Implementierung und Simulation von Cache-Angriffen auf AES

Rafael Funke

15. September 2006

Erklärung:

Ich versichere, dass ich diese Arbeit selbstständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel angefertigt habe.

Paderborn, 15. September 2006

Rafael Funke

1 Einleitung

Der Advanced Encryption Standard (AES) ist ein modernes symmetrisches Kryptosystem. Generell kann AES als sicher angesehen werden. Das heißt, es ist nicht davon auszugehen, dass man eine grundsätzliche Schwäche in diesem System entdecken wird. Es werden jedoch bei Kryptosystemen immer wieder Schwächen entdeckt, die nur unter sehr speziellen Bedingungen eintreten, wie zum Beispiel Schwächen, die nur auftreten, wenn der Schlüssel eine bestimmte Eigenschaft hat. Solche Schwächen nennt man Seitenangriffe.

Cache-Angriffe auf AES sind auch Seitenangriffe. Cache-Angriffe sind nur dann möglich, wenn ein Angreifer die Möglichkeit hat auf dem selben Rechner Programme auszuführen, auf dem auch die AES-Implementierung läuft. Dabei wird die Tatsache ausgenutzt, dass jedes Programm, welches Speicherzugriffe verursacht, ein Speicherzugriffsmuster im Cache hinterlässt, welches von anderen Programmen über Umwege ausgelesen werden kann. Dieses Speicherzugriffsmuster gibt wenige Informationen über Speicheradressen preis. Im Fall von AES lässt das je nach Implementierung Rückschlüsse auf den geheimen Schlüssel zu. Wie wir sehen werden, ist eine sehr effiziente AES-Implementierung und daher auch sehr verbreitete Implementierung besonders anfällig für Cache-Angriffe.

Im Rahmen dieser Studienarbeit habe ich eine Simulationsumgebung für Cache-Angriffe auf AES geschrieben. Die Motivation für das schreiben einer solchen Simulationsumgebung ist, dass Cache-Angriffe in der Praxis schwer zu realisieren sind und Störfaktoren die bei einem solchen Angriff gewonnenen Daten verfälschen. Es ist nur mit viel Aufwand möglich auch in der Praxis brauchbare Daten zu bekommen. In der Simulation hingegen lassen sich ideale Daten produzieren.

In Kapitel 2 werden wir den AES-Algorithmus näher betrachten und seine Funktionsweise beschreiben. Dann werden wir sehen, wie sich der Algorithmus in Implementierungen optimieren lässt. Danach werden wir sehen, wie der Cache funktioniert und welche Eigenschaften er hat.

In Kapitel 3 werden wir Cache-Angriffe betrachten und ihre Funktionsweise kennenlernen. Wir werden zuerst sehen, welche Informationen ein Cache-Angriff im Allgemeinen liefert, und danach, wie wir diese Informationen einsetzen können, um in AES den geheimen Schlüssel zu bestimmen. Dazu werden wir zwei Angriffe betrachten. Der erste Angriff ist ein sehr naheliegender Ansatz. Er liefert uns zwar Teile des geheimen Schlüssels, ermöglicht es uns aber nicht den kompletten Schlüssel zu bestimmen. Der zweite Angriff ist als Fortsetzung des ersten Angriffs gedacht und ermöglicht uns mit Hilfe der Ergebnisse des ersten Angriffs in mehreren Schritten jeweils weitere Teile des Schlüssels zu bestimmen, um schließlich den vollständigen Schlüssel zu bestimmen.

In Kapitel 4 werden wir die Simulationsumgebung kennenlernen, die im Rahmen dieser Studienarbeit entstanden ist. In dem Kapitel wird zunächst erläutert, was die Simula-

tion eines Cache-Angriffs ist und welche Anforderungen sie zu erfüllen hat. Im letzten Abschnitt des Kapitels werden dann einige Implementierungsdetails erläutert.

Kapitel 2 und 3 basieren auf den Ergebnissen verschiedener Veröffentlichungen. Kapitel 4 basiert auf den Ergebnissen dieser Studienarbeit.

2 Grundlagen

In diesem Kapitel beschreiben wir die benötigten Grundlagen. Im ersten Abschnitt beschreiben wir den Advanced Encryption Standard (AES) und im zweiten Abschnitt beschreiben wir den Cache. Zunächst führen wir aber erst eine Notation ein:
Wir bezeichnen die bitweise XOR-Operation mit $\oplus$. Für zwei Bits a, b sei $a \oplus b$ das Ergebnis von a XOR b. Für zwei Bytes c und d, bestehend aus den Bits c_i, d_i für $0 \leq i \leq 7$ sei das Ergebnis $x = c \oplus d$ ein Byte definiert als $x_i = c_i \oplus d_i$.

2.1 AES

Der Advanced Encryption Standard, kurz AES, wurde im Jahr 2001 festgelegt. Ziel war es, ein Kryptosystem zu finden, welches das damals am weitesten verbreitete symmetrische Kryptosystem DES ablösen sollte. Dazu führte man einen Wettbewerb durch, bei dem jeder die Möglichkeit hatte einen Vorschlag einzureichen. Gewinner dieses Wettbewerbs wurde der „Rijndael"-Algorithmus von Joan Daemen und Vincent Rijmen und wurde damit zum AES-Standard [fip01].

Die Informationen in diesem Kapitel basieren auf den Veröffentlichungen „AES Proposal: Rijndael" [DR99], FIPS-197 [fip01], [Sti02] und [DR02]

Mathematische Grundlagen Rechenoperationen in AES werden im Körper $GF(2^8)$ mit $2^8 = 256$ Elementen durchgeführt. Für diesen Körper gibt es mehrere Darstellungsmöglichkeiten, die alle gleichwertig, also isomorph zueinander sind. Wir werden im Folgenden die Darstellungsform als Polynomkörper verwenden, die auch in AES verwendet wird, wie wir später sehen werden. Dabei wird ein Element des Körpers als Polynom vom Grad 7 dargestellt:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0$$

Die Koeffizienten $b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0$ sind Elemente des zweielementigen Körpers mit den Elementen 0 und 1.

Die Additionen im Körper $GF(2^8)$ entsprechen den Additionen der Polynome. Seien $a(x), b(x) \in GF(2^8)$ Polynome mit den Koeffizienten a_i, b_i, für $0 \leq i \leq 7$. Dann ist das Ergebnis ein Polynom $c(x) \in GF(2^8)$ mit den Koeffizienten c_i für $0 \leq i \leq 7$, wobei für die Koeffizienten gilt $c_i = a_i + b_i \mod 2$. Die Addition wird modulo zwei gerechnet, weil das Ergebnis c_i ein Koeffizient des Polynoms $c(x)$ ist und daher wie oben beschrieben c_i ein Element des Körpers mit zwei Elementen ist. Da die Addition modulo zwei der XOR-Operation entspricht, können wir c_i alternativ definieren als $c_i = a_i \oplus b_i$. Weil die

Addition zweier Polynome mit Grad ≤ 7 auch ein Polynom mit Grad ≤ 7 ergibt, ist die Addition abgeschlossen in $GF(2^8)$.

Die inverse Operation zur Addition mit einem Element $a(x)$ im Körper $GF(2^8)$ ist die Addition mit dem additiv inversen Element $a^{(-1)}(x)$. Das additiv inverse Element muss die Gleichung $a(x) + a^{(-1)}(x) = 0$ erfüllen. Da die Addition zweier Elemente der XOR-Verknüpfung der jeweiligen Koeffizienten entspricht, gilt $a_i \oplus a_i^{(-1)} = 0$ für alle i. Dies ist erfüllt, wenn $a^{(-1)}(x) = a(x)$. Das additive Inverse eines Elements $a(x)$ ist also das Element $a(x)$ selbst.

Multiplikationen im Körper $GF(2^8)$ entsprechen der Multiplikation der Polynome. Da die Multiplikation so nicht abgeschlossen wäre, muss man das Ergebnis modulo eines irreduziblen Polynoms vom Grad 8 rechnen. Wir bezeichnen dieses Polynom mit $m(x)$. Seien $a(x), b(x) \in GF(2^8)$ Polynome mit den Koeffizienten a_i, b_i, für $0 \leq i \leq 7$, dann sei das Ergebnis $c(x) = a(x) \cdot b(x) \mod m(x)$. Da das Ergebnis modulo $m(x)$ gerechnet wird, erhält man nur Ergebnisse mit Grad ≤ 7. Somit ist die Multiplikation abgeschlossen in $GF(2^8)$.

Die inverse Operation zur Multiplikation mit einem Element x im Körper $GF(2^8)$ ist die Multiplikation mit dem multiplikativen inversen Element $x^{(-1)}$. Das multiplikative Inverse ist ein Element, so dass $x \cdot x^{(-1)} = 1 \mod m(x)$ im Körper $GF(2^8)$. Dieses Element kann über den Erweiterten Euklidschen Algorithmus gefunden werden.

In AES gibt es einige Besonderheiten für das Rechnen im Körper $GF(2^8)$. So wurde das für die Multiplikation nötige irreduzible Polynom $m(x)$ festgelegt auf $m(x) = x^8 + x^4 + x^3 + x + 1$. Des Weiteren werden in AES Bytes als Elemente des Körpers $GF(2^8)$ interpretiert. Das sieht so aus, dass ein Byte b, bestehend aus der Bitfolge $b_7b_6b_5b_4b_3b_2b_1b_0$ als Koeffizienten des Polynoms $b(x)$ mit Grad sieben aufgefasst wird. Das Polynom ist definiert als

$$b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0$$

Umgekehrt werden Ergebnisse von Rechenoperationen in AES wieder als Bytes interpretiert.
Beispiel: Das Polynom $x^6 + x^4 + x^2 + x^1 + 1$, welches die Koeffizienten $0, 1, 0, 1, 0, 1, 1, 1$ hat, schreiben wir alternativ als Byte mit der Bitfolge 01010111 oder als Hexadezimalzahl 0x57.

Allgemein AES ist ein rundenbasiertes Verfahren, bei dem jeweils Blöcke von 128 Bit in mehreren Runden verarbeitet werden. Dazu wird ein geheimer Schlüssel benötigt, der entweder 128 Bit, 192 Bit oder 256 Bit lang sein kann. Die Anzahl der Runden ist von der Länge des Schlüssels abhängig. Für 128 Bit Schlüssellänge sind es 10 Runden, für 192 Bit 12 Runden und für 256 Bit 14 Runden. Wir werden in dieser Ausarbeitung immer von einer Schlüssellänge von 128 Bit ausgehen.

Grundsätzlich bekommt der AES-Algorithmus eine 128 Bit Eingabe x, auch Klartext genannt, sowie einen 128 Bit langen geheimen Schlüssel k und erzeugt daraus eine 128-Bit Ausgabe y, auch Chiffretext genannt. Dazu wird ein Zwischenwert z benötigt. Der Zwischenwert ist zu Beginn die Eingabe x. Dann werden eine Reihe von Operationen auf

den Zwischenwert z angewendet. Danach ist der Wert des Zwischenwerts der Chiffretext y. Dabei führt der AES-Algorithmus die in Algorithmus 1 beschriebenen Schritte durch.

Algorithmus 1 AES Verschlüsselung

1. Initialisiere den Zwischenwert z mit der Eingabe x.
2. Wende die Operation *AddRoundKey* mit dem Rundenschlüssel auf den Zwischenwert z an.
3. Für jede der Runden eins bis neun wende die Operationen *SubBytes*, *ShiftRows*, *MixColumns* und *AddRoundKey* in der Reihenfolge auf den Zwischenwert z an.
4. Für Runde zehn wende die Operationen *SubBytes*, *ShiftRows* und *AddRoundKey* in der Reihenfolge auf den Zwischenwert z an.
5. Gib den Zwischenwert z als Ausgabe y aus.

Rundenoperationen Wie wir wissen, sind der Zwischenwert z, sowie die Eingabe x und die Ausgabe y, 128 Bit, also 16 Byte, lang. Um die Arbeitsweise der einzelnen Operationen zu erklären stellen wir den Zwischenwert z als 4x4-Matrix dar. Diese Matrix hat 16 Elemente und jedes Element ist ein Byte groß. Für die 128-Bit Eingabe x, bestehend aus den 16 Bytes x_0 bis x_{15} sei die Zuweisung an die Matrix wie folgt definiert:

$$\begin{bmatrix} z_{0,0} & z_{0,1} & z_{0,2} & z_{0,3} \\ z_{1,0} & z_{1,1} & z_{1,2} & z_{1,3} \\ z_{2,0} & z_{2,1} & z_{2,2} & z_{2,3} \\ z_{3,0} & z_{3,1} & z_{3,2} & z_{3,3} \end{bmatrix} := \begin{bmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \\ x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{bmatrix}$$

Analog dazu definieren wir die 4x4 Matrix k bestehend aus den 16 Bytes des Rundenschlüssels k_0 bis k_{15}:

$$\begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix} := \begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$

In AES gibt es folgende Rundenoperationen:

- *SubBytes*
  SubBytes bewirkt, dass jedes Element der Matrix einzeln durch eine bijektive, nicht-lineare Funktion verändert wird. Für jedes Element gilt, dass der neue Wert weder von den anderen Elementen der Matrix, noch von seiner Position innerhalb

der Matrix abhängig ist, sondern nur von seinem Wert. In AES wird diese nicht-Linearität dadurch erreicht, dass man das Element durch sein multiplikatives Inverses ersetzt und weitere Additionen und Multiplikationen auf den einzelnen Bits durchführt. Eine detaillierte Beschreibung dieser Funktion findet man im „AES Proposal: Rijndael“ [DR99] im Kapitel 4.2.1.

Solch eine nicht-lineare Ersetzungsoperation wird auch S-Box (Substitution Box) genannt, da sie in Implementierungen in der Regel nicht als Rechenoperation, sondern als Tabelle auftaucht. Wie wir später sehen werden, wird das aus Effizienzgründen auch in AES so gemacht. Wir schreiben die Anwendung der S-Box formal als
$z_{i,j} := S[z_{i,j}]$

- *ShiftRows*
  ShiftRows verschiebt die Bytes zyklisch innerhalb einer Zeile der Matrix. Dabei wird die erste Zeile nicht verschoben, die Zweite um eine Stelle, die Dritte um zwei Stellen und die Vierte um drei Stellen, siehe Abbildung 2.1. ShiftRows hat in AES die Aufgabe Abhängigkeiten zwischen verschiedenen Spalten der Matrix herzustellen.

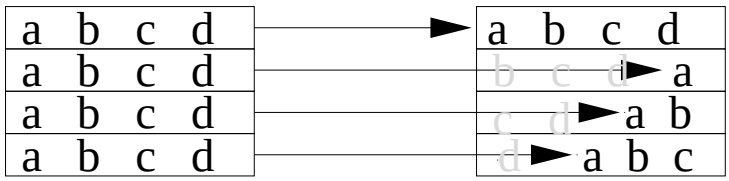

Abbildung 2.1: *ShiftRows* verschiebt die Elemente innerhalb der Zeilen der Matrix

Die Operation *ShiftRows* ist definiert durch

$$\begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} := \begin{bmatrix} z_{0,j} \\ z_{1,j-1} \\ z_{2,j-2} \\ z_{3,j-3} \end{bmatrix}$$

wobei die Indizes jeweils modulo 4 zu rechnen sind, d.h. $z_{1,-1} = z_{1,3} = z_{1,7}$.

- *MixColumns*
  MixColumns multipliziert jede Spalte der Matrix mit einer fest definierten, invertierbaren Matrix im Körper $GF(2^8)$. Dieses bewirkt einerseits, dass die einzelnen Bytes verändert werden und andererseits, dass der neue Wert eines Bytes von allen anderen Bytes der Spalte abhängig ist. *MixColumns* hat in AES die Aufgabe Abhängigkeiten zwischen den Elementen innerhalb einer Spalte der Matrix zu erstellen. Die Operation *MixColumns* ist definiert durch

$$\begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} := \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix}$$

für $0 \leq j \leq 3$.

- *AddRoundKey*
  Die Operation *AddRoundKey* führt für jedes Matrixelement $z_{i,j}$ eine Addition von $z_{i,j}$ mit dem Element $k_{i,j}$ der Schlüsselmatrix im Körper $GF(2^8)$ durch. Aufgrund der Eigenschaften der Addition in diesem Körper entspricht diese Operation der bitweisen XOR-Verknüpfung zwischen Eingabe und Schlüssel. *AddRoundKey* hat in AES die Aufgabe Abhängigkeiten zwischen zwischen den Elementen der Matrix und dem Schlüssel zu erzeugen. Die Operation *AddRoundKey* ist definiert durch

  $$z_{i,j} := z_{i,j} \oplus k_{i,j}$$

Erweiterung des Schlüssels Wie wir sehen benötigt die Operation *AddRoundKey* einen Rundenschlüssel. Der Rundenschlüssel ist eine Erweiterung des normalen Schlüssels. Für jede *AddRoundKey*-Operation benötigen wir jeweils einen Rundenschlüssel. Da der AES-Algorithmus elf *AddRoundKey*-Operationen durchführt, benötigen wir elf Rundenschlüssel, die insgesamt $11 \cdot 128 = 1408$ Bit lang sind. Die Rundenschlüssel werden durch den Erweiterungsalgorithmus aus dem Schlüssel bestimmt. Der erste Rundenschlüssel ist identisch mit dem normalen Schlüssel. Alle anderen Rundenschlüssel werden aus diesem berechnet.

Der Erweiterungsalgorithmus KeyExpansion wird in Algorithmus 2 beschrieben. Er bekommt als Eingabe den Schlüssel k. Die Operation *RotWord* führt eine zyklische Rotation der Bytes eines Wortes durch. Das Wort (a, b, c, d) wird also zu (b, c, d, a). Die Operation *SubWord* wendet die S-Box auf die einzelnen Bytes eines Wortes an. Das Wort (a, b, c, d) wird also zu $(S[a], S[b], S[c], S[d])$.

Beispiel: Wenden wir den Erweiterungsalgorithmus auf den Schlüssel k mit den Bytes $k_0 \dots k_{15}$ an, so erhalten wir folgenden ersten Rundenschlüssel $k^{(1)}$:

$$k^{(1)} := \begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$

Des Weiteren erhalten wir folgenden zweiten Rundenschlüssel $k^{(2)}$:

$$k^{(2)} := \begin{bmatrix} S[k_{13}] \oplus 01 \oplus k_0 & S[k_{13}] \oplus 01 \oplus k_0 \oplus k_4 & S[k_{13}] \oplus 01 \oplus k_0 \oplus k_4 \oplus k_8 & S[k_{13}] \oplus 01 \oplus k_0 \oplus k_4 \oplus k_8 \oplus k_{12} \\ S[k_{14}] \oplus k_1 & S[k_{14}] \oplus k_1 \oplus k_5 & S[k_{14}] \oplus k_1 \oplus k_5 \oplus k_9 & S[k_{14}] \oplus k_1 \oplus k_5 \oplus k_9 \oplus k_{13} \\ S[k_{15}] \oplus k_2 & S[k_{15}] \oplus k_2 \oplus k_6 & S[k_{15}] \oplus k_2 \oplus k_6 \oplus k_{10} & S[k_{15}] \oplus k_2 \oplus k_6 \oplus k_{10} \oplus k_{14} \\ S[k_{12}] \oplus k_3 & S[k_{12}] \oplus k_3 \oplus k_7 & S[k_{12}] \oplus k_3 \oplus k_7 \oplus k_{11} & S[k_{12}] \oplus k_3 \oplus k_7 \oplus k_{11} \oplus k_{15} \end{bmatrix}$$

Algorithmus 2 KeyExpansion(k)

RCon[1] := 01000000
RCon[2] := 02000000
RCon[3] := 04000000
RCon[4] := 08000000
RCon[5] := 10000000
RCon[6] := 20000000
RCon[7] := 40000000
RCon[8] := 80000000
RCon[9] := 1B000000
RCon[10] := 36000000
für jedes $0 \leq i \leq 3$:
  $w[i] := (k_{4 \cdot i}, k_{4 \cdot i+1}, k_{4 \cdot i+2}, k_{4 \cdot i+3})$
für jedes $4 \leq i \leq 43$:
  temp := $w[i-1]$
  wenn $i \equiv 0 \mod 4$:
    temp := $SubWord(RotWord(\text{temp})) \oplus \text{RCon}[i/4]$
  $w[i] := w[i-4] \oplus$ temp
für jedes $0 \leq i \leq 10$
  RundenSchlüssel$[i+1] := (w[i \cdot 4], w[i \cdot 4+1], w[i \cdot 4+2], w[i \cdot 4+3])$

Entschlüsselung Für die Entschlüsselung müssen wir den AES-Algorithmus rückwärts anwenden und die Rundenoperationen durch ihre Umkehrfunktion ersetzen. Der AES-Algorithmus zur Entschlüsselung ist in Algorithmus 3 definiert.

Wie wir sehen, wurden alle Operationen außer *AddRoundKey* durch ihr Inverses ersetzt. Der Grund dafür ist, dass *AddRoundKey* sein eigenes Inverses ist, da die Addition mit einem Element im Körper $GF(2^8)$ selbstinvers ist. Für die anderen Operationen definieren wir die Umkehroperation wie folgt:

- *InvSubBytes*
  Wir haben *SubBytes* als bijektive Funktion definiert. Daher gibt es auch eine Umkehrfunktion $S^{(-1)}$, so dass $S^{(-1)}[S[x]] = x$. Wir schreiben die Anwendung der S-Box formal als
  $z_{i,j} := S^{(-1)}[z_{i,j}]$

- *InvShiftRows*
  ShiftRows verschiebt die Bytes innerhalb einer Zeile der Matrix. Die Umkehroperation *InvShiftRows* muss daher nur die Verschiebung rückgängig machen, indem es die Bytes in die andere Richtung verschiebt. Die Operation *InvShiftRows* ist definiert durch

Algorithmus 3 AES Entschlüsselung

1. Initialisiere den Zwischenwert z mit der Eingabe y
2. Wende die Operation *AddRoundKey* auf den Zwischenwert z an
3. Für Runde eins wende die Operationen *AddRoundKey*, *InvShiftRows* und *InvSubBytes* in der Reihenfolge auf den Zwischenwert z an
4. Für jede der Runden zwei bis zehn wende die Operationen *AddRoundKey*, *InvMixColumns*, *InvShiftRows* und *InvSubBytes* in der Reihenfolge auf den Zwischenwert z an
5. Gib den Zwischenwert z als Ausgabe x aus

$$\begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} := \begin{bmatrix} z_{0,j} \\ z_{1,j+1} \\ z_{2,j+2} \\ z_{3,j+3} \end{bmatrix}$$

wobei die Indizes jeweils modulo 4 zu rechnen sind, d.h. $z_{1,-1} = z_{1,3} = z_{1,7}$.

- *InvMixColumns*
  Da *MixColumns* Multiplikationen mit einer fest definierten Matrix im Körper $GF(2^8)$ durchführt, müssen wir alle Werte der Matrix durch ihr multiplikatives Inverses ersetzen. Daher ist die Operation *InvMixColumns* definiert durch

$$\begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} := \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix}$$

  für $0 \leq j \leq 3$.

Effiziente Implementierungen Wir betrachten jetzt die Effizienz der einzelnen Rundenoperationen:

- *SubBytes* wendet, wie wir gesehen haben, die S-Box auf jedes Byte einzeln an. Dabei wird für jedes Byte das multiplikative Inverse im Körper $GF(2^8)$ berechnet und es werden mehrere Additionen und Multiplikationen durchgeführt. Pro Runde werden 16 mal diese Operationen durchgeführt.
- *ShiftRows* verschiebt lediglich Bytes in der Matrix. Da alle Bytes, außer die vier Bytes in der ersten Zeile der Matrix, verschoben werden müssen, müssen 12 Elemente verschoben werden.

- *MixColumns* multipliziert jeweils eine Spalte der Matrix mit einer fest definierten Matrix. Da die Matrix vier Spalten hat, werden pro Runde vier Matrixmultiplikationen durchgeführt.

- *AddRoundKey* führt eine Bitweise XOR-Verknüpfung durch.

Die Operationen *ShiftRows* und *AddRoundKey* benötigen sehr wenig Rechenaufwand, da *ShiftRows* nur Verschiebungen im Speicher durchführt und *AddRoundKey* für jedes der 128 Bit eine einfache logische Operation durchführt. Aufwendiger ist da schon *MixColumns*, das in jeder Runde vier Matrixmultiplikationen durchführt. Sehr Rechenaufwendig ist jedoch die *SubBytes*-Operation, da in jeder Runde 16 mal ein multiplikatives Inverses bestimmt werden muss. Daher können wir die Effizienz von AES verbessern, indem wir alle möglichen Werte für *SubBytes* im Voraus berechnen und in einer Tabelle speichern. Dann kann man in der Implementierung statt für jedes Byte den S-Box-Wert zu berechnen einfach den zur Eingabe zugehörigen Ausgabewert aus der Tabelle lesen.

Wie wir wissen berechnet die S-Box zu einer ein Byte großen Eingabe eine ein Byte große Ausgabe. Für ein Byte gibt es 256 mögliche Werte, die es annehmen kann. Daher müssen wir in der S-Box-Tabelle 256 Werte speichern, die jeweils ein Byte groß sind. Daher ist die Tabelle 256 Byte groß.

Weiter betrachten wir die Operation *ShiftRows*. Diese Operation verschiebt Elemente innerhalb der Matrix unabhängig von deren Wert. Im Gegensatz dazu ersetzt *SubBytes* jedes Element der Matrix unabhängig von dessen Position. Aus diesem Grund können wir die Reihenfolge der Operationen *SubBytes* und *ShiftRows* tauschen, ohne das Ergebnis zu verändern. Wenden wir erst *ShiftRows* auf den Zwischenwert z an und dann *SubBytes*, dann können wir die Hintereinanderausführung der beiden Operationen mit den formalen Definitionen der einzelnen Operationen wie folgt definieren:

$$\begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} := \begin{bmatrix} S[z_{0,j}] \\ S[z_{1,j-1}] \\ S[z_{2,j-2}] \\ S[z_{3,j-3}] \end{bmatrix}$$

Fügen wir jetzt noch die *MixColumns*- und *AddRoundKey*-Operationen hinzu, erhalten wir folgende Definition einer Runde:

$$\begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} := \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S[z_{0,j}] \\ S[z_{1,j-1}] \\ S[z_{2,j-2}] \\ S[z_{3,j-3}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Durch Umformen der Matrix-Multiplikation erhalten wir

$$\begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} = S[z_{0,j}] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus S[z_{1,j-1}] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus S[z_{2,j-2}] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus S[z_{3,j-3}] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Neben Zugriffen auf S-Box-Tabellen besteht eine Runde jetzt aus XOR-Operationen und Multiplikationen. Wie wir in unserer umgeformten Rundendefinition sehen, werden die S-Box-Werte jeweils mit einem festen Spaltenvektor multipliziert. Daher können wir weitere Rechenzeit sparen, indem wir nicht die S-Box-Werte in Tabellen speichern, sondern die S-Box-Werte multipliziert mit dem Spaltenvektor. Da das Ergebnis der Multiplikation des S-Box-Wertes mit dem Spaltenvektor auch ein Spaltenvektor ist und der Spaltenvektor aus 4 Bytes besteht, benötigen wir dann eine Tabelle mit 256 Elementen, die jeweils vier Byte groß sind. Dadurch erhöht sich der Speicherbedarf auf $256 \cdot 4$ Byte = 1 KByte und da jeder der vier S-Box-Werte mit einem anderen Spaltenvektor multipliziert wird, benötigen wir vier verschiedene Tabellen. Für die letzte Runde benötigen wir, wie wir sehen werden, eine weitere Tabelle. Daher ist der Gesamtspeicherbedarf für die fünf Tabellen 5 KByte. Aus der umgeformten Rundendefinition ergibt sich folgende Definition der Tabellen:

$$T_0[a] = \begin{bmatrix} S[a] \cdot 02 \\ S[a] \\ S[a] \\ S[a] \cdot 03 \end{bmatrix} T_1[a] = \begin{bmatrix} S[a] \cdot 03 \\ S[a] \cdot 02 \\ S[a] \\ S[a] \end{bmatrix} T_2[a] = \begin{bmatrix} S[a] \\ S[a] \cdot 03 \\ S[a] \cdot 02 \\ S[a] \end{bmatrix} T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \cdot 03 \\ S[a] \cdot 02 \end{bmatrix}$$

Mit den Tabellen sieht unsere Rundendefinition nun so aus:

$$\begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} := T_0[z_{0,j}] \oplus T_1[z_{1,j}] \oplus T_2[z_{2,j}] \oplus T_3[z_{3,j}] \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Die letzte Runde unterscheidet sich von den anderen Runden dadurch, dass die *MixColumns*-Operation nicht durchgeführt wird. Dadurch benötigen wir für die letzte Runde nur eine Tabelle:

$$T_4[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \\ S[a] \end{bmatrix}$$

Mit dieser Tabelle sieht die letzte Runde nun so aus:

$$\begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} := T_4[z_{0,j}] \oplus T_4[z_{1,j}] \oplus T_4[z_{2,j}] \oplus T_4[z_{3,j}] \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

2.2 Cache

Der Cache ist ein Zwischenspeicher, dessen Aufgabe es ist, Daten aus dem Hauptspeicher, auf die häufig zugegriffen wird, zwischenzuspeichern. Der Cache ist deutlich schneller als der Hauptspeicher, aber auch teurer. Daher ist der Cache in der Regel sehr viel kleiner als der Hauptspeicher.

Ein Cache ist unterteilt in Lines und Sets. Ein Cache besteht aus S Sets, jedes Set besteht aus W Lines. Jede Cache-Line ist B Bytes groß. Die Gesamtgröße des Caches ist also $S \cdot W \cdot B$ Bytes.

In welches Set Daten zwischengespeichert werden, ist von der Adresse abhängig, auf die zugegriffen wurde. Dazu wird die Adresse A, bestehend aus a Bits, aufgeteilt in t Tag-Bits, s Set-Bits und b Offset-Bits, siehe Abbildung 2.2. Die niederwertigsten $b = \log_2(B)$ Bits (Offset-Bits) dienen nur zur Adressierung eines Bytes innerhalb einer Cacheline. Die nächsten $s = \log_2(S)$ Bits (Set-Bits) bestimmen, in welches Cache-Set die Daten gespeichert werden. Die restlichen $t = a - (s + b)$ Bits (Tag-Bits) werden als Meta-Daten zusammen mit den Daten im Cache gespeichert [BO03].

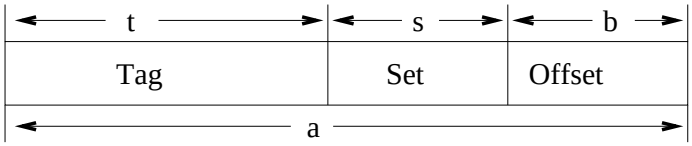

Abbildung 2.2: Aufteilung der Adresse

Gibt es innerhalb eines Cache-Sets mehrere Cache-Lines, so können mehrere Daten von verschiedenen Adressen, die in das selbe Cache-Set gespeichert werden müssen, also Daten mit verschiedenen Tag-Bits, aber mit gleichen Set-Bits, gleichzeitig zwischengespeichert sein. Innerhalb eines Cache-Sets ist es dabei egal, in welche Cache-Line die Daten geschrieben werden. Einen Cache, der n Cache-Lines pro Set hat, nennt man n-fach assoziativ. Einen 1-fach assoziativen Cache nennt man direct-mapped-Cache. Einen Cache, der nur aus einem Cache-Set besteht, nennt man voll-assoziativ.

Bei einem Speicherzugriff wird zuerst geprüft, ob die gewünschten Daten bereits im Cache sind. Dazu wird die Adresse aufgeteilt in Tag-Bits *tb*, Set-Bits *sb* und Offset-Bits *ob*. Da die Daten nur im Set mit der Adresse *sb* sein können, braucht der Cache sie auch nur dort zu suchen. Dazu vergleicht er *tb* mit den Tag-Bits aller Cache-Lines im Set. Hat er eine Cache-Line gefunden, bei der das Tag-Bit gleich *tb* ist, so liest er das Byte mit Adresse *ob* aus der Cache-Line. Ist keine Cache-Line mit dem Tag-Bit *tb* vorhanden, so sind die Daten nicht im Cache. Dann lädt der Cache die 2^b Bytes mit den Tag-Bits *tb* und den Set-Bits *sb* in eine Cache-Line im Cache-Set mit der Adresse *sb* [BO03].

Beispiel: Ein PC hat einen direct-mapped-Cache (eine Line pro Set) mit 16 Cache-Sets und einer Größe von 256 Byte pro Cache-Line. Es wird ein Programm ausgeführt, das auf die Adresse $0x12345678$ zugreift. Um Festzustellen, wohin die Daten im Cache geladen werden, müssen wir die Adresse in Tag-Bits, Set-Bits und Offset-Bits unterteilen. Die niederwertigsten $b = \log_2(256) = 8$ Bits der Adresse sind die Offset-Bits, also $0x78$. Die mittleren $s = \log_2(16) = 4$ Bits sind die Set-Bits, also $0x6$. Die Restlichen $t = 32 - (4 + 5) = 23$ Bits sind die Tag-Bits, also $0x12345$. Beim Zugriff auf die Adresse werden daher die Daten mit den Adressen von $0x12345600$ bis $0x123456ff$ in die Cache-Line des Cache-Sets mit der Adresse $0x6$ geladen. Zusammen mit den Daten werden als Meta-Information die Tag-Bits $0x12345$ der Adresse in der Cache-Line gespeichert. Durch die Tag-Bits weiß der Cache für jedes Byte im Cache, an welcher Adresse es im Hauptspeicher steht, da er neben Tag-Bits für jedes Byte auch die Set-Bits und Offset-Bits kennt.

3 Cache-Angriffe

In diesem Kapitel werden wir sehen, wie ein Angreifer einen Cache, den sich alle Prozesse auf einem Rechner teilen, dazu nutzen kann, um Informationen über andere Prozesse zu bekommen. Im ersten Abschnitt werden wir das allgemeine Funktionsprinzip des Angriffs kennenlernen und im zweiten Abschnitt werden wir sehen, wie man dieses Verfahren für einen Seitenangriff auf die sehr verbreiteten effizienten AES-Implementierungen mit großen Tabellen benutzen kann, um den geheimen Schlüssel zu bestimmen. Die in diesem Kapitel beschriebenen Cache-Angriffe sind Ergebnisse der Veröffentlichungen [OST06], [Ber05] und [Per05]. Wir werden hier grundsätzlich von einem direct-mapped Cache ausgehen. Cache-Angriffe auf assoziative Caches werden in [OST06] beschrieben.

3.1 Grundsätzliche Funktionsweise

Moderne Betriebssysteme ermöglichen es, verschiedene Prozesse von verschiedenen Benutzern auf einem Rechner laufen zu lassen. Zudem unterstützen solche Betriebssysteme die Vergabe von Rechten, so dass Daten im Speicher oder im Dateisystem gegen Zugriffe von anderen Benutzern geschützt sind. Dies ist nötig, da sich alle Prozesse, die auf einem Rechner laufen, verschiedene Komponenten, wie zum Beispiel Speicher, Cache, CPU und Festplatte teilen. Wir werden nun sehen, wie es unabhängig von den Rechten eines Prozesses möglich ist, mit Hilfe des Caches Informationen über die Daten anderer Prozesse zu bekommen.

Wir gehen grundsätzlich davon aus, dass ein Angreifer sowie Prozesse des Angreifers Kenntnisse über den Aufbau des Caches haben. Er kennt insbesondere die Anzahl der Sets S, die Anzahl der Lines pro Set W und die Größe einer Line B. Mit diesen Informationen kann er auch die Anzahl der Adressbits t, s und b bestimmen und ist so in der Lage eine Adresse in Tag-, Set- und Offset-Bits zu zerlegen. Daher kann er auch zu jeder Adresse bestimmen, in welche Cache-Line und an welche Stelle in der Cache-Line das Byte mit jener Adresse zwischengespeichert wird. Zudem weiß er, welche der angrenzenden Daten zusätzlich in die Cache-Line geladen werden. Des Weiteren gehen wir davon aus, dass Prozesse des Angreifers keine besonderen Rechte haben und auch nicht auf den Speicher anderer Prozesse zugreifen dürfen.

Zunächst zeigen wir, wie ein Prozess alle Daten anderer Prozesse aus einem Cache verdrängen kann, so dass nur seine eigenen Daten zwischengespeichert sind. Dazu führt er den Algorithmus 4 aus, den wir *cache_leeren* nennen. Dieser Algorithmus reserviert einen Speicherbereich und greift auf jedes B-te Byte zu. Da die Größe einer Cache-Line B Bytes ist, bewirkt die Addition von B auf eine Adresse die Inkrementierung des Wertes der Set-Bits um eins. Somit wird auf S verschiedene Adressen mit aufeinanderfolgenden

Algorithmus 4 cache_leeren

1. Reserviere einen $S \cdot B$ Byte großen zusammenhängenden Speicherbereich und weise die Adresse des ersten Bytes an x zu.
2. Lese nacheinander die Bytes mit den Adressen $x + k \cdot B$ für $0 \leq k \leq S - 1$ und verwerfe die gelesenen Daten.

Set-Bits-Werten zugegriffen. Dadurch werden bei jedem Zugriff Daten in ein anderes Cache-Set geladen.

Des Weiteren zeigen wir, wie ein Prozess feststellen kann, ob seine Daten im Cache zwischengespeichert sind oder nicht. Nehmen wir an, ein Prozess hat einen X Bytes großen zusammenhängenden Speicherbereich, der bei Adresse x beginnt. Er führt den Algorithmus 5 aus, den wir *cache_test* nennen: Wie auch bei *cache_leeren* wird bei *cache_test* auf jedes B-te Byte zugegriffen, so dass jedes Byte, auf das zugegriffen wird, einem anderen Cache-Set zugeordnet ist. Anhand der Zeitdifferenzen kann man nun bestimmen, ob die Daten im Cache waren oder nicht. Wenn die Daten aus dem Cache geladen werden, geht das deutlich schneller, als wenn die Daten aus dem Hauptspeicher geladen werden.

Algorithmus 5 cache_test

Für alle Adressen $x + k \cdot B$ mit $0 \leq k \leq S - 1$:

1. Messe die Zeit.
2. Lese das Byte an der Adresse.
3. Messe erneut die Zeit und berechne und speichere die Zeitdifferenz.

Mit diesen beiden Verfahren ist es nun relativ einfach Informationen über Speicherzugriffe anderer Prozesse zu bekommen. Nehmen wir an, dass auf einem Rechner ein Prozess a läuft. Ein Angreifer möchte nun Informationen über die Speicherzugriffe von Prozess a bekommen. Dazu startet er einen Prozess b, der zuerst den Algorithmus *cache_leeren* ausführt, dann einige Millisekunden wartet und dann den Algorithmus *cache_test* ausführt.

Zuerst sorgt Prozess b dafür, dass alle Daten anderer Prozesse aus dem Cache verdrängt werden. Während er dann einige Millisekunden wartet, wird der Prozessor in der Zwischenzeit einen anderen Prozess ausführen. In diesem Fall ist das der Prozess a. Da der Cache nun vollständig mit den Daten von Prozess b gefüllt ist, wird jeder Speicherzugriff von Prozess a dafür sorgen, dass Daten von Prozess b verdrängt werden. Nach Ablauf der Wartezeit wird Prozess a wieder unterbrochen und Prozess b weiterlaufen. Prozess b führt den Algorithmus *cache_test* durch und weiß nun, welche seiner Daten von Prozess a aus dem Cache verdrängt wurden. Dadurch kennt er auch die Set-Bits der Daten von Prozess a, die seine Daten verdrängt haben, da sie identisch mit denen der

verdrängten Daten sein müssen.

Wenn der Angreifer den Bytecode des Prozesses a kennt, kann er möglicherweise weitere Informationen bekommen. Beim Kompilieren und Linken eines Programms, werden allen statischen Speicherreferenzen Adressen zugewiesen. Daher kann der Angreifer für alle nicht-dynamischen Speicherreservierungen die Größen und Adressen dieser Speicherbereiche aus dem Bytecode lesen. Haben alle diese Adressen die selben Tag-Bits, so kennt der Angreifer auch die Tag-Bits der Adresse. Die vollständige Adresse lässt sich jedoch auf diesem Weg nicht rekonstruieren. Die noch unbekannten Offset-Bits der Adresse lassen sich ohne weitere Informationen nicht bestimmen.

Auf den ersten Blick scheint diese Art von Angriff harmlos zu sein, da man lediglich Informationen über Speicheradressen bekommt, aber keine Informationen über die Daten, die an dieser Speicheradresse stehen. Wie wir später sehen werden, sieht das anders aus, wenn es in einem Programm von Daten abhängig ist, auf welche Speicheradresse zugegriffen wird. Konkret werden wir sehen, dass in AES Speicherzugriffe abhängig vom geheimen Schlüssel sind und es so für einen Angreifer möglich ist, mit dieser Technik Informationen über diesen Schlüssel zu bekommen.

3.2 Angriffe auf AES

Wir werden jetzt einen Angriff auf AES zeigen, bei dem unter bekanntem Klartext, jedoch ohne Kenntnis des Chiffretextes, durch einen Cache-Angriff Teile des geheimen Schlüssels bestimmt werden können. Dass dieser Angriff funktioniert, beruht auf der Tatsache, dass die Tabellenzugriffe in AES direkt abhängig vom Schlüssel sind. Des weiteren werden wir sehen, wie durch mehrere Cache-Angriffe der komplette geheime Schlüssel bestimmt werden kann. Wir werden diese Angriffe unter idealen Bedingungen zeigen, die in der Praxis praktisch nicht vorkommen. Wie diese Angriffe jedoch auch unter realen Bedingungen durchgeführt werden können, werden wir danach kurz erläutern.

Wir gehen im Folgenden immer von den besonders effizienten AES-Implementierungen mit den fünf großen jeweils ein KByte großen Tabellen aus. Da die Gesamtgröße der Tabellen nur 5 KByte beträgt, aber die Caches moderner Rechner um ein vielfaches größer sind, gehen wir davon aus, dass der Cache größer ist, als die Tabellen und somit Tabellenelemente sich nicht gegenseitig verdrängen können. Des weiteren gehen wir davon aus, dass wir der AES-Implementierung einen Klartext geben können und diese den bekannten Klartext mit dem unbekannten Schlüssel verschlüsselt. In der Praxis tritt so eine Situation zum Beispiel auf, wenn ein Betriebssystem einem Benutzer erlaubt Daten in ein verschlüsseltes Dateisystem zu schreiben oder Daten über eine verschlüsselte VPN-Verbindung zu senden, wobei die Daten jeweils vom Betriebssystem mit einem dem Benutzer nicht bekannten Schlüssel verschlüsselt werden. Zudem gehen wir davon aus, dass der Bytecode der AES-Implementierung bekannt ist und daher auch die virtuellen Adressen der AES-Tabellen.

Wir bezeichnen die Anzahl der Tabellenelemente, die in eine Cache-Line passen mit δ. Da eine Cache-Line B Byte groß ist und ein Tabellenelement 4 Byte groß ist, ist $\delta = B/4$.

Funktionsweise Prinzipiell führen wir wieder den oben beschriebenen Cache-Angriff durch. Prozess a sei jetzt eine AES-Implementierung und Prozess b der Prozess des Angreifers. Mit dem Algorithmus *cache_leeren* füllt Prozess b wieder den Cache mit seinen Daten. Dann jedoch sorgt er dafür, dass die AES-Implementierung den Klartext p verschlüsselt. Prozess a startet mit der Verschlüsselung und wird nach einiger Zeit unterbrochen. Dann führt Prozess b wieder den Algorithmus *cache_test* durch und bekommt so Informationen über die Speicherzugriffe von Prozess a.

Nun haben wir die Set-Bits der Adressen, auf die Prozess a zugegriffen hat. Diese Set-Bits sind per Definition die Adressen der Cache-Sets, in die Daten von Prozess a zwischengespeichert wurden. Da wir aus dem Bytecode der AES-Implementierung die virtuellen Adressen der AES-Tabellen kennen, können wir nun bestimmen, welche der Adressen Tabellenzugriffe sind und welche nicht. Wir bezeichnen die Menge der Adressen, die Tabellenzugriffe sind mit T.

Für jede Cache-Set-Adresse $t \in T$ wissen wir, dass δ aufeinanderfolgende Tabellenelemente in dieses Cache-Set geladen wurden. Wir wissen sogar, welche Tabellenelemente in dieses Cache-Set geladen wurden, jedoch wissen wir nicht, auf welches dieser δ Tabellenelemente Prozess a zugegriffen hat.

Cache-Angriffe auf die erste Runde Nehmen wir an, wir führen einen Cache-Angriff auf einen AES-Prozess so durch, dass der AES-Prozess nach durchführen der ersten AES-Runde unterbrochen wird. Folglich besteht unsere Menge der Cache-Set-Adressen T dann nur aus den Set-Bits, die den Tabellenzugriffen in der ersten Runde entsprechen.

In der ersten Runde entspricht der Index jedes Tabellenzugriffs $n = p_i \oplus k_i$ mit $0 \leq i \leq 15$, also der Bitweisen XOR-Verknüpfung je eines Bytes des Klartextes und eines Bytes des Schlüssels. Für jedes $t \in T$ können wir δ aufeinanderfolgende Indizes der Tabellenelemente bestimmen, von denen wir wissen, dass auf eines von ihnen zugegriffen wurde. Von diesen Elementen betrachten wir nun ihren Index in der Tabelle. Der Index aller δ Elemente unterscheidet sich nur in den niederwertigsten $\log_2(\delta)$ Bits. Die anderen $8 - \log_2(\delta)$ hochwertigsten Bits sind bei allen Indizes identisch und bekannt. Dies sind aber die $8 - \log_2(\delta)$ hochwertigsten Bits von n. Die Gleichung $n = p_i \oplus k_i$ können wir umstellen nach $k_i = n \oplus p_i$. Wenden wir diese Gleichung auf die hochwertigsten Bits von n und p_i an, bekommen wir die hochwertigsten Bits des Schlüssels k_i. Da in der ersten Runde für jedes der 16 Bytes des Schlüssels ein Tabellenzugriff stattfindet, können wir im Idealfall mit diesem Verfahren $16 \cdot (8 - \log_2(\delta))$ Bits des Schlüssels bestimmen.

Führen wir Cache-Angriffe nur auf die erste Runde durch, können wir immer nur die $8 - \log_2(\delta)$ hochwertigsten Bits jedes Bytes des Schlüssels bestimmen. Daher bringt hier eine Wiederholung desselben Angriffs keine neuen Bits des Schlüssels, auch nicht, wenn wir den Klartext p verändern.

Cache-Angriffe auf die ersten beiden Runden Wollen wir mehr Bits des Schlüssels bestimmen, müssen wir den Angriff auf mehrere Runden ausweiten. Wir werden jetzt zeigen, wie sich mit den Schlüsselbits aus der ersten Runde in der zweiten Runde der Schlüssel bestimmen lässt.

Aus Kapitel 2.1 können wir folgende Definition des Zwischenwerts z nach dem ersten *AddRoundKey* und vor der ersten Runde ableiten:

$$z_{i,j} = x_{i+4\cdot j} \oplus k_{i+4\cdot j}$$

Wir setzen dies in die effiziente Rundendefinition aus Kapitel 2.1 ein und erhalten so:

$$\begin{bmatrix} z_{0,j} \\ z_{1,j} \\ z_{2,j} \\ z_{3,j} \end{bmatrix} := T_0[x_{4j} \oplus k^{(1)}_{4j}] \oplus T_1[x_{1+4j} \oplus k^{(1)}_{1+4j}] \oplus T_2[x_{2+4j} \oplus k^{(1)}_{2+4j}] \oplus T_3[x_{3+4j} \oplus k^{(1)}_{3+4j}] \oplus \begin{bmatrix} k^{(2)}_{4j} \\ k^{(2)}_{1+4j} \\ k^{(2)}_{2+4j} \\ k^{(2)}_{3+4j} \end{bmatrix}$$

Hierbei sei $k^{(i)}$ der i-te Rundenschlüssel. Die Rundenschlüssel können wir mit Algorithmus 2 aus dem Schlüssel k berechnen.

Ersetzen wir die Tabellen durch ihre Definition, so können wir folgende Gleichungen für folgende Elemente des Zwischenwerts nach der ersten Runde ableiten. Wir ersetzen dabei die Rundenschlüssel durch ihre Definition in Abhängigkeit des Schlüssels k, die auch im Beispiel im Abschnitt „Erweiterung des Schlüssels“ des Kapitels 2.1 definiert wurden:

$$\begin{aligned}
z_{2,0} &= S[x_0 \oplus k_0] \oplus S[x_5 \oplus k_5] \oplus 2 \cdot S[x_{10} \oplus k_{10}] \oplus 3 \cdot S[x_{15} \oplus k_{15}] \oplus S[k_{15}] \oplus k_2 \\
z_{1,1} &= S[x_4 \oplus k_4] \oplus 2 \cdot S[x_9 \oplus k_9] \oplus 3 \cdot S[x_{14} \oplus k_{14}] \oplus S[x_3 \oplus k_3] \oplus S[k_{14}] \oplus k_1 \oplus k_5 \\
z_{0,2} &= 2 \cdot S[x_8 \oplus k_8] \oplus 3 \cdot S[x_{13} \oplus k_{13}] \oplus S[x_2 \oplus k_2] \oplus S[x_7 \oplus k_7] \oplus S[k_{13}] \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1 \\
z_{3,3} &= 3 \cdot S[x_{12} \oplus k_{12}] \oplus S[x_1 \oplus k_1] \oplus S[x_6 \oplus k_6] \oplus 2 \cdot S[x_{11} \oplus k_{11}] \oplus S[k_{12}] \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_{11}
\end{aligned}$$

Zunächst betrachten wir die Gleichung für $z_{2,0}$. Wir kennen von den Schlüsselbytes k_0, k_2, k_5, k_{10} und k_{15} jeweils die hochwertigsten $8 - \log_2(\delta)$ Bits. Die noch unbekannten niederwertigsten Bits von k_2 wirken sich nur auf die niederwertigsten Bits von $z_{2,0}$ aus, da k_2 damit nur XOR-verknüpft ist.

In Runde zwei findet der Tabellenzugriff $T_2[z_{2,0}]$ statt. Die niederwertigsten Bits von $z_{2,0}$ wirken sich nicht auf das Cache-Set aus, auf das bei dem Tabellenzugriff zugegriffen wird. Daher wirken sich auch die niederwertigsten Bits von k_2 nicht darauf aus. Die einzigen unbekannten Bits, die daraus Einfluss haben, sind die niederwertigsten Bits von k_0, k_5, k_{10} und k_{15}. Für die niederwertigsten Bits dieser vier Schlüsselbytes gibt es $(2^{\log_2(\delta)})^4 = \delta^4$ mögliche Kombinationen. Wir können nun alle Kombinationen ausprobieren, indem wir sie in die Gleichung einsetzen und so für $z_{2,0}$ jeweils einen möglichen Kandidaten $z^*_{2,0}$ bestimmen. Dafür legen wir die für uns uninteressanten niederwertigsten Bits von k_2 auf einen beliebigen Wert fest. Wir prüfen mit dem Kandidaten nun, ob $T_2[z^*_{2,0}]$ einen Zugriff auf dasselbe Cache-Set verursachen würde, wie $T_2[z_{2,0}]$. Wenn nein, können wir ausschließen, dass dieser Kandidat richtig ist und verwerfen den Kandidaten. Wenn ja, merken wir uns den Kandidaten.

Auf diesem Weg werden wir nur wenige Kandidaten ausschließen können. Das liegt zum Einen daran, dass wir bei dem Angriff pro Gleichung nur einen Tabellenzugriff

betrachten, also insgesamt vier Tabellenzugriffe. Zum Anderen liegt es daran, dass wir bei dem Angriff Daten über die Cache-Zugriffe von der ersten und zweiten Runde erhalten. Da in jeder Runde auf jede Tabelle vier Zugriffe erfolgen, haben wir Daten über $4+4 = 8$ Zugriffe auf Cache-Sets, die wir der Tabelle T_2 zuordnen können. Wir können aber nicht bestimmen, welcher dieser Zugriffe nun der ist, den wir analysieren möchten. Deswegen können wir nur die Indizes ausschließen, die nicht einen Zugriff auf eins dieser acht Cache-Sets verursachen würden.

Wir können aber den Angriff auf die zweite Runde mehrfach, jeweils mit einem anderen Klartext, wiederholen und so weitere Kandidaten ausschließen. Wie viele Kandidaten wir bei jedem Angriff ausschließen können und wie viele Angriffe wir durchführen müssen, bis die Schlüsselbits eindeutig bestimmen werden können, variiert von Fall zu Fall. Wir können aber die erwartete Anzahl von benötigten Angriffen bestimmen. Sei n die Adresse eines Cache-Sets, in das Tabellenelemente der Tabelle T_l bei Zugriff zwischengespeichert werden. Laut [OST06] ist die Wahrscheinlichkeit, dass bei einem Tabellenzugriff auf T_l nicht auf das Cache-Set n zugegriffen wird, mindestens $(1 - \delta/256)^3$. Da in den ersten beiden Runden $4 + 4 - 1 = 7$ weitere Zugriffe auf Tabelle T_l erfolgen, ist die Wahrscheinlichkeit, dass in den ersten beiden Runden nicht auf das Cache-Set n zugegriffen wird $(1 - \delta/256)^{3+7}$. Laut [OST06] folgt daraus, dass wir pro Angriff einen Anteil von $(\delta/256) \cdot (1 - \delta/256)^{3+7}$ an falschen Kandidaten ausschließen können[1]. Weiterhin folgt aus ihren Annahmen, dass wir, um den richtigen Schlüssel aus den δ^4 Kandidaten zu bestimmen, ungefähr $\log_2(\delta^{-4}) / \log_2(1 - \delta/256 \cdot (1 - \delta/256)^{10})$ Angriffe benötigen. Für den laut [OST06] in der Praxis häufig auftretenden Fall $\delta = 16$ werden daher ca. 333 Angriffe benötigt.

Genauso wie mit der Gleichung für $z_{2,0}$, können wir mit den anderen drei Gleichungen jeweils vier unterschiedliche Bytes des Schlüssels bestimmen, wodurch wir mit diesen vier Gleichungen $4 \cdot 4 = 16$ unterschiedliche Bytes des Schlüssels, also den kompletten Schlüssel bestimmen können.

Cache-Angriffe unter realen Bedingungen Unter realen Bedingungen haben wir mehr als zwei Prozesse, die auf einem Rechner laufen und den Zustand des Caches beeinflussen. Wir können so nicht eindeutig feststellen, ob eine Cache-Line vom AES-Prozess oder von irgendeinem anderen Prozess verdrängt wurde. Das verfälscht die Ergebnisse. Ausgleichen können wir dies, indem wir jeden Cache-Angriff viele Male wiederholen und dann aus den Messwerten die Mittelwerte bestimmen.

Zudem ist das Timing ein Problem. Wir gehen bei den idealen Angriffen davon aus, dass wir genau eine Runde „messen“ können, d.h. dass wir bestimmen können, dass der AES-Prozess genau nach Durchführung einer (oder zwei) Runden unterbrochen wird. Es ist fraglich, ob man das in der Praxis so gut steuern kann. In den Veröffentlichungen, auf die in dieser Ausarbeitung Bezug genommen wird, wird das nicht näher erläutert.

Weitere Informationen zu Cache-Angriffen auf AES unter realen Bedingungen findet man in [OST06].

[1] Die Berechnungen von [OST06] wurden an unsere Voraussetzungen angepasst, da [OST06] davon ausgeht, dass Angriffe nur über alle 10 Runden durchgeführt werden können.

4 Simulation von Cache-Angriffen auf AES

In diesem Kapitel werden wir uns dem eigentlichen Thema dieser Studienarbeit zuwenden, nämlich der Simulation von Cache-Angriffen auf AES. Wir werden im ersten Abschnitt erläutern was genau die Simulation von Cache-Angriffen ist und welche Vorteile sie gegenüber echten Angriffen hat, werden dann im zweiten Abschnitt erläutern, nach welchen Vorgaben die Implementierung erstellt wurde und werden zum Schluss im dritten Abschnitt die Implementierung selbst beschreiben.

4.1 Idee

Die Durchführung von Cache-Angriffen ist in der Praxis sehr kompliziert. Die Idee der Simulation von Cache-Angriffen ist daher, dass man die Daten, die ein Cache-Angriff liefern würde, berechnet. Wir müssen dafür nur den Quelltext des Programms, das wir angreifen wollen, so modifizieren, dass es die Informationen, die man über einen Cache-Angriff bekommen könnte, berechnet. Das ist nicht besonders schwierig oder kompliziert, da man nur bei jedem Speicherzugriff mit der bekannten Adresse, auf die zugegriffen wurde, das Cacheverhalten zu Simulieren braucht.

Wir wollen speziell eine Simulation für Cache-Angriffe auf AES implementieren. Die Simulation soll die Daten eines Cache-Angriffs liefern, die dann als Grundlage für weitere Forschungen über die Bestimmung des Schlüssels aus diesen Daten dienen sollen.

4.2 Anforderungen

Als Grundlage für die Implementierung dient eine effiziente AES-Implementierung von Vincent Rijmen, Antoon Bosselaers und Paulo Barreto, geschrieben in C. Diese Implementierung soll um eine Cache-Simulation erweitert werden, so dass die Informationen, die man bei Cache-Angriffen bekommen würde, bestimmt werden können.

Diese Erweiterungen sollen in C++ implementiert werden, damit man die Daten kapseln kann und dadurch die Daten vom AES-Algorithmus und der Simulation des Cache-Angriffs so trennen kann, dass nicht aus Versehen die Simulation auf Daten zugreift, die nicht über den Cache-Angriff bestimmt wurden. Zudem sollen die AES-Tabellen durch andere Tabellen, die aus einer Datei geladen werden, ersetzt werden können. Die Cache-Parameter sollen frei bestimmt werden können, insbesondere die Anzahl der Sets S und die Größe einer Cacheline B. Für die Cacheangriffe sollen nur die Zugriffe auf AES-Tabellen ausgewertet werden. Dabei soll man den Zeitraum, in dem der Cacheangriff stattfindet einschränken können. Dazu gibt es zwei Varianten, entweder beschränkt man

die Simulation auf bestimmte Zugriffe oder auf bestimmte Runden. Bei der ersten Variante soll man festlegen können, dass der Cacheangriff bei Tabellenzugriff i beginnt und bei Tabellenzugriff j endet. Dann werden für den Cacheangriff nur die Tabellenzugriffe dazwischen ausgewertet. Alternativ soll man festlegen können, dass der Cacheangriff in AES-Runde i beginnt und in Runde j endet. Dann werden für den Cacheangriff nur die Tabellenzugriffe ausgewertet, die in den Runden dazwischen stattfinden.

4.3 Implementierung

Die Implementierung besteht aus folgenden Klassen:

- ConfRead
  Die Aufgabe der Klasse ConfRead ist es eine Konfigurationsdatei einzulesen und zu speichern. Des weiteren verfügt sie über Funktionen um auf einzelne Einträge der Konfigurationsoptionen zuzugreifen und deren Wert zu lesen.

- CacheSim
  Die Klasse CacheSim implementiert einen Cache-Simulator. Aufgabe dieses Cache-Simulators ist es anhand einer Folge von Speicheradressen die Zugriffe auf diese Adressen und das daraus resultierende Verhalten eines Caches zu simulieren.

- AESData
  Die Klasse AESData stellt Funktionen für den Zugriff auf Tabellen zur Verfügung und leitet die Adressen, auf die zugegriffen wurde an die Cache-Simulation weiter.

Kombination von C und C++ Da die gegebene AES-Implementierung in C geschrieben ist, die Simulation der Cache-Angriffe jedoch in C++ geschrieben ist, können wir aus der AES-Implementierung nicht direkt Methoden des C++-Teils aufrufen oder Klassen instanziieren. Dieses Problem umgehen wir, indem wir im C++-Teil neben Klassen und Methoden auch Funktionen und Variablen außerhalb von Klassen definieren. Wir übersetzen dann die in C geschriebene AES-Implementierung mit dem C-Compiler und die in C++ geschriebene Cacheangriff-Simulation mit dem C++-Compiler. Dann linken wir alles zu einer ausführbaren Datei. Wenn wir das so machen, werden wir feststellen, dass der letzte Schritt, das Linken, fehlschlägt. Da C und C++ verschiedene Namenskonventionen für Funktionen haben, kann der Linker einen Funktionsaufruf aus dem C-Code nicht einer im C++-Code definierten Funktion zuordnen. Um das zu vermeiden, sagen wir dem C++-Compiler, dass er die Funktionen, die wir aus dem C-Code heraus aufrufen wollen, nach C-Namenskonvention übersetzen soll. Das tun wir, indem wir die Funktionsdefinitionen in einen **extern** "C"{...} Block schreiben.

Änderungen an der AES-Implementierung Zur Implementierung der gewünschten Funktionalität musste die Implementierung des AES-Algorithmus wie folgt verändert werden:
Die AES-Tabellen wurden vollständig aus der AES-Implementierung entfernt und in die

Klasse AESData verschoben. Damit bei jedem Zugriff auf ein Tabellenelement eine Aktion durchgeführt werden kann, wurden die Tabellen als „private“ deklariert und folgende Funktionen zum Zugriff auf die Tabellenelemente definiert:

```
const u32 getTe0(int pos, int round);
const u32 getTe1(int pos, int round);
const u32 getTe2(int pos, int round);
const u32 getTe3(int pos, int round);
const u32 getTe4(int pos, int round);
const u32 getTd0(int pos, int round);
const u32 getTd1(int pos, int round);
const u32 getTd2(int pos, int round);
const u32 getTd3(int pos, int round);
const u32 getTd4(int pos, int round);
```

Jede dieser Funktionen dient zum Zugriff auf ein Element je einer Tabelle. Jeder Tabelle ist eine dieser Funktionen zugeordnet. Der Parameter „pos“ gibt dabei den Index des Tabellenelements an, das zurückgegeben werden soll. Der Parameter round gibt an, in welcher AES-Runde dieser Zugriff stattfindet. Im AES-Algorithmus der Implementierung wurden daher alle Referenzen auf eine der Tabellen durch eine dieser Funktionen ersetzt. Als Parameter round wurde jeweils die Nummer der Runde, in der der Zugriff stattfindet, eingetragen (zwischen 1 und 10) und für alle Tabellenzugriffe, die nicht in einer der AES-Runden stattfinden, wurde 0 eingesetzt.

Zudem wurden folgende Funktionsaufrufe zur AES-Implementierung hinzugefügt:

- **void** init_cache_sim(**char** ∗conffile);
  Die Funktion init_cache_sim initialisiert die Cache-Simulation und erstellt Instanzen der drei eben vorgestellten Klassen. Als Parameter bekommt sie den Dateinamen der Konfigurationsdatei, in der unter anderem die Cache-Parameter festgelegt werden.

- **void** do_statistics ();
  Die Funktion do_statistics gibt die Ergebnisse des Cache-Angriffs aus. Das ist eine Zuordnung zwischen Cache-Set-Nummern und der Anzahl von Cache-Lines im Set, auf die zugegriffen wurde.

Cache-Simulator Der Cache-Simulator ist in der Klasse CacheSim implementiert. Der Cache-Simulator erstellt anhand der Cache-Parameter, also der Anzahl der Cache-Sets S, der Anzahl der Cache-Lines pro Set W und der Größe einer Cache-Line B, eine Datenstruktur, die einem Cache ähnelt. Die Klasse CacheSim verfügt über folgende Methode, mit der Speicherzugriffe simuliert werden:

```
void memory_access(unsigned int address);
```

Diese Funktion bekommt als Parameter eine Adresse. Diese Adresse muss keine echte Adresse sein, sondern kann ein beliebiger Integer-Wert sein. Der Cache-Simulator teilt

diese Adresse auf in Tag-, Set- und Offset-Bits auf und bestimmt, in welche Cache-Line in seinem simulierten Cache die Daten an dieser Adresse zwischengespeichert würden. Dies macht er nach demselben Verfahren, wie es ein Cache machen würde. Genauso wie ein Cache, belegt er im entsprechenden Cache-Set eine Cache-Line und speichert die Tag-Bits in dieser Cache-Line. Im Gegensatz zum richtigen Cache speichert er aber keine Daten im Cache. Da wir nur das Verhalten eines Caches analysieren, aber nicht die Funktion eines Caches benötigen, ist das überflüssig.

Über die folgende Methode der Klasse CacheSim lassen sich vom Cache-Simulator genau die Informationen abfragen, die die Simulation des Cache-Angriffs ergeben hat:

```
unsigned int *getCacheStats();
```

Diese Funktion liefert ein Array von Integer-Werten zurück. Die Anzahl der Integer-Werte entspricht der Anzahl der Cache-Sets des simulierten Caches. Nehmen wir an, der Cache-Simulator simuliert einen Cache mit S Cache-Sets. Dann haben die Cache-Sets Adressen von 0 bis $S - 1$. Für jede Adresse eines Cache-Sets i, mit $0 \leq i \leq S - 1$, ist der Integer-Wert n an der Stelle i im Array null, wenn der Cache-Simulator keinen Zugriff auf eine Cache-Line in dem Cache-Set simuliert hat. Der Wert n ist eins, wenn der Cache-Simulator Zugriffe auf genau eine Cache-Line in dem Cache-Set simuliert hat. Für mehrfach assoziative Caches gilt, dass der Wert n der Anzahl der Cache-Lines in dem Cache-Set entspricht, auf die zugegriffen wurde. Grundsätzlich gilt $n \leq W$, da nicht auf mehr Cache-Lines zugegriffen worden sein kann, als es Cache-Lines in dem Cache-Set gibt.

Literaturverzeichnis

[Ber05] D.J. Bernstein. Cache-timing attacks on AES. *preprint available from http://cr. yp. to/papers. html# cachetiming*, 2005.

[BO03] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmers Perspective*. Prentice Hall, 2003.

[DR99] J. Daemen and V. Rijmen. AES Proposal: Rijndael, 1999.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[fip01] Specification for the Advanced Encryption Standard (AES)(FIPS 197). Technical report, National Institute of Standards and Technology, 2001.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[Per05] C. Percival. Cache missing for fun and profit. *Proc. of BSDCan*, 2005.

[Sti02] Douglas R. Stinson. *Cryptography, Theory and Practice, Second Edition*. Chapman&Hall/CRC, 2002.