Зміст

[ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І ТЕРМІНІВ 2](#_Toc381612759)

[ВСТУП 3](#_Toc381612760)

[1 ОЗНАЧЕННЯ НЕОБХІДНИХ В РОБОТІ ТЕРМІНІВ ТА ПОНЯТЬ 5](#_Toc381612761)

[1.1 Потокові шифри 5](#_Toc381612762)

[1.2 Класифікація потокових шифрів 6](#_Toc381612763)

[1.3 Синхронні потокові шифри 6](#_Toc381612764)

[1.4 Потокові шифри із самосинхронізацією 7](#_Toc381612765)

[1.5 Атака на основі вибраного ШТ 8](#_Toc381612766)

[2 ОГЛЯД ТА АНАЛІЗ ШИФРІВ 10](#_Toc381612767)

[2.1 Шифр Moustique 10](#_Toc381612768)

[2.2 Опис шифру SSS 15](#_Toc381612769)

[2.3 Опис атаки на основі вибраного ШТ на SSS 16](#_Toc381612770)

[ДОДАТОК 1 – ПРОГРАМНИЙ КОД 18](#_Toc381612771)

[ПЕРЕЛІК ПОСИЛАНЬ 25](#_Toc381612772)

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І ТЕРМІНІВ

# ВСТУП

Потокові шифри на базі [зсувних регістрів](https://uk.wikipedia.org/w/index.php?title=%D0%97%D1%81%D1%83%D0%B2%D0%BD%D0%B8%D0%B9_%D1%80%D0%B5%D0%B3%D1%96%D1%81%D1%82%D1%80&action=edit&redlink=1) активно використовувалися в роки війни, ще задовго до появи електроніки. Вони були прості в проектуванні та реалізації.

1965 Ернст Селмер, головний криптограф норвезького уряду, розробив теорію послідовності зсувних регістрів. Пізніше [Соломон Голомб](https://uk.wikipedia.org/w/index.php?title=%D0%A1%D0%BE%D0%BB%D0%BE%D0%BC%D0%BE%D0%BD_%D0%92%D0%BE%D0%BB%D1%8C%D1%84_%D0%93%D0%BE%D0%BB%D0%BE%D0%BC%D0%B1&action=edit&redlink=1), [математик](https://uk.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA) [Агентства Національної Безпеки США](https://uk.wikipedia.org/wiki/%D0%90%D0%B3%D0%B5%D0%BD%D1%82%D1%81%D1%82%D0%B2%D0%BE_%D0%BD%D0%B0%D1%86%D1%96%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D1%97_%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D0%BA%D0%B8), написав книгу під назвою «Shift Register Sequences» («Послідовності зсувних регістрів»), в якій виклав свої основні досягнення в цій галузі, а також досягнення Селмер.

Велику популярність потоковим шифрів принесла робота [Клода Шеннона](https://uk.wikipedia.org/wiki/%D0%9A%D0%BB%D0%BE%D0%B4_%D0%A8%D0%B5%D0%BD%D0%BD%D0%BE%D0%BD), опублікована в 1949 році, в якій Шеннон довів абсолютну стійкість [шифру Вернама](https://uk.wikipedia.org/wiki/%D0%A8%D0%B8%D1%84%D1%80_%D0%92%D0%B5%D1%80%D0%BD%D0%B0%D0%BC%D0%B0). У шифрі Вернама [ключ](https://uk.wikipedia.org/wiki/%D0%9A%D0%BB%D1%8E%D1%87_(%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D1%96%D1%8F)) має довжину, рівну довжині самого переданого повідомлення. Якщо кожен [біт](https://uk.wikipedia.org/wiki/%D0%91%D1%96%D1%82) ключа вибирається випадково, то розкрити шифр неможливо (тому що всі можливі відкриті тексти будуть рівноймовірними). Шифри, в яких довжина ключа менша від довжини тексту, згідно з Шенноном, не можуть бути «ідеально безпечними» [1].

З лютого 2000р. по лютий 2003р. проходив європейський дослідницький проект NESSI (New European Schemes of Signatures, Integrity and Encryption) для визначення безпечних алгоритмів шифрування. На цьому проекті було представлено лише 6 потокових шифрів, причому наприкінці конкурсу жоден з них не був схвалений як такий, котрий міг би задовольнити всім вимогам. Саме цей проект став поштовхом до оголошення нового європейського конкурсу, присвяченого виключно потоковим шифрам. Цим конкурсом став проект eSTREAM. На конкурс eSTREAM було пред’явлено 34 шифри, що порівняно з конкурсом NESSI було великим кроком уперед.

Головною метою конкурсу було отримання шифру широкого використання, котрий працював би швидше за AES (у режимі лічильника), але був би вільним від його.

**Актуальність роботи**. В процесі створення систем шифрування гостро постає проблема знаходження компромісу між високою швидкістю роботи та низькими вимогами до використання обчислювальних ресурсів. Саме потокові шифри виступають таким компромісом – питання побудови стійких і водночас ефективних потокових систем шифрування знаходиться в центрі уваги сучасної криптографічної спільноти. Підтвердженням цього є проведення у 2005-2008р.р. у рамках проекту ECRYPT загальноєвропейського конкурсу eSTREAM, присвяченого виключно потоковим шифрам. Цей конкурс став найвизначнішою подією у розвитку теоретичної та прикладної криптографії останнього десятиліття. eSTREAM сприяв пожвавленню науково-практичної роботи в галузі розробки та криптоаналізу потокових шифрів. Вивчення, дослідження, осмислення результатів цього проекту – одна з актуальних задач сучасної криптографії.

**Мета і завдання дослідження**. Метою роботи є опис та аналіз деяких сучасних потокових шифрів із самосинхронізацією, огляд та імплементація існуючих криптоатак на них.

*Об’єкт дослідження*: алгоритми потокового шифрування із самосинхронізацією, що були представлені на проекті eSTREAM.

*Предмет дослідження*: опис та аналіз деяких шифрів-учасників проекту eSTREAM за їх характеристиками, особливостями будови, швидкодією, огляд та імплементація криптоатак на ці шифри; виявлення на цій основі сучасних тенденцій розвитку потокового шифрування.

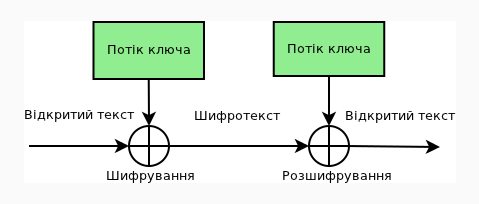
**Практичне значення одержаних результатів.** Отримані результати можна використовувати, як аналіз та інструкцію по реалізації деяких потокових шифрів із самосинхронізацією та криптоатак для них.

# 1 ОЗНАЧЕННЯ НЕОБХІДНИХ В РОБОТІ ТЕРМІНІВ ТА ПОНЯТЬ

## Потокові шифри

Потоковий шифр – це симетричний шифр, в якому кожен символ ВТ перетворюється в символ ШТ в залежності не тільки від вибраного ключа, але від розміщення даного символу в потоці ВТ.

Схематичне зображення потокового шифру представлене на малюнку.



Генератор гами видає ключовий потік (гаму): . Позначимо потік бітів ВТ . Тоді потік бітів ШТ отримують з допомогою застосування операції XOR: ,  , де . Розшифрування здійснюється операцією XOR між тою ж гамою і зашифрованим текстом: .

Якщо послідовність бітів гамми не має періоду і обирається випадково, то «зламати» шифр неможливо. Але ключі з довжиною близькою до довжини ВТ, важко використовувати на практиці. Саме через це застосовують ключі меншої довжини (наприклад, 128 біт). З його допомогою генеруються псевдовипадкова гамуюча послідовність. Саме псевдовипадковість гами може бути використана при атаці на потоковий шифр.

## Класифікація потокових шифрів

Якщо при передачі по каналу зв’язку виникло спотворення одного знаку ШТ, то в цьому випадку всі знаки, прийняті без спотворення, будуть розшифровані правильно. Буде втрата лише одного знака тексту. А у випадку, коли один із знаків ШТ при передачі по каналу зв’язку був втрачений, то це спричинить невірне розшифрування всього тексту, що слідує за втраченим знаком.

В усіх каналах передачі даних є шум. Тому для запобігання втрат інформації вирішують проблему синхронізації шифрування і розшифрування тексту. За способом рішення цієї проблеми шифросистеми поділяються на синхронні і системи із самосинхронізацією.

## Синхронні потокові шифри

*Означення:*

Синхронні потокові шифри (СПШ) — шифри, в яких потік ключів генерується незалежно від ВТ і ШТ. При шифруванні генератор потоку ключів видає біти потоку ключів, які ідентичні бітам потоку ключів при дешифруванні. Втрата знаку ШТ приведе до порушення синхронізації між цими двома генераторами і неможливості розшифрування залишкової частини повідомлення. В цій ситуації відправник і адресат повинні повторно синхронізуватися для продовження роботи.

Зазвичай синхронізація здійснюється вставкою в передане повідомлення спеціальних маркерів. Як результат, пропущений при передачі знак призводить до невірного розшифрування лише до тих пір, поки не буде прийнято один із маркерів.

Виконуватись синхронізація повинна так, щоб жодна частина потоку ключів не була повторена. Тому переводити генератор в більш ранній стан не має сенсу.

Позитивні сторони СПШ:

* відсутність ефекту розповсюдження помилок (тільки створений біт буде розшифрований невірно);
* убезпечують від будь-яких вставок і видалення частин ШТ, адже вони спричинять втрату синхронізації і будуть виявлені.

Негативні сторони СПШ:

* Уразливі до змін окремих бітів ШТ. Якщо Криптоаналітику відомий ВТ, то може змінити ці біти так, щоб вони розшифровувались, як йому це потрібно.

## Потокові шифри із самосинхронізацією

*Означення:*

Потокові шифри із самосинхронізацією (асинхронні потокові шифри (АПШ)) – шифри, в яких ключовий потік створюється функцією ключа і фіксованим числом знаків ШТ.

Внутрішні стани генератора потоку ключів є функцією попередніх N бітів ШТ. Саме тому розшифровуючий генератор потоку ключів, прийнявши N бітів, автоматично синхронізується із шифруючим генератором.

Реалізація цього режиму проходить наступним чином: кожне повідомлення починається випадковим заголовком довжини N бітів; заголовок шифрується, передаєтся і розшифровується; розшифрування невірне, але після цих N біт обидва генератори будуть синхронізовані.

Позитивні сторони АПШ:

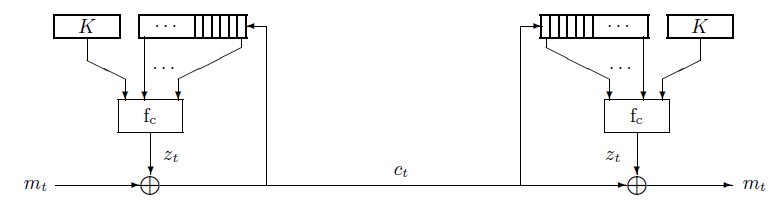
* Змішування статистики ВТ. Оскільки кожен знак ВТ впливає на наступний ШТ, статистичні властивості ВТ розповсюджується на весь ШТ. АПШ може бути стійкішим до атак на основі збитковості ВТ, чим СПШ.

Негативні сторони АПШ:

* Розповсюдження помилки (кожному неправильному біту ШТ відповідають N помилок у ВТ);

чутливі до злому через повторну передачу.

Схематичне спрощене зображення потокового шифру із самосинхронізацією:



За такого дизайну шифру, наступний знак (біт) ключового потоку  повністю визначається останніми знаками (бітами) ШТ та ключем К. Модель: на вхід деякої функції приходить стан регістру зсуву, що містить символів ШТ, та ключ, як результат отримуємо потоковий символ. Для перших  символів ВТ або ШТ немає, їх емулюють з допомогою ініціалізуючих символів, що називаються вектором ініціалізації IV.

## Атака на основі вибраного ШТ

Це модель атаки в криптоаналізі, яка передбачає збір інформації крипто аналітиком характерним чином: для обраного ШТ є змога отримати відповідний йому ВТ без додаткових відомостей про секретний ключ. По суті, противник може використовувати систему шифрування для розшифрування необхідного ШТ для отримання відповідного ВТ. Ці часткові відомості можуть надати змогу противнику відновити секретний ключ.

*Приклад.*

Нехай генерал А надсилає повідомлення генералу Б, використовуючи шифр Віженера. Криптоаналітик якимсь чином втрутився в їхній канал зв’язку і замінив зашифроване повідомлення на вибраний ним набір літер («вибраний ШТ»), нехай це буде повідомлення «NLLCJOVFXXHMLY». Генерал Б розшифровує його і в результаті отримує наступне «AKRUWNBXKWNEYX», що для нього не несе жодного змісту. Вважаючи, що ця інформація не має жодної цінності, він телефонує по незакритому каналу генералу А та запитує: «Що Ви мали на увазі під AKRUWNBXKWNEYX? Ви змінили секретний ключ без мого відому?». В цей час Криптоаналітик підслуховує цю розмову і зміг встановити відповідність, що ШТ «NLLCJOVFXXHMLY» відповідає ВТ «AKRUWNBXKWNEYX». Враховуючи використання шифру Віженера, одного розшифрованого повідомлення достатньо, щоб відновити секретний ключ.

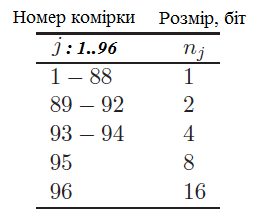
# 2 ОГЛЯД ТА АНАЛІЗ ШИФРІВ

## 2.1 Шифр Moustique



Рис. 1 Схема за шифрування-розшифрування шифру Moustique

Ключ K – 96 бітів подається на вхід спеціального регістру зсуву CCSR, що складається із 96 комірок, розмір кожної з комірок в бітах задано таблицею:



Пояснення позначень, що будуть використовуватись далі:



Розглянемо схему самого CCSR:



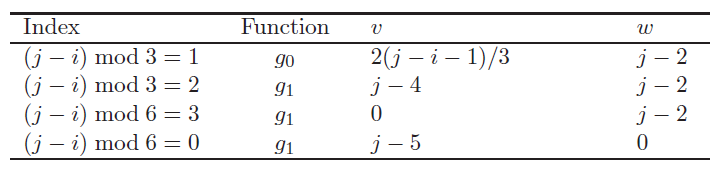
Розглянемо як заповнюється регістр на прикладі кожної комірки та кожного її біта.

Для :

Для є загальна формула:

*,*

де , а значення та всіх комбінацій визначені в таблиці:



Для прикладу розглянемо, як отримують значення деякі біти цих комірок:

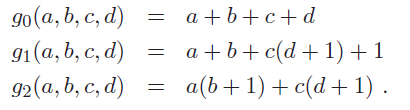
…

…

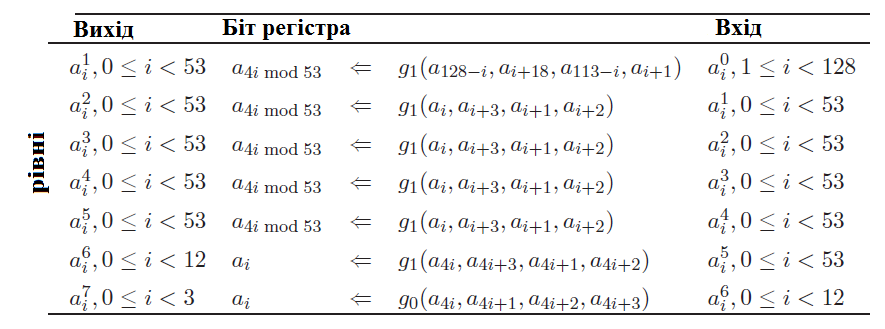
Для є окрема формула:

Для прикладу, обрахуємо випадок, коли :

Що ми забули визначити – це функції :



Далі йдуть інші рівні шифру – інші регістри. Їх заповнення теж визначено таблицею:



, де *i* – номер біту регістра *k-го –* рівня.

Приклад: 11-й біт, регістру 1-го рівня, знаходимо, коли

,

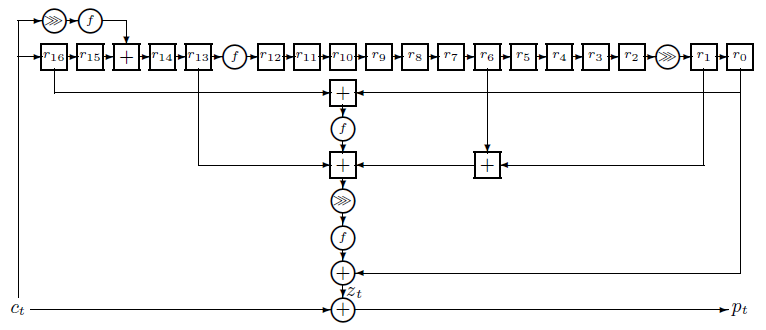
де – це біти CCSR

Аналогічно рахуємо за схемою, зображеній в таблиці вище, для всіх рівнів. Останній 7-й рівень має лише 3 біти.

Біт шифр тексту , відповідно

## 2.2 Опис шифру SSS

Розглянемо аспекти дизайну даного шифру з огляду на те, як саме реалізовуватиметься атака. Схематично формування потокових символів можна відобразити так:



Основні операції:

* - виключне або (XOR);

 - звичайне додавання;

- циклічний зсув на 8 бітів вправо;

Внутрішній стан шифру SSS складається із регістру зсуву , що складається із 17 слів, по 16 біт кожне. Основним будівельним блоком є деяка функція *f*, що залежить від ключа, а по суті являється перестановкою бітів в 16-бітному слові. Формульно її можна представити так:

,

де  - старший байт слова (MSB) ***x***, а SBox – залежна від ключа функція, що визначається на етапі встановлення ключа. Вважатимемо, що це таблиця із 256 16-бітних слів.

## 2.3 Опис атаки на основі вибраного ШТ на SSS

З опису шифру випливає, що секретним ключем для шифрування можна вважати саме таблицю Sbox із 256 16-бітних слів. Тому головною ціллю атаки є відтворити цю таблицю.

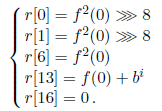
Ми будемо розшифровувати один ШТ, що складається із 263 однакових патернів, отримаємо відповідний йому ВТ та ключовий потік як результат. Патерни  складатимуться із 18 16-бітних слів та завжди має наступний формат:



де  має певне значення для кожного патерну, визначимо це значення згодом. Із схематичного зображення шифру випливає, що



Значення цих регістрів в момент часу *t = 18*:



По суті, вміст регістрів сталий для всіх патернів, окрім регістру r[13]. Зробимо деяку заміну:

,

в результаті отримаємо наступний вираз:



Важливо зауважити, що доданки в дужках – це 2-байтові слова, в яких ми будемо позначати старші байти (MSB) відповідно  та , молодші байти -  та . Щоб віднайти старший байт , використаємо 8 патернів із 256 по 263, що мають настуний вигляд:



Виходячи із вигляду патернів вище, перетворимо формулу для потокового символу на 18 часовому кроці:



здійснимо операцію XOR для часового кроку 18 та 0 і отримаємо наступне:



Виходячи із результату останньої операції, можна судити про значення :

 дорівнює , то відповідний біт  буде 0, інакше – 1.

Як тільки  відновлено, то можна спробувати відновити цілком усю таблицю, що відповідає Sbox. Надалі це здійснюватиметься простим перебором: вгадуємо значення  та  - загалом 24 біти, тобто складність такого перебору 224. Зробимо це з допомогою патернів від 0 до 255. Формат у них наступний:

 та  для  Проробимо аналогічні дії як для :



Враховуючи, що атака на основі обраного ШТ, тому ми завжди можемо порівняти вгадані значення із істинними, розшрифровуючи за вгаданою таблицею Sbox.



Якщо пройтись по всім 256 патернам, можемо відновити всю таблицю.

# ДОДАТОК 1 – ПРОГРАМНИЙ КОД

// SSSproject.cpp.

#include "stdafx.h"

#include <stdlib.h>

#include "sss.h"

#include "ssssbox.h"

#include "sssmultab.h"

#include <iostream>

// Number of extra patterns to recover a\_H

#define NumberOfExtraPatterns 8

/\* some useful macros -- machine independent little-endian 2-byte words \*/

#define B(x,i) ((UCHAR)(((x) >> (8\*i)) & 0xFF))

#define BYTE2WORD(b) ( \

(((WORD)((b)[1]) & 0xFF)<<8) | \

(((WORD)((b)[0]) & 0xFF)) \

)

#define WORD2BYTE(w, b) { \

(b)[1] = B(w,1); \

(b)[0] = B(w,0); \

}

#define XORWORD(w, b) { \

(b)[1] ^= B(w,1); \

(b)[0] ^= B(w,0); \

}

/\* key-dependent Sbox \*/

WORD

Sfunc(UCHAR \*Key, int KeyLength, WORD w)

{

register int i;

WORD t;

UCHAR b;

t = 0;

b = HIGHBYTE(w);

for (i = 0; i < KeyLength; ++i) {

b = ftable[b ^ Key[i]];

t ^= ROTL(Qbox[b], i);

}

return ((b << (WORDBITS-8)) | (t & LOWMASK)) ^ (w & LOWMASK);

}

/\* cycle the contents of the shift register \*/

static void

cycle(sss\_ctx \*State, WORD ctxt)

{

register int i;

for (i = 0; i < N-1; ++i)

State->ShiftRegister[i] = State->ShiftRegister[i+1];

State->ShiftRegister[16] = ctxt;

State->ShiftRegister[14] += SBoxFromWord(State, ROTR(ctxt, 8));

State->ShiftRegister[12] = SBoxFromWord(State, State->ShiftRegister[12]);

State->ShiftRegister[1] = ROTR(State->ShiftRegister[1], 8);

}

/\*

\* Return a non-linear function of some parts of the register.

\* The positions of the state bytes form a maximal span full positive

\* difference set, and are 0, 1, 6, 13, 16.

\*/

static WORD

nltap(sss\_ctx \*c)

{

register WORD t;

t = c->ShiftRegister[0] + c->ShiftRegister[16];

//printf("OK? %04x\n",t);

t = SBoxFromWord(c,t) + c->ShiftRegister[1] + c->ShiftRegister[6] + c->ShiftRegister[13];

t = ROTR(t, 8);

//printf("inputlastf %04x\n",t);

return SBoxFromWord(c,t) ^ c->ShiftRegister[0];

}

#if SBOX\_PRECOMP

/\*

\* Precompute the key-dependent Sbox for later efficiency.

\*/

void

sss\_key(sss\_ctx \*State, UCHAR Key[], int KeyLength)

{

int i;

if (KeyLength > MAXKEY)

abort();

for (i = 0; i < 256; ++i)

State->SBox[i] = Sfunc(Key, KeyLength, (WORD)(i << (WORDBITS-8))) ^ (i << (WORDBITS-8));

/\* in case no nonce... \*/

sss\_nonce(State, (UCHAR \*)0, 0);

}

#else

/\*

\* Just save the key for later Sbox on-the-fly generation

\*/

void

sss\_key(sss\_ctx \*c, UCHAR key[], int keylen)

{

int i;

if ((c->keylen = keylen) > MAXKEY)

abort();

for (i = 0; i < keylen; ++i)

c->key[i] = key[i];

/\* in case no nonce... \*/

sss\_nonce(c, (UCHAR \*)0, 0);

}

#endif /\*SBOX\_PRECOMP\*/

void

sss\_nonce(sss\_ctx \*State, UCHAR nonce[], int nlen)

{

int i;

UCHAR nb[2];

if ((nlen % WORDBYTES) != 0)

abort();

/\* first fill both registers with zeros \*/

for (i = 0; i < N; ++i)

State->ShiftRegister[i] = State->CRC[i] = 0;

/\* now process words of the nonce \*/

for (i = 0; i < nlen; i += WORDBYTES) {

nb[0] = nonce[i];

nb[1] = nonce[i+1];

sss\_decrypt(State, nb, WORDBYTES);

}

/\* now MAC N words of zeros \*/

nb[0] = nb[1] = 0;

for (i = 0; i < N; ++i) {

sss\_maconly(State, nb, WORDBYTES);

}

State->NumberOfBitsBuffered = 0;

}

/\* encryption \*/

void

sss\_enconly(sss\_ctx \*c, UCHAR \*buf, int nbytes)

{

WORD t = 0;

/\* handle any previously buffered bytes \*/

if (c->NumberOfBitsBuffered != 0) {

while (c->NumberOfBitsBuffered != 0 && nbytes != 0) {

\*buf ^= c->StreamBuf & 0xFF;

c->CipherTextBuf ^= \*buf << (WORDBITS - c->NumberOfBitsBuffered);

c->StreamBuf >>= 8;

++buf;

c->NumberOfBitsBuffered -= 8;

--nbytes;

}

if (c->NumberOfBitsBuffered != 0) /\* still not a whole word yet \*/

return;

/\* Accrue that ciphertext word \*/

cycle(c, c->CipherTextBuf);

}

/\* handle whole words \*/

while (nbytes >= WORDBYTES)

{

t = nltap(c) ^ BYTE2WORD(buf);

WORD2BYTE(t, buf);

cycle(c, t);

buf += WORDBYTES;

nbytes -= WORDBYTES;

}

/\* handle any trailing bytes \*/

if (nbytes != 0) {

c->StreamBuf = nltap(c);

c->CipherTextBuf = 0;

c->NumberOfBitsBuffered = WORDBITS;

while (c->NumberOfBitsBuffered != 0 && nbytes != 0) {

\*buf ^= c->StreamBuf & 0xFF;

c->StreamBuf >>= 8;

c->CipherTextBuf ^= \*buf << (WORDBITS - c->NumberOfBitsBuffered);

c->NumberOfBitsBuffered -= 8;

--nbytes;

}

}

}

/\* decryption \*/

void

sss\_deconly(sss\_ctx \*State, UCHAR \*CipherTextBuffer, int CipherTextLength)

{

WORD t = 0, t2 = 0;

/\* handle any previously buffered bytes \*/

if (State->NumberOfBitsBuffered != 0) {

while (State->NumberOfBitsBuffered != 0 && CipherTextLength != 0) {

State->CipherTextBuf ^= \*CipherTextBuffer << (WORDBITS - State->NumberOfBitsBuffered);

\*CipherTextBuffer ^= State->StreamBuf & 0xFF;

State->StreamBuf >>= 8;

State->NumberOfBitsBuffered -= 8;

++CipherTextBuffer;

--CipherTextLength;

}

if (State->NumberOfBitsBuffered != 0) /\* still not a whole word yet \*/

return;

/\* Accrue that ciphertext word \*/

cycle(State, State->CipherTextBuf);

}

/\* handle whole words \*/

while (CipherTextLength >= WORDBYTES)

{

t = nltap(State);

t2 = BYTE2WORD(CipherTextBuffer);

cycle(State, t2);

t ^= t2;

WORD2BYTE(t, CipherTextBuffer);

CipherTextBuffer += WORDBYTES;

CipherTextLength -= WORDBYTES;

}

/\* handle any trailing bytes \*/

if (CipherTextLength != 0) {

State->StreamBuf = nltap(State);

State->CipherTextBuf = 0;

State->NumberOfBitsBuffered = WORDBITS;

while (State->NumberOfBitsBuffered != 0 && CipherTextLength != 0) {

State->CipherTextBuf ^= \*CipherTextBuffer << (WORDBITS - State->NumberOfBitsBuffered);

\*CipherTextBuffer ^= State->StreamBuf & 0xFF;

State->StreamBuf >>= 8;

State->NumberOfBitsBuffered -= 8;

--CipherTextLength;

}

}

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

sss\_ctx State; // the State we are trying to determine

sss\_ctx StateGuess; // our guess for the State

UCHAR CipherText[36\*256 + 36\*NumberOfExtraPatterns]; // our chosen CipherText, which will be always the same to recover any secret key

UCHAR PlainText[36\*256 + 36\*NumberOfExtraPatterns]; // the decrypted PlainText corresponding to our CipherText

int i,j,CipherTextLength,correct;

int ctr\_aL, ctr\_SaL;

UCHAR aL, aH, aHg1,aHg2;

UCHAR SboxInput;

WORD SboxGuess;

WORD a, Rotated\_a, a\_plus\_i, Rotated\_a\_plus\_i;

int EverythingCorrect;

CipherTextLength = 36\*256 + 36\*NumberOfExtraPatterns;

//We use a zero key and print the actual secret key-dependent SBox - You can put any other key here.

UCHAR ZeroKey[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

sss\_key(&State, ZeroKey, 16);

// for (i=0;i<256;i++) printf("%02x \t %04x\n",i,State.SBox[i]);

// Now we choose our CipherText

for (i = 0; i < CipherTextLength; i++)

CipherText[i] = 0; // all values init to 0

for (i = 0; i < 256; i++)

CipherText[36\*i + 26] = i; // 255 modifications in LSB of r[13]

for (i = 0; i < NumberOfExtraPatterns; i++)

CipherText[256\*36 + 36\*i + 27] = i; // NumberOfExtraPatterns modifications in MSB of r[13]

// And obtain the corresponding PlainText

for (i = 0; i < CipherTextLength; i++)

PlainText[i] = CipherText[i];

sss\_deconly(&State, PlainText, CipherTextLength);

for (ctr\_aL = 0; ctr\_aL < 256; ctr\_aL++)

{

//for (ctr\_aL=0x11;ctr\_aL<0x12;ctr\_aL++){

aL = ctr\_aL;

// determine a\_H

aH = 0;

//for (i=0;i<256;i++){ // try all possible aH

for (i = 0; i < 256; i++)

{ // try all possible aH

aHg1 = i;

correct = 1;

for (j = 0; j < NumberOfExtraPatterns; j++)

{

aHg2 = aHg1 + j;

if ( (PlainText[34]^PlainText[256\*36 + 36\*j + 34]) != (aHg1^aHg2) )

correct = 0; //{correct=0; printf("probleem\n");} else printf("geen probleem!\n");

}

if (correct)

aH = aHg1; //{aH=aHg1; printf("%02x\n",aH);}

}

a = (aH<<8)^aL;

Rotated\_a = (aL<<8)^aH;

for (ctr\_SaL = 0; ctr\_SaL < 65536; ctr\_SaL++)

{

//for (ctr\_SaL=0x4cd9;ctr\_SaL<0x4cda;ctr\_SaL++){

// Set the guess back on the beginning values

for (i = 0; i < 256; i++)

StateGuess.SBox[i] = 0;

StateGuess.SBox[aL] = ctr\_SaL;

// determine the entire SBox

for (i = 1; i < 256; i++)

{

a\_plus\_i = a+i;

Rotated\_a\_plus\_i = ( (a\_plus\_i & 0xff) << 8) ^ ( (a\_plus\_i & 0xff00) >> 8);

//printf("a\_plus\_i %04x\t rotated %04x\n",a\_plus\_i,Rotated\_a\_plus\_i);

SboxInput = (aL + i); //printf("%d\n",SboxInput);

StateGuess.SBox[SboxInput] = (PlainText[35]<<8)^PlainText[34]^(PlainText[36\*i+35]<<8)^PlainText[36\*i+34]^StateGuess.SBox[aL]^Rotated\_a^Rotated\_a\_plus\_i;

}

//for (i=1;i<256;i++) printf("%02x\t Actual: %04x \t Guess: %04x\n",i,State.SBox[i],StateGuess.SBox[i]);

// Print the solution if it's the correct one:

EverythingCorrect = 1;

for (i=0;i<256;i++)

{

if (State.SBox[i] != StateGuess.SBox[i])

{

EverythingCorrect = 0;

i = 256;

}

}

if (EverythingCorrect)

{

printf("The key has been recovered entirely!\n");

for (i = 0; i < 256; i++)

std::cout<<i<<"\t"<<State.SBox[i]<<"\t"<<StateGuess.SBox[i]<<std::endl;

//printf("%02x\t Actual: %04d \t Guess: %04d\n",i,State.SBox[i],StateGuess.SBox[i]);

ctr\_aL = 256;

ctr\_SaL = 65536;

}

}

}

getchar();

return 1;

}

# ПЕРЕЛІК ПОСИЛАНЬ

1. https://uk.wikipedia.org/wiki/Потоковий\_шифр