РЕФЕРАТ

Обсяг роботи 74 сторінки, 13 ілюстрацій, 17 таблиць, 9 додатків, 17 джерел літератури.

Об’єкт дослідження – алгоритми потокового шифрування із самосинхронізацією: HBB, SSS, Moustique.

Предмет дослідження – аналіз характеристик та особливостей будови шифрів із самосинхронізацією, огляд та реалізація криптоатаки на шифр SSS; виявлення на цій основі сучасних тенденцій розвитку потокового шифрування та криптоаналізу потокових шифрів із самосинхронізацією.

ПОТОКОВІ ШИФРИ, САМОСИНХРОНІЗАЦІЯ, HBB, SSS, MOUSTIQUE, АТАКА

РЕФЕРАТ

Работа объемом 74 страниц содержит 13 иллюстраций, 17 таблиц, 9 дополнений и 17 литературных источника.

Объект исследования – алгоритмы потокового шифрования с самосинхронизацией: HBB, SSS, Moustique.

Предмет исследования – анализ характеристик и особенностей построения шифров с самосинхронизацией, обзор и реализация криптоатаки на шифр SSS; выявление на этой основе современных тенденций развития потокового шифрования и криптоанализа поточных шифров с самосинхронизацией.

ПОТОКОВЫЕ ШИФРЫ, САМОСИНХРОНИЗАЦИЯ, HBB, SSS, MOUSTIQUE, АТАКА

ABSTRACT

Work up to 74 pages contains 13 illustrations, 17 tables, 9 additions and 17 references.

Object of the research – algorithms of self-synchronizing stream ciphers: HBB, SSS, Moustique.

Subject of the research is analysis of characteristics and structural features of self-synchronizing stream, review and implementation of cryptographical attack for cipher SSS; detection on this basis main current trends in the streaming encryption and cryptanalysis of self-synchronizing stream ciphers.

STREAM CIPHERS, SELFSYNCHRONIZATION, HBB, SSS, MOUSTIQUE, DIFFERENTIAL ATTACK

Зміст

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І ТЕРМІНІВ

АПШ	Асинхронний потоковий шифр
ВТ	Відкритий текст
ГПВЧ	Генератор псевдовипадкових чисел
КА	Клітинний автомат
ЛРЗ	Лінійний регістр зсуву
НФФ	Нелінійна фільтруюча функція
СПШ	Синхронний потоковий шифр
ШТ	Шифр текст
CCSR	Назва регістр зсуву, що використовується в шифрі Moustique
IV	Вектор ініціалізації
MAC	«Message Authentication Code» – режим аутентифікації повідомлення
MSB	«Most significant bit (byte)» – старший біт (байт) залежно від контексту
MSVC	«Microsoft Visual C++» – інтегроване середовище розробки програмного забезпечення на мові C++
NESSI	«New European Schemes of Signatures, Integrity and Encryption» – європейський дослідницький проект для визначення безпечних шифрів
SSS	Потоковий шифр із самосинхронізацією із сімейства шифрів SOBER
XOR	Виключне АБО

ВСТУП

За останні десятиліття ми стали свідками швидкого розвитку інформаційних технологій, зокрема зростання цифрових сховищ зберігання інформації та обміну даними. Причиною такого динамічного руху технологій уперед можна пояснити популяризацією мережі Інтернет та бездротових мереж. Нові комунікаційні технології вимагають належного рівня технологій безпеки інформації.

Криптологія – це наука, що забезпечує інформаційний захист у сучасному цифрову світі. Зазвичай цю науку розділяють на дві сфери: криптографію та криптоаналіз. Криптографія вивчає дизайн алгоритмів та протоколів для інформаційної безпеки. Ідеальною вважається ситуація, коли можливо розробити алгоритми, які доказово стійкі до відомої множини атак, але зазвичай, таке можливо в обмеженій кількості випадків. Криптоаналіз в свою чергу займається математичними методами для обходу чи злому криптографічних примітивів.

Криптографічні алгоритми зазвичай розбивають на два сімейства: симетричні та асиметричні. Симетричні алгоритми вимагають наявності секретного ключа, що розподіляється між сторонами комунікації. Асиметричні алгоритми засновані на системах із відкритим ключем, який відомий усім сторонам, та із секретним ключем, що зберігається у секреті однією стороною.

Існує два типи симетричних алгоритмів: блочні та потокові шифри. Представники останнього класу і є предметом вивчення даної роботи.

Блочні шифри – це сфера симетричної криптографії, вивченням якої займались найбільше. Поштовхом до розвитку блочних шифрів можна вважати прийняття шифру DES, як національного стандарту шифрування в США в 1976 році. Наслідком посиленої уваги світової криптографічної спільноти до принципів побудови і роботи DES та спроб його зламу можна вважати виникнення двох нових потужних методів криптоаналізу: диференціального та лінійного криптоаналізу.

Незважаючи на успіх блочних шифрів, існує потреба в потокових шифрах, які надають нові переваги в багатьох сценаріях.

Потокові шифри на базі регістрів зсуву 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 регістрів зсуву. Пізніше Соломон Голомб, математик Агентства Національної Безпеки США, написав книгу під назвою «Shift Register Sequences» («Послідовності регістрів зсуву»), в якій виклав свої основні досягнення в цій галузі, а також досягнення Селмер.

Велику популярність потоковим шифрів принесла робота Клода Шеннона, опублікована в 1949 році, в якій Шеннон довів абсолютну стійкість шифру Вернама. У шифрі Вернама ключ має довжину, рівну довжині самого переданого повідомлення. Якщо кожен біт 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».

З лютого 2000р. по лютий 2003р. проходив європейський дослідницький проект NESSIE [13] 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 – eSTREAM. На конкурс eSTREAM було пред’явлено 34 шифри, що порівняно з конкурсом NESSI було великим кроком уперед. 2 шифри-учасники працювали в режимі із самосинхронізацією: SSS та Moustique.

Головною метою конкурсу eSTREAM було отримання шифру широкого використання, котрий працював би швидше за AES (у режимі лічильника).

В 2003 році на конференції Indocyrypt’03 [10], що проводилась в Нью-Делі, Індія, один із учасників Палаш Саркар (Palash Sarkar) представив шифр із самосинхронізацією HBB, що на відмінну від своїх попередників містив замість лінійного регістру зсуву два клітинних автомати.

Три вищезгаданих шифри: HBB, SSS та Moustique як одні із найновіших представників потокових шифрів із самосинхронізацією представляють значний інтерес для дослідження. 2003-2004 роки можна вважати початком зародження потокових шифрів із самосинхронізацією, тому враховуючи їх незначну кількість, представлених криптографічній спільноті, можна із впевненістю сказати, що цей напрямок дуже перспективний.

Режим із самосинхронізацією для потокових шифрів подібний до режиму зворотного зв'язку за ШТ блочних шифрів. Саме тому для вищезгаданих шифрів уже існують атаки, зокрема атаки на основі вибраного ШТ. Детальне вивчення, реалізація потокових шифрів із самосинхронізацією шифрів та аналіз криптоатак для них є необхідним фундаментом для реалізації нових шифрів стійких до даних атак чи модифікації уже існуючих.

Актуальність роботи. В процесі створення систем шифрування гостро постає проблема знаходження компромісу між високою швидкістю роботи та обмеженнями на об'єм використовуваних обчислювальних ресурсів. Саме потокові шифри із самосинхронізацією дозволяють досягнути такого компромісу. 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 та щорічних криптографічних конференцій. Вивчення, дослідження, осмислення результатів проектів eSTREAM та інших заходів, присвячених потоковим шифрам – одна з актуальних задач сучасної криптографії.

Мета і завдання дослідження. Метою роботи є дослідження та аналіз сучасних потокових шифрів із самосинхронізацією, зокрема HBB, SSS та Moustique, огляд та реалізація криптоатаки для шифру SSS.

Об'єкт дослідження: алгоритми потокового шифрування із самосинхронізацією HBB, SSS, Mostique.

Предмет дослідження: аналіз характеристик та особливостей будови шифрів із самосинхронізацією , огляд та реалізація криптоатаки на шифр SSS; виявлення на цій основі сучасних тенденцій розвитку потокового шифрування та криптоаналізу потокових шифрів із самосинхронізацією.

Наукова новизна результатів полягає в тому, що реалізована і обґрунтована модифікація атаки на один із шифрів із самосинхронізацією, здійснена порівняльна характеристика атак на основі обраного ШТ для двох шифрів із самосинхронізацією.

Практичне значення одержаних результатів. Враховуючи той факт, що потокові шифри із самосинхронізацією є недосконалими та новими, то результати аналізу можна використовувати для покращення їх проектування та будови.

1 ОЗНАЧЕННЯ НЕОБХІДНИХ В РОБОТІ ТЕРМІНІВ ТА ПОНЯТЬ

1.1 Базові поняття

Текст – це послідовність букв деякого алфавіту. Надалі вважатимемо, що алфавіт є скінченним. Позначимо його як $Z_m = \{z_1,...,z_m\}$. Відкритий текст (ВТ) - це текст, що підлягає шифруванню. Відповідно, шифр текст (ШТ) – текст, що підлягає розшифруванню.

Криптографічна система – це набір апаратних і програмних засобів, інструкцій і правил, за допомогою яких, використовуючи криптографічні перетворення, можна зашифрувати повідомлення і розшифрувати криптограму різними способами, один із яких вибирається за допомогою секретного ключа, а також здійснювати інші криптографічні протоколи.

Криптографічна стійкість – у широкому розумінні це – здатність криптосистеми або криптоалгоритму протистояти атакам з використанням методів криптоаналізу; у вузькому розумінні – чисельна характеристика складності розкриття криптографічного алгоритму з урахуванням тих науково-технічних методів та засобів, які може використати криптоаналітик.

Стійкість (за Шенноном) розділяється на два типи: теоретична та практична.

- Теоретична стійкість – стійкість криптосистеми за наявності у криптоаналітика необмеженого часу, необмежених обчислювальних ресурсів, якнайкращих методів криптоаналізу.
- Практична стійкість – стійкість криптосистеми на теперішній час з урахуванням того, що криптоаналітик володіє обмеженим часом, обмеженими обчислювальними ресурсами і сучасними методами криптоаналізу.

1.2 Основні типи криптографічних атак залежно від типу відомої інформації

Атаки на криптографічні системи, алгоритми бувають різної складності, практично реалізовані та теоретичні. Характер атаки залежить від багатьох факторів, тому важливо структурувати атаки за певними правилами чи ієрархією. Скористаємось типізацією атак залежно від типу інформації, що відома криптоаналітику. Дана типізація запропонована в [1].

1. Атака на основі (з використанням) тільки ШТ. У криптоаналітика є ШТ декількох повідомлень, зашифрованих одним і тим же алгоритмом шифрування. Задача криптоаналітика полягає в розкритті ВТ як можна більшого числа повідомлень або, що краще, отриманні ключа (ключів), використаного для шифрування повідомлень з метою дешифрування також і інших повідомлень, зашифрованих тими ж ключами.
2. Атака на основі (з використанням) ВТ. У криптоаналітика є доступ не тільки до ШТ декількох повідомлень, але і до відповідних відкритих текстів цих повідомлень. Його задача полягає в отриманні ключа (або ключів), використаного (використаних) для шифрування повідомлень з метою дешифрування інших повідомлень, зашифрованих тим же ключем (ключами).
3. Атака на основі вибраного відкритого тексту. У криптоаналітика не тільки є доступ до ШТ і відповідних відкритих текстів декількох повідомлень, але є і можливість вибирати відкритий текст (тексти) і отримати зашифрований. Це надає більше варіантів, ніж атака з використанням відкритого тексту, оскільки криптоаналітик може вибирати шифровані блоки відкритого тексту із спеціальними властивостями, що може надати більше інформації про ключ. Його задача полягає в отриманні ключа (або ключів), використаного для шифрування повідомлень, або алгоритму, що дозволяє дешифрувати нові повідомлення, зашифровані тим же ключем (або ключами).

4. Адаптивна атака з використанням відкритого тексту. Криптоаналітик не тільки може вибирати тексти для шифрування, але також може будувати свій подальший вибір текстів на базі одержаних результатів шифрування. При розкритті з використанням вибраного відкритого тексту криптоаналітик міг вибрати для шифрування тільки один великий блок відкритого тексту, при адаптивному розкритті з використанням вибраного відкритого тексту він може вибрати менший блок відкритого тексту, потім вибрати наступний блок, використовуючи результати першого вибору і так далі. Атаки 2-4 можливі, наприклад, при шифруванні з відкритим ключем.
5. Атака на основі вибраного ШТ. Криптоаналітик може вибрати різні ШТ для розшифрування і має доступ до розшифрованих відкритих текстів (наприклад, криптоаналітик має доступ до апарату-шифратора).
6. Адаптивна атака на основі вибраного ШТ (аналогічно п.4).
7. Атака на основі вибраного тексту (адаптивна) – об'єднує можливості атак п.3, п.5 (п.4, п.6).

Атаки в цьому списку з більшим номером сильніші і небезпечніші ніж з меншим. Для всіх сучасних шифраторів обов'язкова вимога – стійкість до атак типу 1 і 2. Якщо у криптоаналітика є деяка інформація про ключі або про зв'язок між різними ключами, то напади на криптосистему стають ще небезпечнішими.

В даній роботі буде розглянуто реалізацію атаки на основі вибраного ШТ. Розглянемо простий приклад-пояснення цієї атаки для кращого її розуміння.

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.

1.3 Блочні шифри

У сучасних шифраторах ВТ і ШТ записуються у двійковому алфавіті. При блочному шифруванні ВТ поділяється на блоки завдовжки n, тобто кожен блок можна розглядати як двійковий вектор довжини n (якщо число знаків у ВТ не кратне довжині блоку, то останній блок доповнюється зарані обумовленими символами, наприклад, нулями). Таким чином, блочний шифр являє собою взаємно-однозначне перетворення множини двійкових векторів довжини n у себе, тобто не що інше, як підстановку на алфавіті з 2^n символів. Кількість можливих підстановок на такому алфавіті (кількість можливих варіантів зашифрування) дорівнює $(2^n)!$. У сучасних блочних шифраторах довжина блоку n досить велика (частіше за все 64, 128 або 256 бітів), тому алфавіт виходить величезним, а кількість підстановок на ньому – тим більше. Довжина ключа, який задавав би довільну підстановку з цієї множини (шифр простої заміни), дорівнює $\log(2^n)!$ біт і є надто великою, щоб таким шифром можна було користуватися на практиці. Наприклад, вже при $n=16$ $\log_2(2^n!)\approx 10^6$. Тому практично шифри реалізують більш вузькі множини підстановок, що мають, однак, ряд необхідних криптографічних властивостей.

Існує декілька різних режимів роботи блочних шифраторів, які відповідають певним практичним потребам і мають свої переваги й недоліки.

Надалі будемо позначати через C_i i-й блок ШТ, а P_i – i-й блок відповідного йому відкритого тексту $i=1,2,\ldots$.

1.4 Режими роботи блочних шифраторів

Розглянемо базові режими роботи блочних шифрів, оскільки деякі режими мають спільні характеристики із відповідними їм серед потокових шифрів. Коротка характеристика даних режимів запозичена із [1].

1. Режим ECB (Electronic Code Book) – режим електронної кодової книги

Це найпростіший режим роботи блочних шифраторів. Рівняння шифрування та розшифрування в цьому режимі задаються співвідношеннями:

$$C_i = E_K(P_i) \qquad (1.1)$$
$$P_i = D_K(C_i),$$

де E_K, D_K – операції шифрування та розшифрування блочним алгоритмом з ключем K.

Власне, тільки в режимі *ECB* блочні шифратори реалізують алгоритми блочного шифрування. В решті режимів вони породжують залежність блоку ШТ від його місця, а також, можливо, й від інших блоків, що притаманне потоковим шифрам.

2. Режим CBC (Cipher Block Chaining) – режим зчеплення блоків ШТ

Рівняння шифрування та розшифрування:

$$C_i = E_K(P_i \oplus C_{i-1}), \qquad (1.2)$$
$$P_i = C_{i-1} \oplus D_K(C_i).$$

C_0 – так звана синхропосилка – є спільною для відправника й одержувача і може бути як секретною, так і відкритою.

3. Режим CFB (Cipher Feed Back) – режим зворотного зв’язку за шифрованим текстом

Рівняння шифрування та розшифрування:

$$C_i = P_i \oplus E_K\left(C_{i-1}\right) \qquad (1.3)$$
$$P_i = C_i \oplus E_K(C_{i-1})$$

Блок C_0 називається синхропосилкою і є спільним для відправника та одержувача.

Властивості цього режиму схожі на властивості режиму *CBC*. Блок ВТ впливає на всі наступні блоки ШТ, в той час як спотворення або втрата блоку ШТ призводить до помилок при розшифруванні тільки у двох послідовних блоках ВТ, тобто режим є таким, що самосинхронізується. Розпаралелювання можливе тільки при розшифруванні. Цей режим має ще одну перевагу: для шифрування і для розшифрування використовується одна й та сама функція E_K.

4. Режим OFB (Output Feed Back) – режим зворотного зв’язку за виходом

Рівняння шифрування та розшифрування:

$$C_i = P_i \oplus E_K(S_{i-1}) \qquad (1.4)$$
$$P_i = C_i \oplus E_K(S_{i-1})$$

$S_i = E_K(S_{i-1})$, де S_0 – синхропосилка (не є секретною).

5. Режим лічильника (Counter)

Деякий пристрій (лічильник) генерує послідовність блоків S_i за правилом:

$$S_i = f(S_{i-1}), \tag{1.5}$$

де S_0 – синхропосилка .

Функція f звичайно проста, наприклад:

$$S_i = S_{i-1} + 1 \tag{1.6}$$

(звідси і назва – лічильник). Те, що послідовні блоки, які генерує лічильник, мало відрізняються, не впливає на стійкість. Адже у доброму блочному шифраторі при зміні одного біта на вході, на виході змінюється в середньому половина бітів.

Рівняння шифрування та розшифрування:

$$\begin{aligned} C_i &= P_i \oplus E_K(S_{i-1}) \\ P_i &= C_i \oplus E_K(S_{i-1}) \end{aligned} \tag{1.7}$$

У режимах *OFB* та лічильника блочні шифратори працюють як потокові шифри адитивного типу: шифруюча послідовність (гама) складається побітово за $\mathrm{mod}\,2$ з ВТ. При цьому гама генерується незалежно від ВТ.

1.5 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.

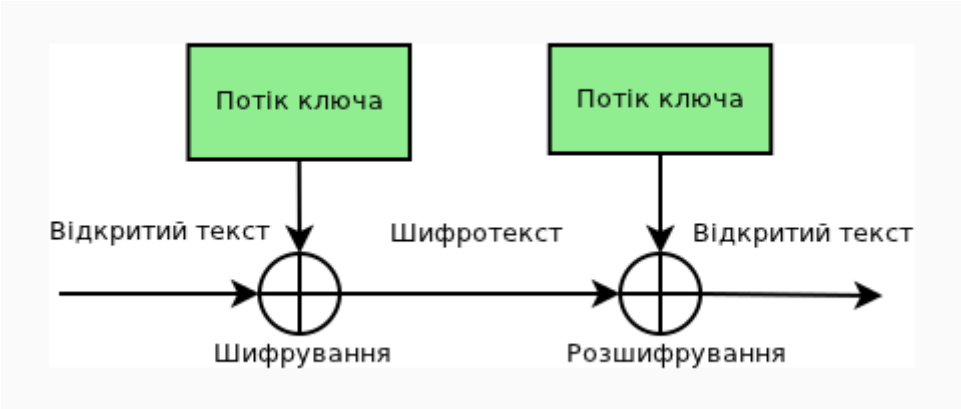


Рисунок 1.1 – Схематичне зображення потокового шифру

Послідовність $k_1, k_2, k_3, \dots, k_L$ називається гаммою, а шифри, в яких ШТ $c_1, c_2, c_3, \dots, c_L$ одержується шляхом додавання знаків ВТ $m_1, m_2, m_3, \dots, m_L$ та ключової послідовності

$$c_i = m_i \oplus k_i \tag{1.8}$$

називаються шифрами гамування.

Генератор гамми видає ключовий потік (гамму): $k_1, k_2, k_3, \dots, k_L$.

Розшифрування здійснюється операцією XOR між тою ж гаммою і зашифрованим текстом:

$$m_i = c_i \oplus k_i \tag{1.9}$$

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.

На відміну від блочних систем, у потокових системах шифрування відкритого тексту виконується посимвольно, причому перетворення, за допомогою якого здійснюється шифрування символу, може залежати від часу (тобто від місця даного символу у послідовності ВТ). Крім того, шифруюче перетворення може мати пам’ять, тобто залежати від попередніх символів тексту.

Таким чином, при блочному шифруванні один і той самий блок символів ВТ шифрується однаково незалежно від його розташування. Це дозволяє зловмиснику непомітно для отримувача спотворювати інформацію, виключати її частину або ж вводити хибну інформацію. Очевидно, що внаслідок залежності шифруючого перетворення від часу потокові системи вільні від цього недоліку. Вказана перевага потокових систем шифрування породжує, однак, проблему синхронізації. Помилка при передачі (втрата або спотворення знаку) може призвести до того, що весь подальший ШТ буде розшифрований неправильно. В залежності від того, як розв’язується ця проблема, потокові шифратори розділяють на синхронні системи та системи з самосинхронізацією.

1.6 Синхронні потокові шифри

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.

1.7 Потокові шифри із самосинхронізацією

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,

передається 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.

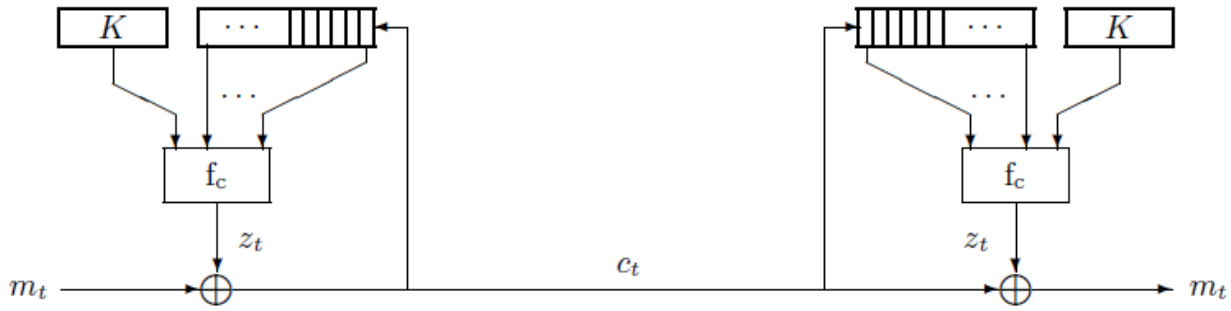


Рисунок 1.2 – Схематичне спрощене зображення потокового шифру із самосинхронізацією

За такого дизайну шифру, наступний знак (біт) ключового потоку z_t повністю визначається останніми n_mзнаками (бітами) ШТ та ключем К. Модель: на вхід деякої функції f_c приходить стан регістру зсуву, що містить n_m символів ШТ, та ключ, як результат отримуємо символ ключової послідовності. Для перших n_m символів ВТ ще немає відповідного ШТ, їх задають з допомогою n_m ініціалізуючих символів, що називаються вектором ініціалізації IV.

Потокові системи з самосинхронізацією дають можливість правильно розшифровувати не тільки у випадку спотворення знаку ШТ, але й у разі його втрати. Прикладом таких систем є режими роботи блочних шифраторів CBC (зчеплення блоків ШТ) та CFB (зворотного зв'язку за ШТ). Для режиму CFB, наприклад, розшифрування відбувається за правилом $P_i = C_i \oplus E_K(C_{i-1})$, де $P_i - i$-й блок ВТ, $C_i - i$-й блок ШТ, E_K - алгоритм шифрування з ключем K. При розшифруванні ШТ $..C_{i-1}, C_{i+1}, C_{i+2,}...$ з втраченим i-м блоком маємо:

$$P_i^{'} = C_{i+1} \oplus E_K(C_{i-1}) \neq P_i, \tag{1.10}$$

$$P_{i+1}^{'} = C_{i+2} \oplus E_K(C_{i+1}) \neq P_{i+1}, \text{ але } = P_{i+2}.$$

Отже, один блок ВТ спотворюється і один губиться, а далі все розшифровується правильно.

Режими ж OFB (зворотного зв'язку за виходом) та лічильника є синхронними системами. Їх особливістю є те, що на ВТ посимвольно накладається ключова послідовність (гама) $k_1, k_2, ...$, яка генерується незалежно від ВТ. Переважна більшість систем потокового шифрування є шифрами гамування.

Зазвичай замість чисто випадкової ключової послідовності використовується псевдовипадкова, що виробляється генератором псевдовипадкових чисел (ГПВЧ), який керується відносно коротким ключем. (Власне шифратор у цьому випадку і являє собою ГПВЧ). Для того, щоб було можливо застосувати псевдовипадкову послідовність $\boldsymbol{z} = z_0, z_1, z_2...$, в якості ключової, вона повинна мати добрі криптографічні характеристики, тобто бути "схожою" на випадкову. Це необхідно для того, щоб ключова послідовність для супротивника, який не знає ключа, була непередбачуваною. Тому до ГПВЧ, які використовуються у криптографії, ставляться більш жорсткі вимоги, ніж при використанні у інших сферах.

Найбільш розповсюдженими та вивченими вузлами, що складають основу багатьох ГПВЧ, в тому числі і сучасних потокових шифраторів, є регістри зсуву з лінійним зворотним зв'язком (РЗЛЗЗ) або коротше – лінійні регістри зсуву (ЛРЗ). Короткий опис ЛРЗ наведений у наступному розділі, запозичений в [1].

1.8 Лінійні регістри зсуву

ЛРЗ генерують псевдовипадкові послідовності $\boldsymbol{s} = s_0, s_1, s_2 \ldots$, що задовольняють лінійним рекурентним співвідношенням:

$$s_{n+j} = a_0 s_j + a_1 s_{j+1} + \ldots + a_{n-1} s_{j+n-1}, \quad j = 0,1,\ldots \tag{1.11}$$

для фіксованого $n > 0$. У виразі (2) члени послідовності **s**, а також коефіцієнти $a_0, a_1, \ldots, a_{n-1}$ належать скінченному полю $GF(p^m)$, а операції додавання та множення – це операції у цьому полі. Такі послідовності називають лінійними рекурентними послідовностями порядку n над скінченним полем $GF(p^m)$.

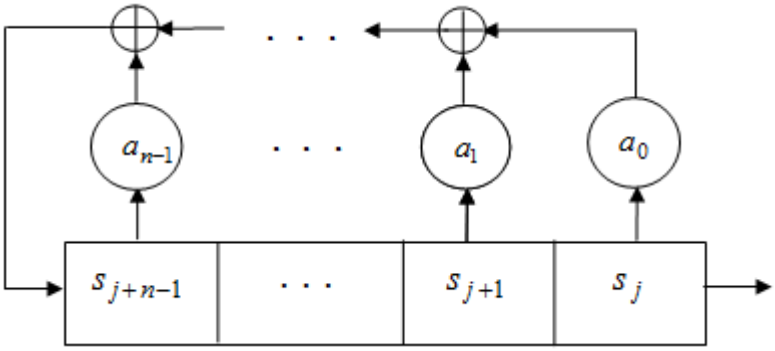


Рисунок 1.3 – Схема регістра із лінійним зсувом

Власне регістр складається з послідовності комірок, у кожній з яких записаний елемент скінченного поля $GF(p^m)$. Кількість комірок регістра n називається довжиною цього регістра, а значення $\{s_j, \ldots, s_{j+n-1}\}$, що містяться в

комірках у момент j, - станом регістра у момент j. Стан ЛРЗ у момент $j=0$ називається початковим заповненням регістра.

Регістр працює у дискретному часі. На кожному такті роботи регістра відбувається наступне: вміст кожної комірки множиться на відповідний коефіцієнт a_i, а результати додаються; далі вміст усіх комірок зсувається праворуч на одну комірку, число з крайньої правої комірки подається на вихід пристрою, а звільнена крайня ліва комірка заповнюється одержаною сумою $\sum_{i=0}^{n-1} a_i s_{j+i}$, де j – номер такту.

Якщо основним скінченним полем є GF(2), то коефіцієнти $a_i, 0 \le i \le n-1$ є нулями або одиницями і в схемі регістра можна не вказувати зйоми з комірок, яким відповідають нульові коефіцієнти, а ті, яким відповідають $a_i = 1$, замінити на провідники без множників. Таким чином, схема регістра набуває вигляду:

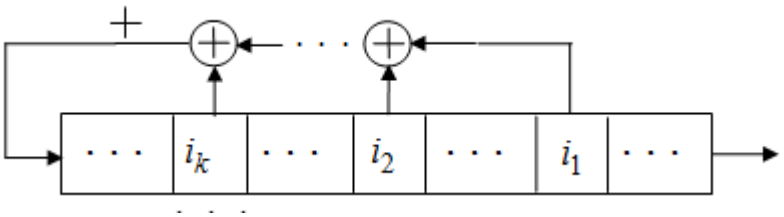


Рисунок 1.4 – Зображення лінійного регістра зсуву над полем GF(2)

де $a_{i_1} = a_{i_2} = ... = a_{i_k} = 1$, решта $a_i = 0$.

Лінійні рекурентні послідовності є періодичними. Оскільки кількість різних станів ЛРЗ скінченна, то рано чи пізно деякий стан ЛРЗ повториться, а вся подальша послідовність залежить тільки від стану регістра у даний момент. Якщо коефіцієнт $a_0 \neq 0$ (на практиці це завжди саме так, інакше можна розглядати регістри меншої довжини), то послідовності, що генеруються таким ЛРЗ, не мають передперіоду. Надалі вважатимемо, що умова $a_0 \neq 0$ виконана.

Висновки до розділу 1

В даному розділі було означено загальні поняття, що використовуються у роботі; наведений опис основних видів атак залежно від типу відомої інформації; розглянуто два типи шифрів: потокові та блочні; детально описано два підвиди потокових шифрів: синхронні та асинхронні. Для останніх, як шифрів із самосинхронізацією, було надано особливу увагу в даному розділі, оскільки шифри цього типу виступають предметом дослідження даної роботи. Також, розглянули лінійні регістри зсуву, як основну складову в побудові потокових шифрів.

2 ОГЛЯД ТА АНАЛІЗ ШИФРІВ ІЗ САМОСИНХРОНІЗАЦІЄЮ

2.1 Шифр Moustique

Попередня версія шифру називалась Mosquito. В конструкції шифру використовується нелінійна фільтруюча функція і шифрування відбувається побітово. Самосинхронізація спрацьовує після успішного отримання 105 бітів ШТ.

2.1.1 Характеристики шифру Moustique

Автор: Joan Daemen.

Параметри: шифр, що самосинхронізується, довжина ключа 96 біт, довжина IV від 0 до 13 байтів.

Основні операції: додавання та множення за модулем 2.

Конструктивні особливості: зчеплення елементів шифрованого тексту (подібно до режиму CFB у блочному шифруванні); використання регістрів з умовним доповненням.

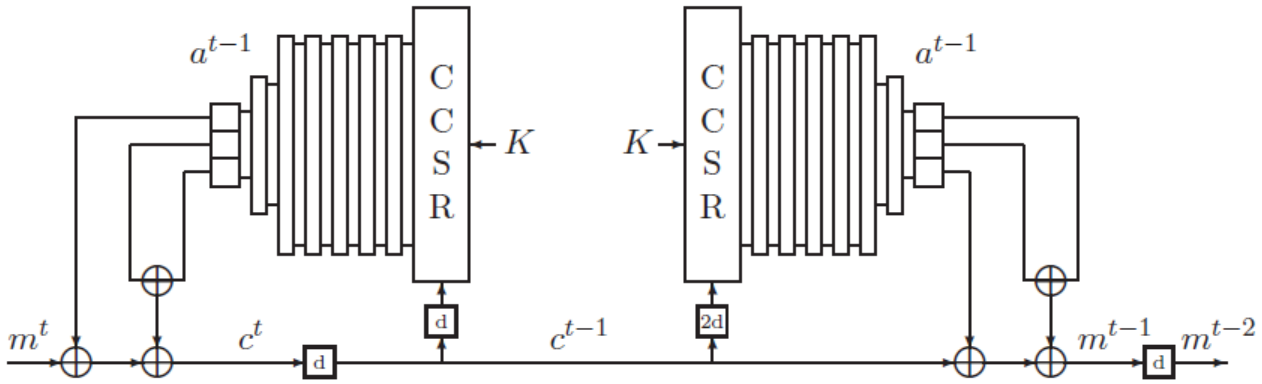
Переваги: незвичайна архітектура, можливість використання для аутентифікації повідомлення (MAC), наявність режиму, в якому шифр працює як синхронний.

Криптоатаки: знайдено корельовані шифруючі гамми [6].

Конкурс/Конференція: конкурс eStream [7].

2.1.2 Опис шифру Moustique

Алгоритм шифру складається із 7 рівнів: регістру зсуву CCSR в 128 бітів, який на вхід отримує біти ключа, 4 послідовних рівнів в 53 біти, 1 рівень в12 бітів та останній – в 3 біти. Робота шифру на кожному із рівнів відображена на Рисунку 2.1 та описана в Таблиці 2.3.



Рисинок 2.1 – Схема зашифрування–розшифрування шифру Moustique [4]

Розглянемо алгоритм роботи шифру згідно оригінально статті [4]. Ключ K – 96 бітів подається на вхід спеціального регістру зсуву CCSR, що складається із 96 комірок, розмір кожної з комірок в бітах задано таблицею:

Таблиця 2.1 – Нумерація комірок та бітів для внутрішнього регістру CSSR шифру

Номер комірки, j: 1..96	Розмір комірки n_j в бітах
1-88	1
89-92	2
93-94	4
95	8
96	16

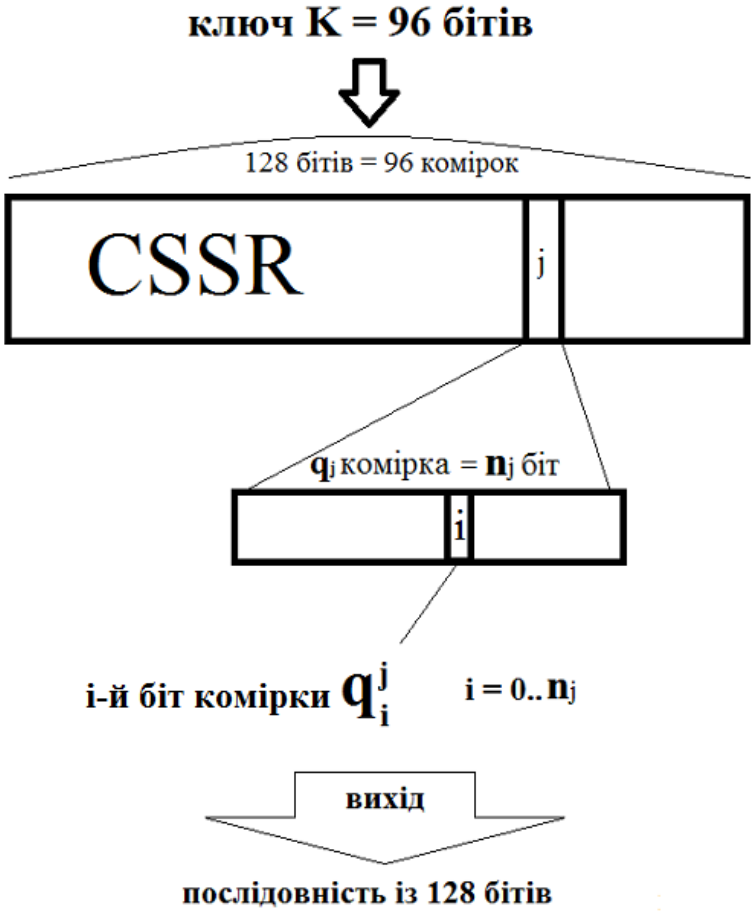


Рисунок 2.2 – Схема позначень регістру зсуву CCSR для шифру Moustique

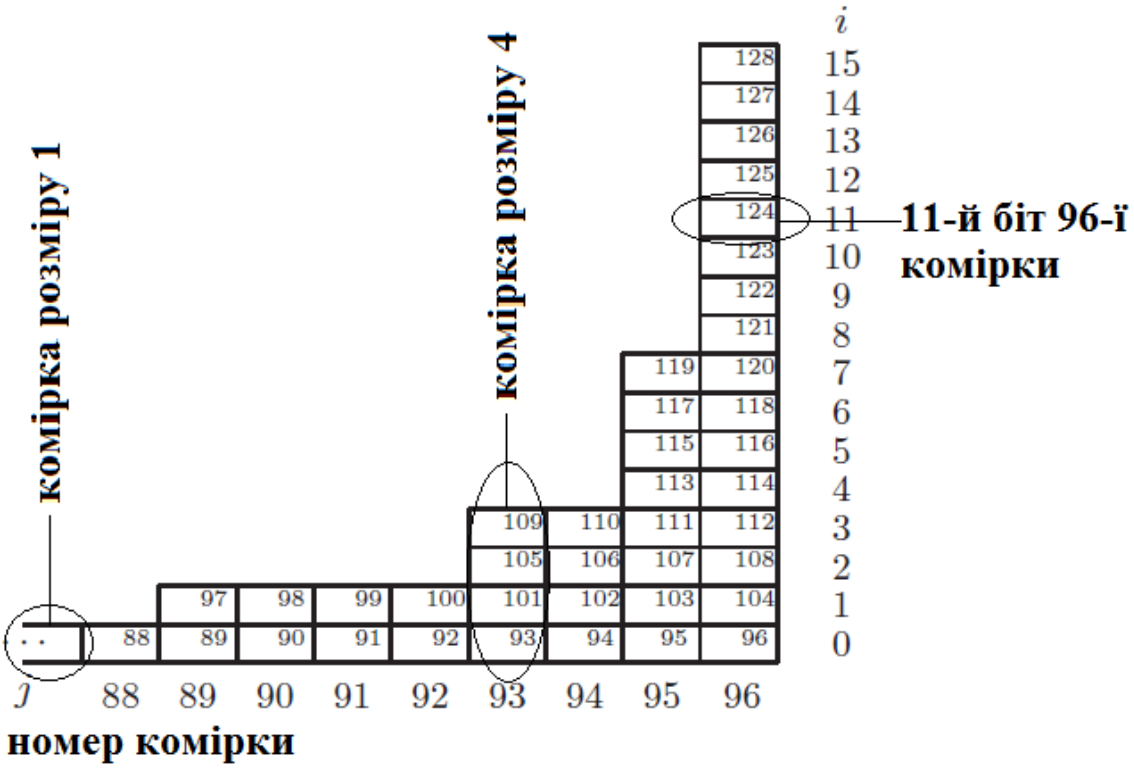


Рисунок 2.3 – Схема ЛРЗ CCSR

Розглянемо як заповнюється регістр: математичні вирази для кожної комірки та кожного її біта.

Для $j \leq 2$:

$$(i,j) = (0,1) : \ q_0^1 \coloneqq \ g_0(1, k_0, 0,0) \tag{2.1}$$
$$(i,j) = (0,2) : \ q_0^2 \coloneqq \ g_1(q_0^1, k_1, 0,0)$$

Для $j > 2$ та $j < 96$ є загальна формула:

$$q_i^j \coloneqq \ g_x\left(q_{i \bmod n_{j-1}}^{j-1}, k_{j-1}, q_{i \bmod n_v}^v, q_{i \bmod n_w}^w\right), \tag{2.2}$$

де $0 \leq v, w < j - 1$, а значення x, v, w та всіх комбінацій (i,j) в Таблиці 2.2.

Таблиця 2.2 – Значення v, w залежно від значення пари індексів (i,j)

Індекс	Функція	v	w
$(j-i) mod 3 = 1$	g_0	$2(j-i-1)/3$	$j-2$
$(j-i) mod 3 = 2$	g_1	$j-4$	$j-2$
$(j-i) mod 6 = 3$	g_1	0	$j-2$
$(j-i) mod 6 = 0$	g_1	$j-5$	0

Розглянемо, як отримують значення деякі біти цих комірок:

$$(i,j) = (0,3) : \ q_0^3 \coloneqq \ g_1(q_0^2, k_2, 1, q_0^1)$$
$$(i,j) = (0,4) : \ q_0^4 \coloneqq \ g_0(q_0^3, k_3, q_0^2, q_0^2)$$
$$\ldots \tag{2.3}$$

$$(i,j) = (0,89) : \ q_0^{89} \coloneqq \ g_1(q_0^{88}, k_{88}, q_0^{85}, q_0^{87})$$
$$(i,j) = (1,89) : \ q_0^{89} \coloneqq \ g_0(q_0^{88}, k_{88}, q_0^{58}, q_0^{87})$$
$$\dots$$

Для $j = 96$ є окрема формула:

$$q_i^{96} \coloneqq \ g_2\left(q_{i \ mod \ 8}^{95}, q_0^{95-i}, q_{i \ mod \ 4}^{94}, q_{i \ mod \ n_{94-i}}^{94-i}\right) \quad (2.4)$$

Обрахуємо випадок, коли $i = 14, \ q_{14}^{96}$:

$$q_{14}^{96} \coloneqq \ g_2\left(q_{14 \ mod \ 8=6}^{95}, q_0^{95-14=81}, q_{14 \ mod \ 4=2}^{94}, q_{14 \ mod \ n_{94-14=80}=14 \ mod \ 1=0}^{94-14=80}\right) =$$
$$= \ g_2\left(q_6^{95}, q_0^{81}, q_2^{94}, q_0^{80}\right) \quad (2.5)$$

Визначимо функцію g_i, для $i = 0,1,2$:

$$g_0(a,b,c,d) = a + b + c + d$$
$$g_1(a,b,c,d) = a + b + c(d+1) + 1$$
$$g_2(a,b,c,d) = a(b+1) + c(d+1)$$

Заповнення інших рівнів шифру описано в Таблиці 2.3.

Таблиця 2.3 – Правила заповнення різних рівнів шифру

Вихід рівня	Біт регістра	Значення регістра	Вхід
1	2	3	4
$a_i^1, 0 \ \le i \ \le 53$	$a_{4i \ mod \ 53}$	$g_1(a_{128-i}, a_{i+18}, a_{113-i}, a_{i+i})$	$a_i^0, 0 \ \le i \ \le 128$
$a_i^2, 0 \ \le i \ \le 53$	$a_{4i \ mod \ 53}$	$g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^1, 0 \ \le i \ \le 53$
$a_i^3, 0 \ \le i \ \le 53$	$a_{4i \ mod \ 53}$	$g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^2, 0 \ \le i \ \le 53$
$a_i^4, 0 \ \le i \ \le 53$	$a_{4i \ mod \ 53}$	$g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^3, 0 \ \le i \ \le 53$

Продовження таблиці 2.3

1	2	3	4
$a_i^5, 0 \le i \le 53$	$a_{4i \bmod 53}$	$g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^4, 0 \le i \le 53$
$a_i^6, 0 \le i \le 12$	a_i	$g_1(a_{4i}, a_{4i+3}, a_{4i+1}, a_{4i+2})$	$a_i^5, 0 \le i \le 53$
$a_i^7, 0 \le i \le 3$	a_i	$g_0(a_{4i}, a_{4i+1}, a_{4i+2}, a_{4i+3})$	$a_i^6, 0 \le i \le 12$

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

Приклад: 11-й біт, регістру 1-го рівня, знаходимо, коли $i = 16$

$$a_{11}^1: a_{4*16 mod 53=11} = g_1(a_{128-16}, a_{16+18}, a_{113-16}, a_{16+1}) = \\ = a_{11} = g_1(a_{112}, a_{34}, a_{97}, a_{17}), \quad (2.6)$$

де $a_{112}, a_{34}, a_{97}, a_{17}$ – це біти CCSR.

Аналогічні розрахунки за схемою, зображеній в Таблиці 2.3, здійснюємо для всіх рівнів. Останній 7-й рівень має лише 3 біти. Наступна формула:

$$z_t = a_0^7 + a_1^7 + a_2^7 \quad (2.7)$$

задає вираз для обрахунку символу ключової послідовності, що є завершальним етапом формування символів ключової послідовності.

2.2 Шифр SSS

2.2.1 Характеристики шифру SSS

Автори: P. Hawkes, G. Rose, M. Paddon.

Параметри: шифр, що самосинхронізується, довжина ключа до 128 біт, максимальна допустима довжина вихідної послідовності на фіксованому ключі 2^{128} 16-бітових слів; генерує за такт 16 бітів.

Основні операції: додавання за модулем 2 (XOR), додавання за модулем 2^{16}, циклічний зсув, підстановка (S-блок) 8×16.

Конструктивні особливості: є продовженням родини потокових шифрів SOBER, використана схема нелінійної фільтрації лінійного регістра зсуву над полем GF(2^{16}).

Переваги: вимагає невеликої пам’яті, можливість використання для аутентифікації повідомлення (MAC).

Криптоатаки: існує атака, що дозволяє знайти секретний ключ з вибраним ШТ довжиною менше 10 Кбайт.

Конкурс/Конференція: конкурс eStream [7].

2.2.2 Сімейство шифрів SOBER

SSS створений на основі шифрів із сімейства SOBER, що вперше були запропоновані Грегорі Роузом в 1998 році. Назва сімейства – це акронім-назва для шифрів, що засновані на регістрах зсуву, в яких 17 комірок розміром в 1 байт (Seventeen Octet Byte Enabled Register). Завданням цих шифрів було замінити слабкі до атак шифри, що застосовувались у мобільних телефонах. Початкові версії шифрів із даного сімейства містили в собі 17 байтні регістри зсуву та нелінійні фільтруючі функції виходу. Але вони легко піддавались до атак

розпізнавання. Згодом для покращення стійкості шифрів було запропоновано використовувати 2-байтні слова в якості комірок регістру замість 1-байтних.

2.2.3 Основні операції, необхідні для імплементації шифру

«⊕» – виключне або (XOR);

«+» – звичайне додавання;

«>>>» – циклічний зсув на 8 бітів вправо;

«^» – побітове «і» для 16-бітного слова;

«~» – бітова інверсія 16-бітного слова;

«a_H »– старший байт 16-бітного слова.

2.2.4 Генерація ключової послідовності

Потоковий генератор шифру SSS в основі має звичайний регістр зсуву та нелінійну фільтруючу функцію. Базові операції здійснюються з 16-бітними блоками ключової послідовності, кожен 16-бітний блок називатимемо «словом». Вектор «слів» $\sigma_t = (r_t[0], \dots, r_t[16])$ будемо називати «станом» регістру в час *t*, а стан $\sigma_0 = (r_0[0], \dots, r_0[16])$ – «початковим станом».

Тепер визначимо, яким чином здійснюється перехід до наступного стану регістру згідно [3]:

1. $r_t[0]$ залишається без змін;
2. $r_{t+1}[i] = r_t[i+1]$, для $i = 0..15$;
3. $r_{t+1}[16] = c_t$;
4. $r_{t+1}[14] = r_{t+1}[14] + f(c_t >>> 8)$;
5. $r_{t+1}[12] = f(r_{t+1}[12])$;
6. $r_{t+1}[1] = r_{t+1}[1] >>> 8$.

Опісля стан регістру знімається і передається на вхід в нелінійну фільтруючу функцію (НФФ), на виході з якої отримуємо символ ключової послідовності z_t:

$$z_t = \text{НФФ}(\sigma_t) = f((f(r[0] + r[16]) + r[1] + r[6] + r[13]) >>> 8) \oplus r[0].$$

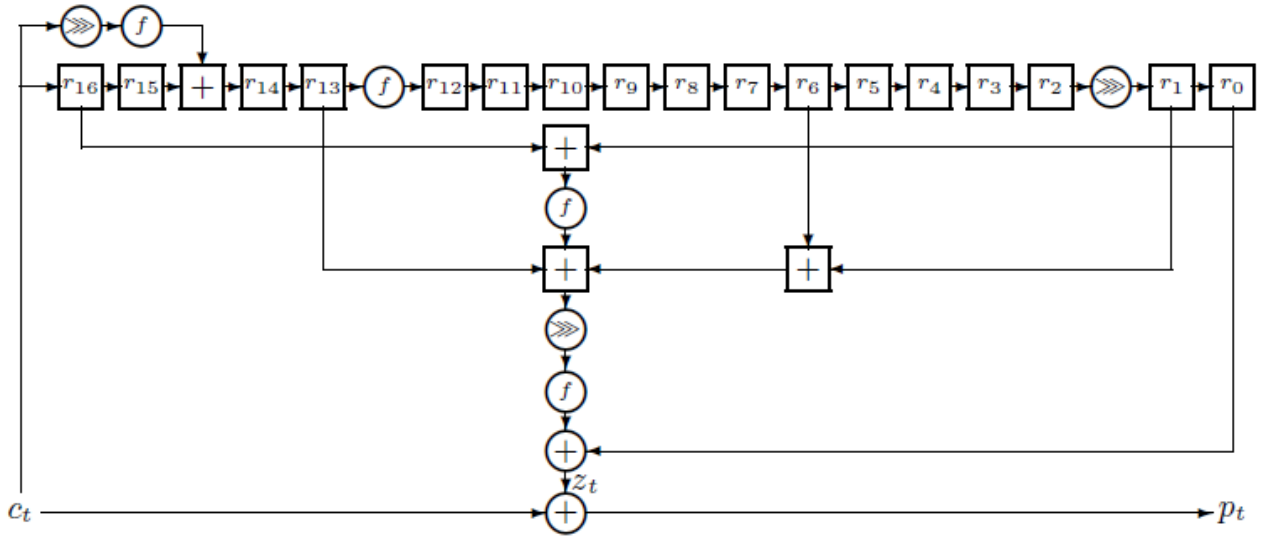


Рисунок 2.4 – Схема формування потокових символів [3]

Основним будівельним блоком є НФФ f, що залежить від ключа, а по суті являється перестановкою бітів в 16-бітному слові. Формульно її можна представити так:

$$f(x) = SBox(x_H) \oplus x, \tag{2.8}$$

де x_H - старший байт слова (MSB) $\boldsymbol{x}$, а S-Box – залежна від ключа функція, що визначається на етапі встановлення ключа. Вважатимемо, що це таблиця із 256 16-бітних слів.

2.2.5 S-Box

НФФ *f,* визначена у формулі (2.8) вище, – нелінійна функція, що залежить від ключа шифру. По суті, це єдиний елемент шифру, що залежить від секретного ключа. Тому попередньо обраховане значення цієї функції у вигляді таблиці із 16 бітних слів цілком може вважатись ключем всього шифру. Всього можливих комбінацій таких таблиць – 2^{128}, звідки і випливає складність перебору такого ключового простору.

Функція S-Box, що фігурує у формулі (2.8), – це комбінація Skipjack [8] S-Box (інша назва F-table) та Q-Box [5] Центру Розробки в галузі Інформаційної Безпеки (Information Security Research Centre, ISRC).

Skipjack S-Box – нелінійна перестановка 8-бітних векторів. Q-Box – нелінійна таблиця, на вхід якої приходить 8-бітний вектор, а на виході отримуємо 16-бітний. Розглянемо, яким чином поєднуються ці дві функції в одну в імплементації S-Box шифру SSS.

Нехай довжина ключа в бітах становить N, i-й біт ключа – $K_i, i = 0..N-1$. Вхідний байт x поєднується із відповідним байтом ключа для обрахунків власне функції S-Box:

$$t = Ftable\big(K_{N-1} \oplus Ftable(K_{N-2} \oplus \dots Ftable(K_0 \oplus x)\dots)\big) \qquad (2.9)$$

де t – це тимчасовий проміжний результат, що йде як вхід для Q-Box. Наглядно це можна продемонструвати на прикладі програмного коду на мові C++ (Додаток А, функція «Sfunc», рядки 26-41).

Для оптимізації часу роботи шифру, ця функція може буде обрахована і представлена у вигляді таблиці до виконання операцій зашифрування-розшифрування. Як результат, її використання зводиться до виконання операції XOR між вхідним словом та відповідним словом із попередньо обрахованої таблиці S-Box.

2.3 Шифр HBB

В оригінальній статті [15] із специфікацією шифру вказується, що HBB – це новий потоковий шифр, в основі якого лежить поєднання нелінійного та лінійного відображень. Лінійне відображення базується на використанні двох 256-бітних клітинних автоматів. Нелінійна частина шифру реалізована таким чином, щоб уникнути використання арифметики над скінченними полями. Автори шифру наполягають на тому, що максимальна кореляція, яка може виникати між входом та виходом вищезгаданих відображень, становить не більше ніж 2^{-13}. Цей шифр в базовому режимі є власне синхронним, але підтримує режим із самосинхронізацією на ряду із іншим режимом – MAC (Message Authentication Code). Саме режим самосинхронізації для HBB ми розглядатимемо в даній роботі. Як тільки 4 блоки розміром в 128 бітів отримані без наявності помилок, система автоматично синхронізується.

2.3.1 Характеристики шифру HBB

Автори: Palash Sarkar.

Параметри: шифр, що самосинхронізується, довжина ключа 128 (256) біт, внутрішній стан описується 640 бітами.

Основні операції: додавання за модулем 2 (XOR) та стандартні бітові операції.

Конструктивні особливості: складається із лінійної компоненти, що представлена двома клітинними автоматами, як альтернатива регістрам зсуву.

Переваги: вимагає невеликої пам’яті, можливість використання для аутентифікації повідомлення (MAC).

Криптоатаки: існує атака, що дозволяє знайти секретний ключ з вибраним ШТ довжиною приблизно в 2 Кбайт.

Конкурс/Конференція: конференція Indocrypt’03.

2.3.2 Короткий опис клітинних автоматів

Лінійну систему можна представити як деяке лінійне перетворення. Нехай деякий вектор $\left(s_1^{(t)}, \dots, s_k^{(t)}\right)$ – визначає стан k-бітної лінійної системи в час t. Тоді наступний стан $\left(s_1^{(t+1)}, \dots, s_k^{(t+1)}\right)$ в момент часу $(t+1)$ визначається як множення вектору $\left(s_1^{(t)}, \dots, s_k^{(t)}\right)$ із фіксованою матрицею M розмірності $k \times k$.

Клітинний автомат (КА) – це лінійна система, що визначається деякою матрицею M. У випадку клітинного автомату ця матриця – тридіагональна. Характеристичний поліном такої матриці – це характеристичний поліном для всіх послідовностей, що можуть бути отримані з кожної клітини такого автомату. Якщо такий поліном примітивний, то період лінійної послідовності на виході є максимальним: $2^k - 1$. Справедливе також наступне твердження, що якщо характеристичний поліном примітивний, то верхня та нижня субдіагоналі матриці M складаються із одиниць. Саме в такому випадку КА називається 90/150 КА. Наступні рівняння описують вектори стану такого 90/150 КА.

$$\begin{cases} s_1^{(t+1)} = c_1 s_1^{(t)} \oplus s_2^{(t)} \\ s_i^{(t+1)} = s_{i-1}^{(t)} \oplus c_i s_i^{(t)} \oplus s_{i+1}^{(t)}, \text{для } 2 \le i \le k-1 \\ s_k^{(t+1)} = s_{k-1}^{(t)} \oplus c_k s_k^{(t)} \end{cases} \qquad (2.10)$$

Вектор $(\mathrm{c}_1, \dots, \mathrm{c}_k)$ називається вектором-правилом 90/150 КА. Значення 0 визначає правило 90, а значення 1 - правило 150.

В шифрі, що розглядається в даній роботі, клітинні автомати визначаються наступними характеристичними поліномами:

$$p_0(x) = a_0 \oplus a_1 x \oplus \dots \oplus a_{255} x^{255} \oplus x^{256}, \qquad (2.11)$$
$$p_1(x) = b_0 \oplus b_1 x \oplus \dots \oplus b_{255} x^{255} \oplus x^{256}.$$

Бінарні рядки $a_0 \dots a_{255}$ та $b \dots b_{255}$ мають довжину в 256 бітів та вагу 128 кожний. В шістнадцятковій системі числення ці рядки мають наступний вигляд:

$$(a_0 \dots a_{255})_{16}$$
$$= f9169344ec191960a5bc37331451501906cad5d1663e2bb7d3cd5aaa75d7cca7$$
$$(b_0 \dots b_{255})_{16}$$
$$= f36f6b011149b6bc986b889d3c224e102e67793a98cf32d94c8de2567aacf82f$$

Правила 90/150 для даних клітинних автоматів знайдено з допомогою алгоритму Тезуки та Фушимі [16] та мають наступне шістнадцяткове представлення:

$$\mathcal{R}_0: 2d240f0e5308f30bd460bab9265cffd11279819e92dc69a50b9da4c018b274d5$$
$$\mathcal{R}_1: 91070f8787e737b546f6934aa14b3f26bc87113e6a2e8096da0bd5e7f34e718c$$

2.3.3 Основні функції шифру HBB

Шифр може працювати в трьох різних режимах: базовий (**B**), із аутентифікацію повідомлень (**MAC**), із самосинхронізацією (**SS**). В ході даної роботи буде реалізовано базовий режим та на його основі режим із самосинхронізацією, буде проведено аналіз атаки на останній.

Введемо наступні позначення:

- M – повідомлення ВТ, що буде подано на вхід функції шифрування;
- M_i – 128-бітний блок повідомлення ВТ;
- $M = M_0 \,||\, M_1 \,||\, \dots ||M_{n-1}$, де $n -$ кількість 128-блоків, з яких складається повідомлення ВТ;
- $KEY -$ секретний ключ системи шифрування;
- $K = K_0 \,||\, K_1 \,||\, \dots ||K_{n-1}$ – біти ключової послідовності;
- $C = C_0 \,||\, C_1 \,||\, \dots ||C_{n-1} -$ 128-бітні блоки ШТ;

- $LC-$ лінійна компонента внутрішнього стану шифру, займає 512 бітів;
- $NLC-$ нелінійна компонента внутрішнього стану шифру, на яку припадає 128 бітів;

Отже, внутрішній стан шифру описується 640 бітами, які визначаються лінійною та нелінійною компонентами.

Опишемо основні функції, що використовуються під час роботи шифру: $Fold(), Exp(), Round(), NLSub()$.

- $Fold(S, i)$: S – це рядок із бітів, довжина якого додатне число кратне i.

  $Fold(S, i)$

  1. $S = S_1 \,||\, S_2 \,||\, \dots ||S_k$, де $|S| = k \times i$ та $|S_j| = i$ для всіх $1 \le j \le k$;
  2. $output\ S_1 \oplus S_2 \oplus \dots \oplus S_k$.

- $Exp(KEY) = KEY||\overline{KEY} \,||\, \overline{KEY} \,||KEY,$ де розмір ключа становить 128 бітів.

- $Round()$ – функція, що на вхід приймає лінійну та нелінійну компоненту шифру LC та NLC відповідно, та на виході видає наступні 128 бітів ключової послідовності. Будемо вважати кожен із клітинних автоматів LC масивом довжини 16, де кожен елемент - це 32-бітне слово. Тому стан першого клітинного автомату позначатимемо як $LC[0, \dots, 7]$, другого – $LC[8, \dots, 15]$, K_0, K_1, K_2, K_3 – 128-бітні блоки ключової послідовності.

  $Round(LC, NLC)$

  1. $NLC = NLSub(NLC)$;
  2. $\Delta = NLC_0 \oplus NLC_1 \oplus NLC_2 \oplus NLC_3$, де $|NLC_i| = 32$;
  3. $for\ i = 0\ to\ 3\ NLC_i = (\Delta \oplus NLC_i) <<< (8 \times i + 4)$;
  4. $NLC = FastTranspose(NLC)$;

5. $NLC = NLSub(NLC)$;
6. $LC = NextState(LC)$;
7. $K_0 = NLC_0 \oplus LC_0$; $K_1 = NLC_1 \oplus LC_7$; $K_2 = NLC_2 \oplus LC_8$; $K_3 = NLC_3 \oplus LC_{15}$;
8. $NLC_0 = NLC_0 \oplus LC_3$; $NLC_1 = NLC_1 \oplus LC_4$; $NLC_2 = NLC_2 \oplus LC_{11}$; $NLC_3 = NLC_3 \oplus LC_{12}$;
9. $return\ (K_0 || K_1 || K_2 || K_3, LC, NLC)$.

- $NLSub(NLC)$ – нелінійне перестановка компоненти шифру NLC. Це перетворення використовує функцію $byteSub(\ )$ для перестановки бітів у кожному байті NLC. Функція $byteSub(\ )$ - це функція байт-перестановки із AES.

- $FastTranspose(NLC)$ – це функція, в яку на вхід приходить 128 бітів NLC, що трактуються як елементи матриці 4×32, де кожний рядок це 32-бітне слово. Дана функція здійснює часткове транспонування вхідної матриці.

- $NextState(LC)$ – це функція, що задає відповідність між 512 бітовими рядками, тобто це лінійне відображення із 512-бітового вектору в 512-бітовий вектор.

$NextState(LC)$

1. $LC[0, \dots, 7] = Evolve(LC[0, \dots, 7], \mathcal{R}_0)$;
2. $LC[8, \dots, 15] = Evolve(LC[8, \dots, 15], \mathcal{R}_1)$;
3. $return\ LC$.

Змінні $\mathcal{R}_0$ та $\mathcal{R}_1$ – це правила 90/150 для клітинних автоматів, що використовуються в даному шифрі.

- $EvolveCA(state, rule)$
    1. $state = (state \ll 1) \oplus (rule \wedge state) \oplus (state \gg 1)$;
    2. $return\ state$.

2.3.4 Алгоритм роботи шифру HBB

Знаючи усі необхідні складові шифру, сформуємо псевдокод роботи алгоритму шифру на основі попереднього опису та оригінальної статті:

$HBB(M_0 \,||\, M_1 \,||\, \dots ||M_{n-1}, KEY)$

1. $LC = Exp(KEY)$; $\quad NLC = Fold(KEY)$;
2. $for\ i = 0\ to\ 3\ do\ (T_i, LC, NLC) = Round(LC, NLC)$;
3. $LC_{-1} = LC \oplus (T_3 \,||\, T_2 \,||T_1||T_0)$; $\ NLC_{-1} = NLC$; $\ C_{-3} = C_{-2} = \ C_{-1} = 0^{128}$;
4. $for\ i = 0\ to\ n-1\ do$
5. $(K_i, LC_i, NLC_i) = Round(LC_{i-1}, NLC_{i-1})$;
6. $C_i = M_i \oplus K_i$;
7. $if\ MODE = SS : LC = Exp(KEY) \oplus (C_i \,||\, C_{i-1} \,||C_{i-2}||C_{i-3})$;
8. $NLC = Fold(KEY) \oplus C_i \oplus C_{i-1} \oplus C_{i-2} \oplus C_{i-3}$;
9. enddo;
10. output $C_0 \,||\, C_1 \,||\, \dots ||C_{n-1}$.

Даний псевдокод представляє роботу функції зашифрування, щоб отримати відповідний алгоритм дешифрування, достатньо змінити декілька рядків:

6. $M_i = C_i \ \oplus K_i$;
10. output $M_0 \,||\, M_1 \,||\, \dots ||M_{n-1}$.

Режим із самосинхронізацією реалізується шляхом введення залежності для символів ключової послідовності від попередніх символів ШТ. Рядки 7-8

вищезгаданого алгоритму роботи шифру HBB описують конкретну реалізацію введення даної залежності.

В кінці ітерації i останні 4 блоки ШТ (кожен блок по 128 бітів), визначені як $C_i, C_{i-1}, C_{i-2}, C_{i-3}$, додаються за $mod2$ із бітами ключа KEY. Результатом даної операції є новий стан нелінійної компоненти (Рядок 8 алгоритму HBB).

Новий стан лінійної компоненти отримується шляхом додаванням за $mod2$ конкатинованих останніх чотирьох блоків ШТ та розширеного ключа. (Рядок 7 алгоритму HBB).

Таким чином, на початку кожної нової ітерації внутрішній стан шифру лінійно залежить лише від секретного ключа та останніх 512 бітів ШТ.

2.3.5 Предметна область та імплементація шифру HBB

Оскільки шифр HBB не представлявся на конкурсах типу eStream, то його конкретної реалізації немає у відкритому доступі. Тому, було запропоновано предметну область для шифру та на її основі реалізовано власне шифр HBB на мові C# засобами програмного середовища MS Visual Studio та .Net.

Запропоновано 3 класи для імплементації основних складових шифру:

- Block – для опису блоків, з яких складаються ВТ, ШТ та ключова послідовність символів;
- CA – для опису двох клітинних автоматів, що відіграють роль лінійної компоненти шифру;
- NLC – для опису нелінійної компоненти шифру.

Коротко опишемо ці класи на прикладі коду.

Повна імплементація шифру викладена у Додатках Ґ-Ж .

Таблиця 2.4 – Опис класу «Block»

Код класу	Пояснення

public class Block
{
public UInt32 first;
public UInt32 second;
public UInt32 third;
public UInt32 fourth;
 	
}
	Повідомлення ВТ та ШТ розбиваються на блоки по 128 бітів. Кожен блок складається із чотирьох 32-бітних слів, щоб таким чином реалізувати структуру, що описує 128-бітне представлення 1 блоку.

Таблиця 2.5 – Опис класу «CA»

Код класу	Пояснення

public class CA
{
public UInt32 state0;
public UInt32 state1;
public UInt32 state2;
public UInt32 state3;
public UInt32 state4;
public UInt32 state5;
public UInt32 state6;
public UInt32 state7;
CAorder order;
UInt32 RULE00 = 0x80ffaf46;
UInt32 RULE01 = 0x977969e9;
UInt32 RULE02 = 0x71553bb5;
UInt32 RULE03 = 0x99be6b2b;
UInt32 RULE04 = 0x4b337295;
UInt32 RULE05 = 0x2308c787;
UInt32 RULE06 = 0xb84c7cce;
UInt32 RULE07 = 0x36d501e6;
UInt32 RULE10 = 0xdd18c62b;
UInt32 RULE11 = 0x153df31a;
UInt32 RULE12 = 0xc98e86c1;
UInt32 RULE13 = 0x910fee24;
UInt32 RULE14 = 0x2942d51b;
UInt32 RULE15 = 0x4201eb3d;
UInt32 RULE16 = 0xc1d1a85f;
UInt32 RULE17 = 0x57b8919b;
 	
public void Exp()
public void EvolveCA256()
}
	Клас, що реалізовує клітинний автомат, стан якого описують 256 бітів - вісім 32-бітних слів, що відповідають за опис внтріщнього стану автомата, його номер (всього в шифрі застосовується 2 таких автомати), та набір функцій, що застосовуються до відповідного автомату, наприклад, Exp() та Evolve256().

Таблиця 2.6 – опис класу «NLC»

Код класу	Пояснення

[StructLayoutAttribute(LayoutKind.Explicit)]
public class NLC
{
[FieldOffsetAttribute(0)]
public UInt32 word;
 	
[FieldOffsetAttribute(0)]
public byte byte0;
 	
[FieldOffsetAttribute(1)]
public byte byte1;
 	
[FieldOffsetAttribute(2)]
public byte byte2;
 	
[FieldOffsetAttribute(3)]
public byte byte3;
}
	Клас, що описує нелінійну компоненту. Основним атрибутом цього класу, є 32-бітне слово, що розбивається на 4 байти. Особливість реалізації цього класу полягає в тому, що зміна будь-якого із чотирьох байтів провокує зміну основного слова «word».

Висновки до розділу 2

В даному розділі описано три шифри із самосинхронізацією: Moustique, HBB та SSS. Для кожного шифру надані короткі відомості та характеристика, проаналізовано основні складові функції та компоненти, описано загальний алгоритм їхньої роботи за допомогою псевдокоду або математичних виразів. Для шифру HBB запропонована предметна область для імплементації шифру на мові C#.

3 ОГЛЯД ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ КРИПТОАТАК

В даному розділі розглянуто дві атаки на основі обраного ШТ для потокових шифрів із самосинхронізацією: SSS та HBB. Також наведено порівняльну характеристику цих атак з метою виявлення тенденцій в криптоаналізі потокових шифрів із самосинхронізацією.

3.1 Аналіз атаки на основі вибраного ШТ на шифр SSS

З опису шифру випливає, що секретним ключем для шифрування можна вважати саме таблицю Sbox із 256 16-бітних слів. Тому головною ціллю нижче наваденої атаки, що описана в [3], 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 i $(i = 0,1,2,\dots,263)$ складатимуться із 18 16-бітних слів та завжди має наступний формат:

$$\begin{cases} c_t^i = 0, \text{для } t = 0,\dots,12,14,\dots,18 \\ c_t^i = b^i, \end{cases} \tag{3.1}$$

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

$$z_{18}^i = f((f(r[0] + r[16]) + r[1] + r[6] + r[13]) >>> 8) \oplus r[0] \tag{3.2}$$

Значення цих регістрів в момент часу t = 18:

$$\begin{cases} r[0] = f^2(0) >>> 8 \\ r[1] = f^2(0) >>> 8 \\ r[6] = f^2(0) \\ r[13] = f(0) + b^i \\ r[16] = 0 \end{cases} \quad (3.3)$$

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

$$a = f\big((r[0] + r[16]) + r[1] + r[6] + f(0)\big) = \quad (3.4)$$
$$= f(f^2(0) >>> 8) + (f^2(0) >>> 8) + f^2(0) + f(0),$$

в результаті отримаємо наступний вираз:

$$z_{18}^i = f\left(\left(a + b^i\right) >>> 8\right) \oplus r_0. \quad (3.5)$$

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

$$b_L^i = 0, b_H^0 = 0, b_H^i = 2^{i-1}, \text{де } i = 1,2,\dots,7. \quad (3.6)$$

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

$$z_{18}^i = f\left(\left(a + b^i\right) >>> 8\right) \oplus r_0 =$$
$$= SBox\left(\left(a + b^i\right)_L\right) \oplus \left(2^8 \cdot a_L + a_H + 2^{i-1}\right) \oplus r_0 = \quad (3.7)$$
$$= SBox(a_L) \oplus \left(2^8 \cdot a_L + a_H + 2^{i-1}\right) \oplus r_0$$

здійснимо операцію XOR для значень символів ключової послідовності, що відповідають часовим крокам 18 та 0 і отримаємо наступне:

$$z_{18}^{i} \oplus z_{18}^{0} = \left[SBox(a_L) \oplus \left(2^8 \cdot a_L + a_H + 2^{i-1}\right) \oplus r_0\right] \oplus$$
$$\oplus [SBox(a_L) \oplus (2^8 \cdot a_L + a_H) \oplus r_0] = \quad (3.8)$$
$$= a_H \oplus \left(a_H + 2^{i-1}\right).$$

Виходячи із результату останньої операції, можна судити про значення a_H:
$z_{18}^{i} \oplus z_{18}^{0}$ дорівнює 2^{i-1}, то відповідний біт a_H буде 0, інакше – 1.
Як тільки a_H відновлено, то можна спробувати відновити цілком усю таблицю, що відповідає Sbox. Надалі це здійснюватиметься простим перебором: вгадуємо значення a_L та $SBox(a_L)$ - загалом 24 біти, тобто складність такого перебору 2^{24}. Зробимо це з допомогою патернів від 0 до 255. Формат у них наступний:
$b_L^j = j$ та $b_H^j = 0$ для $j = 0,1, \dots ,255$. Проробимо аналогічні дії як для a_H:

$$z_{18}^{j} \oplus z_{18}^{0} = SBox(a_L) \oplus SBox(a_L + j) \oplus \quad (3.9)$$
$$\oplus \left(\left((2^8 \cdot a_H + a_L) \oplus (2^8 \cdot a_H + a_L + j)\right) >>> 8\right).$$

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

$$SBox(a_L + j) = z_{18}^{j} \oplus z_{18}^{0} \oplus SBox(a_L) \oplus \quad (3.10)$$
$$\oplus \left(\left((2^8 \cdot a_H + a_L) \oplus (2^8 \cdot a_H + a_L + j)\right) >>> 8\right).$$

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

3.2 Засоби для реалізації атаки на основі вибраного ШТ для шифру SSS

Атака реалізована у вигляді консольного програмного додатку Win32. Для реалізації атаки на основі ШТ для шифру із самоорганізацією SSS були використанні наступні засоби:

1. Microsoft Visual Studio Express 2012 – інтегроване середовище розробки програмного забезпечення;
2. Microsoft Visual C++ (MSVC) — інтегроване середовище розробки програмного забезпечення на мові C++, розроблена фірмою Microsoft. Visual C++.NET підтримує розробку програмних додатків як на Managed C++, так і на звичайному C++, і тим самим дозволяє генерувати код як для платформи .NET Framework, так і для виконання в середовищі операційної системи Windows;
3. Вихідний код шифру SSS, представлений на конкурсі eStream [7].

Обґрунтування вибору засобів реалізації атаки: відкритий код шифру реалізований на мові програмування C, що є звичним явищем для шифрів. Саме програми реалізовані мовою C дозволяють максимально реалізувати швидкодію роботи шифру за рахунок керування пам’яттю комп’ютера. Код мови C мало відрізняється від мови C++ для додатків, що використовують структурне програмування, тому код шифру легко конвертується під засоби Microsoft Visual Studio та Visual C++. Виходячи з того, що атака вимагає наявності дешифратора, що реалізований на мові C++, атака теж здійснювалась засобами інтегрального середовища Visual C++.

3.3 Реалізація атаки на основі вибраного ШТ для шифру SSS

Основною ціллю атаки є відтворення значень таблиці S-box, що залежать від бітів ключа. Для швидкодії ця таблиця може обраховуватись на

початковому етапі роботи шифру, що передує етапові зашифрування повідомлення. Відповідно, для реалізації атаки нам необхідно лише 2 базові функції із відкритого коду шифру та 4 допоміжні, що необхідні в процесі розшифрування:

1. «sss_key» – базова функція, що обраховує значення S-box, як результат змішування бітів ключа та двох таблиць: ftable та Qbox;
2. «sss_deconly» – базова функція розшифрування на основі значень S-box та бітів ШТ;
3. «BYTE2WORD», «WORD2BYTE» – дві допоміжні функції для конвертації «слів» шифру в байти та навпаки;
4. «cycle» – функція циклічного зсуву для регістра;
5. «nltap» – функція, що здійснює нелінійне перетворення від деяких частин регістру.

Допоміжні функції описані в пунктах 3-5 використовуються під час виконання базових функцій 1-2. Код всіх функцій представлений в додатку #.

Для реалізації атаки з допомогою програмного додатку, представленого в даній роботі, необхідно отримати розшифроване повідомлення, що відповідає ШТ із патернів описаних в розділі 3.1.

Опишемо короткий алгоритм роботи програмного додатку, що реалізовує атаку на SSS на прикладі програмного коду в Додатку А:

1. Обрахунок таблиці S-box на основі ключа (рядки 70-81, 232);
2. Генерація ШТ із 263 патернів (рядки 235-246);
3. Розшифрування ШТ із патернів та отримання відповідного ВТ (рядки 169-211, 248);
4. Визначення за результатами попереднього пункту значення старшого байту a_H, описаної в розділі 3.1 (рядки 250-268);
5. Визначення значення всієї таблиці S-box через повний перебір («brute force» (рядки 269-292);
6. Валідація отриманого ключа-таблиці (рядки 293-302).

Дана атака була практично реалізована та здійснена на звичайному персональному комп’ютері. Час її виконання складає менше 10 секунд. Результатом атаки є значення всіх полів таблиці S-box, що застосовуються в процесі зашифрування та розшифрування повідомлень.

3.4 Модифікація атаки на шифр SSS

В оригінальній статті [3], що описує атаку на даний шифр, автор говорить про можливу модифікацію атаки, використовуючи патерни для ШТ, що перетинаються. Під перетином мається на увазі, що одні і ті самі патерни використовуються як для знаходження старшого біту a_H так і для загального перебору значень таблиці S-box.

Неважко помітити, що не має сенсу генерувати обраний ШТ більшого розміру, якщо деякі патерни однакові. Програмна модифікація дозволяє віднайти такі патерни за лінійний час, що залежить від кількості патернів. Провівши невелике дослідження, можна помітити, що патерни $c_i, для\ i = 0,1,\dots,8$ та $i = 257,258,\dots,263$ повністю співпадають. В коді, що викладений в Додатку А, кількість надлишкових патернів описана змінною «NumberOfExtraPatterns». Відповідно, обраний ШТ можна скоротити в розмірі до 256 патернів, замість 263.

Автору атаки знадобилось близько 9,5 Кбайт обраного ШТ, щоб зламати ключ шифру – таблицю S-box. Виходячи із опису та запропонованої модифікації, для реалізації атаки необхідно 256 патернів, кожен з яких становить 36 байтів, загалом необхідно 9 Кбайт. Тобто зменшення розміру необхідного ШТ становить близько 6%.

3.5 Аналіз атаки на основі вибраного ШТ на шифр HBB

У пункті 2.3.3 даної роботи зазначили, що кожна ітерація шифру HBB в режимі із самосинхронізацією складається із фіксованого числа функцій, що застосовуються до ключа та останніх чотирьох блоків ШТ. Дана конструкція шифру не стійка перед атакою на основі обраного ШТ.

Оскільки атака на основі обраного ШТ, тому вважатимемо, що є оракул на вхід якого приходить ШТ і на виході правильно розшифрований ВТ.

Головна мета даної атаки, повний опис якої наведений в статті Антуана Жу (Antoine Joux) [11] – отримати два виходи нелінійної компоненти NLC для функції оновлення стану, що відрізняються лише в 1 байті із різницею в δ. Щоб досягти такої різниці, можемо просто згенерувати дві послідовності із блоків ШТ наступним чином:

$$C_1, C_2, C_3, C_4 \text{ та } C_1, C_2, C_3, C_4 \oplus \delta \tag{3.11}$$

Нехай j – це позиція цієї різниці в 1 байт, K_j – відповідний байт секретного ключа. На перший погляд, криптоаналітик не має доступу до внутрішнього стану, щоб дізнатись різницю між виходами після роботи функції, що оновлює стан нелінійної компоненти. Але як зазначалось в кінці розділу 2.3.3, на початку кожної нової ітерації внутрішній стан шифру лінійно залежить лише від секретного ключа та останніх 4 блоків ШТ. Тому криптоаналітик, що володіє бітами ключової послідовності, може відмінити дію гамування та отримати у відкритому вигляді різницю для двох виходів функції оновлення стану нелінійної компоненти.

Як зазначалось вище в описі самого шифру, функція, що оновлює стан нелінійної компоненти – $NLSub(\,)$ складається із декількох етапів, першим з яких є деяка функція $byteSub(\,)$ перестановки бітів для кожного байту нелінійної компоненти. Тому, початкова різниця δ в 1 байт, після дії цієї перестановки,

змінить лише 1 байт за тією ж самою позицією j. Ця різниця матиме наступний вигляд:

$$\delta' = S(K_j \oplus x) \oplus S(K_j \oplus x \oplus \delta) \quad (3.12)$$

де x – відомий байт, залежно від заданого ШТ. Другим етапом функції оновлення – є лінійне перетворення. Якщо різниця δ' має гамуючий ефект ваги 1, то згідно [15], гамуючий ефект після лінійного перетворення буде ваги 3. Тобто, лише 3 байти будуть відрізнятись після застосування останнього кроку функції оновлення.

Рисунок 3.1 – Ланцюжок гамування для нелінійної компоненти

Узагальнимо результат: якщо гамуючий ефект для δ' буде ваги 1, то різниця між виходами нелінійної компоненти нуль для 13 байтів із 16. Згідно з статтею [11], подія, коли виникає гамуючий ефект ваги 1, трапляється з ймовірністю 0,03125. Згідно «парадоксу про дні народження» [17], перебору 2^3 значень для δ, має бути достатньо, щоб отримати вище описаний випадок, коли для 13 із 16 байтів гамуючий ефект рівний нулю. Використовуючи формулу (3.12) та знання про вищезгадану подію, отримаємо умову для K_j:

$$weight\left(S(K_j \oplus x) \oplus S(K_j \oplus x \oplus \delta)\right) = 1 \quad (3.13)$$

Як результат, 1 байт секретного ключа можна зламати, опрацювавши 2^3 блоків ключової послідовності. Щоб зламати весь ключ, треба 16 разів повторити дану процедуру. Складність такої атаки можна оцінити з точки зору розміру необхідного обраного ШТ. Щоб опрацювати $2^3 \times 16$ блоків ключової послідовності, має бути достатньо повідомлення розміром $3 + 2^3 \times 16$, що вимагає приблизно 2 Кбайт обраного ШТ.

3.6 Порівняльна характеристика атак на основі вибраного ШТ для шифрів HBB та SSS

Оскільки розглянуті в роботі шифри одного типу – потокові шифри із самосинхронізацією та розглянуті для них атаки на основі обраного ШТ теж схожі, як узагальнення та підсумок для даного дослідження доцільно привести порівняльну характеристику цих атак.

Таблиця 3.1 – Порівняльна характеристика атак для шифрів HBB та SSS

Складова/характеристика	HBB	SSS
1	2	3
Тип шифру	Потоковий із самосинхронізацією	
Тип атаки	На основі обраного ШТ	
Необхідний розмір ШТ, байт	2048	9216
Передумови	Наявність оракула – функції розшифрування	
Елемент, на який здійснюється атака	Нелінійна компонента шифру NLC	Регістр зсуву

Продовження таблиці 3.1

1	2	3
Необхідні попередні обчислення	Знаходження колізій, при яких виникає гамуючий ефект ваги 1 згідно формул (3.12). Генерація обраного ШТ до виконання умови (3.13)	Генерація патернів для створення обраного ШТ. Знаходження старшого байту a_H згідно формул 3.4-3.10.
Складність частини атаки, де здійснюється перебір	2^{12} – перебір для пошуку колізій: 16 разів для кожного байту ключа	2^{24} – перебір можливих параметрів для таблиці S-box згідно формули (3.10)
Відновлення ключа	Повністю відтворюється весь ключ	Відновлення внутрішнього стану S-box, що по суті виступає ключем
Можливі модифікації	Аналогічні атаки, що підтримують колізії для гамуючих ефектів ваги 2, 3, 4	Скорочення необхідного ШТ через знаходження патернів, що перетинаються

Порівняльна характеристика даних атак наштовхує на думку, що для шифрів із самосинхронізацією важливою умовою їх проектування має бути стійкість до атак на основі обраного ШТ. Можливість та існування таких атак можна пояснити тим, що режим із самосинхронізацією для потокових шифрів подібний до режиму із зворотним зв’язком за ШТ.

Також, на етапі проектування важливо окремо проводити тестування складових шифру. Наприклад, для шифру HBB найслабшим елементом можна вважати нелінійну компоненту та функцію, яка оновлює стан цієї компоненти перед кожною ітерацією шифру до одного і того ж самого стану. Існування

слабкого гаммуючого ефекту, описаного в підрозділі 3.5, дозволяє відтворити біти ключа. Для шифру SSS найслабшою ланкою став регістр зсуву. Але наявні кореляції для регістру в поєднанні із роботою таблиці S-Box дозволяють відновити внутрішнє заповнення всієї таблиці, що дозволяє імітувати роботи шифру повністю. Звісно, автори шифрів переконливо доводять доцільність використання тих чи інших елементів-складових функцій алгоритму, але проблеми даних шифрів виникають на етапі взаємодії цих функцій. Тому інтеграційне тестування взаємодії елементів шифрів на етапі проектування дозволить протидіяти деяким атакам та виявити слабкі місця шифрів.

Також, надлишкове ускладнення регістру зсуву не веде до покращення його криптографічної стійкості, це показано на прикладі шифру Moustique, для якого існують корельовані шифруючі гамми. Хорошою альтернативою для регістрів зсуву є клітинні автомати, запропоновані в такій ролі індійським криптографом Палаш Саркаром (Palash Sarkar), розглянуті в розділі, що описує шифр HBB.

Висновки до розділу 3

В даному розділі проаналізована атака на основі вибраного ШТ для шифру SSS, описана її математична частина та алгоритм програмної реалізації атаки на мові Visual C++. Наведено обґрунтування використання засобів необхідних для реалізації даної атаки. Також, описано практичну складність реалізації атаки та можлива її модифікація, що дозволяє скоротити величину обраного ШТ на 6%.

Для іншого шифру HBB проаналізовано атаку на основі обраного ШТ та можливість практичної її реалізації.

Проведена порівняльна характеристика двох вищезгаданих атак на основі обраного ШТ для шифрів SSS та HBB.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Аналіз умов праці

Роботи, що проводяться при проектуванні системи інформації для менеджменту, можна кваліфікувати як творчу роботу з ПК та іншими периферійними пристроями. Вивчення і вирішення проблем, пов'язаних із забезпеченням здорових та безпечних умов, в яких працює людина - одна з найбільш важливих завдань при розробці нових систем. Вивчення та виявлення можливих причин виробничих нещасних випадків, професійних захворювань, аварій, пожеж, і розробка заходів та вимог, спрямованих на усунення цих причин, дозволяють створити безпечні та сприятливі умови для праці людини. Робота співробітників безпосередньо пов'язана комп'ютером, а відповідно з додатковим шкідливим впливом цілої групи факторів, що істотно знижує продуктивність їх праці. До таких факторів можна віднести:

- Порушення мікроклімату;
- Неправильне освітлення;
- Ненормальний рівень шуму;
- Вплив шкідливих випромінювань;
- Наявність електричної напруги;
- Небезпека пожежі;
- Надзвичайні ситуації.

4.2 Опис приміщення

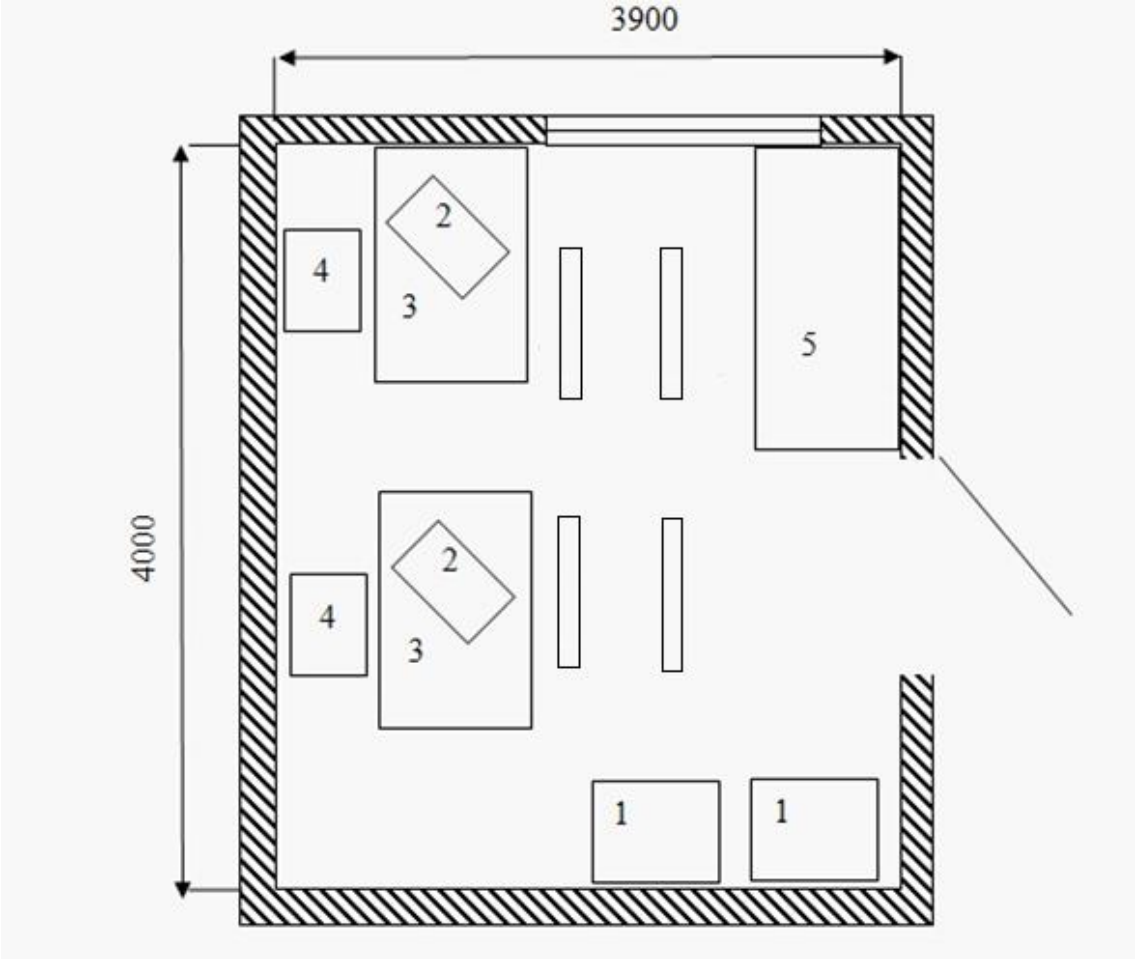


Рисунок 4.1 – План приміщення

Таблиця 4.1 – Розміри офісного приміщення

Довжина (м)	4.0
Ширина (м)	3.9
Висота стелі (м)	2.8
Загальна площа ($м^2$)	15.6
Загальний об’єм ($м^3$)	43.68
Кількість робочих місць	2

Таблиця 4.2 – Нормативні та фактичні значення площі та об’єму

	Норматив	Фактичне значення
Площа для 1 робочого місця, $м^2$	>6	7.8
Об’єм для 1 робочого місця, $м^3$	>20	21.84

На рисунку 4.1 зображено план офісного приміщення, у таблиці 4.2 подано нормативні (за НПАОП 0.00-1.28-10 «Правила охорони праці під час експлуатації ЕОМ») та розраховані фактичні значення параметрів робочого приміщення. З чого робимо висновок, що габарити приміщення є такими, що відповідають діючим нормам.

4.3 Аналіз шкідливих факторів

4.3.1 Повітря робочої зони

Основний принцип нормування мікроклімату - створення оптимальних умов для теплообміну тіла людини з довкіллям. У ДСанПІН 3.3.2.007-98 встановлені величини параметрів мікроклімату, що створюють комфортні умови.

Таблиця 4.3 – Норми мікроклімату для приміщень з ЕОМ

Пора року	Температура повітря, град.С	Відносна вологість повітря, %	Швидкість руху повітря, м/с
Холодний	22-24	40-60	0,1
Теплий	23-25	40-60	0,1

Таблиця 4.4 – Фактичні параметри мікроклімату у приміщенні

Пора року	Температура повітря, град.С	Відносна вологість повітря, %	Швидкість руху повітря, м/с
Холодний	22-24	40-60	0,1
Теплий	23-25	40-60	0,1

Висновок: повітря робочої зони відповідає нормам, взимку працює опалення.

4.3.2 Освітлення

У робочому приміщенні використовується штучне освітлення, доступ сонячного світла обмежують жалюзі на вікнах.

В якості освітлювальних приборів використовуються 4 люмінесцентні лампи ЛД 65 виробництва ООО «Завод ГРЛ», м. Полтава. Характеристики лампи наведені у Таблиці 4.5

Таблиця 4.5 – Характеристики лампи ЛД 65

Потужність, Вт	Світловий потік, лм	Напруга, В	Струм, А
65	3900	100-120	0.67

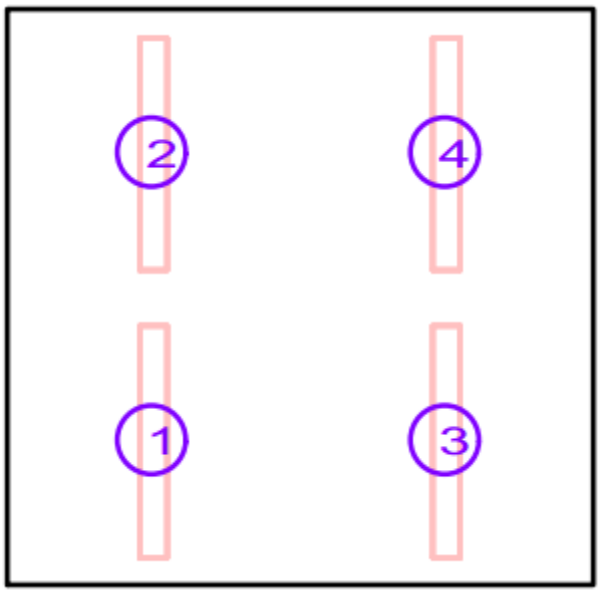


Рисунок 4.2 – Схема розташування ламп у приміщенні

На рисунку 4.2 зображено схему розташування ламп у приміщенні. У таблиці 4.6 зазначено точні координати ламп, що були використані для розрахунку освітлення у приміщенні за допомогою програми DIALux 4.10. Результати розрахунку відображені на рисунку 4.3

Таблиця 4.6 – Розташування ламп у приміщенні

	X	Y	Z
1	1.000	0.970	2.800
2	1.000	2.920	2.800
3	3.000	0.970	2.800
4	3.000	2.920	2.800

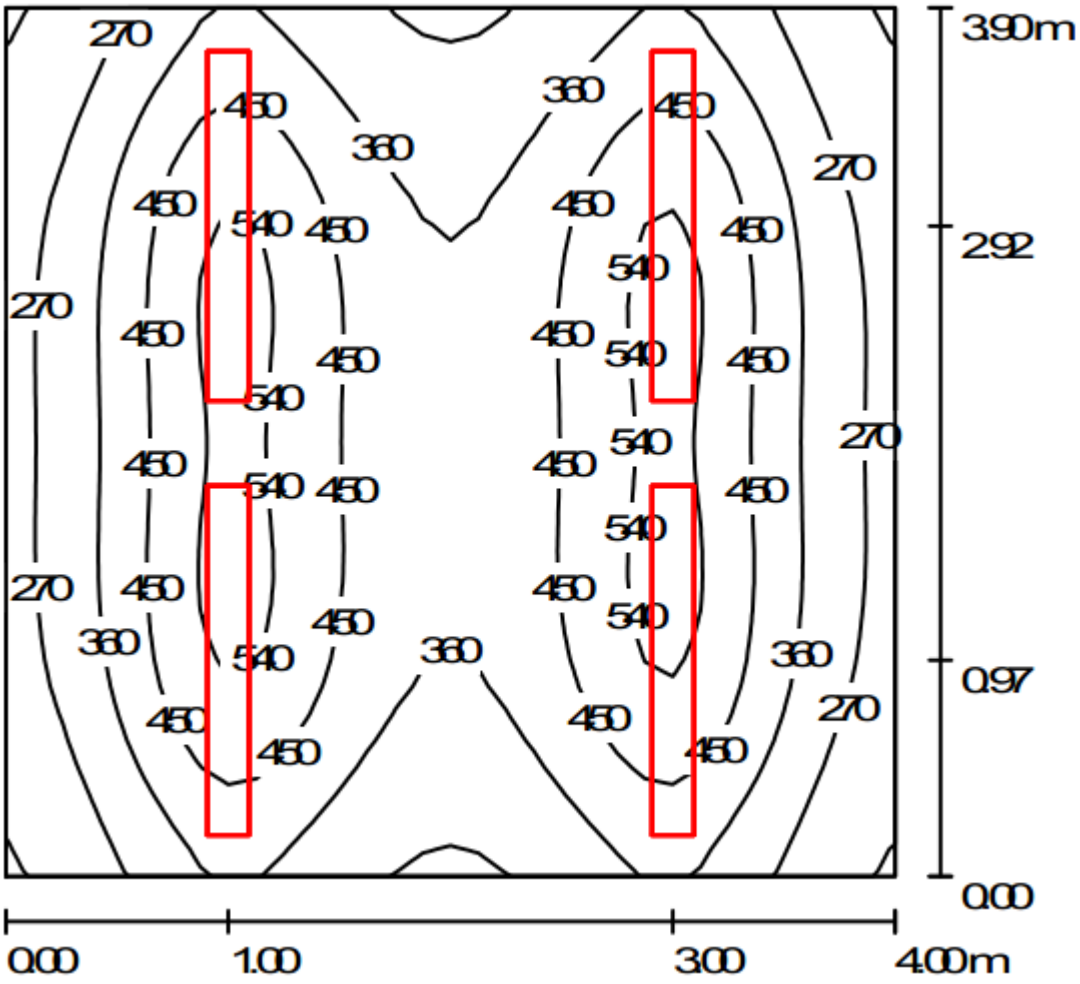


Рисунок 4.3 – Розрахунок освітлення приміщення

Таблиця 4.7 – Нормативні та фактичні значення освітленості

	Норматив	Фактичне значення
Мінімальна освітленність на поверхні робочого столу, лк	300	300
Максимальна освітленність на поверхні робочого столу, лк	500	450

У Таблиці 4.7 подано нормативні (за ДСанПіН 3.3.2-007-98. «Гігієнічні вимоги до організації роботи з візуальними дисплейними терміналами електронно-обчислювальних машин») та розраховані фактичні значення освітленості в робочих зонах працівників. Як бачимо, фактичні значення освітленості відповідають нормативним.

4.3.3 Шум

Таблиця 4.8 – Рівні шуму комплектуючих ПК

Назва джерела шуму	Рівень шуму, дБА	Звуковий тиск, Вт/м^2
Блок живлення	17	5.01 × 10-11
Корпус	19	7.94 × 10-11
Відеокарта	32	1.58 × 10-9
Кулер процесора	35	3.16 × 10-9

У Таблиці 4.8 зібрано список комплектуючих ПК, які є джерелами шуму, зазначено відповідні рівні шуму у дБ. Для розрахунку загального рівня шуму спочатку обрахуємо звуковий тиск кожного джерела шуму за формулою:

$$P_i = P_0 \times 10^{Di/10} \quad (4.1)$$

Де P_0 – еталонний звуковий тиск (10^{-12} Вт/м^2)
Обрахуємо сумарний рівень шуму за формулою:

$$D = 10 \times lg(P / P_0) \quad (4.2)$$

Де P – сумарний звуковий тиск комплектуючих

Таблиця 4.9 – Нормативні та фактичні значення рівня шуму

	Норматив, дБА	Фактичне значення, дБА
Рівень шуму ПК	<50	37

Висновок: рівень шуму відповідає нормативному.

4.3.4 Випромінювання

Для роботи використовуються монітори 27" Samsung S27A850D, що сертифіковані за стандартом TCO 5.0 / Energy Star.

4.3.5 Електробезпека

Для роботи освітлення, ЕОМ та іншого електроустаткування у робочому приміщенні використовується електроживлення з однофазної мережі змінного струму з частотою 50 Гц і напругою 220 В.

За класифікацією приміщень за рівнем електробезпеки (відповідно до НПАОП 0.00-1.28-10) робоче приміщення належить до категорії без підвищеної небезпеки електротравматизму.

ЕОМ та комплектуючі становлять потенційну небезпеку для працівника. Враження струмом може трапитися в результаті торкання до відкритих частин, які перебувають під напругою, при ушкодженні мережних шнурів, при пробої, короткому замиканні, або в результаті необережних дій людини.

ЕОМ відносяться до установок, які споживають напругу < 1000 В. Корпуси сучасних ЕОМ виготовлені із пластмас (передня панель, з якої працює оператор) і металу (верхня кришка й задня панель). Це може призвести до електротравми, при торканні людини до металевих частин у випадку пробою на корпус. Тому конструкцією передбачена спеціальна мережна вилка із трьома контактами (два контакти служать для підключення живлення, а третій - для підключення до проводу на землю) у системі занулення.

Корпуса дисплеїв виготовляються з матеріалів, що не проводять струм, а живлення здійснюється за допомогою спеціального кабелю, що підключається до системного блоку. Це служить для виключення ймовірності ураження струмом.

Робочі місця розташовані на відстані 2 та 3 метра від батареї, що виключає випадкові торкання до батареї.

Висновок: електрообладнання, що використовується у робочому приміщенні, відповідає вимогам електробезпеки для побутових приладів і не вимагають додаткових методів захисту.

4.3.6 Надзвичайні ситуації. Пожежна безпека

Серед ймовірних надзвичайних ситуацій (стихійних лих, аварій, катастроф, впливів зброї масового ураження), найбільш вірогідною в робочому приміщенні є пожежа.

У приміщенні знаходяться наступні речі, які можуть горіти у випадку пожежі:

- папір;
- ПК;
- шафи;
- столи;
- стільці;
- підлога.

За класифікацією приміщень щодо пожежної небезпеки робоче приміщення відноситься до категорії В (важкозаймисті тверді й волокнисті речовини й матеріали) і до класу П-IIа.

Приміщення обладнане пожежною сигналізацією РУОП-1. У приміщенні знаходяться 2 вуглекислотних вогнегасника ОУ-5 (відповідно до норми: 2 шт. на 20м^2).

У разі виникнення пожежі, загрози вибуху або руйнування будинку працівникам необхідно негайно звільнити офісне приміщення згідно плану евакуації. План евакуації зображено на рисунку 4.4 і розташовано на видному місці у приміщенні.

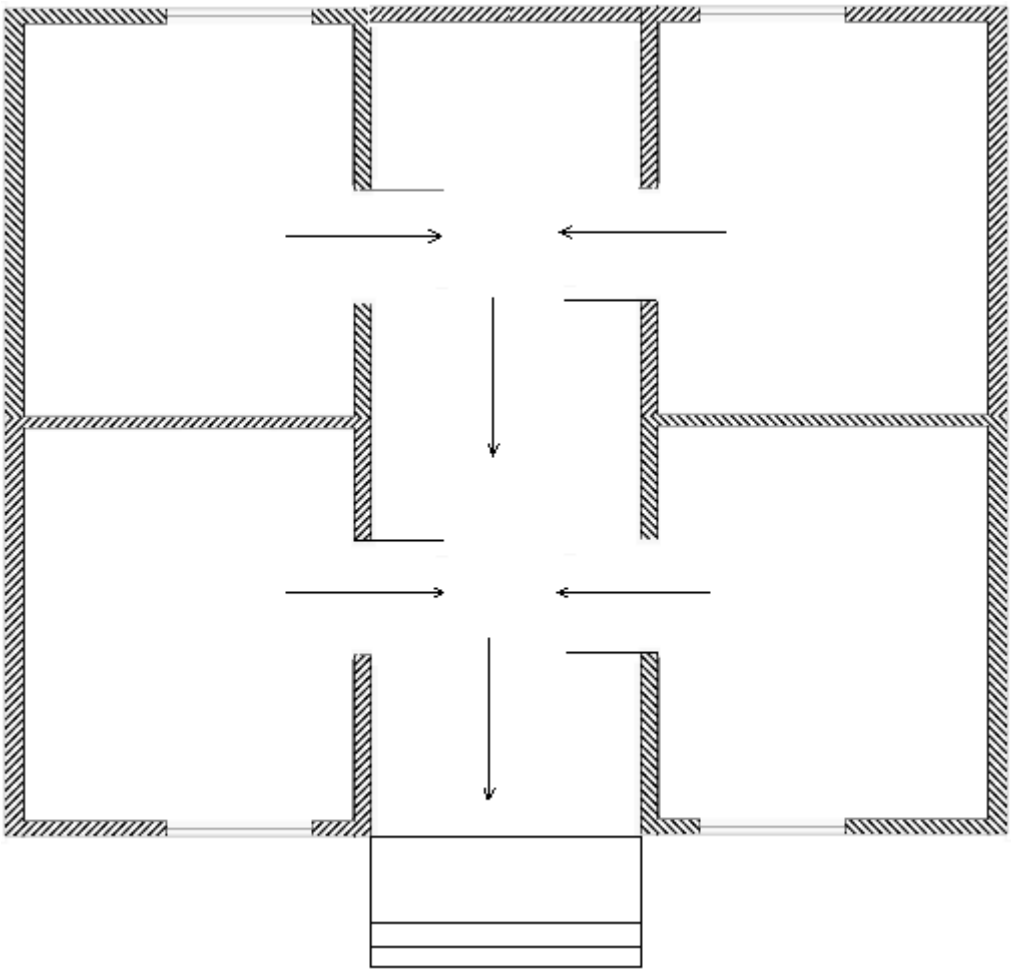


Рисунок 4.4 – План евакуації приміщення

Таблиця 4.10 – Характеристики еваковиходів

Характеристика	Існуючі значення	Нормативні значення
Висота дверних прорізів, м	2	не менш 2,0
Ширина дверних прорізів, м	1,3	не менш 0,8
Ширина проходу для евакуації, м	2,0	не менш 1,0
Ширина коридору, м	2,5	не менш 2,0
Число виходів з коридору	1	не менш 2

Згідно до Таблиці 4.10 маємо, що характеристики еваковиходів відповідають нормативним вимогам. Необхідність встановлення у автоматичних

датчиках пожежі відсутня, через те що у денний час в приміщенні знаходиться людина (що дає змогу завчасно виявити ознаки пожежі), а на ніч техніка не залишається ввімкненою та приміщення замикається. Проаналізувавши існуючі умови протипожежної безпеки й відповідність їх нормативам, можна зробити висновок, що в даному приміщенні виконані всі основні норми пожежної безпеки, крім числа виходів з коридору. Зазначені норми взяті з ДБН В.1.1–7–2002.

Висновки до розділу 4

У даному розділі проведено аналіз умов праці та аналіз ймовірних шкідливих факторів, серед яких порушення мікроклімату, неправильне освітлення, ненормальний рівень шуму, вплив шкідливих випромінювань, небезпека пожежі та ураження електричним струмом. Рівень шуму, повітря робочої зони, освітлення, електрообладнання відповідають нормам. Взимку у приміщенні працює опалення. Монітори сертифіковані за стандартом TCO/5.0, що також відповідає нормам. Приміщення обладнане пожежною сигналізацією РУОП-1, встановлено 2 вуглекислотних вогнегасника. На видному місці розташовано план евакуації, якого слід дотримуватися в разі загрози виникнення пожежі, вибуху та інших надзвичайних ситуацій.

ВИСНОВКИ

В роботі виконано огляд потокових шифрів із самосинхронізацією, 2 з яких були учасниками проекту eSTREAM, третій був представлений на конференції Indocrypt'03, проаналізовано алгоритми їх роботи з точки зору їх параметрів, конструктивних особливостей, наявності криптоатак. Перелічені основні елементарні операції, використані в згаданих шифрах, нові типи елементів їх будови, зокрема використання клітинних автоматів як альтернативу регістрам зсуву. Програмно реалізовано шифр HBB на основі об'єктно-орієнтованого підходу, з допомогою структурного підходу реалізовано атаку на основі вибраного ШТ для шифру SSS, запропонована її модифікація, здійснено порівняльну характеристику атак на основі вибраного ШТ для шифрів SSS та HBB.

Проведений аналіз дає можливість визначити певні тенденції розвитку сучасних потокових шифрів із самосинхронізацією. Серед них можна назвати такі:

- широка різноманітність конструктивних особливостей сучасних потокових шифрів із самосинхронізацією;
- використання клітинних автоматів як альтернативу регістрам зсуву, які донедавна ще практично не використовувались у криптографії;
- схожі риси потокових шифрів із самосинхронізацією до відповідних їм блочних шифрів, що працюють в режимі зворотного зв'язку за ШТ ;
- перспективність розвитку потокових шифрів з самосинхронізацією, незважаючи на перевагу синхронних алгоритмів серед сучасних потокових шифрів;
- необхідність запровадження інтегрального тестування взаємодії елементів шифрів як простий спосіб своєчасного виявлення недоліків, що призводять до можливості атак, зокрема атак на основі вибраного ШТ.

ПЕРЕЛІК ПОСИЛАНЬ

1 Савчук М. М. Математичні методи захисту інформації / М. М. Савчук, Л. О. Завадська. – Київ: НТУУ "КПІ", 2008. – 128 с. – (Курс лекцій).

2 Archived eSTREAM Phase 1 page of SSS [Електронний ресурс] – Режим доступу до ресурсу: http://www.ecrypt.eu.org/stream/sss.html.

3 Daemen J. Chosen Ciphertext Attack on SSS [Електронний ресурс] / J. Daemen, J. Lano. – 2005. – Режим доступу до ресурсу: http://www.ecrypt.eu.org/stream/papersdir/044.pdf.

4 Daemen J. The Self-synchronizing Stream Cipher Moustique / J. Daemen, P. Kitsos // New Stream Cipher Designs / J. Daemen, P. Kitsos., 2008. – (Lecture Notes in Computer Science). – С. 210–223.

5 E. Dawson, W. Millan, L. Burnett, G. Carter, On the Design of 8*32 S-boxes. Information Systems Research Centre, Queensland University of Technology, 1999.

6 E. Kasper, V. Rijmen, T.Biorstad, C. Rechberger, M. Robshow, G. Sekar. Correlated Keystreams in Moustique. Africacrypt 2008, LNCS.

7 eStream'08 [Електронний ресурс] – Режим доступу до ресурсу: http://www.ecrypt.eu.org/stream/.

8 FIPS 185- Escrowed Encryption Standard [Електронний ресурс] – Режим доступу до ресурсу: http://www.itl.nist.gov.fipspubs/fip185.htm.

9 Hawkes P. Primitive Specification for SSS [Електронний ресурс] / P. Hawkes, G. G. Rose, M. Paddon. – 2005. – Режим доступу до ресурсу: http://www.ecrypt.eu.org/stream/ciphers/sss/sss.pdf.

10 Indocrypt'03 [Електронний ресурс] – Режим доступу до ресурсу: http://www.informatik.uni-trier.de/~ley/db/conf/indocrypt/indocrypt2003.html.

11 Joux A. Two attacks against the HBB Stream Cipher / A. Joux, F. Muller. // FSE Proceedings of the 12th international conference on Fast Software Encryption. – 2005. – №5. – С. 330–341.

12 Masoodi F. SOBER Family of Stream Ciphers: A Review / F. Masoodi, S. Alam, M. Bokhari. // International Journal of Computer Applications.

13 NESSI [Електронний ресурс] – Режим доступу до ресурсу: http://ru.wikipedia.org/wiki/NESSIE.

14 Robshaw M. Stream Ciphers [Електронний ресурс] / M.J.B. Robshaw // RSA Laboratories. – 1995. – Режим доступу до ресурсу: ftp://nic.funet.fi/.m/archive1e/idea.sec.dsi.unimi.it/rsa.com/pdfs/tr701.pdf.gz.

15 Sarkar P. Hiji-bij-bij: A new Stream Cipher with Self-Synchronizing and MAC Modes of Operation / Palash Sarkar – India: Indocrypt, 2003. – (Progress in Cryptology). – (LNCS 2904).

16 S. Tezuka, M. Fushimi. A method of designing cellular automata as pseudo random number generators for built-in self-test for VLSI. Finite Fields: Theory. Applications and Algorithms. Contemporary Mathematics. AMS. 363-367. 1994.

17 The Birthday Paradox [Електронний ресурс] – Режим доступу до ресурсу: http://en.wikipedia.org/wiki/Birthday_problem.

ДОДАТКИ

Додаток А

Код атаки на шифр SSS

Файл SSSproject.cpp

```
1   #include "stdafx.h"
2   #include <stdlib.h>
3   #include "sss.h"
4   #include "ssssbox.h"
5   #include "sssmultab.h"
6   #include <iostream>
7
8   // Число додаткових патернів для знаходження a_H
9   #define NumberOfExtraPatterns 8
10
11  /* платформо-незалежні little-endian 2-байтові слова */
12  #define B(x,i) ((UCHAR)(((x) >> (8*i)) & 0xFF))
13  #define BYTE2WORD(b) ( \
14          (((WORD)((b)[1]) & 0xFF)<<8) | \
15          (((WORD)((b)[0]) & 0xFF)) \
16  )
17  #define WORD2BYTE(w, b) { \
18          (b)[1] = B(w,1); \
19          (b)[0] = B(w,0); \
20  }
21  #define XORWORD(w, b) { \
22          (b)[1] ^= B(w,1); \
23          (b)[0] ^= B(w,0); \
24  }
25
26  /* залежна від ключа таблиця Sbox */
27  WORD
28  Sfunc(UCHAR *Key, int KeyLength, WORD w)
29  {
30      register int     i;
31      WORD             t;
32      UCHAR            b;
33
34      t = 0;
35      b = HIGHBYTE(w);
36      for (i = 0; i < KeyLength; ++i) {
37         b = ftable[b ^ Key[i]];
38         t ^= ROTL(Qbox[b], i);
39      }
40      return ((b << (WORDBITS-8)) | (t & LOWMASK)) ^ (w & LOWMASK);
41  }
42
43  /* зміна стану регістру зсуву = 1 цикл */
44  static void
45  cycle(sss_ctx *State, WORD ctxt)
46  {
47      register int     i;
48
49      for (i = 0; i < N-1; ++i)
50         State->ShiftRegister[i] = State->ShiftRegister[i+1];
51      State->ShiftRegister[16] = ctxt;
52      State->ShiftRegister[14] += SBoxFromWord(State, ROTR(ctxt, 8));
53      State->ShiftRegister[12] = SBoxFromWord(State, State->ShiftRegister[12]);
54      State->ShiftRegister[1] = ROTR(State->ShiftRegister[1], 8);
55  }
56
```

```
57  /* Нелінійне перетворення для деяких частин шифру. */
58  static WORD
59  nltap(sss_ctx *c)
60  {
61      register WORD    t;
62
63      t = c->ShiftRegister[0] + c->ShiftRegister[16];
64      t = SBoxFromWord(c,t) + c->ShiftRegister[1] + c->ShiftRegister[6] + c-
65  >ShiftRegister[13];
66      t = ROTR(t, 8);
67      return SBoxFromWord(c,t) ^ c->ShiftRegister[0];
68  }
69
70  void
71  sss_key(sss_ctx *State, UCHAR Key[], int KeyLength)
72  {
73      int i;
74      if (KeyLength > MAXKEY)
75         abort();
76      for (i = 0; i < 256; ++i)
77         State->SBox[i] = Sfunc(Key, KeyLength, (WORD)(i << (WORDBITS-8))) ^ (i <<
78  (WORDBITS-8));
79
80      sss_nonce(State, (UCHAR *)0, 0);
81  }
82
83  void
84  sss_key(sss_ctx *c, UCHAR key[], int keylen)
85  {
86      int           i;
87
88      if ((c->keylen = keylen) > MAXKEY)
89         abort();
90      for (i = 0; i < keylen; ++i)
91         c->key[i] = key[i];
92      sss_nonce(c, (UCHAR *)0, 0);
93  }
94
95  /* Ініцілізуючий вектор */
96  void
97  sss_nonce(sss_ctx *State, UCHAR nonce[], int nlen)
98  {
99      int              i;
100     UCHAR     nb[2];
101
102     if ((nlen % WORDBYTES) != 0)
103        abort();
104     for (i = 0; i < N; ++i)
105        State->ShiftRegister[i] = State->CRC[i] = 0;
106
107     for (i = 0; i < nlen; i += WORDBYTES) {
108        nb[0] = nonce[i];
109        nb[1] = nonce[i+1];
110        sss_decrypt(State, nb, WORDBYTES);
111     }
112
113     nb[0] = nb[1] = 0;
114     for (i = 0; i < N; ++i) {
115        sss_maconly(State, nb, WORDBYTES);
116     }
117
118     State->NumberOfBitsBuffered = 0;
119 }
120
121
```

```
122     /* Зашифрування */
123     void
124     sss_enconly(sss_ctx *c, UCHAR *buf, int nbytes)
125     {
126         WORD      t = 0;
127
128         if (c->NumberOfBitsBuffered != 0) {
129            while (c->NumberOfBitsBuffered != 0 && nbytes != 0) {
130                *buf ^= c->StreamBuf & 0xFF;
131                c->CipherTextBuf ^= *buf << (WORDBITS - c->NumberOfBitsBuffered);
132                c->StreamBuf >>= 8;
133                ++buf;
134                c->NumberOfBitsBuffered -= 8;
135                --nbytes;
136            }
137            if (c->NumberOfBitsBuffered != 0)
138                return;
139            cycle(c, c->CipherTextBuf);
140         }
141
142         while (nbytes >= WORDBYTES)
143         {
144            t = nltap(c) ^ BYTE2WORD(buf);
145            WORD2BYTE(t, buf);
146            cycle(c, t);
147            buf += WORDBYTES;
148            nbytes -= WORDBYTES;
149         }
150
151         if (nbytes != 0) {
152            c->StreamBuf = nltap(c);
153            c->CipherTextBuf = 0;
154            c->NumberOfBitsBuffered = WORDBITS;
155            while (c->NumberOfBitsBuffered != 0 && nbytes != 0) {
156                *buf ^= c->StreamBuf & 0xFF;
157                c->StreamBuf >>= 8;
158                c->CipherTextBuf ^= *buf << (WORDBITS - c->NumberOfBitsBuffered);
159                c->NumberOfBitsBuffered -= 8;
160                --nbytes;
161            }
162         }
163     }
164
165
166
167     /* Розшифрування */
168     void
169     sss_deconly(sss_ctx *State, UCHAR *CipherTextBuffer, int CipherTextLength)
170     {
171         WORD      t = 0, t2 = 0;
172
173         if (State->NumberOfBitsBuffered != 0) {
174            while (State->NumberOfBitsBuffered != 0 && CipherTextLength != 0) {
175                State->CipherTextBuf ^= *CipherTextBuffer << (WORDBITS - State-
176     >NumberOfBitsBuffered);
177                *CipherTextBuffer ^= State->StreamBuf & 0xFF;
178                State->StreamBuf >>= 8;
179                State->NumberOfBitsBuffered -= 8;
180                ++CipherTextBuffer;
181                --CipherTextLength;
182            }
183            if (State->NumberOfBitsBuffered != 0)
184                return;
185            cycle(State, State->CipherTextBuf);
186         }
```

```
187         while (CipherTextLength >= WORDBYTES)
188         {
189           t = nltap(State);
190            t2 = BYTE2WORD(CipherTextBuffer);
191            cycle(State, t2);
192            t ^= t2;
193            WORD2BYTE(t, CipherTextBuffer);
194            CipherTextBuffer += WORDBYTES;
195            CipherTextLength -= WORDBYTES;
196         }
197
198         if (CipherTextLength != 0) {
199            State->StreamBuf = nltap(State);
200            State->CipherTextBuf = 0;
201            State->NumberOfBitsBuffered = WORDBITS;
202            while (State->NumberOfBitsBuffered != 0 && CipherTextLength != 0) {
203                State->CipherTextBuf ^= *CipherTextBuffer << (WORDBITS - State-
204      >NumberOfBitsBuffered);
205                *CipherTextBuffer ^= State->StreamBuf & 0xFF;
206                State->StreamBuf >>= 8;
207                State->NumberOfBitsBuffered -= 8;
208                --CipherTextLength;
209            }
210         }
211      }
212
213      int _tmain(int argc, _TCHAR* argv[])
214      {
215        sss_ctx State;      // внутрішній стан, який намагаємось визначити
216        sss_ctx StateGuess; // стан-здогад, що використовується в ході перебору
217        UCHAR CipherText[36*256 + 36*NumberOfExtraPatterns]; // обраний ШТ
218        UCHAR PlainText[36*256 + 36*NumberOfExtraPatterns];  // ВТ, що відповідає обраному
219      ШТ
220        int i,j,CipherTextLength,correct;
221        int ctr_aL, ctr_SaL;
222        UCHAR aL, aH, aHg1,aHg2;
223        UCHAR SboxInput;
224        WORD SboxGuess;
225        WORD a, Rotated_a, a_plus_i, Rotated_a_plus_i;
226        int EverythingCorrect;
227
228        CipherTextLength = 36*256 + 36*NumberOfExtraPatterns;
229
220        // Нехай ключ буде складатись лише із нулів.
231        UCHAR ZeroKey[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
232        sss_key(&State, ZeroKey, 16);
233
234        // ініціалізація масиву ШТ лише нулями
235        for (i = 0; i < CipherTextLength; i++)
236               CipherText[i] = 0;
237
238        for (i = 0; i < 256; i++)
239               CipherText[36*i + 26] = i;        // 256 модифікацій в молодший байт r[13]
240
241        for (i = 0; i < NumberOfExtraPatterns; i++)
242               CipherText[256*36 + 36*i + 27] = i; // 8 модифікацій старшого байту r[13]
243
244        // Оракул розшифровує Відриктий Текст
245        for (i = 0; i < CipherTextLength; i++)
246               PlainText[i] = CipherText[i];
247
248        sss_deconly(&State, PlainText, CipherTextLength);
249
250        for (ctr_aL = 0; ctr_aL < 256; ctr_aL++)
251        {
```

```
252         aL = ctr_aL;
253         // визначаємо a_H
254         aH = 0;
255         // перебір aH
256         for (i = 0; i < 256; i++)
257            {
258           aHg1 = i;
259           correct = 1;
260           for (j = 0; j < NumberOfExtraPatterns; j++)
261              {
262                   aHg2 = aHg1 + j;
263                   if ( (PlainText[34]^PlainText[256*36 + 36*j + 34]) != (aHg1^aHg2) )
264                         correct = 0;
265           }
266           if (correct)
267                     aH = aHg1;
268         }
269         a = (aH<<8)^aL;
270         Rotated_a = (aL<<8)^aH;
271         for (ctr_SaL = 0; ctr_SaL < 65536; ctr_SaL++)
272            {
273           // Якщо помилка, вертаємо таблицю здогад до початкових значень
274           for (i = 0; i < 256; i++)
275                     StateGuess.SBox[i] = 0;
276
277           StateGuess.SBox[aL] = ctr_SaL;
278
279           // Дізнаємось значення усієї таблиці SBox
280           for (i = 1; i < 256; i++)
281              {
282                   a_plus_i = a+i;
283                   Rotated_a_plus_i = ( (a_plus_i & 0xff) << 8) ^ ( (a_plus_i &
284     0xff00) >> 8);
285                   SboxInput = (aL + i);
286                   StateGuess.SBox[SboxInput] = (PlainText[35]<<8) ^ PlainText[34]^
287     (PlainText[36*i+35]<<8) ^ PlainText[36*i+34] ^ StateGuess.SBox[aL]^ Rotated_a ^
288     Rotated_a_plus_i;
289           }
290           // Друк рішення у разі, якщо воно вірне
291           EverythingCorrect = 1;
292
293           for (i=0;i<256;i++)
294              {
295                   if (State.SBox[i] != StateGuess.SBox[i])
296                         {
297                               EverythingCorrect = 0;
298                               i = 256;
299                         }
300              }
301
302              if (EverythingCorrect)
303              {
304                   printf("The key has been recovered entirely!\n");
305                   for (i = 0; i < 256; i++)
306
307            std::cout<<i<<"\t"<<State.SBox[i]<<"\t"<<StateGuess.SBox[i]<<std::endl;
308                   ctr_aL = 256;
309                   ctr_SaL = 65536; // Кінець.
310           }    }     }
311       getchar();
312       return 1; }
```

Додаток Б

Open Source бібліотека для реалізації функцій шифру SSS

Файл «sss.h»

```
1   #ifndef _SSS_DEFINED
2   #define _SSS_DEFINED 1
3
4   #define N 17
5   #define MAXKEY 128/8
6   #define WORDBITS 16
7   #define WORDBYTES (WORDBITS >> 3)
8   #define WORD unsigned short
9   #define UCHAR unsigned char
10
11  #define SBOX_PRECOMP 1
12
13  typedef struct {
14      WORD      ShiftRegister[N];          /* Working storage for the shift register
15  */
16      WORD      CRC[N];              /* working storage for the CRC register */
17  #if SBOX_PRECOMP
18      WORD      SBox[256];    /* key dependent SBox */
19  #else
20      UCHAR     key[MAXKEY];  /* copy of key */
21      int               keylen;              /* length of key in bytes */
22  #endif /*SBOX_PRECOMP*/
23      WORD      StreamBuf;           /* partial word stream buffer */
24      WORD      CipherTextBuf;               /* partial word ciphertext buffer */
25      WORD      MacBuf;              /* partial word MAC input buffer */
26      int               NumberOfBitsBuffered;             /* number of part-word stream
27  bits buffered */
28  } sss_ctx;
29
30  #if SBOX_PRECOMP
31  /* Run a word through the SBox, assuming it is precalculated */
32  #define SBoxFromWord(State,Word) ((State->SBox[HIGHBYTE(Word)]) ^ (Word))
33  #else
34  #define SBoxFromWord(c,w) Sfunc((c)->key, (c)->keylen, (WORD)(w))
35  #endif /*SBOX_PRECOMP*/
36
37  #define HIGHBYTE(w) (((w) >> (WORDBITS - 8)) & 0xFF)
38  #define LOWMASK 0x00FF /* all the word except the high byte */
39  #define ROTL(w,x) (((w) << (x))|((w) >> (16 - (x))))
40  #define ROTR(w,x) (((w) >> (x))|((w) << (16 - (x))))
41
42  /* interface definitions */
43  void sss_key(sss_ctx *c, UCHAR key[], int keylen);     /* set key */
44  void sss_nonce(sss_ctx *c, UCHAR nonce[], int nlen);   /* set nonce */
45  void sss_enconly(sss_ctx *c, UCHAR *buf, int nbytes); /* stream encryption */
46  void sss_deconly(sss_ctx *c, UCHAR *buf, int nbytes); /* stream decryption */
47  void sss_maconly(sss_ctx *c, UCHAR *buf, int nbytes); /* accumulate MAC */
48  void sss_encrypt(sss_ctx *c, UCHAR *buf, int nbytes); /* encrypt + MAC */
49  void sss_decrypt(sss_ctx *c, UCHAR *buf, int nbytes); /* decrypt + MAC */
50  void sss_finish(sss_ctx *c, UCHAR *buf, int nbytes);  /* finalise MAC */
51
52  #endif /* _SSS_DEFINED */
```

Додаток В

Open Source бібліотека для реалізації функцій шифру SSS [#]

Файл «sssbox.h»

```
/*
 * 8->16 Sbox generated by Millan et. al. at Queensland University of
 * Technology. See: E. Dawson, W. Millan, L. Burnett, G. Carter,
 * "On the Design of 8*32 S-boxes". Unpublished report, by the
 * Information Systems Research Centre,
 * Queensland University of Technology, 1999.
 *
 * Used in SSS to generate the key-dependent S-box.
 */
WORD Qbox[256] = {
    0x1887,   0x435c,   0xc042,   0x6ef4,
    0xee20,   0xfed3,   0xc502,   0xe8ae,
    0xe9d9,   0x38d4,   0x9b5d,   0xdf3c,
    0x4249,   0x3963,   0x429f,   0x2c35,
    0x0325,   0xdd70,   0x3ded,   0xdc5e,
    0x5b42,   0x12bf,   0xd78c,   0xb26b,
    0x1b9a,   0x8146,   0x8ec5,   0xc28f,
    0x5c0f,   0x101c,   0xb082,   0x29e1,
    0x43de,   0x99fc,   0xbc4b,   0x15dd,
    0x03fa,   0xb2de,   0x3342,   0xe7c3,
    0x07ef,   0xebab,   0x859b,   0x2e2f,
    0x71da,   0x269a,   0xc3d1,   0x6b36,
    0xdef2,   0xfc5f,   0xb3a3,   0x6ddf,
    0xb510,   0x85a7,   0x2e71,   0x8816,
    0x1e2a,   0xf6af,   0xc2b3,   0xf55d,
    0x6214,   0x83e3,   0xa6f5,   0x41af,
    0x1f17,   0x99ee,   0x5ec0,   0x16c6,
    0x09a4,   0x6e01,   0x80d9,   0x1418,
    0xf227,   0x8203,   0x9d96,   0xa8c0,
    0xbf6e,   0x7888,   0xfe64,   0x93cd,
    0x0184,   0x4930,   0x4f36,   0x7088,
    0x6c2a,   0xc678,   0x4de7,   0xe759,
    0x248e,   0x446b,   0x9fc2,   0xa895,
    0xc3a1,   0xf170,   0x9155,   0x8a66,
    0x5e69,   0x623e,   0xfa35,   0x68cc,
    0x6acd,   0xe936,   0x2db9,   0x13c1,
    0xb16d,   0xb83c,   0x3763,   0xa911,
    0xbc13,   0x79d7,   0x2fa8,   0x196e,
    0x5476,   0xa866,   0x16ad,   0xc515,
    0xeb3c,   0xa306,   0x99d9,   0x9133,
    0x66dd,   0x5dcd,   0x8f50,   0xb226,
    0xcef3,   0x6189,   0x19b1,   0x3084,
    0xed5c,   0xc58f,   0xe421,   0x47fb,
    0x715e,   0xff99,   0x2f0f,   0x5184,
    0x5e6c,   0x18bc,   0xc6e0,   0xe420,
    0x523f,   0xb8a2,   0x1a6b,   0x8c02,
    0xe354,   0x7d79,   0x7753,   0x9655,
    0x9da1,   0x90a7,   0xc149,   0x7f1c,
    0x9b69,   0xf2b7,   0x58fa,   0x4418,
    0x8c76,   0xd9f0,   0x0d4d,   0xc473,
    0x10e9,   0x4211,   0x082b,   0x334a,
    0x8ed2,   0xcc1b,   0x0ff3,   0x64a0,
    0x5a4f,   0xf8e7,   0xf15f,   0xfe21,
    0x37d6,   0x06f1,   0x0973,   0xde36,
    0x0fa8,   0xab9e,   0xb618,   0x52f5,
    0xeb4f,   0xe343,   0x77dd,   0x3da6,
```

```
    0xd52d,   0x12f8,   0x3360,   0x3ad0,
    0x0f1c,   0xed0b,   0xc1ec,   0x6795,
    0x9d15,   0x46d7,   0xbe76,   0xe0a0,
    0x7c02,   0x49b7,   0xd6ba,   0x7f78,
    0xffbd,   0xca84,   0xf4da,   0x35da,
    0xaa44,   0x52ac,   0x74a7,   0xa46a,
    0x152a,   0xb7aa,   0x5927,   0xb118,
    0x758d,   0x687b,   0xf0b3,   0x54ed,
    0x7271,   0xacab,   0x4aec,   0x94cd,
    0x9e81,   0x3730,   0x21e8,   0x7f0b,
    0xb5d6,   0xadf8,   0x0431,   0xc921,
    0x5d46,   0x0a36,   0x4022,   0xa65e,
    0x70ba,   0xa8cc,   0xae8b,   0x24d5,
    0x8a5a,   0x6b81,   0x2522,   0x1cb8,
    0xfe1d,   0xc697,   0x4f83,   0x6376,
    0x224c,   0x3b35,   0xc0fe,   0xa19a,
    0xb24f,   0xa998,   0x2d71,   0x96a8,
    0x053f,   0xd300,   0xcbcc,   0x3d40,
};

/*
 * skipjack ftable
 */
const unsigned char ftable[256] = {
0xa3,0xd7,0x09,0x83,0xf8,0x48,0xf6,0xf4,0xb3,0x21,0x15,0x78,0x99,0xb1,0xaf,0xf9,
0xe7,0x2d,0x4d,0x8a,0xce,0x4c,0xca,0x2e,0x52,0x95,0xd9,0x1e,0x4e,0x38,0x44,0x28,
0x0a,0xdf,0x02,0xa0,0x17,0xf1,0x60,0x68,0x12,0xb7,0x7a,0xc3,0xe9,0xfa,0x3d,0x53,
0x96,0x84,0x6b,0xba,0xf2,0x63,0x9a,0x19,0x7c,0xae,0xe5,0xf5,0xf7,0x16,0x6a,0xa2,
0x39,0xb6,0x7b,0x0f,0xc1,0x93,0x81,0x1b,0xee,0xb4,0x1a,0xea,0xd0,0x91,0x2f,0xb8,
0x55,0xb9,0xda,0x85,0x3f,0x41,0xbf,0xe0,0x5a,0x58,0x80,0x5f,0x66,0x0b,0xd8,0x90,
0x35,0xd5,0xc0,0xa7,0x33,0x06,0x65,0x69,0x45,0x00,0x94,0x56,0x6d,0x98,0x9b,0x76,
0x97,0xfc,0xb2,0xc2,0xb0,0xfe,0xdb,0x20,0xe1,0xeb,0xd6,0xe4,0xdd,0x47,0x4a,0x1d,
0x42,0xed,0x9e,0x6e,0x49,0x3c,0xcd,0x43,0x27,0xd2,0x07,0xd4,0xde,0xc7,0x67,0x18,
0x89,0xcb,0x30,0x1f,0x8d,0xc6,0x8f,0xaa,0xc8,0x74,0xdc,0xc9,0x5d,0x5c,0x31,0xa4,
0x70,0x88,0x61,0x2c,0x9f,0x0d,0x2b,0x87,0x50,0x82,0x54,0x64,0x26,0x7d,0x03,0x40,
0x34,0x4b,0x1c,0x73,0xd1,0xc4,0xfd,0x3b,0xcc,0xfb,0x7f,0xab,0xe6,0x3e,0x5b,0xa5,
0xad,0x04,0x23,0x9c,0x14,0x51,0x22,0xf0,0x29,0x79,0x71,0x7e,0xff,0x8c,0x0e,0xe2,
0x0c,0xef,0xbc,0x72,0x75,0x6f,0x37,0xa1,0xec,0xd3,0x8e,0x62,0x8b,0x86,0x10,0xe8,
0x08,0x77,0x11,0xbe,0x92,0x4f,0x24,0xc5,0x32,0x36,0x9d,0xcf,0xf3,0xa6,0xbb,0xac,
0x5e,0x6c,0xa9,0x13,0x57,0x25,0xb5,0xe3,0xbd,0xa8,0x3a,0x01,0x05,0x59,0x2a,0x46
};
```

Додаток Г

Open Source бібліотека для реалізації функцій шифру SSS [#]

Файл «sssmultab.h»

```
/* tables for multiplication by 0x100 in GF(2^16) mod 0x500F */
unsigned short tab500F[256] = {
0x0000, 0x500F, 0xA01E, 0xF011,
0x1033, 0x403C, 0xB02D, 0xE022,
0x2066, 0x7069, 0x8078, 0xD077,
0x3055, 0x605A, 0x904B, 0xC044,
0x40CC, 0x10C3, 0xE0D2, 0xB0DD,
0x50FF, 0x00F0, 0xF0E1, 0xA0EE,
0x60AA, 0x30A5, 0xC0B4, 0x90BB,
0x7099, 0x2096, 0xD087, 0x8088,
```

```
11      0x8198, 0xD197, 0x2186, 0x7189,
12      0x91AB, 0xC1A4, 0x31B5, 0x61BA,
13      0xA1FE, 0xF1F1, 0x01E0, 0x51EF,
14      0xB1CD, 0xE1C2, 0x11D3, 0x41DC,
15      0xC154, 0x915B, 0x614A, 0x3145,
16      0xD167, 0x8168, 0x7179, 0x2176,
17      0xE132, 0xB13D, 0x412C, 0x1123,
18      0xF101, 0xA10E, 0x511F, 0x0110,
19      0x533F, 0x0330, 0xF321, 0xA32E,
20      0x430C, 0x1303, 0xE312, 0xB31D,
21      0x7359, 0x2356, 0xD347, 0x8348,
22      0x636A, 0x3365, 0xC374, 0x937B,
23      0x13F3, 0x43FC, 0xB3ED, 0xE3E2,
24      0x03C0, 0x53CF, 0xA3DE, 0xF3D1,
25      0x3395, 0x639A, 0x938B, 0xC384,
26      0x23A6, 0x73A9, 0x83B8, 0xD3B7,
27      0xD2A7, 0x82A8, 0x72B9, 0x22B6,
28      0xC294, 0x929B, 0x628A, 0x3285,
29      0xF2C1, 0xA2CE, 0x52DF, 0x02D0,
30      0xE2F2, 0xB2FD, 0x42EC, 0x12E3,
31      0x926B, 0xC264, 0x3275, 0x627A,
32      0x8258, 0xD257, 0x2246, 0x7249,
33      0xB20D, 0xE202, 0x1213, 0x421C,
34      0xA23E, 0xF231, 0x0220, 0x522F,
35      0xA67E, 0xF671, 0x0660, 0x566F,
36      0xB64D, 0xE642, 0x1653, 0x465C,
37      0x8618, 0xD617, 0x2606, 0x7609,
38      0x962B, 0xC624, 0x3635, 0x663A,
39      0xE6B2, 0xB6BD, 0x46AC, 0x16A3,
40      0xF681, 0xA68E, 0x569F, 0x0690,
41      0xC6D4, 0x96DB, 0x66CA, 0x36C5,
42      0xD6E7, 0x86E8, 0x76F9, 0x26F6,
43      0x27E6, 0x77E9, 0x87F8, 0xD7F7,
44      0x37D5, 0x67DA, 0x97CB, 0xC7C4,
45      0x0780, 0x578F, 0xA79E, 0xF791,
46      0x17B3, 0x47BC, 0xB7AD, 0xE7A2,
47      0x672A, 0x3725, 0xC734, 0x973B,
48      0x7719, 0x2716, 0xD707, 0x8708,
49      0x474C, 0x1743, 0xE752, 0xB75D,
50      0x577F, 0x0770, 0xF761, 0xA76E,
51      0xF541, 0xA54E, 0x555F, 0x0550,
52      0xE572, 0xB57D, 0x456C, 0x1563,
53      0xD527, 0x8528, 0x7539, 0x2536,
54      0xC514, 0x951B, 0x650A, 0x3505,
55      0xB58D, 0xE582, 0x1593, 0x459C,
56      0xA5BE, 0xF5B1, 0x05A0, 0x55AF,
57      0x95EB, 0xC5E4, 0x35F5, 0x65FA,
58      0x85D8, 0xD5D7, 0x25C6, 0x75C9,
59      0x74D9, 0x24D6, 0xD4C7, 0x84C8,
60      0x64EA, 0x34E5, 0xC4F4, 0x94FB,
61      0x54BF, 0x04B0, 0xF4A1, 0xA4AE,
62      0x448C, 0x1483, 0xE492, 0xB49D,
63      0x3415, 0x641A, 0x940B, 0xC404,
64      0x2426, 0x7429, 0x8438, 0xD437,
65      0x1473, 0x447C, 0xB46D, 0xE462,
66      0x0440, 0x544F, 0xA45E, 0xF451,
67      };
68      #define mul500F(x) (tab500F[(x) >> 8] ^ ((x) << 8))
```

Додаток Ґ

Файл необхідний для реалізації шифру HBB на мові C#

Файл «Program.cs»

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace HBB_Sharp
8   {
9       public class Program
10      {
11          public static UInt32[] KEY = new UInt32[4] { 1, 5, 34, 23 }; // 4 * 4 = 16
12  bytes = 16 * 8 = 128 bits
13          public static UInt32[] KeyStream = new UInt32[4] { 0, 0, 0, 0 }; //
14  refreshes on each round function call
15          public static UInt32[] M = new UInt32[24] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
16  11, 12, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
17          public static UInt32[] C = new UInt32[24] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
18  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
19          public static List<Block> Messages = new List<Block>();
20          public static List<Block> CipherBlocks = new List<Block>();
21
22          static void Main(string[] args)
23          {
24              CipherHelpers.InitBlockList(Messages, ref M);
25              CipherHelpers.InitBlockList(CipherBlocks, ref C);
26
27              CipherHelpers.HBB(CipherHelpers.Action.Encrypt, Messages, CipherBlocks);
28              CipherHelpers.PrintBlocks(Messages);
29              CipherHelpers.HBB(CipherHelpers.Action.Decrypt, Messages, CipherBlocks);
30              CipherHelpers.PrintBlocks(Messages);
31          }
32      }
33  }
```

Додаток Д

Файл необхідний для реалізації шифру HBB на мові C#

Файл «NLC.cs»

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using System.Runtime.InteropServices;
7
8   namespace HBB_Sharp
9   {
10      [StructLayoutAttribute(LayoutKind.Explicit)]
11      public class NLC
```

```
12          {
13              [FieldOffsetAttribute(0)]
14              public UInt32 word;
15
16              [FieldOffsetAttribute(0)]
17              public byte byte0;
18
19              [FieldOffsetAttribute(1)]
20              public byte byte1;
21
22              [FieldOffsetAttribute(2)]
23              public byte byte2;
24
25              [FieldOffsetAttribute(3)]
26              public byte byte3;
27          }
28      }
```

Додаток Е

Файл необхідний для реалізації шифру HBB на мові C#

Файл «Block.cs»

```
1       using System;
2       using System.Collections.Generic;
3       using System.Linq;
4       using System.Text;
5       using System.Threading.Tasks;
6       using System.Runtime.InteropServices;
7
8       namespace HBB_Sharp
9       {
10          public class Block
11          {
12              public UInt32 first;
13              public UInt32 second;
14              public UInt32 third;
15              public UInt32 fourth;
16
17              public Block(ref UInt32[] initBlock, int index)
18              {
19                  first   = initBlock[index + 0];
20                  second  = initBlock[index + 1];
21                  third   = initBlock[index + 2];
22                  fourth  = initBlock[index + 3];
23              }
24
25              public Block()
26              {
27                  first   = 0;
28                  second  = 0;
29                  third   = 0;
30                  fourth  = 0;
31              }
32          }
33      }
```

Додаток Є

Файл необхідний для реалізації шифру HBB на мові C#

Файл «CA.cs»

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using System.Runtime.InteropServices;
7
8   namespace HBB_Sharp
9   {
10      public enum CAorder
11      {
12          first = 1,
13          second = 2
14      }
15
16      public class CA
17      {
18          public UInt32 state0;
19          public UInt32 state1;
20          public UInt32 state2;
21          public UInt32 state3;
22          public UInt32 state4;
23          public UInt32 state5;
24          public UInt32 state6;
25          public UInt32 state7;
26
27          CAorder order;
28
29          UInt32 RULE00 = 0x80ffaf46;
30          UInt32 RULE01 = 0x977969e9;
31          UInt32 RULE02 = 0x71553bb5;
32          UInt32 RULE03 = 0x99be6b2b;
33          UInt32 RULE04 = 0x4b337295;
34          UInt32 RULE05 = 0x2308c787;
35          UInt32 RULE06 = 0xb84c7cce;
36          UInt32 RULE07 = 0x36d501e6;
37
38          UInt32 RULE10 = 0xdd18c62b;
39          UInt32 RULE11 = 0x153df31a;
40          UInt32 RULE12 = 0xc98e86c1;
41          UInt32 RULE13 = 0x910fee24;
42          UInt32 RULE14 = 0x2942d51b;
43          UInt32 RULE15 = 0x4201eb3d;
44          UInt32 RULE16 = 0xc1d1a85f;
45          UInt32 RULE17 = 0x57b8919b;
46
47          public CA(CAorder InOrder)
48          {
49              state0 = 0;
50              state1 = 0;
51              state2 = 0;
52              state3 = 0;
53              state4 = 0;
54              state5 = 0;
55              state6 = 0;
56              state7 = 0;
57              order = InOrder;
```

```
58          }
59
60          public void Exp()
61          {
62              if (order == CAorder.first)
63              {
64                  state0 = Program.KEY[0];
65                  state1 = Program.KEY[1];
66                  state2 = Program.KEY[2];
67                  state3 = Program.KEY[3];
68                  state4 = ~Program.KEY[0];
69                  state5 = ~Program.KEY[1];
70                  state6 = ~Program.KEY[2];
71                  state7 = ~Program.KEY[3];
72              }
73              else
74              {
75                  state0 = ~Program.KEY[0];
76                  state1 = ~Program.KEY[1];
77                  state2 = ~Program.KEY[2];
78                  state3 = ~Program.KEY[3];
79                  state4 = Program.KEY[0];
80                  state5 = Program.KEY[1];
81                  state6 = Program.KEY[2];
82                  state7 = Program.KEY[3];
83              }
84          }
85
86          public void MergeWithCipher(UInt32 C0, UInt32 C1, UInt32 C2, UInt32 C3)
87          {
88              state7 = state7 ^ C3;
89              state5 = state5 ^ C2;
90              state3 = state3 ^ C1;
91              state1 = state1 ^ C0;
92          }
93
94          public void EvolveCA256()
95          {
96              if (order == CAorder.first)
97              {
98                  UInt32 tmp0 = ((state0 << 1) ^ (state1 >> 31)) ^ (RULE00 & state0) ^
99      (state0 >> 1);
100                 UInt32 tmp1 = ((state1 << 1) ^ (state2 >> 31)) ^ (RULE01 & state1) ^
101     ((state1 >> 1) ^ (state0 << 31));
102                 UInt32 tmp2 = ((state2 << 1) ^ (state3 >> 31)) ^ (RULE02 & state2) ^
103     ((state2 >> 1) ^ (state1 << 31));
104                 UInt32 tmp3 = ((state3 << 1) ^ (state4 >> 31)) ^ (RULE03 & state3) ^
105     ((state3 >> 1) ^ (state2 << 31));
106                 UInt32 tmp4 = ((state4 << 1) ^ (state5 >> 31)) ^ (RULE04 & state4) ^
107     ((state4 >> 1) ^ (state3 << 31));
108                 UInt32 tmp5 = ((state5 << 1) ^ (state6 >> 31)) ^ (RULE05 & state5) ^
109     ((state5 >> 1) ^ (state4 << 31));
110                 UInt32 tmp6 = ((state6 << 1) ^ (state7 >> 31)) ^ (RULE06 & state6) ^
111     ((state6 >> 1) ^ (state5 << 31));
112                 UInt32 tmp7 = (state7 << 1) ^ (RULE07 & state7) ^ ((state7 >> 1) ^
113     (state6 << 31));
114                 state0 = tmp0; state1 = tmp1; state2 = tmp2; state3 = tmp3;
115                 state4 = tmp4; state5 = tmp5; state6 = tmp6; state7 = tmp7;
116             }
117             else
118             {
119                 UInt32 tmp0 = ((state0 << 1) ^ (state1 >> 31)) ^ (RULE10 & state0) ^
120     (state0 >> 1);
121                 UInt32 tmp1 = ((state1 << 1) ^ (state2 >> 31)) ^ (RULE11 & state1) ^
122     ((state1 >> 1) ^ (state0 << 31));
```

```
123                 UInt32 tmp2 = ((state2 << 1) ^ (state3 >> 31)) ^ (RULE12 & state2) ^
124 ((state2 >> 1) ^ (state1 << 31));
125                 UInt32 tmp3 = ((state3 << 1) ^ (state4 >> 31)) ^ (RULE13 & state3) ^
126 ((state3 >> 1) ^ (state2 << 31));
127                 UInt32 tmp4 = ((state4 << 1) ^ (state5 >> 31)) ^ (RULE14 & state4) ^
128 ((state4 >> 1) ^ (state3 << 31));
129                 UInt32 tmp5 = ((state5 << 1) ^ (state6 >> 31)) ^ (RULE15 & state5) ^
130 ((state5 >> 1) ^ (state4 << 31));
131                 UInt32 tmp6 = ((state6 << 1) ^ (state7 >> 31)) ^ (RULE16 & state6) ^
132 ((state6 >> 1) ^ (state5 << 31));
133                 UInt32 tmp7 = (state7 << 1) ^ (RULE17 & state7) ^ ((state7 >> 1) ^
134 (state6 << 31));
135                 state0 = tmp0; state1 = tmp1; state2 = tmp2; state3 = tmp3;
136                 state4 = tmp4; state5 = tmp5; state6 = tmp6; state7 = tmp7;
137             }
138         }
139     }
140 }
```

Додаток Ж

Файл необхідний для реалізації шифру HBB на мові C#

Файл «CipherHelpers.cs»

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace HBB_Sharp
8  {
9      public static class CipherHelpers
10     {
11         public static UInt32 MASK0 = 0x55555555;
12         public static UInt32 MASK1 = 0x33333333;
13
14         // S-Box
15         public static byte[] byteSub = new byte[256]
16         {
17        0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
18        0xca,0x82,0xc9,0x7b,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
19        0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
20        0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
21        0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
22        0x53,0xd1,0x00,0xed,0x20,0xfc,0xd1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
23        0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
24        0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
25        0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
26        0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
27        0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
28        0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
29        0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
30        0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
31        0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
32        0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16
33         };
34
35
36
```

```
37          public static void NLSub(List<NLC> allNLC)
38          {
39              allNLC[0].byte0 = byteSub[allNLC[0].byte0];
40              allNLC[0].byte1 = byteSub[allNLC[0].byte1];
41              allNLC[0].byte2 = byteSub[allNLC[0].byte2];
42              allNLC[0].byte3 = byteSub[allNLC[0].byte3];
43              allNLC[1].byte0 = byteSub[allNLC[1].byte0];
44              allNLC[1].byte1 = byteSub[allNLC[1].byte1];
45              allNLC[1].byte2 = byteSub[allNLC[1].byte2];
46              allNLC[1].byte3 = byteSub[allNLC[1].byte3];
47              allNLC[2].byte0 = byteSub[allNLC[2].byte0];
48              allNLC[2].byte1 = byteSub[allNLC[2].byte1];
49              allNLC[2].byte2 = byteSub[allNLC[2].byte2];
50              allNLC[2].byte3 = byteSub[allNLC[2].byte3];
51              allNLC[3].byte0 = byteSub[allNLC[3].byte0];
52              allNLC[3].byte1 = byteSub[allNLC[3].byte1];
53              allNLC[3].byte2 = byteSub[allNLC[3].byte2];
54              allNLC[3].byte3 = byteSub[allNLC[3].byte3];
55          }
56
57          public static UInt32 CLShift(UInt32 A, int I)
58          {
59              UInt32 result = (((A) >> (32 - I)) ^ ((A) << (I)));
60              return result;
61          }
62
63          public static void Fold(List<NLC> allNLC)
64          {
65              UInt32 tmp0 = Program.KEY[0] ^ Program.KEY[2];
66              UInt32 tmp1 = Program.KEY[1] ^ Program.KEY[3];
67              allNLC[0].word = tmp0;
68              allNLC[1].word = tmp1;
69              allNLC[2].word = ~tmp0;
70              allNLC[3].word = ~tmp1;
71          }
72
73          public static void transpose32(ref UInt32 arr0, ref UInt32 arr1, ref UInt32
74      arr2, ref UInt32 arr3)
75          {
76              UInt32 tmp0 = arr0 & MASK0;
77              UInt32 tmp1 = arr0 & ~MASK0;
78              UInt32 tmp2 = arr1 & MASK0;
79              UInt32 tmp3 = arr1 & ~MASK0;
80
81              tmp0 = tmp0 << 1;
82              tmp3 = tmp3 >> 1;
83              arr0 = tmp1 ^ tmp3;
84              arr1 = tmp0 ^ tmp2;
85
86              tmp0 = arr2 & MASK0;
87              tmp1 = arr2 & ~MASK0;
88              tmp2 = arr3 & MASK0;
89              tmp3 = arr3 & ~MASK0;
90
91              tmp0 = tmp0 << 1;
92              tmp3 = tmp3 >> 1;
93              arr2 = tmp1 ^ tmp3;
94              arr3 = tmp0 ^ tmp2;
95
96              tmp0 = arr0 & MASK1;
97              tmp1 = arr0 & ~MASK1;
98              tmp2 = arr2 & MASK1;
99              tmp3 = arr2 & ~MASK1;
100
101             tmp0 = tmp0 << 2;
```

```
102             tmp3 = tmp3 >> 2;
103             arr0 = tmp1 ^ tmp3;
104             arr2 = tmp0 ^ tmp2;
105
106             tmp0 = arr1 & MASK1;
107             tmp1 = arr1 & ~MASK1;
108             tmp2 = arr3 & MASK1;
109             tmp3 = arr3 & ~MASK1;
110
111             tmp0 = tmp0 << 2;
112             tmp3 = tmp3 >> 2;
113             arr1 = tmp1 ^ tmp3;
114             arr3 = tmp0 ^ tmp2;
115         }
116
117         public static void Round(CA FirstCA, CA SecondCA, List<NLC> allNLC)
118            {
119             NLSub(allNLC);
120             UInt32 tmp0 = allNLC[0].word ^ allNLC[1].word ^ allNLC[2].word ^
121     allNLC[3].word;
122             allNLC[0].word = CLShift(tmp0 ^ allNLC[0].word, 4);
123             allNLC[1].word = CLShift(tmp0 ^ allNLC[1].word, 12);
124             allNLC[2].word = CLShift(tmp0 ^ allNLC[2].word, 20);
125             allNLC[3].word = CLShift(tmp0 ^ allNLC[3].word, 28);
126             transpose32(ref allNLC[0].word, ref allNLC[1].word, ref allNLC[2].word,
127     ref allNLC[3].word);
128             NLSub(allNLC);
129             FirstCA.EvolveCA256();
130             SecondCA.EvolveCA256();
131             Program.KeyStream[0] = allNLC[0].word ^ FirstCA.state0;
132             Program.KeyStream[1] = allNLC[1].word ^ FirstCA.state7;
133             Program.KeyStream[2] = allNLC[2].word ^ SecondCA.state0;
134             Program.KeyStream[3] = allNLC[3].word ^ SecondCA.state7;
135             allNLC[0].word = allNLC[0].word ^ FirstCA.state3;
136             allNLC[1].word = allNLC[1].word ^ FirstCA.state4;
137             allNLC[2].word = allNLC[2].word ^ SecondCA.state3;
138             allNLC[3].word = allNLC[3].word ^ SecondCA.state4;
139            }
140
141         public static void KeySetup(CA FirstCA, CA SecondCA, List<NLC> allNLC)
142         {
143             UInt32[,] Temp = new UInt32[4, 4];
144             Temp[0,0] = Program.KeyStream[0]; Temp[0,1] = Program.KeyStream[1];
145     Temp[0,2] = Program.KeyStream[2]; Temp[0,3] = Program.KeyStream[3];
146             Round(FirstCA, SecondCA, allNLC);
147             Temp[1, 0] = Program.KeyStream[0]; Temp[1, 1] = Program.KeyStream[1];
148     Temp[1, 2] = Program.KeyStream[2]; Temp[1, 3] = Program.KeyStream[3];
149             Round(FirstCA, SecondCA, allNLC);
150             Temp[2, 0] = Program.KeyStream[0]; Temp[2, 1] = Program.KeyStream[1];
151     Temp[2, 2] = Program.KeyStream[2]; Temp[2, 3] = Program.KeyStream[3];
152             Round(FirstCA, SecondCA, allNLC);
153             Temp[3, 0] = Program.KeyStream[0]; Temp[3, 1] = Program.KeyStream[1];
154     Temp[3, 2] = Program.KeyStream[2]; Temp[3, 3] = Program.KeyStream[3];
155             Round(FirstCA, SecondCA, allNLC);
156
157             FirstCA.state0 = FirstCA.state0 ^ Temp[3, 0];
158             FirstCA.state1 = FirstCA.state1 ^ Temp[3, 1];
159             FirstCA.state2 = FirstCA.state2 ^ Temp[3, 2];
160             FirstCA.state3 = FirstCA.state3 ^ Temp[3, 3];
161             FirstCA.state4 = FirstCA.state4 ^ Temp[2, 0];
162             FirstCA.state5 = FirstCA.state5 ^ Temp[2, 1];
163             FirstCA.state6 = FirstCA.state6 ^ Temp[2, 2];
164             FirstCA.state7 = FirstCA.state7 ^ Temp[2, 3];
165             SecondCA.state0 = SecondCA.state0 ^ Temp[1, 0];
166             SecondCA.state1 = SecondCA.state1 ^ Temp[1, 1];
```

```
167             SecondCA.state2 = SecondCA.state2 ^ Temp[1, 2];
168             SecondCA.state3 = SecondCA.state3 ^ Temp[1, 3];
169             SecondCA.state4 = SecondCA.state4 ^ Temp[0, 0];
170             SecondCA.state5 = SecondCA.state5 ^ Temp[0, 1];
171             SecondCA.state6 = SecondCA.state6 ^ Temp[0, 2];
172             SecondCA.state7 = SecondCA.state7 ^ Temp[0, 3];
173         }
174
175         public static void InitBlockList(List<Block> BlockList, ref UInt32[]
176     inArray)
177         {
178             int NumberOfBlocks = inArray.Length / 4;
179
180             for (int i = 0; i < inArray.Length; i = i + 4)
181             {
182                 Block currentBlock = new Block(ref inArray, i);
183                 BlockList.Add(currentBlock);
184             }
185         }
186
187         public static void Encrypt(CA FirstCA, CA SecondCA, List<NLC> allNLC,
188     List<Block> Messages, List<Block> Ciphers)
189         {
190             int BlockNumber = Messages.Count;
191
192             for (int i = 0; i < BlockNumber; i++)
193                {
194                 Round(FirstCA, SecondCA, allNLC);
195                       // encryption
196                 Ciphers[i].first    = Messages[i].first  ^ Program.KeyStream[0];
197                 Ciphers[i].second   = Messages[i].second ^ Program.KeyStream[1];
198                 Ciphers[i].third    = Messages[i].third  ^ Program.KeyStream[2];
199                 Ciphers[i].fourth   = Messages[i].fourth ^ Program.KeyStream[3];
200                } // end of key generation
201         }
202
203         public static void Decrypt(CA FirstCA, CA SecondCA, List<NLC> allNLC,
204     List<Block> Messages, List<Block> Ciphers)
205         {
206             int BlockNumber = Messages.Count;
207
208             for (int i = 0; i < BlockNumber; i++)
209             {
210                 Round(FirstCA, SecondCA, allNLC);
211                 // decryption
212                 Messages[i].first   = Ciphers[i].first  ^ Program.KeyStream[0];
213                 Messages[i].second  = Ciphers[i].second ^ Program.KeyStream[1];
214                 Messages[i].third   = Ciphers[i].third  ^ Program.KeyStream[2];
215                 Messages[i].fourth  = Ciphers[i].fourth ^ Program.KeyStream[3];
216             }
217         }
218
219         public enum Action
220         {
221             Encrypt = 1,
222             Decrypt = 2
223         }
224
225         public static List<NLC> NLC_generate(int count)
226         {
227             List<NLC> allNLC = new List<NLC>();
228             for (int i = 0; i < count; i++)
229             {
220                 allNLC.Add(new NLC());
231             }
```

```
232             return allNLC;
233         }
234
235         public static void HBB(Action action, List<Block> Messages, List<Block>
236     Ciphers)
237         {
238             List<NLC> allNLC = NLC_generate(4);
239             CA FirstCa = new CA(CAorder.first); CA SecondCa = new
240     CA(CAorder.second);
241
242             FirstCa.Exp(); SecondCa.Exp();
243
244             CipherHelpers.Fold(allNLC);
245
246             for (int i = 0; i <= 12; i++)
247             {
248                 CipherHelpers.Round(FirstCa, SecondCa, allNLC);
249             }
250
251             CipherHelpers.KeySetup(FirstCa, SecondCa, allNLC);
252
253             if (action == Action.Encrypt)
254             {
255                 CipherHelpers.Encrypt(FirstCa, SecondCa, allNLC, Messages, Ciphers);
256             }
257             else if (action == Action.Decrypt)
258             {
259                 CipherHelpers.Decrypt(FirstCa, SecondCa, allNLC, Messages, Ciphers);
260             }
261         }
262
263         public static void PrintBlocks(List<Block> Blocks)
264         {
265             string coma = ", ";
266             foreach (Block block in Blocks)
267                 Console.Write(block.first + coma + block.second + coma + block.third
268     + coma + block.fourth + coma);
269             Console.WriteLine();
270         }
271     }
272 }
```