THE STATE COMMITTEE FOR COMMUNICATION, INFORMATION AND TELECOMMUNICATION TECHNOLOGIES OF THE REPUBLIC OF UZBEKISTAN

TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES

COURSE WORK

Subject: Cryptography methods

Theme: The Stability Theory of Stream ciphers

Completed: **Shazimov A**

Group № 233-12

***Examiner:*Gulomov Sh**

Tashkent – 2014

CONTENT

CHAPTER 1. THE STABILITY THEORY OF STREAM CIPHERS

The Outline:

- What do we mean by stability?
- The stability of stream ciphers.
- The stability of building blocks of stream ciphers.
- Concluding remarks.

Part I: An Introduction to Stability

1.1 What do we mean by stability?

There is no uniform definition for the word "stability". It could mean different things in different systems.

• **Atmospheric stability:** a measure of the turbulence in the ambient atmosphere.

• **Ecological stability:** measure of the probability of a population returning quickly to a previous state, or not going extinct.

• **Social stability:** lack of civil unrest in a society.

Our definition: The resistance of of a system to small changes in some system parameters.

1.2 Factors affecting social stability

Social stability includes:

• The stability of economy, political situation, and living situation.

The factors include:

• The distribution of social wealth.

• The distribution of political rights.

How to achieve social stability?

E.g., by law one-husband-one-wife, income tax.

Part II: Introduction to the Stability of Stream Ciphers

2.1 Block ciphers versus stream ciphers

Definition: Stream and block ciphers depending on if Ek is time-varying for a fixed k

Comment: Although many block ciphers available, in most cases people use stream ciphers rather than block ciphers. Why?

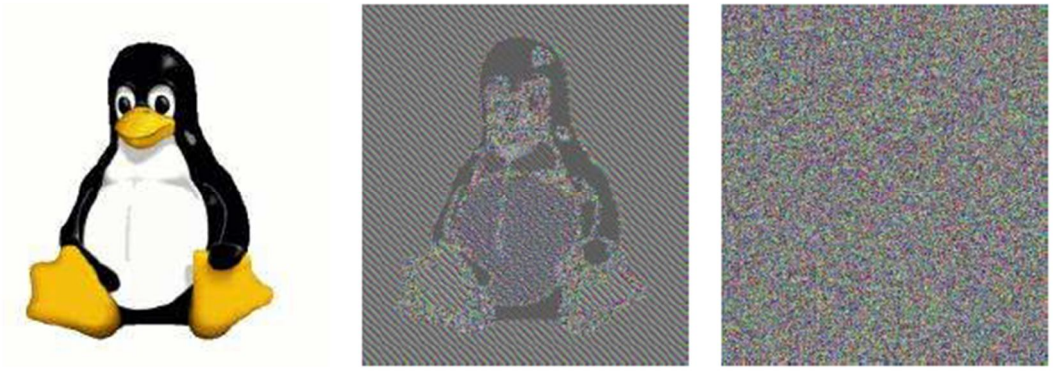
• Stream ciphers destroy statistical properties in natural languages, while block ciphers cannot.

• Some stream ciphers are very faster in both hardware and software.

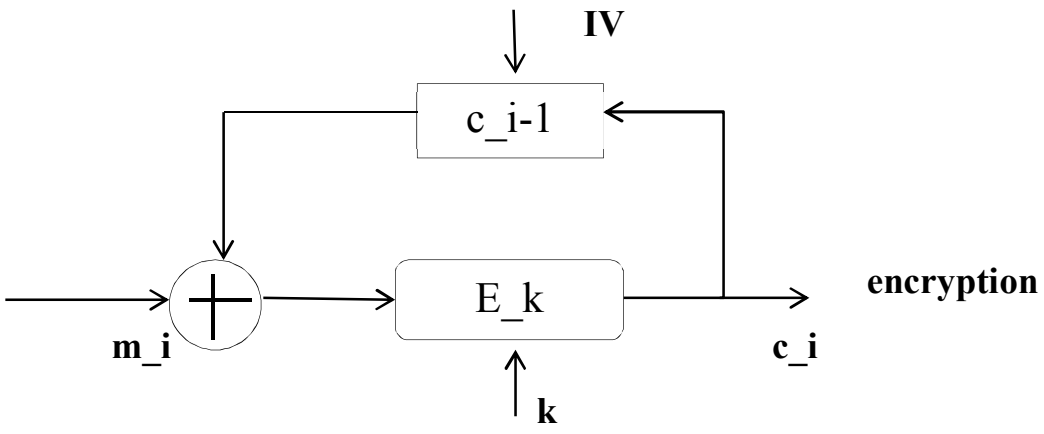
Comment: Block ciphers, such as 3DES and AES, are used in CBC mode. In this case, we are using a stream cipher.

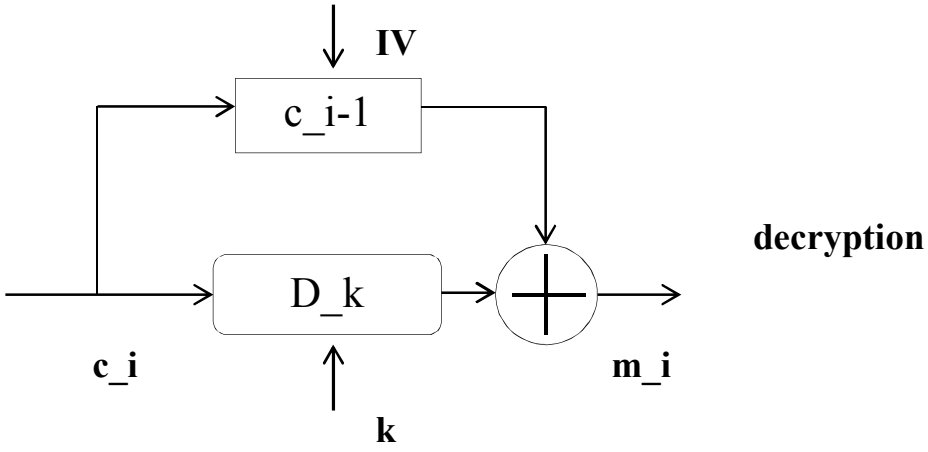
2.2 Block ciphers versus stream ciphers

Conclusion: Stream ciphers could destroy statistical properties in natural languages, while block ciphers cannot.



2.3 Examples of stream ciphers (CBC mode)





Question: What do we mean by stability here? We ask the same question for the Cipher Feedback mode and the Output Feedback mode.

2.4 Examples of stream ciphers

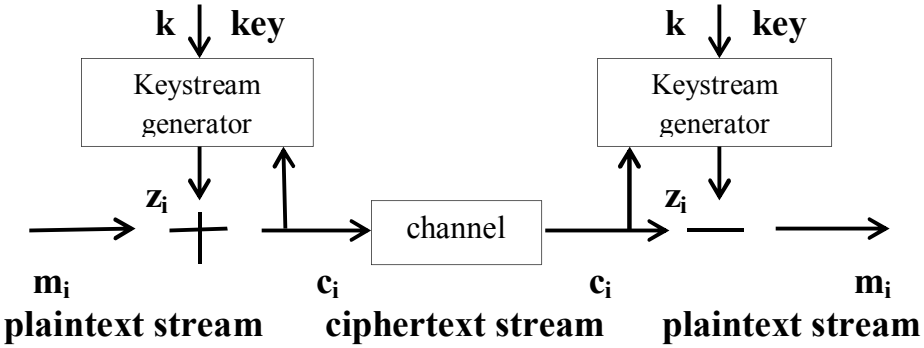


Figure 1: Additive self-synchronous stream ciphers.

• What do we mean by stability here?

• Is the linear complexity of the keystream important? How do you control it?

2.5 Examples of stream ciphers

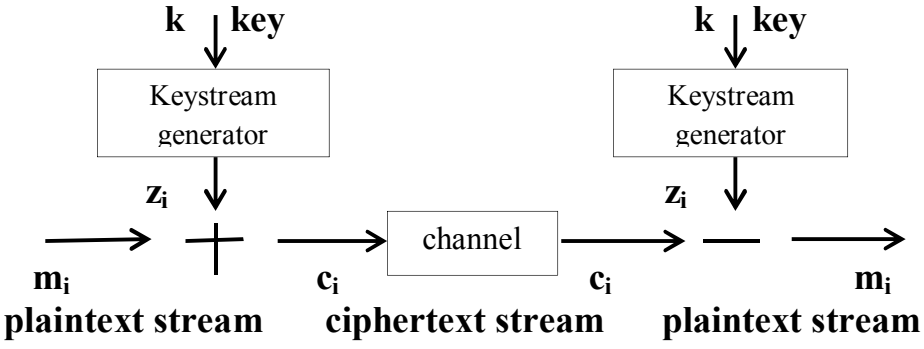


Figure 2: Additive synchronous stream ciphers.

• What do we mean by stability here?

The answer may depend on the design of the keystream generator.

Part III: The Stability of Additive Synchronous Stream Ciphers

3.1 Stability of additive synchronous stream ciphers

• The stability of linear complexity (defined later) .

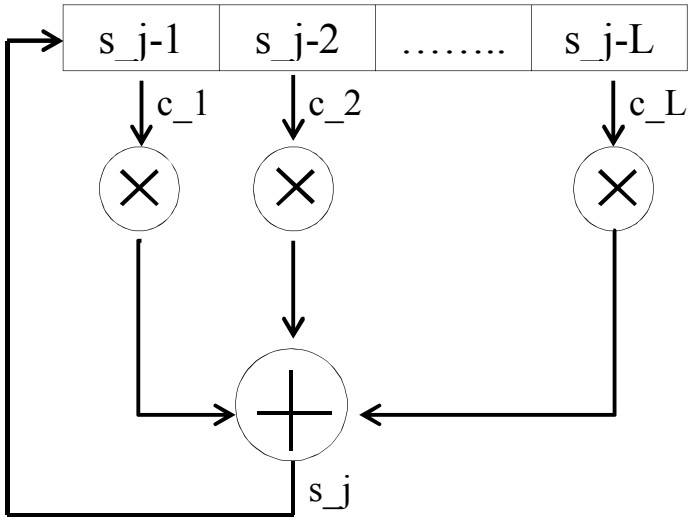
• The stability of building blocks of the keystream generator (defined later)

3.2 Linear feedback shift registers

A binary LFSR is a device for implementing a linear recursion:

$sn = c1sn-1 + c2sn-2 + \cdots + cLsn-L, n \_ L,$

where ci 2 {0, 1} and the operations are modulo- 2.



L is the *length* of the LFSR, and $c(x) = 1 + c1x + \cdots + cLxL$ the *feedback* or *connection* polynomial of the LFSR.

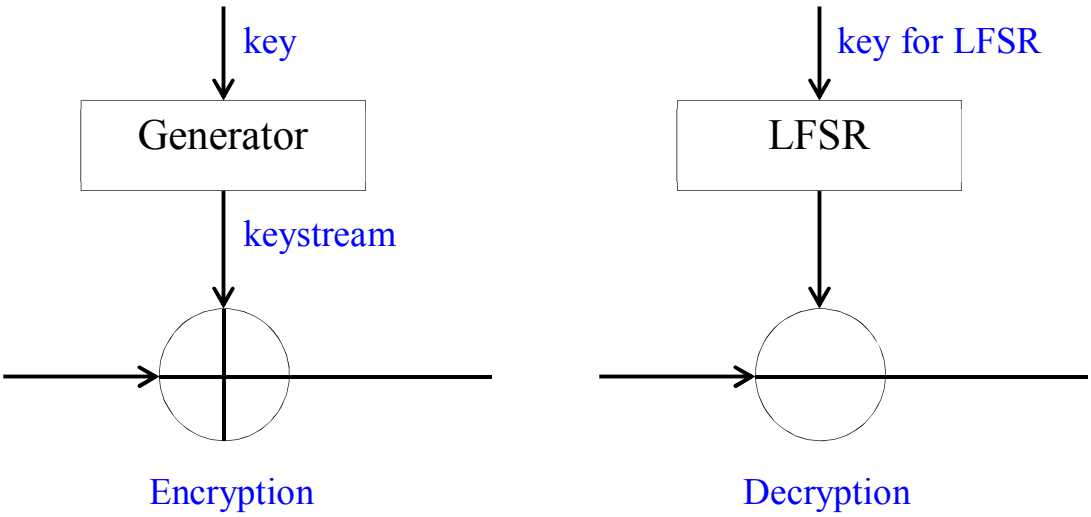
The linear complexity

Theorem: For any given binary sequence of length n or binary ultimately periodic sequence, there is an algorithm (Berlekamp-Massey) that finds a shortest LFSR generating this sequence. The complexity of this algorithm is O(n2).

Definition: The length of the shortest LFSR that can produce a given finite or ultimately periodic sequence is defined to be the *linear complexity (span)* of the sequence.

Security measure: The linear complexity of the keystream should be a security measure for additive synchronous stream ciphers.

3.3 The stability of linear complexity



LFSR approximation attack: For any keystream generator, we construct an LFSR whose output sequence is "almost the same" as the output sequence of the original keystream generator.

Linear complexity stability: Changing a small number of bits in a periodic segment will not result in a new sequence with low linear complexity.

3.4 The weight complexity or sphere surface complexity

Weight complexity for finite sequences: Let x be a sequence of length n. The weight complexity of x is defined to be

$$WC_k(x) = \min_{W_{H(y)=k}} L(x+y)$$

where WH(y) is the Hamming weight of y, and L(x) the linear complexity of x.

The weight complexity or sphere surface complexity

Weight complexity for periodic sequences: Let x∞ be a sequence of period n. The weight complexity of x∞ is defined to be

$$\mathrm{WC}_k(x^\infty) = \min_{\substack{\mathrm{Per}(y^\infty)=n \\ W_H(y^n)=k}} L(x^\infty + y^\infty),$$

where yn denotes the first periodic segment of y∞, and Per(x) the period of x.

3.5 The sphere complexity

Sphere complexity for finite sequences: Let x be a sequence of length n. The sphere complexity of x is defined to be

$$\mathrm{SC}_k(x) = \min_{0<W_H(y)\le k} L(x+y) = \min_{0<\ell\le k} \mathrm{WC}_\ell(x),$$

where WH(y) is the Hamming weight of y, and L(x) the linear complexity of x.

The sphere complexity

Sphere complexity for periodic sequences: Let x∞ be a sequence of period n. The sphere complexity of x∞ is defined to be

$$\mathrm{SC}_k(x^\infty) = \min_{\substack{\mathrm{Per}(y^\infty)=n \\ 0<W_H(y^n)\le k}} L(x^\infty + y^\infty) = \min_{0<\ell\le k} \mathrm{WC}_\ell(x^\infty),$$

where yn denotes the first periodic segment of y∞, and Per(x) the period of x.

3.6 The k-error linear complexity

The k-error linear complexity for finite sequences: Let x be a sequence of length n. The k-error linear complexity of x is defined to be

Lk(x) = min{L(x), SCk(x)}.

The k-error linear complexity for periodic sequences: Let x∞ be a sequence of period n. The k-error linear complexity of x∞ is defined to be

Lk(x∞) = min{L(x∞), SCk(x∞)}.

3.7 The stability of linear complexity for periodic sequences

- There are about 200 papers on the stability of linear complexity:
- Computing the sphere complexity SCk(s) is very hard in general, through it can be done in certain special cases.
- For application purposes, computing the exact value of SCk(s) is not necessary. We need only a good lower bound on SCk(s).
- The control of the sphere complexity for periodic sequences is easy. The main technique is the choice of the format of the least period.
- The control of the sphere complexity of a segment of a periodic sequence is very hard, but more important in applications.

3.8 Control of the stability of linear complexity

Basic Theorem: Suppose N = pe1

1 · · · pet

t , where p1, · · · , pt are t pairwise

distinct primes, and q is a positive integer such that gcd(q,N) = 1. Then for each nonconstant sequence s∞ of period N over GF(q),

$L(s^\infty) \geq \min\{Ordp_1(q), \cdots, Ordp_t(q)\}$,

$SC_k(s^\infty) \geq \min\{Ordp_1(q), \cdots, Ordp_t(q)\}$, if $k < \min\{WH(sN), N - WH(sN)\}$.

• Control the sphere complexity by choosing proper pi and q.

• This is a bridge between cryptography and number theory.

• Many papers on the computation of the sphere complexity are based on the proof of this basic theorem.

3.9 Advances on the stability of linear complexity

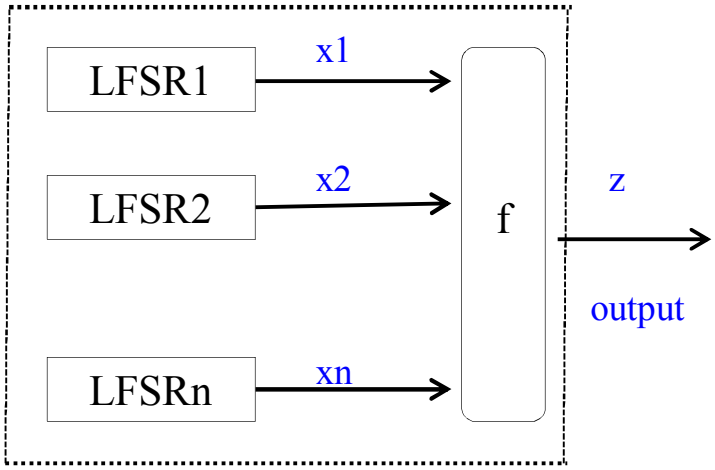
• A number of papers on developing algorithms for the computation of the sphere complexity $SC_k(s^\infty)$. But they work efficiently only for special cases. No efficient algorithm for the computation of sphere complexity $SC_k(s^\infty)$ is known, and is expected due to the inherent difficulty of this problem.

• A lot of papers on the computation of the sphere complexity $SC_k(s^\infty)$. But only for small values of k. Typically, k = 1 and k = 2.

Open problem: How to control the **local** linear complexity stability of keystreams?

Part IV: Stability of Functions

Stability of the nonlinear combiner (I)



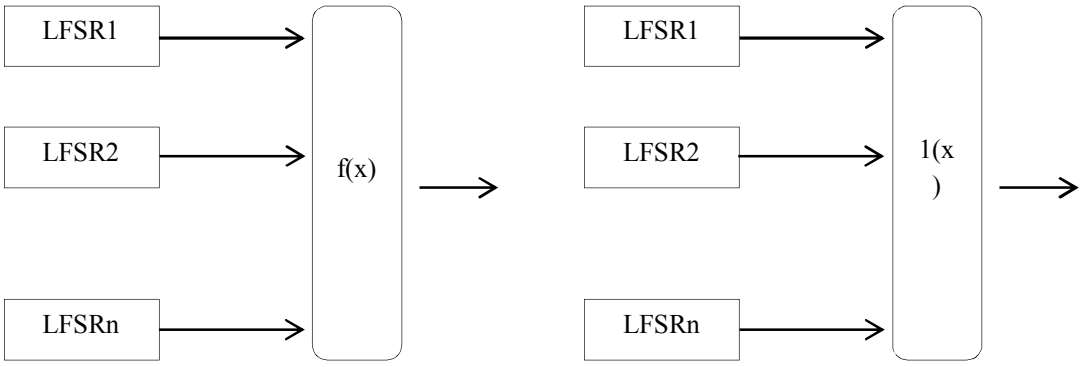
keystream generator

• This keystream generator for additive synchronous stream ciphers has been studied for a long time.

• We do not have good results on its linear complexity stability.

• What else do we mean by stability of this generator?

Stability of the nonlinear combiner (II)



Original generator New generator from BAA

• The **best affine approximation attack** (BAA) is to use an affine function $l(x) = a_1x_1 + \cdots + a_nx_n$ to approximate the nonlinear function f(x). The new generator is then used to approximate the original keystream generator.

• It then follows that the Hamming distance between f(x) and any affine function should be as large as possible.

Question: Can this be achieved?

Stability of the nonlinear combiner (III)

Let f(x1, · · · , xn) be Boolean function. Then the **2nd kind of Walsh transform**

$$S_{(f)}(w) = 2^{-n} \sum_{z \in \mathrm{GF}(2)^n} (-1)^{f(z)-w\cdot z} = 2^{-n}\left(2^n - 2d_H(f(x), w\cdot x)\right),$$

where dH denotes the Hamming distance.

• **Energy Conservation Law:** $\sum_{w\in \mathrm{GF}(2)^n} S_{(f)}(w)^2 = 1.$

• The best situation $|S_{(f)}(w)| = 2^{-n/2}$ for every w. Hence f is a **Bent** function. In this case, the Hamming distance between f and every is either 2n−1 − 2(n−2)/2 or 2n−1 + 2(n−2)/2.

• One stability measure is: maxw $|S_{(f)}(w)|$

Stability of the nonlinear combiner (IV)

Correlation-immune functions: A Boolean function f(x1, · · · , xn) is called m-*th order correlation immune* if for any subset {i1, i2, · · · , im} _ {1, 2, · · · , n}, the mutual information

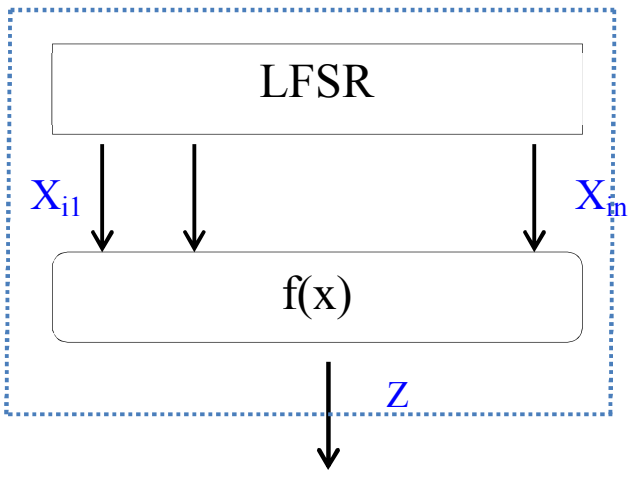
$$I(f(x); \{x_{i1}, x_{i2}, \cdots, x_{im}\}) = 0.$$

Xiao-Massey: f is m-th order correlation-immune iff

$$S_{(f)}(w) = 0 \text{ for all } w \text{ with } 1 \_ WH(w) \_ m.$$

Warning: Correlation-immunity is misleading, although correlation-immune functions of small order may be employed in the nonlinear combiner.

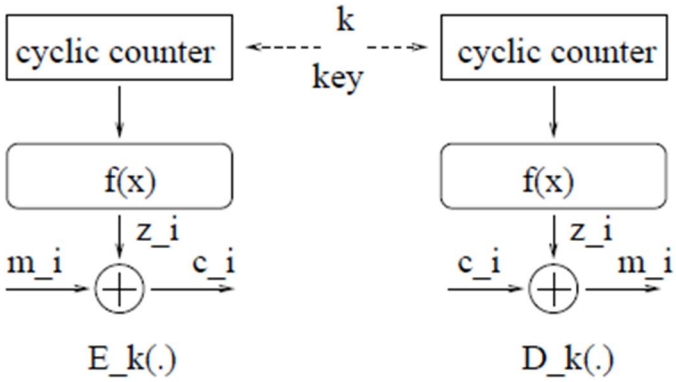
Stability of the nonlinear filter



keystream sequence

• This is similar to the stability of the nonlinear combiner.

Stability of the binary cyclic counter generator



• How to choose f in order to make the keystream have large linear complexity?

• How to choose f in order to make the keystream have good linear complexity stability?

Stability of the binary cyclic counter generator

Key approximation: Use a key k′ to decrypt the ciphertext encrypted by a key k.

• The rate of agreement is given by (1 + C(k − k′))/2, where

$$C(\tau) = \frac{1}{N} \sum_{i=0}^{N-1} (-1)^{s_{i+\tau} - s_i}.$$

• So C(_) should be zero or as small as possible for all $1 \leq \tau \leq N - 1.$

• **Conservation:** WH(z^N) the weight of the first period of the keystream.

$$\sum_{\tau=1}^{N-1} C(\tau) = (N - 2W_H(s^N))^2/N - 1.$$

The stability of the cyclic counter generator

Key approximation: Use a key k′ to decrypt the ciphertext encrypted by a key k.

max

$$\max_{1\le\tau\le N-1}|C(\tau)| = \max_{1\le\tau\le N-1}\left|\frac{2|\{x\in Z_N : f(x+\tau)-f(x)=0\}|-N}{N}\right|$$
$$= \max_{1\le\tau\le N-1}\left|\frac{N-2|\{x\in Z_N : f(x+\tau)-f(x)=1\}|}{N}\right|$$

It follows that $\max_{1\le\tau\le N-1}|C(\tau)|$ is minimal if and only if

$$p_f = \max_{a\neq 0}\max_{b\in\{0,1\}}\frac{|\{x\in Z_N : f(x+a)-f(x)=b\}|}{N}$$

is minimal.

The quantity pf is a measure of nonlinearity of f. To thwart this attack, the underlying function f should have high nonlinearity.

The stability of the cyclic counter generator

Key approximation: Use a key k′ to decrypt the ciphertext encrypted by a key k.

Perfect nonlinear functions: For any f : (A,+) ! (B,+), we have that

$$P_f \ge \frac{1}{|B|}.$$

A function $f : A \to B$ has perfect nonlinearity if $P_f = \frac{1}{|B|}$.

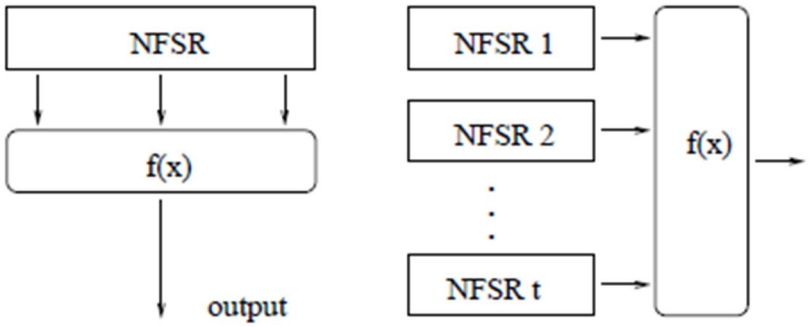
Part V: Summary and Open Problems

The stability theory is far from mature

• For additive synchronous stream ciphers, we have techniques to control the linear complexity stability. However, local linear complexity stability seems hard to control.

• For the nonlinear combiner and nonlinear filter, we have *partial theory* on stability. Other stability aspects remain to be investigated.

The stability theory is far from mature



• For most stream ciphers, stability is open. E.g., the generators above.

• Every stability aspect must be associated with a cryptographic attack.

• Stability issues differ from cipher to cipher.

• Stability is a promising area, where a lot of work can be done.

CHAPTER 2. ABOUT RC4

In cryptography, **RC4** (also known as **ARC4** or **ARCFOUR** meaning Alleged RC4, see below) is the most widely used software stream cipher and is used in popular protocols such asTransport Layer Security (TLS) (to protect Internet traffic) and WEP (to secure wireless networks). While remarkable for its simplicity and speed in software, RC4 has weaknesses that argue against its use in new systems. It is especially vulnerable when the beginning of the output keystream is not discarded, or when nonrandom or related keys are used; some ways of using RC4 can lead to very insecure protocols such as WEP.

As of 2015, there is speculation that some state cryptologic agencies may possess the capability to break RC4 even when used in the TLS protocol. Mozilla and Microsoft recommend disabling RC4 where possible. In 2014 Ronald Rivest gave a talk and published a paper on an updated redesign called Spritz.

History

RC4 was designed by Ron Rivest of RSA Security in 1987. While it is officially termed "Rivest Cipher 4", the RC acronym is alternatively understood to stand for "Ron's Code" (see also RC2, RC5 and RC6).

RC4 was initially a trade secret, but in September 1994 a description of it was anonymously posted to the Cypherpunks mailing list. It was soon posted on the sci.crypt newsgroup, and from there to many sites on the Internet. The leaked code was confirmed to be genuine as its output was found to match that of proprietary software using licensed RC4. Because the algorithm is known, it is no longer a trade secret. The name *RC4* is trademarked, so RC4 is often referred to as *ARCFOUR* or *ARC4* (meaning *alleged RC4*) to avoid trademark problems. RSA Security has never officially released the algorithm; Rivest has, however, linked to the English Wikipedia article on RC4 in his own course notes and confirmed the history of RC4 and its code in a paper by him. RC4 has become part of some commonly used encryption protocols and standards, including WEP and WPA for wireless cards andTLS.

The main factors in RC4's success over such a wide range of applications are its speed and simplicity: Efficient implementations in both software and hardware are very easy to develop.

Description

RC4 generates a pseudorandom stream of bits (a keystream). As with any stream cipher, these can be used for encryption by combining it with the plaintext using bit-wise exclusive-or; decryption is performed the same way (since exclusive-or with given data is an involution). (This is similar to the Vernam cipher except that generated *pseudorandom bits*, rather than a prepared stream, are used.) To generate the keystream, the cipher makes use of a secret internal state which consists of two parts:

1. A permutation of all 256 possible bytes (denoted "S" below).
2. Two 8-bit index-pointers (denoted "i" and "j").

The permutation is initialized with a variable length key, typically between 40 and 256 bits, using the *key-scheduling* algorithm (KSA). Once this has been completed, the stream of bits is generated using the *pseudo-random generation algorithm* (PRGA).

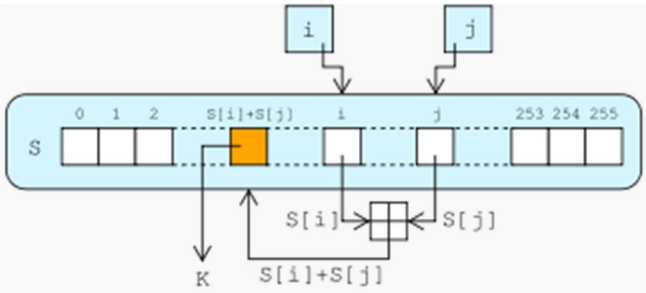
Key-scheduling algorithm (KSA)

The key-scheduling algorithm is used to initialize the permutation in the array "S". "keylength" is defined as the number of bytes in the key and can be in the range $1 \leq \text{keylength} \leq 256$, typically between 5 and 16, corresponding to a key length of 40 – 128 bits. First, the array "S" is initialized to the identity permutation. S is then processed for 256 iterations in a similar way to the main PRGA, but also mixes in bytes of the key at the same time.

```
for i from 0 to 255
   S[i] := i
endfor
j := 0
for i from 0 to 255
   j := (j + S[i] + key[i mod keylength]) mod 256
```

```
    swap values of S[i] and S[j]
endfor
```

Pseudo-random generation algorithm (PRGA)



The lookup stage of RC4. The output byte is selected by looking up the values of S(i) and S(j), adding them together modulo 256, and then using the sum as an index into S; S(S(i) + S(j)) is used as a byte of the key stream, K.

For as many iterations as are needed, the PRGA modifies the state and outputs a byte of the keystream. In each iteration, the PRGA increments *i*, looks up the *i*th element of S, S[*i*], and adds that to *j*, exchanges the values of S[*i*] and S[*j*], and then uses the sum S[*i*] + S[*j*] (modulo 256) as an index to fetch a third element of S, (the keystream value K below) which is XORed with the next byte of the message to produce the next byte of either ciphertext or plaintext. Each element of S is swapped with another element at least once every 256 iterations.

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    K := S[(S[i] + S[j]) mod 256]
    output K
endwhile
```

RC4-based random number generators

Several operating systems include arc4random, an API originating in OpenBSD providing access to a random number generator originally based on

RC4. In OpenBSD 5.5, released in May 2014, arc4random was modified to useChaCha20. As of January 2015, implementation of arc4random in NetBSD also uses ChaCha20, however, implementation of arc4random in FreeBSD, Linux's libbsd, and Mac OS X are still based on RC4.

Proposed new random number generators are often compared to the RC4 random number generator.

Unfortunately, several attacks on RC4 are able to distinguish its output from a random sequence.

Implementation

Many stream ciphers are based on linear feedback shift registers (LFSRs), which, while efficient in hardware, are less so in software. The design of RC4 avoids the use of LFSRs, and is ideal for software implementation, as it requires only byte manipulations. It uses 256 bytes of memory for the state array, S[0] through S[255], k bytes of memory for the key, key[0] through key[k-1], and integer variables, i, j, and K. Performing a modular reduction of some value modulo 256 can be done with a bitwise AND with 255 (which is equivalent to taking the low-order byte of the value in question).

Test vectors

These test vectors are not official, but convenient for anyone testing their own RC4 program. The keys and plaintext are ASCII, the keystream and ciphertext are in hexadecimal.

Key	Keystream	Plaintext	Ciphertext
Key	EB9F7781B734CA72A719...	Plaintext	BBF316E8D940AF0AD3
Wiki	6044DB6D41B7...	pedia	1021BF0420
Secret	04D46B053CA87B59...	Attack at dawn	45A01F645FC35B383552544B9BF5

Security

Unlike a modern stream cipher (such as those in eSTREAM), RC4 does not take a separate nonce alongside the key. This means that if a single long-term key is to be used to securely encrypt multiple streams, the protocol must specify how to combine the nonce and the long-term key to generate the stream key for RC4. One approach to addressing this is to generate a "fresh" RC4 key by hashing a long-term key with a nonce. However, many applications that use RC4 simply concatenate key and nonce; RC4's weak key schedule then gives rise to related key attacks, like the Fluhrer, Mantin and Shamir attack (which is famous for breaking the WEP standard).

Because RC4 is a stream cipher, it is more malleable than common block ciphers. If not used together with a strong message authentication code (MAC), then encryption is vulnerable to a bit-flipping attack. The cipher is also vulnerable to a stream cipher attack if not implemented correctly. Furthermore, inadvertent double encryption of a message with the same key may accidentally output plaintext rather than ciphertext because the involutorynature of the XOR function would result in the second operation reversing the first.

It is noteworthy, however, that RC4, being a stream cipher, was for a period of time the only common cipher that was immune to the 2011 BEAST attack on TLS 1.0. The attack exploits a known weakness in the way cipher block chaining mode is used with all of the other ciphers supported by TLS 1.0, which are all block ciphers.

In March 2013, there were new attack scenarios proposed by Isobe, Ohigashi, Watanabe and Morii, as well as AlFardan, Bernstein, Paterson, Poettering and Schuldt that use new statistical biases in RC4 key table to recover plaintext with large number of TLS encryptions.

Roos' biases and key reconstruction from permutation

In 1995, Andrew Roos experimentally observed that the first byte of the keystream is correlated to the first three bytes of the key and the first few bytes of the permutation after the KSA are correlated to some linear combination of the key

bytes. These biases remained unproven until 2007, when Goutam Paul, Siddheshwar Rathi and Subhamoy Maitra proved the keystream-key correlation and in another work Goutam Paul and Subhamoy Maitra proved the permutation-key correlations. The latter work also used the permutation-key correlations to design the first algorithm for complete key reconstruction from the final permutation after the KSA, without any assumption on the key or IV. This algorithm has a constant probability of success in a time which is the square root of the exhaustive key search complexity. Subsequently, many other works have been performed on key reconstruction from RC4 internal states. Subhamoy Maitra and Goutam Paul also showed that the Roos type biases still persist even when one considers nested permutation indices, like `S[S[i]]` or `S[S[S[i]]]`. These types of biases are used in some of the later key reconstruction methods for increasing the success probability.

Biased outputs of the RC4

The keystream generated by the RC4 is biased in varying degrees towards certain sequences making it vulnerable to distinguishing attacks. The best such attack is due to Itsik Mantin and Adi Shamir who showed that the second output byte of the cipher was biased toward zero with probability 1/128 (instead of 1/256). This is due to the fact that if the third byte of the original state is zero, and the second byte is not equal to 2, then the second output byte is always zero. Such bias can be detected by observing only 256 bytes.

Souradyuti Paul and Bart Preneel of COSIC showed that the first and the second bytes of the RC4 were also biased. The number of required samples to detect this bias is 2^{25} bytes.

Scott Fluhrer and David McGrew also showed such attacks which distinguished the keystream of the RC4 from a random stream given a gigabyte of output.

The complete characterization of a single step of RC4 PRGA was performed by Riddhipratim Basu, Shirshendu Ganguly, Subhamoy Maitra, and Goutam Paul. Considering all the permutations, they prove that the distribution of the

output is not uniform given i and j, and as a consequence, information about j is always leaked into the output.

Fluhrer, Mantin and Shamir attack

Main article: Fluhrer, Mantin and Shamir attack

In 2001, a new and surprising discovery was made by Fluhrer, Mantin and Shamir: over all possible RC4 keys, the statistics for the first few bytes of output keystream are strongly non-random, leaking information about the key. If the long-term key and nonce are simply concatenated to generate the RC4 key, this long-term key can be discovered by analysing a large number of messages encrypted with this key. This and related effects were then used to break the WEP ("wired equivalent privacy") encryption used with 802.11 wireless networks. This caused a scramble for a standards-based replacement for WEP in the 802.11 market, and led to the IEEE 802.11i effort andWPA.

Protocols can defend against this attack by discarding the initial portion of the keystream. Such a modified algorithm is traditionally called "RC4-drop[n]", where n is the number of initial keystream bytes that are dropped. The SCAN default is n = 768 bytes, but a conservative value would be n = 3072 bytes.

The Fluhrer, Mantin and Shamir attack does not apply to RC4-based SSL, since SSL generates the encryption keys it uses for RC4 by hashing, meaning that different SSL sessions have unrelated keys.

Klein's attack

In 2005, Andreas Klein presented an analysis of the RC4 stream cipher showing more correlations between the RC4 keystream and the key. Erik Tews, Ralf-Philipp Weinmann, and Andrei Pychkine used this analysis to create aircrack-ptw, a tool which cracks 104-bit RC4 used in 128-bit WEP in under a minute. Whereas the Fluhrer, Mantin, and Shamir attack used around 10 million messages, aircrack-ptw can break 104-bit keys in 40,000 frames with 50% probability, or in 85,000 frames with 95% probability.

Combinatorial problem

A combinatorial problem related to the number of inputs and outputs of the RC4 cipher was first posed by Itsik Mantin and Adi Shamir in 2001, whereby, of the total 256 elements in the typical state of RC4, if x number of elements ($x \leq 256$) are *only* known (all other elements can be assumed empty), then the maximum number of elements that can be produced deterministically is also x in the next 256 rounds. This conjecture was put to rest in 2004 with a formal proof given by Souradyuti Paul and Bart Preneel.

Royal Holloway attack

In 2013, a group of security researchers at the Information Security Group at Royal Holloway, University of London reported an attack that can become effective using only 2^{24} connections. While yet not a practical attack for most purposes, this result is sufficiently close to one that it has led to speculation that it is plausible that some state cryptologic agencies may already have better attacks that render RC4 insecure. Given that as of 2013 a large amount of TLS traffic uses RC4 to avoid recent attacks on block ciphers that use cipher block chaining, if these hypothetical better attacks exist, then this would make the TLS-with-RC4 combination insecure against such attackers in a large number of practical scenarios.

RC4 variants

As mentioned above, the most important weakness of RC4 comes from the insufficient key schedule; the first bytes of output reveal information about the key. This can be corrected by simply discarding some initial portion of the output stream. This is known as RC4-drop$N$, where N is typically a multiple of 256, such as 768 or 1024.

A number of attempts have been made to strengthen RC4, notably Spritz, RC4A, VMPC, and RC4+.

RC4A

Souradyuti Paul and Bart Preneel have proposed an RC4 variant, which they call RC4A.

RC4A uses two state arrays S1 and S2, and two indexes *j1* and *j2*. Each time *i* is incremented, two bytes are generated:

1. First, the basic RC4 algorithm is performed using S1 and *j1*, but in the last step, S1[*i*] + S1[*j1*] is looked up in S2.
2. Second, the operation is repeated (without incrementing *i* again) on S2 and *j2*, and S1[S2[*i*]+S2[*j2*]] is output.

Thus, the algorithm is:

```
All arithmetic is performed modulo 256
i := 0
j1 := 0
j2 := 0
while GeneratingOutput:
   i := i + 1
   j1 := j1 + S1[i]
   swap values of S1[i] and S1[j1]
   output S2[S1[i] + S1[j1]]
   j2 := j2 + S2[i]
   swap values of S2[i] and S2[j2]
   output S1[S2[i] + S2[j2]]
endwhile
```

Although the algorithm required the same number of operations per output byte, there is greater parallelism than RC4, providing a possible speed improvement.

Although stronger than RC4, this algorithm has also been attacked, with Alexander Maximov and a team from NEC developing ways to distinguish its output from a truly random sequence.

VMPC

"Variably Modified Permutation Composition" is another RC4 variant. It uses similar key schedule as RC4, with j := S[(j + S[i] + key[i mod keylength]) mod 256] iterating 3 x 256 = 768 times rather than 256, and with an optional additional 768 iterations to incorporate an initial vector. The output generation function operates as follows:

```
All arithmetic is performed modulo 256.
i := 0
while GeneratingOutput:
   a := S[i]
   j := S[j + a]
   b := S[j]
   output S[S[b] + 1]
   S[i] := b     (Swap S[i] and S[j])
   S[j] := a
   i := i + 1
endwhile
```

This was attacked in the same papers as RC4A, and can be distinguished within 2^{38} output bytes.

RC4$^+$

RC4$^+$ is a modified version of RC4 with a more complex three-phase key schedule (taking about 3× as long as RC4, or the same as RC4-drop512), and a more complex output function which performs four additional lookups in the S array for each byte output, taking approximately 1.7× as long as basic RC4.

```
All arithmetic modulo 256.  << and >> are left and right shift, ⊕ is exclusive OR
while GeneratingOutput:
   i := i + 1
   a := S[i]
   j := j + a
   b := S[j]
   S[i] := b     (Swap S[i] and S[j])
   S[j] := a
   c := S[i<<5 ⊕ j>>3] + S[j<<5 ⊕ i>>3]
   output (S[a+b] + S[c⊕0xAA]) ⊕ S[j+b]
endwhile
```

This algorithm has not been analyzed significantly.

Spritz

Ron Rivest and Jacob Schuldt have proposed replacing RC4 with an improved and slightly modified version:

```
All arithmetic is performed modulo 256
while GeneratingOutput:
   i := i + w
   j := k + S[j + S[i]]
   k := k + i + S[j]
   swap values of S[i] and S[j]
   output z := S[j + S[i + S[z + k]]]
endwhile
Pseudocode with modulo 256 arithmetic
while GeneratingOutput:
   i := (i + w) mod 256
   j := (k + S[(j + S[i]) mod 256]) mod 256
   k := (k + i + S[j]) mod 256
   swap values of S[i] and S[j]
   output z := S[(j + S[(i + S[(z + k) mod 256]) mod 256]) mod 256]
endwhile
```

The value *w*, is relatively prime to the size of the S array. So after 256 iterations of this inner loop, the value *i* (incremented by w every iteration) has taken on all possible values 0..255, and every byte in the S array has been swapped at least once.

Like other sponge functions, Spritz can be used to build a cryptographic hash function, a deterministic random bit generator (DRBG), an encryption algorithm that supports authenticated encryption with associated data (AEAD), etc.

RC4-based protocols

- WEP
- WPA (default algorithm, but can be configured to use AES-CCMP instead of RC4)
- BitTorrent protocol encryption

- Microsoft Office XP (insecure implementation since nonce remains unchanged when documents get modified)
- Microsoft Point-to-Point Encryption
- Transport Layer Security / Secure Sockets Layer (optionally)
- Secure Shell (optionally)
- Remote Desktop Protocol
- Kerberos (optionally)
- SASL Mechanism Digest-MD5 (optionally, *historic*, obsoleted in RFC 6331)
- Gpcode.AK, an early June 2008 computer virus for Microsoft Windows, which takes documents hostage for ransom by obscuring them with RC4 and RSA-1024 encryption
- PDF
- Skype (in modified form)

Where a protocol is marked with "(optionally)", RC4 is one of several ciphers the system can be configured to use.

LIST OF REFERENCES

1. The stability theory of stream ciphers C.Ding, G.Xiao, W.Shan. (July, 2004)
2. RC4 Stream Cipher and Its Variants. Paul, Goutam; S. Maitra (2011).
3. "VMPC One-Way Function and Stream Cipher" Bartosz Zoltak (2004),
4. Security of RC4 Stream Cipher Isobe, Takanori; Ohigashi, Toshihiro (Mar 10–13, 2013).

APPENDIX

```
unit frmMain_;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TfrmMain = class(TForm)
    memText: TMemo;
    btnOpenFile: TButton;
    btnEncDec: TButton;
    btnSaveToFile: TButton;
    btnClearText: TButton;
    lblPathToFile: TLabel;
    edtPathToFile: TEdit;
    odOpenFile: TOpenDialog;
    SaveDialog1: TSaveDialog;

    procedure btnOpenFileClick(Sender: TObject);
    procedure btnSaveToFileClick(Sender: TObject);
    procedure btnEncDecClick(Sender: TObject);
    procedure btnClearTextClick(Sender: TObject);

    function  ApplyRC4ToText(tx: AnsiString): AnsiString;
    function  InitRC4Cipher: Boolean;
    function  GetRC4CharCiphered(ch: AnsiChar): AnsiChar;

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmMain: TfrmMain;
     s: array [0..255] of Byte;
     i,j: Byte;

implementation
  //uses frmProgress_;
{$R *.dfm}
```

```
//Инициализация S-бокса в зависимости от ключа

 function TfrmMain.InitRC4Cipher: Boolean;
var
   x: AnsiString;
   k: array [0..255] of Byte;
   t: Byte;
   l: Cardinal;
 i0,j0: Byte;

begin
  x := InputBox('RC4','Введите ключ','');

  if x = '' then
    begin
      ShowMessage('Вы не ввели ключ.');
      Result := false;
      Exit;
    end;

  for i0 := 0 to 255 do s[i0] := i0;

  j0 := 1; l := sizeof(x);
  for i0 := 0 to 255 do
    begin
      k[i0] := Ord(x[j0]);
      if j0 = l then j0 := 1;
      Inc(j0);
    end;

  for i0 := 0 to 255 do
    begin
      j0 := (j0 + k[i0] + s[i0]) mod 256;

      t    := s[i0];
      s[i0] := s[j0];
      s[j0] := t;
    end;
  i := 0;
  j := 0;
  Result := true;
end;

//Шифрование конкретного символа
function TfrmMain.GetRC4CharCiphered(ch: AnsiChar): AnsiChar;
```

```
var
  t: Byte;

begin
  i := (i + 1)    mod 256;
  j := (j + s[i]) mod 256;

  t := s[i];
  s[i] := s[j];
  s[j] := t;

  t := (s[i] + s[j]) mod 256;

  Result := Chr(Ord(ch) XOR s[t]);
end;

//функция шифрования всего текста
function TfrmMain.ApplyRC4ToText(tx: AnsiString): AnsiString;
var
  i: Cardinal;
  x: AnsiString;

begin
  if InitRC4Cipher = false then
    begin
      Result := tx;
      Exit;
    end;

  Result := '';
  for i := 1 to Length(tx) do
    Result := Result + GetRC4CharCiphered(tx[i]);
end;

 //Открыть файл
procedure TfrmMain.btnOpenFileClick(Sender: TObject);
begin
 if odOpenFile.Execute = false then Exit;

  memText.ReadOnly   := false;
  edtPathToFile.Text := odOpenFile.FileName;

  memText.Lines.LoadFromFile(edtPathToFile.Text);

end;
```

```
  //Сохраняем в файл

procedure TfrmMain.btnSaveToFileClick(Sender: TObject);
begin
   memText.Lines.SaveToFile(odOpenFile.FileName);
  ShowMessage('Сохранение прошло успешно');

end;
  //Шифрование данных

procedure TfrmMain.btnEncDecClick(Sender: TObject);
begin
 if memText.Lines.Text <> '' then
    memText.Lines.Text := ApplyRC4ToText(memText.Lines.Text)
  else
    begin
      ShowMessage('Поле ввода пусто.');
      Exit;
    end;

end;
  //Вывод исходного текста
procedure TfrmMain.btnClearTextClick(Sender: TObject);
begin

  if memText.Lines.Text <> '' then
   memText.Lines.LoadFromFile(edtPathToFile.Text)
   else
     ShowMessage('Нужно открыть сначала файл!!!');


end;

end.
```