Analyse kryptographischer Algorithmen: Serpent

Michael Ueckerdt
mailto://ueckerdt@informatik.hu-berlin.de

23. August 2002

1 Einleitung

Dieses Seminar beschäftigt sich mit modernen aber etwas weniger bekannten kryptographischen Algorithmen. Als Ausgangspunkt wurden dabei Kandidaten für den Advanced Encryption Standard (AES) betrachtet.

In dieser Abhandlung möchte ich den kryptographischen Algorithmus Serpent vorstellen. Serpent wurde von Ross Anderson, Eli Biham und Lars Knudsen entwickelt und als Vorschlag für den AES eingereicht. Zusammen mit Rijndael, MARS, RC6 und Twofish erreichte Serpent die zweite Runde, in welcher Sicherheits- und Performanceaspekte der fünf Kandidaten vergleichend analysiert wurden. Aus der zweiten Runde ging Rijndael als Sieger und damit als AES hervor.

Die Spezifikation von Serpent wurde von den Entwicklern offengelegt und der Algorithmus zur Verwendung freigegeben. Dennoch liegen mir derzeit keine Hinweise auf den praktischen Einsatz von Serpent vor.

2 Der Algorithmus

2.1 Eigenschaften

Serpent ist ein symmetrisches Blockchiffre, welches mit 128-Bit Blöcken arbeitet. Intern wird mit 256-Bit Schlüsseln gearbeitet, kürzere Schlüssel werden zu 256 Bit aufgefüllt. Damit sind die Mindestanforderungen an die AES Kandidaten (128 Bit Blockgröße, mögliche Schlüssellängen: 128, 192 und 256 Bit) erfüllt.

Die Verschlüsselung besteht aus 32 Runden, welche nach Einschätzung der Entwickler einen hohen Sicherheitsspielraum schaffen. Der Algorithmus ist sehr konservativ aufgebaut, d.h. es werden ausschließlich bereits länger bekannte und

daher gut untersuchte Verfahren verwendet. Dadurch ist es nach Ansicht der Entwickler unwahrscheinlicher, daß neue wesentlich effizientere Angriffe gegen Serpent gefunden werden.

2.2 Die Verschlüsselungsfunktion

Zur Verschlüsselung werden auf jeden einzelnen Block nacheinander die 32 Rundenfunktionen angewandt.

$$\begin{aligned} B_0 &= IP(plaintext) \\ B_{i+1} &= Round_i(B_i, Key) \qquad i = 0, .., 31 \\ ciphertext &= FP(B_{32}) \end{aligned}$$

Dabei bezeichnen IP und FP schlüsselunabhängige Permutationen der übergebenen Blöcke. Sie haben keine kryptographische Bedeutung sondern sollen eine effizientere Implementation gestatten, auf die später noch eingegangen wird. FP ist dabei die Inverse von IP.

Serpent verwendet acht 4x4 S-Boxen, d.h. injektive Funktionen von und auf die Menge der 4-Bit-Folgen, welche von den Entwicklern in der Algorithmusspezifikation angegeben sind. Sie wurden deterministisch aus den DES S-Boxen generiert, um so die Sicherheitseigenschaften dieser bereits sehr gut untersuchten S-Boxen zu übernehmen. So garantieren sie unter anderem, daß sich bei jeder Änderung eines Eingabe-Bits mindestens zwei Ausgabe-Bits verändern. Jede Runde verwendet jeweils eine der S-Boxen in 32-fach paralleler Ausführung (32 * 4 Bit = 128 Bit = Blockgröße).

$$\begin{aligned} Round_i(X, Key) &= LT(S_{i \bmod 8}(X \oplus K_i)) \qquad i = 0, .., 30 \\ Round_{31}(X, Key) &= S_0(X \oplus K_{31}) \oplus K_{32} \end{aligned}$$

S_i bezeichnet hierbei die 32-fach parallele Anwendung der i-ten S-Box, die K_i sind die einzelnen 128-Bit Rundenschlüssel, die aus dem originalen 256-Bit Schlüssel (Key) generiert werden und LT bezeichnet eine lineare Transformation.

Zunächst fällt auf, daß sich die letzte Runde von allen vorhergehenden unterscheidet. Würde man hier wie in den anderen Runden die lineare Transformation verwenden, könnte diese von einem Angreifer problemlos wieder rückgängig gemacht werden, da sowohl die Transformation als auch ihre Inverse in der Algorithmusspezifikation angegeben sind. In den anderen Runden dient die lineare Transformation hauptsächlich zur Erzeugung des sogenannten Lawineneffekts,

d.h. daß sich die Änderung eines einzelnen Eingabe-Bits sehr schnell auf sehr viele Ausgabe-Bits auswirkt. Die lineare Transformation $LT(X)$ kann wie folgt durchgeführt werden:

$$\begin{aligned}
X_0, X_1, X_2, X_3 &= FP(X) \\
X_0 &= X_0 <<< 13 \\
X_2 &= X_2 <<< 3 \\
X_1 &= X_1 \oplus X_0 \oplus X_2 \\
X_3 &= X_3 \oplus X_2 \oplus (X_0 << 3) \\
X_1 &= X_1 <<< 1 \\
X_3 &= X_3 <<< 7 \\
X_0 &= X_0 \oplus X_1 \oplus X_3 \\
X_2 &= X_2 \oplus X_3 \oplus (X_1 << 7) \\
X_0 &= X_0 <<< 5 \\
X_2 &= X_2 <<< 22 \\
LT(X) &= IP(X_0, X_1, X_2, X_3)
\end{aligned}$$

Die Eingabe X wird also zurückpermutiert und in vier 32-Bit Wörter zerlegt. Diese werden dann rotiert ($<<<$), geshiftet ($<<$), XOR-verknüpft und am Ende aneinandergesetzt und wieder zurechtpermutiert. Da nur einfache bitweise Operationen verwendet werden, läßt sich die Transformation einfach und effizient in Soft- und Hardware implementieren. Über diese lineare Transformation stellen die Entwickler sicher, daß nach drei Runden jedes Eingabe-Bit jedes Ausgabe-Bit beeinflußt hat.

2.3 Die Schlüsselgenerierung

Wie aus der Algorithmusbeschreibung ersichtlich wird für jede Runde ein und für die letzte zwei 128-Bit Schlüssel benötigt. Diese werden aus dem originalen Schlüssel erzeugt. Besteht der gewählte Schlüssel aus weniger als 256 Bit wird er durch Anhängen einer 1 und einer variablen Anzahl von Nullen auf die nötigen 256 Bit aufgefüllt. Dieser Schlüssel wird nun in acht 32-Bit Wörter zerlegt:

$$w_{-8}, w_{-7}, w_{-6}, w_{-5}, w_{-4}, w_{-3}, w_{-2}, w_{-1} = Key$$

Daraus werden nun iterativ 132 sogenannte prekeys gebildet.

$$\begin{aligned} w_i &= (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \Phi \oplus i) <<< 11 \qquad i = 0, .., 131 \\ \Phi &= \frac{\sqrt{5}+1}{2} \end{aligned}$$

Dabei entspricht Φ der Nachkommastelle des goldenen Schnitts. Diese Bildungsvorschrift soll sicherstellen, daß die Schlüsselbits möglichst gleichmäßig auf die einzelnen prekeys verteilt werden. Die XOR-Verknüpfung mit i verhindert darüberhinaus schwache Schlüssel.

Diese prekeys werden nun schließlich nach folgendem System zu den 33 Rundenschlüsseln zusammengesetzt:

$$\begin{aligned} K_0 &= S_3(IP(w_0, w_1, w_2, w_3)) \\ K_1 &= S_2(IP(w_4, w_5, w_6, w_7)) \\ K_2 &= S_1(IP(w_8, w_9, w_{10}, w_{11})) \\ K_3 &= S_0(IP(w_{12}, w_{13}, w_{14}, w_{15})) \\ K_4 &= S_7(IP(w_{16}, w_{17}, w_{18}, w_{19})) \\ K_5 &= S_6(IP(w_{20}, w_{21}, w_{22}, w_{23})) \\ &\dots \\ K_{31} &= S_4(IP(w_{124}, w_{125}, w_{126}, w_{127})) \\ K_{32} &= S_3(IP(w_{128}, w_{129}, w_{130}, w_{131})) \end{aligned}$$

2.4 Implementation im bitslice-Modus

Eine effiziente Möglichkeit zur Implementation von Serpent bietet der sogenannte bitslice-Modus. Die Idee der Implementation soll hier kurz vorgestellt werden, da das Design des Algorithmus explizit darauf zugeschnitten ist. Die Idee des bitslice-Modus ist es, eine S-Box nicht als eine Funktion von und auf 4-Bit Folgen, sondern als vier boolesche Funktionen zu betrachten.

Klassische Modellierung einer S-Box:

$$(out_0, out_1, out_2, out_3) = S(in_0, in_1, in_2, in_3)$$

Alternative Modellierung der S-Box:

$$\begin{aligned} out_0 &= f_0(in_0, in_1, in_2, in_3) \\ out_1 &= f_1(in_0, in_1, in_2, in_3) \\ out_2 &= f_2(in_0, in_1, in_2, in_3) \\ out_3 &= f_3(in_0, in_1, in_2, in_3) \end{aligned}$$

Die 32-fach parallele Anwendung einer S-Box entspricht also entweder der Berechnung der entsprechenden 4-Bit Funktion für 32 Eingaben, oder aber einer Permutation der Eingabe-Bits, der einmaligen Berechnung der vier booleschen Funktionen auf 32-Bit Wörtern, sowie einer Permutation der Ausgabe-Bits (invers zur Permutation der Eingabe). Die bitweise Verknüpfung von 32-Bit Wörtern kann im allgemeinen wesentlich schneller durchgeführt werden, als 32 Zugriffe auf die Tabelle der S-Box.

Da bei Serpent der bitslice-Modus bereits während der Entwicklungsphase als Standardimplementation vorgesehen war, entschieden sich die Entwickler die Bits nicht beim bitslice-Modus zu permutieren, sondern nur in dem anderen Modus. Diese Permutationen sind in der Algorithmusbeschreibung in Form von IP (initial permutation) und FP (final permutation) präsent. Somit wird durch Entfernen aller Anwendungen von IP und FP aus den obigen Formeln, die Darstellung des Algorithmus in den bitslice-Modus übertragen.

3 Analyse von Serpent

3.1 Sicherheit

Serpent ist seit 1998 veröffentlicht und wurde seitdem vor allem im Rahmen der AES-Konferenzen untersucht. Es sind derzeit keine Schwächen bekannt, die zu einem wirksamen Angriff führen könnten. Zunächst sollen einige der Angriffe vorgestellt werden, welche die Entwickler selbst betrachtet haben.

- **vollständige Schlüsselsuche**
  Ein klassischer Brute Force-Angriff, der bei allen AES-Kandidaten funktioniert. Hierbei werden alle Schlüssel durchprobiert, der Rechenaufwand entspricht somit der Menge der möglichen Schlüssel. Es wird nur ein Klartext / Kryptotext-Beispiel benötigt.

- **Wörterbuch-Angriff**
  Ebenfalls ein Brute Force-Angriff, der auf alle AES-Kandidaten angewandt werden kann. Es werden dazu alle möglichen Klartextblöcke sowie die dazugehörigen Kryptotextblöcke benötigt. Der Angreifer kann damit jeden Kryptotextblock entschlüsseln, indem er den entsprechenden Klartextblock in seiner Sammlung sucht. Der Platzbedarf ist ausschließlich von der Anzahl der möglichen Blöcke und damit von der Blocklänge abhängig (welche für die AES-Kandidaten auf 128 Bit festgelegt ist).

- **CBC- und CFB-Modi**
  Werden 2^{64} Kryptotextblöcke im CBC- oder CFB-Modus übertragen, liegt die Wahrscheinlichkeit bereits bei 1/2 daß zwei identische Kryptotextblöcke

dabei waren. Daraus lassen sich dann einzelne Bits der nachfolgenden Klartexte rekonstruieren (je mehr identische Kryptotextblöcke auftauchen, desto mehr kann vom Angreifer rekonstruiert werden). Eine typische Gegenmaßnahme ist der Wechsel des verwendeten Schlüssels nach der Übertragung einer gewissen Anzahl von Blöcken in einem der beiden Modi. Dieser Angriff kann ebenfalls auf jedes Blockchiffre angewandt werden, und der Platzbedarf ist ausschließlich von der Blocklänge abhängig.

- **differentielle Kryptoanalyse**
  Diese Verfahren basieren auf der Untersuchung sehr ähnlicher Klartexte und den Unterschieden in den dazugehörigen Verschlüsselungen. Da bei Serpent jedoch die Änderungen eines einzelnen Eingabe-Bits bereits nach wenigen Runden alle Ausgabe-Bits beeinflußt, sehen die Entwickler keine Ansatzpunkte für solche Verfahren.

- **lineare Kryptoanalysis**
  Hierbei sucht der Angreifer nach linearen Abhängigkeiten zwischen Eingabe- und Ausgabe-Bits der S-Boxen, die mit einer Wahrscheinlichkeit ungleich 50% (das wäre der ideale Zufall) auftreten. Solche Abhängigkeiten können unter Umständen zu linearen Abhängigkeiten zwischen der Eingabe und der Ausgabe des gesamten Algorithmus führen, welche dann ebenfalls mit Wahrscheinlichkeiten ungleich 50 % auftreten. Aufgrund der sorgfältigen Auswahl der S-Boxen und der hohen Rundenzahl liegen diese Wahrscheinlichkeiten bei Serpent (für den kompletten Algorithmus, nicht für jede einzelne S-Box) jedoch so nahe an den 50%, daß sie nicht für Angriffe genutzt werden können.

- **verwandte Schlüssel** Der Angreifer verschlüsselt einen bekannten Klartext mit verschiedenen Schlüsseln, die sich an bestimmten Stellen unterscheiden. Durch den Vergleich der Ergebnisse mit dem echten Kryptotext versucht er dann Rückschlüsse auf den dort verwendeten Schlüssel zu ziehen. Die Entwickler begründen die Immunität von Serpent gegenüber solchen Angriffen damit, daß sich bereits geringste Änderungen an den Rundenschlüsseln nach wenigen Runden auf sehr viele Schlüssel-Bits der nachfolgenden Rundenschlüssel auswirken.

- **Timing Attack** Dieses Verfahren geht davon aus, daß dem Angreifer Daten über die Dauer der Rechenoperationen zur Verfügung stehen, die während der Verschlüsselungen benutzt wurden. Daraus versucht er dann Rückschlüsse auf die beteiligten Operanden zu ziehen. Die von Serpent verwendeten Operationen `and`, `or`, `xor`, `not`, `rotate` und `shift` sind für solche Verfahren allerdings kaum anfällig, da sie annähernd feste (d.h. von den Operanden unabhängige) Ausführungszeiten haben.

Auch im AES-Report zur zweiten Runde finden sich Untersuchungen zur Sicherheit der fünf Finalisten (Rijndael, Serpent, MARS, RC6 und Twofish). Bei keinem dieser Algorithmen konnten Hinweise auf mögliche Angriffe gefunden werden. Um dennoch ein Maß für die Sicherheit der Algorithmen zu finden, hat man Angriffe auf vereinfachte Versionen der Algorithmen versucht. Anhand des Grades der dazu nötigen Vereinfachung, wurde den Algorithmen dann ein großer bzw. ein angemessener Sicherheitsspielraum bescheinigt.

Die Vereinfachungen bei Serpent bestanden im wesentlichen in der Reduktion der Rundenzahl. Die nachfolgende Tabelle gibt einen Überblick über mögliche Angriffe auf Serpent-Versionen mit reduzierter Rundenzahl und den jeweils dazu benötigten Klartextblöcken. Hierbei wurde alles als Angriff gewertet, was weniger Klartexte als der Brute Force-Angriff benötigt (der Wörterbuchangriff benötigt 2^{128} Klartextblöcke).

Anzahl der Runden	**Verfahren**	**benötigte Klartexte**
6 Runden	differentielle Kryptoanalysis	2^{41} Klartextblöcke
7 Runden	differentielle Kryptoanalysis	2^{122} Klartextblöcke
8 und 9 Runden	amplified Boomerang	2^{110} Klartextblöcke

Es konnten also maximal neun Runden angegriffen werden, was bei einer Gesamtrundenzahl von 32 eine beträchtliche Vereinfachung darstellt. Untersuchungen im Rahmen der AES-Konferenzen ergaben außerdem, daß nach vier und mehr Runden die Ausgabe von Serpent Eigenschaften einer zufälligen Bitfolge hat. Serpent hat eine einfache und gut zu analysierende Struktur und verwendet nur konventionelle und bereits gut untersuchte Verfahren, bei denen das Risiko daß ein neuer effizienter Angriff gefunden wird, vergleichsweise gering ist. Insgesamt wurde Serpent deshalb ein hoher Sicherheitsspielraum bescheinigt (ebenso wie MARS und Twofish), während RC6 und Rijndael ein angemessener Sicherheitsspielraum attestiert wurde.

3.2 Performance

Der zweite wesentliche Punkt der AES-Untersuchungen war die Performance der Algorithmen. Da alle fünf Kandidaten als sicher eingestuft wurden, gab das Ergebnis der Performance-Analyse den Ausschlag für den zukünftigen AES-Algorithmus. Nachfolgend nun eine kurze Zusammenfassung der Ergebnisse dieser Untersuchungen.

In Software-Implementationen stellte sich Serpent sprach- und systemübergreifend als relativ langsam im Vergleich zu den anderen Finalisten dar. Verantwortlich dafür dürfte vor allem die hohe Rundenzahl sein. Der Speicherbedarf einer Serpent-Implementation ist gering (hier war Serpent zweitbester der Algorithmen).

Für Hardware-Implementationen ist Serpent mit einem sehr guten Datendurchsatz dagegen gut geeignet. Beim Speicherbedarf wirkte sich nachteilig aus, daß für die Entschlüsselung die inverse lineare Transformation und die inversen S-Boxen extra vorgehalten werden müssen. Außerdem müssen die Rundenschlüssel zwischengespeichert werden, da für die Entschlüsselung die Rundenschlüssel in umgekehrter Reihenfolge benötigt werden (d.h. der zuletzt erzeugte wird zuerst verwendet). Serpent verwendet nur einfache bitweise Operationen, so daß die Hardware-Implementation sehr einfach ist.

Im Vergleich zu den anderen Algorithmen schnitten die Software-Implementationen von Serpent schlecht und die Hardware-Implementation gut ab. Rijndael konnte dagegen in beiden Kategorien überzeugen, und wurde schließlich als AES-Algorithmus gewählt.

4 Quellen

- Ross Anderson, Eli Biham, Lars Knudsen: 'Serpent: A Proposal for the Advanced Encryption Standard'
  `http://www.cl.cam.ac.uk/ftp/users/rja14/serpent.pdf`
- Ross Anderson, Eli Biham, Lars Knudsen: 'The Case for Serpent'
  `http://www.cl.cam.ac.uk/ftp/users/rja14/serpentcase.pdf`
- James Nechtvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, Edward Roback:
  'Report on the Development of the Advanced Encryption Standard (AES)'
  `http://csrc.nist.gov/encryption/aes/round2/r2report.pdf`
- Reinhard Wobst: 'Abenteuer Kryptologie', Addison Wesley, 1998