssel symmetrisch verschlüsselt kommuniziert. Es ist sogar möglich, einen Schlüssel neu auszuhandeln, falls der Verdacht besteht, dass er kompromittiert ist. Außerdem bietet TLS eine ganze Reihe an Schlüsselaustausch- und Verschlüsselungsalgorithmen an, auf die die beiden Parteien sich einigen können.

Dadurch, dass TLS auf der Transportschicht verschlüsselt, ist es vergleichsweise einfach, Anwendungsprotokolle wie *http*, *smtp* oder *ftp* darauf anzupassen.

8.3.1 TLS-Handshake

Der für das Schlüsselaustauschproblem interessante Teil von TLS besteht aus einem Handshake, der vereinfacht in Abbildung 8.5 dargestellt ist. Dafür signalisiert der Client dem Server, dass er den Aufbau eines verschlüsselten Kanals wünscht (*client_hello*). Er liefert dem Server eine Zufallszahl sowie eine nach seiner Präferenz sortierte Liste von Algorithmen (Hashfunktionen, symmetrische Verschlüsselungsverfahren und Schlüsselaustauschprotokolle). Der Server generiert seinerseits eine Zufallszahl, wählt einen Satz Algorithmen aus der Liste des Clients aus und schickt diese zurück (*server_hello*). Im Folgenden werden die vom Server ausgewählten Verfahren verwendet.

Im nächsten Schritt schickt der Server dem Client seinen öffentlichen Schlüssel sowie das dazugehörige Zertifikat, damit der Client die Identität seines Gesprächspartners überprüfen kann. Haben sich Client und Server auf beidseitige Authentifikation geeinigt, fordert der Server außerdem das Zertifikat des Clients an. Wie genau diese Authentifizierung abläuft, wurde im vorigen Schritt durch die Auswahl der entsprechenden Algorithmen festgelegt. Der Client antwortet mit seinem Zertifikat und seinem öffentlichen Schlüssel. Um die Inte-

grität der bisherigen Kommunikation sicherzustellen, berechnet der Client außerdem den Hashwert der bisher ausgetauschten Nachrichten und signiert diesen mit seinem privaten Schlüssel. Der Server prüft das Zertifikat, die Signatur und den Hashwert.

Nun berechnet der Client eine weitere Zufallszahl, das so genannte *pre-master-secret*, und schickt es verschlüsselt mit dem zertifizierten öffentlichen Schlüssel an den Server. Beide Teilnehmer besitzen nun einen selbstgewählten Zufallswert sowie einen des Kommunikationspartners und das pre-master-secret. Aus diesen drei Zufallszahlen berechnen Client und Server nun mithilfe eines öffentlich bekannten Algorithmus den Masterkey, aus dem wiederum die für die Kommunikation verwendeten Sessionkeys abgeleitet werden.

Im letzten Teil des Handshakes signalisiert der Client, dass er nun verschlüsselt kommunizieren wird (*ChangeCipherSpec*) und damit der Handshake beendet ist (*Finished*). Der Server antwortet analog. Beide verwenden für die fortlaufende Kommunikation den vereinbarten Verschlüsselungsalgorithmus und den gemeinsamen Sessionkey.

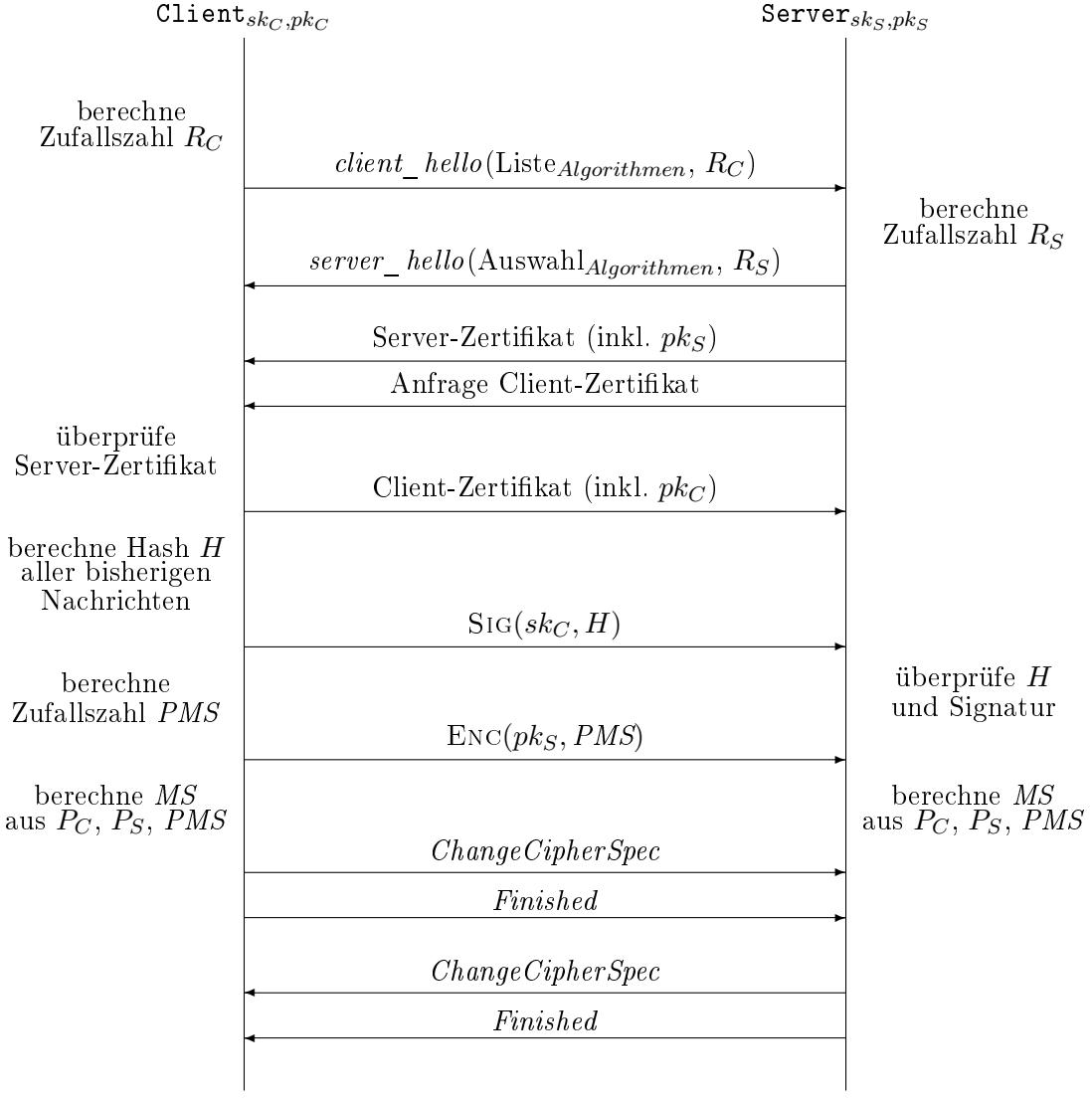


Abbildung 8.5: Vereinfachter Ablauf eines SSL/TLS-Handshakes mit beidseitiger Authentifikation.

8.3.2 Angriffe auf TLS

Unter Verwendung einer idealen Verschlüsselung, also im idealen Modell, ist TLS sicher. Auch in der Praxis gilt die Sicherheit von TLS in der neuesten Version und Verwendung der richtigen Parameter und Algorithmen als etabliert. Allerdings mussten konkrete Implementierungen als Reaktion auf veröffentlichte Angriffe immer wieder gepatcht werden und es existieren einige Angriffe auf bestimmte Varianten und Kombinationen von eingesetzten Algorithmen, von denen im Folgenden einige erklärt werden.

8.3.2.1 `ChangeCipherSpec` Drop

Dieser Angriff entstammt dem Jahr 1996 und richtet sich gegen SSL unter Version 3.0, also gegen das Protokoll *vor* seiner Standardisierung als TLS.

Beobachtung: Server und Client tauschen zu Beginn ihrer Kommunikation eine Reihe unverschlüsselter Nachrichten aus (öffentliche Schlüssel, Präferenzen für verwendete Algorithmen, Details der Authentifikation ...), die es einem Angreifer erlauben, den Status des Schlüsselaustauschs zu erkennen. Kurz vor Ende des Handshakes sendet der Client, ebenfalls im Klartext, *ChangeCipherSpec*, um auf verschlüsselte Kommunikation umzuschalten.

Angriff: Ein aktiver Angreifer unterdrückt den *ChangeCipherSpec*-Hinweis des Clients.

Konsequenz: Falls der Server sofort danach Nutzdaten sendet, werden diese nicht verschlüsselt und können vom Angreifer von der Leitung gelesen werden.

Gegenmaßnahme: Bevor die Nutzdaten gesendet werden, muss der Server auf die Bestätigung des Clients warten.

8.3.2.2 Beispielangriff auf RSA-Padding

1998 wurde ein Angriff auf das RSA-Padding bekannt, der bei entsprechender Algorithmenwahl in SSL ausgenutzt werden kann, um Einblick in den für die gemeinsame Kommunikation verwendeten Schlüssel zu erlangen.

Beobachtung: Die von SSL eingesetzte Variante von RSA verwendet beim Transport des Master Keys „naives“ Padding:

$$C = \textsc{Enc}(pk, \text{pad}(M)) = (\text{pad}(M))^e \mod N$$

Dabei kann durch homomorphe Veränderungen des Chiffrats C und ständige Überprüfung, ob C noch immer gültig ist, auf die Beschaffenheit von M geschlossen werden.

Angriff: Eine vereinfachte Darstellung des zu übertragenden Schlüssels K ist:

$$C = \text{pad}(K)^e = (\text{0x0002} \parallel \texttt{rnd} \parallel \text{0x00} \parallel K)^e \mod N$$

Klar ist, dass K vergleichsweise kurz sein und deshalb mit vielen Nullbits beginnen muss. Ziel ist es nun, möglichst viele gültige Faktoren α_i zu finden, sodass

$$M_i := \alpha_i \cdot (\text{0x0002} \parallel \texttt{rnd} \parallel \text{0x00} \parallel K)^e \mod N = \textsc{Dec}(\alpha_i^e \cdot C \mod N)$$

gültig ist. Die Gültigkeit wird festgestellt, indem die M_i zur Überprüfung an den Server weitergeleitet werden. Der Server gibt in älteren SSL-Versionen Hinweise, wenn das Padding fehlerhaft ist.

Konsequenz: Viele gültige M_i liefern ein grobes Intervall, in dem K liegt.

Gegenmaßnahme: Wähle K zufällig, wenn das Padding ungültig ist. (Zu diesem Zeitpunkt stand eigentlich bereits RSA-OAEP zur Verfügung.)

Aus diesem Angriff geht das Theorem von von Håstad und Näslund hervor, das besagt, dass jedes Bit von RSA *hardcore* ist.

Theorem 8.1 (Håstad und Näslund)**.** *Sei N, e, d wie bei RSA, $M^* \in \mathbb{Z}_N$ und $i \in \{1, \ldots, \lfloor \log_2(N) \rfloor\}$ beliebig. Sei $\mathcal{O}$ ein Orakel, das bei Eingabe C das i-te Bit von $M = C^d \mod N$ ausgibt. Dann existiert ein (von N, e, d unabhängiger) Polynomialzeit-Algorithmus, der bei Eingabe N, e, i und $C^* := (M^*)^e \mod N$ und mit $\mathcal{O}$-Zugriff M^* berechnet.*

8.3.2.3 CRIME

Dieser Angriff aus 2002 (Aktualisierung in 2012) funktioniert bei eingeschalteter Kompression.

Beobachtung: Bei eingeschalteter Kompression wird nicht mehr M sondern $comp(M)$ übertragen. TLS verwendet *DEFLATE*-Kompression. Bereits einmal aufgetretene Muster werden also nach dem Prinzip *comp*`(Fliegen fliegen) = Fliegen f(-8,6)` wiederverwendet.

Angriff: Ein Angreifer kann über die Länge des Chiffrats feststellen, ob im nachfolgenden (unbekannten) Teil des Klartextes Kompression verwendet wurde, indem er einen vorangegangenen Teil manipuliert. Die Länge des Chiffrats sinkt dann und der Angreifer weiß, dass zumindest ein Teil seines selbst eingefügten Textes im Rest des Chiffrats vorgekommen sein muss.

Konkret kann sich ein Angreifer, der in der Lage ist, einem Client einen Teil seiner Kommunikation mit dem Server zu diktieren, Stück für Stück dem von ihm gesuchten Klartext nähern. Wenn er beispielsweise den Session-Cookie des Clients (mit dem geheimen Inhalt `ABCD`) stehlen möchte, so kann er (z.B. über Schadcode) dem Client eine Eingabe (z.B. `WXYZ`) diktieren, die dieser vor dem Verschlüsseln der Nachricht hinzufügt. Er komprimiert und verschlüsselt also nicht mehr nur `ABCD` sondern `WXYZABCD`. Aus dem belauschten Chiffrat $C := \text{ENC}(K, comp(\texttt{WXYZABCD}))$ kann der Angreifer die Länge von *comp*(`WXYZABCD`) extrahieren und so den Abstand seines eingeschleusten Textstücks `WXYZ` zu dem vom Client geheim gehaltenen Cookie bestimmen.

Konsequenz: Mit mehreren Wiederholungen kann der Angreifer den Inhalt des Cookies immer weiter einschränken und ihn schließlich rekonstruieren.

Gegenmaßnahme: Keine Kompression verwenden.

8.3.2.4 Fazit

TLS ist ein historisch gewachsenes Protokoll mit hoher Relevanz. Allerdings bietet es durch die hohe Anzahl an Versionen und Einstellungsmöglichkeiten auch eine große Angriffsfläche, die häufiger durch Fixes als durch Einführung sichererer Algorithmen reduziert wird. Dazu kommt, dass von vielen Browsern ausschließlich der TLS-Standard von 1999 unterstützt wird, was zwar Schwierigkeiten in der Kompatibilität mit anderen Systemen umgeht, aber auch dazu führt, dass einige bereits bekannte Ansatzpunkte für Angriffe noch immer flächendeckend bestehen.

8.4 Weitere Protokolle

8.4.1 IPsec

IPsec (*Internet Protocol Security*) ist eine Sammlung von Standards, die zur Absicherung eines IP-Netzwerks entworfen wurden. Es setzt demnach nicht wie TLS auf der Transportschicht sondern auf der Internetschicht des TCP/IP-Protokollstapels auf. Es soll die Schutzziele Vertraulichkeit, Integrität und Authentizität in IP-Netzwerken sicherstellen. Allerdings liegt der Fokus von IPsec dabei nicht auf dem Schlüsselaustausch, der deshalb vorher getrennt stattfinden muss (aktuell durch IKE). Stattdessen bietet IPsec Maßnahmen zur Integritätssicherung der Daten an (u.A. HMAC), soll die Vortäuschung falscher IP-Adressen (IP-Spoofing) verhindern und bietet verschiedene Modi zur Verschlüsselung von IP-Paketen an.

Obwohl IPsec nicht sonderlich stark verbreitet und nicht sehr gut untersucht ist, haben sich bereits einige Angriffe herauskristallisiert, auf die hier jedoch nicht näher eingegangen wird.

8.4.2 Password Authentication Key Exchange (PAKE)

Dieses Protokoll basiert auf der Annahme, dass Alice und Bob, die miteinander kommunizieren wollen, ein gemeinsames Geheimnis `passwort` besitzen. Über dieses Passwort wollen sie einander authentifizieren und einen Schlüssel für ihre Kommunikation errechnen. Natürlich kann ein Angreifer trotz allem noch eine vollständige Suche über die möglichen Passwörter durchführen, es sollte ihm jedoch nicht möglich sein, schneller ans Ziel zu kommen.

Es handelt sich dabei eher um ein grundlegendes Prinzip als um ein feststehendes Protokoll. Bei der Konstruktion eines PAKE ist darauf zu achten, dass die simpelste Variante, das Senden von $\text{ENC}(\texttt{passwort}, K)$ keine forward-secrecy bietet. Das bedeutet, wenn im Nachhinein ein Angreifer das Passwort eines Teilnehmers knackt, ist er nicht nur zukünftig in der Lage, dessen Identität zu simulieren sondern kann außerdem sämtliche vergangene Kommunikation nachvollziehen.

Eine funktionierende Konstruktion ist *Encrypted Key Exchange* (EKE), bei dem zunächst $\text{ENC}(\texttt{passwort}, pk)$ gesendet und infolgedessen asymmetrisch kommuniziert wird. Bei *Simple Password Exponential Key Exchange* wird ein Diffie-Hellman-Schlüsselaustausch auf der Basis von einem nur den Teilnehmern bekannten $g = H(\texttt{passwort})^2$ durchgeführt. Der beweisbare PAKE von Goldreich-Lindell nutzt Zero-Knowledge, um die Teilnehmer zu authentifizieren, ohne das dafür nötige Geheimnis aufzudecken.

PAKE wird z.B. als Basis für EAP (*Extensible Authentication Protocol*) in WPA verwendet und ist formal modellierbar und seine Sicherheit unter bestimmten theoretischen Annahmen beweisbar.

Kapitel 9

Identifikationsprotokolle

Nachdem wir jetzt Authentifikation von Nachrichten und den authentifizierten Austausch von Schlüsseln betrachtet haben, befasst sich dieses Kapitel mit der asymmetrischen Identifikation von Kommunikationsteilnehmern. Das bedeutet, Alice ist im Besitz eines geheimen Schlüssels sk und Bob, der den dazugehörigen öffentlichen Schlüssel pk kennt, möchte sicher sein, dass er mit einer Instanz redet, die in Besitz von sk ist. Üblicherweise geht es bei dieser Prüfung um den Nachweis einer Identität, der an bestimmte (Zugangs-)Rechte gekoppelt ist.

Da Alice im Folgenden *beweisen* muss, dass sie den geheimen Schlüssel besitzt, und Bob ihre Identität *überprüft*, heißen die beiden für den Rest dieses Kapitels `Prover` und `Verifier`.

Der einfachste Weg, dem Verifier zu beweisen, dass der Prover das Geheimnis sk kennt, ist es, ihm den Schlüssel einfach direkt zu schicken. Der Verifier kann dann die Zugehörigkeit zu pk feststellen und sicher sein, dass der Prover das Geheimnis kennt. Allerdings wird bei diesem Vorgehen sk allgemein bekannt und garantiert nach der ersten Verwendung keine Zuordnung mehr zu einer bestimmten Identität.

Die Protokollanforderungen steigen also darauf, dass der Verifier sicher sein kann, dass der Prover das Geheimnis kennt, der Verifier selbst jedoch sk nicht lernt.

Ein zweiter Versuch umfasst die bereits entwickelten Signaturschemata. Der Prover schickt $\sigma := \text{SIG}(sk_A, \text{„ich bin's, P“})$ an den Verifier. $\text{VER}(pk_A, \text{„ich bin's, P“}, \sigma)$ liefert dem Verifier die Gültigkeit der entsprechenden Signatur und damit die Identität des Absenders. Um die Signatur zu fälschen, müsste ein Angreifer also das dahinterstehende Signaturverfahren brechen. Allerdings kann er die Signatur σ mit dieser trivialen Nachricht einfach wiederverwenden und sich so entweder als Man-in-the-Middle oder mithilfe eine Replay-Attacke Ps Identität zunutze machen.

Aus den ersten beiden Versuchen geht hervor, dass wir ein interaktives Protokoll wie in Abbildung 9.1 benötigen, um den geheimen Schlüssel gleichzeitig zu verbergen und den Besitz dieses Geheimnisses zu beweisen.[1]

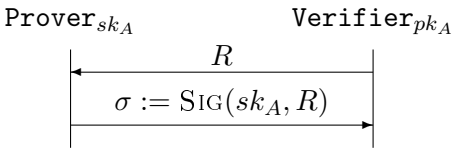


Abbildung 9.1: Interaktives Protokoll, in dem der Verifier dem Prover eine Zufallszahl R gibt, um dessen Identität durch eine Signatur sicherzustellen.

[1] In der Praxis mag es sinnvoll sein, nicht nur die Zufallszahl R zu signieren, sondern dieser noch das aktuelle Datum und die aktuelle Uhrzeit hinzuzufügen. So kann, selbst wenn der Verifier irgendwann zum zweiten Mal die selbe Zufallszahl ausgibt, eine gerade erzeugte von einer alten Signatur unterschieden werden.

9.1 Sicherheitsmodell

Ein Public-Key-Identifikationsprotokoll ist definiert durch das Tupel $(\text{GEN}, \text{P}, \text{V})$ von PPT-Algorithmen. Dabei gibt GEN wie gewohnt bei Eingabe eines Sicherheitsparameters 1^k das Schlüsselpaar (pk, sk) aus. Der Prover P und der Verifier V sind zustandsbehaftet und interagieren während des Identitätsnachweises miteinander.

1. V erhält den öffentlichen Schlüssel pk_P als Eingabe und gibt out_V aus
2. P erhält Vs Ausgabe out_V und den privaten Schlüssel sk_V und gibt out_P aus
3. V erhält Ps Ausgabe out_P und gibt out_V aus
4. ist $\text{out}_V \in \{0, 1\}$ beende die Interaktion, ansonsten springe zurück zu Schritt 2

Der Verifier erzeugt also eine Ausgabe, mit deren Hilfe P beweisen muss, dass er das Geheimis sk kennt. P liefert auf Basis des Geheimnisses und der Ausgabe von V seinerseits eine Ausgabe und gibt diese an V weiter. V prüft das Ergebnis und entscheidet, ob die Prüfung erfolgreich abgeschlossen wurde. Falls ja, gibt er 1 aus, falls nein 0.

Das Verfahren muss *korrekt* sein, also muss schließlich gelten:

$$\forall (pk, sk) \leftarrow \text{GEN}(1^k) : V(\text{out}_P) \rightarrow 1$$

$\langle \text{P}(sk), \text{V}(pk) \rangle$ bezeichnet im Folgenden das Transkript der Interaktion zwischen Prover und Verifier.

Einem Angreifer $\mathcal{A}$ darf es nun intuitiv nicht möglich sein, gegenüber einem Verifier die Identität eines anderen anzunehmen. Um das überprüfen zu können, führen wir ein neues Spiel ein. Zunächst erzeugt das Spiel i (pk, sk)-Paare und ordnet die privaten Schlüssel i Provern zu.

1. $\mathcal{A}$ darf nun mit beliebig vielen dieser gültigen Prover interagieren. Dabei nimmt er die Rolle des Verifiers ein und hat demnach Zugriff auf die passenden öffentlichen Schlüssel pk_i, während die gültigen Prover seine Anfragen mit ihren privaten Schlüsseln sk_i beantworten.
2. $\mathcal{A}$ wählt sich nun einen der pk_{i^*} aus und stellt sich damit als Prover dem *echten* Verifier mit der Eingabe pk_{i^*}.
3. $\mathcal{A}$ gewinnt, wenn der Verifier als Ergebnis schließlich 1 ausgibt.

Wir nennen ein Public-Key-Identifikationsprotokoll $(\text{GEN}, \text{P}, \text{V})$ sicher, wenn kein PPT-Angreifer $\mathcal{A}$ das oben genannte Spiel häufiger als vernachlässigbar oft gewinnt.

Allerdings verhindert das oben genannte Spiel keinen Man-in-the-Middle-Angriff, in dem A die Ausgaben einfach weiterreicht.

9.2 Protokolle

Unter diesem Aspekt können wir nun unseren Vorschlag aus Abbildung 9.1 aufgreifen und untersuchen. Dieser Ansatz basiert auf einem Signaturverfahren. Seine Sicherheit ist demnach von der Sicherheit des verwendeten Signaturalgorithmus abhängig.

Theorem 9.1. *Ist das verwendete Signaturverfahren EUF-CMA-sicher, so ist das in Abbildung 9.1 gezeigte PK-Identifikationsprotokoll* $(\text{GEN}, \text{P}, \text{V})$ *sicher.*

Beweisidee. *Angenommen, es gibt einen Angreifer A, der das PK-Identifikationsprotokoll bricht. Dann ist er in der Lage, nicht-vernachlässigbar oft aus dem öffentlichen Schlüssel pk_{i^*} und einer vom Verifier ausgewählten Zufallszahl R eine Signatur $\sigma := \textsc{Sig}(sk_{i^*}, R)$ zu berechnen.*

Aus A kann nun ein Angreifer B konstruiert werden, der die Ergebnisse von A nutzt, um das EUF-CMA-sichere Signaturverfahren zu brechen.

Ein weiterer Ansatz für ein funktionierendes Identifikationsprotokoll auf Public-Key-Basis ist in Abbildung 9.2 dargestellt. Hier wird R vor der Übertragung über die Leitung vom Verifier mit pk_{i^*} verschlüsselt, sodass die Kenntnis von sk_{i^*} durch einen Entschlüsselungsvorgang überprüft wird.

Es ist hierbei darauf zu achten, dass das Schlüsselpaar, das für dieses Identifikationsprotokoll verwendet wird, nicht auch zum Verschlüsseln gebraucht werden sollte. Ansonsten kann ein Angreifer in der Rolle des Verifiers die Entschlüsselung von ihm bekannten Chiffraten herbeiführen und somit jedes beliebige Chiffrat entschlüsseln lassen.

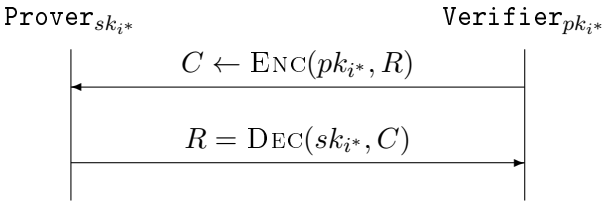


Abbildung 9.2: Dieses Identifikationsprotokoll profitiert von der Sicherheit des verwendeten Public-Key-Verschlüsselungsverfahrens.

Theorem 9.2. *Ist das in Abbildung 9.2 verwendete Verschlüsselungsverfahren IND-CCA-sicher, so ist das darauf basierende PK-Identifikationsprotokoll* $(\textsc{Gen}, \mathrm{P}, \mathrm{V})$ *sicher.*

Beweisidee. *Der Beweis dafür läuft analog zum obigen. Aus einem Angreifer A, der das Identifikationsprotokoll nicht vernachlässigbar oft bricht, wird ein Angreifer B konstruiert, der das IND-CCA-sichere Verschlüsselungsverfahren bricht.*

Identifikationsprotokolle wie die in Abbildungen 9.1 und 9.2 gezeigten heißen auch „Challenge-Response-Verfahren“, denn der Verifier stellt dem Prover eine Aufgabe (oder Herausforderung, die „Challenge“), die nur der echte Prover lösen kann. In dem Protokoll aus Abbildung 9.1 ist diese Aufgabe die Erstellung einer Signatur für einen Zufallsstring R; in Abbildung 9.2 ist diese Aufgabe die Entschlüsselung eines zufälligen Chiffrats $C = \textsc{Enc}(pk_{i^*}, R)$. Die Lösung des Provers wird daher auch als die Antwort, oder „Response“ bezeichnet.

Kapitel 10

Zero-Knowledge

Im vorigen Kapitel wurden zwei Voraussetzungen entwickelt, die für Identifikationsprotokolle wünschenswert sind.

- Verifier V lernt sk_P nicht
- Prover P beweist, dass er sk_P kennt

Diese Eigenschaften konnten wir im vorigen Kapitel nur teilweise erfüllen. Beispielsweise ist es dem Verifier im Protokoll aus Abbildung 9.1 möglich, Teilinformationen über sk_P zu erlangen. Vielleicht kennt P außerdem nur eine Art Ersatzschlüssel und nicht den echten sk_P. All das reicht für eine Identifikation aus, kann jedoch dazu führen, dass der geheime Schlüssel mit der Zeit korrumpiert wird.

10.1 Zero-Knowledge-Eigenschaften

Wir wollen nicht nur erreichen, dass V sk_P nicht lernt, sondern verlangen strikter, dass V *nichts* über den geheimen Schlüssel von P lernt. Wir müssen dabei allerdings berücksichtigen, dass er in Form von pk_P bereits eine mit sk_P verknüpfte Information besitzt (z.B. mit $sk_P = x$ und $pk_P = g^x$). Wir verlangen also, dass V während der Kommunikation mit P nichts über sk_P lernt, was er nicht schon aus pk_P berechnen kann.

Wir modellieren dafür zu dem Verifier V einen Simulator $\mathcal{S}$, der dieselbe Ausgabe erzeugt wie V, jedoch ohne mit P kommuniziert zu haben.

Dazu benötigen wir den folgenden Hilfssatz.

Definition 10.1 (Ununterscheidbarkeit)**.** Zwei (möglicherweise vom Sicherheitsparameter $k \in \mathbb{N}$ abhängige) Verteilungen X, Y sind ununterscheidbar (geschrieben $X \overset{c}{\approx} Y$), wenn für alle PPT-Algorithmen $\mathcal{A}$ die Differenz

$$\Pr[\mathcal{A}(1^k, x) = 1 | x \leftarrow X] - \Pr[\mathcal{A}(1^k, x) = 1 | y \leftarrow Y]$$

vernachlässigbar in k ist.

Intuitiv sind also Elemente aus X nicht effizient von Elementen aus Y unterscheidbar.

Definition 10.2 (Zero-Knowledge)**.** Ein PK-Identifikationsprotokoll $(\textsc{Gen}, \mathrm{P}, \mathrm{V})$ ist Zero-Knowledge (ZK), falls für jeden PPT-Algorithmus $\mathcal{A}$ (der Angreifer) ein PPT-Algorithmus $\mathcal{S}$ (der Simulator) existiert, so dass die folgenden Verteilungen ununterscheidbar sind (wobei $(pk, sk) \leftarrow \textsc{Gen}(1^k)$:

$$\langle \mathrm{P}(sk), \mathcal{A}(1^k, pk) \rangle \qquad \text{und} \qquad (\text{Ausgabe von}) \ \mathcal{S}(1^k, pk)$$

$\mathcal{S}$ simuliert also die Interaktion zwischen P und $\mathcal{A}$. Da $\mathcal{S}$ ein PPT-Algorithmus ist, dessen einzige Informationsquelle über sk der gegebene Public Key pk ist, kann die Ausgabe von $\mathcal{S}$ nur Informationen enthalten, die bereits mit geringem Aufwand aus pk berechnet werden können. Ist die Zero-Knowledge Eigenschaft erfüllt, dann ist ein solches simuliertes Transkript von einem echten Transkript $\langle \mathrm{P}(sk), \mathcal{A}(1^k, pk)\rangle$ nicht unterscheidbar, also kann auch das echte Transkript nicht mehr Informationen über sk enthalten als bereits in pk enthalten sind.

Wir untersuchen nun als Beispiel, ob das oben vorgestellte Identifikationsprotokoll (vgl. Abbildung 9.1) ein Zero-Knowledge-Protokoll ist. Im ersten Schritt des Protokolls sendet der Verifier V einen Zufallsstring R an den Prover P. Im zweiten Schritt sendet P eine Signatur der Nachricht R an V zurück.

Um ein glaubwürdiges simuliertes Transkript zu erstellen müsste der Simulator also einen Zufallsstring R und eine gültige Signatur $\sigma := \textsc{Sig}(sk, R)$ erzeugen, um diese in das simulierte Transkript einzubetten. Das würde aber einen Bruch des Signaturverfahrens erfordern, da $\mathcal{S}$ nur über pk verfügt. Das Protokoll ist also *nicht* Zero-Knowledge.

Bevor wir jedoch ein Zero-Knowledge-Identifikationsprotokoll vorstellen, benötigen wir noch *Commitments* als Hilfskonstruktion.

10.2 Commitments

Ein Commitment-Schema besteht aus einem PPT-Algorithmus Com. Dieser erhält eine Nachricht M als Eingabe. Außerdem schreiben wir den von Com verwendeten Zufall R explizit hinzu. Eine Ausführung von Com wird also als $\textsc{Com}(M; R)$ geschrieben. Die Ausgabe von Com wird als *Commitment* bezeichnet. Dieses Commitment muss folgende Eigenschaften erfüllen:

Hiding $\textsc{Com}(M; R)$ verrät zunächst keinerlei Information über M.

Binding $\textsc{Com}(M; R)$ legt den Ersteller des Commitments auf M fest, d.h. der Ersteller kann später nicht glaubhaft behaupten, dass $M' \neq M$ zur Erstellung des Commitments verwendet wurde.

Ein klassisches Anwendungsbeispiel für Commitment-Schemas sind Sportwetten, z.B. auf Pferderennen. Hier möchte Alice eine Wette auf den Ausgang eines Rennens bei der Bank abgeben. Alice befürchtet jedoch, dass die Bank den Ausgang des Rennens manipulieren könnte, wenn die Bank Alices Wette erfahren würde. Deshalb möchte Alice ihren Wettschein nicht vor dem Ereignis der Bank übergeben. Andererseits muss die Bank darauf bestehen, dass Alice die Wette vor dem Wettstreit abgibt, denn sonst könnte Alice betrügen, indem sie den Wettschein erst nach Ende des Sportereignisses ausfüllt.

Commitment-Schemas bieten eine einfache Lösung für dieses Dilemma: Alice setzt ihre Wette M und legt sich mittels des Commitment-Schemas darauf fest. Sie berechnet also ein Commitment $\textsc{Com}(M; R)$, und händigt dieses der Bank aus. Wegen der Hiding-Eigenschaft kann die Bank Alices Wette nicht in Erfahrung bringen und deshalb das Rennen nicht gezielt manipulieren. Alice ist also vor Manipulation zu ihren Ungunsten geschützt. Sobald das Rennen abgeschlossen ist, deckt Alice ihr Commitment auf. Nun erfährt die Bank was Alice gewettet hat und kann ggf. den Gewinn auszahlen. Die Binding-Eigenschaft des Commitments garantiert der Bank, dass Alice nur ihre echte, vorher gesetzte Wette M aufdecken kann. Damit ist ausgeschlossen, dass Alice die Bank betrügen kann.

Definition 10.3 (Hiding)**.** Ein Commitmentschema Com ist *hiding*, wenn für beliebige $M \neq M' \in \{0,1\}^*$ und unabhängig zufälliges R $\textsc{Com}(M; R) \overset{c}{\approx} \textsc{Com}(M'; R)$ ist.

Definition 10.4 (Binding)**.** Ein Commitmentschema Com ist *binding*, wenn für jeden PPT-Angreifer $\mathcal{A}$, der M, R, M', R' ausgibt, $\Pr[\textsc{Com}(M; R) = \textsc{Com}(M'; R')$ und $M \neq M']$

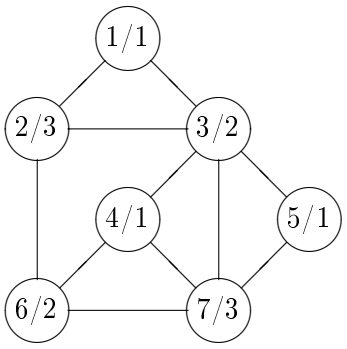


Abbildung 10.1: Ein dreifärbbarer Graph. Für jeden Knoten sind Nummer (links) und Farbe (rechts) angegeben. Da kein Knoten mit einem gleichgefärbten Knoten direkt benachbart ist, ist die hier gezeigte Dreifärbung gültig.

vernachlässigbar im Sicherheitsparameter k ist.

In der Literatur existieren verschiedene Konstruktionen für solche Commitment-Verfahren. Ein bekanntes Beispiel sind Pedersen-Commitments [10].

10.3 Beispielprotokoll: Graphendreifärbbarkeit

Als Beispiel für ein Zero-Knowledge-Identifikationsprotokoll geben wir ein Protokoll an, das auf dem Problem der Dreifärbbarkeit von Graphen beruht. Wir rekapitulieren zunächst dieses Problem.

Definition 10.5. Gegeben sei ein Graph $G = (V, E)$ mit Knotenmenge V und Kantenmenge $E \subseteq V^2$. Eine Dreifärbung von G ist eine Abbildung $\phi : V \to \{1, 2, 3\}$, die jedem Knoten $v \in V$ eine „Farbe“ $\phi(V) \in \{1, 2, 3\}$ zuordnet[1], wobei jede Kante $(i, j) \in E$ zwei verschiedenfarbige Knoten i, j verbindet. Es muss also für jede Kante (i, j) gelten, dass $\phi(i) \neq \phi(j)$. Ein Graph G heißt dreifärbbar, wenn eine Dreifärbung für G existiert.

Abbildung 10.1 zeigt beispielhaft einen Graphen zusammen mit einer Dreifärbung.

Das Entscheidungsproblem, ob ein gegebener Graph dreifärbbar ist, ist NP-vollständig [11].

Zwar lässt sich für bestimmte Klassen von Graphen G leicht entscheiden, ob sie dreifärbbar sind oder nicht.[2] Es gibt aber auch Wahrscheinlichkeitsverteilungen von Graphen, für die es im Mittel sehr schwierig ist, die Dreifärbbarkeit zu entscheiden. Die Details sind hier für uns nicht weiter interessant.

Wir betrachten nun das folgende Protokoll. Zuvor wird der Algorithmus Gen ausgeführt, der einen zufälligen Graphen G zusammen mit einer Dreifärbung ϕ erzeugt. Der öffentliche Schlüssel ist $pk = G$, der geheime Schlüssel $sk = (G, \phi)$.

1. Der Prover P wählt eine zufällige Permutation π der Farben $\{1, 2, 3\}$. Mit dieser Permutation werden im nächsten Schritt die Farben von G vertauscht.

[1] Man kann grundsätzlich drei beliebige Farben für die Definition wählen, z.B. „rot“, „grün“ und „blau“; „cyan“, „magenta“ und gelb; oder auch „pastell“, „purpur“ und „pink“. Die Definition bleibt dabei im Wesentlichen die Gleiche. Um sich um eine konkrete, willkürliche Wahl dieser drei Farben zu drücken verwendet man schlicht 1,2 und 3.

[2] Graphen mit maximalem Knotengrad 2 sind z.B. immer dreifärbbar. Graphen, die eine 4-Clique enthalten (also 4 Knoten, die untereinander alle direkt verbunden sind), sind niemals dreifärbbar.

2. Mit dieser Permutation werden nun die Farben von G vertauscht. Anschließend berechnet P für jeden Knoten i das Commitment auf die (neue) Farbe $com_i = \textsc{Com}(\pi(\phi(i)); R_i)$ und sendet alle Commitments an V, d.h. P legt sich gegenüber V auf den Graphen mit vertauschten Farben fest.

3. V wählt eine zufällige Kante (i, j) und sendet diese an P.

4. P öffnet die Commitments com_i und com_j gegenüber V.

5. V überprüft, ob die Commitments korrekt geöffnet wurden und ob $\pi(\phi(i)) \neq \pi(\phi(j))$. Wenn beides der Fall ist akzeptiert V. Wenn eines nicht der Fall ist, lehnt V ab.

Wenn P ehrlich ist, (also tatsächlich eine Dreifärbung von G kennt), dann kann P V immer überzeugen. Das bisherigen Protokoll ist aber noch nicht sicher, denn ein Angreifer der keine Dreifärbung von G kennt, könnte einfach eine zufällige Abbildung $\phi' : V \to \{1, 2, 3\}$ erstellen. Mit dieser zufälligen Färbung, die im Allgemeinen keine gültige Dreifärbung ist, führt der Angreifer das Protokoll regulär durch, d.h. er wählt eine zufällige Permutation π und berechnet die Commitments wie oben angegeben. Für eine zufällige, vom Verifier gewählte Kante (i, j) gilt dann mit Wahrscheinlichkeit $2/3$ $\phi'(i) \neq \phi'(j)$, also auch $\pi(\phi'(i)) \neq \pi(\phi'(j))$. Der Angreifer kann den Verifier also mit einer Wahrscheinlichkeit von $2/3$ überzeugen.

Diese Schwäche kann man ausräumen, indem man das Protokoll mehrfach ausführt. Der Verifier akzeptiert P nur dann, wenn P in *allen* Durchläufen erfolgreich ist. Scheitert P in auch nur einer einzigen Runde, lehnt V ab. Für das Protokoll mit mehrfacher Wiederholung kann man die Sicherheit auch formal zeigen. Dazu muss man aber natürlich *alle* möglichen Angriffsstrategien betrachten, nicht nur die oben gezeigt Rate-Strategie.

Wir möchten uns hier jedoch lieber mit der Zero-Knowledge-Eigenschaft befassen. Zunächst wollen wir dazu an einem Beispiel zeigen, dass der Verifier im obigen Protokoll keine Information über die geheime Dreifärbung ϕ von G gewinnt. Im Anschluss werden wir die Zero-Knowledge-Eigenschaft nachweisen.

Beispiel 10.6. *Wir betrachten die ersten zwei Runden eines Protokollablaufs zwischen Verifier und Prover. Beide Parteien kennen den öffentlichen Schlüssel, einen Graphen $G = (V, E)$. Der Prover kennt den geheimen Schlüssel, eine Dreifärbung ϕ. Es seien $a, b, c \in V$ drei Knoten des Graphen, die mit $\phi(a) = 1$, $\phi(b) = 2$ und $\phi(c) = 3$ gefärbt sind.*

Zu Beginn der ersten Runde wählt P die Permutation π_1 zufällig, hier $\pi_1 = (2, 3, 1)$, also $\pi_1(1) = 2$, $\pi_1(2) = 3$ und $\pi_1(3) = 1$. Anschließend erzeugt P Commitments auf $\pi_1(\phi(i))$ für alle $i \in V$.

Der Verifier wählt eine Kante, hier beispielsweise (a, b), und sendet diese an den Prover. Der Prover öffnet daraufhin die Commitments für die Knoten a und b, und so lernt der Verifier $\pi_1(\phi(a)) = 2$ und $\pi_1(\phi(b)) = 3$.

In der nächsten Runde wählt P eine neue, zufällige Permutation π_2, unabhängig von π_1. Hier sei $\pi_2 = (2, 1, 3)$. Er erzeugt wieder Commitments $\pi_2(\phi(i))$ für alle $i \in V$, und sendet diese an den Verifier.

Dieser wählt nun seinerseits eine neue, unabhängig zufällige Kante. Dabei tritt zufällig a erneut auf: Die gewählte Kante sei (a, c).

P öffnet also die Commitments für a und c. Der Verifier erfährt nun, dass $\pi_2(\phi(a)) = 2$ und $\pi_2(\phi(c)) = 3$ gelten. Da hier $\pi_2(\phi(a)) = \pi_1(\phi(a))$ gilt, wurde die Farbe $\phi(a)$ offensichtlich in beiden Runden auf die selbe Farbe, nämlich 2, abgebildet. Tatsächlich ist sogar $\pi_1(\phi(b)) = \pi_2(\phi(c))$. Dadurch erfährt der Verifier jedoch nichts darüber, ob b und c gleich gefärbt sind, denn es könnte sowohl sein dass

- *b und c gleich gefärbt sind und P zufällig zwei mal hintereinander die selben Permutation gewählt hat (dann gälte also $\pi_1 = \pi_2$), als auch dass*

- *b und c unterschiedlich gefärbt sind und nur die Permutationen π_1 und π_2 unterschiedlich sind.*

Wenn π_1 und π_2 unabhängig voneinander gleichverteilt gezogen werden, sind beide Fälle gleich wahrscheinlich. Deshalb lernt der Verifier hier nichts *über die Färbung der Knoten a, b und c, und ganz allgemein auch nichts über die vollständige Färbung ϕ von G.*

Nach diesem Beispiel zeigen wir nun die Zero-Knowledge-Eigenschaft des Protokolls. Hierfür müssen wir einen Simulator $\mathcal{S}$ angeben, dessen Ausgabe ununterscheidbar von echten Transskripten $\langle \mathrm{P}(sk), \mathcal{A}(1^k, pk)\rangle$ ist.

Um dies zu erreichen, simuliert $\mathcal{S}$ intern eine Interaktion mit $\mathcal{A}$. $\mathcal{S}$ setzt sich dabei selbst in die Rolle des Provers und setzt $\mathcal{A}$ in die Rolle des Verifiers. $\mathcal{S}$ zeichnet dabei alle Ausgaben von $\mathcal{A}$ und sich selbst auf, da diese das auszugebende Transkript bilden. $\mathcal{S}$ verfährt wie folgt:

1. $\mathcal{S}$ speichert den Zustand von $\mathcal{A}$.
2. $\mathcal{S}$ wählt zufällige Farben c_i für jeden Knoten i und gibt die entsprechenden Commitments gegenüber dem Verifier, also $\mathcal{A}$, ab.
3. Anschließend simuliert $\mathcal{S}$ die weitere Ausführung von $\mathcal{A}$, bis $\mathcal{A}$ eine Kante (i, j) ausgibt.
4. Ist $c_i \neq c_j$, dann deckt $\mathcal{S}$ die entsprechenden Commitments für c_i und c_j auf und führt das Protokoll regulär weiter aus.

   Ist jedoch stattdessen $c_i = c_j$, dann kann $\mathcal{S}$ nicht einfach die Commitments öffnen, denn dann wäre das Transkript offensichtlich von echten Transkripten unterscheidbar: In echten Transkripten werden beim Öffnen der Commitments immer verschiedene Farben gezeigt, in diesem falschen Transkript werden jedoch gleiche Farben aufgedeckt.

   Um dennoch ein echt wirkendes Transkript erstellen zu können, setzt $\mathcal{S}$ den Algorithmus $\mathcal{A}$ auf den in Schritt 1 gespeicherten Zustand zurück, ändert eine der Farben c_i oder c_j, gibt dem zurückgesetzten Algorithmus $\mathcal{A}$ nun die entsprechenden neuen Commitments und führt diesen wieder aus.

   Nun wird $\mathcal{A}$ wieder (i, j) ausgeben, doch diesmal wird $c_i \neq c_j$ gelten. $\mathcal{S}$ kann die Commitments also bedenkenlos öffnen und $\mathcal{A}$ zu Ende ausführen.
5. Sobald $\mathcal{A}$ terminiert hat gibt $\mathcal{S}$ das Transkript der Interaktion von sich selbst und $\mathcal{A}$ aus.

Wir vergleichen nun ein so entstandenes Transkript mit echten Transkripten $\langle \mathrm{P}(sk), \mathcal{A}(1^k, pk)\rangle$.

Ein echtes Transkript besteht aus allen Commitments com_i, die eine gültige Dreifärbung des Graphen enthalten, der Wahl (i, j) des Angreifers $\mathcal{A}$, sowie der Information zur Öffnung der Commitments com_i und com_j.

Das vom Simulator $\mathcal{S}$ ausgegebene Transkript enthält ebenfalls alle Commitments com_i, die Wahl des Angreifers (i, j) sowie der Information zur Öffnung der Commitments com_i und com_j. Durch die Konstruktion des Simulators werden dabei immer verschiedene Farben aufgedeckt, d.h. in diesem Schritt ist keine Unterscheidung möglich.

Ein Unterschied tritt jedoch bei den Commitments auf: Im echten Protokoll enthalten diese Commitments eine gültige Dreifärbung des Graphen. Im simulierten Transkript enthalten diese eine zufällige Färbung des Graphen, und dies ist im Allgemeinen keine gültige Dreifärbung. Glücklicherweise lässt sich jedoch wegen der Hiding-Eigenschaft der Commitments nicht effizient feststellen, ob diese eine gültige Dreifärbung oder eine zufällige Färbung des Graphen beinhalten.

Deshalb sind die so entstehenden Transkripte gemäß Definition 10.1 ununterscheidbar, und die Zero-Knowledge-Eigenschaft (Definition 10.2) erfüllt.

Mit dem hier gezeigten Protokoll kann man übrigens theoretisch beliebige NP-Aussagen beweisen. Um für einen beliebigen Bitstring b eine bestimmte Eigenschaft (die als Sprache $L \subset \{0,1\}^*$ aufgefasst werden kann) nachzuweisen, transformiert man das Problem $b \stackrel{?}{\in} L$ in eine Instanz I des Graphdreifärbbarkeitsproblems L_{G3C}. (Dies ist möglich, weil das Graphdreifärbbarkeitsproblem NP-vollständig ist.) Dann kann man mit obigem Protokoll nachweisen, dass der so entstehende Graph I dreifärbbar ist (also $I \in L_{G3C}$), also $b \in L$ ist. Der Verifier kann dabei wegen der Zero-Knowledge-Eigenschaft keine Information über b gewinnen, außer das $b \in L$ ist.

Solche Beweise sind zwar extrem ineffizient, aber theoretisch möglich. Z.B. kann man für zwei Chiffrate $C_1 = \textsc{Enc}(pk, M)$ und $C_2 = \textsc{Enc}(pk, M)$ so nachweisen, dass beide Chiffrate die selbe Nachricht enthalten, ohne die Nachricht preiszugeben. Dies wird z.B. bei kryptographischen Wahlverfahren benötigt. Dort werden jedoch effizientere Verfahren verwendet, die aber dann speziell auf ein Verschlüsselungsverfahren zugeschnitten sind.

10.4 Proof-of-Knowledge-Eigenschaft

Nun haben wir gezeigt, dass im vorherigen Protokoll der Verifier *nichts* über sk_P lernt, was er nicht bereits aus pk_P selbst hätte berechnen können. Nun wenden wir uns der zweiten wünschenswerten Eigenschaft von Identifikationsprotokollen zu: P soll beweisen, dass er tatsächlich sk_P kennt.

Wir definieren dazu die Proof-of-Knowledge-Eigenschaft:

Definition 10.7. (Proof of Knowledge) Ein Identifikationsprotokoll $(\textsc{Gen}, P, V)$ ist ein Proof of Knowledge, wenn ein PPT-Algorithmus $\mathcal{E}$ (der „Extraktor“) existiert, der bei Zugriff auf einen beliebigen erfolgreichen Prover P einen [3] geheimen Schlüssel sk zu pk extrahiert.

Diese Definition scheint zunächst im Widerspruch zur Zero-Knowledge-Eigenschaft zu stehen. Schließlich forderte die Zero-Knowledge-Eigenschaft doch, dass ein Verifier nichts über sk_P lernt, während die Proof-of-Knowledge-Eigenschaft fordert, dass man einen vollständigen geheimen Schlüssel aus P extrahieren kann. Tatsächlich sind diese Eigenschaften jedoch nicht widersprüchlich, da wir dem Extraktor $\mathcal{E}$ weitergehende Zugriffsmöglichkeiten auf P zugestehen als einem Verifier: Ein Verifier ist nämlich auf die Interaktion mit P beschränkt, während wir dem Extraktor $\mathcal{E}$ auch gestatten P zurückzuspulen.

Für unser Graphdreifärbbarkeits-Identifikationsprotokoll können wir diese Eigenschaft auch tatsächlich nachweisen.

Theorem 10.8. *Das Graphdreifärbbarkeits-Identifikationsprotokoll ist ein Proof of Knowledge.*

Beweis. *Wir geben einen Extraktor $\mathcal{E}$ an, der einen gültigen sk extrahiert. Dazu sei P ein beliebiger erfolgreicher Prover.*

1. *Der Extraktor simuliert zunächst einen ehrlichen Verifier V. Er führt P solange aus, bis P die zufällige Farbpermutation π gewählt und Commitments $com_i = \textsc{Com}(\pi(\phi(i)); R)$ auf die Farben jedes Knotens abgegeben hat.*

2. *Nun speichert $\mathcal{E}$ den Zustand von P.*

[3] Im Allgemeinen kann es mehrere gültige geheime Schlüssel zu einem Public-Key geben. In unserem Beispielprotokoll auf Basis der Graphdreifärbbarkeit ist z.B. jede Permutation einer gültigen Dreifärbung selbst eine gültige Dreifärbung. Es kann darüber hinaus aber auch vorkommen, dass ein Graph zwei verschiedene Dreifärbungen hat, die nicht durch Permutation auseinander hervorgehen.

3. *$\mathcal{E}$ lässt den von ihm simulierten Verifier nun die erste Kante (i_1, j_1) des Graphen G wählen und diese an P übermitteln.*

4. *P muss daraufhin die Commitments com_{i_1} und com_{j_1} aufdecken. Der Extraktor lernt also die (vertauschten) Farben der Knoten i_1 und j_1, nämlich $\pi(\phi(i_1))$ und $\pi(\phi(j_1))$.*

5. *Anstatt des Protokoll weiter auszuführen setzt $\mathcal{E}$ nun P auf den in Schritt 2 zurück. Zu diesem Zeitpunkt hatte P bereits alle Commitments abgegeben und erwartet vom Verifier eine Aufforderung, eine Kante offenzulegen.*

6. *$\mathcal{E}$ wählt nun eine zweite Kante (i_2, j_2) und lässt diese dem Prover vom Verifier übermitteln. Daraufhin deckt P die Commitments com_{i_2} und com_{j_2} auf, und $\mathcal{E}$ lernt die Farben der Knoten i_2 und j_2, nämlich $\pi(\phi(i_2))$ und $\pi(\phi(j_2))$.*

7. *So verfährt $\mathcal{E}$ so lange, bis $\mathcal{E}$ die Farben aller Knoten erfahren hat.* [4]

8. *Schließlich gibt $\mathcal{E}$ die Farben $\pi(\phi(i))$ aller Knoten i aus. Da P ein erfolgreicher Prover ist, muss P auch tatsächlich eine gültige Dreifärbung ϕ von G besitzen. Dann ist aber auch $\pi \circ \phi$ eine gültige Dreifärbung, und die Ausgabe von $\mathcal{E}$ damit* ein möglicher *sk zu pk.*

Der wesentliche Unterschied, warum ein Verifier keinerlei Informationen aus den aufgedeckten Kanten über ϕ lernt, ein Extraktor aber schon, ist, dass die dem Verifier aufgedeckten Farben stets einer anderen Permutation unterzogen werden (vgl. Beispiel 10.6, während die Kanten, die der Extraktor in Erfahrung bringt immer der selben Permutation unterliegen.

10.5 Fazit

[4] Streng genommen kann der Extraktor hiermit nur die Farben von Knoten in Erfahrung bringen, die mindestens eine Kante haben. Knoten ohne Kanten können jedoch beliebig gefärbt werden, ohne das eine Dreifärbung ihre Gültigkeit verliert.

Kapitel 11

Benutzerauthentifikation

In den vorherigen beiden Kapiteln haben wir betrachtet, wie sich ein Prover gegenüber einem Verifier identifizieren kann. Dabei konnten wir durchaus beachtliche Resultate vorweisen.

Leider kommen die bisher betrachteten Protokolle nur für die computergestützte Identifizierung des Provers gegebenüber dem Verifier in Frage. Denn kaum ein Mensch wird sich einen komplizierten geheimen Schlüssel für ein Signaturverfahren merken wollen, geschweige denn den Signaturalgorithmus von Hand ausführen wollen. Man stelle sich dies im Fall von RSA-basierten Signaturen vor: Allein der geheime Schlüssel wird eine im Allgemeinen über 600 Stellen lange Zahl sein[1]. Auch das Protokoll auf Basis der Graphdreifärbbarkeit ist nur mühsam von Hand auszuführen, da das Protokoll oft genug wiederholt werden muss, um echte Sicherheit zu bieten.

Aus diesem Grund wollen wir uns in diesem Kapitel damit auseinandersetzen, wie sich Menschen authentifizieren (können), und wie man eine solche Authentifikation möglichst sicher gestalten kann.

11.1 Passwörter

Die wohl verbreitetste Methode, die Menschen zur Authentifikation benutzen sind Passwörter. Heutzutage begegnen uns Passwörter fast überall. Ob bei Twitter, Facebook, Youtube, in den eigenen E-Mail-Konten, auf dem eigenen Computer, auf den Computern der Universität, Amazon, Ebay, in einem Online-Shop oder andernorts, beinahe überall werden Passwörter verwendet.

Wir modellieren dieses Szenario ganz allgemein: Ein Nutzer U möchte sich auf einem Server S mittels Passwort `pw` einloggen. Dabei wünschen wir uns folgende Sicherheitseigenschaften:

- Niemand außer U kann sich bei S als U einloggen.
- Niemand soll das Passwort `pw` erfahren, nach Möglichkeit auch nicht S.

Wir betrachten die zwei Angreifer Eve und Mallory. Eve kann die Kommunikation zwischen U und S abhören, aber nicht verändern. Mallory hat keinen Zugriff auf diese Kommunikation, ist dafür jedoch in der Lage, die auf dem Server gespeicherte Benutzerdatenbank zu erlangen, z.B. in dem er den Server hackt.[2] Wir betrachten diese Angreifer

[1] Für 2048-Bit RSA

[2] Es ist übrigens durchaus keine Seltenheit, dass Hacker Benutzerdatenbanken von gehackten Webseiten öffentlich ins Internet stellen. Dies ist besonders dann gefährlich, wenn Benutzer ihre Passwörter bei anderen Diensten wiederverwenden. Noch schlimmer wird es, wenn das Passwort für einen Benutzeraccount mit der (üblicherweise in Benutzerdatenbanken ebenfalls hinterlegten) E-Mail-Adresse geteilt wird. Denn ein solches E-Mail-Konto kann leicht zum Generalschlüssel zu den Benutzeraccounts des Opfers bei vielen

getrennt, d.h. Eve und Mallory kooperieren nicht. Sollten sich Eve und Mallory doch zusammentun, so können sie zusammen mindestens das erreichen, was zuvor schon einer allein erreichen konnte.

Im einfachsten Verfahren verfügen sowohl U als auch S über das Passwort `pw`. Die Authentifikation geschieht, indem U S das Passwort im Klartext übersendet. Dieses Verfahren ist in Abbildung 11.1 dargestellt.

$$U_{\texttt{pw}} \xrightarrow{\texttt{pw}} S_{\texttt{pw}}$$

Abbildung 11.1: Einfache Benutzerauthentifikation mit Passwort.

Dieses Verfahren bietet jedoch noch keinerlei Sicherheit. Eve, die die Kommunikation abhören kann, erfährt unmittelbar das Passwort. Auch Mallory, der die auf S gespeicherte Passwortliste einsehen kann, erfährt hier das Passwort.

Eine einfache Verbesserung bieten kryptographische Hashfunktionen. Der Server speichert dann einen Hashwert des Passworts anstatt des Passworts im Klartext. Dies ist in Abbildung 11.2 gezeigt.

$$U_{\texttt{pw}} \xrightarrow{\texttt{pw}} S_{H(\texttt{pw})}$$

Abbildung 11.2: Einfache Benutzerauthentifikation mit gespeichertem Passworthash.

Der Server kann nun immer noch überprüfen, ob das gesendete Passwort `pw` mit dem gespeicherten Passwort übereinstimmt, indem er den Hashwert des gesendeten Passworts mit dem gespeicherten Hashwert vergleicht. Wegen der Kollisionsresistenz der Hashfunktion werden unterschiedliche Passwörter zu unterschiedlichen Hashwerten führen. Wird jedoch das richtige Passwort verwendet, so stimmen die Hashwerte überein.

In diesem Verfahren kann Eve zwar immer noch das Passwort erhalten und sich damit später als Benutzer U bei S anmelden. Mallory jedoch, der nur auf die Benutzerdatenbank von S zugreifen kann, gelangt nur in Besitz des Passworthashes $H(\texttt{pw})$, nicht jedoch von `pw` selbst. Mallory kann sich also gegenüber S nicht als der Benutzer U ausgeben.[3]

In einer weiteren Variante sendet U nicht sein Passwort im Klartext an S, sondern hasht `pw` selbst und sendet diesen Hashwert an S. Dies ist in Abbildung 11.3 dargestellt.

$$U_{\texttt{pw}} \xrightarrow{H(\texttt{pw})} S_{H(\texttt{pw})}$$

Abbildung 11.3: Einfache Benutzerauthentifikation mit Hashfunktion und Passwort.

In dieser Variante erfährt Eve zwar nur den Hashwert $H(\texttt{pw})$ des Passworts, dies reicht ihr jedoch, um sich später gegenüber S als U auszugeben. Auch Mallory, der $H(\texttt{pw})$ kennt,

anderen Webseiten werden. Dafür muss der Angreifer nur die „Passwort Vergessen“-Funktion auf diesen Webseiten nutzen. Häufig erhält der Nutzer dann entweder ein neues Passwort zugesendet oder erhält eine Möglichkeit, selbst ein neues Passwort zu wählen. Hat der Angreifer aber Zugriff auf den E-Mail-Account des Opfers, so kann er diese Funktion selbst nutzen und sich mit den neuen Passwörtern auch bei anderen Internetseiten als das Opfer anmelden.

[3] Ist Mallory jedoch ein besonders gewiefter Hacker und hat Kontrolle über S, so könnte er jedoch auch eine Zeit lang alle an den Server gesendeten Passwörter aufzeichnen und so an eine große Zahl von Passwörtern gelangen. Meldet sich Benutzer U in dieser Zeit bei S an, so gelangt Mallory ebenfalls an das Passwort `pw`.

kann sich später als U bei S anmelden. Dafür erfahren jedoch weder Eve noch Mallory das tatsächliche Passwort `pw`. Dies schränkt die Wahrscheinlichkeit, dass sich Eve oder Mallory bei einem anderen Server anmelden können, bei dem U das selbe Passwort verwendet, ein.

Die Sicherheitseigenschaften dieser drei einfachen Protokolle sind in Tabelle 11.1 zusammengefasst.

	Eve		Mallory	
	lernt `pw`	Anmelden als U	lernt `pw`	Anmelden als U
Verfahren 1 (Abb. 11.1)	X	X	X	X
Verfahren 2 (Abb. 11.2)	X	X		
Verfahren 3 (Abb. 11.3)		X		X

Tabelle 11.1: Übersicht über die Sicherheitseigenschaften der drei betrachteten Protokolle. Die Spalten „lernt `pw`“ geben an, ob der jeweilige Angreifer das Passwort `pw` direkt lernt. Die Spalten „Anmelden als U“ geben an, ob sich der jeweilige Angreifer gegenüber S als U ausgeben kann.

Man sieht, dass das dritte Verfahren zwar das Passwort `pw` besser schützt als das zweite Verfahren. Dafür eröffnet es Mallory jedoch wieder die Möglichkeit, sich bei S als U auszugeben.

11.2 Wörterbuchangriffe

Wir betrachten nun noch einmal genauer die Möglichkeiten, aus $H(\texttt{pw})$ das benutzte Passwort `pw` zu rekonstruieren.

Wegen der Einwegeigenschaft von $H(\texttt{pw})$ ist es im Allgemeinen schwierig, `pw` durch „rückrechnen“ von H zu erhalten. Die Einwegeigenschaft von H garantiert sogar, dass es sehr schwierig ist, das Passwort `pw` zu finden, wenn das Passwort gleichverteilt zufällig gewählt wurde.

Unglücklicherweise sind Passwörter jedoch meist alles Andere als gleichverteilt zufällige Bitstrings.[4]

Natürliche Sprachen wie deutsch oder englisch verfügen nur über wenige tausend bis zehntausend Worte. Wird ein solches natürlichsprachliches Wort als Passwort verwendet, ist es ausreichend alle Worte dieser Sprache zu hashen und die Hashwerte mit $H(\texttt{pw})$ zu vergleichen. Stimmt der Hashwert eines natürlichen Wortes mit dem bekannten Hashwert $H(\texttt{pw})$ überein, so hat man `pw` gefunden. Es ist also offensichtlich, dass natürlichsprachliche Worte keine guten Passwörter sind.

Auch das Verwenden von gebräuchlichen Namen bringt keine wesentliche Verbesserung, da es auch von diesen nur wenige tausend gibt. Auch das Anhängen von Ziffern, Geburtstagen oder -jahren ergibt nicht genug Kombinationsmöglichkeiten, um eine vollständige Suche ausreichend zu erschweren.

11.3 Brute-Force-Angriffe

Solange also der Vorrat an Passwörtern klein genug ist, ist es mit relativ wenig Aufwand möglich, zu gegebenem $H(\texttt{pw})$ das ursprüngliche Passwort `pw` zu rekonstruieren. Deshalb konzentrieren wir uns nun auf den Fall, wenn der Vorrat an Passworten sehr groß ist.

[4] Eine Suche im Internet fördert verschiedene Listen der am häufigsten benutzten Passwörter zutage, darunter „123456“, „qwertz“ (im englischsprachigen Raum auch „qwerty“), „password“, oder „abc123“; außerdem findet man auch Programme, die unter Nutzung solcher Listen versuchen, Urbilder zu einer Liste von Hashes zu finden. Es gibt jedoch auch zahlreiche Anleitungen, wie gute Passworte erstellt werden können.

In diesem Fall ist es sehr aufwendig, für jeden zu brechenden Hashwert $H(\mathtt{pw})$ alle möglichen Passworte durchzuprobieren. Gibt es insgesamt N Passwörter, dann muss man H etwa $\mathcal{O}(N)$ mal auswerten. Es ist daher (aus Angreifersicht) wünschenswert, eine vollständige Liste aller möglichen Passworte $\mathtt{pw}$ und ihrer Hashwerte $H(\mathtt{pw})$ zu besitzen. Dies ist in Abbildung 11.4 illustriert.

$$\begin{array}{ccc} (& H(\mathtt{pw}_1) & , & \mathtt{pw}_1 &) \\ (& H(\mathtt{pw}_2) & , & \mathtt{pw}_2 &) \\ & \vdots & & \vdots & \end{array}$$

Abbildung 11.4: Eine Liste aller Passwörter und ihrer Hashwerte.

Ist diese Liste nach $H(\mathtt{pw})$ sortiert, kann man zu einem gegebenen Hashwert sogar durch binäre Suche sehr effizient das zugrundeliegende Passwort bestimmen, man braucht dazu nur $\mathcal{O}(\log_2 N)$ Operationen. Für sehr große Mengen an möglichen Passwörtern werden jedoch auch diese Listen sehr groß ($\Omega(N)$), und es entsteht ein Speicherplatzproblem.

11.4 Kompression von Hashtabellen/Time Memory Tradeoff

Einen Mittelweg zwischen sehr großer Suchzeit (ohne vorberechnete Tabelle aller Passworte und Hashwerte) und sehr viel Speicherplatzverbrauch (mit vollständiger Liste aller Passworte und ihrer Hashwerte) liefert die Kompression von Hashtabellen. Man bezeichnet diese Technik auch als „Time Memory Tradeoff“.

Unglücklicherweise sind gute, kryptographische Hashwerte quasi zufällig und nur sehr schwer zu komprimieren. Daher können gängige Kompressionsverfahren nicht angewendet werden.

Tatsächlich ergibt sich jedoch ein sehr einfaches, maßgeschneidertes Kompressionsverfahren für solche Hashtabellen, dass sogar eine sehr effiziente Suche erlaubt. Hierzu betrachtet man Hash*ketten*. Eine *Hashkette* (vgl. Abbildung 11.5) beginnt mit einem Passwort $\mathtt{pw}_1$ aus dem Vorrat aller Passwörter. Anschließend wird dieses Passwort gehasht, um $H(\mathtt{pw}_1)$ zu erhalten. Nun wird eine sogenannte *Reduktionsfunktion* f benutzt, um diesen Hashwert auf ein neues Passwort $\mathtt{pw}_2 = f(H(\mathtt{pw}_1))$ aus dem Passwortraum abzubilden. Anschließend wird dieses wieder zu $H(\mathtt{pw}_2)$ gehasht. Dieser Hashwert wird erneut durch f auf ein Passwort $\mathtt{pw}_3$ abgebildet, usw. Dieser Prozess kann theoretisch beliebig lange fortgeführt werden. Man beschränkt dies jedoch auf eine frei wählbare Anzahl von Iterationen m.

$$\mathtt{pw}_1 \xrightarrow{H} H(\mathtt{pw}_1) \xrightarrow{f} \mathtt{pw}_2 \xrightarrow{H} H(\mathtt{pw}_2) \xrightarrow{f} \quad \ldots \quad \xrightarrow{H} H(\mathtt{pw}_{m-1}) \xrightarrow{f} \mathtt{pw}_m \xrightarrow{H} H(\mathtt{pw}_m)$$

Abbildung 11.5: Eine Hashkette.

Eine solche Kette stellen wir auch wie in Abbildung 11.6 dar.

$$(H(\mathtt{pw}_1), \mathtt{pw}_1) \xrightarrow{f} (H(\mathtt{pw}_2), \mathtt{pw}_2) \xrightarrow{f} \quad \ldots \quad \xrightarrow{f} (H(\mathtt{pw}_m), \mathtt{pw}_m)$$

Abbildung 11.6: Eine alternative Darstellung für Hashketten.

Es ist leicht einzusehen, dass zur Konstruktion einer solchen Hashkette nur das Passwort $\mathtt{pw}_1$ benötigt wird. Man kann $\mathtt{pw}_1$ also als stark komprimierte Form der Hashkette

verstehen, da man die gesamte Kette aus $\mathtt{pw}_1$ berechnen kann.

Anstelle einer vollständigen Liste aller möglichen Passwörter speichert man nun eine Menge von n Hashketten. Diese kann man tabellarisch wie in Abbildung 11.7 darstellen.

$$
\begin{array}{c}
(H(\mathtt{pw}_{1,1}), \mathtt{pw}_{1,1}) \xrightarrow{f} (H(\mathtt{pw}_{1,2}), \mathtt{pw}_{1,2}) \xrightarrow{f} \ldots \xrightarrow{f} (H(\mathtt{pw}_{1,m}), \mathtt{pw}_{1,m}) \\
(H(\mathtt{pw}_{2,1}), \mathtt{pw}_{2,1}) \xrightarrow{f} (H(\mathtt{pw}_{2,2}), \mathtt{pw}_{2,2}) \xrightarrow{f} \ldots \xrightarrow{f} (H(\mathtt{pw}_{2,m}), \mathtt{pw}_{2,m}) \\
\vdots \\
(H(\mathtt{pw}_{n,1}), \mathtt{pw}_{n,1}) \xrightarrow{f} (H(\mathtt{pw}_{n,2}), \mathtt{pw}_{n,2}) \xrightarrow{f} \ldots \xrightarrow{f} (H(\mathtt{pw}_{n,m}), \mathtt{pw}_{n,m})
\end{array}
$$

Abbildung 11.7: Tabellarische Darstellung von n Hashketten der Länge m.

Hierbei nimmt man in Kauf, dass möglicherweise nicht alle Passwörter in der so entstehenden Tabelle auftauchen. Den Anteil dieser Passwörter kann man jedoch verringern, in dem man die Anzahl der Hashketten n oder die Länge der Hashketten m erhöht.

Um nun eine Kompression der Tabelle bei gleichzeitiger effizienter Suche zu erreichen, speichert man für jede Hashkette i nur das erste Passwort $\mathtt{pw}_{i,1}$ und den letzten Hashwert $H(\mathtt{pw}_{i,m})$. Wenn $m \cdot n$ ungefähr der Zahl aller Passwörter entspricht, dann ist die so entstehene Tabelle ungefähr um den Faktor m kleiner als eine vollständige Auflistung aller Passwörter und ihrer Hashwerte.

Die „komprimierte“ Tabelle hat also die Form:

$\mathtt{pw}_{1,1}$	$H(\mathtt{pw}_{1,m})$
$\mathtt{pw}_{2,1}$	$H(\mathtt{pw}_{2,m})$
$\vdots$	$\vdots$
$\mathtt{pw}_{n,1}$	$H(\mathtt{pw}_{n,m})$

Tabelle 11.2: Die komprimierte Hashtabelle.

Diese Tabelle wird nun nach der Spalte der Hashwerte $H(\mathtt{pw}_{i,m})$ sortiert, um eine effiziente Suche nach Hashwerten zu ermöglichen.

Nun sei $H(\mathtt{pw}^*)$ der dem Angreifer bekannte Passworthash. Das Ziel des Angreifers ist es, mittels der oben gezeigten Tabelle das Passwort $\mathtt{pw}^*$ zu rekonstruieren.

Zunächst nimmt der Angreifer an, dass das gesuchte Passwort $\mathtt{pw}^*$ als letztes Passwort in einer der Hashketten auftaucht. Es soll also $\mathtt{pw}^* = \mathtt{pw}_{i,m}$ für ein i gelten. Wenn diese Annahme zutrifft, dann ist $H(\mathtt{pw}^*)$ also $H(\mathtt{pw}_{i,m})$. Deshalb sucht der Angreifer in der zweiten Spalte von Tabelle 11.2 nach $H(\mathtt{pw}^*)$. Dies ist effizient mittels binärer Suche möglich. War die Hypothese korrekt, dann liefert diese Suche einen Treffer in der i-ten Zeile. Dann kann der Angreifer die Hashkette von $\mathtt{pw}_{i,1}$ ausgehend rekonstruieren und erhält so $\mathtt{pw}_{i,m}$. Dies ist das gesuchte Passwort $\mathtt{pw}^*$. War die Hypothese falsch, dann liefert diese binäre Suche keinen Treffer.

In diesem Fall stellt der Angreifer eine neue Hypothese auf: „Das gesuchte Passwort $\mathtt{pw}^*$ ist als zweitletztes Passwort in einer der Hashketten enthalten.“ Dann gilt also $\mathtt{pw}^* = \mathtt{pw}_{i,m-1}$ für ein i, und daher auch $H(\mathtt{pw}_{i,m}) = H(f(H(\mathtt{pw}^*)))$, denn $\mathtt{pw}_{i,m}$ ist genau $f(H(\mathtt{pw}_{i,m-1}))$. Um zu überprüfen, ob diese Hypothese stimmt, berechnet der Angreifer daher $H(f(H(\mathtt{pw}^*)))$

und sucht in Tabelle 11.2 nach dem Ergebnis dieser Berechnung. Liefert die Suche einen Treffer in Kette i, kann der Angreifer diese Kette wieder von $\mathtt{pw}_{i,1}$ neu aufbauen und erfährt so $\mathtt{pw}_{i,m-1} = \mathtt{pw}^*$. Liefert die Suche keinen Treffer, dann war die Hypothese falsch, und der Angreifer fährt mit der nächsten Hypothese fort: das gesuchte Passwort soll als drittletztes in einer der Hashketten zu finden sein. Diese Hypothese testet der Angreifer durch eine Suche nach $H(f(H(f(H(f(H(\mathtt{pw}^*)))))))$, usw.

Nacheinander testet der Angreifer so alle Positionen in den Hashketten. Liefert eine der Suchen einen Treffer, so hat der Angreifer das Passwort gefunden. Andernfalls ist das gesuchte Passwort nicht in der Hashtabelle enthalten.

Beispiel 11.1. *Wir betrachten als Raum aller möglichen Passwörter die Buchstaben „a“ bis „z“. Die angewendete Hashfunktion sei die schon bereits erwähnte SHA-1-Funktion. Um einen Hashwert zurück auf ein Passwort abzubilden, interpretieren wir den Hashwert als natürliche Zahl h in Hexadezimal-Darstellung, und berechnen $h \mod 26$. Die so entstehenden Zahlen von 0 bis 25 bilden wir auf natürliche Weise zurück auf die Buchstaben „a“ bis „z“ ab. Wir wählen $m = 4$ als Kettenlänge. Da es insgesamt 26 mögliche Passwörter gibt, könnte $n = 7$ ausreichen, damit alle Passworte an irgendeiner Stelle der Hashketten vorkommen, denn insgesamt gibt es $7 \cdot 4 = 28$ Passwörter in den Hashketten. Wir wollen es mit $n = 7$ Hashketten versuchen.*

Als Startpassworte der Hashketten wählen wir die Buchstaben „a“ bis „g“. Die so entstehenden Hashketten sind in Abbildung 11.8 gezeigt.

$$
\begin{array}{llllllllll}
\mathrm{a} & \xrightarrow{H} & 86\mathrm{f}7\ldots & \xrightarrow{f} & \mathrm{o} & \xrightarrow{H} & 7\mathrm{a}81\ldots & \xrightarrow{f} & \mathrm{w} & \xrightarrow{H} \; \mathrm{aff}0\ldots \\
\mathrm{b} & \xrightarrow{H} & \mathrm{e}9\mathrm{d}7\ldots & \xrightarrow{f} & \mathrm{i} & \xrightarrow{H} & 042\mathrm{d}\ldots & \xrightarrow{f} & \mathrm{u} & \xrightarrow{H} \; 51\mathrm{e}6\ldots \\
\mathrm{c} & \xrightarrow{H} & 84\mathrm{a}5\ldots & \xrightarrow{f} & \mathrm{w} & \xrightarrow{H} & \mathrm{aff}0\ldots & \xrightarrow{f} & \mathrm{q} & \xrightarrow{H} \; 22\mathrm{ea}\ldots \\
\mathrm{d} & \xrightarrow{H} & 3\mathrm{c}36\ldots & \xrightarrow{f} & \mathrm{c} & \xrightarrow{H} & 84\mathrm{a}5\ldots & \xrightarrow{f} & \mathrm{w} & \xrightarrow{H} \; \mathrm{aff}0\ldots \\
\mathrm{e} & \xrightarrow{H} & 58\mathrm{e}6\ldots & \xrightarrow{f} & \mathrm{v} & \xrightarrow{H} & 7\mathrm{a}38\ldots & \xrightarrow{f} & \mathrm{w} & \xrightarrow{H} \; \mathrm{aff}0\ldots \\
\mathrm{f} & \xrightarrow{H} & 4\mathrm{a}0\mathrm{a}\ldots & \xrightarrow{f} & \mathrm{l} & \xrightarrow{H} & 07\mathrm{c}3\ldots & \xrightarrow{f} & \mathrm{l} & \xrightarrow{H} \; 07\mathrm{c}3\ldots \\
\mathrm{g} & \xrightarrow{H} & 54\mathrm{fd}\ldots & \xrightarrow{f} & \mathrm{n} & \xrightarrow{H} & d185\ldots & \xrightarrow{f} & \mathrm{w} & \xrightarrow{H} \; \mathrm{aff}0\ldots
\end{array}
\begin{array}{llllll}
\xrightarrow{f} & \mathrm{q} & \xrightarrow{H} & 22\mathrm{ea}\ldots \\
\xrightarrow{f} & \mathrm{g} & \xrightarrow{H} & 54\mathrm{fd}\ldots \\
\xrightarrow{f} & \mathrm{i} & \xrightarrow{H} & 042\mathrm{d}\ldots \\
\xrightarrow{f} & \mathrm{q} & \xrightarrow{H} & 22\mathrm{ea}\ldots \\
\xrightarrow{f} & \mathrm{q} & \xrightarrow{H} & 22\mathrm{ea}\ldots \\
\xrightarrow{f} & \mathrm{l} & \xrightarrow{H} & 07\mathrm{c}3\ldots \\
\xrightarrow{f} & \mathrm{q} & \xrightarrow{H} & 22\mathrm{ea}\ldots
\end{array}
$$

Abbildung 11.8: Die in Beispiel 11.1 erzeugten Hashketten. Aus Platzgründen sind die Hashwerte auf die ersten vier Hexadezimalstellen gekürzt.

Gespeichert werden von diesen Hashketten nur die Startpassworte sowie die letzten Hashwerte. Die Tabelle wird nach den Hashwerten sortiert. Das Ergebnis ist in Tabelle 11.3 zu sehen.

c	042d...
f	07c3...
a	22ea...
d	22ea...
e	22ea...
g	22ea...
b	54fd...

Tabelle 11.3: Die komprimierte Hashtabelle aus Beispiel 11.1.

Der dem Angreifer bekannte Hashwert sei nun $H(\mathtt{pw}^*) = 042d\cdots$. Der Angreifer stellt nun zunächst die Hypothese auf, dass das gesuchte Passwort $\mathtt{pw}^*$ als letztes in einer der Hashketten auftaucht. Er sucht deshalb in Tabelle 11.3 nach dem ihm bekannten Hashwert

$042d\cdots$. *Diese suche liefert einen Treffer in Zeile* $i = 1$. *Die Hypothese war also korrekt. Nun weiß der Angreifer, dass das gesuchte Passwort* $pw^* = pw_{1,m}$ *ist. Er rekonstruiert also die Hashkette ausgehend vom Startpasswort „c“ und erhält so das gesuchte Passwort „i“.*

Beispiel 11.2. *Wir betrachten wieder die komprimierte Hashtabelle aus dem vorherigen Beispiel. Diesmal sei der dem Angreifer bekannte Hashwert aber* $H(pw^*) = 51e6\cdots$. *Der Angreifer möchte nun testen, ob das gesuchte Passwort als letztes in einer der Hashketten aus Abbildung 11.8 auftritt. Doch der gesuchte Hashwert* $51e6\cdots$ *taucht nicht in der zweiten Spalte von Tabelle 11.3 auf. Daher war diese erste Hypothese falsch. Der Angreifer berechnet* $f(H(pw^*)) = g$ *und* $H(f(H(pw^*))) = 54fd\ldots$. *Eine suche nach diesem Hashwert liefert tatsächlich einen Treffer in Zeile* $i = 7$ *der Tabelle 11.3. Der Angreifer rekonstruiert also die Hashkette ausgehend vom Buchstaben b und erhält* $pw_{i,m-1} = pw_{i,3} = u$. *Dies ist das gesuchte Passwort* pw^*.

Beispiel 11.3. *Wir betrachten wieder die selbe Hashtabelle, diesmal sei der gesuchte Hashwert jedoch* $H(pw^*) = 7a38\cdots$. *Dieser kommt in der zweiten Spalte von Tabelle 11.3 nicht vor, also taucht* pw^* *nicht an der letzten Stelle einer Hashtabelle auf. Der Angreifer berechnet daraufhin* $f(H(pw^*)) = w$ *und* $H(f(H(pw^*))) = aff0\cdots$. *Auch dieser Hashwert taucht nicht in Tabelle 11.3 auf, daher ist das gesuchte Passwort auch nicht als zweitletztes Passwort in einer der Hashketten enthalten. Deshalb setzt der Angreifer die Berechnung fort: er erhält* $f(H(f(H(pw^*)))) = q$ *und* $H(f(H(f(H(pw^*))))) = 22ea\cdots$. *Dieser Wert taucht gleich vier Mal in Tabelle 11.3 auf. Der Angreifer rekonstruiert daher die vier Ketten ausgehend von „a“, „d“, „e“ und „g“, und findet schließlich in der von „e“ ausgehenden Hashkette das gesuchte Passwort „v“.*

Dieses Beispiel illustriert bereits eines der Probleme solcher Hashtabellen: Es kann passieren, dass mehrere Hashketten, die mit verschiedenen Passworten beginnen, „zusammenlaufen“. Dies kann passieren, wenn f eine Kollision liefert, also verschiedene Hashwerte auf das selbe Passwort abbildet. (Dies lässt sich nur schwer vermeiden, da es im Allgemeinen wesentlich mehr Hashwerte als Passworte gibt.) Tritt ein solcher Fall auf, laufen die Hashketten ab diesem Punkt auch identisch weiter.

Im obigen Beispiel ist z.B. $f(H(o)) = f(H(c)) = f(H(v)) = f(H(n)) = w$. Deshalb laufen in Abbildung 11.8 die Hashketten „a“, „d“, „e“ und „g“ zusammen, und enden schließlich gemeinsam auf $H(q) = 22ea\cdots$. Tatsächlich tauchen die Passwörter „w“ und „q“ sogar noch in Hashkette „c“ auf. Dort befinden sie sich jedoch weiter vorne, deshalb endet diese Kette nicht auf $H(q) = 22ea\cdots$ sondern auf $H(i)$.

Dies führt einerseits dazu, dass gewisse Passworte mehrfach in der Hashtabelle vorkommen. Dies ist aus Angreifersicht noch kein Problem. Andererseits nehmen diese mehrfach vorkommenden Passworte jedoch auch Platz für andere Passwörter weg.

Beispiel 11.4. *Wir betrachten wieder die obigen Hashtabellen, dieses Mal ist der gesuchte Hashwert* $H(pw^*) = 95cb\cdots$.

Dieser Hashwert taucht jedoch nicht in Tabelle 11.3 auf, daher ist das gesuchte Passwort nicht an letzter Stelle einer der Hashketten.

Anschließend sucht der Angreifer nach $H(f(H(pw^*))) = 22ea\cdots$. *Diese Suche liefert vier Treffer, in den Hashketten „a“, „d“, „e“ und „g“. Der Angreifer rekonstruiert also diese Hashketten bis zur zweitletzten Position, und findet den Buchstaben w an an allen Stellen. Es gilt aber* $H(w) = aff0\cdots \neq 95cb\cdots = H(pw^*)$. *Dieses Passwort ist also* nicht *korrekt.*

Der Angreifer setzt die Suche fort und berechnet $H(f(H(f(H(pw^*))))) = 042d\cdots$. *Dies liefert wieder einen Treffer in Hashkette „c“, aber auch diesmal liefert die Rekonstruktion der Hashkette wieder das falsche Passwort „w“.*

Deshalb fährt der Angreifer weiter fort und berechnet $H(f(H(f(H(f(H(pw^*)))))))) =$

51e6 $\cdots$. *Dies liefert keinen Treffer.*

Nun hat der Angreifer alle möglichen Hypothesen getestet: Dass das Passwort als letztes (viertes), zweitletztes (drittes), drittletztes (zweites) oder viertletztes (erstes) in einer der Hashketten vorkommt. All diese Hypothesen waren falsch, also ist das gesuchte Passwort nicht in der Hashtabelle enthalten. (Das gesuchte Passwort war „y“.)

Dieses Beispiel illustriert noch ein weiteres Problem von Kollisionen: Diese können zu falsch-positiven Treffern führen. Dieses Problem lässt sich jedoch leicht beheben, in dem man jeden gefunden Passwort-Kandidaten hasht und den so entstehenden Hashwert mit dem vorgegebenen Hashwert $H(\texttt{pw}^*)$ vergleicht.

Von den insgesamt 26 möglichen Passwörtern lassen sich mit Hilfe der Tabelle 11.3 15 Passwörter rekonstruieren. Die Tabelle überdeckt also nur etwa 58% des Passwortraums.

Es sei wieder N die Zahl aller Passwörter. Zum Speichern der komprimierten Tabelle 11.2 braucht man etwa $\Omega(n)$ Speicherplatz.[5] Ist $m \cdot n \approx N$, so schrumpft der Platzbedarf gegenüber einer vollständigen Tabelle aller Passwörter und ihrer Hashwerte also etwa um den Faktor m.

Um nach einem Hashwert zu suchen, benötigt man hier $\mathcal{O}(m \cdot \log_2(n))$ Operationen, während man bei einer vollständigen Tabelle nur $\mathcal{O}(\log_2(N))$ Operationen benötigt. Der Zeitbedarf zur Suche nach einem Passwort wächst also etwa um einen Faktor von $m \cdot \frac{\log_2(n)}{\log_2(N)}$.

11.5 Rainbow Tables

Eine Technik das Zusammenlaufen von Ketten zumindest partiell zu verhindern sind sogenannte Rainbow Tables. Dabei verwendet man nicht *eine* Reduktionsfunktion, sondern $m-1$ verschiedene Reduktionsfunktionen f_i, wobei jede Reduktionsfunktion f_i für die i-te Reduktions in einer Hashkette verwendet wird. Abbildung 11.9 zeigt eine solche Hashkette.

$$\texttt{pw}_1 \xrightarrow{H} H(\texttt{pw}_1) \xrightarrow{f_1} \texttt{pw}_2 \xrightarrow{H} H(\texttt{pw}_2) \xrightarrow{f_2} \quad \ldots \quad \xrightarrow{H} H(\texttt{pw}_{m-1}) \xrightarrow{f_{m-1}} \texttt{pw}_m \xrightarrow{H} H(\texttt{pw}_m)$$

Abbildung 11.9: Eine Hashkette mit verschiedenen Reduktionsfunktionen.

Diese Änderung verhindert, dass Hashketten zusammenlaufen, solange die Kollision an verschiedenen Stellen in den Hashketten auftreten.

Beispiel 11.5. *Wir wollen dies wieder mit Hilfe von Beispiel 11.1 verdeutlichen. Wir definieren dazu die Reduktionsfunktionen f_i, die jeden Hashwert wieder als natürliche Zahl h in Hexadezimaldarstellung interpretieren. Zu dieser Zahl wird dann i addiert, und das Ergebnis modulo 26 reduziert. Es ist also $f_i(h) = h + i \mod 26$. Dies führt zu den in Abbildung 11.10 gezeigten Hashketten.*

Man sieht, dass die Hashketten „b“ und „f“ zusammenlaufen, da die Funktion f_2 eine Kollision liefert. Andererseits laufen z.B. die Hashketten „d“ und „e“ nicht *zusammen, obwohl beide ein e enthalten. Denn hier liegen die „e“s an* verschiedenen *Positionen, und die Hashwerte werden deshalb im Anschluss von verschiedenen Reduktionsfunktionen auf unterschiedliche Passworte abgebildet.*

In Abbildung 11.8 wurde noch der Buchstabe „l“ immer auf sich selbst abgebildet. Deswegen enthielt die Kette „f“ dort 3 „l“s nacheinander. So etwas tritt hier nicht auf. Zwar

[5] Zur Berechnung aller Hashketten benötigt man aber $\Omega(m \cdot n) \approx N$ Operationen. Anschließend müssen diese Hashketten noch sortiert werden.

$$
\begin{array}{llllllllll}
\text{a} & \xrightarrow{H} & \text{86f7}\ldots & \xrightarrow{f_1} & \text{p} & \xrightarrow{H} & \text{516b}\ldots & \xrightarrow{f_2} & \text{j} & \xrightarrow{H} & \text{5c2d}\ldots & \xrightarrow{f_3} & \text{r} & \xrightarrow{H} & \text{4dc7}\ldots \\
\text{b} & \xrightarrow{H} & \text{e9d7}\ldots & \xrightarrow{f_1} & \text{j} & \xrightarrow{H} & \text{5c2d}\ldots & \xrightarrow{f_2} & \text{q} & \xrightarrow{H} & \text{22ea}\ldots & \xrightarrow{f_3} & \text{l} & \xrightarrow{H} & \text{07c3}\ldots \\
\text{c} & \xrightarrow{H} & \text{84a5}\ldots & \xrightarrow{f_1} & \text{x} & \xrightarrow{H} & \text{11f6}\ldots & \xrightarrow{f_2} & \text{u} & \xrightarrow{H} & \text{51e6}\ldots & \xrightarrow{f_3} & \text{j} & \xrightarrow{H} & \text{5c2d}\ldots \\
\text{d} & \xrightarrow{H} & \text{3c36}\ldots & \xrightarrow{f_1} & \text{d} & \xrightarrow{H} & \text{3c36}\ldots & \xrightarrow{f_2} & \text{e} & \xrightarrow{H} & \text{58e6}\ldots & \xrightarrow{f_3} & \text{y} & \xrightarrow{H} & \text{95cb}\ldots \\
\text{e} & \xrightarrow{H} & \text{58e6}\ldots & \xrightarrow{f_1} & \text{w} & \xrightarrow{H} & \text{aff0}\ldots & \xrightarrow{f_2} & \text{s} & \xrightarrow{H} & \text{a0f1}\ldots & \xrightarrow{f_3} & \text{g} & \xrightarrow{H} & \text{54fd}\ldots \\
\text{f} & \xrightarrow{H} & \text{4a0a}\ldots & \xrightarrow{f_1} & \text{k} & \xrightarrow{H} & \text{13fb}\ldots & \xrightarrow{f_2} & \text{q} & \xrightarrow{H} & \text{22ea}\ldots & \xrightarrow{f_3} & \text{l} & \xrightarrow{H} & \text{07c3}\ldots \\
\text{g} & \xrightarrow{H} & \text{54fd}\ldots & \xrightarrow{f_1} & \text{m} & \xrightarrow{H} & \text{6b0d}\ldots & \xrightarrow{f_2} & \text{m} & \xrightarrow{H} & \text{6b0d}\ldots & \xrightarrow{f_3} & \text{n} & \xrightarrow{H} & \text{d185}\ldots
\end{array}
$$

Abbildung 11.10: Die in Beispiel 11.5 erzeugten Hashketten.

werden immer noch Buchstaben auf sich selbst abgebildet (siehe z.B. das „m“ in Kette „g“). Da jedoch immer verschiedene Reduktionsfunktionen verwendet werden, wird das zweite „m“ nicht mehr auf sich selbst sondern auf „n“ abgebildet.

Wegen dieser Eigenschaft haben Rainbow Tables im Allgemeinen eine bessere Abdeckung des Passwort-Raums als gleich große Hashtabellen mit nur einer Reduktionsfunktion. Unsere Rainbow Table hier deckt z.B. 20 der 26 möglichen Passwörter ab, also ca. 77% des Passwortraums. Die Hashtabelle mit nur einer Reduktionsfunktion deckte nur 15 Passwörter (58%) ab.[6]

Der Begriff „Rainbow Tables“ bezieht sich auf die verschiedenen „Farben“ der Reduktionsfunktionen f_i.

Die Suche in Rainbow Tables funktioniert konzeptionell genau wie bei Hashtabellen mit nur einer Reduktionsfunktion: Man testet nacheinander die Hypothesen „Das gesuchte Passwort taucht als j-tes ein einer Hashkette i auf.“ Um zu testen, ob das gesuchte Passwort $\mathtt{pw}^*$ an Stelle m ist, muss man also nach $H(\mathtt{pw}^*)$ suchen. Um zu testen, ob das gesuchte Passwort an Stelle $m-1$ ist, berechnet man $H(f_{m-1}(H(\mathtt{pw}^*)))$ und sucht nach diesem Hashwert in der Rainbow Table. Um zu testen, ob $\mathtt{pw}^*$ an Stelle $m-2$ liegt, berechnet man $H(f_{m-1}(H(f_{m-2}(H(\mathtt{pw}^*)))))$ und sucht nach diesem Ergebnis, usw.

Beispiel 11.6. *Wir verwenden die Hashketten aus Abbildung 11.10. Aus diesen ergibt sich die komprimierte Rainbow Table 11.4.*

b	07c3...
f	07c3...
a	4dc7...
e	54fd...
c	5c2d...
d	95cb...
g	d185...

Tabelle 11.4: Die komprimierte Rainbow Table für die Hashketten aus Abbildung 11.10.

Der gesuchte Hashwert sei $H(\mathtt{pw}^*) = 11f6\cdots$. Die Hypothese, dass $\mathtt{pw}^* = \mathtt{pw}_{i,m}$ für ein i sei, stellt sich als falsch heraus, denn $H(\mathtt{pw}^*)$ taucht in der zweiten Spalte der Rainbow Table auf.

[6] Man kann auch vorberechnete Rainbow Tables für wenige hundert Euro kaufen. Diese erreichen häufig Abdeckungsraten von weit über 90%, und werden wegen ihrer Größe gleich auf mehreren externen Terabyte-Festplatten geliefert.

Man testet daher, ob $\mathtt{pw}^* = \mathtt{pw}i, m-1$ *für ein* i *ist. Dazu berechnet man* $f_{m-1}(H(\mathtt{pw}^*)) = v$ *und* $H(f_{m-1}(H(\mathtt{pw}^*))) = 7a38\cdots$. *Die binäre Suche nach diesem Wert liefert ebenfalls kein Ergebnis, daher war auch diese Hypothese falsch.*

Die nächste Hypothese ist, dass $\mathtt{pw}^* = \mathtt{pw}_{i,m-2}$ *für ein* i *sein soll. Man berechnet* $f_{m-2}(H(\mathtt{pw}^*)) = u$, $H(f_{m-2}(H(\mathtt{pw}^*))) = 51e6\cdots$, $f_{m-1}(H(f_{m-2}(H(\mathtt{pw}^*)))) = j$ *und* $H(f_{m-1}(H(f_{m-2}(H(\mathtt{pw}^*))))) = 5c2d\cdots$. *Diesmal liefert die Suche in Tabelle 11.4 einen Treffer in Zeile* $i = 5$. *Das Startpasswort der Hashkette war „c“. Deshalb rekonstruiert man die Hashkette ausgehend von* c *und findet* $\mathtt{pw}^* = \mathtt{pw}_{5,2} = x$.

Anders als bei Hashtabellen mit nur einer Reduktionsfunktion benötigt man hier jedoch $\mathcal{O}(m^2 \cdot \log_2(n))$ Operationen für eine Suche, da man für jede Hypothese die Berechnung des entsprechenden Hashwerts neu beginnen muss.

11.6 Gegenmaßnahmen

Nachdem wir nun gesehen haben, wie man bekannte Passworthashes mit Hilfe von vorberechneten Tabellen relativ effizient auf ihr Passwort zurück abbilden kann, wollen wir uns nun noch einmal damit befassen, wie man solche Angriffe erschwert.

Eine einfache Lösung bieten sogenannte „gesalzene“ Hashwerte. In diesem Szenario ist jedem Benutzer noch ein individuelles „Salz“ s (englisch „salt“) zugeordnet. Der Hashwert des Passwortes ist dann $H(s, \mathtt{pw})$. In der Praxis ist dies oft ein zufälliger String, der vorn oder hinten an das Passwort angehängt wird.

Vorberechnete Hash-Tabellen (wie z.B. Rainbow Tables) werden dadurch nutzlos. Die Erstellung von Rainbow Tables o.Ä. wird ist erst dann sinnvoll, wenn der Angreifer den Salt kennt. Und selbst dann hilft die Rainbow Table nur beim Knacken *eines* Passworthashes, da verschiedene Benutzer im Allgemeinen verschiedene Salts haben. Dann liefert die Vorberechnung von Rainbow Tables aber auch keinen Vorteil mehr gegenüber dem Ausprobieren aller möglichen Passworte.

Theoretisch wäre es zwar auch möglich, eine Rainbow Table über *alle* Kombinationen von Salt und Passwort zu erstellen. Für ausreichend lange und zufällige Salts ist der Aufwand hierfür jedoch nicht praktikabel.

Eine zweite einfache Möglichkeit ist die Wiederholung der Hashfunktion. Dies wird auch als „Key Stengthening“ bezeichnet.

In diesem Fall ist der gespeicherte Passworthash nicht mehr $H(\mathtt{pw})$, sondern $H(H(\ldots H(\mathtt{pw})\ldots))$. Wiederholt man die Funktion H z.B. n mal, so wird der Aufwand, der zum Knacken von Passwörtern oder zur Erstellung einer Rainbow Table benötigt wird, ver-n-facht.

Andererseits wird auch der Aufwand zur Verifikation eines Passworts um den Faktor n gesteigert, da der Server S nun bei jeder versuchten Anmeldung die Hashfunktion H insgesamt n mal ausführen muss.

Diese Methode kann jedoch trotzdem sinnvoll sein, da S im Allgemeinen weniger Passworthashes berechnen muss als ein Angreifer. Selbst bei einem sehr viel genutzten Dienst sind höchstens wenige Milliarden Login-Versuche pro Tag zu erwarten. Wiederholt man die Funktion H 1000 mal, so muss S etwa 10^{12} Auswertungen von H pro Tag durchführen. Ist der Passwortraum aber größer als 10^9, z.B. 10^{15}, so müsste der Angreifer insgesamt 10^{18} mal die Funktion H auswerten. Dies stellt den Angreifer vor eine deutlich größere Herausforderung als den Betreiber des Servers S.

Kapitel 12

Zugriffskontrolle

Nachdem wir uns in Kapitel 11 mit der Benutzerauthentifikation beschäftigt haben, ist der nächste Schritt, authentifizierten Nutzern Rechte zuzuweisen, um Zugriff auf Informationen regulieren zu können. Die Zugriffskontrolle ist ein Mechanismus, um Vertraulichkeit und Datenschutz in einem zusammenhängenden System zu gewährleisten. Betrachten wir als naheliegendes Szenario ein Unternehmen, in dem Informationen unterschiedlich schützenswert sind. Beispielsweise muss die Finanzabteilung Zugriff auf die Löhne aller Mitarbeiter haben, die anderen Mitarbeiter hingegen nicht. Andererseits soll sie Produktplanungen der Ingenieur-Abteilung nicht einsehen dürfen.

Ein erster möglicher Ansatz stellt eine feste Zuweisung von Rechten an verschiedene Benutzergruppen dar. Formalisieren wir obiges Beispiel, benötigen wir:

- eine Menge $\mathcal{S}$ von Subjekten, die Mitarbeiter
- eine Menge $\mathcal{O}$ von Objekten aller Informationen (Gehälter, Produktskizzen,...)
- eine Menge $\mathcal{R}$ von Zugriffsrechten (Leserecht, Schreibrecht,...)
- eine Funktion $f : \mathcal{S} \times \mathcal{O} \to \mathcal{R}$, die das Zugriffsrecht eines Subjektes auf ein Objekt angibt

Ein Problem dieses Modells ist, dass eigentlich vertrauliche Informationen, gewollt oder ungewollt, an nicht autorisierte Personen gelangen. Ein Ingenieur mit Leserechten auf eine Produktskizze kann diese in ein öffentliches Dokument eintragen, sofern er die notwendigen Schreibrechte besitzt. Nach dem Lesen vertraulicher Daten sollte das Schreiben auf öffentliche Dokumente untersagt sein.

Für ein Unternehmen ist das vorgeschlagene Modell daher in vielen Fällen zu statisch. Praktische Verwendung findet es dennoch, zum Beispiel in der UNIX-Rechteverwaltung.

12.1 Das Bell-LaPadula-Modell

Ein Modell mit dynamischer Zugriffskontrolle ist Bell-LaPadula. Um die Zugriffskontrolle zu ermöglichen, betrachten wir zunächst die elementaren Bestandteile und formalisieren ähnlich wie oben:

- eine Menge $\mathcal{S}$ von Subjekten
- eine Menge $\mathcal{O}$ von Objekten
- eine Menge $\mathcal{A} = \{\texttt{read}, \texttt{write}, \texttt{append}, \texttt{execute}\}$ von Zugriffsoperation

- eine halbgeordnete[1] Menge $\mathcal{L}$ von Sicherheitsleveln, auf der ein eindeutiges Maximum definiert ist

Die obige Auflistung beschreibt das System, für das Bell-LaPadula einen Zugriffskontrollmechanismus realisieren soll. Da Bell-LaPadula im Gegensatz zum bereits vorgeschlagenen Modell dynamisch sein soll, interessieren wir uns vor allem für den Systemzustand. Den Systemzustand formalisieren wir dabei als Tripel (B, M, F), wobei

- $B \subseteq \mathcal{S}\times\mathcal{O}\times\mathcal{A}$ die Menge aller aktuellen Zugriffe ist,
- $M = (m_{i,j})_{i=1,\ldots,|\mathcal{S}|,j=1,\ldots,|\mathcal{O}|}$ die Zugriffskontrollmatrix ist, deren Eintrag $m_{i,j} \subseteq \mathcal{A}$ die erlaubten Zugriffe des Subjektes i auf das Objekt j beschreibt und
- $F = (f_s, f_c, f_o)$ ein Funktionstripel ist, mit:
  - $f_s : \mathcal{S} \to \mathcal{L}$ weist jedem Subjekt ein maximales Sicherheitslevel zu
  - $f_c : \mathcal{S} \to \mathcal{L}$ weist jedem Subjekt sein aktuelles Sicherheitslevel zu
  - $f_o : \mathcal{O} \to \mathcal{L}$ weist jedem Objekt ein Sicherheitslevel zu

In einem Unternehmen U, in dem $\mathcal{S} = \{\texttt{Smith}, \texttt{Jones}, \texttt{Spock}\}$ die Menge der Angestellten und $\mathcal{O} = \{\texttt{salary.txt}, \texttt{mail}, \texttt{fstab}\}$ die schützenswerten Informationen sind, könnte eine Zugriffskontrollmatrix M demnach wie folgt aussehen:

	salary.txt	mail	fstab
Smith	{read}	{execute}	$\emptyset$
Jones	{read, write}	{read, write, execute}	$\emptyset$
Spock	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$

Sei weiterhin $\mathcal{L} = \{\texttt{topsecret}, \texttt{secret}, \texttt{unclassified}\}$ die Menge der Sicherheitslevel, die U zur Realisierung der Zugriffskontrolle mittels Bell-LaPadula verwendet. Die darauf definierte Halbordnung sei $\texttt{unclassified} < \texttt{secret} < \texttt{topsecret}$. Ein Beispiel für das Funktionstripel wäre somit:

	$f_s(\cdot)$	$f_c(\cdot)$
Smith	unclass.	unclass.
Jones	secret	unclass.
Spock	topsecret	unclass.

	$f_o(\cdot)$
salary.txt	secret
mail	unclass.
fstab	topsecret

Es fällt auf, dass ein Matrixeintrag $m_{i,j}$ nicht leer sein muss, selbst wenn das maximale Sicherheitslevel des Subjektes kleiner dem des Objektes ist. Wäre der Systemzustand (B, M, F) von U mit obiger Matrix und Funktionstripel allerdings sicher, sollte Smith lesend auf salary.txt zugreifen wollen? Intuitiv natürlich nicht. Wie aber können wir formal korrekt einen sicheren Systemzustand beschreiben und was heißt sicher im Kontext der Zugriffskontrolle überhaupt? Hierfür müssen wir zunächst eine Menge an Eigenschaften definieren:

Definition 12.1 (Discretionary-Security/ds-Eigenschaft)**.** Ein Systemzustand (B, M, F) hat die ds-Eigenschaft, falls: $\forall\ (s, o, a) \in B : a \in m_{s,o}$.

[1] Unter einer halbgeordneten Menge versteht man eine Menge, auf der eine reflexive, transitive und antisymmetrische Relation definiert ist, zum Beispiel $(\mathbb{N}, \geq)$.

Die ds-Eigenschaft ist die erste naheliegende Forderung, die ein sicherer Systemzustand nach Bell-LaPadula erfüllen sollte. So wird sichergestellt, dass alle aktuellen Zugriffe konsistent mit der Zugriffsmatrix sind.

Definition 12.2 (Simple-Security/ss-Eigenschaft)**.** Ein Systemzustand (B, M, F) hat die ss-Eigenschaft, falls: $\forall\ (s, o, \texttt{read}) \in B : f_s(s) \geq f_o(o)$.

Ebenso naheliegend ist es, dass kein Subjekt lesend auf Objekte zugreifen sollte, deren Sicherheitslevel das maximale Sicherheitslevel des zugreifenden Subjekts übersteigt. Insbesondere ist zu beachten, dass die ss-Eigenschaft in diesem Skript ausschließlich für Leseoperationen definiert ist. Oftmals wird sie daher auch als "no read up" bezeichnet.

Ist für eine gegebene Anfrage $(s, o, \texttt{read})$ die ss-Eigenschaft erfüllt, wird das aktuelle Sicherheitslevel des Subjektes angepasst: $f_c(s) = \max\{f_c(s), f_o(o)\}$. Die ss-Eigenschaft ist somit zentral für den dynamischen Ansatz des Bell-LaPadula-Modells.

Definition 12.3 (Star Property/$\star$-Eigenschaft)**.** Ein Systemzustand (B, M, F) hat die $\star$-Eigenschaft, falls: $\forall\ (s, o, \texttt{\{write, append\}}) \in B : f_o(o) \geq f_c(s)$.

Ein bisschen weniger intuitiv ist die $\star$-Eigenschaft, die verhindert, dass sensitive Informationen in weniger sensitive Objekte geschrieben werden. Sie verlangt, dass Subjekte, die lesend auf (sensitive) Objekte zugegriffen haben, nur noch in Objekte schreiben dürfen, deren Sicherheitslevel mindestens genauso hoch ist. Als alternative Bezeichnung dieser Eigenschaft wird deswegen oft "no write down" verwendet.

Wir bezeichnen einen Systemzustand (B, M, F) als sicher, falls es keinen Zugriff $b \in B$ gibt, der eine der drei Eigenschaften verletzt. Bell-LaPadula erlaubt einen Zugriff ausschließlich bei Erhalt des Systemsicherheit.

12.1.1 Nachteile des Bell-LaPadula-Modells

Ein offensichtlicher Nachteil dieses Modells ist, dass die aktuellen Sicherheitslevel nie herabgesetzt werden. Das Lesen eines Objektes o mit $f_o(o) > f_c(s)$ schränkt folglich dauerhaft die Menge an Objekten ein, auf die ein Subjekt s schreibend zugreifen kann. Ein Zurücksetzen des aktuellen Sicherheitslevels zu einem Zeitpunkt ist nicht realistisch, da die Subjekte in der Regel nicht gezwungen werden können, gelesene Informationen zu vergessen.

Ein anderer Lösungsansatz für dieses Problem stellt die Einteilung der Subjekte in vertrauenswürdige und nicht vertrauenswürdige Subjekte dar. Für Erstere wird, ausgehend davon, dass keine Weitergabe von Informationen an nicht berechtigte Subjekte erfolgt, die $\star$-Eigenschaft ausgesetzt. Hier ist die Qualität der Prüfung, ob ein Subjekt vertrauenswürdig ist oder nicht, entscheidend für die Sicherheit des Modells.

Ein zweiter gravierender Nachteil ergibt sich daraus, dass Subjekte auf Objekte höheren Sicherheitslevels schreibend zugreifen dürfen. Somit können gezielt sensitive Objekte von nicht autorisierten Subjekten geändert werden, was zu hohen Schäden führen kann. Ein subtilerer Nachteil ergibt sich aus der Tatsache, dass Subjekte beispielsweise die Existenz von sensitiven Objekten erfahren können. Zwar ist Bell-LaPadula dynamischer als das in der Einleitung vorgestellte Modell, doch sind die Zugriffskontrollmatrix M und die Funktionen f_s, f_o unveränderlich.

Zusammenfassend betrachtet realisiert das Bell-LaPadula-Modell eine Zugriffskontrolle, die zuverlässig vor Informationsweitergabe an unautorisierte Subjekte schützt, jedoch in vielen Szenarien auf Dauer zu unflexibel ist und auch nicht vor Datenmanipulation schützen kann.

12.2 Das Chinese-Wall-Modell

Das Chinese-Wall-Modell realisiert, ähnlich dem Bell-LaPadula-Modell, eine dynamische Zugriffskontrolle. Das Szenario, in welchem die beiden Modelle jeweils angewandt werden, unterscheidet sich allerdings fundamental. Während das Bell-LaPadula-Modell grundsätzlich Informationen in einem geschlossenen System, wie zum Beispiel einer Firma, schützen soll, ist das Chinese-Wall-Modell beispielsweise für Szenarien konzipiert, in denen Interessenskonflikte zwischen mehreren Firmen entstehen können. Stellen wir uns vor, eine Menge von Beratern berät Firmen zu einer Menge von Objekten. Dieses Gedankenspiel ist gänzlich unproblematisch, solange jeder Berater maximal eine Firma berät. Ist allerdings bereits ein Berater bei mehreren Firmen unter Vertrag, so kann es, sollten die Firmen konkurrieren, zu einem Interessenkonflikt kommen. Bell-LaPadula liefert auf diese Problemstellung keine Antwort. Es gibt weder Sicherheitslevel, noch kann ein Systemzustand formuliert werden, da Zugriffskontrollmatrix und Funktionen fehlen. Um eine Lösung anbieten zu können, müssen wir zunächst die neue Wirklichkeit als System formalisieren. Wir brauchen gemäß unserem Szenario

- eine Menge $\mathcal{C}$ von Firmen,
- eine Menge $\mathcal{S}$ von Beratern,
- eine Menge $\mathcal{O}$ von Objekten,
- eine Menge $\mathcal{A} = \{\texttt{read}, \texttt{write}\}$ von Zugriffsoperationen,
- eine Funktion $y : \mathcal{O} \to \mathcal{C}$ die jedem Objekt seine eindeutige Firma zuweist und
- eine Relation $x : \mathcal{O} \to \mathcal{C}$ die jedem Objekt die **Menge** an Firmen zuweist, mit denen es in Konflikt steht.

Das Ziel ergibt sich ebenfalls aus unserem Gedankenspiel: Eine konfliktfreie Zuordnung von Beratern zu Objekten. Doch wie kann garantiert werden, dass eine Zuordnung konfliktfrei ist? Es ist naheliegend, dass bei jedem Schreib- oder Lesezugriff $(s, o, a) \in \mathcal{S}\times\mathcal{O}\times\mathcal{A}$ ein Konflikt entsteht, falls s in der Vergangenheit bereits Zugriff auf ein Objekt hatte, das in Konflikt mit $y(o)$ steht. Formal definieren wir:

Definition 12.4 (Simple-Security/ss-Eigenschaft)**.** Eine Anfrage $(s, o, a) \in \mathcal{S}\times\mathcal{O}\times\mathcal{A}$ hat die ss-Eigeschaft, falls: $\forall\, o' \in \mathcal{O}$, auf die s schon Zugriff hatte, gilt:
$y(o) = y(o') \vee y(o) \notin x(o')$.

Eine konfliktfreie Zuordnung muss jeden Zugriff ablehnen, für den die ss-Eigenschaft nicht gilt. Hinreichend ist das jedoch nicht, da ein ungünstiges Zusammenspiel von Beratern ungewollten Informationsfluss ermöglichen kann.

Beispiel 12.5. *Für zwei Berater $s_1, s_2 \in \mathcal{S}$ wird folgender Ablauf betrachtet:*

1. *Lesezugriff* $(s_1, o_1, \texttt{read})$
2. *Schreibzugriff* $(s_1, o_2, \texttt{write})$
3. *Lesezugriff* $(s_2, o_2, \texttt{read})$
4. *Schreibzugriff* $(s_2, o_3, \texttt{write})$

Es ist denkbar, dass $y(o_3) \in x(o_1)$, die Firma von o_3 also mit o_1 in Konflikt steht. Durch den letzten Schreibzugriff könnte demnach indirekt Information geflossen sein, die das Chinese-Wall-Modell eigentlich hätte schützen sollen.

Um indirekten Informationsfluss zu verhindern, brauchen wir neben der ss-Eigenschaft eine zusätzliche Forderung. Eine Schreibanfrage eines Beraters soll nur dann erlaubt werden, falls alle von ihm zuvor gelesen Objekte entweder aus der gleichen Firma stammen oder mit keiner Firma in Konflikt stehen. Formalisieren wir unsere Forderung als Eigenschaft, ergibt sich:

Definition 12.6 (Star Property/$\star$-Eigenschaft)**.** Eine Schreibanfrage $(s, o, \texttt{write}) \in \mathcal{S} \times \mathcal{O}$ hat die $\star$-Eigenschaft, falls: $\forall\, o' \in \mathcal{O}$, auf die s schon lesend zugegriffen hat, gilt:
$y(o) = y(o') \vee x(o') = \emptyset$.

Erlauben wir ausschließlich Anfragen, die beide Eigenschaften erfüllen, können wir eine konfliktfreie Zuordnung garantieren. Ungewollter Informationsfluss - direkter, sowie indirekter - kann ausgeschlossen werden. Auffällig ist jedoch die Striktheit der $\star$-Eigenschaft, die gerade mit zunehmender Dauer den Beratern enge Grenzen steckt. Eine Möglichkeit, die gelesenen Objekte eines Beraters (nach einer gewissen Zeit) zurückzusetzen, gibt es nicht. Um höchstmögliche Sicherheit zu bieten, ist das Fehlen eines solchen Mechanismus allerdings sinnvoll.

Kapitel 13

Analyse von umfangreichen Protokollen

Bisher haben wir in dieser Vorlesung hauptsächlich kryptographische *Bausteine* betrachtet, z.B. Chiffren, Hashfunktionen, Nachrichtenauthentifikation mit MACS oder digitalen Signaturen und Schlüsselaustauschprotokolle. Die Konstruktion solcher Bausteine ist jedoch kein Selbstzweck. Vielmehr sind diese Bausteine lediglich Hilfsmittel. Um „sichere“ Kommunikation zu ermöglichen, müssen diese Bausteine geeignet miteinander kombiniert werden.

Das bei weitem nicht jede mögliche Kombination auch die erwünschten Sicherheitseigenschaften hat, zeigt folgendes einfaches Beispiel.

Beispiel 13.1. *Es soll ein einfaches Kommunikationsprotokoll zwischen zwei Teilnehmern Alice und Bob erstellt werden. Dabei soll folgendes gelten:*

- *Der Inhalt der Kommunikation bleibt geheim, nur Alice und Bob kennen ihn. (Confidentiality)*
- *Nachrichten können vom Angreifer nicht verändert werden. (Integrity)*
- *Alice und Bob können sich sicher sein, dass ihr Kommunikationspartner tatsächlich Bob bzw. Alice ist. (Authenticity)*[1]

Als Bausteine sollen hierfür ein symmetrisches Verschlüsselungsverfahren, das auch die Unveränderbarkeit von Nachrichten garantiert, ein Schlüsselaustauschprotokoll und ein Protokoll zur gegenseitigen Identifikation verwendet werden.

Da das Schlüsselaustauschprotokoll rechenintensiv ist, kommt der Protokolldesigner auf die Idee, dass sich Alice und Bob zunächst gegenseitig identifizieren sollen, bevor sie das Schlüsselaustauschprotokoll, und anschließend die symmetrische Chiffre verwenden. Das zusammengesetzte Protokoll hat also den in Abbildung 13.1 gezeigten Ablauf.

Dieses Protokoll bietet jedoch keinen Schutz gegen den Angreifer Mallory, der Nachrichten abfangen kann. Mallory kann nämlich abwarten, bis Alice und Bob das Identifikationsprotokoll ausgeführt haben. Dann kann Mallory alle Nachrichten von und zu Alice abfangen, und stattdessen an Alice' Stelle das Schlüsselaustauschprotokoll mit Bob durchführen und anschließend unter Alice' Identität mit Bob kommunizieren. Bob hat in diesem Protokoll keine Möglichkeit, dies zu erkennen und wird glauben, mit Alice zu kommunizieren.[2]

[1] Eine reale Implementierung eines solchen großen Protokolls ist z.B. das schon erwähnte TLS (siehe Kapitel 8.3), das jedoch andere Primitive benutzt.

[2] Eine bessere Alternative wäre, dass Alice und Bob zunächst das Schlüsselaustauschprotokoll ausführen, und dann verschlüsselt das Identifikationsprotokoll ausführen.

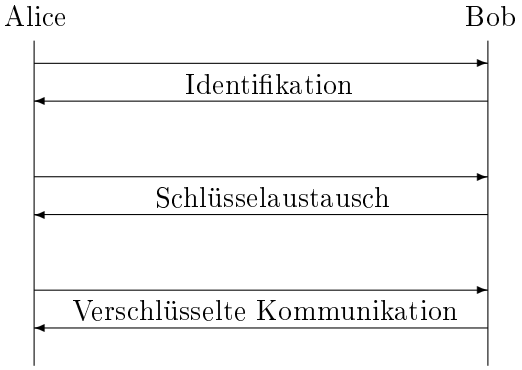


Abbildung 13.1: Das Kommunikationsprotokoll aus Beispiel 13.1.

Selbst wenn also alle Bausteine ihr *eigenes* Sicherheitsziel optimal erreichen, bleibt das zusammengesetzte Protokoll unsicher. Man sagt auch: „Sicherheit komponiert nicht.“[3]

In diesem Kapitel wollen wir uns damit befassen, wie man „zusammengesetzte“ Protokolle auf ihre Sicherheit hin untersuchen kann. Hier unterscheiden wir zwei verschiedene Ansätze:

- Der „Security“-Zugang definiert für ein Protokoll zunächst eine Reihe von Sicherheitseigenschaften. Diese werden dann einzeln nachgewiesen.
- Der kryptographische Ansatz definiert ein hypothetisches, idealisiertes, optimales Protokoll, und vergleicht anschließend die Implementierung mit diesem.

13.1 Der Security-Ansatz

Kern des Security-Ansatzes ist die Liste der erwünschten Sicherheitseigenschaften. Genau diese Liste ist aber auch die Schwachstelle des Security-Ansatzes. Denn wie stellt man sicher, dass die Liste vollständig ist, dass also nichts vergessen wurde? Und wie formalisiert man die erwünschten Eigenschaften genau?

Diese Fragen sind leider bis heute ungeklärt, entziehen sich aber auch der systematischen Erforschung.

Dem ersten Problem (Vollständigkeit) kann man z.B. mit einem Mehr-Augen-Prinzip begegnen, und mit etwas Erfahrung mag auch manch einer fähig sein, eine „gute“ Liste an benötigten Sicherheitseigenschaften aufzustellen. Außerdem kann es hilfreich sein, eine Liste von Sicherheitseigenschaften zu haben, die in anderen Protokollen erwünscht waren:

Vertraulichkeit/Confidentiality Bestimmte Informationen bleiben geheim. Dabei muss man definieren, wer die Information erhalten darf, und wer nicht.

Integrität/Integrity Nachrichten/Informationen bleiben unverändert.

Authentizität/Authenticity Man kann Nachrichten nicht unter fremder Identität verschicken.

Verfügbarkeit/Availability Ein Service bleibt auch unter Angriffen verfügbar. Dies ist im Wesentlichen die Robustheit gegen Denial-of-Service-Angriffe.

[3] Die Konstruktion von Protokollen, die immer und unter allen Umständen komponieren („Universal Composability“), ist ein aktuelles Forschungsthema.

Autorisierung/Authorization Jeder Benutzer eines Systems kann nur Aktionen durchführen oder Informationen einsehen, zu denen er berechtigt ist.

Nicht-Abstreitbarkeit/Non-Repudiability Man kann nicht glaubhaft abstreiten, Urheber einer Information zu sein. Dies ist z.B. bei digital unterschriebenen Verträgen wichtig.

Abstreitbarkeit/Plausible Deniability Man kann nicht beweisen, dass jemand Urheber einer Information ist. Dies ist z.B. für Whistleblower wünschenswert, wenn sie geheime Informationen an Journalisten übergeben.

Die konkreten Formen, die diese abstrakten Sicherheitseigenschaften in verschiedenen Protokollen annehmen, können sich unterscheiden. Bei Verschlüsselungen z.B. bedeutet die Vertrauchlichkeit, dass nur die legitimen Protokollteilnehmer, d.h. die beiden kommunizierenden Parteien, die Information kennen dürfen. Bei Commitments aber darf der Empfänger die Information zunächst nicht lernen (wegen der Hiding-Eigenschaft).

13.2 Der kryptographische Ansatz

Dem Problem bei der Formulierung der Sicherheitsziele begegnet der kryptographische Ansatz zumindest teilweise.

Hier definiert man zunächst ein idealisiertes Protokoll, dass unter Ausschluss von Angreifern und ausschließlich mit ehrlichen und vertrauenswürdigen Parteien arbeitet. Insbesondere kann man hier auch einen vertrauenswürdigen „Notar“ einführen, der Geheimnisse der anderen Parteien erfahren darf, sie aber niemals weitergibt.

Der Nachweis der Sicherheit erfolgt dann durch Vergleich des realen Protokolls mit dem idealisierten Protokoll. Kern dieses Vergleichs ist eine „mindestens-so-sicher-wie“-Relation auf Protokollen, die wir hier als $\geq$ bezeichnen.

Definition 13.2 (Simulierbarkeit, informell)**.** Protokoll π_1 ist *so sicher wie* Protokoll pi_2 (kurz: $\pi_1 \geq \pi_2$), falls für jeden effizienten Angreifer $\mathcal{A}$ auf π_1 ein effizienter Simulator $\mathcal{S}$ auf π_2 existiert, so dass nicht effizient zwischen $(\pi_1, \mathcal{A})$ und $(\pi_2, \mathcal{S})$ unterschieden werden kann.

Diese Definition bedeutet, dass jede Schwäche im realen Protokoll π_1, die ein effizienter Angreifer ausnutzen kann, schon im idealen Protokoll π_2 enthalten ist Umgekehrt besitzt π_1 keine Schwachstellen, die nicht schon in π_2 enthalten sind. Durch entsprechende Modellierung des idealen Protokolls erhält man eine Aussage über die Sicherheit von π_1.

Auch dieser Ansatz stößt aber an gewisse Grenzen, wie folgendes Beispiel zeigt.

Beispiel 13.3. *Wir möchten einen sicheren Kanal mit Hilfe einer Verschlüsselung realisieren. Unser ideales Protokoll π_2 ist also der sichere Kanal, π_1 ist ein unsicherer Kanal, über den jedoch verschlüsselt kommuniziert wird.*

Abbildung 13.2 zeigt unser idealisiertes Protokoll π_2. Dieses Protokolls soll nun durch das reale Protokoll π_1, das in Abbildung 13.3 gezeigt ist, implementiert werden.

Hier gilt jedoch nicht *$\pi_1 \geq \pi_2$, denn in π_1 erfährt der Angreifer, dass bzw. ob Kommunikation stattfindet. Außerdem kann der Angreifer aus dem Chiffrat die ungefähre Länge der Nachricht ermitteln. Im idealen Protokoll π_2 erfährt der Angreifer dies jedoch nicht. Deshalb gilt hier $\pi_1 \not\geq \pi_2$.*

Um die Sicherheit von π_1 zu beweisen, muss man hier die Definition des idealen Protokolls π_2 ändern, sodass der Angreifer ebenfalls diese Information erhält. Das neue Protokoll π_2' ist in Abbildung 13.4 gezeigt.

Mit dieser Änderung kann man tatsächlich $\pi_1 \geq \pi_2'$ beweisen, sofern das eingesetzte Verschlüsselungsverfahren IND-CPA-sicher ist.

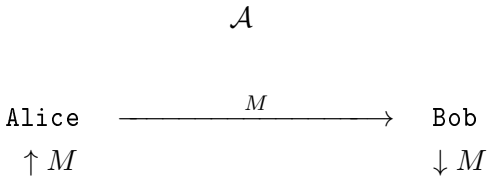


Abbildung 13.2: Das ideale Protokoll: ein sicherer Kanal. Der Angreifer $\mathcal{A}$ erhält *keinerlei* Information über M.

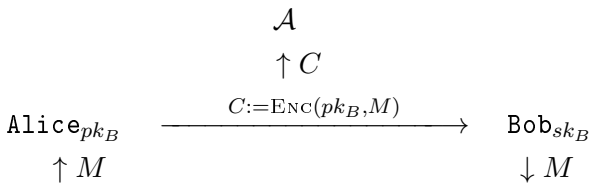


Abbildung 13.3: Die Implementierung eines sicheren Kanals durch Verschlüsselung.

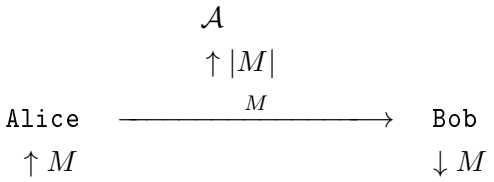


Abbildung 13.4: Die abgeschwächte Idealisierung π_2', eines sicheren Kanals.

Um die Sicherheit von π_1 zu zeigen, haben wir also nicht etwa π_1 geändert, sondern nur unsere Anforderungen von π_2 zu π_2' abgeschwächt.

Dennoch bietet die kryptographische Herangehensweise einige Vorteile:

- Die Formulierung von Sicherheitszielen wird deutlich vereinfacht. Anstelle des Aufstellens einer Liste von nachzuweisenden Eigenschaften wie beim Security-Ansatz, wird hier nur ein ideales Protokoll formuliert, das durch das reale Protokoll angenähert werden muss.

- Die Relation $\geq$ erlaubt auch die modulare Analyse von größeren Protokollen. Es gilt nämlich das folgende Theorem:

Theorem 13.4 (Kompositionstheorem, informell)**.** *Sei π^τ ein Protokoll, das ein Unterprotokoll τ benutzt. Sei weiter ρ ein Protokoll mit $\rho \geq \tau$, und sei π^ρ das Protokoll, welches ρ statt τ als Unterprotokoll benutzt. Dann gilt $\pi^\rho \geq \pi^\tau$.*

Mit diesem Werkzeug lässt sich nämlich das Protokoll π^τ mit einem *idealen* Unterprotokoll τ analysieren. Gelingt hier ein Beweis, dass π^τ ein größeres, ideales Protokoll π' implementiert (also $\pi^\tau \geq \pi'$), dann gilt sofort $\pi^\rho \geq \pi^\tau \geq \pi'$.

Auf der anderen Seite ist $\geq$ jedoch technisch sehr schwer zu handhaben. Deshalb werden größere Protokolle hauptsächlich mit dem Security-Ansatz untersucht.

Kapitel 14

Implementierungsprobleme

In den bisherigen Kapiteln haben wir das Thema „Sicherheit“ hauptsächlich aus einem kryptographischen Blickwinkel betrachtet und eine Vielzahl von kryptographischen Primitiven vorgestellt.

In diesem Kapitel wollen wir uns nun mit der einer anderen Seite der Sicherheit befassen: Der Sicherheit bzw. Unsicherheit von Software. Wir betrachten Sicherheitslücken in Software, wie sie täglich von Computerviren und Ähnlichem ausgenutzt werden. Solche Sicherheitslücken entstehen fast immer durch kleine oder große Schlampereien bei der Implementierung.

Die „Common Vulnerabilites and Exposures“ (CVE) ist eine öffentlich zugängliche Liste bekannter Schwachstellen. Sie ist unter http://cve.mitre.org/cve/ erreichbar, und zählte im Dezember 2013 knapp 60.000 Einträge. Die amerikanische „National Vulnerabilities Database“ (NVD, http://nvd.nist.gov/) des „National Institute for Standards and Technology“ (NIST) bietet eine Suchfunktion in dieser Datenbank, inklusive einfacher statistischer Anfragen. Das „Open Web Application Security Project“ (OWASP, https://www.owasp.org/) erstellt alle drei Jahre eine Top-Ten-Liste der Sicherheitslücken in Web-Anwendungen.

Wir stellen im Folgenden einige übliche Angriffstechniken von Hackern auf anfällige Software vor. Wir werden uns jedoch auch kurz mit Implementierungsproblemen von kryptographischen Operationen befassen.

14.1 Buffer Overflows

In einigen Programmiersprachen (allen voran C und C++) erfolgen Zugriffe auf Puffer (oder Arrays/Felder) ohne eine Überprüfung der Größe der Puffer. Z.B. liefert folgendes C-Programm keinen Fehler:

```
#include <stdio.h>

char greeting[8] = "Hello, ";
char greeted[6] = "World";

int main() {
        printf("%c\n", greeting[8]);
        return 0;
}
```

Dabei hat in diesem Beispiel das Feld `greeting` nur 8 Elemente, die von 0 bis 7 durchnummeriert sind.[1] Ein Zugriff auf das Element mit der Nummer 8 ist also eigentlich nicht

[1] In C erhalten Strings immer noch ein terminierendes Null-Symbol \0, das das Ende des Strings markiert. Deshalb benötigt die Zeichenkette `Hello,`, die aus sieben Zeichen besteht, dennoch 8 Byte Speicher-

möglich, die Rückgabe bestenfalls undefiniert. Dennoch löst das Programm keinen Fehler aus, sondern gibt den Buchstaben „W“ aus.[2]

Dies liegt an der Implementierung von Puffern in C: Ein Puffer oder Feld ist in C äquivalent zu einem Zeiger auf das erste Pufferelement (Index 0). Die Elemente des Puffers sind dann unmittelbar hintereinander angeordnet. Um die Speicheradresse des i-ten Elements (Index $i-1$) zu bestimmen, wird daher der Platzbedarf der vorherigen $i-1$ Elemente berechnet und dieser Wert zur Startadresse des Puffers hinzuaddiert. Diese Berechnung erfolgt im Allgemeinen ohne Abgleich mit der Größe des Puffers.

Im obigen Beispiel ist `greeting` im Wesentlichen ein Zeiger auf einen Speicherbereich, in dem die Zeichen `Hello, ` hintereinander abgelegt sind. Der Zugriff auf `greeting[8]` erfolgt, in dem der Platzbedarf von 8 `char`s zum Zeiger `greeting` hinzuaddiert werden.

Da der Compiler in diesem Beispiel die beiden Speicherbereiche für die Zeichenketten `Hello, ` und `World` hintereinander angeordnet hat, liegt 8 Positionen hinter dem Speicherbereich `greeting` der Buchstabe `W` aus der Zeichenkette `World`. Das Speicherlayout wird in Abbildung 14.1 gezeigt.

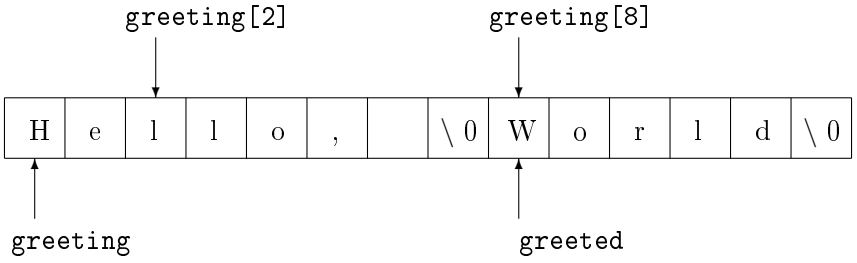


Abbildung 14.1: Anordnung der zwei Speicherbereiche `greeting` und `greeted` in unserem Beispiel. Ein Zugriff auf `greeting[8]` addiert den Speicherplatzbedarf von 8 `char`s zu dem Zeiger `greeting`. Beim Zugriff wird deshalb tatsächlich auf `greeted[0]` zugegriffen.

Ein Schreibzugriff auf `greeting[8]` liefert in diesem Beispiel ebenfalls keinen Fehler, selbiges gilt Schreibzugriffe auf `greeting[9]`, `greeting[10]`, usw. bis zumindest `greeting[12]`.

Dieses Verhalten führt dazu, dass ganze Speicherbereiche überschrieben werden. Wir betrachten hierzu das folgende Beispiel:

```
char name[8] = "World";
char greeting[8] = "Hello, ";

int main() {
        printf("What's your name?\n");
        scanf("%s", name);
        printf("%s%s\n", greeting, name);
        return 0;
}
```

Dieses Programm liest zunächst den Namen des Benutzers mittels der Funktion `scanf` in den Speicherbereich `name` ein, und gibt dann die zwei Strings `greeting` und `name` aus. Ist der Name des Benutzers jedoch länger als 7 Zeichen, dann überschreibt die Funktion `scanf` nicht nur den Speicherbereich der Variable `name`, sondern auch den Speicherbereich des Strings `greeting`.

platz. Analog benötigt die Zeichenkette `World` (5 Zeichen) 6 Byte.

[2] Kompiliert mit GCC 4.6.1

Im Allgemeinen wird die Funktion `scanf` so viel Speicherplatz überschreiben, wie sie zum Speichern der eingegebenen Daten benötigt. Bietet der bereitgestellte Puffer nicht genug Speicherplatz, so wird `scanf` auf den dahinterliegenden Speicherbereich zugreifen und diesen überschreiben.[3] Der Puffer „läuft also über“. Man bezeichnet so etwas deshalb als „Buffer Overflow“.

Dieses unerwünschte Verhalten kann ein Angreifer ausnutzen um gezielt bestimmte Daten im Arbeitsspeicher des ausgeführten Programms zu überschreiben. Befindet sich der übergelaufene Puffer auf dem Stack des Programms, dann kann der Angreifer mit dieser Technik sogar die Rücksprungadresse überschreiben und somit den Programmfluss lenken.

Hat der Angreifer zuvor eigenen Maschinencode (d.h. Prozessorinstruktionen) in den Arbeitsspeicher des Programms geschrieben, dann kann der Angreifer somit eigenen Programmcode auf dem Prozessor ausführen lassen.

Da diese Angriffstechnik über Jahre hinweg genutzt wurde, wurden inzwischen eine ganze Reihe von Gegenmaßnahmen entwickelt. Im Folgenden wollen wir einige dieser Gegenmaßnahmen vorstellen.[4]

Eine offensichtliche Gegenmaßnahme ist das vollständige Verhindern von Buffer Overflows. Hierzu kann man z.B.:

- vor jedem Schreibzugriff auf einen Puffer explizit die Puffergröße kontrollieren,
- Funktionen benutzen, die diese Kontrolle automatisch durchführen, (z.B. `strncat` oder `strncpy`; bei `scanf` kann man in obigem Beispiel `scanf("\%7s", name)` verwenden), oder
- eine Datenstruktur oder Programmiersprache verwenden, die beim Zugriff auf Puffer automatisch die Grenzen überprüft (z.B. Arrays in Java).

Da die erste Methode sehr anfällig für menschliche Vergesslichkeit oder Bequemlichkeit ist, sind die zweite oder dritte Maßnahme hier eindeutig vorzuziehen.

Diese Gegenmaßnahmen sind jedoch nicht immer anwendbar. Z.B. existieren einige weit verbreitete und sehr umfangreiche Programme mit mehreren Millionen Zeilen Quellcode, die noch ohne derartige Gegenmaßnahmen implementiert wurden. Den Quellcode dieser Programme zu überarbeiten ist praktisch kaum umsetzbar. Deshalb wurden auch eine Reihe von ad-hoc-Gegenmaßnahmen entwickelt. Hierzu zählen Stack Canaries, die sogenannte Data Execution Prevention und die Address Space Layout Randomization.

Stack Canaries werden von modernen Compilern in den generierten Maschinencode eingebettet. Hierbei handelt es sich um zufällige Dummy-Zahlen, die vor Rücksprungadressen auf dem Stack platziert werden. Tritt ein Buffer Overflow auf bei dem die Rücksprungadresse überschrieben wird, so muss dieser Buffer Overflow auch den Stack Canary, der zwischen Puffer und Rücksprungadresse liegt, überschreiben.

Der Compiler fügt vor jedem Rücksprung im generierten Code noch einige Befehle ein, die überprüfen ob der Stack Canary verändert wurde. Ist dies der Fall, so wird das Programm beendet. Ist der Stack Canary unverändert, so geht das Programm davon aus, dass kein Buffer Overflow auftrat und setzt die Ausführung fort.

[3] Selbiges gilt für eine Vielzahl anderer Funktionen in C, z.B. `strcat` zum Konkatenieren von zwei Strings, `strcpy` zum Kopieren von Strings, uvm.

[4] Es gibt aber zahlreiche Abwandlungen des gezeigten Angriffs, die diese Schutzmaßnahmen umgehen und deshalb auch heute noch funktionsfähig sind.

Bei der Data Execution Prevention, die vom Prozessor unterstützt und vom Betriebssystem aktiviert werden muss, erzwingt der Prozessor eine Trennung von Code- und Speicherbereichen. Daten in Speicherbereichen können dann nicht als Programmcode interpretiert werden, und Daten in Code-Bereichen können nicht überschrieben werden. Dadurch kann der Angreifer den von ihm eingeschleusten Code nicht ausführen lassen.

Bei der Address Space Layout Randomization (auch als Speicherverwürfelung bezeichnet) platziert das Betriebssystem die Speicherbereiche des Programms nicht deterministisch, sondern zufällig. Um den eingeschleusten Code auszuführen, muss der Angreifer nämlich die Rücksprungadresse auf dem Stack mit der Adresse des eingeschleusten Codes überschreiben. Wegen der zufälligen Platzierung der Speicherbereiche kann der Angreifer diese Adresse jedoch nicht kennen.

14.2 SQL-Injection

SQL ist eine weit verbreitete Sprache zur Formulierung von Datenbankabfragen. Zum Beispiel bewirkt die Abfrage

```
SELECT * FROM cd WHERE interpret = "Fall Out Boy";
```

die Rückgabe aller Zeilen in der Tabelle `cd`, in denen als Interpret „Fall Out Boy“ angegeben ist. Nun könnte diese Tabelle in einer Datenbank eines Online-Musikshops liegen. Dieser Online-Musikshop bietet dem Nutzer eine Suchfunktion. Sucht der Nutzer nach CDs von „Fall Out Boy“, dann wird z.B. obige Anfrage an die Datenbank geschickt. Sucht der Nutzer stattdessen nach dem Album „Folie à Deux“, so wird stattdessen die Anfrage

```
SELECT * FROM cd WHERE album = "Folie à Deux";
```

an die Datenbank geschickt.

Eine naheliegende Implementierung zur Generierung solcher Datenbankabfragen in der Programmiersprache PHP ist z.B. Folgende.

```
$alb = $_GET['album'];
sql_query($db,"SELECT * FROM cd WHERE album = \"$alb\";");
```

Hierbei enthält die Variable `$_GET['album']` die Benutzereingabe. Diese wird zunächst in die Variable `$alb` kopiert. Der String

```
"SELECT * FROM cd WHERE album = \"$alb\";"
```

wird automatisch in eine Konkatenation des Strings „`SELECT * FROM cd WHERE album = "`“, dem Inhalt von `$alb` und des Strings „ `\";`“ umgesetzt. Das Ergebnis dieser Konkatenation wird dann durch die Funktion `sql_query` als Abfrage an die Datenbank geschickt.

Leider erlaubte diese einfache Implementierung einem Angreifer, selbst festgelegte Befehle an die Datenbank zu senden. Hierfür muss er nur – anstelle eines Albums – Strings wie „`\"; DROP TABLE cd; #`“ in die Suchmaske eingeben.

Die Stringkonkatenation führt dann zur Abfrage

$$\underbrace{\texttt{SELECT * FROM cd WHERE album = "}}_{\text{Konstant}}\underbrace{\texttt{"; DROP TABLE cd; \#}}_{\text{Benutzereingabe}}\underbrace{\texttt{" ;}}_{\text{Konstant}},$$

die an die Datenbank geschickt wird. Die Datenbank interpretiert diese Abfrage als zwei Anweisungen:[5]

[5] Das #-Symbol leitet einen Kommentar ein. Dadurch wird verhindert, dass die folgenden Zeichen „";“ einen Syntaxfehler auslösen.

1. die Suche nach Alben, deren Name das leere Wort ist, und

2. die Anweisung, die Tabelle „`cd`“ zu löschen.

Beide Anweisungen werden von der Datenbank ausgeführt, und so wird die Tabelle „`cd`“ tatsächlich gelöscht. Der Angreifer hat also eigene Befehle in die Datenbankabfrage „injiziert“, daher rührt die Bezeichnung „SQL-Injection“ für solche Sicherheitslücken.

SQL-Injection-Angriffe können aber noch ungleich gefährlicher werden, wenn die Software besondere Funktionen wie das Ausführen von Kommandozeilenbefehlen oder das Erstellen von Dateien erlaubt. Ersteres ist z.B. bei Microsoft-SQL-Servern der Fall, Letzteres z.B. bei MySQL-Servern.

Solche Angriffe kann man z.B. mit den folgenden Methoden verhindern:

- Gründliche Überprüfung der Benutzereingabe, bevor diese an die Datenbank geschickt wird. Sinnvoll wäre z.B. eine Überprüfung, ob die Benutzereingabe nur aus Buchstaben, Zahlen und Leerzeichen besteht. (Dies kann allerdings unnötig restriktiv sein. Z.B. könnte die Eingabe des Zeichens à dadurch zurückgewiesen werden, obwohl es ein Album mit einem solchen Namen gibt.)

- Das „Escapen“ von Sonderzeichen in der Benutzereingabe, so dass diese als Bestandteil des Strings interpretiert werden. Hierfür gibt es in den APIs der Datenbank häufig besondere Funktionen, z.B. `mysql_real_escape_string`.

- Das Benutzen von Prepared Statements. Hierbei wird zunächst ein Abfrage mit Platzhalter an die Datenbank geschickt: `SELECT * FROM cd WHERE album=?;` In einem zweiten Schritt wird dann die Benutzereingabe an die Datenbank übergeben. Hierdurch wird verhindert, dass die Benutzereingabe von der Datenbank als Befehl interpretiert und ausgeführt wird.

14.3 Cross Site Scripting

Cross-Site-Scripting-Sicherheitslücken (auch CSS- oder XSS-Lücken) funktionieren konzeptuell ähnlich wie SQL-Injections, tauchen jedoch in einer etwas anderen Umgebung auf.

Bei Cross Site Scripting injiziert ein Angreifer nicht eigene SQL-Befehle in eine Datenbankabfrage, sondern stattdessen eigene HTML-Anweisungen in eine Webseite. Da HTML-Anweisungen auch JavaScript-Programme enthalten können, die ihrerseits die im Browser des Opfers dargestellte Webseite vollständig kontrollieren können, kann der Angreifer hiermit die Kontrolle über den Inhalt des Browserfensters erlangen. Dies klingt zunächst harmlos, kann aber sehr ernsthafte Konsequenzen haben.

- Gelingt es dem Angreifer z.B. eigenen JavaScript-Code auf einer Login-Seite zu platzieren, so kann der Angreifer hiermit die von einem Opfer eingegebenen Login-Daten abgreifen. Doch auch wenn ein Benutzer bereits eingeloggt ist, kann der Angreifer mit entsprechendem JavaScript-Code das Login-Cookie des Benutzers kopieren und damit selbst unter der Identität des Nutzers auf der Webseite surfen.

- Java-Script-Würmer können in Sozialen Netzwerken auf Pinnwände oder Ähnliches geschrieben werden, wo sie von anderen Benutzern eingesehen werden. Sie werden daraufhin im Browser des Opfers ausgeführt und kopieren sich selbstständig auf die Pinnwand des Betrachters. Beispielsweise wurde MySpace im Jahr 2005 zeitweilig wegen einem solchen Wurm abgeschaltet.[3, 7].

- Die Computer-Forensik-Software X-Way Forensics bietet z.B. die Möglichkeit, ihre Ergebnisse als HTML-Seite darzustellen. Bettet ein Angreifer z.B. einen öffnenden HTML-Kommentar in die Windows-Registry-Key ein, und einen schließenden Kommentar in einen späteren Registry-Key, dann werden die dazwischen liegenden Registry-Keys dem Nutzer nicht angezeigt. Diese Sicherheitslücke wurde 2011 bekannt und behoben [13].

- Cross Site Scripting kann auch als Implementierung für den CRIME-Angriff auf TLS verwendet werden (siehe Kapitel 8.3.2.3).

Eine typische Gegenmaßnahmen gegen XSS-Angriffe ist es, Daten entsprechend zu escapen, damit sie nicht als HTML-Befehle interpretiert werden können. Die Programmiersprache PHP bietet hierfür z.B. die Funktionen `htmlspecialchars` und `htmlentities`, die Zeichen mit spezieller Bedeutung in HTML (z.B. <, >, und \") ersetzen.

14.4 Denial of Service

Denial of Service (DOS) Angriffe zielen, anders als die bisher vorgestellten Angriffe, nicht darauf ab, selbstbestimmten Code auf einem fremden Server ausführen zu lassen. Ziel ist es bei solchen Angriffen nur einen bestimmten Dienst lahmzulegen, z.B. das Online-Banking einer Bank oder einen Onlineshop. Bei solchen Angriffen handelt es sich jedoch nicht immer nur um digitalen Vandalismus.

In einigen Fällen versuchten Kriminelle mit solchen Angriffen z.B. Geld von Onlineshops zu erpressen. Die Gruppe Anonymous protestierte auf diese Art auch dagegen, dass einige Banken Spenden an Wikileaks nicht mehr ausführten.

14.4.1 DDOS

Eine technisch einfache Möglichkeit für DOS-Angriffe ist es, den Server, der den Dienst erbringt, mit Anfragen zu überhäufen. Dann kann der Server nämlich aufgrund seiner beschränkten Ressourcen nur einen kleinen Teil der Anfragen bearbeiten, so dass der Service für die eigentlichen Nutzer effektiv nicht mehr zur Verfügung steht.

Je nach Ausstattung des Servers werden dabei die Datenleitungen zum Server überlastet, das Betriebssystem des Servers, dass die Netzwerkverbindungen verwalten muss, oder der Prozessor des Servers, der für die Bearbeitung der Anfragen Rechenleistung erbringen muss.

Im Allgemeinen werden für eine solche Überlastung jedoch eine ganze Reihe von Angreifern benötigt, die gemeinsam versuchen den Server zu überlasten. Deshalb werden solche Angriffe auch als „Distributed Denial Of Service“-Angriffe (DDOS-Angriffe) bezeichnet. In der Realität werden solche Angriffe üblicherweise durch Bot-Netze ausgeführt. Ein Botnetz ist ein Netzwerk von mit Viren oder anderer Schadsoftware infizierter Computer. Auf den Befehl des Autors der Schadsoftware hin führen diese bestimmte Aufgaben aus, z.B. eben eine DDOS-Attacke auf ein bestimmtes Ziel.

Eine Variante von solchen DDOS-Angriffen, die darauf abzielt, die Verwaltung von Netzwerkverbindungen durch das Betriebssystem zu überlasten, ist das sogenannte „SYN-Flooding“. SYN-Pakete werden verwendet, um TCP-Verbindungen aufzubauen. Erhält ein Server ein SYN-Paket mit einer bestimmten Sequenznummer, und möchte der Server die Verbindung akzeptieren, so antwortet er auf das SYN-Paket mit einem SYN+ACK-Paket und einer eigenen Sequenznummer. Unterdessen speichert er einige Informationen zur noch nicht vollständig aufgebauten Verbindung, z.B. die IP-Adresse des Clients, den vom Client verwendeten TCP-Port und die selbst vergebene Sequenznummer. Bei einem normalen

Verbindungsaufbau antwortet der Client dann noch einmal mit einem „ACK“-Paket, um den Aufbau der TCP-Verbindung abzuschließen. Abbildung 14.2 zeigt ein Beispiel eines normalen TCP-Verbindungsaufbaus.

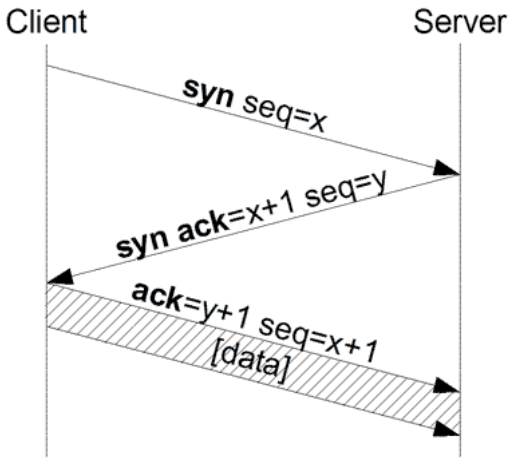


Abbildung 14.2: Beispiel eines TCP-Verbindungsaufbaus. Quelle: http://commons.wikimedia.org/wiki/File:300px-Tcp-handshake.png Lizenz: CC-BY-SA 3.0 Unported Autor: vermutlich Caos

Beim SYN-Flooding jedoch sendet der Client niemals das abschließende ACK-Paket, sondern sendet weitere SYN-Pakete um noch mehr TCP-Verbindungen aufzubauen. Dadurch zwingt der Angreifer den Server dazu, Informationen zu jeder noch nicht vollständig aufgebauten Verbindung zu speichern. Da der zur Verfügung stehende Speicherplatz jedoch beschränkt ist, lässt sich damit die Verwaltung der Netzwerkverbindungen im Betriebssystem belasten. Arbeiten mehrere Angreifer zusammen, so kann man den Server hiermit überlasten.

14.5 Andere DOS-Angriffe

DDOS-Angriffe sind jedoch nicht die einzige Möglichkeit, einen Server lahmzulegen. Wir betrachten nun noch einen anderen beispielhaften Angriff, der technisch etwas interessanter ist.

Einige Sprachen, wie z.B. PHP, Python und JavaScript bieten die Möglichkeit Strings als Indizes von Arrays zu verwenden. Wir haben dies bereits in unserem Beispiel zu SQL-Injection auf Seite 14.2 gesehen. Dort wurde auf das (vordefinierte) Array `$_GET` zugegriffen. Dieses Array wird von PHP automatisch mit allen Parametern gefüllt, die der Client dem Server beim Aufruf einer Webseite per `GET`-Methode übergibt. Z.B. wird bei der Anfrage

`http://www.example.com/?q=mad+magazine`

der Wert „`mad magazine`“ unter dem Schlüssel „`q`“ in das (vordefinierte) Array `$_GET` eingefügt. Es ist also `$_GET['q'] == 'mad magazine'`.

Intern wird hierfür eine Dictionary-Datenstruktur bzw. eine Hashtabelle verwendet. Solchen Datenstrukturen liegt ein gewöhnliches (mit Zahlen indiziertes) Array $\overline{\texttt{\$\_GET}}$ der Länge l sowie eine Hashfunktion h zugrunde. Um einen Wert v_1 (z.B. “`mad magazine`“) unter einem Index (oder Schlüssel) s_1 (z.B. „`q`“) zu speichern, wird der Schlüssel s_1 zunächst zu $h(s_1)$ gehasht. Das Ergebnis wird modulo l reduziert, und das Paar (s_1, v_1) an der Position $h(s_1) \mod l$ im Array gespeichert.

Die Hashfunktion h bietet jedoch keine kryptographische Kollisionsresistenz, da kryptographische Hashfunktionen zu aufwendig auszuwerten sind. Deshalb kann es vorkommen,

dass für ein zweites Schlüssel-Wert-Paar (s_2, v_2) gilt, dass $h(s_2) \equiv h(s_1) \pmod{l}$ ist. In diesem Fall müssen beide Paare (s_1, v_1) und (s_2, v_2) an der selben Stelle im Array gespeichert werden. Deshalb werden beide Paare üblicherweise in eine verkettete Liste eingefügt, die dann an dieser Stelle im Array `$_GET` gespeichert wird.[6] Werden weitere Paare $(s_3, v_3), \ldots)$ mit $h(s_3) \equiv h(s_2) \pmod{l}$, ..., so werden diese Paare ebenfalls in die verkettete Liste eingefügt.

Tritt keine Hashkollision auf, so benötigt man für n Zugriffe auf eine solche Dictionary-Struktur nur $\mathcal{O}(n)$ Operationen. Treten jedoch *nur* Kollisionen auf, d.h. für alle i, j gilt $h(s_i) \mod l = h(s_j) \mod l$, dann werden alle Elemente der Datenstruktur in *nur einer* verketteten Liste gespeichert. Für n Zugriffe werden dann $\Omega(n^2)$ Operationen benötigt.

Dies kann sich ein Angreifer zunutze machen. Da h keine kryptographische Kollisionsresistenz bietet, kann der Angreifer eine große Anzahl entsprechender Schlüssel erzeugen, so dass diese in der Dictionary-Struktur des Servers alle in der selben Liste gespeichert werden. Dadurch kann der Angreifer gezielt eine hohe Last auf dem Server erzeugen.

Weitere Informationen zu diesem Angriff finden sich im Vortrag [6].

[6] Eine Andere Strategie ist, (s_2, v_2) an der nachfolgenden Stelle im Array zu speichern, sofern diese frei ist.

Literaturverzeichnis

[1] Matt Blaze, Whitefield Diffie, Ronald L. Rivest, Bruce Schneier, Tsutomu Shimomura, Eric Thompson, and Michael Wiener. Minimal key lengths for symmetric ciphers to provide adequate commercial security. http://www.fortify.net/related/cryptographers.html, January 1996.

[2] Larry Ewing. Tux der pingiun, erstellt mit "the gimp". lewing@isc.tamu.edu, 1996.

[3] Samy Kamkar. I'm popoular, 2005. http://namb.la/popular/.

[4] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[5] Auguste Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, 9:5–38, Januar 1883.

[6] Alexander Klink and Julian Wälde. Efficient denial of service attacks on web application platforms, 2011. Aufzeichnung online verfügbar: https://events.ccc.de/congress/2011/wiki/Documentation.

[7] Sophos Ltd. Detailed analysis of js/spacehero-a. http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/JS~Spacehero-A/detailed-analysis.aspx.

[8] U. M. Matsui. Linear cryptanalysis method for des cipher. In *Advances in Cryptology — Proceedings of EUROCRYPT '93*, pages 386–397. Springer LNCS, 1994.

[9] National Institute of Standards and Computer Security Resource Center Technology. Modes development. http://csrc.nist.gov/groups/ST/toolkit/BCM/modes_development.html.

[10] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer Berlin Heidelberg, 1992.

[11] Larry Stockmeyer. Planar 3-colorability is polynomial complete. *SIGACT News*, 5(3):19–25, July 1973.

[12] Martin Thoma. https://github.com/MartinThoma/LaTeX-examples/, 2013.

[13] Martin Wundram. Antiforensik, 2011. Aufzeichnung online verfügbar:http://events.ccc.de/congress/2011/wiki/Documentation.