Universität Hamburg
Fachbereich Informatik

Seminararbeit

Funktionsweise der Honey Encryption

vorgelegt von

Konstantin Kobs
geb. am 22. Februar 1993 in Geesthacht
Matrikelnummer 6414943
Studiengang Informatik

Tom Petersen
geb. am 13. Dezember 1990 in Hannover
Matrikelnummer 6359640
Studiengang Informatik

eingereicht am 06. Januar 2015

Betreuer: Prof. Dr.-Ing. Hannes Federrath

Inhaltsverzeichnis

1 Einleitung

Auf der EUROCRYPT 2014 wurde von Ari Juels und Thomas Ristenpart ein neues Verfahren zur Absicherung von Verschlüsselung unter Verwendung von schwachen Schlüsseln (wie nutzergewählten Passwörtern) vorgestellt, das die Autoren in Anlehnung an bereits bestehende, ähnliche Systeme als *Honey Encryption* bezeichnet haben.

Das Neuartige an diesem Verfahren ist, dass jede Entschlüsselung eines Ciphertextes unter einem zufälligen, nicht korrekten Schlüssel zu einer plausiblen Nachricht führt. Ein Angreifer, der keine weiteren Informationen über die Nachricht besitzt, kann die Nachricht so im Gegensatz zu bestehenden Verfahren nicht durch bloßes Ausprobieren aller möglichen Schlüssel knacken. Gerade nachdem in den letzten Jahren nach Angriffen auf Informationssysteme immer wieder Passwörter von Millionen von Nutzern bekannt geworden sind, erlangten Angreifer so relativ klare Vorstellungen von der Passwortauswahl der Nutzer. Honey Encryption kann hier einen Fortschritt in der Sicherheit passwort-basierter Verschlüsselung bieten.

In dieser Seminararbeit soll das Verfahren der Honey Encryption näher beleuchtet werden. Dazu wird in Abschnitt 3 ein grober Überblick über heute verwendete Verfahren und ihre Schwächen gegeben und anhand eines Beispiels in die Funktionsweise und die notwendigen Einzelschritte der Honey Encryption eingegangen. Abschnitt 4 und 5 erklären diese Einzelschritte dann ausführlicher und erläutern das Vorgehen auch anhand von weiteren, konkreten Beispielen. In Abschnitt 6 werden Einschränkungen betrachtet, die aus dem Verfahren selbst entstehen oder die bei seiner Anwendung beachtet werden müssen.

Da das Verfahren erst vor kurzer Zeit vorgestellt und noch keine relevante Sekundärliteratur zu diesem Thema veröffentlicht wurde, folgt diese Ausarbeitung in weiten Teilen der Argumentation von Juels und Ristenpart in [JR14a] bzw. [JR14b].

2 Notationen

In den folgenden Abschnitten, in denen auf die Funktionsweise der Honey Encryption eingegegangen wird, werden die unten stehenden Notationen verwendet. Diese stimmen nicht vollständig mit denen in der verwendeten Literatur überein.

$\mathcal{M}$	der Message Space, die Menge aller möglichen Nachrichten.
$\mathcal{K}$	der Key Space, die Menge aller möglichen Schlüssel.
$\mathcal{C}$	die Menge aller möglichen Ciphertexte.
$\mathcal{S}$	der Seed Space. Näheres dazu insbesondere in Abschnitt 4.
$P(X)$	die Wahrscheinlichkeit für das Eintreten des Ereignisses X.
$\overset{<r>}{=}$	eine nicht-deterministische Zuweisung. Dies kann entweder komplett zufällig geschehen (wie bei der Generierung von zufälligen Bitstrings) oder zumindestens vom Zufall mitbestimmt werden (wie bei der Kodierung einer Nachricht durch die DTE).
$a \oplus b$	die XOR-Verknüpfung von a und b.
$S\|\|T$	die Konkatenation der Zeichenketten S und T.
$S[1..n]$	die Nutzung der ersten n Zeichen von S.
ϵ	die leere Zeichenkette.

3 Funktionsweise

Honey-Objekte werden in der IT-Sicherheit in vielfacher Weise zum Aufdecken, Abwehren oder Untersuchen von Angriffen auf Systeme genutzt. Am bekanntesten dürften die Honeypots sein: Server oder Programme, die Systeme simulieren, um Informationen über das Verhalten von Angreifern zu erlangen oder Einbrüche aufzudecken. Aber auch weniger bekannte Verfahren wie Honeydocuments, Honeyfiles oder Honeywords sind in diesem Bereich anzusiedeln. Grundsätzlich geht es hier darum, ein echtes Objekt zwischen Täuschungen (den Honey-Objekten) zu verstecken.

All diese Verfahren verbinden zwei Eigenschaften: Ununterscheidbarkeit, d.h. Honey-Objekte sollten nur schwer vom echten Objekt differenzierbar sein, und Geheimhaltung, d.h. die bloße Kenntnis der Existenz von Honey-Objekten darf einem Angreifer auf das System keinen Vorteil bringen (frei nach dem Kerkhoffs'schen Prinzip: Nicht das Verfahren, sondern lediglich der Index des echten Objekts in der Liste aller Objekte ist geheim zu halten) [Jue14].

In [JR14b] stellen die Autoren Honey Encryption vor — ein neues Verfahren, welches die Nutzung von Honey-Objekten auf Passwort-basierte Verschlüsselung (*Password-Based Encryption*, PBE) anwendet.

3.1 Passwort-basierte Verschlüsselung und die Brute-Force Bound

Grundsätzlich besteht ein PBE-Schema aus einer Verschlüsselungsfunktion *Enc* und einer Entschlüsselungsfunktion *Dec*. Eine Nachricht M wird mit einem Schlüssel K durch die Verschlüsselungsfunktion in den Ciphertext C überführt: $\text{Enc}_K(M) = C$. Die Entschlüsselung erfolgt analog dazu: $\text{Dec}_K(C) = M$.

Ein Angreifer, der außer C keine weiteren Informationen besitzt, wird versuchen, per Brute-Force-Angriff (also durch rohes Durchprobieren aller möglichen Schlüssel) an die Nachricht M zu gelangen. Er wählt einen Schlüssel $K' \in \mathcal{K}$ und bildet $\text{Dec}_{K'}(C) = M'$. Durch die Nutzung von Authenticated Encryption (AE, z.B. Encrypt-then-MAC, siehe [BN00]) erfährt der Angreifer sofort, ob er den richtigen Schlüssel gefunden hat. Bei diesen Verfahren wird schon vor der Entschlüsselung anhand eines Message Authentication Codes (MAC) überprüft, ob die verschlüsselte Nachricht nicht verändert wurde und der Schlüssel stimmt. Aber auch bei nicht authentifizierter Verschlüsselung lässt sich in den meisten Fällen leicht herausfinden, ob der versuchte Schlüssel K' korrekt war, also $K' = K$ und damit auch $M' = M$ gilt. Dies kann aus unterschiedlichen Gründen möglich sein. Beispielsweise sei bekannt, dass M natürlichsprachig ist. Dadurch könnten viele entschlüsselte Nachrichten gleich verworfen werden. Andere Gründe können ein zum Teil bekanntes Nachrichtenformat (z.B. offene Dateiformate oder feste Dateiheader) oder andere bekannte Eigenschaften sein (z.B. könnte bekannt sein, dass M eine Primzahl darstellt, ...).

Dieser Brute-Force-Angriff wird dann zum Problem, wenn Schlüssel geringer Entropie[1] gewählt

[1] Die Entropie steht für den mittleren Informationsgehalt einer Nachricht. Wenn also ein Passwort häufig gewählt wird, so ist der Informationsgehalt und damit auch die Entropie des Passworts geringer, da es eher erwartet wird.

werden und Angreifer damit in vielen Fällen nur wenige Versuche benötigen, um den richtigen Schlüssel zu finden.[2] Durch Verfahren wie Salting (siehe [Sch06]) oder mehrfache Anwendung beispielsweise von Hashfunktionen bei der Zwischenschlüsselgenerierung (vergleiche [Kal00]) lässt sich diese Art von Angriffen zwar verlangsamen, aber nicht aufhalten. Es lässt sich zeigen, dass eine PBE-verschlüsselte Nachricht mit Wahrscheinlichkeit $\frac{q}{c \cdot 2^{\mu}}$ per Brute-Force entschlüsselt werden kann, wobei q für die Anzahl der Versuche, c als verfahrensabhängige Konstante und μ für die Min-Entropie der Passwortverteilung p_k steht. Diese Wahrscheinlichkeit wird in [JR14b] als *Brute-Force Bound* bezeichnet und wird weiterhin $\mu < 7$ für realistische Passwortverteilungen angegeben. Diese Grenze ist selbst bei Erhöhung von c durch oben erwähnte Verfahren sehr gering.

Würde man weiterhin einen (in Rechenzeit und Speicherplatz) unbeschränkten Angreifer annehmen, so würde die Nachricht auf jeden Fall geknackt werden. Honey Encryption bietet hierfür eine Gegenmaßnahme an.

3.2 Einführung in die Honey Encryption

> *„HE is designed to produce a ciphertext which, when decrypted with any of a number of incorrect keys, yields plausible-looking but bogus plaintexts called honey messages.“* [JR14b]

Durch diese Eigenschaft kann ein Angreifer, der per Brute-Force-Angriff vorgeht, nie sicher sein, ob die in jedem Fall erhaltene, korrekt aussehende Nachricht wirklich die vorher verschlüsselte Nachricht M ist. Dies gilt selbst für unbeschränkte Angreifer und insbesondere auch für Schlüssel geringer Entropie. Es lässt sich zeigen (und die Autoren tun dies in [JR14b] für einige konkrete Anwendungsfälle), dass die Wahrscheinlichkeit, die Nachricht per Brute-Force-Angriff auf den Ciphertext C zu knacken, bei korrekter Implementation des Verfahrens (bis auf einen vernachlässigbaren Summanden) nicht größer ist als die Wahrscheinlichkeit, M durch Entschlüsselung von C mit dem Schlüssel $K*$, der am wahrscheinlichsten in $\mathcal{K}$ ist, zu erhalten. Dies entspricht auch in etwa der Wahrscheinlichkeit, mit Auswahl von $M*$, der Nachricht mit der größten Wahrscheinlichkeit in $\mathcal{M}$, die richtige Nachricht M erhalten zu haben. Diese Aussage lässt sich jedoch auch ohne Kenntnis von C treffen, also hat der Angreifer durch Kenntnis von C keinen Vorteil erlangt.

Ein kleines Beispiel soll nun die grundsätzliche Funktionsweise darstellen (Abbildung 1). Angenommen, es soll der favorisierte RGB-Farbanteil verschlüsselt werden. Es sei bekannt, dass jeder zweite Mensch blau und jeder vierte jeweils rot oder grün am liebsten mag. Damit ist die Verteilung der Nachrichten p_m bekannt. Der erste Schritt besteht nun darin, den liebsten Farbanteil grün durch eine DTE (Distribution Transforming Encoder, näheres in Abschnitt 4) auf den

[2] So wird in [JR14a] beispielsweise der Diebstahl von 32 Millionen Klartext-Passwörtern von Kunden der Firma RockYou im Dezember 2009 erwähnt. Hierbei stellte sich heraus, dass in etwa einem Prozent der Fälle *123456* als Passwort gewählt worden war und auch andere ähnlich schwache Passwörter häufig vertreten waren.

sogenannten Seed Space $\mathcal{S}$ abbilden[3] zu lassen. Für grün erhält man so den Seed $S = 01$. Im zweiten Schritt wird S nun mit dem gewählten Schlüssel $K = 10$ XOR-verknüpft und es entsteht der Ciphertext $C = 01 \oplus 10 = 11$ (auf konkrete Verfahren zur Verschlüsselung wird in Abschnitt 5 eingegangen).

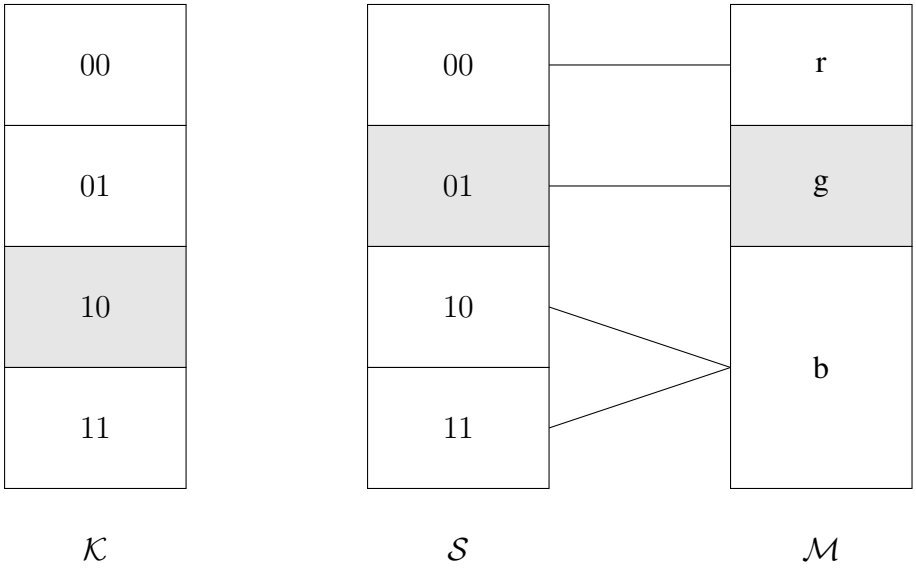

Abbildung 1: Visualisierung des Beispiels

Zur Entschlüsselung wird der Ciphertext C zunächst mit dem Schlüssel K XOR-verknüpft. Es entsteht der Seed $S = 11 \oplus 10 = 01$. Dieser lässt sich nun von der inversen DTE wieder auf den favorisierten Farbanteil abbilden und man erhält wiederum grün. Ein Angreifer, dem nur C vorliegt, kann nun versuchen, den Farbanteil per Brute-Force herauszufinden. Dazu wählt er einen Schlüssel $K' = 00$ (beispielsweise, weil dieser am häufigsten verwendet wird) und bildet $C \oplus K' = M' = 11$. Er erhält also blau. Ebenso gut hätte er rot oder auch grün erhalten können, aber in keinem Fall kann er sicher sein, die richtige Nachricht geknackt zu haben. Seine größte Erfolgswahrscheinlichkeit erhält er also dadurch, entweder C mit dem am häufigsten verwendeten Schlüssel zu entschlüsseln oder direkt die wahrscheinlichste Nachricht M (in diesem Fall also blau) anzunehmen.

3.3 Unterschiede zu bestehenden Verfahren

Um abgrenzen zu können, was Honey Encryption ausmacht, soll an dieser Stelle auf einige Verfahren eingegangen werden, die ähnlich arbeiten oder auf den ersten Blick ähnlich aussehen (die Beispiele sind entnommen aus [JR14a] und [JR14b]).

Das erste Verfahren, das betrachtet werden soll, ist das sogenannte One Time Pad. Hierbei handelt es sich um simple XOR-Verknüpfung der Nachricht mit einem mindestens ebenso langen,

[3] Abbildung ist hier nicht rein mathematisch zu verstehen, denn es handelt sich nicht zwingend um eine deterministische Relation.

zufälligen Schlüssel, der nur ein einziges Mal verwendet werden darf. Bei der Entschlüsselung lässt sich so je nach gewähltem Schlüssel jede andere (gleich lange) Nachricht mit gleicher Wahrscheinlichkeit erhalten. Bei dem One Time Pad handelt es sich jedoch nicht um ein Honey Encryption-Schema, denn erstens liefert die Entschlüsselung mit zufälligem Schlüssel nur in den wenigstens Fällen eine plausible Nachricht und zweitens werden auf jeden Fall Schlüssel hoher Entropie benötigt, Honey Encryption soll jedoch auch mit Schlüsseln geringer Entropie Sicherheit bieten.

Auch existierende AE- oder PBE-Schemata eignen sich, wie oben erklärt, nicht für die Honey Encryption, da die meisten entschlüsselten Nachrichten leicht als nicht plausibel verworfen werden können.

Als Letztes sei noch der Unterschied zu explizit gespeicherten Täuschobjekten erwähnt, wie sie beispielsweise in [JR13] verwendet werden. Bei diesen Verfahren werden neben der echten Nachricht $n - 1$ andere Honey Messages explizit gespeichert und (übertragen auf das hier betrachtete Gebiet) im Falle eines falschen Schlüssels bei der Entschlüsselung zurückgegeben. Im Gegensatz zur optimalen Honey Encryption ist die Erfolgswahrscheinlichkeit bei diesen Verfahren immer durch $\frac{1}{n}$ beschränkt und unabhängig von der Entropie der Schlüsselverteilung. Außerdem haben diese Systeme einen Speicherbedarf von $\mathcal{O}(n)$, Honey Encryption gelingt dies im Optimalfall mit einem Bedarf von $\mathcal{O}(1)$.

4 DTE

Die DTE[4], 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

$$P(decode(encode(M)) = M) = 1 \qquad \forall M \in \mathcal{M}$$

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, aber jeder Seed verweist auf genau eine Nachricht (zu sehen in Abbildung 2).

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.

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 dem Message Space $\mathcal{M}$ erzeugt.

[4] *DTE* lässt sich mit *Verteilungsumwandelnde Codiermaschine* übersetzen, weshalb in dieser Ausarbeitung der feminine Genus für den Fachbegriff verwendet wird.

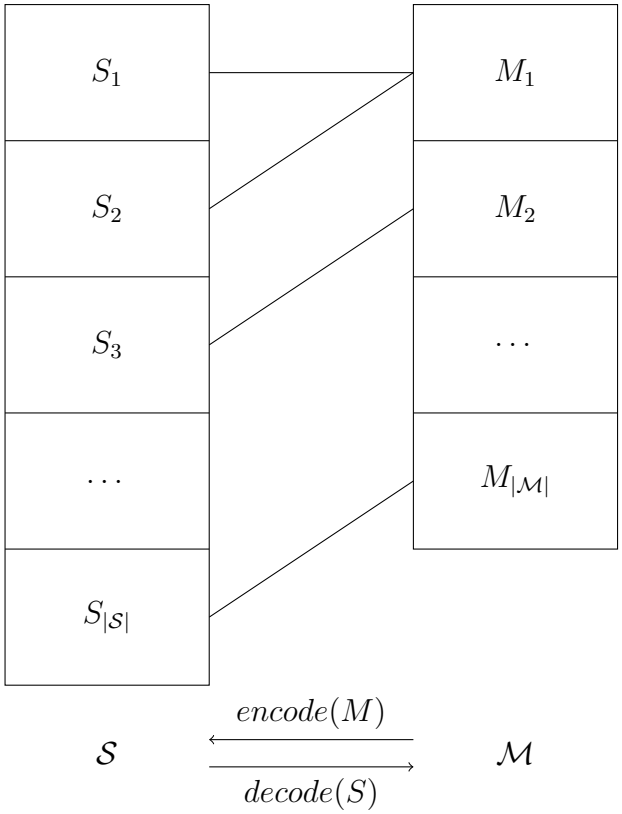

Abbildung 2: Relationen zwischen $\mathcal{M}$ und $\mathcal{S}$

Diese wird definiert als die Wahrscheinlichkeit, dass zufällig und gleichverteilt gewählte Seeds durch die $decode$-Funktion auf eine bestimmte Nachricht M abgebildet werden.

$$p_d(M) = P(M' = M : S \overset{<r>}{=} \mathcal{S} \wedge M' = decode(S))$$

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.

4.1 Generierung einer DTE mithilfe der Inversionsmethode

Ein mögliches Vorgehen zur Erstellung einer DTE, die von Juels und Ristenpart in [JR14b] vorgeschlagen wird, ist die Nutzung der sogenannten Inverse Sampling Methode, zu Deutsch Inversionsmethode. Sie wird in der Informatik und Stochastik angewendet, um mithilfe von gleichverteilten Zufallszahlen Zahlen anderer gewünschter Wahrscheinlichkeitsverteilungen zu erhalten (siehe [Kol08]). 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, im Gegensatz zu den normalerweise von ihm generierbaren Zahlen im Intervall $[0,1)$.

Wie schon beschrieben wurde, muss für die Generierung einer DTE die Wahrscheinlichkeitsverteilung für die Nachrichten des Message Spaces p_m 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.

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

Um die Grenzen festzulegen, sei $F_m(M_0) = 0$ und logischerweise $F_m(M_{|\mathcal{M}|}) = 1$. Die Visualisierung für eine Verteilungsfunktion solcher Art — hier des Beispiels aus Abschnitt 3.2 — ist in Abbildung 3 zu sehen.

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 M_6 als ursprüngliche Nachricht zurück. Eine anders geschriebene und leichter in Programmen umsetzbare Schreibweise der Übersetzung von Seed in Nachricht ist

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

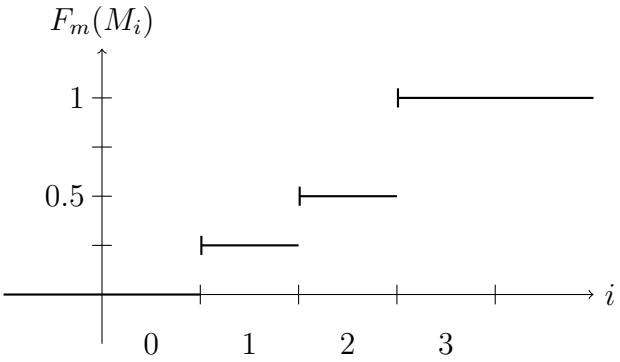

Abbildung 3: Veranschaulichung der Verteilungsfunktion F_m
mit $M_1 = r$, $M_2 = g$ und $M_3 = b$

Anschaulich sei ein Maßband der Länge 1 betrachtet, auf der die Nachrichten entsprechend ihrer Wahrscheinlichkeiten aufeinanderfolgend und disjunkt aufgetragen (in Abbildung 4 wieder am Beispiel aus Abschnitt 3.2 zu sehen) 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.

Beispielsweise wird $decode$ mit dem Parameter 0.4 aufgerufen. Auf dem Maßband liegt an dieser Stelle M_2, weshalb diese als ursprüngliche Nachricht zurück gegeben wird.

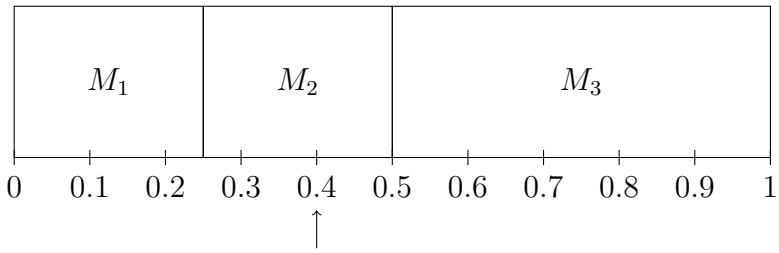

Abbildung 4: Ein Maßband als Analogie zum *decode*-Algorithmus

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 seien in dieser Ausarbeitung vernachlässigt.

Auch hier ist klar, dass es ein größeres Seedintervall für eine Nachricht gibt, je wahrscheinlicher diese Nachricht ist. Daher ist diese 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, gelöst werden. 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.

4.2 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, siehe [fSidI14]) 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 bestimmt werden (im Folgenden werden Primzahlen aus dem Intervall $[2^{l-1}, 2^l)$ gefordert). Normalerweise werden zufällige Zahlen aus dem Intervall gewählt und durch einen Primzahltest (z.B. Miller-Rabin-Test) auf Primzahleigenschaften hin überprüft. 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 nicht plausibel erkennen können.

Daher schlagen die Autoren in [JR14b] ein anderes Vorgehen vor. Zur Kodierung von p und q wird ein Vektor von 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 Primzahlen 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 diese Wahrscheinlichkeit also durch Nutzung eines größeren Vektors verringern (allerdings auf Kosten größeren Speicherplatzbedarfs).

Ein anderer Ansatz, der in [JR14b] erwähnt wird, ist 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 Bitstring handelt.

5 Verschlüsselungsschema

Nachdem eine Nachricht aus dem Message Space — wie in Abschnitt 4 beschrieben — durch eine DTE auf den Seed Space abgebildet wurde, folgt die Verschlüsselung des Ergebnisses. Hierfür schlagen die Autoren in [JR14b] zwei verschiedene Vorgehensweisen vor. Die unter Verwendung dieser Vorgehensweisen entstehenden Honey Encryption-Schemata werden im Folgenden dargestellt. Zuvor wird noch kurz auf die in den Schemata verwendeten Verfahren eingegangen.

5.1 Verwendete Verfahren

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 One Time Pad geschieht [Sch06].

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.

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].

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].

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].

5.2 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 5).

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

Abbildung 5: Hashbasierte Verschlüsselung

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

Abbildung 6: Hashbasierte Entschlüsselung

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 Bedarf 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.

Zur Entschlüsselung wird das Verfahren in ähnlicher Weise durchlaufen (Abbildung 6). 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.

5.3 Auf Blockchiffren basierte Verschlüsselung

Für die Verschlüsselung können jedoch auch Blockchiffren genutzt werden. Hierbei muss aber beachtet werden, dass der Eingangsraum der Blockchiffre gleich dem Seed Space sein muss und alle Ciphertexte unter allen möglichen Schlüsseln zu Werten aus dem Seed Space entschlüsselt werden müssen.

Im Folgenden wird das Schema unter Nutzung einer Blockchiffre mit dem CTR-Modus skizziert. Andere Betriebsmodi sollten ebenfalls nutzbar sein. So erwähnen die Autoren in [JR14b] explizit den CBC-Modus, weisen jedoch auch darauf hin, dass die Länge eines Seeds in diesem Fall ein Vielfaches der Blocklänge der Blockchiffre sein muss, so dass kein Padding [5] benötigt wird.

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

Abbildung 7: Verschlüsselung mit Blockchiffre (CTR-Modus)

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

Abbildung 8: 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 Er-

[5] Als Padding werden die zum Auffüllen des letzten, nicht vollständig belegten Blocks verwendeten Bits bezeichnet, die benötigt werden, wenn die Länge der Nachricht kein Vielfaches der Blocklänge ist.

zeugung 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 7) wird der aus der Kodierung entstandene Seed S mit den $|S|$ ersten Bits des wie bereits beschrieben berechneten Schlüsselstroms P XOR-verknüpft und bildet so den Chiffretext C, der zusammen mit R nun gespeichert oder übertragen werden kann.

Bei der Entschlüsselung (Abbildung 8) 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.

6 Einschränkungen der Honey Encryption

Durch die hohe Sicherheit der Verschlüsselung ergeben sich vor allem Anwendungsfälle, bei denen hochsensible Daten verschlüsselt werden sollen. Juels und Ristenpart geben dafür einige Beispiele, wie das Verschlüsseln von RSA-Schlüsseln oder Kreditkartennummern (siehe [JR14b, JR14a]). 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 4 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.

Es muss nicht nur die Menge aller Nachrichten bekannt sein, sondern auch die Verteilung ihrer Wahrscheinlichkeiten. 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.

Ebenfalls sollte beachtet werden, dass für jeden Anwendungsbereich, in dem Honey Encryption genutzt werden soll, eine eigene DTE konstruiert werden muss. Ein universaler Ansatz existiert dazu nicht.

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. Gibt es keine Funktion, die aus einem Seed eine Nachricht und zurück berechnen kann, so muss eine Datenstruktur gespeichert werden, die sowohl alle Nachrichten als auch alle Seeds speichern muss. 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 benötigt m Bits, dann werden *mindestens*

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

Bits zur Speicherung der Datenstruktur benötigt.
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 nötig, 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) und den resultierenden Ciphertext (nach Abschnitt 5) benötigt. Einen String von maximal 10 ASCII-Zeichen aus der Menge der Nachrichten zu verschlüsseln, führt 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 diese aber auf kurze Seeds abgebildet, so sind die Ciphertexte (abhängig von der gewählten symmetrischen Verschlüsselung) 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 es ergeben sich möglicherweise zum Schluss immer noch kleinere zu speichernde Datenmengen als bei anderen Verschlüsselungsverfahren mit gleicher Ciphertextanzahl.

Es sei noch einmal explizit darauf hingewiesen, dass eine solche, wie oben beschriebene Datenstruktur nicht vonnöten ist, wenn sich eine Funktion finden lässt, die die Aufgaben der DTE ohne größeren Speicherplatzbedarf erfüllen kann. Hierfür sei ein Beispiel die beschriebene Verschlüsselung von RSA-Schlüsseln in Abschnitt 4.2.

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 bei herkömmlichen AE- und PBE-Schemata 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 in [Jue, JR14b] insgesamt drei Ansätze vor, 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. Allerdings wäre es möglich, diese Technik zum Beispiel im Online-Bereich anzuwenden. Durch eine passend gewählte Anzahl von maximalen Anmeldeversuchen wäre es für einen Dienstleister möglich zu erkennen, wann ein Angreifer versucht, das Passwort für ein Konto zu erraten. Schließlich probiert dieser nur Passwörter, für die die Prüfsumme stimmt. Ein Vertippen des wahren Nutzers beim Passwort würde im wahrscheinlichsten Fall eine falsche Prüfsumme hervorrufen.

Ein weiterer Ansatz ist die Online-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 zugreift, wäre es nach Juels und Ristenpart möglich, beispielsweise die ersten beiden Ziffern der Kreditkartennummer als Klartext im Honey Encryption-Verfahren zu speichern. Will ein Angreifer die richtige Kreditkartennummer erraten, würde dieser auf jeden Fall die beiden im Klartext gespeicherten Ziffern übernehmen. Bei einem Tippfehler des Nutzers hingegen wären die Ziffern sehr wahrscheinlich nicht korrekt. Der Dienstleister kann also durch die Korrektheit der ersten beiden Ziffern in Verbindung mit einem falschen Rest erkennen, dass mit hoher Wahrscheinlichkeit ein Angriff stattfindet. Stimmen die ersten beiden Ziffern nicht überein, liegt höchstwahrscheinlich ein Schreibfehler vor und der Dienstleister kann den Nutzer bitten, die Passworteingabe erneut zu tätigen. Bei dieser Methode wird der Message Space verkleinert, was auch zu weniger Sicherheit führt. Wie schon bei dem ersten Ansatz sollte diese Methode dementsprechend mit Vorsicht angewendet werden.

Der dritte und letzte Vorschlag seitens der beiden Autoren ist es, eine Farbe, ein Bild oder ein Muster mit den anderen Informationen zusammen zu verschlüsseln. Diese Komponente wird ebenfalls wieder hergestellt, wenn das richtige Passwort eingegeben wurde, und der Ciphertext wird zu einer anderen, aber plausibel wirkenden Komponente entschlüsselt, wenn es eine Fehleingabe gab (beispielsweise *blau* statt *rot*). Es wird also der größte Vorteil der Honey Encryption angewendet, um dem Nutzer einen Hinweis darauf zu geben, ob er den richtigen Schlüssel eingegeben hat. Ein Angreifer kennt auch das Muster nicht, welches mit gespeichert wurde, der Nutzer hingegen kann sich daran erinnern. Er muss sich allerdings die ursprüngliche Komponente 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, ein neues Muster präsentiert. Bei der Unmenge an Logins, die jeder Mensch heutzutage tätigt, könnte das für Verwirrung sorgen. Dass der Nutzer diese Komponente selbst belegen darf, wäre hier ein Sicherheitsrisiko. Schließlich nutzen viele Menschen für ihre unterschiedlichen Benutzerkonten dasselbe Passwort, weil sie sich nicht mehrere Passwörter merken wollen. Dies wäre bei Farben, Bildern oder Mustern nicht anders. Ein Angreifer kann also die Menge der Nachrichten einschränken, *wenn* er die Wahl des Nutzers bei anderen Verschlüsselungen mithilfe von Honey Encryption kennt. Bei einem einzigen Login, wie zum Beispiel im Falle eines Passwort-Managers, kann diese Methode aber sehr hilfreich sein.

Honey Encryption funktioniert nur dann, wenn ein Angreifer keine Informationen über die verschlüsselte Nachricht besitzt. Das Beispiel im Abschnitt 4.2 macht deutlich, dass Honey Encryption nur so lange eine starke Verschlüsselung für private RSA-Schlüssel bietet, wie der Angreifer den öffentlichen Schlüssel nicht kennt. Ist dieser im Besitz des Angreifers, kann er sehr leicht überprüfen, ob er den richtigen Klartext entschlüsselt hat. Die auf die herkömmliche Verschlüsselungsmethode aufbauende Honey Encryption wäre dann nutzlos. Die Sicherheit würde auf das Level der verwendeten Verschlüsselungsmethode und damit einer PBE-Verschlüsselung zurückfallen.

Ähnliches geschieht auch, wenn die Nachrichten und die Schlüssel korreliert sind und dies dem Angreifer bekannt sind. Schlüssel, die in Abhängigkeit zur Nachricht gewählt werden, helfen dem Angreifer beim Überprüfen der Plausibilität einer Schlüssel-Nachricht-Konstellation. Da-

durch kann der Angreifer viele, wenn nicht sogar alle bis auf eine, Kombinationen herausfiltern. Das Gleiche gilt für korrelierte Nachrichten, die mit demselben Schlüssel verschlüsselt wurden. Beispielsweise sind Chatverläufe abhängig voneinander, da sich Nachrichten meist auf die Nachrichten davor beziehen. Ein Angreifer würde dann die Plausibilität des Verlaufs leichter überprüfen können. Unabhängige Nachrichten mit demselben Schlüssel zu verschlüsseln, gilt aber als sicher, da die Nachrichten in jedem Fall keinen Zusammenhang haben.

7 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 einen Mehrwert dar.

Allerdings überwiegen im jetzigen Forschungszustand die Nachteile und Probleme (aufgezeigt in Abschnitt 6). Die verlangte Kenntnis über die Menge aller Nachrichten und ihrer Wahrscheinlichkeitsverteilung verhindert die großflächige Anwendung von Honey Encryption. Ebenfalls sollte das Problem der *Typo-Safety* weiter untersucht werden, um die Nutzererfahrung zu verbessern.

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 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 (siehe [Jue]).

Ein weiteres Forschungsziel sollte es sein, die Möglichkeiten und Anwendungsbereiche von Honey Encryption zu erweitern. So könnte eine spannende Forschungsfrage sein, inwieweit 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 sinnverwandter 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. Mithilfe immer besser funktionierender Sprachverarbeitung, komplexerer stochastischer Modelle und weiterer interdisziplinärer Erkenntnisse wird Honey Encryption eventuell der Weg zu einer großflächigeren Anwendung geebnet.

Literatur

[BN00] Mihir Bellare und Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In Advances in Cryptology - ASIACRYPT 2000. Springer Berlin Heidelberg, 2000.

[fSidI14] Bundesamt für Sicherheit in der Informationstechnik. Technische Richtlinie: Kryptographische Verfahren: Empfehlungen und Schlüssellängen. TR-02102-1, Januar 2014.

[JR] Ari Juels und Thomas Ristenpart. Honey Encryption: Security Beyond the Brute-force Bound. http://ec14.compute.dtu.dk/talks/19.pdf. Zugriff am 16.12.2014.

[JR13] Ari Juels und Ronald L. Rivest. Honeywords: Making Password-Cracking Detectable. In Proceedings ACM CCS'13, Berlin, Germany, November 2013.

[JR14a] Ari Juels und Thomas Ristenpart. Honey Encryption - Encryption beyond the Brute-Force Barrier. IEEE Security & Privacy, 12(4):59–62, April 2014.

[JR14b] Ari Juels und Thomas Ristenpart. Honey Encryption - Security Beyond the Brute-Force Bound. In EUROCRYPT 2014, Kopenhagen, Dänemark, Mai 2014.

[Jue] Ari Juels. The Password That Never Was. http://crcs.seas.harvard.edu/event/ari-juels-the-password-that-never-was. Zugriff am 27.10.2014.

[Jue14] Ari Juels. A Bodyguard of Lies - The Use of Honey Objects in Information Security. In 19. ACM SACMAT, London, Kanada, Juni 2014.

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

[Kol08] M. Kolonko. Stochastische Simulation: Grundlagen, Algorithmen und Anwendungen. Vieweg Studium. Vieweg Verlag, Friedr, & Sohn Verlagsgesellschaft mbH, 2008.

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