Escuela
Politécnica
Superior

Analysis and Design of a Stream Cipher

Máster Universitario en Ciberseguridad

Trabajo Fin de Máster

Autor:
Petar Alexandrov Nikolov

Tutor/es:
Dr. Rafael Ignacio Álvarez Sánchez

Junio 2019

Universitat d'Alacant
Universidad de Alicante

I would like to acknowledge the help received from the 2 people that made this project real. These are my partner, Iris, and my supervisor, Rafa.

Thank you, Iris, for being so patient with me and help me when I needed.

And thank you, Rafa, for leading me through this project.

Table of Contents

Sinopsis en castellano **9**

1. Introduction **13**

1.1 Encryption Algorithms *13*

1.1.1 Stream Ciphers 14

1.1.2 Block Ciphers 15

1.2 Motivation and Proposal *15*

2. State of the Art **16**

2.1 RC4 .... *16*

2.2 Salsa20 .... *17*

2.3 ChaCha *18*

2.4 A5 .... *19*

2.5 Snow3G *20*

2.6 e-Stream *21*

2.7 Block Cipher CTR Mode *21*

3. Stream Cipher Proposal: Strounter **23**

3.1 Description *24*

3.1.1 Key Scheduling 25

3.1.2 Iteration 27

3.1.3 Output Filtering 28

3.2 Analysis *32*

3.2.1 Randomness 32

3.2.2 Performance 38

3.2.3 Security 41

3.3 Uses and Recommendations *44*

4. Conclusions 46

5. Future Work 47

6. References 48

Appendices 52

Glossary 52

Appendix A 54

Appendix B 55

Appendix C 56

Appendix D 57

Appendix E 59

Appendix F 60

Appendix G 61

Appendix H 63

Table of Figures

Figure 1 - CTR mode 22

Figure 2 - Strounter scheme 24

Figure 3 - Prefiltration 28

Figure 4 - Union representation 29

Figure 5 - Filtration scheme 30

Sinopsis en castellano

Un estudio preliminar sobre los números hizo que nos diésemos cuenta de lo importantes que son los números aleatorios, cómo estos son generados y sus usos. Por ello, se propuso realizar un generador de números pseudoaleatorios que pudiera ser utilizado en muchos contextos diferentes, incluido como cifrador en flujo. Un cifrador en flujo es, de hecho, un generador de números pseudoaleatorios en el que su salida se utiliza como secuencia cifrante combinándola, mediante la operación lógica XOR, con el mensaje en claro y, obteniendo así, un criptograma o texto cifrado.

Una vez decidido el objetivo de este estudio, se procedió a analizar los algoritmos más destacados a lo largo de los últimos años, donde destacan: RC4, Salsa20, ChaCha, A5 y Snow3G. También se estudió el modo de funcionamiento CTR de los cifradores en bloque, el cual transforma un cifrador en bloque en uno en flujo.

Terminado este estudio, se procedió al diseño del algoritmo propuesto, el cual hemos llamado Strounter. Recibe este nombre debido a su diseño: 4 contadores de 32 bits con 4 cajas de sustitución que se combinan para formar un cifrador en flujo.

En Strounter se inicializan las 4 cajas de sustitución de 8x32 bits haciendo uso del *framework* de inicialización de cajas de sustitución estudiado en (1). Una vez inicializadas estas cajas de sustitución, se usan para extraer (filtrar) 32 bits de cada caja, que son combinadas junto con una constante (distinta para cada uno de los contadores) mediante una XOR. Este diseño está orientado a evitar que los contadores sean inicializados a 0 o al mismo valor.

A continuación, describimos el proceso de generación de secuencia cifrante:

1) Uno de los contadores es incrementado (al tener 4 contadores sin signo de 32 bits, tenemos un periodo de 2^{128}) y es combinado con los otros 3 siguiendo la siguiente expresión:

$$(CTR_1 + CTR_2) \bigoplus (CTR_3 + CTR_4)$$

En cada iteración se incrementa un contador distinto.

2) Al resultado de esta combinación se le aplica la operación lógica XOR junto con un estado interno del algoritmo.

3) Partiendo del valor de este nuevo estado interno, se extrae un valor de cada caja de sustitución y, finalmente, son todos combinados mediante la operación XOR. El resultado final son 4 bytes pseudoaleatorios que forman la secuencia cifrante.

Terminado el diseño del algoritmo, se realizaron los siguientes análisis:

⇒ Aleatoriedad: se hizo uso de 2 suits de tests, TestU01 (2) y PractRand (3) y se hicieron distintas pruebas (*BigCrush* de TestU01 y analizar 128GiB de secuencia cifrante con PractRand) con distintas claves (una clave de 16 bytes aleatoria, otra de 256 bytes aleatoria, y una clave de todo 0's y otra de todo 1's) y los resultados han sido excelentes, demostrando el buen diseño del algoritmo.

⇒ Rendimiento: se ejecutó una prueba de rendimiento que midiese la fase de inicialización y otra que midiese la fase de generación. Aunque la fase de inicialización fue lenta (lo cual se acabó concluyendo que era bueno para otros fines como KDFs), la fase de generación daba unos resultados muy buenos, superiores incluso a AES con aceleración por *hardware*.

⇒ Seguridad: se analizó, de manera teórica, la seguridad asociada al algoritmo en sí. Se concluyó que el proceso de inicialización, al estar basado en un *framework* moderno que combina partes de RC4 y Spritz, aportaba un alto grado de seguridad a nuestro algoritmo, pues también involucraba procesos que están más asociados a los cifradores en bloque. Por otro lado, se estudió las repercusiones que tendría sobre Strounter efectuar un *side-channel attack* y, aunque en un principio susceptible, sería bastante más complejo ejecutar algunos ataques (como el ataque *cache trace-drive attack*) debido al tamaño reducido de las *s-box*.

Este trabajo concluye enumerando algunos usos que tendría nuestro algoritmo, como pueden ser: generador de números pseudoaleatorios, cifrador en flujo (aplicable, sobretodo, a sistemas de comunicación en tiempo real debido a su alto rendimiento) y funciones de derivación de clave (KDF), entre otros.

Se propone como futuras mejoras implementar el algoritmo en *hardware* y estudiar la mejora de rendimiento que tendría; así como investigar cómo podría optimizarse el proceso de inicialización que, si bien no es recomendable para protocolos de *forward secrecy*, sí sería interesante tener una fase del algoritmo más lenta que dificulte los ataques de fuerza bruta.

1. Introduction

Numbers. They are used every day, by everyone, in many different situations. Prices, measures, calculations, engineering, physics, computer science, and so on.

Most important ones, at least in computer science, can be random numbers. They are used in many fields within software development and cryptography. From the simplest casino-like game to the most complex cryptographic algorithm.

Focusing on the cryptographic uses, they are important for challenge values, stream ciphers, shared symmetric keys, session cookies, initialization vectors, etc. In this paper we are going to focus on the encryption algorithms, in particular, in stream ciphers.

1.1 Encryption Algorithms

There are many ways to classify encryption algorithms: by their key type (public or private), by their purpose (key sharing, general purpose data encryption or digital signature) or by their underlying design (block, stream); but we will define the classification by their key.

There can be symmetric (private key) or asymmetric (public key) algorithms. The first ones use the same key for encryption and decryption (or the decryption key is easily derivable from the encryption one). These algorithms are used for general encryption data because of their good performance. The second ones use two different keys: one for encryption and one for decryption; and one cannot be obtained from the other one without knowing a secret. These are used to share a symmetric key (as they are too much slower) and to sign/verify data.

The symmetric algorithms can be classified, attending to their underlying design, as stream ciphers and block ciphers. Block ciphers encrypt a message by splitting it into blocks of the same length and applying to each block the same algorithm with the same key. Stream ciphers generate a pseudo-random sequence of bytes and encrypt the message applying Vernam's scheme (4).

1.1.1 Stream Ciphers

Stream ciphers are the fastest encryption algorithms, as their encryption algorithm consists of applying an XOR between the algorithm output and the message.

$$c = m \bigoplus k$$

This operation is reversible, so the decryption is applying, again, the XOR between the algorithm output and the cipher text.

$$m = c \bigoplus k$$

$$m = m \bigoplus k \bigoplus k$$

Stream ciphers must generate a large sequence of random numbers (stream) which shall be used to encrypt a message. This stream should be generated very fast and, as we have mentioned, it should be impossible for an attacker to determine the sequence, since it is (or, at least, appears to be) random.

There are two kinds of stream ciphers: synchronous and self-synchronizing. Synchronous stream ciphers are those whose output does not depend on previous ciphertext. In contrast, self-synchronizing stream cipher output does depend on previous ciphertext.

1.1.2 Block Ciphers

As stated, a block cipher is defined as applying the same algorithm with the same key to each of the split blocks. This is known as ECB mode of operation and is considered insecure nowadays. To deal with this insecure operation mode, the NIST has defined other modes (5): CBC, OFB, CFB and CTR. There are other operation modes like XTS (used to encrypt hard disks), GCM and CCM (authenticated encryption modes).

CTR mode is, essentially, a stream cipher mode. It has a counter, which iterates for each block and is encrypted with the algorithm and the key. This encrypted block is XOR-ed with the plaintext block to produce the ciphertext block. So, the decryption is identical as encryption. This is considered as the most secure mode, and it allows to parallelize the encryption/decryption process.

1.2 Motivation and Proposal

As we have seen in the previous section, random numbers are quite relevant, as well as stream ciphers. That is why we have decided to focus our research on this topic, specially on pseudo-random number generators and their relationship with stream ciphers (which are, essentially, fast and secure pseudo-random number generators). After doing this research, we will design a stream cipher that will meet current requirements.

These requirements are, as we will see, performance and security (randomness, avoidance of cycles, unpredictability, etc.).

2. State of the Art

In the past years, many stream ciphers have been proposed and some of them are still considered secure and used in many applications. In this chapter we will analyze those that are most important and in widespread use, even those that are no longer in use but were important in the past. We will also include the most relevant algorithms from the e-Stream project.

2.1 RC4

RC4 is a stream cipher created by Ron Rivest in 1987 and, until 1994, it was proprietary. It uses an s-box 8x8 bits balanced by columns (it takes all values from 0 to 255, permuted) (6).

RC4 is composed of 2 algorithms (7): the first one (KSA: Key-Scheduling Algorithm) is intended to initialize the s-box by permuting its values, while the second one (PRGA: Pseudo-Random Generation Algorithm) is intended to generate the keystream (that means that RC4 is a synchronous stream cipher algorithm), which will be used to be XOR-ed with the plaintext in order to perform encryption, or with the ciphertext to decrypt. So, the encryption algorithm and the decryption one is the same (as per Vernam's scheme).

RC4 accepts a wide range of key sizes: from 8 to 2048 bits length (8) and its period depends on the key (although it is difficult to calculate it).

This algorithm is used in WEP, WPA, SSL/TSL (but it was prohibited in RFC 7465 (9) as some biases of the algorithm could lead to the decryption of session cookies and possible session hijacking, only requiring *some* ciphertexts of the same plaintext encrypted with different keys) and is optional in many other protocols (SSH, RDP, Kerberos, etc.).

2.2 Salsa20

Salsa20 (10) is a stream cipher designed by Daniel J. Bernstein and submitted in 2005 to the eStream (*ECRYPT Stream Cipher Project*).

It's a synchronous stream cipher that generates its output as 64-byte blocks derived from the key, the *nonce* (value used once per message) and block number; so Salsa20 allows obtaining an output block for any position independently from any previously generated blocks.

Salsa20 does not use any s-box and does not preprocess any of its inputs, so it calculates an output block applying directly the algorithm to the key and nonce input values.

It was designed to be used with a 32-byte key (256 bits) and 20 rounds. But it can be used with its 8 and 12 rounds versions and with a smaller key. The author of Salsa20 does not recommend using a smaller key (16-byte key) since the algorithm duplicates it to transform it into a 32-byte key.

The algorithm itself is based on addition, XOR and rotations of constant-distance operations, all these over a 4x4 matrix of 32-bit word elements (key, nonce, block counter and constant words). The output of this matrix after applying these operations for *n* rounds is a 16-word (64 bytes) stream used to encrypt (XOR) the plaintext.

The choice of these operations is based on a performance decision taken by the author, as he wants it to be fast regardless of the architecture it is implemented on. That's why the operations over the data are specified in little-endian order, as big-endian access may take more time.

Salsa20 is used in many different applications (11), such as: DNSCurve, DNSCrypt, Shadowsocks (a socks5 proxy), Chromium OS, Linux Kernel, KeePass, Viber (chat application), and many more.

There are some satisfactory attacks to Salsa20/5-8 rounds (12) (13), but none of these attacks affects the recommended version of Salsa20/20 with 32-byte key. Salsa20/5-7 is broken by differential cryptanalysis, and Salsa20/8 by Probabilistic Neutral Bit (PNB).

2.3 ChaCha

ChaCha (14) is a synchronous stream cipher from the same author as Salsa20, Daniel J. Bernstein.

As it stands, ChaCha is based on Salsa20 and was designed to improve it. Just like Salsa20, ChaCha needs a 256-bit key and has 3 modes of operation: ChaCha8, ChaCha12 and ChaCha20; where the number indicates how many rounds it performs.

These improvements affect how the matrix is generated. The new way to generate the matrix improves speed on SIMD (*Single Instruction, Multiple Data*) platforms and also diffusion.

Another improvement was made on the quarter-round, updating each word twice instead of once. Also, this makes ChaCha about 50% faster than Salsa20.

All those improvements make ChaCha safer than Salsa20 with similar performance results. As ChaCha is based on Salsa20, the same attacks can be performed against ChaCha. But the introduced improvements make ChaCha more resilient and reduce in 1 round the effectiveness of these attacks. That is, the known attacks on Salsa20 affect the operation modes from 5 to 8 rounds. But those attacks affect ChaCha only from 4 to 7-rounds modes.

ChaCha is used in some of the following systems (15): QUIC, TLS, OpenBSD, OpenSSH, LibreSSL, Android, NetBSD, Linux Kernel, KeePass, and many others.

2.4 A5

GSM was the first digital mobile communication system implanted worldwide. It allowed people to communicate through voice calls and send SMSs with good quality. To guarantee calls confidentiality, it uses the A5 stream cipher algorithm.

A5 has some variants: A5/0, A5/1 and A5/2 (16). A5/1 is the strongest one and used in Europe and America. Because of restrictions on exporting cryptography, A5/0 (the weakest) is used in third world countries and those sanctioned by the UN. A5/2 is used in Asia.

A5/1 uses 3 LFSRs (17) to produce a 64-bit key (LFSRs of 19, 22 and 23 bits each) and its output is XOR-ed with the plaintext. This algorithm is hardware-implemented in most cellphones to allow real time encryption and decryption.

On the other hand, A5/0 (18) does not use any pseudo-random number generator. It uses as output stream, the same input, negated. So, this algorithm actually does not provide any confidentiality.

There are few attacks to A5. For example, it was discovered (19) that in the A5/1 version, the 10 least significant bits of the key were zero. This reduces the brute-force attack complexity from 2^{64} to 2^{54}.

Other attacks allow to break A5/1 in 2^{40} steps. But, in the year 2000 an attack was published (20), allowing an attacker to break A5/1 in real time and to listen to the whole conversation between two people.

Furthermore, the security of this algorithm is by obscurity (companies have to pay to have access to the algorithm specification to implement it). This concept (*security by obscurity*) contrasts with Kerchoff's principle: "*A cryptosystem should be secure even if everything about the system, except the key, is public knowledge*".

All these things lead to the need of a new algorithm for the next generation of mobile communication (3G).

2.5 Snow3G

Snow3G (21) (22) is the stream cipher used in the third generation of mobile communications (the second of digital one).

Snow3G uses two s-boxes (one of them is the Rijndael s-box (23)) and an LFSR of 32 cells (its feedback function is a primitive polynomial). Each of these cells has 32 bits. That means that for every clock, it outputs 32 bits as keystream.

It also uses a Finite State Machine (FSM), which is used to initialize the LFSR (with a 128-bit initialization vector). This FSM has 3 registers (R1, R2 and R3) of 32 bits each, and R2 and R3 are updated through the s-boxes. These registers are used within the FSM to get the value F, used as IV to the LFSR.

There are no known successful attacks against this algorithm, but there is an attack (24) to its variant $Snow3G^{\oplus}$.

2.6 e-Stream

e-Stream (25) was a project running from 2004 to 2008 and its purpose was to get a suite of stream ciphers for general purpose. It had two profiles of stream ciphers: software and hardware profiles. The first group was focused on performance, while the second group focused on power requirements as well as storage capacity.

Some of the algorithms of the e-Stream portfolio have not been popular, while others, like Salsa20 (described in 2.2) and HC-128 have been used in many different applications and have served as the base for future algorithms. Both of these algorithms belong to the software profile.

This portfolio is periodically revised and updated.

2.7 Block Cipher CTR Mode

Stream ciphers are fast and produce pseudo-random numbers, as well as approximate Shannon's perfect cryptosystem. That is why an operation mode for block cipher algorithms was designed to operate as a stream cipher (actually, CFB and OFB modes operate as stream ciphers too, but they are not as common as the CTR mode). We have mentioned this in 1.1.2, and now we are going to describe it in more detail.

CTR mode (5), or counter mode, is an operation mode that employs a block cipher algorithm (*f*) to produce a keystream that will be XOR-ed with the plaintext to encrypt it, or with the ciphertext to decrypt it.

It takes a counter as plaintext and, using *f*, encrypts it with the input key and generates a block. This block will be used as keystream in a Vernam scheme. In order to generate the next block, the counter is increased and the process repeated.

In the next figure we summarize this process:

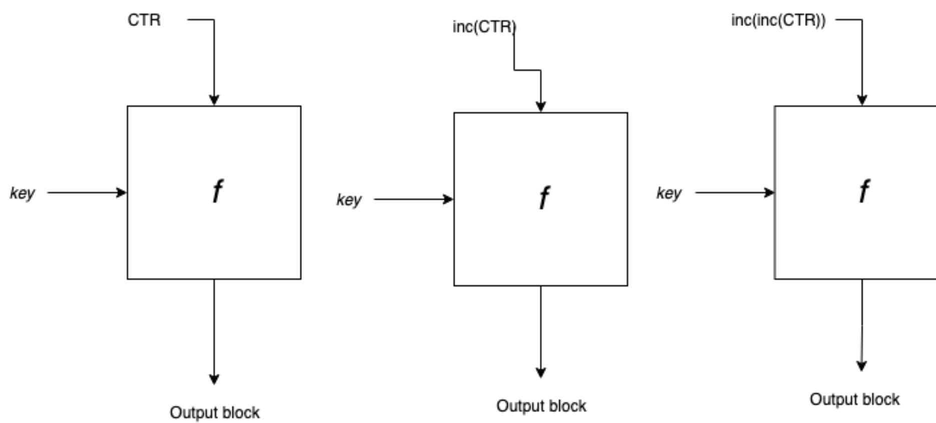

Figure 1 - CTR mode

As specified by NIST, the counter is b bits length (for AES this is b = 128), choosing m bits to be incremented and the remaining bits ($b - m$) kept constant. The standard increment function is $x + 1 \ mod \ 2^m$.

3. Stream Cipher Proposal: Strounter

In this section we are going to describe our proposed stream cipher, which we called Strounter, designed to satisfy all requirements established in section 1.

In section 3.1 Strounter's structure and initialization process will be described, including its key scheduling, iteration and output filtering. Since the iteration is relatively simple (as we will see), we are focusing on the key scheduling and output filtering.

Next, on section 3.2, we will discuss its randomness analyzing its output with some famous randomness test suites, such as TestU01 (2) and Practically Random (PractRand) (3). We will see how, even with an all zeroes key, it is able to produce pseudo-random bytes sequences that can pass all tests. Along with that, we will see some ways that our algorithm can be broken and how secure is its initialization process.

Also, we will analyze its performance (as this is a key feature of this kind of algorithms); measuring, independently, its key scheduling and encryption times. These results will be compared with other algorithms and see if ours is fast enough or has a long way ahead to improve.

After that, we will analyze its security attending to the key setup and side-channel attacks.

And, finally, we are going to see some uses of our proposed algorithm as well as some recommendations of use.

3.1 Description

In this section we are going to describe the structure of our stream cipher and how its components interact to produce our fast and random cipher.

As the name "Strounter" indicates, is a "stream cipher composed by counters". It has 4 counters of 32 bits length each one. Also, it uses 4 s-boxes (substitution box where an input is used to index that box and produce its value as output) 8x32 bits to filter the output of these counters and an internal state to enhance the randomness. It produces 4 bytes per iteration.

Its working process is divided into two phases: in the first one, the 4 counters are initialized as well as the 4 s-boxes are filled (this initial process is key-dependent); in the second one, it produces keystream through filtering the iteration values.

In the next picture we summarize the whole process of generating the keystream. It will be explained in more depth in the following sections.

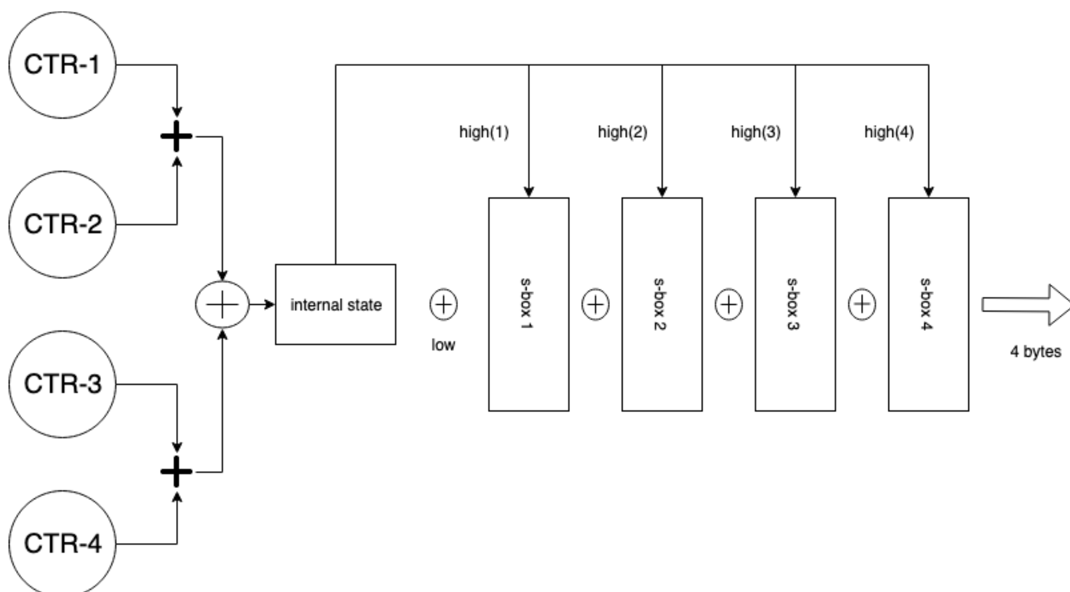

Figure 2 - Strounter scheme

3.1.1 Key Scheduling

The key has a minimum length of 16 bytes, and a maximum of 256 bytes.

After checking that the key is in this range, it replicates the key to fill that 256 bytes of maximum length, just as RC4 does.

After that, we create the motor that will be used to filter the output of our stream cipher. This motor is created as follows:

1) An auxiliary 8x8 s-box is created and initialized, setting each *s-box[i]* element to i. This way, looking at it by columns, it is balanced (same number of 0's and 1's per column). An internal state, *is,* is created and initialized to 0. This state, *is,* will never be reinitialized to 0 again.

2) For each 8x32 s-box, the following process is repeated: swap all positions of the auxiliary 8x8 s-box according to the internal state *is* and the key. The result of this auxiliary s-box is assigned to the first byte of each element of the 8x32 s-box. This is repeated for the 4 bytes of each registry of the main s-box. This way, if we split the 8x32 s-box into 4 independent 8x8 s-boxes, they will be still balanced.

After this initialization process of the s-boxes, there is only one thing to do: initialize the counters. These initial values should be key-dependent to avoid an attacker can discover them.

As many other algorithms do (specially, *hash* algorithms), we use 4 constants (one per counter) that are combined with the key through an XOR to initialize the counters. This is done to avoid the counters being initialized to the same value (it is still possible, but with a very low probability), and to avoid all counters to be initialized to 0 (as before, it is still possible, but with the same low probability).

These are the constants chosen for each counter:

```
#define COUNTER_A 0x12B9B0A1
```

For the first counter, the constant is the value 314,159,265. Which corresponds to the integer part of $\pi \times 10^8$.

```
#define COUNTER_B 0x1033C4D6
```

For the second counter, the constant is the value 271,828,182. Corresponding to the integer part of $e \times 10^8$.

```
#define COUNTER_C 0x277E949C
```

For the third counter, the constant is: 662,607,004. That is the is the integer part of $h \times 10^{43}$, being h the Planck constant (26).

```
#define COUNTER_D 0x11DE784A
```

For the fourth counter, the constant is 299,792,458; or the speed of light in vacuum in meters per second (27).

Once we have defined the constants, we take 4 bytes of the key (the first 4 for the first counter, the next 4 for the next counter, etc.) and filter them (the filtering process will be explained in 3.1.3). Its output is XOR-ed with that counter's constant.

When all 4 counters are initialized, the key scheduling process is done.

3.1.2 Iteration

In this section we will describe how the counters are iterated as well as establish Strounter's period.

As mentioned previously, our stream cipher is composed by 4 32-bit long counters. In total, they form a 128-bit counter. For each iteration (in stream ciphers, there are as many iterations as keystream is needed) one of the counters is increased by iteration. Along with that only counter, an auxiliary counter is increased to know which one should be increased per iteration:

```
void Iterate(motor m)
{
 m->P.counters[m->P.which%NUM_COUNTERS]++;
 m->P.which++;
}
```

This way, concatenating all of them, we have a 128-bit counter, that is, a period of 2^{128}. This is a great period and, if we consider the additional period that the filter gives us potentially (this will be discussed later), it will be substantially increased further.

But, as one can deduce, 4 counters do not have enough randomness to be used neither as a pseudo-random number generator nor a stream cipher. The only prerequisite they accomplish is the performance one, as there is probably no generator that is faster than a counter.

That is why, in the next section, we are going to describe how these counters output is filtered and how this improves the security of Strounter.

3.1.3 Output Filtering

As we have seen in previous sections, Strounter already has a large period and great performance (analyzed more in depth in 3.2.2), but practically no randomness; thus, a filter is required. Here we will describe how it works, how it is applied to Strounter, and how it enhances it. So, first, we are going to focus on the prefiltration stage (as shown in Figure 3):

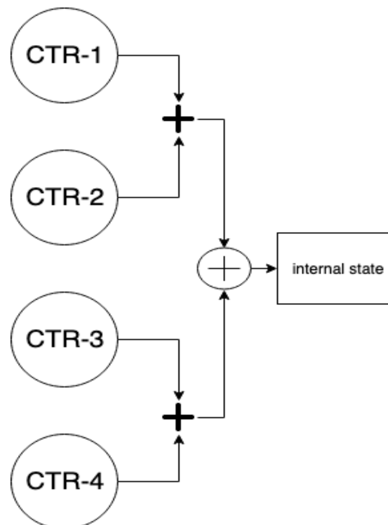

Figure 3 - Prefiltration

The filter used here has an internal state and is represented by the following structure in the C programming language:

```
union
{
        struct
        {
          uint32_t high;
          uint32_t low;
        } half;
        uint64_t all;
} filter;
```

What this structure represents is the internal state called "all". As we can see, it is a 64-bit unsigned integer. What the *union* (28) type does is make "all" accessible from 2 variables ("high" and "low"). Each of these variables are 32-bit unsigned integers and through them we can access "all" as shown in Figure 4:

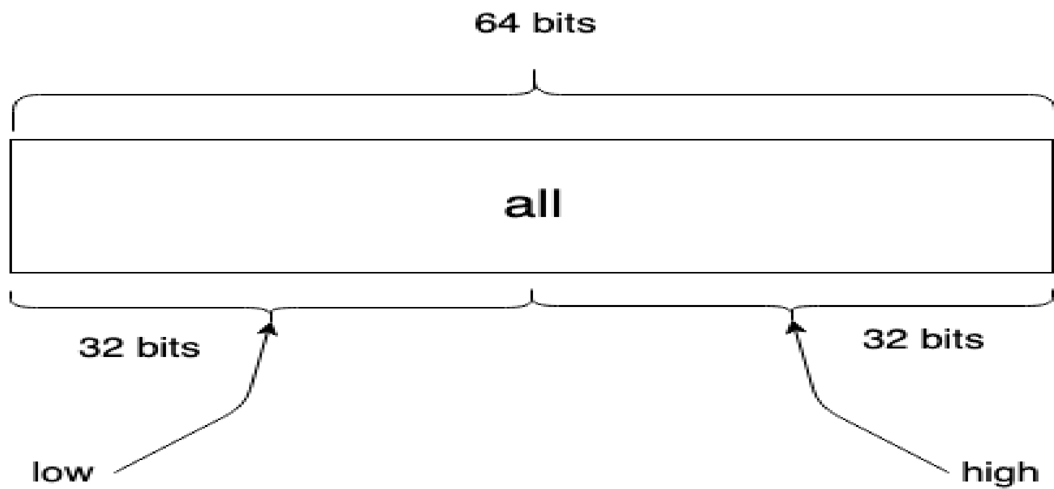

Figure 4 - Union representation

Where "low" and "high" point (first 32 bits or last ones) depends on the platform.

Before using the s-boxes to filter the counters' output, they are prefiltered with the internal state. But, before this happens, they must pass from 4 32-bit counters (128 bits in total) to a smaller value that can be used with our internal state: they must be combined. To combine them we had to choose a function that does not omit any value, does not introduce any cycles and is considerably fast.

The resulting function was shown in Figure 3:

$$(CTR_1 + CTR_2) \bigoplus (CTR_3 + CTR_4)$$

This function returns a 32-bit value that will be added to the “high” part of our internal state. This “high” part is initialized to 0 on the “key scheduling” phase and will never be reset to 0 again: it will keep the value taken from the previous iteration for the next one.

Next, we will describe how the internal state interacts with the s-boxes to produce the filtered output.

So, now, we are going to focus on the next part of the algorithm, the filtration scheme as shown in Figure 5:

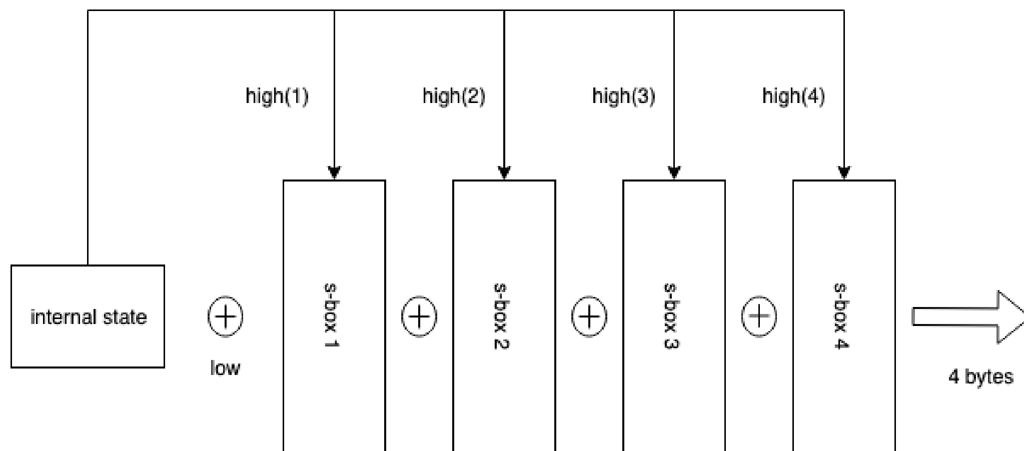

Figure 5 - Filtration scheme

Once the internal state has been updated for the given iteration, it proceeds to filter the output. It is important to remember that each s-box is an 8x32 one.

As the “high” part of the internal state is 32 bits (4 bytes), we take each of its bytes to index each of the s-boxes: we take the value from the first s-box according to the position given by the first byte of the “high” part; the second value, taken from the second s-box, is taken indexing the s-box by the second byte of the “high” part; and the process is repeated for the two remaining s-boxes.

Once all these values are taken, they are XOR-ed together with the "low" part of the internal state (which is 32 bits as well). The resulting value of this process is a 4-byte output that will be used as keystream (or as a pseudo-random number for any given purpose).

Before finishing each iteration, a rotation is done as follows:

```
m->filter.all = ((m->filter.all)<<31)|((m->filter.all)>>33);
```

Ideally, all possible states of the s-boxes (2^{128}) can be applied for each of the states of the counters (2^{128}). So, the maximum period is: $2^{128} \times 2^{128} = 2^{256}$. This is more than enough for current applications.

3.2 Analysis

In this section we are going to analyze the results of our stream cipher attending to its randomness, performance and security. Finally, we will discuss these results comparing them to other stream cipher algorithms.

For the performance analysis, we have used a Virtual Machine of Windows 10 x64 configured with 4GB RAM and 4 virtual processors. This virtual machine was run on a MacBook Pro with 8GB RAM and an intel-i7 4750HQ running macOS Mojave (version 10.14.5).

3.2.1 Randomness

Before starting with the randomness analysis, we will focus first on which tests we have run and why we have chosen them.

As mentioned before, we will use PractRand (3) and TestU01 (2).

TestU01 is an old suite of tests that has the scientific community approval regarding its completeness and test depth. Since it is a bit old, it is not very efficient, but it is very thorough since if it discovers some undesirable properties in the sequence, it can add more tests dynamically to perform more in-depth testing for that section of the sequence. The way TestU01 is used requires implementing a program that calls its tests with the output of a given pseudo-random number generator. This is required since TestU01 is, actually, a library written in C, and not a program itself.

Although TestU01 has many different test suites, those most commonly used are the "crush" tests: Small Crush, Medium Crush and Big Crush, since they are the most complete and thorough ones. From these tests, we are going to use "BigCrush" to test Strounter's randomness, as it is most complete one and a super-set of the others (while SmallCrush is executed for less than a minute and MediumCrush for less than an hour, BigCrush takes about 8 hours to execute).

By the other hand, PractRand is comparatively newer and it is not as widespread as TestU01. Nevertheless, it has many other features that makes it better in some respects. Firstly, PractRand is not a library the user should use and call from a separate program, but a complete program that can read the pseudo-random number generator's output from many different sources (e.g., files and standard output). Also, for every power of 2 (starting at 128MB) it prints if, so far, it has found a sequence that is not random and how many tests the subject PRNG has passed. Furthermore, PractRand allows its execution in n threads, so it can test far faster than TestU01.

PractRand classifies the results in the next states:

1. normal
2. normalish
3. unusual
4. mildly suspicious
5. suspicious
6. very suspicious
7. VERY SUSPICIOUS
8. FAIL
9. FAIL !
10. FAIL !!
11. FAIL !!!

12. FAIL !!!!
13. FAIL !!!!!
14. FAIL !!!!!!
15. FAIL !!!!!!!
16. FAIL !!!!!!!!

Note: It must be remarked that anything not a "FAIL" is a "PASS"; and any "PASS" result different from "normal" means that the test was passed on the outer edge of the valid interval.

The keys used to test Strounter's randomness have been:

⇒ 16-byte random key. Remember that our algorithm replicates it as many times as necessary to fill a 256-byte key, so this test will see if Strounter is good enough to produce pseudo-random numbers with the shortest key it accepts. In addition, testing with this length of key is in particularly interesting as 16 bytes is the most common key-length used in symmetric encryption algorithms.
This key is (in hexadecimal notation):
`29392d49747d4d5f40392b242821373b`

⇒ 256-byte random key. This way we will test how it works with a "perfect key": one that is random in all its length (without needing to replicate it).
This key is (in hexadecimal notation):

```
403e2e322a7d4826354f3c24313358396a256a4728772c67306c3e776
63e764c5e594f71513e3f3e63762c23553b5928643c3269683b24654a
374a6370786b6c43713a70584d75762133692a5d324b6d34466650585
04c6a2d3e702873795d2a592c374c2d3238432328227d6e4151625272
6971744a556f515c3436285340795a556422212177493377333750685
b437d74436b6f4c40733f565b7968783d685f40714d4d35704b3a2c36
6f593f37263d785d317b26407b2f59765d575a2976586e38244b59625
539484d253362216d577d4052313c47763f3b5e3653263745646731 5f
4b664b4b316a30587269556b6e695d6b712874585a3764586d372a526
63e
```

⇒ All 0's. This test will let us see if Strounter can generate pseudo-random numbers even if a user is using the worst key possible. This will only affect how the s-boxes are initialized, as counters' initial state won't be 0 as they are XOR-ed with a constant different from 0. Furthermore, it would not be 0 anyway as it is also initialized using the s-boxes and the probability to extract 4 zeros (one from each s-box) is 0, at least with this key.

⇒ All 1's. Same test as the before one, but with the inverse key.

For each of these keys, we are going to do the next tests:

⇒ Generate 128GiB of keystream and test them with PractRand.

⇒ BigCrush.

The results of these tests are specified in the appendices, as shown in Table 1:

Table 1 - Appendices classification

	PractRand	**BigCrush**
16-byte key	Appendix A	Appendix B
256-byte key	Appendix C	Appendix D
All 0's	Appendix E	Appendix F
All 1's	Appendix G	Appendix H

NOTE: TestU01 reports are very long, so we only include the final result: whether the tests were passed or not. If some report shows that a test might have failed (or any result different from "normal"), the result of that test will be put too.

Before we start analyzing the reports and see whether Strounter is as random as it should be, we must first clarify that there is an *alpha* value that says with which probability we can get a "fake positive" or a "fake negative". So, according to this, we can foresee that some of the failed tests might be, actually, passed, and some of the passed might be failed.

Let’s start first analyzing **PractRand’s results**.

As we can see in Appendices E and G, with keys like all 0’s and all 1’s the results are pretty good. We have 2 “unusual” results, one in each test, but according to PractRand’s documentation it is still a passed test. The only thing we should care about is that, in Appendix G (all 1’s), we get a “mildly suspicious” result in 16GB of output analyzed. But, as we can see, in the next tests until 128GB this message is not repeated again, so we can ignore it. Also, remember that in PractRand anything but a “FAIL” is a “PASS”.

If we use a real key (like the 16-byte and 256-byte key), a random one that users shall use, the results are perfect, as we can see in appendices A (here we can see an “unusual”, but this is insignificant as it is not repeated in following tests) and C.

Now, let’s analyze **TestU01’s results**.

Here we can see that there is only one test failed (see appendix F) with the worst key: all 0’s. But, seeing the other tests for the same key and the reports from the other keys, we might guess that this failed test it’s maybe a fake positive. Even if it is a real failed test, it is insignificant as it is only 1 failed out of 160 tests and, it has failed with the worst key that a user can use.

In addition, we can find that, in appendix D (random key of 256 bytes length) we can find 1 failed test. The range of the *p*-value of a test to get a “pass” is [0.001, 0.9990] and, as we can see, we obtained a 0.9991, so this is not statistically significant. After seeing the other results, we may consider that it is not about the algorithm itself, but the chosen key which may affect the results.

After analyzing all these reports, taken from different test suites and different keys, we can say that our proposed stream cipher is random enough to meet current requirements.

3.2.2 Performance

In this section we are going to test Strounter's performance and compare it to AES and 5 other stream cipher algorithms such as HC-128, RC4, Salsa20, Snow3G and Spritz.

The performance comparison will be done at key scheduling and encryption stages. Separating these two stages is fundamental, as one can have a very slow initialization process but be very fast in the encryption one, and vice versa.

This analysis was made as follows:

⇒ Key scheduling level: we have initialized the algorithm 100.000 times and measured how much time it took. After that, it was divided by the same number and obtained the mean time for this process. This has to be done like this because an initialization process might be so fast that measuring it only once may lead to an incorrect result if the processor cannot measure it accurately.

⇒ Encryption level: we have encrypted 128MiB and measured how much time it took. After that, we have calculated how many MiB are encrypted per second dividing the amount encrypted (128MiB) by the time spent.

Here are the results:

Table 2 - Performance results

	Key Scheduling (ns)	Encryption (MiB/s)
AES	2177	59.71
AES (Hardware Accelerated)	420	329.01
RC4	4534	227.14
Spritz	725	51.44
HC-128	34767	148.98
Salsa20	0	160.46
Snow3G	1012	106.84
Strounter	2032379420	337.76

Let's start first comparing the **key scheduling** times.

Salsa20 has one of the fastest key scheduling processes, it is so simple and fast, that cannot be measured, and we got 0ns for it. If we see section 2.2, it really does not need to initialize anything, as it builds a new state for each iteration.

Comparing with the other algorithms, Strounter is, by far, the one with the slowest initialization process. This is due to the 4 8x32 s-boxes it uses. The implementation of how these are initialized involves 16 8x8bit s-boxes, which are combined to form our 4 8x32 s-boxes. This makes it very slow but, as this process is made only once, it is not very significant. Another reason why this is slow is because of the big number of RAM accesses required to initialize the s-boxes.

In contrast, this slow process is due to a safer way to initialize the s-boxes and the algorithm itself, what makes it more secure as it is more elaborated. This initialization process is analyzed in (1), where it is shown that key-derived s-boxes are, in security terms, equivalent to random s-boxes.

Focusing on the **encryption performance**, we see how Strounter is the fastest one with 337,76MiB/s. The only algorithm that can compete with ours is AES but executed with hardware acceleration.

This performance is due to the fact that Strounter has one of the simplest encryption algorithms: increase a counter and extract 4 bytes from 4 s-boxes. This means that, for each iteration it does, it produces 4 times more keystream than other algorithms such as RC4.

Even without hardware acceleration, it gets a slightly better performance than AES with hardware acceleration.

3.2.3 Security

In this section we are going to discuss how secure our algorithm is and what kind of attacks could disrupt its security.

3.2.3.1 Key Setup

As mentioned in section 3.2.2, we have based our s-boxes initialization process on the framework designed in (1). This research shows how combining concepts from RC4 and Spritz, it can create a way to initialize s-boxes being key-dependent and perform like random s-boxes, attending to their security and balanced s-boxes (even splitting them into 8x8 s-boxes and analyzing them independently, a requirement that Blowfish (29) does not meet).

This framework covers also the avalanche concept. This means that, even if a user employs two very similar keys (or two different users using similar keys), the resulting s-boxes would be significantly different.

In addition, introducing block cipher level complexity (key derivation, rounds, initialization process) to a stream cipher, that usually do not have any of these, should further improve Strounter's security.

To sum up, the key scheduling is strong enough to avoid any attacks and attempts to extract sensitive information from the s-boxes state.

3.2.3.2 Side-Channel Attacks

A side-channel attack is an attack done with information relative to the computer system instead of the algorithm itself. There are many different side-channel attacks, such as: cache attacks (timing-driven attacks, access-driven attack and trace-driven attack) (30), power-monitoring attacks, acoustic attacks, and so on. In this section we describe which side-channel attacks can break our algorithm and how they can get the key to decrypt a ciphertext.

The first one is the cache-timing attack. This attack consists on measuring how much time an operation of the algorithm consumes and infer, going backwards, what the input key was, attempting to recover it. This attack has successfully cryptanalyzed AES in software and recovered the key as described in (31).

This is done by having a thread running on the same CPU and, measuring its own access-time to cache, determining (after many executions, since they are based on statistical interferences) what the key is and decrypt the ciphertext. To successfully execute this attack, an attacker does not need any known plaintext or ciphertext. As shown in (32), the bigger the tables used are, the more efficient these attacks can be performed.

Another way to attack Strounter is through a trace-driven attack. This attack is based on monitoring the cache to see if it gets a "miss" or a "hit". A "miss" is obtained where the needed data is not in cache and has to look it up on RAM. A "hit" is obtained where the needed data is in cache and does not need to look it up. These misses or hits can reveal information about the input used in an algorithm, that is, the key.

These attacks, if performed while the initialization process of our algorithm, the key can be obtained, as the s-boxes accesses are key-dependent. But if these attacks are done during the generation of the keystream, the counter values can be inferred as s-boxes accesses depends on them, not on the key anymore. If the counter values are obtained, along with the s-boxes, an attacker can obtain the keystream (not the key) and decrypt the message. Although any algorithm that uses an s-box can be susceptible to attack, Strounter uses 4 8x32 bits s-boxes and therefore this attack cannot be performed against it as that is only 4KiB, which is small enough to be held in the processor cache.

Although these attacks can be performed, they are more theorical than practical, as they need an evil process running on the victim's machine and, if an attacker can execute a program in someone else's machine, it would be more efficient to install a keylogger or any other malware that could potentially provide more benefit.

3.3 Uses and Recommendations

After analyzing Strounter's performance and security, let us now describe potential applications for Strounter.

As an obvious choice, Strounter could be good as a pseudo-random number generator. As it has been demonstrated in section 3.2.1, even with some of the worsts seeds (keys) it can generate pseudo-random number sequences that pass the most stringent test suites.

This leads us to the next application of Strounter: stream cipher. As explained on previous sections, a stream cipher is quite the same as a pseudo-random number generator, but its keystream (production) is used, applying the XOR operation, to encrypt or decrypt a message (Vernam's scheme).

Taking the advantage of Strounter's performance, it could be use in real-time communications, like phone calls or video chats, where encryption is mandatory but should be as fast as possible and not add any delay on the communication. According to our results in 3.2.2, Strounter can be a perfect candidate as encryption algorithm for future real-time communication systems.

Another use we can attribute to our algorithm is operating as a KDF. A KDF (Key Derivation Function) is a function that, given a key of a given length (e.g., 32 bytes), it iterates as many times as necessary to derive a key or a set of keys (for example, it would derive 15 keys of 32 bytes) that can be used in other purposes (e.g., in a chat application). This is done taking the 32 bytes of key provided by the user and using it as input key (seed) in Strounter and generating 480 bytes of keystream. After that, that keystream is split into chunks of 32 bytes, resulting in the 15 keys the user needed. And, after testing its key scheduling performance, it can be a good KDF as its initialization process is slower enough to be defended against brute-force attacks.

It can even be used as PBKDF (Password-Based Key Derivation Function) just like *bcrypt*, a PBKDF based upon *blowfish*, a block cipher algorithm with s-boxes and a very slow initialization process.

4. Conclusions

After the initial research regarding how modern stream ciphers are constructed and what key features they must have, we have designed an algorithm that introduces additional security in the initialization process and is very fast encrypting and producing pseudo-random numbers. Its excellent randomness and performance characteristics make the proposed algorithm a candidate to compete with commonly used algorithms.

This was achieved by designing an algorithm that uses just 4 counters and 4 s-boxes, making it really simple and easy to understand regarding cryptanalysis. This design was driven by the requirements mentioned in the first section: efficiency, randomness and security; therefore allowing any application requiring a fast and secure algorithm: general purpose encryption, real-time communications, KDFs (its slow initialization process is a plus with this application), pseudo-random number generators, and so on. The randomness results were really good even with potentially problematic seeds, as well as the performance achieved.

Finally, according to our initial cryptanalysis, it can be broken only by side-channel attacks, which are not algorithm-dependent.

5. Future Work

Algorithms, unlike protocols, usually do not need to be updated. But, as we have seen in section 3.2.2 (performance analysis), it has a very slow process of initialization, and this could be the first thing to work on. Trying to reduce this time may involve reimplementing the whole initialization code or redesigning the way the s-boxes are constructed. Anyway, this initialization process should be improved especially in a communication system that uses *forward secrecy* (encryption of each message with a different key).

Another interesting improvement would be to implement it in hardware and analyze its performance characteristics in comparison to AES. If software implementation of Strounter is as fast as AES accelerated by hardware, it appears only logical that we could achieve excellent results if we implement Strounter in hardware. Furthermore, this could help reduce initialization times as well.

6. References

1. *Randomness Analysis and Generation of Key-Derived S-Boxes.* **Álvarez, Rafael and Zamora, Antonio.** 1, s.l. : Login Journal of the IGPL, 2015, Vol. 24.

2. **L'Ecuyer, Pierre.** TestU01. [Online] http://simul.iro.umontreal.ca/testu01/tu01.html.

3. **Ouz.** Practically Random. [Online] April 9, 2004. https://sourceforge.net/projects/pracrand/.

4. **Christensen, Chris.** Stream Ciphers. [Online] https://www.nku.edu/~christensen/Stream%20ciphers.pdf.

5. **Dworkin, Morris.** Recommendation for Block Cipher Modes of Operation. *NIST Publications.* [Online] December 2001. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.

6. **Schneier, Bruce.** *Applied Cryptography.* s.l. : John Wiley & Sons, 1996.

7. **Rise, Ralph (Eddie), Cho, Suk-Hyun and Kaylor, Devin.** RC4 Encryption. [Online] https://sites.math.washington.edu/~nichifor/310_2008_Spring/Pres_RC4%20Encryption.pdf.

8. **Wash, Rick.** Lecture Notes on Stream Ciphers and RC4. [Online] https://www.rickwash.com/papers/stream.pdf.

9. **Popov, Andrei.** Prohibiting RC4 Cipher Suites. *Internet Engineering Task Force (IETF).* [Online] February 2015. https://tools.ietf.org/html/rfc7465.

10. **Bernstein, Daniel J.** The Salsa20 family of stream ciphers. [Online] 12 25, 2007. https://cr.yp.to/snuffle/salsafamily-20071225.pdf.

11. **IANIX.** Salsa20 Usage & Deployment. *IANIX.* [Online] March 8, 2019. https://ianix.com/pub/salsa20-deployment.html.

12. **Crowley, Paul.** Truncated differencial cryptanalysis of five rounds of Salsa20. [Online] October 17, 2005. http://www.ciphergoth.org/crypto/salsa20/salsa20-cryptanalysis.pdf.

13. **Maitra, Subhamoy, Paul, Goutam and Meier, Willi.** Salsa20 Cryptoanalysis: New Moves and Revisiting Old Styles. [Online] 2015. https://eprint.iacr.org/2015/217.pdf.

14. **Bernstein, Daniel J.** ChaCha, a variant of Salsa20. [Online] January 28, 2008. https://cr.yp.to/chacha/chacha-20080128.pdf.

15. **IANIX.** ChaCha Usage & Deployment. *IANIX.* [Online] March 17, 2019. https://ianix.com/pub/chacha-deployment.html.

16. **Srinivas, Suraj.** The GSM Standard (An overview of its security). *SANS Institute.* [Online] https://www.sans.org/reading-room/whitepapers/telephone/gsm-standard-an-overview-security-317.

17. **Stockinger, Thomas.** GSM network and its privacy - the A5 stream cipher. [Online] November 2005. http://www.nop.at/gsm_a5/GSM_A5.pdf.

18. **Jensen, Oliver Damsgaard and Andersen, Kristoffer Alvern.** A5 Encryption In GSM. [Online] June 2017. https://koclab.cs.ucsb.edu/teaching/cren/project/2017/jensen+andersen.pdf.

19. **Briceno, Marc, Goldberg, Ian and Wagner, David.** A pedagogical implementation of A5/1. [Online] http://scard.org/gsm/a51.html.

20. **Biryukov, Alex, Shamir, Adi and Wagner, David.** Real Time Cryptanalysis of A5/1 on a PC. *CRYPTOME.* [Online] https://cryptome.org/a51-bsw.htm.

21. **Orhanou, Ghizlane, El Hajji, Said and Bentaleb, Youssef.** SNOW 3G Stream Cipher Operation and Complexity Study. [Online] March 2010. http://m-hikari.com/ces/ces2010/ces1-4-2010/orhanouCES1-4-2010.pdf.

22. **GSMA.** Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G Specifitacion. [Online] September 6, 2006. https://www.gsma.com/aboutus/wp-content/uploads/2014/12/snow3gspec.pdf.

23. **Distributed Wikipedia.** Rijndael S-box. [Online] May 2017. https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Rijndael_S-box.html.

24. **Biryukov, Alex, Priemuth-Schmid, Deike and Zhang, Bin.** Differencial Resynchronization Attacks on Reduced Round SNOW 3G. [Online] http://orbilu.uni.lu/bitstream/10993/17071/1/analysis_of_snow3g-xor-resynchronization.pdf.

25. **ECRYPT.** eSTREAM: the ECRYPT Stream Cipher Project. [Online] March 2012. http://www.ecrypt.eu.org/stream/.

26. **NIST.** The NIST Reference on Constants, Units, and Uncertainty. [Online] https://physics.nist.gov/cgi-bin/cuu/Value?h.

27. —. The NIST Reference on Constants, Units, and Uncertainty. [Online] https://physics.nist.gov/cgi-bin/cuu/Value?c|search_for=universal_in!.

28. **Cpp Reference.** Union declaration. [Online] https://en.cppreference.com/w/c/language/union.

29. *Description of a new variable-length key, 64-bit block cipher (Blowfish).* **Schneier, Bruce.** Berlin : Springer, 1993, Vol. 809.

30. **Aciiçmez, Onur, Schindler, Werner and Koç, Çetin K.** Cache Based Remote Timing Attack on the AES. [Online] 2007. http://cryptocode.net/docs/c38.pdf.

31. **Bernstein, Daniel J.** Cache-timing Attacks on AES. [Online] 2005. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

32. **Osvik, Dag Arne, Shamir, Adi and Tromer, Eran.** Cache Attacks and Countermeasures: the Case of AES. [Online] November 20, 2015. https://www.cs.tau.ac.il/~tromer/papers/cache.pdf.

Appendices

Glossary

Challenge values: Random values used in authentication protocols to strengthen them.

Symmetric cryptography: Algorithms that use the same key to encrypt and decrypt a message. It's recommended to use at least 128-bit key length and are very fast. They are the recommended option to encrypt data.

Session cookies: Values that identify a user whilst the connection lasts. They should be random values and kept private.

Initialization vector: Random values that are used with an encryption algorithm. They should be different every time a message is encrypted with the same key. Their knowledge does not attempt to the message confidentiality.

Random number generator: A random number is a value obtained through external sources: heat, noise, packets send per second, etc. These generators are very slow, and their sequence cannot be reproduced again.

Pseudo-random number generator (PRNG): A pseudo-random number is a value that has the same properties as random values, and one cannot be distinguish from each other. These generators are based on algorithms and are very fast, as well as deterministic.

Seed: The seed is the initial value in a stream cipher or PRNG (Pseudo-Random Number Generator). It is also known as key.

LFSR: Linear Feedback Shift Register. Register formed by *n* cells. Each one of these cells contains a bit. At every clock beat, bits are shifted one position and XOR-ed those on the LFSR's polynomial function position. LFSR are very fast and used as PRNG and stream ciphers.

Keylogger: A keylogger can be software or hardware. A software-keylogger is a malware installed on someone's computer and logs into a file everything that the user types, subsequently this file is sent to the attacker to retrieve passwords or any other sensitive information. A hardware-keylogger is similar, but it is a physical device (usb-device) installed between the computer connector and the keyboard usb connector that logs everything that is typed; the attacker has to get the hardware-keylogger back to read the logged data.

Appendix A

```
RNG_test using PractRand version 0.94
RNG = RNG_stdin, seed = unknown
test set = core, folding = standard(unknown format)

rng=RNG_stdin, seed=unknown
length= 128 megabytes (2^27 bytes), time= 2.5 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low1/32]DC6-9x1Bytes-1            R=  +6.2  p =  3.0e-3   unusual
  ...and 195 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 256 megabytes (2^28 bytes), time= 5.1 seconds
  no anomalies in 213 test result(s)

rng=RNG_stdin, seed=unknown
length= 512 megabytes (2^29 bytes), time= 9.5 seconds
  no anomalies in 229 test result(s)

rng=RNG_stdin, seed=unknown
length= 1 gigabyte (2^30 bytes), time= 17.7 seconds
  no anomalies in 248 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 gigabytes (2^31 bytes), time= 33.4 seconds
  no anomalies in 266 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 gigabytes (2^32 bytes), time= 64.4 seconds
  no anomalies in 282 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 gigabytes (2^33 bytes), time= 126 seconds
  no anomalies in 299 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 gigabytes (2^34 bytes), time= 267 seconds
  no anomalies in 315 test result(s)

rng=RNG_stdin, seed=unknown
length= 32 gigabytes (2^35 bytes), time= 533 seconds
  no anomalies in 328 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 gigabytes (2^36 bytes), time= 1053 seconds
  no anomalies in 344 test result(s)

rng=RNG_stdin, seed=unknown
length= 128 gigabytes (2^37 bytes), time= 2113 seconds
  no anomalies in 359 test result(s)
```

Appendix B

```
========= Summary results of BigCrush =========

 Version:
 Generator:        GenP2
 Number of statistics:  160
 Total CPU time:   08:19:09.51

 All tests were passed
```

Appendix C

```
RNG_test using PractRand version 0.94
RNG = RNG_stdin, seed = unknown
test set = core, folding = standard(unknown format)

rng=RNG_stdin, seed=unknown
length= 128 megabytes (2^27 bytes), time= 2.3 seconds
  no anomalies in 196 test result(s)

rng=RNG_stdin, seed=unknown
length= 256 megabytes (2^28 bytes), time= 5.1 seconds
  no anomalies in 213 test result(s)

rng=RNG_stdin, seed=unknown
length= 512 megabytes (2^29 bytes), time= 9.6 seconds
  no anomalies in 229 test result(s)

rng=RNG_stdin, seed=unknown
length= 1 gigabyte (2^30 bytes), time= 19.1 seconds
  no anomalies in 248 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 gigabytes (2^31 bytes), time= 38.7 seconds
  no anomalies in 266 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 gigabytes (2^32 bytes), time= 76.6 seconds
  no anomalies in 282 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 gigabytes (2^33 bytes), time= 126 seconds
  no anomalies in 299 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 gigabytes (2^34 bytes), time= 257 seconds
  no anomalies in 315 test result(s)

rng=RNG_stdin, seed=unknown
length= 32 gigabytes (2^35 bytes), time= 474 seconds
  no anomalies in 328 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 gigabytes (2^36 bytes), time= 959 seconds
  no anomalies in 344 test result(s)

rng=RNG_stdin, seed=unknown
length= 128 gigabytes (2^37 bytes), time= 1900 seconds
  no anomalies in 359 test result(s)
```

Appendix D

```
***********************************************************
HOST =

GenP2


smarsa_GCD test:
-----------------------------------------------
   N = 10,  n = 50000000,  r =  0,   s = 30


-----------------------------------------------
Test results for GCD values:

Kolmogorov-Smirnov+ statistic = D+    :    0.43
p-value of test                       :    0.02

Kolmogorov-Smirnov- statistic = D-    : 2.16e-3
p-value of test                       :    0.9978

Anderson-Darling statistic = A2       :    4.82
p-value of test                       :  3.7e-3

Test on the sum of all N observations
Number of degrees of freedom          : 17430
Chi-square statistic                  :16851.59
p-value of test                       :    0.9991    *****



-----------------------------------------------
CPU time used                    :  00:01:49.93

Generator state:




========= Summary results of BigCrush =========

 Version:
 Generator:        GenP2
 Number of statistics:  160
 Total CPU time:   07:29:39.75
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
 25  ClosePairs mNP2, t = 16          9.6e-4
 73  GCD                              0.9991
 ----------------------------------------------
```

```
All other tests were passed
```

Appendix E

```
RNG_test using PractRand version 0.94
RNG = RNG_stdin, seed = unknown
test set = core, folding = standard(unknown format)

rng=RNG_stdin, seed=unknown
length= 128 megabytes (2^27 bytes), time= 2.4 seconds
  no anomalies in 196 test result(s)

rng=RNG_stdin, seed=unknown
length= 256 megabytes (2^28 bytes), time= 5.1 seconds
  no anomalies in 213 test result(s)

rng=RNG_stdin, seed=unknown
length= 512 megabytes (2^29 bytes), time= 9.5 seconds
  no anomalies in 229 test result(s)

rng=RNG_stdin, seed=unknown
length= 1 gigabyte (2^30 bytes), time= 17.6 seconds
  no anomalies in 248 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 gigabytes (2^31 bytes), time= 33.1 seconds
  no anomalies in 266 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 gigabytes (2^32 bytes), time= 63.9 seconds
  no anomalies in 282 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 gigabytes (2^33 bytes), time= 128 seconds
  no anomalies in 299 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 gigabytes (2^34 bytes), time= 255 seconds
  no anomalies in 315 test result(s)

rng=RNG_stdin, seed=unknown
length= 32 gigabytes (2^35 bytes), time= 503 seconds
  no anomalies in 328 test result(s)

rng=RNG_stdin, seed=unknown
length= 64 gigabytes (2^36 bytes), time= 1057 seconds
  no anomalies in 344 test result(s)

rng=RNG_stdin, seed=unknown
length= 128 gigabytes (2^37 bytes), time= 2096 seconds
  no anomalies in 359 test result(s)
```

Appendix F

```
***********************************************************
HOST =

GenP2


scomp_LinearComp test:
-----------------------------------------------
   N =  1,  n = 400020,  r = 29,    s = 1


-----------------------------------------------
Number of degrees of freedom          :   12
Chi2 statistic for size of jumps      :   42.66
p-value of test                       :  2.6e-5    *****

-----------------------------------------------
Normal statistic for number of jumps  :   -3.82
p-value of test                       : 1 -  6.8e-5    *****


-----------------------------------------------
CPU time used                    :  00:02:31.73

Generator state:




========= Summary results of BigCrush =========

 Version:
 Generator:        GenP2
 Number of statistics:  160
 Total CPU time:   08:13:00.06
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
 81  LinearComp, r = 29            1 -  6.8e-5
 81  LinearComp, r = 0              2.6e-5
 ----------------------------------------------
 All other tests were passed
```

Appendix G

```
RNG_test using PractRand version 0.94
RNG = RNG_stdin, seed = unknown
test set = core, folding = standard(unknown format)

rng=RNG_stdin, seed=unknown
length= 128 megabytes (2^27 bytes), time= 2.9 seconds
  no anomalies in 196 test result(s)

rng=RNG_stdin, seed=unknown
length= 256 megabytes (2^28 bytes), time= 6.3 seconds
  no anomalies in 213 test result(s)

rng=RNG_stdin, seed=unknown
length= 512 megabytes (2^29 bytes), time= 11.6 seconds
  no anomalies in 229 test result(s)

rng=RNG_stdin, seed=unknown
length= 1 gigabyte (2^30 bytes), time= 20.4 seconds
  no anomalies in 248 test result(s)

rng=RNG_stdin, seed=unknown
length= 2 gigabytes (2^31 bytes), time= 37.0 seconds
  no anomalies in 266 test result(s)

rng=RNG_stdin, seed=unknown
length= 4 gigabytes (2^32 bytes), time= 82.0 seconds
  no anomalies in 282 test result(s)

rng=RNG_stdin, seed=unknown
length= 8 gigabytes (2^33 bytes), time= 158 seconds
  no anomalies in 299 test result(s)

rng=RNG_stdin, seed=unknown
length= 16 gigabytes (2^34 bytes), time= 288 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+1,13-0,T)                    R=  -9.3  p =1-6.1e-5   mildly
suspicious
  ...and 314 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 32 gigabytes (2^35 bytes), time= 577 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low4/32]DC6-9x1Bytes-1            R=  -4.6  p =1-3.6e-3   unusual
  ...and 327 test result(s) without anomalies

rng=RNG_stdin, seed=unknown
length= 64 gigabytes (2^36 bytes), time= 1236 seconds
  no anomalies in 344 test result(s)

rng=RNG_stdin, seed=unknown
length= 128 gigabytes (2^37 bytes), time= 2424 seconds
  no anomalies in 359 test result(s)
```

Appendix H

```
========= Summary results of BigCrush =========

 Version:
 Generator:        GenP2
 Number of statistics:  160
 Total CPU time:   07:35:31.26

 All tests were passed
```