Implementierung der FEAL-Differentrial-Cryptanalysis Attacke nach Murphy

Lukas Becker Juri Golanov

August 31, 2016

Inhaltsverzeichnis

1 Einleitung

Seit jehher herrscht in der Kryptographie ein Rennen zwischen Forschern, die neue Verfahren entwickeln und Leuten, welche die Schwachstellen in den Verfahren suchen, um diese für ihre Zwecke auszunutzen. Bei der Suche nach *dem* sicheren Krypto-Verfahren helfen Paper, wie das von Sean Murphy [1]. Sie zeigen den Forschern die Schwachstellen ihrer Algorithmen auf und wie diese in einer Attacke ausgenutzt werden. Durch diese Erkenntnisse können dann alte Verfahren verbessert oder neue entwickelt werden, um die jetzt bekannten Schwächen zu beseitigen.

In der folgenden Ausarbeitung werden wir uns der Implementierung der *Krypto-Attacke auf den FEAL Algorithmus mit 20 Plaintextblöcken oder weniger* [1] von Sean Murphy befassen.

1.1 Aufbau

Zunächst werden wir das FEAL Krypto-Verfahren an sich beleuchten. Dazu gehören einmal der Aufbau der Logik, sowie die verwendeten Algorithmen und Funktionen.
Im nächsten Schritt wird auf die von Sean Murphy entwickelte Attacke eingegangen. Hier wird vorallem aufgezeigt welche Schwächen Murphy in dem Verfahren entdeckt hat und wie er diese ausnutzt.
Nach der Theorie folgt dann die Implementierung der Attacke. Dieses Kapitel beschreibt überwiegend den Projektverlauf vom ersten Auseinandersetzen mit dem Paper bis hin zum fertigen Programm.
Im Anschluss wird ein Fallbeispiel einer Attacke durchgespielt, um zu veranschaulichen wie das Programm, also die Attacke, vorgeht, um verschlüsselte Texte ohne Wissen des Schlüssels zu entschlüsseln.
Danach wird auf Probleme eingangen, denen wir beim Bewältigen des Problems begegnet sind, sowie der resultierende Lösungsweg.
Abschließend folgt eine kurze Konklusion zu dem fertigen Projekt.

2 FEAL

FEAL-N steht für *Fast Data Encipherment Algorithm* und ist eine Blockchiffre, welches auf dem Feistel-Algorithmus basiert, zudem ist es ein symmetrisches Kryptoverfahren. FEAL wurde im Jahre 1987 von dem Entwicklerteam Akihiro Shimizu und Shoji Miyaguchi des japanischen Telefonkonzerns *Nippon Telegraph and Telephone* (NTT) veröffentlicht [2]. Das Ziel der Entwicklung war es einen schnellen Verschlüsselungsalgorithmus zu schaffen, der sich effizient in Software zu implementieren ließ. Es sollte eine Alternative zu dem symmetrischen Verschlüsselungsalgorithmus *Data Encryption Standard* (DES) darstellen, welches von der US-Regierung entwickelt wurde und sich damals nur leicht in spezielle Hardware implementieren ließ.

Das N in FEAL-N repräsentiert die Anzahl der Runden der Feistel-Blockchiffren-Operationen auf 64-Bit großen Blöcke bestimmt durch 64-Bit große Schlüssel. Diese Ausarbeitung befasst sich ausschließlich mit der Version FEAL-4.

Die folgenden Punkte zeigen auf, wann, wo und von wem die verschiedenen Versionen von FEAL erfolgreich gebrochen werden konnten:

- **FEAL-4** noch im gleichen Jahr 1988 auf der Eurocrypt '88 von B. den Boer
- **FEAL-4** im Jahr 1990 von Sean Murphy mit differentieller Kryptoanalyse unter Verwendung 20 gewählter Plaintextblöcke (Thema dieser Ausarbeitung)
- **FEAL-8** im Jahr 1989 von Biham und Shamir auf der Konferenz SECURICOM '89
- **FEAL-N** mit einer variablen Anzahl an Runden und **FEAL-NX** mit 128 Bit langen Schlüssel statt 64 Bit auf der SECURICOM '91 wieder von Biham und Shamir

FEAL hat sich aufgrund zahlreicher Sicherheitsmängel nicht durchgesetzt und sollte bei sicherheitskritischen Anwendungen nicht mehr verwendet werden. Es dient heutzutage vor allem zum Testen neuer kryptoanalytischen Angriffsmethoden.

2.1 Kurzer Exkurs: Feistel

Eine Feistel-Chiffre besteht aus einer bestimmten Anzahl an Runden, wobei jeweils aus dem Schlüssel ein Rundenschlüssel gebildet wird. Die untere Abbildung zeigt die typische Vorgehensweise des Feistel-Algorithmus.
Vor jeder Runde wird der Text in eine linke (L) und eine rechte Hälfte (R) eingeteilt. Dann wird auf die rechte Hälfte eine Funktion f angewandt, die Teile

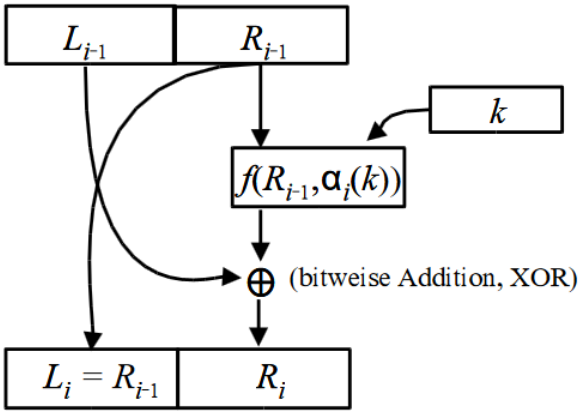

Figure 1: Feistelchiffre

des Schlüssels bzw. des sogenannten Rundenschlüssels k zusätzlich als Parameter mitbekommt. Das Ergebnis wird mit XOR ($\oplus$) mit der linken Texthälfte verknüpft. Das Ergebnis ist dann die rechte Texthälfte für die nächste Runde. Die alte rechte Texthälfte wird die neue linke. Der Parameter i steht für die Anzahl der Runden. Wie der Feistel-Chiffre im FEAL-4 angewandt wird, wird im folgendem Abschnitt behandelt.

2.2 Funktionsweise

In diesem Abschnitt wird detailliert auf die Vorgehensweise des Algorithmus eingegangen, wie das Verschlüsselungsverfahren FEAL-4 funktioniert. Es werden Funktionen vorgestellt und erklärt zu welchem Zweck diese dienen. Zum groben Ablauf wird zunächst einmal aufgezeigt wie aus dem 64-Bit Schlüssel die Subkeys generiert werden, danach wie Klartexte verschlüsselt und
dementsprechend wieder entschlüsselt werden. Dieser Abschnitt richtet sich nach dem Paper von Sean Murphy [1].
Zu aller erst wird die S-Box-Funktion vorgestellt, diese ist der Grundbestandteil der wichtigsten Funktionen im FEAL. Diese sieht folgendermaßen aus:

$$S_i(x, y) = Rot_2((x + y + i) Mod 256)$$

Der Parameter i ist entweder 0 oder 1, x und y sind 8-Bit große binäre Zahlen im Bereich von 0 bis 255. Rot_2 entspricht einer Rotation nach links um zwei Bit. Das Ergebnis/Output der S-Box-Funktion entspricht einer 8-Bit großen binären Zahl.

2.2.1 Generierung der Subkeys

Aus dem 64-Bit Key sollen zwölf 16-Bit Subkeys entstehen, welche anschließend verwendet werden, um die Klartexte zu Verschlüsseln. Damit man von den Subkeys aus nicht so einfach auf den ursprünglichen Schlüssel schließen kann, werden die Bits des Keys in mehreren Stufen systematisch durcheinandergebracht.
Als erstes wird dazu der Key in zwei Hälften aufgeteilt. Dazu werden die 32-Bit langen Hilfsvariablen B benötigt, diese erstecken sich von B_{-2} bis B_6. Dabei wird der linke Teil des Keys (K_L) der Variable B_{-1} der rechte Teil des (K_R) Keys der Variable B_0 zugeordnet, die Variable B_{-2} ist auf null gesetzt.

$$B_{-2} = 0; \qquad B_{-1} = K_L; \qquad B_0 = K_R$$

Die linken und rechten Hälften von B_1 bis B_6 sind die gesuchten Subkeys, sie werden mithilfe der oben genannten Hilfsvariablen und der f_k-Funktion folgendermaßen berechnet.

$$B_i = f_k(B_{i-2}, B_{i-1} \oplus B_{i-3})$$

In der oberen Gleichung wird die f_k-Funktion verwendet, diese ist für das *Verwürfeln* der Bits zuständig. In nachstehender Abbildung wird die f_k-Funktion veranschaulicht:

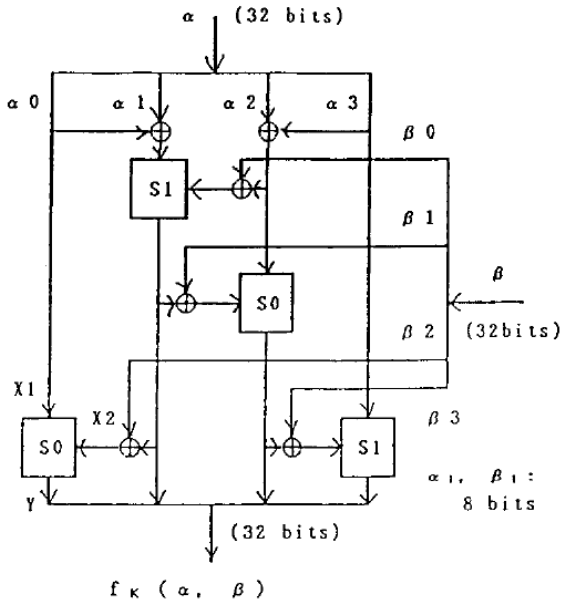

Figure 2: *fk*-Funktion

Die f_k-Funktion ist für $c = f_k(a, b)$ wie folgt definiert:

$$\begin{aligned} d_1 &= a_0 \oplus a_1 \\ d_2 &= a_2 \oplus a_3 \\ c_1 &= S_1(d_1, d_2 \oplus b_0) \\ c_2 &= S_0(d_2, c_1 \oplus b_1) \\ c_0 &= S_0(a_0, c_1 \oplus b_2) \\ c_3 &= S_1(a_3, c_2 \oplus b_3) \end{aligned}$$

Die 32-Bit großen Bitblöcke a und b stellen den Input dar. Diese werden anschließend in vier 8-Bit große Teilblöcke a_i und b_i unterteilt ($i = 0, \ldots, 3$), welche mithilfe der S-Box-Funktion und der XOR-Operation miteinander *vermischt* werden. Daraus resultieren sich die Variablen c_i, welche zusammengesetzt das 32-Bit lange Ergebnis der Funktion liefert.
Zu guter Letzt werden die Hilfsvariablen B_1 bis B_6 aufgeteilt und als die endgültigen Subkeys verwendet. Aus den sechs 32-Bit langen Hilfsvariablen entstehen nun die zwölf 16-Bit langen Subkeys K_0 bis K_{11}.

$$K_{2(i-1)} = B_i^L; \qquad K_{2i-1} = B_i^R$$

Aus oberer Gleichung resultiert folgendes Ergebnis in Worten: die sechs gerade nummerierten Subkeys K_0 bis K_{10} bilden die linken Hälften von B und die sechs ungerade nummerierten Subkeys K_1 bis K_{11} bilden die rechten Hälften von B.
Im Folgendem wird der oben aufgezeigte Vorgang vereinfacht dargestellt. Dazu dient die untere Abbildung, die die Generierung der Subkeys visuell darstellt und verständlicher macht. Zu erwähnen ist, dass in folgender Abbildung zwei Schritte ausgelassen wurden. Die Erstellung der Subkeys K_8 bis K_{11} muss man sich dazu denken.

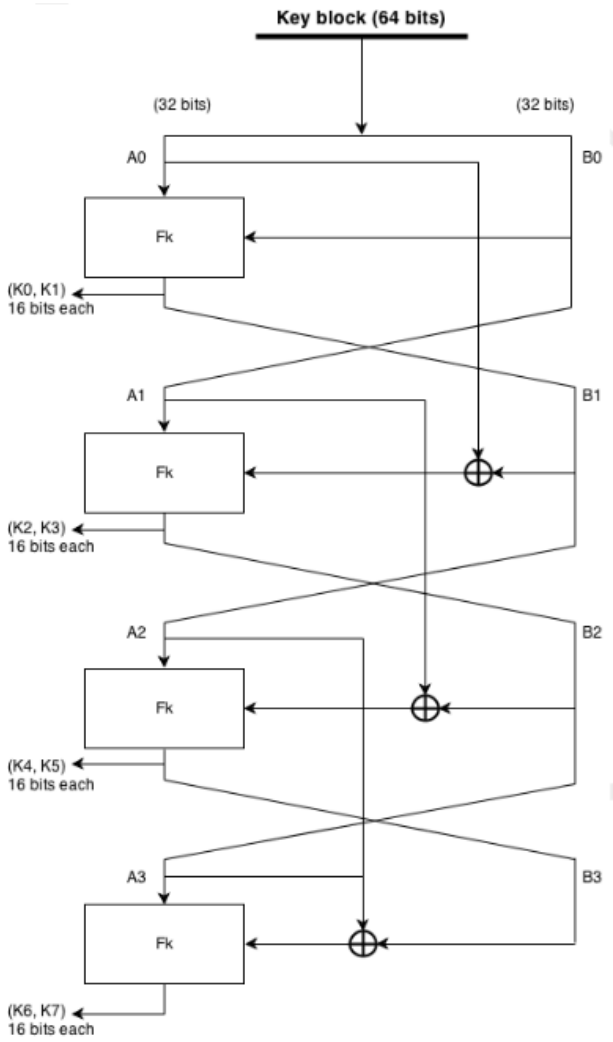

Figure 3: Erstellung der Subkeys vereinfacht

2.2.2 Verschlüsselung der Klartexte

Das Verschlüsseln eines 64-Bit großen Klartextblockes P erfolgt erneut durch Aufteilung des Blockes in eine linke (P_L) und eine rechte (P_R) Hälfte. Diese werden mit den Subkeys per XOR-Operation miteinander verknüpft und als Initialzustand für den Feistel-Chiffre genommen. Dazu werden die Variablen L_0

und R_0 verwendet und werden wie folgt berechnet:

$$L_0 = P_L \oplus (K_4, K_5)$$
$$R_0 = P_L \oplus P_R \oplus (K_4, K_5) \oplus (K_6, K_7)$$

Von hier aus werden nun die vier Runden des Feistel-Algorithmus angewendet. Dafür wird die f-Funktion und die ersten vier Subkeys K_0 bis K_3 benutzt. Die f-Funktion ist der f_k-Funktion von der Form sehr ähnlich und wird in der unteren Abbildung veranschaulicht, die entsprechenden Gleichungen sind ebenfalls aufgeführt.

$$c = f(a, b)$$

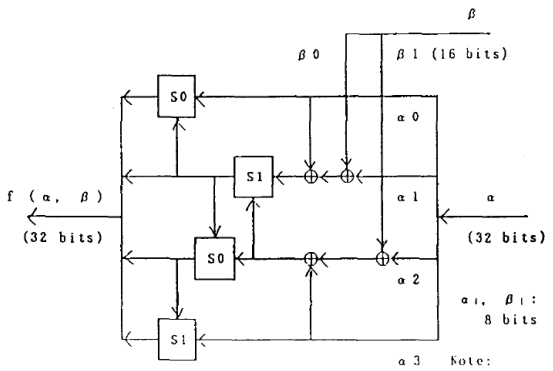

Figure 4: f-Funktion

$$d_1 = a_0 \oplus a_1 \oplus b_1$$
$$d_2 = a_2 \oplus a_3 \oplus b_2$$
$$c_1 = S_1(d_1, d_2)$$
$$c_2 = S_0(d_2, c_1)$$
$$c_0 = S_0(a_0, c_1)$$
$$c_3 = S_1(a_3, c_2)$$

Die Gleichung für den Durchlauf der vier Feistel-Runden sieht wie folgt aus, für i=0,1,2,3 :

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, K_{i-1})$$

Die daraus resultierenden Ergebnisse des Feistel-Algorithmus L_4 und R_4 werden abschließend mit den letzten vier Subkeys K_8 bis K_{11} per XOR miteinander verknüpft und den Variablen C_L und C_R zugewiesen.

$$C_L = R_4 \oplus (K_8, K_9)$$
$$C_R = R_4 \oplus L_4 \oplus (K_{10}, K_{11})$$

Daraus ergibt sich zu guter Letzt der verschlüsselte Ciphertextblock C mit

$$C = (C_L, C_R).$$

Auf die gleiche Weise können wir, wenn wir den Schlüssel kennen, jede verschlüsselte Nachricht dekodieren, indem die oben beschriebene Vorgehensweise einfach in umgekehrter Reihenfolge angewendet wird. Die unteren Abbildungen vereinfachen die oben im Detail beschriebene Vorgehensweise des Verschlüsselns und veranschaulichen als Gegenstück den Vorgang des Entschlüsselns.

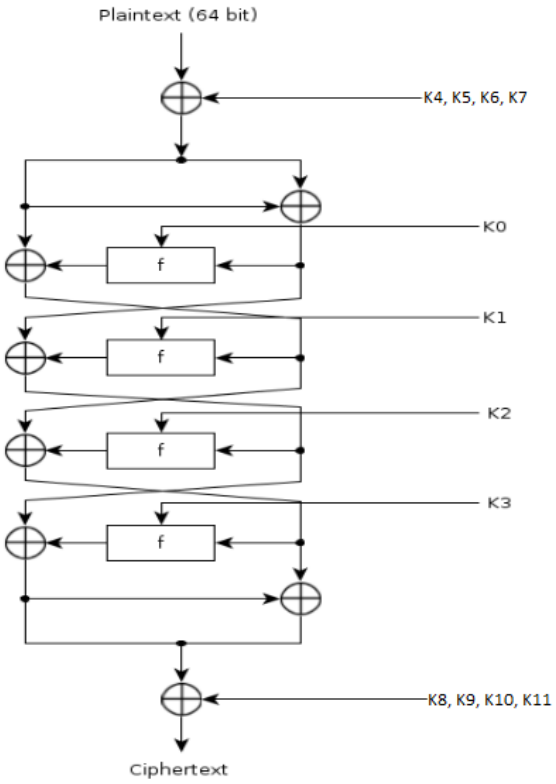

Figure 5: Encode vereinfacht

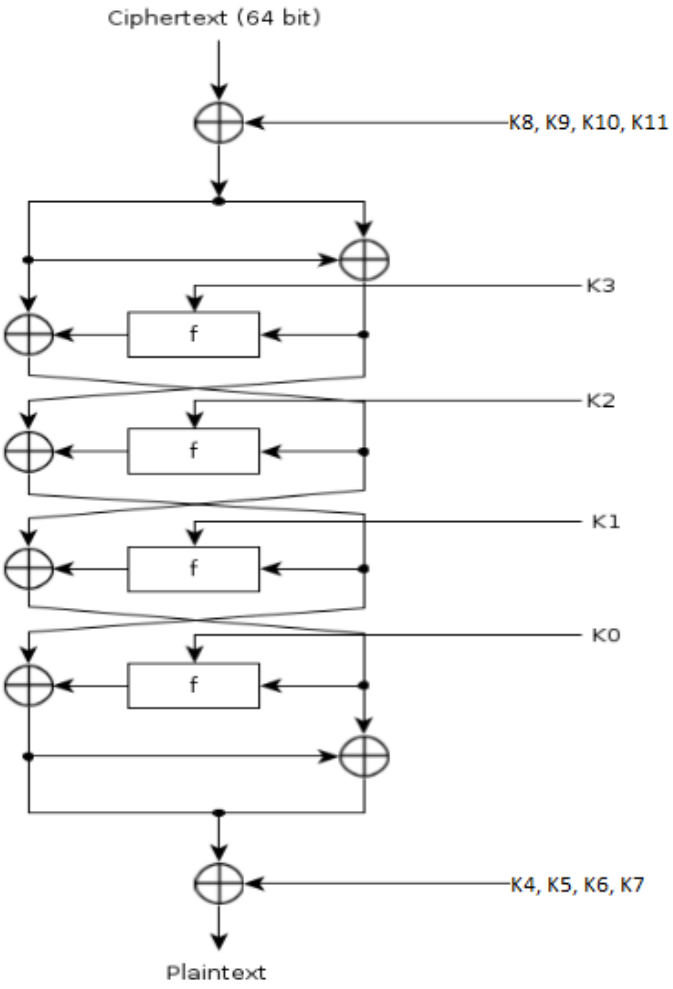

Figure 6: Decode vereinfacht

3 Attacke nach Murphy

In diesem Kapitel wird die Attacke von Sean Murphy nach dem Paper [1] in den Fokus genommen. Es wird aufgezeigt welche Schwächen Murphy in dem Verfahren entdeckt hat und wie er diese ausnutzt, um das Verschlüsselungsverfahren FEAL-4 zu brechen. Murphys Vorgänger Den Boer [3] verwendete 10.000 Plaintexte, um den Schlüssel zu rekonstruieren. Im Gegensatz dazu, und hier liegt die Besonderheit, dass sich Murphy bei der differentiellen Kryptoanalyse nur auf 20 gewählte Plaintextblöcke beschränkte. In diesem Kapitel wird auf viele Gleichungen verwiesen, dabei entsprechen diese der Namenkonvention aus dem Paper von Murphy [1].

3.1 Wahl der 20 Plaintexte

Bevor wir auf die Kryptoanalyse von Murphy eingehen, wird zunächst einmal erklärt auf welche Art und Weise die Plaintexte überhaupt gewählt werden. Die Wahl unterliegt nämlich einem systematischen Prinzip, welche nun vorgestellt wird.
Die Wahl der 20 Plaintexte erfolgt nach bestimmten Regeln, bevor diese aufgezeigt werden, müssen als erstes einige Variablen definiert werden. Dazu werden die linken und rechten Hälften der Plaintextblöcke (P^i) den Variablen P_L^i und P_R^i zugeordnet. Auf dieselbe Art erfolgt die Zuweisung der Ciphertextblöcke (C^i) mit den entsprechenden linken (C_L^i) und rechten (C_R^i) Hälften. Dabei ist zu erwähnen, dass die Zählervariable i sich von 0 bis 19 erstreckt, damit die Anzahl auf 20 Plaintexte zustande kommt. Nun werden die beiden Hälften eines Plaintextes mit der XOR-Operation miteinander verknüpft und in eine weitere neue Variable geschrieben. Analog dazu erfolgt dasselbe mit den Ciphertexten. Dies sieht dann wie folgt aus:

$$Q^i = P_L^i \oplus P_R^i \qquad \textit{und} \qquad D^i = C_L^i \oplus C_R^i$$

Die Wahl der Plaintexte unterliegt folgenden Regeln:

1. Wähle P^0, P^{12}, P^{14}, P^{16}, P^{17}, P^{18}, P^{19} zufällig
2. Wähle P_L^5, P_L^6, P_L^8, P_L^9, P_L^{10}, P_L^{11}, P_L^{13}, P_L^{15} ebenfalls zufällig, also die linken Hälften der jeweiligen Plaintextblöcken
3. Verknüpfe folgende Plaintextblöcke mit Bitmasken unter Verwendung der XOR-Operation:
   - a) $P_L^1 = P_L^0 \oplus 80800000$
   - b) $P_L^2 = P_L^0 \oplus 00008080$

- c) $P_L^3 = P_L^0 \oplus 40400000$

- d) $P_L^4 = P_L^0 \oplus 00004040$

4. Definiere für die restlichen rechten Hälften folgende Gleichungen:

   - a) $P_R^i = P_L^i \oplus Q^0, \quad$ wobei $i = 1, \ldots, 11$

   - b) $P_R^{13} = P_L^{13} \oplus Q^{12}$

   - c) $P_R^{15} = P_L^{15} \oplus Q^{13}$

Somit wurden 7 ganze Plaintextblöcke und 9 Plaintext-Hälften zufällig gewählt, daraus ergeben sich 736 zufällige Bits von insgesamt 1280 Bits. Nun besteht die Aufgabe darin, die Zusammenhänge zwischen den Bits herauszufinden.

3.2 Neuformulierung des FEAL-4 Algorithmus

Um die Kryptoanalyse von Murphy zu verstehen muss noch vorher die Umformulierung des FEAL-4 Algorithmus erläutert werden. Dies stellt nämlich das Grundgerüst dar, auf welche die Attacke baut. Dabei bezieht sich Murphy auf die Methode von Den Boer [3] und definiert die G-Funktion, welche die lineare Eigenschaft der f-Funktion darstellt. Diese wird wie folgt definiert:

$$c = G(a)$$

mit

$$\begin{aligned} d_1 &= a_0 \oplus a_1, \\ d_2 &= a_2 \oplus a_3, \\ c_1 &= S_1(d_1, d_2) \\ c_2 &= S_0(d_2, c_1) \\ c_0 &= S_0(a_0, c_1) \\ c_3 &= S_1(a_3, c_2) \end{aligned}$$

und somit

$$f(a, b) = G(a_0, a_1 \oplus b_1, a_2 \oplus b_2, a_3)$$

Diese Funktion ist die Kernfunktion in der Attacke und wird sehr häufig verwendet.
Bei dieser Umformulierung geht es hauptsächlich darum, eine neue Methode zu finden, um Nachrichten auf einem alternativen Weg zu Verschlüsseln und zu Entschlüsseln. Dazu werden als Alternative zum Schlüssel sechs verschiedene Schlüsselkonstanten definiert. Diese sehen folgendermaßen aus:

$$
\begin{aligned}
M_1 &= B_3 \oplus \theta_R(B_1) \\
N_1 &= B_3 \oplus B_4 \oplus \theta_L(B_1) \\
M_2 &= \theta_L(B_1) \oplus \theta_L(B_2) \\
N_2 &= \theta_R(B_1) \oplus \theta_R(B_2) \\
M_3 &= B_5 \oplus B_6 \oplus \theta_R(B_1) \\
N_3 &= B_5 \oplus \theta_L(B_1)
\end{aligned}
$$

Zu beachten ist, dass die äußeren Bits von M_2 und N_2 null sind. Die Theta-Funktionen (θ) zentrieren lediglich die jeweiligen 32-Bit Block-Hälften und setzen die äußeren Bits auf null:

$$
\begin{aligned}
\theta_L(a_0, a_1, a_2, a_3) &= (0, a_0, a_1, 0) \\
\theta_R(a_0, a_1, a_2, a_3) &= (0, a_2, a_3, 0)
\end{aligned}
$$

$$
\begin{aligned}
\theta_L(B_i) &= (0, K_{2(i-1)}, 0) \\
\theta_R(B_i) &= (0, K_{2i-1}, 0)
\end{aligned}
$$

Die θ_L-Funktion stellt zentriert die geraden Subkeys dar. Die θ_R-Funktion stellt zentriert die ungeraden Subkeys dar.
Durch diese Schlüsselkonstanten ist es nun möglich den FEAL-4 Algorithmus wie folgt umzuschreiben und damit Klartexte zu verschlüsseln:

$$
\begin{aligned}
X_0 &= P_L \oplus M_1 = L_0 \oplus \theta_R(B_1), \\
Y_0 &= P_L \oplus P_R \oplus N_1 = R_0 \oplus \theta_L(B_1) = L_1 \oplus \theta_L(B_1), \\
X_1 &= X_0 \oplus G(Y_0) = R_1 \oplus \theta_R(B_1) = L_2 \oplus \theta_R(B_1), \\
Y_1 &= Y_0 \oplus G(X_1) = R_2 \oplus \theta_L(B_1) = L_3 \oplus \theta_L(B_1), \\
X_2 &= X_1 \oplus G(Y_1 \oplus M_2) = R_3 \oplus \theta_R(B_1) = L_4 \oplus \theta_R(B_1), \\
Y_2 &= Y_1 \oplus G(X_2 \oplus N_2) = R_4 \oplus \theta_L(B_1), \\
C_L &= Y_2 \oplus N_3, \\
C_R &= X_2 \oplus M_3 \oplus C_L.
\end{aligned}
$$

Wendet man die oberen Gleichungen in umgekehrter Reihenfolge an, ist es möglich den verschlüsselten Text wieder zu entschlüsseln. Aus den sechs Schlüsselkonstanten ergeben sich insgesamt 160 unbekannte Bits. Wenn es uns gelingt diese unbekannten Bits auszurechnen, erweist sich uns die Möglichkeit jeden Ciphertext zu entziffern und ebenfalls auf den Schlüssel zurückzuschließen. Damit kommen wir nun im nächsten Abschnitt zur eigentlichen Attacke und Kryptoanalyse von Murphy.

3.3 Kryptoanalyse des FEAL-4 Algorithmus

In diesem Abschnitt wird die Vorgehensweise der Attacke von Murphy beschrieben, um den FEAL-4 Algorithmus zu brechen. Das Ziel der Attacke ist es die sechs Schlüsselkonstanten M_1, N_1, M_2, N_2, M_3 und N_3 herauszufinden.
Zu aller erst werden die 20 Plaintexte gewählt und verschlüsselt, wie im vorherigen Abschnitt beschrieben. Damit entstehen die Plain- und Ciphertextblöcke P^i und C^i, sowie Q^i und D^i. Diese sind notwendig für das weitere Verfahren.
Um die dazugehörigen Bits in Zusammenhang zu bringen formuliert Murphy aus (2.7) neue Gleichungen, und zwar (5.1) und die daraus folgende Gleichung (5.2). Hier wurde von Y_1 ausgegangen und die Werte nach und nach mathematisch eingesetzt. So entsteht aus den Gleichungen

$$Y_1 = Y_0 \oplus G(X_1), X_1 = X_0 \oplus G(Y_0) und X_0 = P_L \oplus M_1$$

Die folgende

$$Y_1 = Y_0 \oplus G(X_1) = Y_0 \oplus G(X_0 \oplus G(Y_0)) = Y_0 \oplus G(P_L \oplus M_1 \oplus G(Y_0)) \ldots$$

Da diese Gleichungen (5.1) und die Gleichung (5.2) sehr umfangreich sind werden sie an dieser Stelle nicht aufgelistet, sowie die anderen Gleichungen aus dem Paper von Murphy. Um genauere Details zu erhalten, empfiehlt es sich im Paper nachzuschlagen [1]. Es soll lediglich das Prinzip verstanden werden, dass Murphy die Werte aus (2.7) und (2.6) in Zusammenhang mit den Ciphertextblöcken bringt. Aus der Gleichung (5.2) schließt er die Gleichungen für die Werte U^i, V^i und W. Diese bezeichnet er als Triple und sind essenziell zur weiteren Vorgehensweise.

$$\begin{aligned} U^i &= Y_0^i \oplus N_3 \\ V^i &= M_1 \oplus G(Y_0^i) \\ W &= M_3 \oplus N_2 \end{aligned}$$

Von hier aus werden Schritt für Schritt diese Werte ausgerechnet. W setzt sich aus den Schlüsselkonstanten M_3 und N_2 zusammen und ist daher konstant. Deshalb wird zu Beginn nach möglichen W Werten gesucht. Aus der dritten Regel zur Wahl der Plaintexte (XOR mit Bitmasken) schließt Murphy auf die Gleichung (5.8) mithilfe der linken Hälften der Ciphertextblöcke C_L^0, C_L^1, C_L^2 und der D-Werte D^0, D^1 und D^2. Diese hat folgende Form:

$$\begin{aligned} G(x \oplus a) \oplus G(x \oplus b) &= d \\ G(x \oplus a) \oplus G(x \oplus c) &= e \end{aligned}$$

Dies entspricht der Gleichung (3.7) im Paper [1] und erfordert 2^{17} Berechnungen und wird zur Berechnung gleichzeitig ausgeführt. Das Ergebnis der Berechnungen aus Gleichung (5.8) liefert eine Menge an möglichen Lösungen für W. Um

die Ergebnismenge zu reduzieren wird geprüft, ob die gefundenen W Werte die Gleichungen (5.9) erfüllen. Diese ist der Gleichung (5.8) sehr ähnlich, hier werden bloß andere Ciphertextblöcke, andere D-Werte und andere Bitmasken verwendet. W Werte, die die Gleichungen (5.9) nicht lösen, werden aussortiert. Auf diese Weise wird die resultierende Ergebnismenge auf bis zu zehn Lösungen für W verkleinert.
Als nächstes wird für jede gefundene Lösung für W und der Gleichungen (5.10) nun Lösungen für V^0 gesucht. Diese Gleichung ähnelt den beiden vorherigen, hier werden andere C und D-Werte benutzt und die Verwendung von Bitmasken entfällt. Gleichung (5.5) liefert mögliche U^0 Werte und damit hätten mehrere Lösungen für das Triplet (W, V^0, U^0). Durch die Gleichung (5.5) folgt, dass die ersten zwölf Werte $Y_0 = Q^i \oplus N_1$ und $G(Y_0)$ konstant sind, also für $i = 0$, $\ldots$, 11, folgt daraus, dass $U^i = U^0$ und $V^i = V^0$. Dies stellt die Bedingung dar, um weniger als 20 Lösungen für das Triplet (W, V^0, U^0) zu liefern.
Murphy fährt weiter fort einzelne U- und V-Werte mit aufgestellten Gleichungen zu suchen bis er die Werte U^{12}, U^{13}, U^{14}, U^{15} sowie V^{12}, V^{13}, V^{14}, V^{15} findet. Damit ist es ihm möglich mit der Gleichung (5.13) mögliche Lösungen für die erste Schlüsselkonstante N_1 auszurechnen. Mithilfe dieser Lösungen lässt sich aus den folgenden Gleichungen auf M_1 und N_3 schließen:

$$\begin{aligned} Y_0 &= Q^0 \oplus N_1 \\ M_1 &= V_0 \oplus G(Y_0) \\ N_3 &= Y_0 \oplus U^0 \end{aligned}$$

Anschließend werden für die P^0, P^{17} und P^{18} die X_1. und Y_1-Werte berechnet, um durch die Gleichungen (5.15) Lösungen für M_2 finden. Mithilfe der Konstante M_2 lassen sich drei Lösungen für M_3 finden, wobei alle drei Lösungen übereinstimmen sollten. Diese genannte Bedingung und das Wissen, dass die äußeren 16 Bit null sind, lässt auf das richtige M_2 schließen. Die letzte Konstante N_2 errechnet sich aus W und M_3, zu beachten ist, dass die äußeren 16 Bit ebenfalls null sind. Durch die Berechnung aller Schlüsselkonstanten ist es nun möglich nach der neuformulierten Methode alle Klartextblöcke zu Verschlüsseln und alle Ciphertextblöcke zu Entschlüsseln.
Mit den letzten beiden Gleichungen im Paper, also Gleichung (5.16) und (5.17) besteht die Möglichkeit den Schlüssel zu rekonstruieren. Aufgrund des Zusammenhangs zwischen M_1, N_1, M_3, N_3 und den äußeren 16 Bits von B_3, B_4, B_5, B_6 lassen sich die In- und Outputs der fk-Funktion errechnen.

4 Implementierung

Im folgenden Abschnitt werden wir sowohl auf die Implementierung des *FEAL-4* Verfahren, als auch auf die der Attacke eingehen. Als Programmiersprache wurde C gewählt, da ein kompiliertes Programm eine höhere Performanz besitzt als zum Beispiel ein Java Programm, welches über einen Interpreter läuft.
Die Software wurde in drei Teilkomponenten unterteilt. Einmal die *FEAL* Komponente, welche alle Funktionen besitzt, um die Funktionalität des *FEAL* Verfahrens bereitzustellen. Die nächste Komponente widmet sich ganz der Attacke, welche von Murphy beschrieben wurde. Abschließend wird noch die Verifizierer Komponente vorgestellt. Diese dient zur Verifikation aller Funktionen, die in den beiden anderen Komponenten erstellt wurden.
Bevor wir jedoch die einzelnen Komponenten im Detail betrachten, werden erst einige Grundkonventionen festgelegt. Diese dienen zur Vereinheitlichung und zum besseren Nachvollziehen des Codes im Bezug auf das vorgegebene Paper [1].

4.1 Konventionen

Da Murphy in seinem Paper die meisten Funktionalitäten sowohl des *FEAL*, als auch des Attacke-Algorithmus als mathematische Funktionen dargestellt hat, ist der Übergang von der Theorie in die Implementierung verhältnisweise einfach. Um den Bezug auf die mathematischen Funktionen nicht zu verlieren, wurde die Mehrzahl der Variablennamen eins zu eins übertragen. Ausnahmen waren zum Beispiel die Bytevariablen eines gesplitteten Doppelwortes. Nehmen wir an eine 32-Bit Zahl hätte den Variablennamen *a*. So haben in dem Paper die 4 Teilbytes von *a* einen zusätzlichen fortlaufenden Index, also *a0, a1, a2, a3*. Unsere Implementierung realisiert eine gesplittete 32-Bit Zahl als Byte-Array der Länge 4 und behält dabei den Variablennamen aus dem Paper. Dadurch ähnelt ein Zugriff auf den entsprechenden Index im Array (z.B. *a[1]*) dem aus dem Paper (*a1*). Damit sich der Namensbereich des Arrays und der initialen 32-Bit Zahl nicht überschneidet, wird der initialen Zahl ihre Größenbezeichnung an den Variablennamen des Papers angehangen. In unserem Beispiel hätte die *UInt32* Repräsentation von *a* also den Variablennamen *aDWord*.
Für erstellte Funktionen gelten die selben Namenskonventionen. Falls eine Funktion von Murphy explizit in einer mathematischen Repräsentierung vorhanden ist, wird der Name dieser Funktion übernommen. Beinhaltet der Funktionname einen griechischen Buchstaben (z.B. θ), so wird in der Implementierung der Buchstabe anhand des repräsentativen Wortes aus lateinischen Buchstaben ausgeschrieben (z.B. *theta*). Wird eine Funktion nicht explizit im Paper genannt oder niedergeschrieben, so ist für sie ein adäquater Funktionsname, der die üblichen Programmierkonventionen einhält zu wählen.

4.2 Implementierung explizit formulierter Funktionen

Wie bereits erwähnt werden die meisten Funktionen in dem Paper von Murphy explizit ausformuliert. Um besser nachvollziehen zu können, wie der Implementierungsprozess einer solchen Funktion abläuft, wird nun die Implementierung der Funktion f aus dem Paper [1] durchgeführt.
Wir betrachten dabei zuerst folgenden Auszug, welcher die Definition von f beinhaltet:

Now suppose that $a_i, c_i \; \epsilon \; Z_2^8$ for i = 0,1,2,3, and also that $b_1, b_2 \; \epsilon \; Z_2^8$, with $b = (b_1, b_2) \; \epsilon \; Z_2^{16}$ and $a = (a_0, a_1, a_2, a_3), c \; \epsilon \; Z_2^{32}$ etc., then we can define

$$c = f(a, b)$$

as follows:

$$\begin{aligned} d_1 &= a_0 \oplus a_1 \oplus b_1 \\ d_2 &= a_2 \oplus a_3 \oplus b_2 \\ c_1 &= S_1(d_1, d_2) \\ c_2 &= S_0(d_2, c_1) \\ c_0 &= S_0(a_0, c_1) \\ c_3 &= S_1(a_3, c_2) \end{aligned}$$

Anhand dieser Definition lässt sich auf alle Eigenschaften unserer zu implementierenden Funktion schließen. Wir sehen, das f die Parameter a und b besitzt, wobei a definiert ist als Konkatenation von 4 Bytes und b als eine Konkatenation von 2 Bytes. Des Weiteren können wir erkennen, das $f(a, b) = c$, wobei c als 32-Bit Zahl definiert ist. Durch diese Informationen lässt sich auf folgende Deklaration in C schließen:

```
1  /**
2   * Implementierung der f Funktion aus dem Paper
3   *
4   * c = f(a, b)
5   *
6   * @param aDWord - a
7   * @param b     - b
8   *
9   * @result c (32 bit)
10  */
11 uint32_t f(uint32_t aDWord, uint16_t b);
```

Codebeispiel 1: Deklartation der Funktion f in C

Wir können in der Definition erkennen, das a und b nicht im Ganzen, sondern ihre jeweiligen Bytes verwendet werden. Das heißt, das bevor wir die Operationen aus der Definition implementieren, müssen wir eine Funktion aufrufen, die uns a und b in Bytes aufsplittet. Zudem müssen am Schluss die 4 Bytes $c0, c1, c2, c3$ zu dem Doppelwort c zusammengefügt werden. Dies führt dann zu folgender Implementierung von f:

```c
uint32_t f(uint32_t aDWord, uint16_t b)
{
  uint8_t b2 = (uint8_t) b;
  uint8_t b1 = (uint8_t)(b >> 8);
  uint8_t a[4] = {0};
  uint8_t c0, c1, c2, c3, d1, d2;

  // Split a to a0, a1, a2, a3
  splitToBytes(aDWord, a);

  d1 = a[0] ^ a[1] ^ b1;
  d2 = a[2] ^ a[3] ^ b2;
  c1 = S(d1, d2, ONE);
  c2 = S(d2, c1, ZERO);
  c0 = S(a[0], c1, ZERO);
  c3 = S(a[3], c2, ONE);

  return bytesToUint32(c0, c1, c2, c3);
}
```

Codebeispiel 2: Implementierung der Funktion f in C

Anhand dieser Vorgehensweise wurden alle weiteren explizit ausformulierten Funktionen implementiert. Vorallem die Implementierung des *FEAL*-Verfahren wurde durch diese Vorgehensweise sehr vereinfacht. Interessant ist nun zu betrachten, wie die Attacke implementiert wurde.

4.3 Implementierung der Attacke

In *Krypto-Attacke auf den FEAL Algorithmus mit 20 Plaintextblöcken oder weniger* [1] wird die Attacke hauptsächlich anhand von Prosa geschildert, mit zusätzlichem Bezug auf vorher aufgestellte Gleichungen. Dabei handelt es sich um zwei verschiedene Formen von Gleichungen, die unterschiedliche Arten der Implementierung mit sich ziehen.
Die erste Form sind Gleichungen, wo ein x gesucht wird, welches die Gleichung löst. Die Vorgehensweise bei der Suche nach x ist dabei häufig das Prüfen von anderen Gleichungen, welche mit Bestandteilen von x zusammenhängen. Wenn diese Gleichungen alle erfüllt sind, haben wir eine Lösung für x. Dies führt dazu, das nicht nur eine, sondern mehrere Lösungen für x gefunden werden.
Betrachten wir als Beispiel die Implementierung der Gleichungen (3.7) aus dem Paper [1]:

$$G(x \oplus a) \oplus G(x \oplus b) = d$$
$$G(x \oplus a) \oplus G(x \oplus c) = e$$

Wir werden uns an dieser Stelle nur die Implementierung betrachten, welche verdeutlicht, wie Lösungen für ein x gefunden werden. Die Theorie zu der Implementierung finden Sie in (3.6) des Papers [1].

```
int getSolutionsForXFrom3_7(uint32_t aDWord, uint32_t
bDWord, uint32_t cDWord, uint32_t dDWord, uint32_t eDWord,
uint32_t ** solutions) {

  int solutionCount = 0;  // Anzahl an Loesungen fuer x
  uint8_t a[4] = {0};
  uint8_t b[4] = {0};
  uint8_t c[4] = {0};
  uint8_t d[4] = {0};
  uint8_t e[4] = {0};

  // Allokiere Plaetze, um die Loesungen fuer x zu speichern.
  // Nehmen wir an, das 100% aller z1, z2 die Gleichung (3.2)
      erfuellen.
  // Dann allokieren wir 2^17 * 32 = 4194304 bit = 524288 Byte =
      512 KB im Heap.
  // Damit sollten alle moeglichen Loesungen fuer x in diesem Array
  // gespeichert werden koennen.

  //2^17 hat nicht funktioniert, also 131072 ausgeschrieben...
  uint32_t *tmpPointer = malloc(131072 * sizeof(uint32_t));


  // Split a to a0, a1, a2, a3 (analog fuer b, c, d, e)
  splitToBytes(aDWord, a);
  splitToBytes(bDWord, b);
  splitToBytes(cDWord, c);
  splitToBytes(dDWord, d);
  splitToBytes(eDWord, e);

  uint8_t z1 = 0;
  uint8_t z2 = 0;
  // Check fuer jedes z1, z2...
  for(int i = 0; i < 256; ++i)
  {
    z1 = i;
    for(int j = 0; j < 256; ++j)
    {
      z2 = j;
      // Wir checken fuer beide Gleichungen in (3.7) gleichzeitig
          !!!
      uint8_t alpha1 = S(z1 ^ a[0] ^ a[1], z2 ^ a[2] ^ a[3], ONE);
      uint8_t beta1  = S(z1 ^ b[0] ^ b[1], z2 ^ b[2] ^ b[3], ONE);
      uint8_t gamma1 = S(z1 ^ c[0] ^ c[1], z2 ^ c[2] ^ c[3], ONE);

      if(((alpha1 ^ beta1) != d[1]) || ((alpha1 ^ gamma1) != e[1]))
        continue;

      uint8_t alpha2 = S(alpha1, z2 ^ a[2] ^ a[3], ZERO);
      uint8_t beta2  = S(beta1, z2 ^ b[2] ^ b[3], ZERO);
      uint8_t gamma2 = S(gamma1, z2 ^ c[2] ^ c[3], ZERO);

      if(((alpha2 ^ beta2) != d[2]) || ((alpha2 ^ gamma2) != e[2]))
        continue;

      for(int k = 0; k < 256; ++k)
      {
```

```
55          uint8_t x0 = k;
56          for(int l = 0; l < 256; ++l)
57          {
58            uint8_t x3 = l;
59            uint8_t s0Alpha1 = S(alpha1, x0 ^ a[0], ZERO);
60            if((s0Alpha1 ^ S(beta1, x0 ^ b[0], ZERO)) != d[0])
61              continue;
63            if((s0Alpha1 ^ S(gamma1, x0 ^ c[0], ZERO)) != e[0])
64              continue;
66            uint8_t s1Alpha2 = S(alpha2, x3 ^ a[3], ONE);
67            if((s1Alpha2 ^ S(beta2, x3 ^ b[3], ONE)) != d[3])
68              continue;
70            if((s1Alpha2 ^ S(gamma2, x3 ^ c[3], ONE)) != e[3])
71              continue;
73            // Jede Gleichung fuer z1, z2, x0, x3 ist korrekt.
74            // Errechne x1, x2 (3.4) und speichere die Loesung fuer x
                  ab.
75            uint32_t x = bytesToUint32(x0, z1 ^ x0, z2 ^ x3, x3);
76            tmpPointer[solutionCount] = x;
77            ++solutionCount;
78          }
79        }
80      }
81    }
82    *solutions = realloc(tmpPointer, solutionCount * sizeof(uint32_t)
          );
83    return solutionCount;
84  }
```

Codebeispiel 3: Implementierung zur Suche von x in (3.7) [1]

Wie ab Zeile 32 zu sehen ist, traversieren wir durch mehrere for-Schleifen, wobei jede den Wert einer Komponente ändert, welche mit x zusammenhängt. Für diese Werte wird dann geprüft, ob sie die nötigen Gleichungen erfüllen (z.B. Zeile 43). Falls nicht, kann der Wert für diese Komponente verworfen werden. Ist die Gleichung erfüllt, kann weiter verfahren werden. Sollten alle gewählten Komponentenwerte ihre jeweiligen Gleichungen erfüllen, kann daraus eine Lösung für x generiert werden (Zeile 75). Bei derartigen Funktionen werden die verschiedenen Lösungen für x immer in einer Pointerstruktur gespeichert und die Anzahl der gefunden Lösungen als Return-Wert zurück gegeben.

Die zweite Form von Gleichungen sind sind quasi Assertions. Im Laufe der Attacke sollen an bestimmten Punkten geprüft werden, ob die bisher gesammelten Werte bestimmte Gleichungen erfüllen. Dies dient in erster Linie der Reduktion möglicher Lösungen. Als Beispiel betrachten wir die Assertion der Gleichung (5.5) aus dem Paper [1]:

```
1  /**
2   * Prueft, ob die Parameter Gleichung 5.5 erfuellen:
3   *
4   *  CiL ^ U0 ^ G(PiL ^ V0) ^ G(Di ^ W) = 0    i = 0     (5.5)
5   *
6   * @param CiL
7   * @param trippel
8   * @param PiL
9   * @param Di
10  *
11  * @return 1, wenn erfuellt ; 0, wenn nicht
12  */
13 int doesSatisfy5_5(uint32_t CiL, struct triplet trippel, uint32_t
        PiL,
14  uint32_t Di)
15 {
16   if(( CiL ^ trippel.U0 ^ G(PiL ^ trippel.V0) ^ G(Di ^ trippel.W))
         == 0)
17     return 1;
18   return 0;
19 }
```

Codebeispiel 4: Implementierung der Assertion für Gleichung (5.5) [1]

Wie zu sehen ist, handelt sich dabei um eine einfache if-Abfrage, welche die Gleichung repräsentiert. In Zeile 13 wird zum ersten mal die neue Datenstruktur *triplet* genannt. Diese ist ein für die Attacke entwickelte Struktur, welche auf den Gleichungen (5.3) [1] beruht:

```
1  /*
2   * Struct fuer das tripel, welches in 5.3 vorgestellt wird
3   */
4  struct triplet{
5    uint32_t W;
6    uint32_t V0;
7    uint32_t U0;
8  };
```

Codebeispiel 5: Datenstruktur für die Werte aus (5.3) [1]

Vorteil dieser Datenstruktur ist eine konsistentere Speicherung von zusammenhängenden W, V^0 und U^0 Werten. Zusätzlich erleichtert es die Nachvollziehbarkeit innerhalb des Codes.

Ein weiterer wichtiger Teil der Attacke ist die Wahl der Plaintexte. Schwierig war dabei 64-Bit Pseudo Zufallszahlen zu generieren, denn die C eigene Zufallszahlenfunktion *rand()* liefert nur Zahlen im Bereich von 0 bis *RAND_MAX*, welches mindestens 32767 ist. Um nun eine 64-Bit Pseudo Zufallszahl zu generieren wurde die *rand()* Funktion vier mal aufgerufen und die resultierenden Werte durch bit-shift und xor Operationen zu einer 64-Bit Zahl zusammen gefügt. Der folgende Ausschnitt ist ein Beispiel für eine solche Generierung:

```
1 P[0]   = ((uint64_t)rand() << 48)^ ((uint64_t)rand() << 32) ^
2     ((uint32_t)rand() << 16) ^ ((uint32_t)rand());
```

Codebeispiel 6: Generierung einer 64 bit Pseudo Zufallszahl

Das Herzstück unserer Attacke ist die *attack()* Funktion. Sie beinhaltet einen gesamten Durchlauf einer Attacke, vom Wählen der Plaintexte bis hin zum Berechnen der Schlüsselkonstanten. Die Attacke wurde bewusst nicht zu fein aufgesplittet, um den Weg, der in dem Paper [1] beschrieben ist, noch nachvollziehen zu können. Wir werden an dieser Stelle nicht auf die detaillierte Implementierung der *attack()* Funktion eingehen. In unserem Fallbeispiel werden wir die komplette Funktion durchlaufen und an wichtigen Stellen Codeausschnitte liefern, welche in Summe eine ausreichende Erläuterung zur Implementierung sein sollten. Doch bevor wir das Fallbeispiel betrachten, müssen wir zunächst noch auf die dritte Komponente der Software eingehen, den Verifizierer.

4.4 Implementierung des Verifizierers

Der Verifizierer stellt in unserer Software eine Kontrollinstanz dar. Um sicher zu stellen, dass jede Funktion, die für *FEAL* oder die Attacke geschrieben wurde korrekt funktioniert, wurde im Verifizierer jeweils eine Testfunktion hinterlegt. Jede Testfunktion prüft, ob die zu prüfende Funktion richtig agiert. Es kann auf verschiedene Arten geprüft werden.
Bei den Funktionen, die einen Wert zurücklieferen sollen, wird vorher ein erwartetes Ergebnis gespeichert. Dieses wird dann mit dem Ergebnis, welches die zu prüfende Funktion zurück gibt verglichen. Nur wenn erwartetes und tatsächliches Ergebnis gleich sind, gilt die Funktion als verifiziert.
Betrachten wir uns als Beispiel die Funktion f aus dem Codebeispiel 2. f gibt für ein bekanntes a und b eine 32-Bit Zahl zurück. Das heißt für unsere Testfunktion, das wir ein a und b festlegen, die f Funktion anhand des Papers [1] unabhängig von der zu testenden Implementierung ausführen und dieses Ergebnis als Erwartung voraussetzen. Dann wird die zu testende Funktion mit den Parametern a und b aufgerufen und dieses Ergebnis gespeichert. Sollten nun das erwartete und tatsächliche Ergebnis gleich sein, gilt f als verifiziert. Codebeispiel 7 zeigt die Implementierung einer solchen Testfunktion:

```c
int verifyFunctionF(int withOutput)
{
  uint32_t a = 0x12345678;
  uint16_t b = 0xbcde;
  uint32_t expected = 0x012e78c7;
  uint32_t result = f(a,b);

  if(withOutput)
  {
    printf("Test f mit a = 0x%" PRIx32", b = 0x%"PRIx32 ". Expected
        : 0x%"PRIx32" Result: 0x%"PRIx32"\n",
        a, b, expected, result);
  }
  if(expected != result)
    return 0;
  return 1;
}
```

Codebeispiel 7: Verifizierung der Funktion f

Andere Funktionen können mit Hilfe bereits verifizierter Funktionen auf ihre Korrektheit geprüft werden. Nehmen wir als Beispiel die *decode*() Funktion für das *FEAL*-Verfahren. Wenn die *encode*() Funktion bereits verifiziert ist, lässt sich die Richtigkeit für *decode*() einfach zeigen. Sollte *decode*() Ciphertextblöcke, die von *encode*() verschlüsselt wurden, wieder in die ursprünglichen Plaintexte dekodieren können, so gilt *decode*() als verifiziert. Codebeispiel 8 zeigt die Implementierung der Testfunktion für *decode*():

```c
int verifyFunctionDecode(int withOutput)
{
  // Testschlüssel
  uint64_t key = 0xFF00FF00FF00FF00;

  // Zuerst werden die 12 16 bit subkeys errechnet
  uint16_t *subkeys = compSubKeys(key);

  // Danach werden 20 Plaintexte nach der Definition aus dem Paper
  // erzeugt.
  uint64_t *P = choosePlainTexts();

  // Allokiere Speicher fuer die Ciphertextbloecke und wende das
  // FEAL-Verfahren zur Verschluesselung an.
  uint64_t *C = malloc(20 * sizeof(uint64_t));
  for(int i = 0; i < 20; ++i)
  {
    C[i] = encode(P[i], subkeys);
  }

  // Entschluessel die 20 Ciphertextbloecke und
  // vergleiche, ob sie mit den urspruenglichen Plaintextbloecken
  // uebereinstimmen.
  uint64_t decodedP[20];
  int isEqual = 1;
  for(int i = 0; i < 20; ++i)
  {
```

```
28      decodedP[i] = decode(C[i], subkeys);
29      if(decodedP[i] != P[i])
30      {
31        isEqual = 0;
32      }
33      if(withOutput)
34      {
35        printf("Urspruenglicher Plaintext: 0x%" PRIx64 "\t", P[i]);
36        printf("Dekodierter Plaintext: 0x%" PRIx64"\n", decodedP[i]);
37      }
38    }

40    return isEqual;
41  }
```

Codebeispiel 8: Verifizierung der Funktion *decode*()

Auf diese Weise ist die Suche nach Fehlern in der späteren fertigen Attacke um einiges leichter, da man bestimmte aufgerufene Funktionen anhand ihrer Verifizierung ausschließen kann.

Alle in diesem Kapitel genannten Maßnahmen führten letztendlich zu einem fertigen Programm, welches erfolgreich die Krypto-Attacke auf das *FEAL*-Verfahren durchführen kann. Im nächsten Kapitel wird nun ein Fallbeispiel einer solchen Attacke erläutert.

5 Fallbeispiel

Wir werden nun ein Beispiel für die Attacke auf *FEAL-4* mit 20 Plaintextblöcken verfolgen. Dabei werden wir uns die gesamte Attacke über in der in Abschnitt 4.3 vorgestellten *attack*() Funktion befinden. Jeder wichtige Abschnitt der Attacke wird anhand eines Code-Ausschnitts der *attack*() Funktion erläutert. Zusätzlich werden noch die bis zu diesen Zeitpunkt wichtigen gesammelten Komponenten zur Schlüsselkonstantenerzeugung aufgelistet.

Bevor die Attacke starten kann, müssen zunächst alle dafür benötigten Bestandteile gewählt werden. Zuerst muss ein 64-Bit Schlüssel gewählt werden:

```
1 // Testschlüssel
2 uint64_t key = 0xFF00FF00FF00FF00;
```

Codebeispiel 9: Wahl des Schlüssels

Danach wird aus dem Schlüssel die 12 16-Bit Subschlüssel generiert:

```
1 // Zuerst werden die 12 16 bit subkeys errechnet
2 uint16_t *subkeys = compSubKeys(key);
```

Codebeispiel 10: Generieren der Subschlüssel

Da bei der Attacke von 20 gewählten Plaintextblöcken und deren entsprechenden Ciphertextblöcken ausgegangen wird, müssen die Plaintexte gewählt und dann mit Hilfe der Subschlüssel verschlüsselt werden:

```
1  // Danach werden 20 Plaintexte nach der Definition aus dem Paper
       erzeugt.
2  uint64_t *P = choosePlainTexts();
3
4  // Allokiere Speicher fuer die Ciphertextbloecke und wende das
5  //  FEAL-Verfahren zur Verschluesselung an.
6  uint64_t *C = malloc(20 * sizeof(uint64_t));
7  for(int i = 0; i < 20; ++i)
8  {
9    C[i] = encode(P[i], subkeys);
10 }
```

Codebeispiel 11: Generieren der Plain- und Ciphertextblöcke

Da laut Definition der 19. und 20. Plaintextblock zufällig gewählt werden, wurden beide so gewählt, dass sie konkateniert den String 'FEAL is safe!!' repräsentieren. Am Ende unserer Attacke versuchen wir diesen String, durch Entschlüsseln des 19. und 20. Ciphertextblocks mittels unserer gefundenen Schlüsselkonstanten, wieder zu erlangen.

Zusätzlich müssen noch die Q-Werte für jeden Plaintextblock und D-Werte für jeden Ciphertextblock berechnet werden. Beide Werte repräsentieren eine xor Operation aus der linken und rechten Hälfte des entsprechenden Textblocks. Da die linken und rechten Hälften der Textblöcke im Laufe der Attacke noch häufiger verwendet werden, werden diese auch seperat abgespeichert:

```
for(int i = 0; i < 20; ++i)
{
  CL[i] = (uint32_t)(C[i]>>32);
  CR[i] = (uint32_t)(C[i]);
  PL[i] = (uint32_t)(P[i]>>32);
  PR[i] = (uint32_t)(P[i]);
  D[i]  = CL[i] ^ CR[i];
  Q[i]  = PL[i] ^ PR[i];
}
```

Codebeispiel 12: Speichern der Q- und D-Werte

Nun beginnt die wahre Attacke. An dieser Stelle wird nicht detailliert auf die entsprechenden Gleichung aus dem Paper [1] eingegangen. Es werden lediglich im Paper vorhandene Nummer für die entsprechende Gleichung genannt. Dieses Fallbeispiel soll nur dazu dienen nachvollziehen zu können, wann nach welchen Werten gesucht wird, um in der Attacke voran zu schreiten. Für tieferen Einblick in die Zusammenhänge der einzelnen Komponenten empfehlen wir das Lesen der Quelle [1].

Wir beginnen die Attacke mit der Suche nach möglichen W Werten. Die Gleichung (5.8) entspricht dem Format der Gleichung (3.7). Die entsprechende Funktion kann Lösungen für diese Form von Gleichungen liefern:

```
// Finde W
uint32_t d = CL[0] ^ CL[1] ^ 0x02000000;
uint32_t e = CL[0] ^ CL[2] ^ 0x00000002;
uint32_t *wSolutions = NULL;

uint32_t wSolutionsCount = getSolutionsForXFrom3_7(D[0], D[1],
 D[2], d, e, &wSolutions);
wSolutionsCount = getSolutionsFor5_9(D[0], D[3], D[4], CL[0],
 CL[3], CL[4], &wSolutions, wSolutionsCount);
```

Codebeispiel 13: Suche nach möglichen W Werten

Der zweite Funktionsaufruf in Codebeispiel 13, $getSolutionsFor5\_9$, verkleinert die Anzahl an Lösungen für W, indem geprüft wird, ob die gefundenen W Werte die Gleichungen (5.9) erfüllen. Die resultierende Ergebnismenge hat in der Regel bis zu zehn Lösungen für W. Der folgende Output sind die Werte, die in unserem Beispieldurchlauf gefunden wurden:

```
8 Solutions for W:
0x1b73b24a
0x1b73a25a
0x1b7332ca
0x1b7322da
0x9bf3b24a
0x9bf3a25a
0x9bf332ca
0x9bf322da
```

Als nächstes werden für jede gefundene Lösung für W Lösungen für V^0 gesucht. Dies geschieht mit Hilfe der Gleichungen (5.10), die wiederum der Form (3.7) entsprechen. Durch die Gleichung (5.5) für den ersten Plaintextblock kann dann ein Wert für U^0 aus den gefundenen W und V^0 Kombinationen errechnet werden. Erfüllen diese nun die Gleichung (5.5) für die nächsten zehn Plaintextblöcke, so haben wir eine mögliche Lösung für unser U^0, V^0, W Tripel:

```
// Finde V0
uint32_t *v0Solutions;
uint32_t v0SolutionCount;
struct triplet *triplets = NULL;
int tripletsCount = 0;

for(int i = 0; i < wSolutionsCount; ++i)
{
  d = CL[0] ^ CL[5] ^ G(D[0] ^ wSolutions[i]) ^ G(D[5] ^
    wSolutions[i]);
  e = CL[0] ^ CL[6] ^ G(D[0] ^ wSolutions[i]) ^ G(D[6] ^
    wSolutions[i]);
  v0SolutionCount  = getSolutionsForXFrom3_7(PL[0], PL[5], PL[6],
    d, e, &v0Solutions);

  triplets = realloc(triplets, (tripletsCount + v0SolutionCount) *
      sizeof(struct triplet));
  for(int j = 0; j < v0SolutionCount; ++j)
  {
    // Berechne dazugehöriges U0 (= CL0 ^ G(PL0 ^ V0) ^ G(D0 ^ W))
    struct triplet triple = getTripletFrom5_5(CL[0], PL[0],
        v0Solutions[j], D[0], wSolutions[i]);

    // Adde nur die Tripel, die fuer die anderen Plaintexte (5.5)
    // erfuellen
    if(doesSatisfy5_5ForOtherPlaintexts(triple, PL, CL, D))
    {
      triplets[tripletsCount] = triple;
      tripletsCount++;
    }
  }
}
```

Codebeispiel 14: Suche nach möglichen Tripeln

In unserem Durchlauf wurden die folgenden Tripellösungen gefunden:

```
16 Solutions for Triplets:
U0: 0xd72bf37 V0: 0x4fc3d634 W: 0x1b73b24a
U0: 0xd72bf35 V0: 0x4fc356b4 W: 0x1b73b24a
U0: 0xf72bf37 V0: 0xcf43d634 W: 0x1b73b24a
U0: 0xf72bf35 V0: 0xcf4356b4 W: 0x1b73b24a
U0: 0xd72bf35 V0: 0x4fc3d634 W: 0x1b7332ca
U0: 0xd72bf37 V0: 0x4fc356b4 W: 0x1b7332ca
U0: 0xf72bf35 V0: 0xcf43d634 W: 0x1b7332ca
U0: 0xf72bf37 V0: 0xcf4356b4 W: 0x1b7332ca
```

Im nächsten Schritt versuchen wir die U und V Werte für die Plaintextblöcke 12, 13, 14 und 15 zu finden. U^{12} und U^{14} lassen sich dabei einfach anhand der Gleichungen (5.11) berechnen. Zusätzlich zu (5.11) kann man durch (4.4) darauf schließen, dass $U^{12} = U^{13}$ und $U^{14} = U^{15}$ ist. Durch die errechneten U Werte und die Gleichungen (5.12), die die Form (3.1) besitzen, lassen sich Lösungen für V^{12} und V^{14} finden:

```
for(int i = 0; i < tripletsCount; ++i)
{
  uint32_t U12 = triplets[i].U0 ^ Q[0] ^ Q[12];
  uint32_t U13 = U12;
  uint32_t U14 = triplets[i].U0 ^ Q[0] ^ Q[14];
  uint32_t U15 = U14;

  uint32_t *V12Solutions = NULL;
  int V12SolutionsCount = getSolutionsForXFrom3_1(PL[12], G(D[12] ^
      triplets[i].W) ^ CL[12] ^ U12, &V12Solutions);
  uint32_t *V14Solutions = NULL;
  int V14SolutionsCount = getSolutionsForXFrom3_1(PL[14], G(D[14] ^
      triplets[i].W) ^ CL[14] ^ U14, &V14Solutions);
```

Codebeispiel 15: Lösungen für V^{12} und V^{14}

Genau so wie bei U, sollte auch $V^{12} = V^{13}$ und $V^{14} = V^{15}$ sein. Für jedes V^{12} und V^{14} kann mit der Gleichung (5.4) geprüft werden, ob diese Bedingung zutrifft. Erfüllen beide die Gleichung, so können mit den gesammelten Werten die Schlüsselkonstanten errechnet werden:

```
for(int j = 0; j < V12SolutionsCount; ++j)
{
  if(doesSatisfy5_4(CL[13], U13, PL[13], V12Solutions[j], D[13],
      triplets[i].W))
  {
    for(int k = 0; k < V14SolutionsCount; ++k)
    {
      if(doesSatisfy5_4(CL[15], U15, PL[15], V14Solutions[k],
          D[15], triplets[i].W))
      {
        //U, V fuer Plaintextbloecke 0-15 speichern
        uint32_t *U = malloc(20 * sizeof(uint32_t));
        uint32_t *V = malloc(20 * sizeof(uint32_t));
        for(int l = 0; l < 12; ++l)
        {
          U[l] = triplets[i].U0;
          V[l] = triplets[i].V0;
        }
        U[12] = U12; U[13] = U13; U[14] = U14; U[15] = U15;
        V[12] = V12Solutions[j];
        V[13] = V12Solutions[j];
        V[14] = V14Solutions[k];
        V[15] = V14Solutions[k];
        // Key Konstanten berechnen.
        uint32_t *calculatedKeyConstants =
            calculateKeyConstants(PL,PR,CL,CR,Q,D,U,V,
                triplets[i].W);
```

Codebeispiel 16: Prüfen, ob V^{12}, V^{14} (5.4) erfüllen

Zur Berechnung der 6 Schlüsselkonstanten gehen wir wie folgt vor. Zuerst finden wir anhand der Gleichung (5.13) Lösungen für N_1. Wir berechnen mittels der Struktur von Gleichung (5.13) V^{16} und U^{16}. Wenn diese Werte die Gleichung (5.4) erfüllen, behalten wir diese Lösung für N_1:

```
// Finde Loesungen fuer N1
uint32_t *N1Solutions = NULL;
uint32_t N1SolutionsCount = getSolutionsForXFrom3_7(Q[0],Q[12],
  Q[14],V[0] ^ V[12], V[0] ^ V[14], &N1Solutions);

for(int i = 0; i < N1SolutionsCount; ++i)
{
  // Berechne V16. Entspricht der Struktur von (5.13)
  // nach V16 aufgeloest.
  uint32_t V16 = G(Q[0] ^ N1Solutions[i]) ^ G(Q[16] ^
  N1Solutions[i]) ^ V[0];
  uint32_t U16 = U[0] ^ Q[0] ^ Q[16];
  // Ueberpruefe, ob (5.4) mit V16 erfuellt ist.
  if(!doesSatisfy5_4(CL[16],U16,PL[16],V16,D[16],W))
    continue;
```

Codebeispiel 17: Finde Lösungen für N_1

Anhand von N_1 lassen sich die Werte für M_1 und N_3 wie folgt errechnen:

```
  // Mittels N1 lassen sich M1 und N3 errechnen.
  uint32_t y0 = Q[0] ^ N1Solutions[i];
  uint32_t m1 = V[0] ^ G(y0);
  uint32_t n3 = y0 ^ U[0];
```

Codebeispiel 18: Berechnen von M_1 und N_3

Wir berechnen nun die X_1 und Y_1 Werte für die Plaintexte 0, 17 und 18. Mit diesen Werten kann man durch die Gleichungen (5.15) Lösungen für M_2 finden:

```
    // Finde Loesungen fuer M2.
    uint32_t x1_0  = PL[0] ^ m1 ^ G(y0);
    uint32_t x1_17 = PL[17] ^ m1 ^ G(Q[17] ^ N1Solutions[i]);
    uint32_t x1_18 = PL[18] ^ m1 ^ G(Q[18] ^ N1Solutions[i]);
    uint32_t y1_0  = y0 ^ G(x1_0);
    uint32_t y1_17 = Q[17] ^ N1Solutions[i] ^ G(x1_17);
    uint32_t y1_18 = Q[18] ^ N1Solutions[i] ^ G(x1_18);

    uint32_t d = x1_0 ^ x1_17 ^ D[0] ^ D[17];
    uint32_t e = x1_0 ^ x1_18 ^ D[0] ^ D[18];

    uint32_t *M2Solutions = NULL;
    int M2SolutionsCount = getSolutionsForXFrom3_7(y1_0,y1_17,
      y1_18,d,e,&M2Solutions);
```

Codebeispiel 19: Finden von Lösungen für M_2

Als nächstes prüfen wir für jedes mögliche M_2, ob die äußeren 16 Bit null sind. Ist dies der Fall, errechnen wir mit dem M_2 Wert drei verschiedene Werte für M_3 aus. Da M_3 eine Konstante ist, sollten alle drei Werte übereinstimmen:

```
for(int j = 0; j < M2SolutionsCount; ++j)
{
  // Check, ob die auesseren 16 bit 0 sind.
  if((M2Solutions[j] & 0xFF0000FF) != 0)
    continue;

  // Mit Hilfe von M2 drei Werte fuer M3 schreiben,
  // die uebereinstimmen sollten.
  uint32_t x2_0  = x1_0 ^ G(y1_0 ^ M2Solutions[j]);
  uint32_t x2_17 = x1_17 ^ G(y1_17 ^ M2Solutions[j]);
  uint32_t x2_18 = x1_18 ^ G(y1_18 ^ M2Solutions[j]);
  uint32_t m3_0  = D[0] ^ x2_0;
  uint32_t m3_17 = D[17] ^ x2_17;
  uint32_t m3_18 = D[18] ^ x2_18;

  if((m3_0 != m3_17) || (m3_17 != m3_18))
    continue;
```

Codebeispiel 20: Berechnen von M_3

Nun fehlt nur noch die Schlüsselkonstante N_2. Diese lässt sich mittels W und M_3 berechnen, wobei die äußeren 16 Bit null sein sollten:

```
// N2 berechnen und pruefen, ob die aeusseren 16 Bit 0 sind.
uint32_t n2 = W ^ m3_0;

if((n2 & 0xFF0000FF) != 0)
  continue;
```

Codebeispiel 21: Berechnen von N_2

Da jetzt alle Schlüsselkonstanten gefunden wurden, können wir versuchen unseren Text vom Anfang aus den letzten beiden Ciphertextblöcken wieder zu erlangen. Der folgende Output zeigt die gefundenen Schlüsselkonstanten im Vergleich mit den durch Wissen des Schlüssels errechneten, sowie der Dekodierung der letzten beiden Ciphertextblöcken:

```
Possible Key Constants:
M1: 0x5621c0cc  Berechnet: 0x5621c0cc
N1: 0xcc1ce1a   Berechnet: 0xcc1ce1a
M2: 0x40ef00    Berechnet: 0x40ef00
N2: 0x44f200    Berechnet: 0x44f200
M3: 0x1b37c0ca  Berechnet: 0x1b37c0ca
N3: 0xb227bb0   Berechnet: 0xb227bb0
FEAL is safe!!
```

Die Laufzeit der kompletten Attacke beträgt im Schnitt etwa fünf Sekunden. In Murphy's Paper [1] wird eine Laufzeit von bis zu zehn Stunden datiert. Das ist der Tatsache geschuldet, dass das Paper im Jahre 1990 erschienen ist. Die Diskrepanz zwischen der damaligen Laufzeit mit der heutigen führt uns wieder einmal den rasanten Fortschritt im Bereich des Computerbaus vor Augen.
Im nächsten Kapitel werden die Schwierigkeiten und Herausforderungen, die wir während des Projekts bewältigen mussten, erläutert.

6 Schwierigkeiten und Herausforderungen

Es ist klar, das ein Projekt, welches einen Angriff auf ein Krypto-Verfahren vorstellt, nicht trivial ist. Das führte während der Bearbeitungszeit zu einigen Schwierigkeiten und Herausforderungen, die nun in diesem Kapitel vorgestellt werden.

6.1 Unbekannte Konventionen

Innerhalb des Papers wurden häufig Ausdrücke der folgenden Art genannt:

$$a = (a_0, a_1, a_2, a_3)$$

In diesem Beispiel sollte eine 32-Bit Zahl a in 4 Bytes aufgesplittet werden. Nun wurde jedoch an keiner Stelle erläutert, ob a_0 das höchstgewichtete oder das niedrigstgewichtete Byte darstellt. Wir sind von letzterem ausgegangen, was sich als Fehler herausstellte. Die Folge war, das zwar die Implementierung von *FEAL-4* richtig zu funktionieren schien, jedoch die Attacke aufgrund falscher Zusammenhänge keine Lösungen finden konnte. Erst das Betrachten des folgenden Ausdruck gab Klarheit:

$$C = (C_L, C_R)$$

Dieser Ausdruck beschreibt das Teilen einer 64-Bit Zahl C in ihre zwei 32-Bit Hälften. Die Struktur ist die selbe, wie in dem Ausdruck davor mit a. Wir sehen, dass die linke, also die höher gewichtete, Hälfte von C als erstes Element in der Klammer steht. Dies ließ uns darauf schließen, das a_0 tatsächlich das höchstgewichtete Byte von a sein muss.

6.2 Erschwerte Fehlersuche

In einem Krypto-Verfahren werden die meisten Operationen auf Bit-Ebene durchgeführt. Dort können sich schnell Fehler einschleichen, die nur schwer auffindbar sind. Und der Fakt, das die meisten Funktionen dem Zweck dienen ihren Input zu verschlüsseln, trägt der Fehlersuche nicht gerade positiv bei.

Abhilfe hat da der Verifizierer geleistet, der in Abschnitt 4.4 vorgestellt wurde. Denn nur so konnte garantiert werden, das alle Funktionen korrekt laufen und nicht die Fehlerquelle darstellen können. Tasächlich wurde der Verifizierer erst in der zweiten Iteration unserer Entwicklung hinzugefügt, nachdem wir die Suche nach einem Fehler nach mehreren Wochen aufgegeben hatten. Durch das Verifizieren wurde uns aber bewusst, das der Fehler nicht von uns, sondern dem Paper ausging.

6.3 Fehler in der Quelle

Nach mehreren Wochen der Suche nach einem Fehler in unserer Implementierung sind wir auf einen Fehler innerhalb der Papers von Murphy gestoßen. In einem Teil der Attacke behauptet er:

$$V^{12} = V^{13}$$
$$V^{14} = V^{15}$$

Die zweite Behauptung war in unseren Durchläufen der Attacke nie gegeben. Um den Fehler zu finden, müssen wir zu erst wissen, wie V^i berechnet wird:

$$V^i = M_1 \oplus G(P_L^i \oplus P_R^i \oplus N_1) = M_1 \oplus G(Q^i \oplus N_1)$$

Dabei sind M_1 und N_1 Schlüsselkonstanten. Das heißt der Zusammenhang von V Werten muss über die Plaintextblöcke erfolgen. In Gleichung (4.4) im Paper gibt Murphy beim Wählen der Plaintexte vor:

$$P_R^{15} = P_L^{15} \oplus Q^{13}$$

Das führt wiederum zu folgenden Gleichungen für V^{15}:

$$\begin{aligned} V^{15} &= M_1 \oplus G(P_L^{15} \oplus P_L^{15} \oplus Q^{13} \oplus N_1) \\ &= M_1 \oplus G(Q^{13} \oplus N_1) \\ &= M_1 \oplus G(P_L^{13} \oplus P_R^{13} \oplus N_1) = V^{13} \end{aligned}$$

Das bedeutet, wenn wir den Zusammenhang $V^{14} = V^{15}$ beabsichtigen, muss P_R^{15} folgendermaßen gebildet werden:

$$P_R^{15} = P_L^{15} \oplus Q^{14}$$

Nach dieser Korrektur konnte unser Implementierung auch endlich eine erfolgreiche Attacke durchführen.

7 Konklusion

Die Arbeit an diesem Projekt war für uns sehr aufschlussreich. Eine praktische Erfahrung zu dem bereits vorhandenen theoretischen Wissen hat unseren Einblick in das Thema Kryptographie noch weiter geschärft. Das Einarbeiten und Implementieren einer uns vorher noch unbekannten Attacke gab uns sehr viel Aufschluss über die Herangehensweise der Entwicklung einer Attacke gegen ein Krypto-Verfahren.
Zusammenfassend lässt sich sagen, dass das Projekt ein Erfolg war und wir durchweg positive Erfahrungen mitnehmen.

Abbildungsverzeichnis

Quellen

[1] S. Murphy, "The cryptanalysis of feal-4 with twenty chosen plaintexts," *Journal of Cryptology. 2*, vol. Nr. 3, Januar 1990.

[2] A. Shimizu and S. Miyaguchi, "Fast data encipherment algorithm feal," *Advances in Crptology - Eurocrypt 87, Lecture Notes in Computer Science 304.*, 1987.

[3] B. D. Boer, "Cryptoanalysis of feal," *Advances in Cryptology – Eurocrypt 88, Lecture Notes in Computer Science, Vol. 330*, 1989.