Universität Hamburg
Fachbereich Informatik

Hinweise für das Erscheinungsbild von Seminar-, Studien- und Bachelor-, Master- und Diplomarbeiten

am Arbeitsbereich Sicherheit in Verteilten Systemen (SVS)

Prof. Dr. Hannes Federrath

4. Dezember 2014

(Muster für das Deckblatt: siehe letzte Seite dieser Hinweise)

Zusammenfassung

Für den eiligen Leser sollen auf etwa einer halben, maximal einer Seite die wichtigsten Inhalte, Erkenntnisse, Neuerungen bzw. Ergebnisse der Arbeit beschrieben werden.

Durch eine solche Zusammenfassung (im engl. auch Abstract genannt) am Anfang der Arbeit wird die Arbeit deutlich aufgewertet. Hier sollte vermittelt werden, warum der Leser die Arbeit lesen sollte.

Inhaltsverzeichnis

1 Einleitung

2 Funktionsweise

Hier sollte noch viel viel mehr stehen...

2.1 Verwendete Verfahren

2.1.1 XOR

Die XOR-(Exklusiv-ODER-)Verknüpfung ist ein bitweiser Operator, der für zwei unterschiedliche Eingangsbits 1 ergibt und ansonsten 0. Seine besondere Bedeutung für die Kryptographie liegt in dem Zusammenhang $K \oplus K = 0$ und somit $(M \oplus K) \oplus K = M$, dass heißt zweifache Verknüpfung von M mit dem Bitstring K ergibt wiederum M. Diese zweifache Verknüpfung lässt sich als Ver- und Entschlüsselung interpretieren, wie es beispielsweise beim OneTimePad geschieht [Sch06].

2.1.2 Hashfunktion

Eine Hashfunktion ist eine Funktion, die eine Eingabe variabler Länge auf einen String fester Länge abbildet.

In der Kryptographie werden insbesondere Einweg-Hashfunktionen eingesetzt. Bei dieser Art von Hashfunktionen ist es leicht aus einer Eingabe den Hashwert zu berechnen, jedoch sehr schwer zu einem gegebenen Hashwert eine Eingabe zu finden die auf diesen Wert abgebildet wird [Sch06]. Beispiele für heute verwendete Hashfunktionen sind MD5 und SHA256.

2.1.3 Blockchiffre

Bei Blockchiffren handelt es sich um symmetrische Verschlüsselungsalgorithmen, die Nachrichten in Blöcken fester Größe verschlüsseln. Es gilt $\text{Enc}_K(M) = C$ und $\text{Dec}_K(C) = M$. Hierbei steht M für die Nachricht, K für den Schlüssel, der verwendet wird, C für den Chiffretext und Enc bzw. Dec für die Ver- bzw. Entschlüsselung[Sch06].

Enc/Dec(?) **mit DTE abstimmen**

2.1.4 Betriebsmodi für Blockchiffren

Kryptographische Modi sind Verfahren, die das Verschlüsseln einer Nachricht per Blockchiffre beschreiben. Sie verknüpfen die Blockchiffre normalerweise mit einer Rückkopplung und wenigen einfachen Operationen. Beispiele für Modi sind CBC (Cipher Block Chaining - XOR-Verknüpfung des zuletzt erhaltenen Chiffretexts mit dem nächsten Klartextblock vor seiner Verschlüsselung) oder CTR (Counter Mode - Verschlüsselung eines Initialisierungsvektors und eines blockweise erhöhten Zählers mit dem Schlüssel und anschließende XOR-Verknüpfung des erhaltenen Zwischenschlüssels mit dem Klartextblock) [Sch06].

2.1.5 Password Based Key Derivation Function

Password Based Key Derivation Functions leiten aus einem Passwort (und möglichen anderen Parametern) einen Schlüssel ab, der dann beispielsweise in symmetrischen Algorithmen weiter verwendet werden kann.

Derzeitige Empfehlung ist die Verwendung von PBKDF2. Innerhalb dieses Algorithmus wird mehrfach eine pseudozufällige Funktion auf die Eingangswerte angewendet. Durch diese Erhöhung der Berechnungszeit steigt der Aufwand für Brute-Force-Angriffe auf Verschlüsselungen, die dieses Verfahren nutzen, stark an [Kal00].

2.2 Notationen

In den folgenden Kapiteln, in denen näher auf die Funktionsweise der Honey Encryption eingegegangen wird, werden folgende Notationen verwendet:

$\overset{<r>}{=}$ steht für eine nicht-deterministische Zuweisung. Dies kann entweder komplett zufällig geschehen (wie bei Belegung von zufälligen Bitstrings) oder zumindestens vom Zufall mitbestimmt werden (wie bei der Kodierung einer Nachricht durch DTE).

$a \oplus b$ steht für die XOR-Verknüpfung von a und b.

$S||T$ steht für die Konkatenierung der Zeichenketten S und T.

$S[1..n]$ steht für die Nutzung der ersten n Zeichen von S.

ϵ steht für die leere Zeichenkette.

3 DTE

Die DTE[1], Abkürzung für *distribution-transforming encoder*, dient zum Abbilden einer Nachricht M aus dem Message Space $\mathcal{M}$ auf einen Seed S aus dem Seed Space $\mathcal{S}$. Gleichermaßen soll sie die Möglichkeit bieten, von einem Seed auf die ursprüngliche Nachricht abzubilden. Eine DTE ist also ein Tupel von Algorithmen

$$DTE = (encode, decode)$$

wobei $encode$ einen meist randomisierten Algorithmus der Form $\mathcal{M} \rightarrow \mathcal{S}$ und $decode$ einen deterministischen Algorithmus der Form $\mathcal{S} \rightarrow \mathcal{M}$ beschreibt.

Ein DTE-Schema $(encode, decode)$ wird als *korrekt* bezeichnet, wenn für jede Nachricht $M \in \mathcal{M}$, die mit $encode$ in den Seed Space $\mathcal{S}$ und mit $decode$ anschließend wieder in den Message Space $\mathcal{M}$ abgebildet wird, das Resultat wieder die ursprüngliche Nachricht M ist. Formal kann dies geschrieben werden als

$$P(decode(encode(M)) = M) = 1 \qquad f.a. M \in \mathcal{M}$$

wobei P ein Maß für die Wahrscheinlichkeit für das in den Klammern stehende Ereignis ist.

Bei der Konstruktion einer DTE ist die Korrektheit nicht das einzige Kriterium, welches es zu beachten gilt. Wichtig ist ebenfalls, die Verteilung der Wahrscheinlichkeiten der Nachrichten im Message Space zu kennen. Sie wird mit p_m bezeichnet. Entsprechend dieser Wahrscheinlichkeiten wird einer Nachricht eine Anzahl von Seeds zur Kodierung zugewiesen. Je wahrscheinlicher eine Nachricht ist, desto mehr Seeds werden ihr zugewiesen.

Bei der Verschlüsselung einer Nachricht $M \in \mathcal{M}$ weist der Algorithmus $encode$ dieser Nachricht einen Seed entsprechend ihrer Wahrscheinlichkeit zu. Da es jedoch mehr als einen Seed zu einer Nachricht geben kann, handelt es sich bei $encode$ um einen randomisierten Algorithmus, der zufällig und gleichverteilt einen der möglichen Seeds auswählt. Da diese Kodierungs-Methode nicht deterministisch ist, handelt es sich bei $encode$ um keine Funktion oder Abbildung im mathematischen Sinne. Der Begriff *Abbildung* ist dennoch eine passende Umschreibung für das Vorgehen zur Kodierung der Nachricht, mit einem Seed als Resultat.

Jede Nachricht kann also durch mehr als einen Seed dargestellt werden, allerdings verweist jeder Seed auf genau eine Nachricht (zu sehen in Abbildung 1).

Seed Space und Message Space anzeigen und Pfeile einmalen

Abbildung 1: Relationen zwischen $\mathcal{M}$ und $\mathcal{S}$

Somit ist leicht zu erkennen, dass es sich bei $decode$ um einen deterministischen Algorithmus handelt. Der Begriff *Abbildung* wäre in diesem Fall auch mathematisch korrekt.

[1] Nach http://leo.org lässt sich *DTE* mit *Verteilungsumwandelnde Codiermaschine* übersetzen, weshalb wir in dieser Ausarbeitung den femininen Genus für den Fachbegriff verwenden.

Wie schon beschrieben, sollte die Wahrscheinlichkeitsverteilung der Nachrichten so gut wie möglich durch die DTE nachgeahmt werden. Juels und Ristenpart [JR14b] führen hierfür eine neue Verteilung p_d ein — die Verteilung, die die DTE über den Message Space $\mathcal{M}$ erzeugt. Diese wird definiert als die Wahrscheinlichkeit, dass zufällig und gleichverteilt gewählte Seeds durch die $decode$-Funktion auf eine bestimmte Nachricht M abgebildet wird.

$$p_d(M) = P(M' = M : U \xleftarrow{\$} \mathcal{S}; M' \leftarrow decode(S))$$

$U \xleftarrow{\$} \mathcal{S}$ bedeutet, dass die Seeds **u**niform, also gleichverteilt sind.

Eine intuitivere Definition von p_d wäre

$$p_d(M) = \frac{|\mathcal{S}_M|}{|\mathcal{S}|}$$

Dabei sei $\mathcal{S}_M$ die Menge aller Seeds, die durch den $decode$-Algorithmus wieder auf M abgebildet werden. Diese Definition bezieht sich auf die diskrete Gleichverteilung des Seed Spaces und die dadurch anwendbare Laplace-Formel.

Bei der Erstellung einer DTE sollte darauf geachtet werden, dass $p_d \approx p_m$ gilt. Bei einer perfekten DTE würde eine Gleichheit der Verteilungen gelten.

3.1 Generierung einer DTE mithilfe der Inversionsmethode

Ein mögliches Vorgehen zur Erstellung einer DTE, die von Juels und Ristenpart [JR14b] vorgeschlagen wird, ist die Nutzung der sogenannten Inverse Sampling Methode, zu Deutsch Inversionsmethode. Sie wird in der Informatik und Stochastik angewendet, "um aus gleichverteilten Zufallszahlen andere Wahrscheinlichkeitsverteilungen zu erzeugen." [Wik14] Meist wird dieses Verfahren in der Informatik für die Simulation von Zufallsvariablen, wie beispielsweise dem Monte-Carlo-Verfahren, verwendet. Dabei wird einer Rechteckverteilung $R(0, 1)$ eine neue Wahrscheinlichkeitsverteilung zugewiesen. So kann ein Computer Zufallszahlen erzeugen, die in einer beliebigen, neuen Verteilung liegen, als die normalerweise von ihm generierbaren Zahlen im Intervall $[0, 1)$.

Wie schon beschrieben wurde, muss für die Generierung einer DTE p_m, also die Wahrscheinlichkeitsverteilung für die Nachrichten des Message Spaces, bekannt sein. Nach den Regeln der Stochastik hat jede Nachricht eine Wahrscheinlichkeit im Intervall $(0, 1)$, wobei die Summe aller Wahrscheinlichkeiten gleich 1 sein muss. Es sei hier explizit darauf hingewiesen, dass die Intervallränder 0 und 1 als Wahrscheinlichkeiten für Nachrichten ungeeignet sind. Gilt nämlich für eine Nachricht $M \in \mathcal{M}$ $P(M) = 0$, dann ist das Vorkommen dieser Nachricht nicht möglich und sollte somit gar nicht beachtet werden. Ein Vorkommen im Message Space ist damit überflüssig. Gilt andererseits für M $P(M) = 1$, so ist dies die einzig mögliche Nachricht. Dann ist es nicht sinnvoll, Honey Encryption zu verwenden, da ein potentieller Angreifer zum eindeutigen Entschlüsseln nicht einmal den Ciphertext kennen müsste. Der Angriff wäre trivial.

Eine DTE wird mit diesem Vorwissen nun wie folgt erstellt: Es wird die Verteilungsfunktion F_m der Verteilung p_m genutzt. Gegeben sei dafür eine Ordnung der Nachrichten im Message Space $\mathcal{M} = \{M_1, \ldots, M_{|\mathcal{M}|}\}$. Die Verteilungsfunktion einer Nachricht $F_m(M_i)$ ist nun die Summe der Wahrscheinlichkeiten der ersten i Nachrichten.

$$F_m(M_i) = \sum_{k=1}^{i} P(M_k)$$

Um die Grenzen festzulegen, sei $F_m(M_0) = 0$ und logischerweise $F_m(M_{|\mathcal{M}|}) = 1$. Eine Visualisierung für eine Verteilungsfunktion solcher Art ist in Abbildung 2 zu sehen.

Visualisierung

Abbildung 2: Eine Visualisierung einer Verteilungsfunktion F_m

Sei nun der Wertebereich aller Seeds $S \in \mathcal{S}$ das Intervall $[0, 1)$. Ein Seed S wird dann in seine Ursprungsnachricht zurückgeführt (*decode*-Algorithmus), indem die Nachricht M_i gefunden wird, für die $F_m(M_{i-1}) \leq S < F_m(M_i)$ gilt. Wenn als Beispiel der Seed zwischen der Summe der ersten 5 und 6 Nachrichtenwahrscheinlichkeiten liegt, dann gibt der *decode*-Algorithmus als ursprüngliche Nachricht zurück. Eine anders geschriebene und leichter in Programmen umsetzbare Schreibweise der Übersetzung von Seed in Nachricht ist

$$min_i\{F_m(M_i) > S\}$$

Anschaulich sei ein Maßband der Länge 1 betrachtet, auf der die Nachrichten entsprechend ihrer Wahrscheinlichkeiten aufeinanderfolgend und disjunkt aufgetragen (Abbildung 3) wurden. Ein Seed liegt nun irgendwo auf der Länge des Maßbandes. An dieser Stelle liegt auch eine Nachricht M_i, die dann als ursprüngliche Nachricht ausgegeben wird. Liegt der Seed dabei auf der Grenze zweier Nachrichten, wird die weiter rechts liegende zurückgeliefert.

Maßband mit Nachrichten und 0 links und 1 rechts

Abbildung 3: Eine anschauliche Darstellung des *decode*-Algorithmus

Es ist an der anschaulichen Darstellung unschwer zu erkennen, dass die Verteilung p_m in der DTE wieder zu finden ist. Die Wahrscheinlichkeit einer Nachricht wird durch ihre Breite dargestellt. Da die Seeds gleichverteilt sind, ist es leichter, einen besonders breiten Abschnitt zu treffen, als einen schmaleren.

Bisher wurde der *decode*-Algorithmus betrachtet. Der *encode*-Algorithmus ist ähnlich anschaulich zu erklären. Dieses Mal wird eine Nachricht gewählt, die auf dem Maßband liegt. Der Bereich an Werten auf dem Maßband, der von dieser Nachricht abgedeckt wird, wird als Grundlage für eine Auswahl eines Seeds verwendet. Dabei wird zufällig gleichverteilt ein Wert aus diesem Bereich ausgewählt.

Auf die Verteilungsfunktion bezogen ist ein Seed, der aus der Nachricht M_i generiert wird, eine reelle Zahl im Intervall $[F_m(M_{i-1}), F_m(M_i))$. Die Auswahl aus diesem Intervall lässt sich mithilfe eines Computers recht einfach bewerkstelligen, da dieser im Normalfall reelle Zufallszahlen im Intervall $[0, 1)$ generieren kann — Sicherheitsaspekte und -bedenken bezüglich vom Computer generierter Zufallszahlen betrachten wir in dieser Ausarbeitung nicht.

Auch hier ist klar, dass es mehr mögliche Seeds für eine Nachricht geben kann, je wahrscheinlicher diese Nachricht ist. In diese Richtung ist die Eigenschaft einer guten DTE ebenfalls gegeben.

Um die erzeugte DTE praktisch anwenden zu können, muss noch das Problem der Übersetzung der stetigen Werte zwischen $[0, 1)$ in die diskreten Werte von in Binärzahlen angegebenen Seeds des Seed Spaces, welcher tatsächlich verwendet wird. Die Anzahl der Seeds sollte so gewählt werden, dass die Abweichungen der relativen Seed-Bereiche einer Nachricht in beiden Repräsentationen möglichst klein sind. Je größer die Abweichungen nämlich sind, desto mehr verzerrt sich die Verteilung der DTE. Die erstrebenswerten Eigenschaften einer guten DTE wären damit nicht mehr gegeben und es könnte zu einem Sicherheitseinbruch der Honey Encryption mit dieser DTE kommen.

3.2 Eine DTE für private RSA-Schlüssel

Das folgende Beispiel ist entnommen aus [JR14b].

Bei RSA handelt es sich um einen asymmetrischen Verschlüsselungsalgorithmus, d.h. das Verfahren beruht auf einem öffentlichen und einem privaten Schlüssel. Zur Generierung der Schlüssel (heutiger Stand: 2000 Bit [**?**]) wählt man zwei große Primzahlen p und q und errechnet aus ihnen öffentlichen und privaten Schlüssel (zu Details siehe [Sch06]). RSA wird beispielsweise bei SSL/TLS oder SSH eingesetzt. Für die Anwendung von Honey Encryption ist jedoch nur der zweite Fall geeignet, denn bei SSL/TLS ist der öffentliche Schlüssel des Servers bekannt und so ließe sich leicht nachprüfen, ob der richtige private Schlüssel entschlüsselt wurde.

Bei SSH lässt sich Honey Encryption jedoch anwenden. In diesem Fall wird der öffentliche Schlüssel zum Zweck der Authentifizierung auf dem Server gespeichert, den man erreichen möchte (und steht somit dem Angreifer nicht zur Verfügung). Der private Schlüssel (genauer p und q zusammen mit weiteren berechneten Werten, um die Ver- bzw. Entschlüsselung nach dem Chinesischen Restsatz zu erleichtern) wird verschlüsselt auf dem Clientsystem gespeichert.

Zur Erstellung einer DTE für RSA-Schlüssel muss betrachtet werden, wie die Primzahlen p und q (im Folgenden werden Primzahlen aus dem Intervall $[2^{l-1}, 2^l)$ gefordert) bezogen werden. Normalerweise werden zufällige Zahlen aus dem Intervall gezogen und durch einen Primzahltest (Miller-Rabin-Test) wird überprüft, ob es sich dabei um Primzahlen handelt. DIes wird solange wiederholt, bis man zwei Primzahlen gefunden hat.

Ein naiver Ansatz für eine DTE wäre es, die beiden gefundenen Primzahlen als $(l-2)$-Bitstrings zu dekodieren (die auf jeden Fall vorhandene führende 1 ist implizit und wird ausgelassen). Da jedoch bei der Entschlüsselung dann auch nicht Primzahlen entstehen würden (und zwar nach dem Primzahlsatz mit etwa Wahrscheinlichkeit $1 - \frac{1}{l}$), würde ein Angreifer viele Ausgaben von vornherein als ffalschërkennen können.

Daher schlägt [JR14b] ein anderes Vorgehen vor. Zur Kodierung von p und q wird ein Vektor vonj t zufälligen $(l-2)$-Bitstrings angelegt. Per Primzahltest werden die Zahlen überprüft. Die ersten beiden Primzahlen in dem Vektor werden durch p und q ersetzt. Enthält der Vektor nur eine Primzahl, so wird diese durch p und die letzte Zahl durch q ersetzt. Enthält der Vektor keine Primzahlen, so ersetzen p und q die letzten beiden Zahlen. Beim Dekodieren werden die ersten beiden Primzahlen im Vektor ausgegeben. Wenn der Vektor keine zwei Primzahlen enthält so werden fest kodierte Proimzahlen ausgegeben.

Es lässt sich zeigen, dass ein Angreifer, der versucht p und q per Brute-Force-Angriff zu erhalten, eine Erfolgswahrscheinlichkeit von höchstens $(1 - \frac{2}{3l})^{t-1}$ besitzt (ebenfalls [JR14b]). Damit lässt sich also die Erfolgswahrscheinlichkeit des Angreifers durch Nutzung eines größeren Vektors verringern (allerdings auf Kosten von größerem Speicherplatzbedarfs).

Ein anderer Ansatz, der in [JR14b] erwähnt wird, wäre die Kodierung des Seeds/Keys, der zur Initialisierung des Zufallszahlengenerators verwendet wurde, um p und q zu generieren. Eine DTE wäre trivial, da es sich bei dem Seed/Key im Allgemeinen um einen kurzen, zufällig gleichverteilten String handelt.

4 Verschlüsselungsschema

Nachdem eine Nachricht aus dem Message Space - wie in Kapitel 3 beschrieben - durch eine DTE auf den Seed Space abgebildet wurde, folgt die Verschlüsselung des Ergebnisses. Hierfür schlägt [JR14b] zwei verschiedene Vorgehensweisen vor. Die unter Verwendung dieser Vorgehensweisen entstehenden Honey Encryption-Schemata werden im Folgenden dargestellt.

4.1 Hashbasierte Verschlüsselung

Das erste Verfahren nutzt zur Verschlüsselung die XOR-Verknüpfung des aus der Nachricht erhaltenen Seeds S mit einem Hash des Schlüssels K (Abbildung 4).

Nach der Kodierung der Nachricht durch die DTE wird der Schlüssel zum Erschweren von Brute-Force-Angriffen durch eine Password Based Key Derivation Function auf den Bitstring K_D abgebildet. Es wird ein zufälliger Bitstring R gewählt, der zusammen mit K_D durch die Hashfunktion HF auf H abgebildet wird. Dieser zufällige Bitstring sorgt dafür, dass auch bei der Verschlüsselung gleicher Nachrichten mit gleichem Schlüssel unterschiedliche Chiffretexte entstehen. Er kann je nach Anwendung in der Länge k variiert werden. Der errechnete Hash H wird nun mit dem im ersten Schritt erhaltenen Seed S XOR-verknüpft und bildet den Chiffretext C. Dieser kann nun zusammen mit dem Bitstring R gespeichert oder übertragen werden.

$\text{HEnc}_{\text{Hash}}(M, K)$
- $S \xleftarrow{<r>} \text{DTE}(M)$
- $K_D = \text{PBKDF}(K)$
- $R \xleftarrow{<r>} \{0,1\}^k$
- $H = \text{HF}(K_D, R)$
- $C = H \oplus S$
- Return (C, R)

Abbildung 4: Hashbasierte Verschlüsselung

$\text{HDec}_{\text{Hash}}((C, R), K)$
- $K_D = \text{PBKDF}(K)$
- $H = \text{HF}(K_D, R)$
- $S = H \oplus C$
- $M = \text{DTE}^{-1}(S)$
- Return M

Abbildung 5: Hashbasierte Entschlüsselung

Zur Entschlüsselung wird das Verfahren in ähnlicher Weise durchlaufen (Abbildung 5). Der Schlüssel K wird wie bei der Verschlüsselung durch eine Password Based Key Derivation Function auf den Bitstring K_D abgebildet. Zusammen mit dem übergebenen Bitstring R wird durch HF der Hash H gebildet. Durch eine XOR-Verknüpfung von H mit C erhält man den ursprünglichen Seed S. Dieser kann dann durch die DTE dekodiert werden und es ergibt sich wieder die Nachricht M.

4.2 Auf Blockchiffren basierte Verschlüsselung

Für die Verschlüsselung können jedoch auch Blockchiffren genutzt werden. Im Folgenden wird das Schema unter Nutzung einer Blockchiffre mit dem CTR-Modus skizziert. Andere Betriebsmodi sollten ebenfalls nutzbar sein (so erwähnt [JR14b] explizit den CBC-Modus), jedoch müssten im Einzelfall Einschränkungen, wie Blocklänge oder eventuell notwendiges Padding, beachtet werden.

HEnc $_{\text{Block}}(M, K)$
- $S \xleftarrow{<r>} \text{DTE}(M)$
- $R \xleftarrow{<r>} \{0,1\}^k$
- $K' = \text{HF}(K, R)$
- $P = \epsilon$
- For $i = 1$ to $\left\lceil \frac{|S|}{n} \right\rceil$
  - $P = P \,\|\, \text{Enc}(K', i)$
- $C = P[1..|S|] \oplus S$
- Return (C, R)

Abbildung 6: Verschlüsselung mit Blockchiffre (CTR-Modus)

HDec $_{\text{Block}}((C, R), K)$
- $K' = \text{HF}(K, R)$
- $P = \epsilon$
- For $i = 1$ to $\left\lceil \frac{|S|}{n} \right\rceil$
  - $P = P \,\|\, \text{Enc}(K', i)$
- $S = P[1..|S|] \oplus C$
- $M = \text{DTE}^{-1}(S)$
- Return M

Abbildung 7: Entschlüsselung mit Blockchiffre (CTR-Modus)

Sowohl bei der Ver- als auch bei der Entschlüsselung wird zum Erzeugen eines Schlüsselstroms (gemäß dem CTR-Modus für Blockchiffren) gleich vorgegangen (sieht man einmal von der Erzeugung des zufälligen Bitstrings R der Länge k während der Verschlüsselung ab, der aus dem gleichen Grund wie bei dem hashbasierten Verfahren genutzt wird). Aus R und dem Schlüssel K wird durch eine Hashfunktion der Schlüssel K' generiert. Der Schlüsselstrom P wird leer initialisiert. Nun wird eine Schleife so oft durchlaufen wie Blöcke der Länge n (Blocklänge der verwendeten Blockchiffre) notwendig sind, um mindestens die Länge eines Seedwertes ($|S|$) zu erreichen. In jedem Durchlauf wird an P die Blockverschlüsselung von K' und dem Schleifenzähler i angehängt.

Bei der Verschlüsselung (Abbildung 6) wird der aus der Kodierung entstandene Seed S mit den $|S|$ ersten Bits des wie bereits beschrieben berechneten Schlüsselstrom P XOR-verknüpft und bildet so den Chiffretext C, der zusammen mit R nun gespeichert oder übertragen werden kann.

Bei der Entschlüsselung (Abbildung 7) müssen nach Berechnung des Schlüsselstroms P nur noch die $|S|$ ersten Bits von P mit C XOR-verknüpft werden und man erhält den ursprünglichen Seed S. Dieser kann dann durch die DTE dekodiert werden und es ergibt sich wieder die Nachricht M.

5 Möglichkeiten und Probleme

Durch die enorm hohe Sicherheit der Verschlüsselung ergeben sich vor allem Anwendungsfälle, bei denen hochsensible Daten verschlüsselt werden sollen. Juels und Ristenpart ([JR14b, JR14a]) geben dafür einige Beispiele, wie das Verschlüsseln von RSA-Schlüsseln oder Kreditkartennummern. Ebenfalls könnten Passwort-Safes/-Manager mit Honey Encryption vor Zugriffen von außen geschützt werden. Generell ist diese Verschlüsselungsmethode auf strukturierte Daten anwendbar, von denen die Generierung und der Aufbau bekannt sind. Dies ist schließlich, wie in Abschnitt 3 beschrieben, notwendig zur Konstruktion einer sicheren und invertierbaren DTE. Dementsprechend ist Honey Encryption nicht oder nur eingeschränkt für Freitext, wie Notizen oder E-Mails, geeignet. Ein Grund dafür ist die Tatsache, dass die Menge aller Nachrichten bekannt sein muss. Diese ist bei Freitext quasi unbegrenzt.

Ebenfalls sollte beachtet werden, dass für jeden Anwendungsbereich, in dem Honey Encrytion genutzt werden soll, eine eigene DTE konstruiert werden muss. Ein universaler Ansatz existiert dazu nicht. Der Nutzen der Verschlüsselung mit Honey Encryption sollte also verhältnismäßig groß im Vergleich zum Aufwand der Erstellung stehen. Dieser Aufwand ist in jedem Fall enorm. Es muss nicht nur die Menge aller Nachrichten bekannt sein, sondern auch die Verteilung der Wahrscheinlichkeiten dieser. Bei Passwort-Managern beispielsweise ist die Menge der vom Nutzer verschlüsselten Passwörter meist abhängig vom Nutzer selbst. Falls der Nutzer nämlich keine zufällig generierten Passwörter verwendet, sind Passwörter, die Teile des Namens, des Geburtsdatums, des Namens der Lieblingsband, des Haustieres oder anderer persönlicher Daten beinhalten, sehr wahrscheinlich. Die Verteilung der Wahrscheinlichkeiten der Nachrichten, in diesem Fall der Passwörter des Nutzers, sind also von Nutzer zu Nutzer unterschiedlich. Diese muss aber zur Konstruktion einer guten DTE bekannt sein. Was passiert aber, wenn die DTE nicht optimal gewählt ist und es für einen Angreifer erkennbar ist, dass er die richtige Lösung, zum Beispiel die Passwörter des Nutzers im Safe, gefunden hat? Dann fällt die Methode auf eine Brute-Force-Attacke zurück. Der Angreifer muss also jede Passwort-Kombination für den Safe ausprobieren, bis er das richtige Passwort gefunden hat. Die Sicherheit von Honey Encryption ist dann ähnlich der Sicherheit von herkömmlichen Verschlüsselungsmethoden, wie zum Beispiel AES. Zwar gibt es bei Honey Encryption mehrere Passwörter, die aufgrund der Hash-Funktion von Passwort auf Seed-Space die gleiche Nachricht erzeugen, was die Wahrscheinlichkeit erhöht, dass ein solches Passwort bei einem Brute-Force-Angriff gefunden wird. Allerdings wird im Gegensatz zu beispielsweise AES nie direkt angegeben, ob das eingegebene Passwort korrekt ist. Der Angreifer hat also einen enormen Mehraufwand, wenn er überprüfen will, ob er ein korrektes Passwort gefunden hat. Schließlich muss er bei jedem Versuch erneut überprüfen, ob die resultierende Nachricht sinnvoll ist oder nicht. Je schlechter die DTE gewählt ist, desto einfacher fällt es dem Angreifer. Dennoch ist es schwer, die Nachrichten auf Plausibilität zu prüfen.

Neben der Erstellung der DTE ist auch die Speicherung der DTE ein Problem. Da sie eine Funktion ist, die für jeden Anwendungsfall neu erstellt werden muss, muss sie auch für jeden Anwendungsfall gespeichert werden. Kann eine Nachricht nicht so einfach aus einem Seed berechnet werden, so, muss eine Datenstruktur gespeichert werden, die von einer Nachricht auf einen Seed und umgekehrt abbilden kann. Diese tabellen-ähnliche Struktur ist für einen ausreichend großen

Message Space sehr umfangreich. Ist die Anzahl der Nachrichten im Message Space gleich n und die Speicherung einer Nachricht m Bits teuer, dann brauchen wir *mindestens*

$$(\lceil \log_2(n) \rceil + m) \cdot n$$

Bits zur Speicherung der Datenstruktur.
Beispiel: Es existieren $n = 2^{16} = 65536$ mögliche Nachrichten, die gleichverteilt auf den Seed Space abgebildet werden sollen. Dabei soll jeder Nachricht nur ein Seed zugewiesen werden. Für jede Nachricht ist zudem ein String mit maximal 10 ASCII-Zeichen, also ein Speicherbedarf von $m = 10$ Bytes, also 80 Bits. Die Datenstruktur zum Speichern aller Nachrichten — mit ihrem Index als Seed — wird dann mindestens

$$\begin{aligned}(\lceil \log_2(65536) \rceil + 80) \cdot 65536 &= (16 + 80) \cdot 65536 \\ &= 96 \cdot 65536 \\ &= 6291456\end{aligned}$$

Bits benötigen. Das sind 786432 Bytes, also ungefähr 800 Kilobytes. Um eine Nachricht aus dem Message Space zu verschlüsseln, wird also Speicherplatz für die DTE (inklusive ihrer Datenstruktur), den resultierenden Ciphertext, sowie die Hashfunktion und eine eventuell vorhandene Verschlüsselungsfunktion (nach Abschnitt 4) benötigt. Ein String von maximal 10 ASCII-Zeichen aus der Menge der Nachrichten zu verschlüsseln, wird zu einem fast ein Megabyte großen Paket. Gerade das Verschicken von geheimen Nachrichten wird dadurch extrem erschwert, da dieses Paket bei neuen Anwendungsfällen erneut erstellt und übermittelt werden muss. Dieses vergleichsweise große Datenverhältnis zwischen *Verfahren zum Ver- und Entschlüsseln* und *Ciphertext* relativiert sich aber bei steigender Anzahl von übermittelten Ciphertexten. Ist die Länge der Nachrichten relativ lang, werden aber auf kurze Seeds abgebildet, sind die Ciphertexte kürzer als bei anderen Verschlüsselungsverfahren. Mit steigender Anzahl an zu speichernden oder zu übertragenden Ciphertexten teilt sich die Größe der Ver- und Entschlüsselungsmethoden auf die einzelnen Cyphertexte auf und wir erhalten möglicherweise zum Schluss immer noch kleinere zu speichernde Datenmengen als bei anderen Verschlüsselungsverfahren mit gleicher Ciphertextanzahl.

Ein anderer Bereich für Probleme von Honey Encryption ist die Erstellung der Hashfunktion, die vom Key Space auf den Seed Space abbildet. Da die Größe des Seed Spaces für jede Menge von Nachrichten unterschiedlich ist, muss auch die Hashfunktion an die Größe des Seed Spaces angepasst werden. Wie bei jeder Hashfunktion muss ebenfalls sicher gestellt werden, dass Werte aus dem Definitionsbereich durch die Funktion nur in den erlaubten Wertebereich abgebildet werden. Außerhalb des Definitionsbereichs ist das Resultat der Hashfunktion egal. Wird zum Beispiel ein vierstelliger Pin zum Ver- und Entschlüsseln von einer Nachricht verlangt, so muss unsere Hashfunktion lediglich die gültige Menge aller Zeichenkette — “0000” bis “9999” — berücksichtigen. Alle Werte außerhalb dieses Bereichs, beispielsweise Zeichenketten mit mehr oder weniger Zeichen, müssen nicht beachtet werden. Dies stellt zwar eine kleine Erleichterung für den Ersteller der Hashfunktion dar, allerdings ist die Generierung einer guten Hashfunktion, die die Anforderungen aus Abschnitt 4 erfüllen, wie schon das Erstellen einer guten DTE, extrem aufwändig.

Einer der größten Vorteile von Honey Encryption ist gleichzeitig auch einer ihrer größten Nachteile. Die Tatsache, dass unter Eingabe jedes möglichen Schlüssels ein plausibler Klartext angezeigt wird, könnte dem Nutzererlebnis schaden. Gibt nämlich ein Nutzer das Passwort falsch ein, bekommt er normalerweise den Hinweis, dass die Eingabe nicht korrekt ist. Bei Honey Encryption wird dem Nutzer diese Hilfestellung nicht gegeben. Der Nutzer weiß gar nicht, ob er das Passwort richtig eingegeben hat, bzw. ob er überhaupt im Besitz des richtigen Passwortes ist (falls ihm das oben erwähnte Paket von einer anderen Person geschickt wurde — der Schlüssel ist dabei beispielsweise mündlich übertragen worden). Diese Problematik lässt sich nicht einfach lösen. Juels und Ristenpart stellen insgesamt drei Ansätze vor ([Jue14, JR14b]), die sogenannte *Typo-Safety* zu gewährleisten. Einerseits könnte zum Ciphertext eine Prüfsumme des Passwortes gespeichert werden. So würden nach einer Fehleingabe des Passwortes die Prüfsummen nicht mehr übereinstimmen und der Nutzer könnte entsprechend gewarnt werden. Logischerweise schränkt dies jedoch den Key Space ein, da die Anzahl der möglichen Schlüssel dezimiert wird. Diese Möglichkeit bietet also weniger Schutz für eine bessere Benutzbarkeit. Ein weiterer Ansatz ist die Überprüfung der entschlüsselten Daten. Dies klappt aber nur bei Daten wie Kreditkarten, bei denen der Anbieter überprüfen kann, ob die Nummer eine gültige und zum Kunden gehörende ist. Damit der Anbieter sicherstellen kann, dass ein Kunde und nicht ein Angreifer auf den Dienst zugreifen, wäre es nach Juels und Ristenpart möglich, beispielsweise die ersten beiden Ziffern der Kreditkartennummer als Klartext im Honey Encryption Verfahren zu speichern. Nutzt ein Angreifer dann irgendeine Kreditkartennummer, weiß der Anbieter, dass es sich hierbei um keinen Tippfehler des Passwortes, sondern um einen Angriff von einem unberechtigten Dritten handelt. Läge ein Schreibfehler vor, stimmten wenigstens die ersten beiden Ziffern überein und der Anbieter kann den Kunden nach einer erneuten Eingabe des Passwortes fragen. Hierbei wird der Message Space verkleinert, was auch zu weniger Sicherheit führt. Wie auch schon der Ansatz davor sollte diese Methode dementsprechend mit Vorsicht angewendet werden. Der dritte und letzte Vorschlag seitens der beiden Autoren ist es, eine Farbe mit den anderen Informationen zusammen zu verschlüsseln. Diese Farbe wird ebenfalls wieder hergestellt, wenn das richtige Passwort eingegeben wurde, und der Ciphertext wird zu einer anderen Farbe entschlüsselt, wenn es eine Fehleingabe gab. Es wird die Stärke von Honey Encryption angewendet, um dem Nutzer einen Hinweis darauf zu geben, dass er den richtigen Schlüssel eingegeben hat. Der Nutzer muss sich allerdings die ursprüngliche Farbe merken, bzw. sie wieder erkennen. Dies ist jedoch leichter, als sich den viel komplizierteren Klartext der Nachricht zu merken, da hierbei das visuelle Gedächtnis des Nutzers angesprochen wird. Diese Methode bringt allerdings nicht nur Vorteile. So wird dem Nutzer bei jeder Verschlüsselung, die er vornimmt, eine neue Farbe präsentiert. Bei der Unmenge an Logins, die wir heutzutage tätigen, könnte das für Verwirrung sorgen.

6 Fazit

Zusammenfassend lässt sich sagen, dass Honey Encryption mit ihrem neuen Ansatz, Nachrichten zu verschlüsseln, ein interessantes neues Forschungsfeld eröffnet. Sie ermöglicht es, auch kurze Schlüssel wie nutzergenerierte Passwörter zu verwenden und stellt mit ihrer Sicherheit eine enorm starke Verschlüsselung dar.

Allerdings überwiegen im jetzigen Forschungszustand die Nachteile und Probleme (aufgezeigt in Abschnitt 5). Die Generierung der einzelnen Teile des Honey Encryption Schemas ist noch zu aufwändig, um das Verfahren großflächig anzuwenden. Die Benutzung muss durch Gewährleistung der *Typo-Safety* gegeben sein und die Anwendungsgebiete der Verschlüsselung sollten weiter erforscht werden.

Die zukünftigen Forschungsbestreben sollten darauf abzielen, das Erstellen einer sicheren DTE zu vereinfachen. Ein Anfang wäre es, eine öffentlich zugängliche Sammlung von Honey Encryption Schemata für bestimmte Anwendungsfälle zur Verfügung zu stellen. Damit könnten Entwickler, die strukturierte Daten verschlüsseln wollen, auf diese zurückgreifen und müssten lediglich die Einbindung in ihr bestehendes System berücksichtigen. Eine immer größer werdende Sammlung führt somit zu weniger Aufwand für Entwickler und damit zu einer größeren und vor allem schnelleren Verbreitung des Verfahrens. Mit Hilfe modernerer Technik, statistischen Verfahren und weiteren Hilfsmitteln könnte es vielleicht sogar möglich sein, DTEs und Hashfunktionen automatisiert generieren zu lassen. Die Probleme, die dabei auftreten und die Bedingungen, die an eine gute DTE gestellt werden, stehen der Entwicklung momentan noch im Weg. Allerdings gibt es Fortschritte in der Analyse und Generierung von natürlicher Sprache, die bei der Entwicklung der genannten Systeme behilflich sein können ([Jue14]).

Ein weiteres Forschungsziel sollte es sein, die Möglichkeiten und Anwendungsbereiche von Honey Encryption zu erweitern. So könnte eine spannende Forschungsfrage sein, in wie weit es nicht doch möglich wäre, Klartexte zu verschlüsseln. Eine Idee könnte sein, die korrekte Nachricht als Grundlage für einen sehr viel größeren Message Space zu nutzen. Dabei würde eine Veränderung von Wörtern ohne Beeinträchtigung des Sinnzusammenhanges wichtig sein. Ein Beispiel wäre die Abwandlung des Satzes “Der geheime Treffpunkt ist Hamburg” in Nachrichten wie “Der geheime Treffpunkt ist Berlin”. Der Sinngehalt bliebe der gleiche, ein potentieller Angreifer könnte dann nicht entscheiden, in welcher Stadt nun der *geheime Treffpunkt* liegt. Die automatische Generierung solcher sinnverwandten Sätze wäre hier allerdings die erste Anlaufstelle, da der Message Space entsprechend groß gewählt werden muss. Dies ist so bei heutigem Kenntnisstand noch nicht möglich, allerdings könnte es in Zukunft solch ein Verfahren geben. Die Analyse der Sicherheit eines solchen Ansatzes wäre dann in einer weiteren wissenschaftlichen Arbeit zu klären.

Honey Encryption ist ein interessanter Ansatz, der noch viel Platz für Fortschritt und Verbesserung bietet. Die gebotene Sicherheit, die damit theoretisch möglich ist, sollte in der heutigen Zeit, mit immer wieder vorkommenden Passwort-Leaks und stärkeren Rechnern, Anreiz sein, weitere Forschung in dieser Richtung zu betreiben.

Literatur

[JR14a] Ari Juels and Thomas Ristenpart. Honey encryption - encryption beyond the brute-force barrier. *IEEE Security & Privacy*, 12/4:59–62, April 2014.

[JR14b] Ari Juels and Thomas Ristenpart. Honey encryption - security beyond the brute-force bound. In *EUROCRYPT 2014*, Kopenhagen, Dänemark, Mai 2014.

[Jue14] Ari Juels. The password that never was. http://crcs.seas.harvard.edu/event/ari-juels-the-password-that-never-was, 2014. Online; Letzter Zugriff: 27.10.2014.

[Kal00] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, 2000.

[Sch06] Bruce Schneier. *Angewandte Kryptographie - Der Klassiker. Protokolle, Algorithmen und Sourcecode in C*. Pearson Studium, München, 2006.

[Wik14] Wikipedia. Inversionsmethode — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?title=Inversionsmethode&oldid=134296036, 2014. Online; Letzter Zugriff: 30. November 2014.