Keygarantie Verschlüsselungsverfahren

Christopher Besch und Katharina Libner

13. März 2021

Inhaltsverzeichnis

1 Abstract

Keygarantie ist ein symmetrisches 128-bit Stromverschlüsselungsverfahren, das eine Anlehnung an und Erweiterung von ChaCha darstellt [J.08]. Damit basiert es auf XOR, was die Ver- und Entschlüsselung mit dem selben Algorithmus erlaubt. Für die Generation des mit XOR verwendeten Stromschlüssels wird ein komplexer und praktisch äußerst schwer umkehrbarer Mechanismus aus mehreren mathematischen Methoden verwendet. Selbst bei kompromittiertem Klartext bleibt daher die verwendete Passphrase geheim.

2 Übersicht

Das Kernstück von Keygarantie bildet die XOR-Verschlüsselung, wobei ein Klartext mit einem Stromschlüssel bitweise verschlüsselt wird. Eine 1 in dem Stromschlüssel dreht den gegenüberliegenden Bit im Klartext um, eine 0 verändert ihn nicht. Wenn nun der so generierte Geheimtext das Verfahren mit dem selben Schlüssel erneut durchläuft, werden die umgedrehten Bits erneut verändert, sodass wieder der Klartext entsteht. Dadurch kann zur Ver- und Entschlüsselung der gleiche Algorithmus verwendet werden.

Allerdings ist davon auszugehen, dass Teile des Klartextes kompromittiert werden. Ist dies der Fall, kann zusammen mit dem öffentlichen Geheimtext auf den verwendeten Stromschlüssel geschlossen werden. Dadurch wären alle zukünftigen Transmissionen ebenfalls kompromittiert. Dies wäre fatal.

Ein Vorteil ist, dass Bits und keine Buchstaben oder Zahlen verschlüsselt werden. Keygarantie findet damit für jegliche Dateien Anwendung. Aes muss den gesamten Input erst in den Speicher laden [M.P12]. Keygarantie muss dies durch die Stromverschlüsselung nicht und kann Streams on-the-fly ver- und entschlüsselt.

Es muss also davon ausgegangen werden, dass unter Umständen der Stromschlüssel öffentlich bekannt ist. Daher ist sicherzustellen, dass jeder Teil des Stromschlüssels praktisch nie erneut für die Verschlüsselung mit XOR verwendet wird. Um dies zu erreichen, diffundiert sich ein 128-bit Schlüssel in einer Matrix über 30 Runden. Dieses Verfahren generiert einen 256-bit Seed.

Ein Pseudozufallsgenerator generiert eine unendliche Menge an Bits für den Stromschlüssels zu generieren, determiniert durch einen 256-bit Seed. Praktisch werden allerdings aus Sicherheitsgründen nur 512 Bits pro Seed produziert, wodurch weniger Information für einen Angreifer vorhanden sind. Diese 512 Bits bilden einen Blockschlüssel. Daher muss der Klartext in 512-bit Blöcke unterteilt werden, die aufsteigend und eindeutig mit einer Blocknummer nummeriert sind. Hieraus folgt, dass für jeden einzelnen Block ein anderer Seed für den Pseudozufallsgenerator verwendet werden muss. Durch diesen Schritt, der mit modernen Methoden praktisch unumkehrbar ist, wird die geforderte Sicherheit garantiert.

Durch die genannte Einteilung in Blöcke ergibt sich ein weiterer Vorteil: Es kann zu einer bestimmten Stelle innerhalb des Streams gesprungen werden, ohne erst alle vorherigen Blockschlüssel berechnen zu müssen.

3 Linearer Kongruenzgenerator

Das Ziel ist es, einen theoretisch unendlich langen Schlüsselstrom zu erzeugen, der von einem Seed abhängt, allerdings nicht auf diesen schließen lässt. Er soll wie zufälliges Rauschen aussehen. Kein Teil des Schlüsselstroms soll in einem praktischen Bereich erneut vorkommen. Hierzu wird davon ausgegangen, dass jeder benutzte Seed einzigartig ist.

Der lineare Kongruenzgenerator erzeugt als Schlüsselstromgenerator zufällig scheinende Werte x_i, die allerdings durch ihren Vorgänger x_{i-1} determiniert sind. Daher ergibt sich eine rekursive Funktion:

$$x_i = ((x_{i-1} \cdot a) + c) \bmod 2^n$$

- n sei aus $\mathbb{N}$ und markiert die Bitintervallgrenze in der x_i liegt. Für das Intervall gilt also: $[0, n]$. n ist eine öffentlich bekannte Konstante mit dem Wert 64.
- c sei aus $\mathbb{N} < 2$ und ist für dieses Verfahren mit 1 festgesetzt.
- a sei aus $\mathbb{N} < 2$ und entspricht 134775813.

Da jeder Wert aus der Menge aller x_i erst durch einen vorherigen Wert entsteht, muss es einen Startwert x_0 geben, den Seed. Der Seed ist abgesehen von den generierten Werten die einzige einem Angreifer unbekannte Variable. Da $n = 64$ produziert der Generator Werte zwischen 0 und 2^{64}.

Dieses Verfahren weist jedoch eine Periode nach spätestens 2^{64} Werten auf, nach der sich die Ausgaben wiederholen. Da jeder Wert durch den Vorgänger determiniert ist, kann jeder pro Periode nur einmal vorkommen. Somit weiß ein Angreifer, dass wenn zum Beispiel die Zahl 143 ausgegeben wurde, diese Zahl erst nach Periodenende erneut vorkommt. Das ist ein Problem.

Um dies zu umgehen wird nur die Hälfte jedes Wertes ausgegeben und für die eigentliche XOR Verschlüsselung eingesetzt. Intern wird geheim weiterhin der ganze Wert für die Generation des Nächsten verwendet. Daher wird der Bereich der ausgegebenen Werte auf bis zu 2^{32} verkleinert.

So wird die Periode nicht umgangen, weshalb nur eine begrenzte Menge an Werten pro Seed verwendet werden darf. Das Problem der in einer Periode nicht wiederkehrenden Werte, wurde so allerdings gelöst; es kann nicht genau gesagt werden, wie oft ein bestimmter Wert pro Periode vorkommt.

4 Matrix

Das Ziel dieses Schrittes ist es, einen einzigartigen Seed zu generieren. Dies setzt der Schlüsselstromgenerator voraus. Zudem ist dieser Rechenschritt praktisch unumkehrbar. Es wird eine 4×4 Matrix verwendet, die aus folgenden Teilen besteht:

- einer öffentlichen Konstante, sie entspricht der ASCII-Repräsentation von

  „ILoveYou“,

- dem geheimen Schlüssel,

- der öffentlichen Blocknummer und
- einer ebenfalls öffentlichen Nonce. Die Nonce ist eine „**N**umber used **ONCE**“, die pro Verschlüsselung neu gewählt werden muss. Hierdurch werden Replay-Angriffe praktisch unmöglich, da so ein kompromittierter Stromschlüssel nie erneut zur Ver- und Entschlüsselung eingesetzt werden kann.

Die Elemente der Matrix sind 16-bit Wörter und sind im Ausgangszustand wie folgt angeordnet:

const	const	const	const
key	key	key	key
key	key	key	key
block_num	block_num	nonce	nonce

Diese Matrix durchläuft 30 Runden an Diffusion, wodurch eine geringfügige Änderung eines der Eingangswörter die gesamte Ausgangsmatrix beeinflusst.

Jede Runde besteht aus acht Rechenschritten. Für alle werden vier Wörter aus der Matrix ausgewählt und dem in Unterabschnitt 4.1 beschriebenen Verfahren übergeben. Die Ausgaben dieser Berechnung ersetzten die entsprechenden Eingangswerte.

Die Auswahl der vier Wörter ist vordefiniert. Diese wird von ChaCha übernommen [J.08]. Wenn die Wörter zeilenweise nummeriert werden, ergibt sich die folgenden Reihenfolge der Rechenschritte:

1. Spalten:
   - Achtelrunde(00, 04, 08, 12)
   - Achtelrunde(01, 05, 09, 13)
   - Achtelrunde(02, 06, 10, 14)
   - Achtelrunde(03, 07, 11, 15)
2. Diagonalen:
   - Achtelrunde(00, 05, 10, 15)
   - Achtelrunde(01, 06, 11, 12)
   - Achtelrunde(02, 07, 08, 13)
   - Achtelrunde(03, 04, 09, 14)

4.1 Achtelrunden

Jede Achtelrunde nimmt vier 2-bit Wörter (a bis d) als Eingabe entgegen und produziert eine Ausgabe von vier 2-bit Wörtern. Das Ziel dieses Verfahren ist es, alle Eingangswörter derartig miteinander in Beziehung zu setzten, dass eine kleine Änderung eines Eingangswortes alle Ausgaben ausschalgebend ändert. Hierzu werden die folgenden Berechnungen durchgeführt:

$$
\begin{aligned}
a &+= b \\
d &\wedge= b \\
d &<<<= 6
\end{aligned}
$$

$$
\begin{aligned}
c &+= d \\
b &\wedge= c \\
c &<<<= 12
\end{aligned}
$$

$$
\begin{aligned}
a &+= b \\
d &\wedge= b \\
d &<<<= 8
\end{aligned}
$$

$$
\begin{aligned}
c &+= d \\
b &\wedge= c \\
c &<<<= 7
\end{aligned}
$$

Es wird die übliche Schreibweise der C Programmiersprache verwendet. [J.08] setzt die selbe ein.

- $\wedge$ steht für XOR,
- $+$ für Addition modulo 2^{16} und
- $<<< b$ bitweise Rotation um b-bits nach links.

5 Hashfunktion

Zur Generierung des in der Matrix verwendeten Schlüssels wird eine Hashfunktion verwendet. Sie muss nicht besonders sicher und unumkehrbare sein, da der Schlüsslestromgenerator und die Matrix diese Aufgabe bereits übernehmen. Der Zweck dieser ist statt dessen die Reduzierung—beziehungsweise Erweiterung—einer Passphrase auf die von der Matrix geforderten 128 Bits. Jegliche ASCII-Zeichen können so für die Passphrase verwendet werden. Daher wird das simple Division-Rest-Verfahren [Ant21] angewendet.

- Sei s_0 bis s_{n-1} die ASCII-Repräsentation einer Passphrase der Länge n,
- h der zu generierende Hashwert und
- $p = 2^{128}$ die Begrenzung des Hashwertes, die von der Matrix vorgegeben ist.

Es gilt:

$$h = \left(s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + ... + s_{n-2} \cdot 31 + s_{n-1}\right) \bmod p$$

6 Implementationsabhängige Eigenschaften

Das genannte Verfahren wird in Java implementiert.

Die Eingabedatei wird ausgelesen und in Blöcke unterteilt. Dabei beträgt die Größe dieser 512 Bits. Andere Größen sind ebenfalls möglich. Allerdings muss eine Balance zwischen

- einer hohen kryptographischen Sicherheit bei vielen, kleineren Blöcken und
- einer geringen Laufzeit bei wenigeren, größeren Blöcken.

Diese variable Blockgröße ergibt sich aus der theoretisch unbegrenzten Länge der Blockschlüssel. Zuerst wird die eingegebene Passphrase in einen 128-bit Hashwert konvertiert.

Das Programm liefert die Option, die Nonce zufällig selbst zu generieren. Zudem erlaubt es die aneinanderfolgende Ver- und Entschlüsselung mehrere Dateien. Wenn hierzu die Nonce vom Nutzer gewählt wird, wird für alle Dateien die Gleiche verwendet. Dies ist ein Sicherheitsrisiko, da so ein Angreifer, der eine dieser Dateien entschlüsselt hat, alle anderen Dateien ebenfalls entschlüsseln kann.

Außerdem weist das Programm abgesehen von der Hashfunktion für eine Datei der gleichen Länge immer eine konstante Laufzeit auf. Dies ist eine Eigenschaft, die mit AES nur in der Implementation erreicht werden kann [M.P12]; hier ist sie immer gegeben. Sei die Laufzeit von der verwendeten Passphrase oder Klartext abhängig. Dann erhält ein Angreifer, der den Stromverbrauch oder die verwendeten CPU-Zyklen misst, eigentlich geheime Informationen.

7 Bespiele

7.1 Textdatei

Das Programm wird mit einigen Beispieldateien ausgeführt. Der folgende Text

```
Is there anything that we associate more closely with intelligence than curiosity? Every intelligent
species on Earth is attracted by the unknown. Our mythologies are full of riddles and mysteries and
divine knowledge. Even the word "apocalypse". Even the word "apocalypse" means revelation.

It seems like our ancestors always imagined that even at the very end, we would solve one last
mystery.
—Alexandra Drennan
```

wird mit diesem Befehl verschlüsselt:

```
\$ java keygarantie -p "Hello World" -n 1984
examples/origin/curiosity.txt examples/enc/curiosity.txt
encrypting 'examples/origin/curiosity.txt' with nonce 1984...
All done, enjoy!
```

Es ergibt sich die folgende Ausgabe, die nur annähernd dargestellt werden kann:

```
I\F1\E43\BBG\A9;\98U I `,\E6^R\92\B3\C9\\A5\E0:\EE\FC\D3Im)\DD\C1:|
ΣII\D3\CA*6\D56sr^P\85\CAb\85W3 5\CC\E9\BAc\E7\E43\BBC\B5~Y\8CI R z2\F9
 \85\A2\9BR\E0\A95\E9\EA\D0Fm/\CC\CAn1ϗ\C7\C2 \DA\D6y<\D7o dEP\85\CAb\98 +\00.\CA\EF\A0e\E6\E4%
\AA \AF6_\D9N\00V y%\E8
u\86\B5\C9F\B9\B43\F2\E3\D3Mm-ƍ{c\C4\F1\C4\D4 Π65\99=:aSH\88\99+\8AM\00 %
\D8\F8\B1r\EB\A14\F3C\B5: \9DR T .
\A8EM\9F\A2\8DL\A5\EE{\D8\F9\D9D$<\C1\C1:fΣBZ\DE\D560\D8#*uDA\CF\C4+
\AEU 1T(\C3\E9\F4w\ED\B6#\F3\00\BA.U\9AZ D ki\E6G_\92\A9\9A \B2\A5-\F8\E3\DD^m'U 撈
\D2 \DA\C8*s\D5&8` K\98\98+\8AM :(\C4\FE\A7 \E3\A80\B2[\A8~S\94Z  T jk\B2B[\87\E7\8C]
\A5\AE{\FC\FB\9C^l-
\89\D2 c\D8\F1\C7\CF \93\85.6\9988<p[@d\87U 2ά\B8a\F1\B0g\BE[\A8*_\8BB@7\8B\FC\9A
\AAOB\92\A9\8DY\A1\E0 \EF\EA\D2De&\A3
```

Nun wird mit dem folgenden Befehl die Datei wieder entschlüsselt:

```
\$ java keygarantie -p "Hello World" -n 1984
examples/enc/curiosity.txt examples/dec/curiosity.txt
encrypting 'examples/enc/curiosity.txt' with nonce 1984...
All done, enjoy!
```

Der originale Text ist so nun wieder lesbar:

```
Is there anything that we associate more closely with intelligence than curiosity? Every intelligent
species on Earth is attracted by the unknown. Our mythologies are full of riddles and mysteries and
divine knowledge. Even the word "apocalypse". Even the word "apocalypse" means revelation.

It seems like our ancestors always imagined that even at the very end, we would solve one last
mystery.
—Alexandra Drennan
```

7.2 Videodatei

Die oben genannten Befehle werden mit angepassten Dateipfaden auf die Videodatei *cute_ cat.mp4*, die sich in dem Unterordner *examples/orig* findet. Wie erwartet, ist die verschlüsselte Datei nicht lesbar, die entschlüsselte hingegen ist es. Diese finden sich in den Unterordnern *examples/enc* respektive *examples/dec*.

8 Fazit

Vorteile

- Das gleiche Verfahren kann zur Ver- und Entschlüsselung verwendet werden.
- Keygarantie kann mit jegliche Dateien umgehen.
- Selbst bei teilweiser Kompromittierung des Schlüsselstroms können keinen weiteren Bereiche durch einen Angriff entschlüsselt werden. Replay Angriffe sind praktisch unmöglich.
- Es können Streams ver- und entschlüsselt werden.
- Der Nutzer kann zu einem beliebigen Teil des Stream springen.
- Der lineare Kongruenzgenerator stellt eine Erweiterung zu ChaCha dar und garantiert weitere Sicherheit.
- Viele Parameter, wie die Blockgröße, die Passphrasenlänge oder die mathematischen Konstanten im linearen Kongruenzgenerator, sind ohne großen Aufwand leicht veränderbar.

- Die Laufzeit wächst linear mit der Länge des Klartextes.
- Es können jegliche mit ASCII repräsentierbare Zeichen für das Passphrase verwendet werden.

Nachteile

- Keygarantie verwendet 16-bit statt 32-bit Wörtern ChaChas in der Matrix.
- Die verwendete Implementation verwendet unter Umständen für mehrere Dateien die gleiche Nonce.

Keygarantie ist ein experimentelles, nicht empfohlenes, Verschlüsselungsverfahren, das viele Aspekte von ChaCha übernimmt, abwandelt und erweitert. Für die praktische Verwendung im Secure Socket Layer ist es ausdrücklich nicht empfohlen, da weitreichende Kryptoanalysen noch nicht durchgeführt wurden.

Ausschließlich für die in RFC 1149 standardisierte Übertragung von IP Datagrams [Wai90] ist dieses Verfahren vertretbar und empfohlen.

9 Literatur

[Ant21] Silies Antje. Implementieren von hash-funktionen. 2021.

[J.08] Bernstein Daniel J. Chacha, a variant of salsa20. April 2008.

[M.P12] Praveen M.Pitchaiah, Philemon Daniel. Implementationof advanced encryption standard algorithm. *International Journal of Scientific & Engineering Research Volume 3*, March 2012.

[Wai90] David Waitzman. Standard for the transmission of IP datagrams on avian carriers. RFC 1149, April 1990.