Implementierung der FEAL-Differentrial-Cryptanalysis Attacke nach Murphy

Lukas Becker Juri Golanov

August 30, 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

3 Attacke nach Murphy

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 gibt es noch eine Verifizierer Komponente. Diese dient zur Verfikation aller Funktionen, die in den beiden anderen Komponenten erstellt wurden.
Bevor wir jedoch die einzelnen Komponenten uns 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. Damit 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:

```c
/**
 * Implementierung der f Funktion aus dem Paper
 *
 * c = f(a, b)
 *
 * @param aDWord - a
 * @param b      - b
 *
 * @result c (32 bit)
 */
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, 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:

```
/*
 * Struct fuer das tripel, welches in 5.3 vorgestellt wird
 */
struct triplet{
  uint32_t W;
  uint32_t V0;
  uint32_t U0;
};
```

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:

```
P[0]   = ((uint64_t)rand() << 48)^ ((uint64_t)rand() << 32) ^
    ((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)
  {
```

```
    decodedP[i] = decode(C[i], subkeys);
    if(decodedP[i] != P[i])
    {
      isEqual = 0;
    }
    if(withOutput)
    {
      printf("Urspruenglicher Plaintext: 0x%" PRIx64 "\t", P[i]);
      printf("Dekodierter Plaintext: 0x%" PRIx64"\n", decodedP[i]);
    }
  }

  return isEqual;
}
```

Codebeispiel 8: Verifizierung der Funktion *decode*()

Auf diese Weise ist 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:

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

Codebeispiel 9: Wahl des Schlüssels

Danach wird aus dem Schlüssel die 12 16 bit Subschlüssel generiert:

```
// Zuerst werden die 12 16 bit subkeys errechnet
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:

```
// 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);
}
```

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 aeusseren 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: Berchnen 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 wieder einmal den rasanten Fortschritt im Bereich des Computerbaus vor.
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 ein keiner Stelle erläutert, ob a_0 das höchstgewichtete oder das niedrigstgewichtete Byte darstellt. Wir sind von letzterem ausgegangen, was sich als Fehler heraus stellte. 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 bei 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

References

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