

Keygarantie Verschlüsselungsverfahren

Christopher Besch und Katharina Libner

13. März 2021

Inhaltsverzeichnis

1	Abstract	2
2	Übersicht	2
3	Linearer Kongruenzgenerator	3
4	Matrix	3
4.1	Achtelrunden	4
5	Hashfunktion	5
6	Implementation und Beispiele	6
7	Zusammenfassung	6
8	Literatur	6

1 Abstract

Keygarantie ist ein symmetrisches 128-bit Stromverschlüsselungsverfahren, das stark an ChaCha angelehnt ist [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, wodurch selbst bei kompromittiertem Klartext die verwendete Passphrase geheim bleibt.

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.

Hierbei ist der Vorteil, dass Bits und beispielsweise keine Buchstaben oder Zahlen verschlüsselt werden. Daher können jegliche Dateien ver- und entschlüsselt werden. Bei Aes muss der gesamte Input erst in den Speicher geladen werden [M.P12]. Dies ist bei Keygarantie durch die Stromverschlüsselung nicht notwendig. Hierdurch können beispielsweise Videostreams on-the-fly ver- und entschlüsselt werden.

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 wird ein 128-bit Schlüssel in einer Matrix über 30 Runden diffundiert. Dieses Verfahren generiert einen 256-bit Seed.

Der Seed wird anschließend in einem Pseudozufallsgenerator eingesetzt, um eine unendliche Menge an Bits für den Stromschlüssels zu generieren. Praktisch werden allerdings aus Sicherheitsgründen nur 512 Bits pro Seed verwendet, wodurch weniger Information für einen Angriff 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 werden. 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 Stream gesprungen werden, ohne erst alle vorherigen Blöcke ver- oder entschlüsseln 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.

Der lineare Kongruenzgenerator erzeugt zufällig scheinende Werte x_i , die allerdings von ihrem Vorgänger x_{i-1} abhängen. 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 geheime und 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. Dies ist ein Problem, da so ein Angreifer weiß, dass, wenn zum Beispiel die Zahl 143 verwendet wurde, diese Zahl erst nach Periodenende erneut vorkommen kann.

Um dies zu umgehen wird nur die Hälfte jedes Wertes ausgegeben und für die eigentliche XOR Verschlüsselung eingesetzt. Intern wird 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 sollte dieser Rechenschritt praktisch unumkehrbar ist. 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 **O**N-**C**E“, die pro Verschlüsselung neu gewählt werden muss. Hierdurch werden Replay-Angriffe praktisch unmöglich, da ein 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 ersetzen 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 Ausgangswörter ausschlaggebend ändert. Hierzu werden die folgenden Berechnungen durchgeführt:

$$a+ = b$$

$$d\wedge = b$$

$$d <<< = 6$$

$$c+ = d$$

$$b\wedge = c$$

$$c <<< = 12$$

$$a+ = b$$

$$d\wedge = b$$

$$d <<< = 8$$

$$c+ = d$$

$$b\wedge = c$$

$$c <<< = 7$$

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 unumkehrbar 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-Buchstaben 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:

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

6 Implementation und Beispiele

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 Zusammenfassung

Keygarantie ist ein experimentelles, nicht empfohlenes, Verschlüsselungsverfahren, das viele Aspekte von ChaCha übernimmt und abwandelt. Für die praktische Verwendung im Secure Socket Layer ist es nicht empfohlen, da weitreichende Kryptoanalysen noch nicht durchgeführt aber für dieses Anwendungsgebiet notwendig wären.

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

8 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. Implementation of 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.