ІНСТИТУТ СПЕЦІАЛЬНОГО ЗВ’ЯЗКУ ТА ЗАХИСТУ ІНФОРМАЦІЇ
НАЦІОНАЛЬНОГО ТЕХНІЧНОГО УНІВЕРСИТЕТУ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Кваліфікаційна наукова
праця на правах рукопису

ГРИШАКОВ СЕРГІЙ ВОЛОДИМИРОВИЧ

УДК 003.26:004.056.55

ДИСЕРТАЦІЯ

МЕТОД ПОБУДОВИ РАНДОМІЗОВАНИХ ПОТОКОВИХ ШИФРОСИСТЕМ З НЕЛІНІЙНИМ ВИПАДКОВИМ КОДУВАННЯМ

21.05.01 – «Інформаційна безпека держави»

Подається на здобуття наукового ступеня
кандидата технічних наук

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело _________________

Науковий керівник:
ОЛЕКСІЙЧУК Антон Миколайович,
доктор технічних наук, доцент

Київ – 2018

АНОТАЦІЯ

Гришаков С.В. Метод побудови рандомізованих потокових шифросистем з нелінійним випадковим кодуванням. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня кандидата технічних наук за спеціальністю 21.05.01 «Інформаційна безпека держави». – Інститут спеціального зв’язку та захисту інформації Національного технічного університету «Київський політехнічний інститут імені Ігоря Сікорського», Київ, 2018.

Дисертаційна робота присвячена вирішенню актуальної наукової задачі розробки методу побудови рандомізованих потокових шифросистем з нелінійним випадковим кодуванням для забезпечення безпеки державних інформаційних ресурсів.

На основі проведеного аналізу доступних наукових публікацій показано, що неухильний розвиток інформаційних технологій поряд з нарощуванням потужності обчислювальних засобів та появою нових методів криптоаналізу створює потенційну загрозу для спеціальних (військових) додатків, де використовуються потокові шифри. Оскільки швидка заміна алгоритму шифрування, криптографічні слабкості якого виявлені на етапі його експлуатації, як правило, через багато років після його створення, є практично неможливою, видається доцільним створення методів підвищення стійкості потокових шифрів без внесення змін в алгоритми шифрування шляхом застосування додаткових перетворень, які не потребують ключів, можуть бути відносно просто реалізовані та забезпечують науково обґрунтований рівень стійкості систем шифрування в цілому. Одним з таких загальних методів є рандомізація або випадкове кодування джерела відкритих повідомлень. Метод рандомізації є відомим достатньо давно, проте, саме для випадку потокових шифрів, його можливості досліджені не повністю. Імовірно, єдиним відомим прикладом рандомізованих потокових шифросистем (РПШ), які будуються на

регулярній основі та, в принципі, можуть бути використані на практиці, є шифросистеми Міхалевича-Імаї.

В роботі вперше отримано аналітичні оцінки параметрів, що визначають стійкість РПШ Міхалевича-Імаї відносно атак на основі відомих шифрованих повідомлень, а також підібраних векторів ініціалізації. Отримані оцінки дозволяють з'ясувати теоретико-кодовий сенс параметрів, які визначають обчислювальну стійкість цих шифросистем, а також встановити, що їх стійкість може бути значно менше, ніж стверджують їх розробники (в окремих випадках – не вище $2^{18,86}$ операцій), що досягається за рахунок розширення можливостей супротивника при проведенні зазначених атак.

Вперше доведено, що клас РПШ Міхалевича-Імаї (незалежно від будови їх компонент) володіє суттєвою слабкістю, яка полягає в зменшенні кількості інформації (в порівнянні з довжиною блоку шифрувальної гами), що необхідна для відновлення за реальний час символів відкритого тексту. Зазначена властивість дозволяє зробити практично важливий висновок про те, що для відновлення символів відкритого тексту супротивнику достатньо мати лише часткову інформацію про секретний ключ РПШ (в окремих випадках 50 бітів замість 255 бітів), що відбувається за рахунок спільного застосування випадкового і завадостійкого кодування повідомлень лінійними кодами.

Вперше отримано аналітичні межі для швидкості передачі інформації в РПШ Міхалевича-Імаї при заданих обмеженнях щодо ймовірності правильного прийому повідомлень законним користувачем та стійкості шифрування. Зазначені межі, за рахунок застосування оцінок Плоткіна та Бассалиго-Елайєса для швидкості передачі лінійних кодів, дозволяють зробити науково обґрунтований висновок про обмежені можливості РПШ Міхалевича-Імаї з погляду сучасних вимог щодо стійкості та практичності в реальних умовах (зокрема, застосування отриманих меж до РПШ з довжиною ключа 512 бітів показує, що їх стійкість не перевищує 2^{79} операцій незалежно від вибору компонент).

Отримав подальший розвиток метод побудови РПШ, який, на відміну від раніше відомих, базується на застосуванні для випадкового кодування нелінійних відображень або безключевих геш-функцій та дозволяє збільшити стійкість (у 2^{242} і більше разів) в порівнянні з РПШ Міхалевича-Імаї за рахунок розширення класу перетворень, які використовуються в конструкції рандомізатора.

Практичне значення одержаних результатів полягає в тому, що дисертантом розроблено програмні реалізації РПШ з нелінійним випадковим кодуванням на основі нелінійних відображень чи безключевих геш-функцій, що є більш стійкими (у 2^{242} і більше разів) і більш швидкісними (у 125 і більше разів) в порівнянні з раніше відомими РПШ при однаковій довжині вихідного повідомлення. Розроблені реалізації РПШ дозволяють здійснювати процедури зашифровування/розшифровування даних в режимі реального часу та можуть бути використані на практиці у спеціальних (військових) додатках, витоки конфіденційної інформації в яких створюють ризики для інформаційної безпеки держави.

Наукові та практичні результати дисертаційної роботи реалізовані в Службі зовнішньої розвідки України – в результаті виконання НДР “Кета” та в науково-технічних розробках ЗАО “Інститут інформаційних технологій”.

Ключові слова: безпека державних інформаційних ресурсів, потоковий шифр, випадкове кодування, методи побудови рандомізованих потокових шифросистем, обчислювальна стійкість, обґрунтування стійкості.

ABSTRACT

Gryshakov S. Method for designing randomized stream ciphers with nonlinear random coding. – Qualifying scientific work as a manuscript.

Thesis for a Candidate of Technical Science degree in specialty 21.05.01 «Information security of the state». – Institute of Special Communication and

Information Protection of National technical university of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, 2018.

This thesis is devoted to solving actual scientific problem of development the method for designing randomized stream ciphers (RSC) with nonlinear random coding to provide the security of state information resources.

Analysis of available scientific publications was carried out. It showed that a steady development of information technologies along with the growth of computing power and appearing of new cryptanalysis methods creates a potential threat to special (military) applications, which use stream ciphers. Note that fast replacement of the encryption algorithm, whose cryptographic weaknesses are found during its exploitation (since many years after its creation), is practically impossible. Given this fact, it seems advisable to create methods for enhancing the security of stream ciphers (without making changes in the encryption algorithms) by applying additional transformations that are not need keys, can be simply implemented and provide a scientifically reasonable security level of encryption systems in general. One of these common techniques is randomization or random coding of the plaintext source. The randomization technique is known for a long time, but its capabilities for stream ciphers are not fully explored. Probably the only known example of randomized stream ciphers that are built on a regular basis and can be used in practice are the Mihalević-Imai ciphers.

Analytical estimates of the parameters that determine the security of the Mihalević-Imai RSC against known ciphertext attacks and chosen initialization vectors attacks are obtained in the thesis for the first time. The obtained estimates allow to find out the code-theoretic sense of the parameters that determine the computational security of these ciphers as well as to establish that the security of these ciphers can be considerably less than their designers claim (in some cases – no more than $2^{18,86}$ operations). This is achieved by enlarging the capabilities of the adversary in carrying out these attacks.

It was proved for the first time that a class of the Mihalević-Imai RSC (irrespective of the structure of their components) has a significant weakness which consists in reducing the amount of information (in comparison with the length of the keystream block) which is necessary for real-time recovery of the plaintext. This property allows to make a practically important conclusion that it is enough for the adversary to have only partial information about the secret key of the RSC to recover the plaintext (in some cases it is enough to have 50 bits instead of 255 bits). This occurs due to the joint employment of random coding and error-correction coding of messages by linear codes.

Analytical bounds of the transmission rate for the Mihalević-Imai RSC given the limitations on the encryption security and the probability of the correct reception of messages by the legitimate receiver are obtained for the first time. These bounds (due to the use of Plotkin estimates and Bassalygo-Eliess estimates) allow to make a scientifically reasonable conclusion about the limited capabilities of the Mihalević-Imai RSC in terms of modern requirements concerning the security and the practicality in real-life environment. In particular, the use of the obtained bounds to the RSC with a key length of 512 bits shows that their security does not exceed 2^{79} operations regardless of the choice of the components.

The technique for designing RSC was further developed. In contrast to before known approaches, the proposed method is based on the employment of the nonlinear transformations or keyless hash functions for random coding and allows to increase the security (in 2^{242} and more times) compared with the Mihalević-Imai RSC by enlarging the class of transformations used in the construction of a randomizer.

The practical significance of the obtained results consists in developing the software implementations of RSC with nonlinear random coding based on the nonlinear mappings or keyless hash functions that are more secure (in 2^{242} and more times) and more fast (in 125 and more times) than the before known RSC with the same length of the input message. The developed implementations of the randomized stream ciphers allow to encrypt/decrypt messages in real time and can be used in

practice for special (military) applications where the leakage of confidential information creates risks for information security of the state.

The scientific and practical results of the thesis were implemented at the Foreign Intelligence Service of Ukraine (in the research scientific work «Keta») and in the scientific and technical developments of CJSC «Institute of Information Technologies».

Keywords: security of state information resources, stream cipher, random coding, methods of designing randomized stream ciphers, computational security, security proving.

Список основних публікацій здобувача:

1. А.Н. Алексейчук, С.В. Гришаков, «Нелинейное случайное кодирование в системах передачи информации по каналу связи с отводом», *Правове, нормативне та метрологічне забезпечення системи захисту інформації в Україні*, В. 8, С. 133-140, 2004.

2. А.Н. Алексейчук, С.В. Гришаков, «Алгоритмы нелинейного случайного кодирования и декодирования сообщений Z_4-линейными кодами в модели wire-tap channel», *Правове, нормативне та метрологічне забезпечення системи захисту інформації в Україні*, В. 2(13), С. 169-176, 2006.

3. А.Н. Алексейчук, С.В. Гришаков, «Неасимптотические оценки эффективности случайного кодирования в системе передачи информации по двоичному симметричному каналу связи с отводом», *Системні дослідження та інформаційні технології*, № 4, С. 37-47, 2011.

4. A.M. Alekseychuk, S.V. Gryshakov, «On the computational security of randomized stream ciphers proposed by Mihalević and Imai», *Захист інформації*, Т. 16, № 4, С. 328-334, 2014.

5. А.М. Олексійчук, С.В. Гришаков, «Метод побудови рандомізованих потокових шифросистем на основі нелінійного випадкового кодування»,

Спеціальні телекомунікаційні системи та захист інформації, В. 2(26), С. 5-14, 2014.

6. A. Alekseychuk, S. Gryshakov, «Randomized stream ciphers with enhanced security based on nonlinear random coding», *Journal of Mathematics and System Science*, V.5, pp. 516-522, 2015.

7. А.Н. Алексейчук, С.В. Гришаков, «Границы для скорости передачи информации в рандомизированных поточных шифрсистемах Михалевича-Имаи», *Радиотехника*, В. 181, С. 31-39, 2015.

8. А.Н. Алексейчук, С.В. Гришаков, «Стойкие и практичные рандомизированные поточные шифры на основе кодов Рида-Соломона», *Кибернетика и системный анализ*, Т. 53, № 2, С. 114-121, 2017.

9. А. Алексейчук, С. Гришаков, «Нелинейное случайное кодирование в системах передачи информации по каналу связи с отводом», *Безпека інформації у інформаційно-телекомунікаційних системах*: *тези доп. VII міжнар. наук.-практ. конф.*, 12-14 травня 2004 р., К., 2004, С. 25.

10. А. Алексейчук, С. Гришаков, «Алгоритмы случайного кодирования Z_4-линейными кодами в системе передачи информации по каналу связи с отводом», *Безпека інформації у інформаційно-телекомунікаційних системах*: *тези доп. IX міжнар. наук.-практ. конф.*, 17-19 травня 2006 р., К., 2006, С. 19.

11. А. Алексейчук, С. Гришаков, «Оценки стойкости защиты многократно переданных сообщений в модели WTC», *Безпека інформації у інформаційно-телекомунікаційних системах*: *тези доп. XI міжнар. наук.-практ. конф.*, 20-22 травня 2008 р., К., 2008, С. 31.

12. А. Алексейчук, С. Гришаков, «Оценки характеристик эффективности системы передачи информации по каналу связи с отводом в случае неидеального основного канала», *Безпека інформації у інформаційно-телекомунікаційних системах*: *тези доп. XII міжнар. наук.-практ. конф.*, 19-21 травня 2009 р., К., 2009, С. 26.

13. A.N. Alekseychuk, S.V. Gryshakov, «Randomized stream ciphers with enhanced security based on nonlinear random coding», *Probability, reliability and stochastic optimization* (*PRESTO 2015*), *Proceedings*, April 7-10, Kyiv, Ukraine, 2015, pp. 27.

14. А. Олексійчук, С. Гришаков, «Рандомізовані шифросистеми Міхалевича-Імаї на основі кодів Ріда-Соломона», *Безпека інформації у інформаційно-телекомунікаційних системах*: тези *доп. XVII міжнар. наук.-практ. конф*., 26-28 травня 2015 р., К., 2015, С. 27.

15. А.Н. Алексейчук, С.В. Гришаков, «Стойкие и практичные рандомизированные поточные шифры на основе кодов Рида-Соломона», *XII Белорусская математическая конференция*, 5-10 сентября 2016 г., Минск, Беларусь, С.28.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 13
ВСТУП 14
РОЗДІЛ 1. АНАЛІЗ ЕФЕКТИВНОСТІ ПОБУДОВИ ТА РЕАЛІЗАЦІЇ РАНДОМІЗОВАНИХ ШИФРОСИСТЕМ, ЩО ВИКОРИСТОВУЮТЬСЯ В СПЕЦІАЛЬНИХ ІНФОРМАЦІЙНО-ТЕЛЕКОМУНІКАЦІЙНИХ СИСТЕМАХ 22
1.1. Роль та практичне значення рандомізованих потокових шифросистем у забезпеченні безпеки державних інформаційних ресурсів 22
1.2. Класифікація, математичні моделі, показники стійкості та ефективності рандомізованих симетричних шифросистем 26
1.3. Аналіз відомих методів побудови рандомізованих симетричних шифросистем 29
1.3.1. Огляд відомих методів побудови рандомізованих блокових шифросистем 30
1.3.2. Огляд відомих методів побудови рандомізованих потокових шифросистем 38
1.3.3. Системи передачі інформації каналом зв’язку з відомим 41
1.3.4. Рандомізовані потокові шифросистеми Міхалевича-Імаї 44
1.4. Основні напрями та окремі задачі дисертаційного дослідження 48
Висновки 49
Список використаних джерел у першому розділі 52
РОЗДІЛ 2. РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ ОБЧИСЛЮВАЛЬНОЇ СТІЙКОСТІ ТА ПРАКТИЧНОСТІ РПШ МІХАЛЕВИЧА-ІМАЇ 63
2.1. Формальне означення та основні показники ефективності РПШ Міхалевича-Імаї 64
2.2. Обчислювальна стійкість РПШ Міхалевича-Імаї 67
2.2.1. Обчислювальна стійкість РПШ Міхалевича-Імаї відносно атак на основі відомих шифрованих повідомлень 68

2.2.2. Обчислювальна стійкість РПШ Міхалевича-Імаї відносно атаки на основі підібраних відкритих повідомлень 71
2.2.3. Обчислювальна стійкість РПШ Міхалевича-Імаї відносно атаки на основі підібраних векторів ініціалізації 72
2.3. Аналітичні межі для швидкості передачі інформації в РПШ Міхалевича-Імаї при заданих обмеженнях відносно стійкості та ймовірності правильного прийому повідомлень законним користувачем 79
Висновки 85
Список використаних джерел у другому розділі 87
РОЗДІЛ 3. МЕТОД ПОБУДОВИ РАНДОМІЗОВАНИХ ПОТОКОВИХ ШИФРОСИСТЕМ З НЕЛІНІЙНИМ ВИПАДКОВИМ КОДУВАННЯМ 89
3.1. Формальне означення рандомізованих потокових шифросистем з нелінійним випадковим кодуванням 90
3.2. Обчислювальна стійкість РПШ з нелінійним випадковим кодуванням 93
3.2.1. Обчислювальна стійкість РПШ з нелінійним випадковим кодуванням відносно атак на основі відомих шифрованих повідомлень 93
3.2.2. Обчислювальна стійкість РПШ з нелінійним випадковим кодуванням відносно атаки на основі підібраних векторів ініціалізації в загальному випадку 99
3.2.3. Обчислювальна стійкість РПШ з нелінійним випадковим кодуванням відносно атаки на основі підібраних векторів ініціалізації в окремому випадку 104
Висновки 111
Список використаних джерел у третьому розділі 113
РОЗДІЛ 4. РЕЗУЛЬТАТИ ПОРІВНЯННЯ СТІЙКОСТІ ТА ЕФЕКТИВНОСТІ ПРОГРАМНИХ РЕАЛІЗАЦІЙ РАНДОМІЗОВАНИХ ПОТОКОВИХ ШИФРОСИСТЕМ 116

4.1. Порівняння запропонованих РПШ з шифросистемами Міхалевича-Імаї за стійкістю та швидкістю передачі 117
4.1.1. Порівняння РПШ за швидкістю передачі при заданих обмеженнях щодо стійкості та довжини шифрованих повідомлень 117
4.1.2. Порівняння РПШ за стійкістю при заданих обмеженнях щодо швидкості передачі та довжини шифрованих повідомлень 120
4.2. Обґрунтування вибору компонент для побудови РПШ з нелінійним випадковим кодуванням 121
4.2.1. Вибір відображення ϕ 122
4.2.2. Вибір генератора гами 129
4.2.3. Вибір генератора випадкових послідовностей 130
4.3. Результати дослідження ефективності програмних реалізацій РПШ з нелінійним випадковим кодуванням 132
Висновки 135
Список використаних джерел у четвертому розділі 137
ВИСНОВКИ 141
ДОДАТКИ 149
Додаток А 149
Додаток Б 169
Додаток В 192
Додаток Д 210

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БШ – блоковий шифр;

ДСК – двійковий симетричний канал;

ЛРЗ – лінійний регістр зсуву;

ОС – операційна система;

ПК – персональний комп’ютер;

РБШ – рандомізована блокова шифросистема;

РІ – розподіл імовірностей;

РПШ – рандомізована потокова шифросистема;

СІТС – спеціальні інформаційно-телекомунікаційні системи;

AES – симетричний алгоритм блокового шифрування (стандарт США);

NIST – Національний інститут стандартизації і технологій США.

ВСТУП

Актуальність теми. З огляду на зростання рівня зовнішніх загроз національній безпеці та суверенітету України, особливої гостроти набувають задачі забезпечення інформаційної безпеки держави. Одним із основних напрямків вирішення цих задач є створення нових та удосконалення існуючих методів криптографічного захисту інформації, спрямованих на забезпечення конфіденційності, цілісності, справжності та доступності інформації.

На сьогодні криптографічні системи є невід'ємним елементом спеціальних інформаційно-телекомунікаційних систем (СІТС). Реалізації криптосистем повинні бути швидкими (здатними функціонувати в режимі реального часу на різних програмних і апаратних платформах) та криптографічно стійкими до всіх відомих криптоаналітичних атак. Особливо це стосується спеціальних (військових) додатків, витоки конфіденційної інформації в яких створюють ризики для інформаційної безпеки держави. Такими додатками є ті, в яких: часто трапляються випадки компрометації шифрувальної апаратури або алгоритму шифрування, відмови системи блокування шифратора, внаслідок чого в канал передачі може потрапити «слабка» гама шифрування; передаються короткі повідомлення (спеціальні команди чи військові накази); є невеликим навантаження на інформаційний трафік; криптографічна стійкість є важливішою за швидкість передачі інформації; є невеликою кількість абонентів; алгоритм шифрування може бути невідомим.

Широку розповсюдженість для захисту інформації в СІТС отримали потокові шифри, які представляють собою шифри з секретним ключем, де кожний символ відкритого тексту перетворюється в символ шифрованого в залежності не тільки від ключа, що використовується, але і від розташування символу у відкритому тексті. Звичайно такі шифри складаються з лінійних регістрів зсуву та компонент, що реалізують нелінійні перетворення. Основною

перевагою потокових шифрів над блоковими є значно вища швидкість шифрування, зокрема, програмні реалізації слово-орієнтованих потокових шифрів є в 5 – 10 разів швидшими у порівнянні з відповідними реалізаціями блокових шифрів. Крім того, потокові шифри є більш придатними для застосування у пристроях з обмеженими обчислювальними ресурсами або з малим споживанням електроенергії.

Відомо, що стійкість будь-якого сучасного шифру залежить від розвитку методів криптоаналізу та можливостей криптоаналітика і має тенденцію зменшуватися з часом. Особливо це стосується потокових шифрів, різноманіття конструкцій яких (в порівнянні з блоковими) надає криптоаналітику більше потенційних можливостей для створення нових методів криптоаналізу та побудови нових атак, зокрема, таких, що можуть бути реалізовані на практиці. Характерним прикладом є алгоритми шифрування A5/1 та A5/2, які були зламані одразу ж після оприлюднення їх описів.

Таким чином, неухильний розвиток інформаційних технологій поряд з нарощуванням потужності обчислювальних засобів та появою нових (або підсиленням відомих) методів криптоаналізу створює потенційну загрозу для СІТС, де використовуються потокові шифри. Оскільки швидка заміна алгоритму шифрування, криптографічні слабкості якого виявлені на етапі його експлуатації (як правило, через багато років після його створення) є практично неможливою (або організаційно та коштовно затратною справою), видається доцільним створення методів підвищення стійкості потокових шифрів без внесення змін в алгоритми шифрування шляхом застосування додаткових перетворень, які не потребують ключів, можуть бути відносно просто реалізовані та забезпечують науково обґрунтований рівень стійкості систем шифрування в цілому.

Одним з таких загальних методів є так звана рандомізація або випадкове кодування джерела відкритих повідомлень. Зауважимо, що метод рандомізації є відомим достатньо давно, проте (саме для випадку потокових шифрів) його

можливості досліджені не повністю. Імовірно, єдиним відомим прикладом рандомізованих потокових шифросистем (РПШ), які будуються на регулярній основі та, в принципі, можуть бути використані на практиці, є шифросистеми Міхалевича-Імаї. Рандомізатори зазначених РПШ будуються на основі двійкових лінійних перетворень, зокрема, завадостійкого кодування відкритих повідомлень лінійними кодами. Проте стійкість РПШ Міхалевича-Імаї суттєво залежить від будови їх компонент і може бути значно менше, ніж стверджують їх розробники. Деякі з цих шифросистем виявляються вразливими навіть до атак на основі відомих шифрованих повідомлень і, отже, не можуть бути використані для захисту державних інформаційних ресурсів. Наведені факти свідчать про актуальність *наукової задачі розробки методу побудови рандомізованих потокових шифросистем з нелінійним випадковим кодуванням для забезпечення безпеки державних інформаційних ресурсів*, розв’язанню якої присвячено дану дисертаційну роботу.

Зв'язок роботи з науковими програмами, планами, темами. Робота над дисертацією проводилася в рамках науково-дослідної роботи “Кета” (номер держреєстрації 0114U004643) на замовлення Служби зовнішньої розвідки України та відповідно до планів науково-дослідної роботи Інституту спеціального зв’язку та захисту інформації Національного технічного університету України “Київський політехнічний інститут імені Ігоря Сікорського”.

Мета та задачі досліджень. *Метою дисертаційної роботи є підвищення криптографічної стійкості потокових шифрів шляхом рандомізації джерела відкритих повідомлень для забезпечення безпеки державних інформаційних ресурсів.*

Для досягнення поставленої мети **необхідно розвязати такі основні задачі:**

1. Провести аналіз відомих методів побудови рандомізованих шифросистем для забезпечення безпеки державних інформаційних ресурсів.

2. Отримати аналітичні оцінки обчислювальної стійкості РПШ Міхалевича-Імаї відносно атак на основі відомих шифрованих повідомлень, а також підібраних векторів ініціалізації.

3. Довести, що клас РПШ Міхалевича-Імаї (незалежно від будови їх компонент) володіє суттєвою слабкістю, яка полягає в зменшенні кількості інформації (в порівнянні з довжиною блоку шифрувальної гами), що необхідна для відновлення за реальний час символів відкритого тексту.

4. Отримати аналітичні межі для швидкості передачі інформації в РПШ Міхалевича-Імаї при заданих обмеженнях відносно стійкості та ймовірності правильного прийому повідомлень законним користувачем.

5. Розробити метод побудови РПШ з нелінійним випадковим кодуванням та отримати аналітичні оцінки обчислювальної стійкості цих шифросистем відносно відомих атак; встановити та обґрунтувати вимоги до нелінійних відображень в конструкціях рандомізаторів РПШ з нелінійним випадковим кодуванням, що визначають стійкість цих РПШ відносно зазначених атак.

6. Провести порівняння РПШ Міхалевича-Імаї і РПШ з нелінійним випадковим кодуванням за швидкістю передачі (при фіксованій стійкості) та стійкістю шифрування (при фіксованій швидкості передачі); розробити програмні реалізації запропонованих РПШ на базі нелінійних відображень і геш-функцій та виконати порівняння ефективності зазначених програмних реалізацій.

Об'єктом дослідження в дисертаційній роботі є процес криптографічного перетворення інформації у рандомізованих потокових шифросистемах, а *предметом дослідження* – методи побудови рандомізованих потокових шифросистем з нелінійним випадковим кодуванням, призначених для забезпечення безпеки державних інформаційних ресурсів.

Методи дослідження. Основу дисертаційних досліджень складають теоретичні дослідження (математичні методи оцінювання обчислювальної стійкості рандомізованих потокових шифросистем). Для аналізу існуючих

методів побудови рандомізованих шифросистем, а також розробки методу побудови РПШ з нелінійним випадковим кодуванням застосовувались методи лінійної алгебри, теорії ймовірностей та теорії інформації. Дослідження стійкості РПШ Міхалевича-Імаї та РПШ з нелінійним випадковим кодуванням здійснювалось з використанням методів лінійної алгебри, теорії ймовірностей, математичної статистики, а також теорії складності обчислень. При побудові аналітичних меж для швидкості передачі інформації в РПШ Міхалевича-Імаї застосовувались методи теорії кодування. Чисельні розрахунки на обчислювальній системі та розробка програмних реалізацій РПШ з нелінійним випадковим кодуванням виконувалися з використанням середовища розробки Microsoft Visual Studio 2013 (компонент Visual C++).

Наукова новизна отриманих результатів. Підсумком розв’язання зазначених наукових задач є такі нові наукові результати, що висуваються на захист:

1. *Вперше* отримано аналітичні оцінки параметрів, що визначають стійкість РПШ Міхалевича-Імаї відносно атак на основі відомих шифрованих повідомлень, а також підібраних векторів ініціалізації. Отримані оцінки *дозволяють* з’ясувати теоретико-кодовий сенс параметрів, які визначають обчислювальну стійкість цих шифросистем, а також встановити, що їх стійкість може бути значно менше, ніж стверджують їх розробники, що досягається *за рахунок* розширення можливостей супротивника при проведенні зазначених атак.

2. *Вперше* доведено, що клас РПШ Міхалевича-Імаї (незалежно від будови їх компонент) володіє суттєвою слабкістю, яка полягає в зменшенні кількості інформації (в порівнянні з довжиною блоку шифрувальної гами), що необхідна для відновлення за реальний час символів відкритого тексту. Зазначена властивість *дозволяє* зробити практично важливий висновок про те, що для відновлення символів відкритого тексту супротивнику достатньо мати лише часткову інформацію про секретний ключ РПШ, що відбувається *за рахунок*

спільного застосування випадкового і завадостійкого кодування повідомлень лінійними кодами.

3. *Вперше* отримано аналітичні межі для швидкості передачі інформації в РПШ Міхалевича-Імаї при заданих обмеженнях щодо ймовірності правильного прийому повідомлень законним користувачем та стійкості шифрування. Зазначені межі, *за рахунок* застосування оцінок Плоткіна та Бассалиго-Елайєса для швидкості передачі лінійних кодів, *дозволяють* зробити науково обґрунтований висновок про обмежені можливості РПШ Міхалевича-Імаї з погляду сучасних вимог щодо стійкості та практичності в реальних умовах.

4. *Отримав подальший розвиток* метод побудови РПШ, який, *на відміну від раніше відомих*, базується на застосуванні для випадкового кодування нелінійних відображень або безключевих геш-функцій та *дозволяє* збільшити стійкість в порівнянні з РПШ Міхалевича-Імаї *за рахунок* розширення класу перетворень, які використовуються в конструкції рандомізатора.

Практичне значення отриманих результатів. Розроблено програмні реалізації РПШ з нелінійним випадковим кодуванням на основі нелінійних відображень чи безключевих геш-функцій, що є більш стійкими (у 2^{242} і більше разів) і більш швидкісними (у 125 і більше разів) в порівнянні з раніше відомими РПШ при однаковій довжині вихідного повідомлення. Розроблені реалізації РПШ дозволяють здійснювати процедури зашифровування/розшифровування даних в режимі реального часу та можуть бути використані на практиці у спеціальних (військових) додатках, витоки конфіденційної інформації в яких створюють ризики для інформаційної безпеки держави.

Наукові та практичні *результати дисертаційної роботи реалізовані* в Службі зовнішньої розвідки України – в результаті виконання НДР “Кета” (акт від 14.09.2016) та в науково-технічних розробках ЗАО “Інститут інформаційних технологій” (акт від 25.07.2016).

Особистий внесок здобувача. У [1, 9] автором отримано неасимптотичні оцінки ймовірності правильного відновлення символу відкритого тексту за символом шифротексту РПШ з нелінійним випадковим кодуванням; в [2, 10] автору належать ефективні алгоритми нелінійного випадкового кодування і декодування повідомлень; в [3, 11, 12] автором отримано аналітичні оцінки ймовірності правильного прийому відкритих повідомлень в системах передачі інформації з випадковим кодуванням; в [4] автором запропоновано атаку на РПШ Міхалевича-Імаї на основі відомих шифрованих повідомлень, а також атаку на основі підібраних векторів ініціалізації, для якої отримано верхні оцінки обчислювальної складності; в [5] автором отримано верхні оцінки обчислювальної складності атаки на РПШ з нелінійним випадковим кодуванням на основі підібраних векторів ініціалізації; в [6, 13] автором запропоновано метод побудови рандомізованих потокових шифросистем з нелінійним випадковим кодуванням та отримано верхні оцінки обчислювальної складності атак на такі шифросистеми; в [7] автором отримано верхні оцінки швидкості передачі інформації в РПШ Міхалевича-Імаї при заданих обмеженнях відносно ймовірності правильного прийому повідомлень законним одержувачем і стійкості шифрування; крім того, встановлено нижню межу для максимальної швидкості передачі інформації, при якій існують РПШ Міхалевича-Імаї із заданою стійкістю; в [8, 14, 15] автору належить неасимптотична нижня межа складності атаки на РПШ Міхалевича-Імаї на основі підібраних векторів ініціалізації.

Апробація результатів дисертації. Результати дисертаційних досліджень доповідалися та обговорювалися *на 7 міжнародних наукових конференціях*: VII – XVII Міжнародних науково-практичних конференціях “Безпека інформації в інформаційно-телекомунікаційних системах” (м. Київ, 2004, 2006, 2008, 2009, 2015 рр.), Міжнародній науковій конференції “Probability, reliability and stochastic optimization” (м. Київ, 2015 р.), Міжнародній науковій конференції “XII Белорусская математическая конференция” (Білорусь, м. Мінськ, 2016 р.).

Публікації. Основні наукові результати дисертаційної роботи опубліковано в 15 наукових працях: з них 8 наукових статей [1 – 8] в наукових спеціалізованих виданнях України та інших країн (4 видання індексуються міжнародними наукометричними базами), 7 тез доповідей на наукових та науково-практичних конференціях [9 – 15].

Структура роботи та її обсяг. Дисертація складається з анотації, змісту, переліку умовних позначень, вступу, чотирьох розділів, загальних висновків, додатків, списку використаних джерел (в кінці кожного розділу основної частини дисертації) і має 127 сторінок основного тексту, 30 рисунків, 3 таблиці, 66 сторінок додатків. Список використаних джерел містить 173 найменування і займає 18 сторінок. Загальний обсяг дисертаційної роботи – 214 сторінок.

РОЗДІЛ 1

АНАЛІЗ ЕФЕКТИВНОСТІ ПОБУДОВИ ТА РЕАЛІЗАЦІЇ РАНДОМІЗОВАНИХ ШИФРОСИСТЕМ, ЩО ВИКОРИСТОВУЮТЬСЯ В СПЕЦІАЛЬНИХ ІНФОРМАЦІЙНО-ТЕЛЕКОМУНІКАЦІЙНИХ СИСТЕМАХ

1.1. Роль та практичне значення рандомізованих потокових шифросистем у забезпеченні безпеки державних інформаційних ресурсів

Сучасні виклики та загрози в інформаційній сфері носять все більш глобальний характер і мають на меті заподіяти максимально деструктивних наслідків в різних галузях економіки. З погляду на це, забезпечення інформаційної безпеки є однією з пріоритетних задач, що стоїть перед нашою державою [1, 2]. Для вирішення цієї задачі необхідно створювати нові та удосконалювати існуючі методи криптографічного захисту інформації, спрямовані на забезпечення конфіденційності, цілісності, справжності та доступності інформації. На сьогодні криптографічні системи є невід'ємним елементом спеціальних інформаційно-телекомунікаційних систем. Реалізації криптосистем повинні бути швидкими (здатними функціонувати в режимі реального часу на різних програмних і апаратних платформах) та криптографічно стійкими до всіх відомих криптоаналітичних атак [3 – 9]. Особливо це стосується спеціальних (військових) додатків, витоки конфіденційної інформації в яких створюють ризики для інформаційної безпеки держави. Такими додатками є ті, в яких: часто трапляються випадки компрометації шифрувальної апаратури або алгоритму шифрування, відмови системи блокування шифратора, внаслідок чого в канал передачі може потрапити «слабка» гама шифрування; передаються короткі повідомлення (спеціальні команди чи військові накази); є невеликим навантаження на

інформаційний трафік; криптографічна стійкість є важливішою за швидкість передачі інформації; є невеликою кількість абонентів; алгоритм шифрування може бути невідомим.

Широку розповсюдженість для захисту інформації в СІТС отримали потокові шифри, які представляють собою шифри з секретним ключем, де кожний символ відкритого тексту перетворюється в символ шифрованого в залежності не тільки від ключа, що використовується, але і від розташування символу у відкритому тексті [10]. Звичайно такі шифри складаються з лінійних регістрів зсуву та компонент, що реалізують нелінійні перетворення. Основною перевагою потокових шифрів над блоковими є значно вища швидкість шифрування (зокрема, програмні реалізації слово-орієнтованих потокових шифрів є в 5 – 10 разів швидшими у порівнянні з відповідними реалізаціями блокових шифрів) [11 – 13]. На сьогодні лише потокові шифри дозволяють забезпечити прийнятну швидкість шифрування на канальному, мережевому та транспортному рівнях [14 – 16]. Крім того, потокові шифри є більш придатними для застосування у пристроях з обмеженими обчислювальними ресурсами або з малим споживанням електроенергії [17 – 31]. З цих причин вони широко використовуються у кабельних мережах з малою пропускною здатністю для зашифровування аудіо/відео контенту (додаток VoIP [32]), системах цифрової відеотрансляції (шифр Chameleon [33]), безпровідних комунікаціях (шифр E0 для радіостандарту малого радіусу дії Bluetooth [34], протокол WPA [35] для організації захищеного доступу за технологією Wi-Fi), мобільній телефонії (шифр A5/1 для стандарту GSM [36]), для створення захищених Інтернет з’єднань (протокол SSL [37]), а також при побудові захищеної VPN мережі (протокол MPPE [38]); див. рис. 1.1.

Відомо, що стійкість будь-якого сучасного шифру залежить від розвитку методів криптоаналізу та можливостей криптоаналітика і має тенденцію зменшуватися з часом. Особливо це стосується потокових шифрів, різноманіття конструкцій яких (в порівнянні з блоковими) надає криптоаналітику більше

потенційних можливостей для створення нових методів криптоаналізу та побудови нових атак, зокрема, таких, що можуть бути реалізовані на практиці. Характерним прикладом є алгоритми шифрування A5/1 та A5/2, які були зламані одразу ж після оприлюднення їх описів (див., наприклад, [5]).

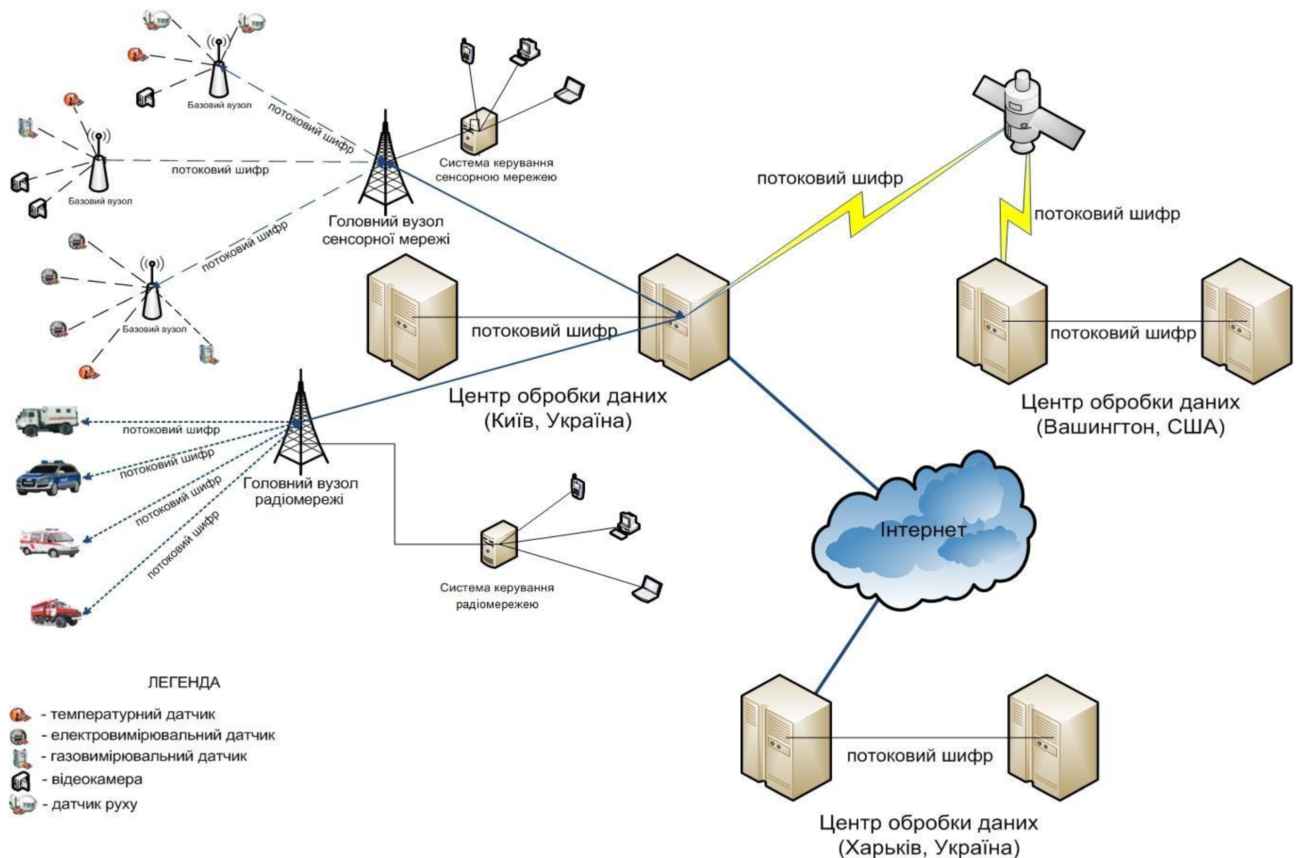


Рис. 1.1. Типова схема спеціальної інформаційно-телекомунікаційної системи [39]

Таким чином, неухильний розвиток інформаційних технологій поряд з постійним нарощуванням потужності обчислювальних засобів та появою нових (або підсиленням відомих) методів криптоаналізу створює потенційну загрозу для СІТС, де використовуються потокові шифри. Оскільки швидка заміна алгоритму шифрування, криптографічні слабкості якого виявлені на етапі його експлуатації (як правило, через багато років після його створення) є практично неможливою (або організаційно та коштовно затратною справою), видається

доцільним створення методів підвищення стійкості потокових шифрів без внесення змін в алгоритми шифрування шляхом застосування додаткових перетворень, які не потребують ключів, можуть бути відносно просто реалізовані та забезпечують науково обґрунтований рівень стійкості систем шифрування в цілому.

Одним з таких загальних методів є так звана рандомізація або випадкове кодування джерела відкритих повідомлень [40 – 49]. Зауважимо, що метод рандомізації є відомим достатньо давно [40, 41, 46 – 54] і в теперішній час сформувався у вигляді відносно самостійного напрямку в області криптографічного захисту інформації, що отримав назву *ймовірносно-криптографічного підходу* або *ймовірнісної криптографії* [47, 50, 55 – 58]. Тим не менше, можливості методу рандомізації (саме для випадку потокових шифрів) досліджені не повністю. Імовірно, єдиним відомим прикладом РПШ, які будуються на регулярній основі та, в принципі, можуть бути використані на практиці, є шифросистеми Міхалевича-Імаї [59 – 64]. Рандомізатори зазначених РПШ будуються на основі двійкових лінійних перетворень, зокрема, завадостійкого кодування відкритих повідомлень лінійними кодами. Проте стійкість РПШ Міхалевича-Імаї суттєво залежить від будови їх компонент і може бути значно менше, ніж стверджують їх розробники. Деякі з цих шифросистем виявляються вразливими навіть до атак на основі відомих шифрованих повідомлень і, отже, не можуть бути використані для захисту державних інформаційних ресурсів.

На сьогодні клас криптографічних систем та алгоритмів, що базуються на рандомізації в тій чи іншій мірі, є дуже широким і охоплює практично всі напрямки криптографії та її додатків. До основних недоліків таких шифросистем можна віднести збільшення довжини повідомлень при зашифровуванні, а також необхідність використання високошвидкісних генераторів випадкових послідовностей для реалізації випадкового кодування. З іншої сторони, в системах попереднього шифрування інформації збільшення

довжини повідомлення не є суттєвим негативним фактором. Більш того, можливості сучасних процесорів і засобів зв’язку дозволяють практично повністю нівелювати наслідки багатократного збільшення довжини повідомлень, що передаються [5, 65]. Серед явних переваг цих шифросистем, як зазначено вище, можна виділити можливість обґрунтованого підвищення стійкості діючих шифросистем без внесення змін в алгоритми шифрування. Так, “введення нерівномірності” в закон руху ЛРЗ генератора гами потокового шифру, як правило, дозволяє значно підвищити практичну стійкість останнього відносно стандартних кореляційних і алгебраїчних атак [66, 67].

1.2. Класифікація, математичні моделі, показники стійкості та ефективності рандомізованих симетричних шифросистем

Рандомізовані шифросистеми можна розділити на теоретично (безумовно) стійкі та практично стійкі; основані на рандомізації джерела відкритих повідомлень і побудовані на основі рандомізованих криптографічних перетворень; шифросистеми з секретним або з загальнодоступним рандомізатором тощо (рис. 1.2). Можливі різні комбіновані варіанти побудови шифросистем, наприклад, системи шифрування, що використовують як загальнодоступний, так і секретний рандомізатори [40 – 42, 52].

Крім того, до рандомізованих можна віднести широкий клас шифросистем або їх елементів, які здійснюють перетворення, залежні від допоміжних секретних даних, що використовуються в якості частини ключа шифрування. Характерними прикладами таких шифросистем є шифри з керованими підстановочними операціями [4], деякі шифри багатозначної (колонної) заміни [68, 69], генератори гами з нерівномірним рухом (де в якості додаткового джерела випадковості використовується зовнішній пристрій – блок керування рухом ЛРЗ) [3, 66, 67, 70].

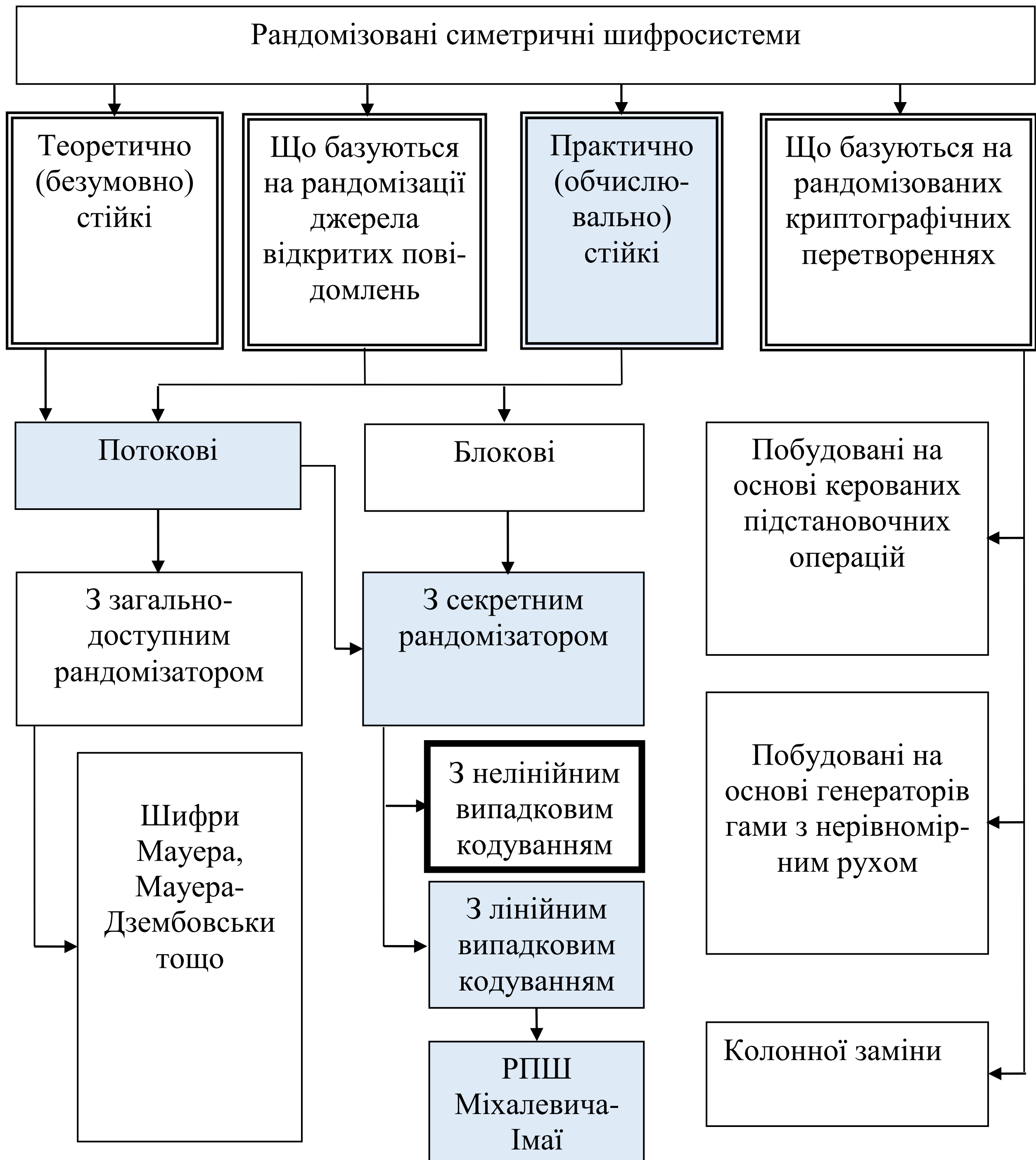


Рис. 1.2. Основні класи симетричних рандомізованих шифросистем

Для визначення математичної моделі рандомізованої шифросистеми введемо наступні позначення. Нехай S, X та Y – непусті скінчені множини, $P_S = (p(s) : s \in S)$ – розподіл імовірностей (РІ) на множині S, де $p(s) > 0$ для будь-якого $s \in S$, $A = (X, K, Y, f)$ – шифр з множинами відкритих повідомлень

X (де $|X| \geq |S|$), ключів K, шифрованих повідомлень Y і функцією шифрування $f: X \times K \to Y$ [45, 56, 57].

Рандомізована (симетрична) шифросистема $\Re$ визначається як впорядкований набір об'єктів $(S, P_S, A, \{C_S : s \in S\})$, де $\{C_S : s \in S\}$ – довільне розбиття множини X [40, 45, 51 – 53, 56]. Для передачі відкритого повідомлення $s \in S$ відправник випадково, з ймовірністю $|C_S|^{-1}$, вибирає повідомлення $x \in C_S$ і зашифровує його на ключі $k \in K$ шифру A, в результаті чого отримує шифрований текст $y = f_k(x) \overset{\text{def}}{=} f(x,k)$.

Одержувач розшифровує y, покладаючи $x = f_k^{-1}(y)$, і відновлює вихідне відкрите повідомлення як єдиний елемент $s \in S$ з властивістю $x \in C_S$. Замітимо, що однозначність розшифрування гарантована попарною диз'юнктивністю множин C_S, $s \in S$.

Нехай на множині ключів шифру A задано розподіл імовірностей (РІ) $(p(k): k \in K)$. Задамо РІ на множині $S \times X \times K \times Y$, покладаючи $p(s,x,k,y) = |C_s|^{-1}\, p(s)p(k)$, якщо $x \in C_S$, $y = f_k(x)$; $p(s,x,k,y) = 0$ – в іншому випадку. Зазначений розподіл імовірностей дозволяє стандартним чином визначити умовні ентропії $H(S|Y)$ и $H(K|Y)$, які називаються, відповідно, *ненадійністю повідомлення* і *ненадійністю ключа* рандомізованої шифросистеми $\Re$, які є показниками її теоретико-інформаційної стійкості [45, 51 – 53] (тут використовуються однакові позначення для скінчених множин і випадкових елементів зі значеннями в цих множинах, що не повинно призводити до непорозумінь). Основним параметром, що характеризує ефективність рандомізованої шифросистеми $\Re$, є *швидкість передачі інформації* $r(\Re) = \dfrac{H(S)}{\log|X|}$, де $H(S)$ – ентропія РІ P_S [45, 51 – 53]. На основі вищенаведеного $r(\Re) \leq 1$.

Відмітимо, що в більшості наукових публікацій, присвячених дослідженню стійкості рандомізованих шифросистем [40 – 42, 45, 51 – 53, 56, 71], використовується описана теоретико-інформаційна модель. При цьому рандомізація джерела повідомлень розглядається, в основному, як спосіб підвищення ненадійності відкритого повідомлення або ненадійності ключа.

Нижче пропонується аналіз відомих методів рандомізації як блокових, так і потокових шифросистем. Крім того, описується використання рандомізації для захисту дискретних повідомлень, що передаються каналами зв’язку з відводом (без застосування процедури шифрування).

1.3. Аналіз відомих методів побудови рандомізованих симетричних шифросистем

Відомі методи рандомізації [40 – 42, 45, 54, 71, 73] шифросистем мають за мету перетворення вихідних повідомлень (бернулієвського чи марківського) джерела у випадкові і рівноймовірні послідовності. Розробка і аналіз таких методів орієнтовані, насамперед, на підвищення їх ефективності за стандартним критерієм максимуму ентропії перетвореної послідовності при обмеженнях на її середню довжину, а також часову і ємнісну складності алгоритму перетворення [41, 42, 45, 71, 73 – 75]. Ці методи широко використовуються в криптографічній практиці [3, 45, 53], дозволяючи, у ряді випадків, будувати (потокові) шифросистеми, що мають безумовну стійкість відносно атак на основі відомого шифрованого тексту. Разом з тим, зазначені шифросистеми є практично нестійкими до атак на основі відомих відкритих текстів [3, 68, 76].

Методи рандомізації, спрямовані на підвищення теоретичної стійкості захисту інформації або ключа в симетричних шифросистемах, описані в роботах [4, 40, 51 – 53] і ряді інших. Зокрема, в [51 – 53] отримані оцінки теоретико-інформаційних показників стійкості простих підстановочних

шифрів, а в [51, 52] доведено низку теорем про існування безумовно стійких рандомізованих симетричних шифросистем, однак не запропоновано практично реалізовні способи їх побудови. Ряд таких способів описано в роботах [3, 4, 40, 53]. По суті всі вони зводяться до певних варіантів лінійного випадкового кодування і наступного зашифровування відкритих повідомлень (більш детальний їх опис наводиться нижче). Крім того, в доступних наукових публікаціях (за виключенням [61 – 63]) відсутні результати дослідження стійкості рандомізованих потокових шифросистем відносно атак на основі відомих відкритих текстів, що свідчить про необхідність продовження наукових досліджень в цьому напрямі.

1.3.1. Огляд відомих методів побудови рандомізованих блокових шифросистем (РБШ). При описі цих методів символами s, t і y будемо позначати, відповідно, відкрите повідомлення, випадкову рівноймовірну послідовність, що виробляється генератором випадкових послідовностей (рандомізатором), і шифроване повідомлення; символом F_k – шифрувальне перетворення вхідного блокового шифру, що відповідає ключу k, а символом $||$ – процедуру конкатенації (об’єднання) двох довільних векторів. Символ V_n позначає множину двійкових векторів довжини n.

Метод 1.1 (рис. 1.3). Дана конструкція РБШ є найпростішою і полягає у конкатенації повідомлення s з випадковою послідовністю t, після чого до результату застосовується шифрувальне перетворення F_k, що формально може бути записано у вигляді наступного правила: $y = F_k(s,t)$, $s \in V_m$, $t \in V_r$. При розшифровуванні вищезазначені дії виконуються у зворотному порядку.

Основні переваги описаного методу:

1) збільшення розміру простору повідомлень, що ускладнює проведення атаки повного перебору;

2) частковий захист відносно атаки на основі вибраних

відкритих/шифрованих повідомлень;

3) захист відносно атаки «переставлення бітів» (*bit twiddling*), при якій супротивник вибірково модифікує біти шифрованого тексту з метою зміни відповідних бітів відкритого тексту [40].

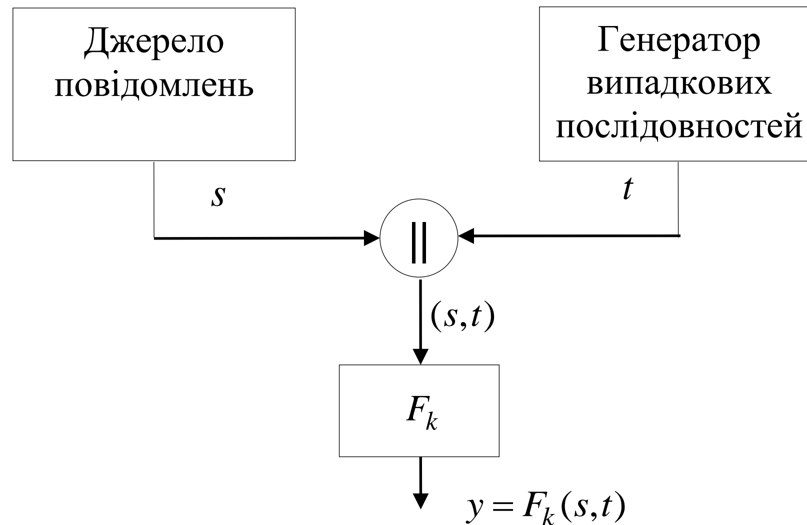


Рис. 1.3. Схема РБШ, побудована за методом 1.1

Метод 1.2 (рис. 1.4). Основна ідея цього методу полягає у застосуванні шифрувального перетворення F_k відповідного блокового шифру тільки до послідовностей випадкових бітів. Як видно з рис. 1.4, вироблена випадкова послідовність t додається за модулем 2 до відкритого повідомлення s, а також зашифровується з допомогою перетворення F_k. Результати зазначених операцій поєднуються у вихідне повідомлення $y = (s \oplus t, F_k(t))$, $s \in V_m$, $t \in V_m$.

Відзначимо, що у порівнянні з методом 1.1, в якому шифрувальне перетворення F_k застосовується для змішування випадкових даних з символами відкритого повідомлення, в даній конструкції безпосередньо рандомізується кожний символ повідомлення. Тим не менше, такий підхід також володіє певними недоліками.

По-перше, існує небезпека проведення активної атаки, при якій супротивник може вибірково змінити певні символи відкритого тексту шляхом

модифікації відповідних символів шифротексту. По-друге, проведення атаки на основі вибраного шифрованого тексту призводить до атаки на основі шифрованого тексту на шифрувальне перетворення F_k. По-третє, незважаючи на те, що входи перетворення F_k обираються випадково, умовний розподіл ймовірностей $\mathbf{P}(t \mid s \oplus t)$ є ідентичним до апріорного розподілу у просторі вхідних повідомлень. І нарешті, дана конструкція є вразливою до *spoofing*-атаки, при якій активний супротивник може вставляти підроблені повідомлення у комунікаційний канал [40].

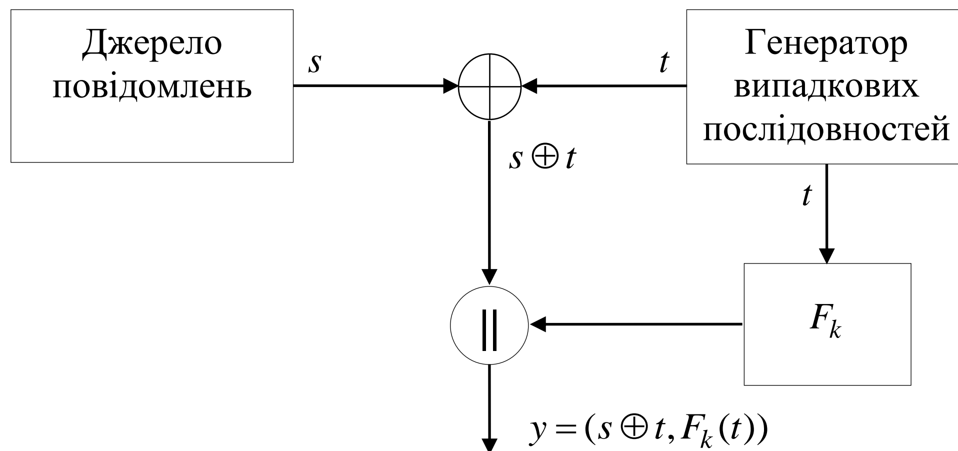


Рис. 1.4. Схема РБШ, побудована за методом 1.2

Метод 1.3 (рис. 1.5). Як видно з рисунку, при застосуванні даного методу перетворення F використовується два рази: спочатку – для зашифровування повідомлення s, де випадкова послідовність t виступає в ролі ключа шифрування, а потім для зашифровування самої послідовності t на сесійному ключі SK. Результати обох операцій поєднуються у вихідну послідовність $y = (F_t(s), F_{SK}(t))$, $s \in V_m$, $t \in V_r$ [40]

Як основну перевагу даної конструкції слід відмітити зменшення загрози атак на основі відомих шифрованих текстів, оскільки це вимагатиме багато шифрованих текстів, вироблених на одному ключі.

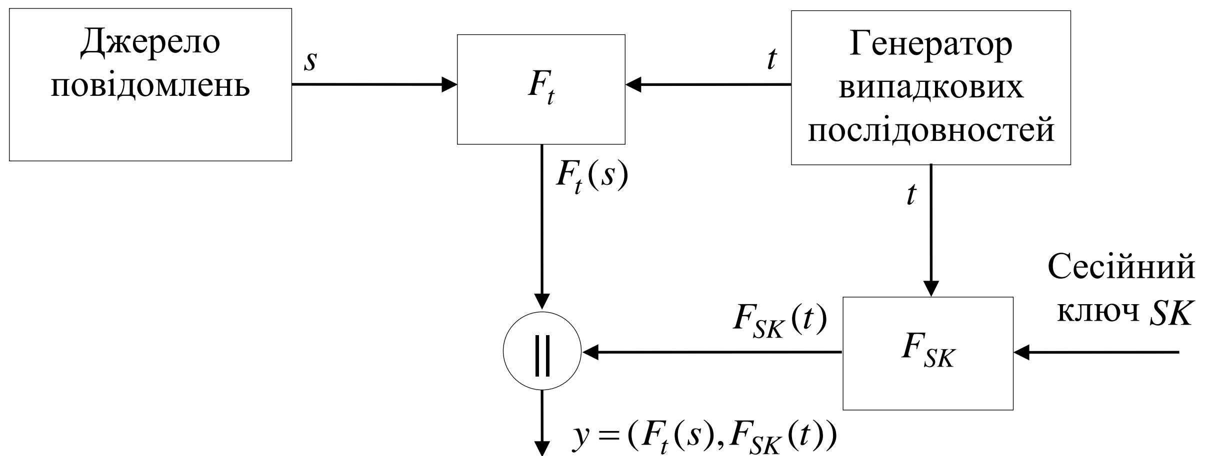


Рис. 1.5. Схема РБШ, побудована за методом 1.3

Метод 1.4 (рис. 1.6). В даній конструкції спочатку здійснюється додавання за модулем 2 відкритого повідомлення s з випадковою послідовністю t, після чого результат зашифровується з допомогою перетворення F_k.

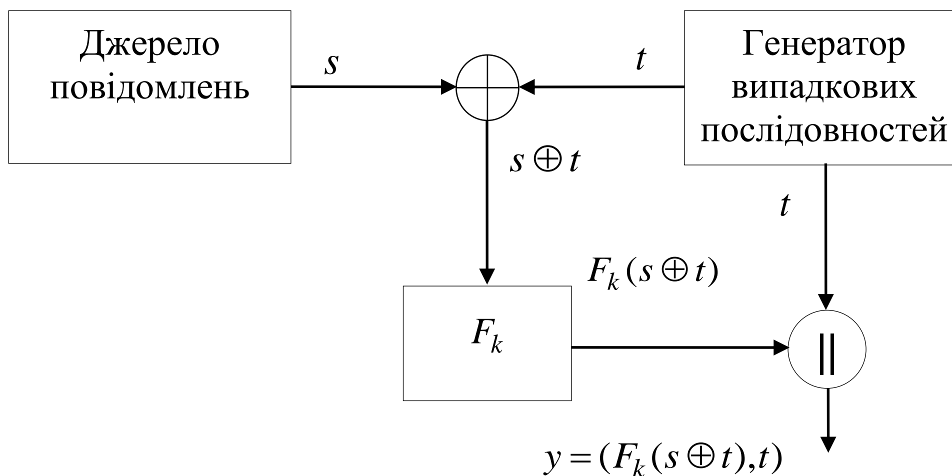


Рис. 1.6. Схема РБШ, побудована за методом 1.4

Формування вихідного вектору y відбувається за наступним правилом: $y=(F_k(s\oplus t),t)$, $s\in V_m$, $t\in V_m$. Дана конструкція РБШ володіє всіма криптографічними властивостями, притаманними конструкції, побудованої за методом 1.2 [40].

Метод 1.5 (рис. 1.7). Даний метод побудови РБШ поєднує в собі ідеї методів 1.1 та 1.2. Як видно з рисунку, спочатку повідомлення s додається за модулем 2 до випадкової послідовності t, після чого результат цієї операції конкатенується з послідовністю t та зашифровується перетворенням F_k у вихідну послідовність $y = (F_k((s \oplus t), t))$, $s \in V_m$, $t \in V_m$. Серед основних переваг описаного методу слід відзначити збільшення простору повідомлень для супротивника, а також рівномірність розподілу на множині аргументів шифрувального перетворення F_k.

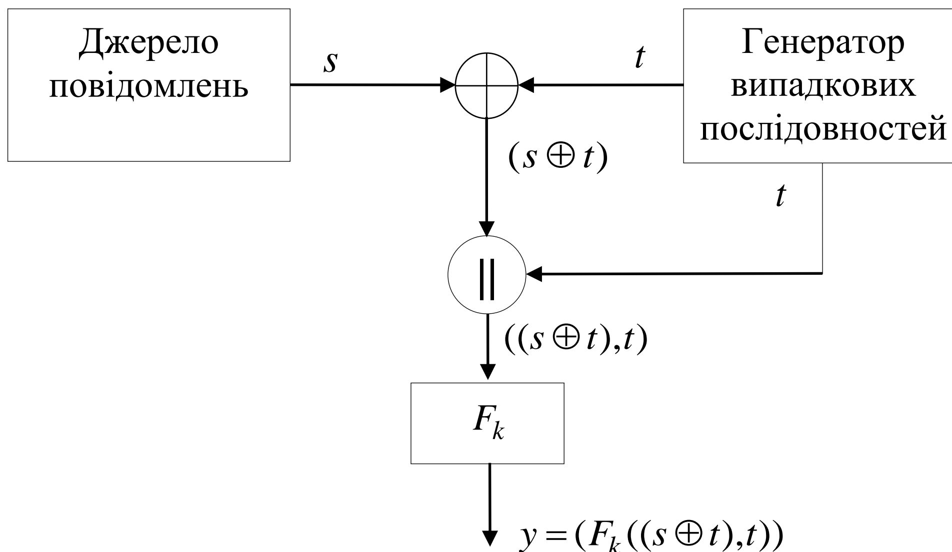


Рис. 1.7. Схема РБШ, побудована за методом 1.5

Крім того, ця конструкція не є вразливою до атаки "переставлення бітів" (*bit twiddling*) і атак на основі вибраного відкритого/шифрованого тексту. Інша потенційна слабкість такої конструкції полягає в тому, що при атаці на основі вибраного відкритого тексту супротивник може примусити шифрувальне перетворення F_k бути застосованим до послідовності вигляду $t \| t$ [40].

Метод 1.6 (рис. 1.8). Даний метод є подібним до методу 1.2 за виключенням того, що випадкова послідовність t передається у відкритому вигляді і додається за модулем 2 до кожного повідомлення s у зашифрованому

вигляді. Формально ця конструкція може бути описана за допомогою такого правила: $y=((F_k(t)\oplus s),t)$, $s\in V_m$, $t\in V_r$.

Наступні два методи побудови рандомізованих блокових шифросистем [40] базуються на використанні лінійних блокових кодів. При описі цих методів символом G позначається код, здатний виправляти усі комбінації, які складаються з не більше ніж u помилок. При цьому вважається, що генератор випадкових послідовностей виробляє двійкові послідовності t, вага яких не перевищує числа u.

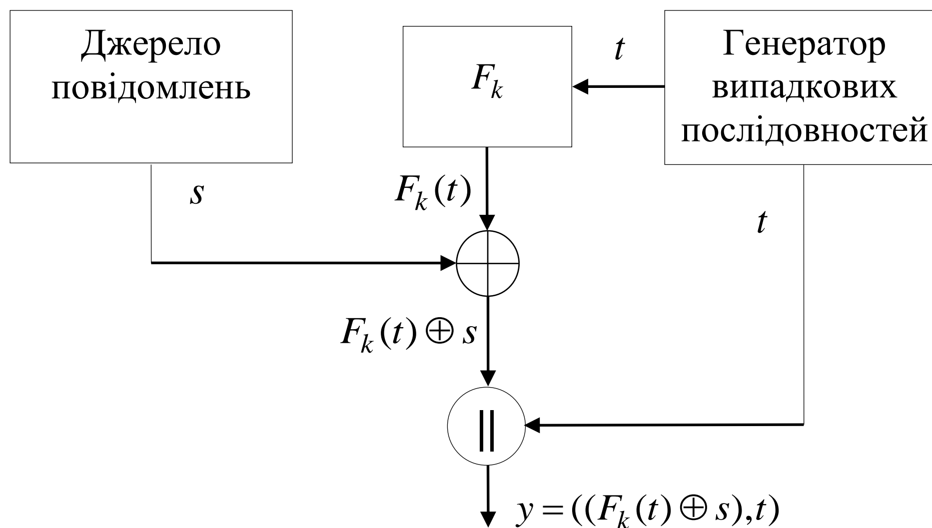


Рис. 1.8. Схема РБШ, побудована за методом 1.6

Метод 1.7 (рис. 1.9). В даній конструкції вихідне повідомлення s спочатку кодується за допомогою секретного лінійного блокового коду G. Після цього виконується додавання за модулем 2 закодованого повідомлення $G(s)$ з випадковою послідовністю t.

Оскільки кількість змінених символів не перевищує корегувальної здатності u вибраного коду, то одержувач зможе відновити вихідне повідомлення s з отриманого повідомлення $y=G(s)\oplus t$, $s\in V_m$, $t\in V_r$. Така конструкція базується на шифросистемі з відкритим ключем, запропонованої

МакЕлісом [77].

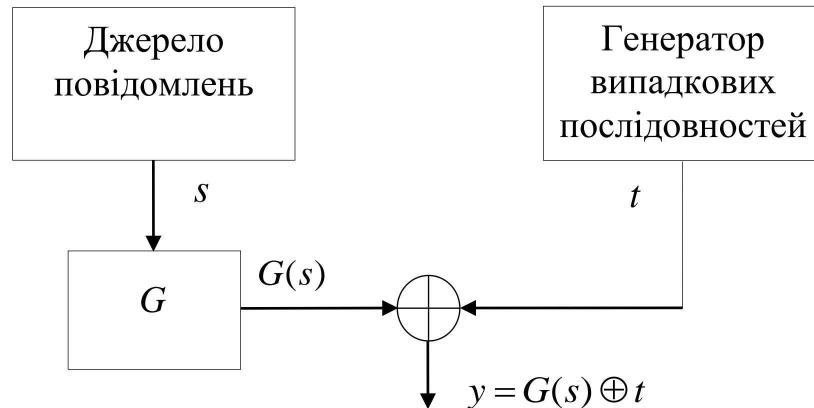


Рис. 1.9. Схема РБШ, побудована за методом 1.7.

Метод 1.8 (рис. 1.10). Як видно з рисунку, при використанні даного методу випадкова послідовність t додається за модулем 2 до повідомлення s, яке спочатку кодується корегувальним кодом G, після чого результуючий блок зашифровується перетворенням F_k. Результуюче повідомлення може бути записане у вигляді $y = F_k(G(s) \oplus t)$, $s \in V_m$, $t \in V_r$.

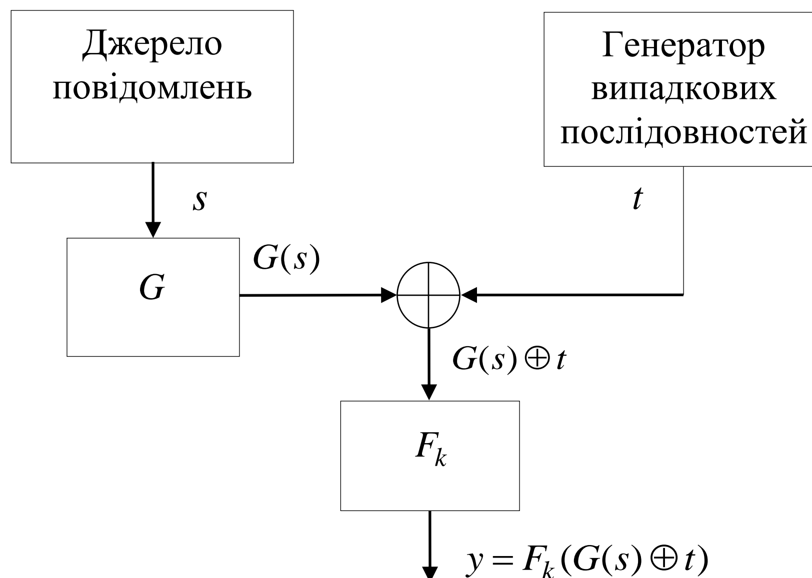


Рис. 1.10. Схема РБШ, побудована за методом 1.8.

Способи побудови РБШ за методами 1.1. і 1.2 описані також у монографіях [3, 4], при цьому не повідомляється жодних відомостей про стійкість цих шифросистем.

В цілому, наведені вище методи 1.1 – 1.10 є достатньо архаїчними і базуються на застосуванні тільки лінійних перетворень в конструкціях рандомізаторів. В [78, 79] запропоновано загальний метод побудови таких РБШ на основі гомоморфізмів групової структури на множині відкритих повідомлень у скінченні абелеві групи. Для цього методу отримано аналітичні оцінки параметрів, що характеризують стійкість побудованих РБШ відносно різницевого криптоаналізу. Поряд з тим, як показано в [80], в окремих випадках (наприклад, при застосуванні групування відкритих повідомлень при криптоаналітичній атаці) лінійне випадкове кодування не дозволяє виключити криптографічну слабкість, яка може бути закладена в конструкцію блокового шифру, а іноді навіть підсилює її ефект.

З метою подолання зазначеної небажаної властивості РБШ з лінійним випадковим кодуванням, в [80, 81] запропоновано *метод побудови РБШ на основі нелінійних перетворень* (рис. 1.11).

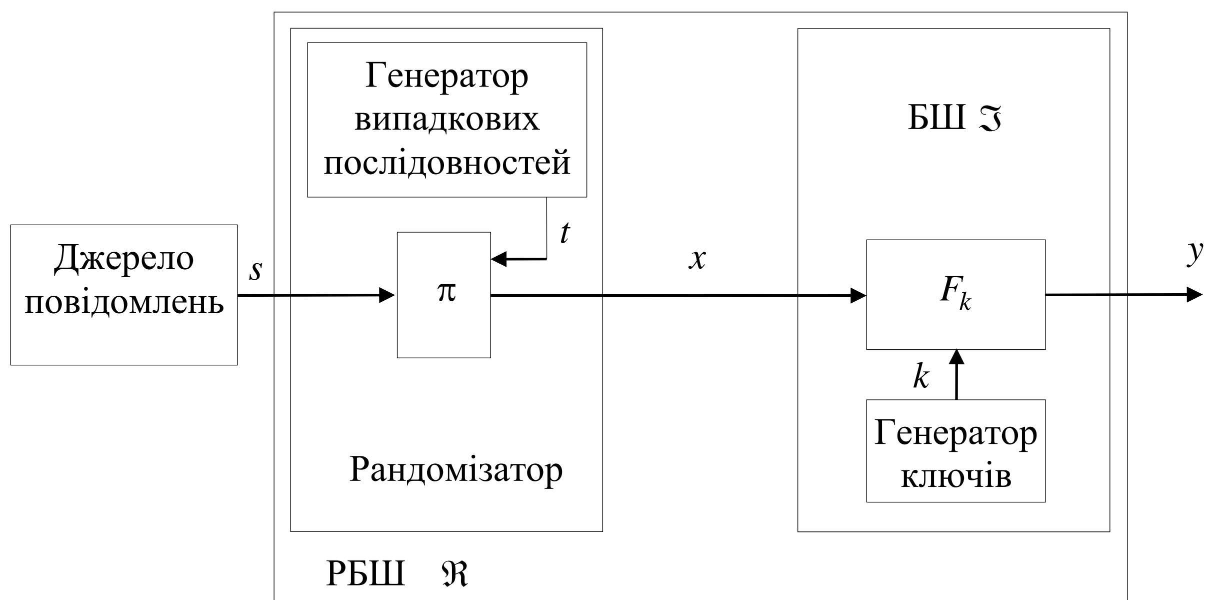


Рис. 1.11. Схема РБШ з нелінійним випадковим кодуванням

В [81] показано, що за певних умов відносно вибору нелінійного перетворення π в конструкції рандомізатора РБШ на рис. 1.11 отримані рандомізовані шифросистеми мають обґрунтовану стійкість як відносно статистичних (лінійних, різницевих), так і відносно алгебраїчних атак на основі комутативних діаграм. В [82] досліджено питання про можливість ефективної алгоритмічної реалізації нелінійних перетворень для побудови РБШ з нелінійним випадковим кодуванням.

В цілому, метод, викладений в [80, 81], видається перспективним для практичних застосувань, проте, враховуючи специфіку атак на потокові шифри, питання про можливість безпосереднього застосування цього методу для побудови РПШ є відкритим.

1.3.2. Огляд відомих методів побудови рандомізованих потокових шифросистем. Конструкції більшості сучасних потокових шифрів базуються на використанні *генераторів псевдовипадкових послідовностей* (*гам*), в ролі яких може виступати, наприклад, нелінійний регістр зсуву зі зворотним зв'язком. Процес шифрування потоковим шифром також залежить від його поточного стану, що дає змогу застосовувати рандомізацію не тільки до повідомлення, але і до самого стану.

Для опису наведених нижче методів побудови рандомізованих потокових шифросистем введемо наступні додаткові позначення: P – генератор псевдовипадкових двійкових послідовностей, $Func$ – функція, що приймає одне з двох можливих значень в залежності від певної умови.

Метод 2.1 (рис. 1.12). Дана конструкція використовує два генератори псевдовипадкових послідовностей ($P1$ і $P2$) та здійснює випадкове вставлення нульових символів у вихідний потік, формування якого описується за допомогою правила

$$y = Func(p1, p2 \oplus s, t),$$

де $Func(p1, p2 \oplus s, t) = t$, якщо $p1 = 0$ або $Func(p1, p2 \oplus s, t) = p2 \oplus s$, якщо $p1 = 1$ [40].

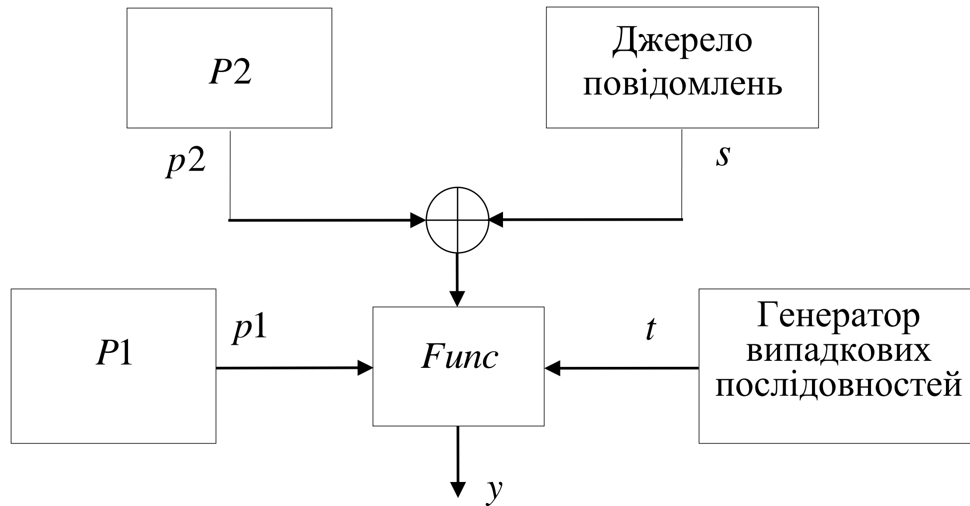


Рис. 1.12. Схема РПШ, побудована за методом 2.1.

Метод 2.2 (рис. 1.13). На відміну від попередньої, дана конструкція здійснює випадкове вставлення нульових символів у вхідний потік перед його зашифровуванням. Як видно з рисунку 1.13, реалізація цього методу передбачає наявність двох генераторів псевдовипадкових послідовностей і двох генераторів випадкових послідовностей.

Формування результуючої послідовності відбувається таким чином: $y = ((p1 \oplus t1), Func(t1, t2, s) \oplus p2)$, де $Func(t1, t2, s) = t2$, якщо $t1 = 0$ або $Func(t1, t2, s) = s$, якщо $t1 = 1$. Наявність компоненти $p1 \oplus t1$ у вихідній послідовності є необхідною для однозначного розшифровування отриманого значення y законним одержувачем.

Метод 2.3 (рис.1.14). Основна ідея цього методу побудови полягає у застосуванні рандомізації до функції переходу станів, а саме до регістру зсуву, в якому зберігається поточний стан шифросистеми.

На кожному кроці шифрування відбувається оновлення поточного стану шифросистеми шляхом зсуву регістру на одну позицію ліворуч та подальшим

занесенням у звільнену праву комірку нового випадкового значення.

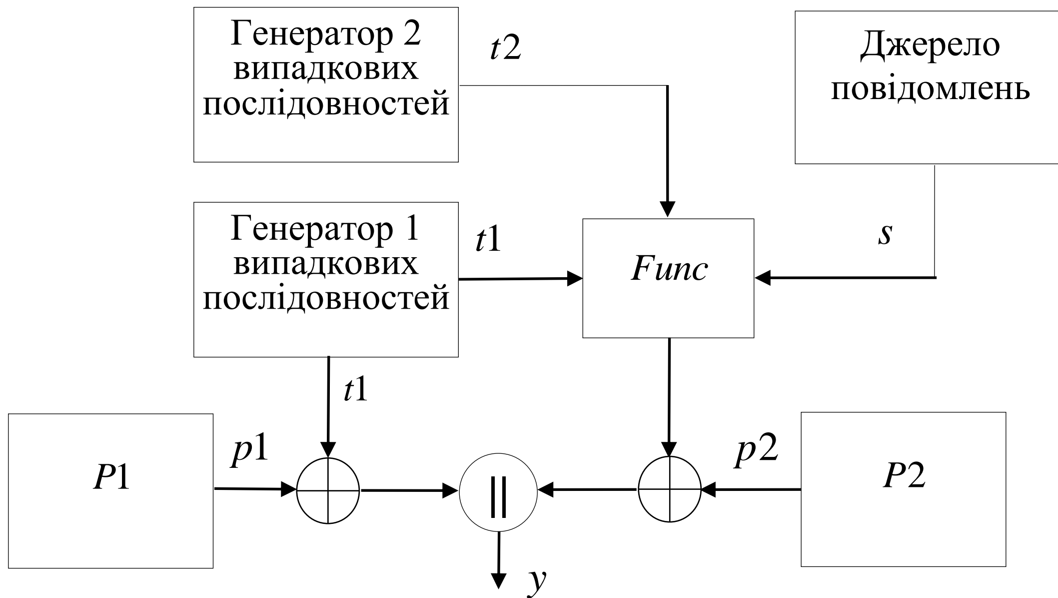


Рис. 1.13. Схема РПШ, побудована за методом 2.2.

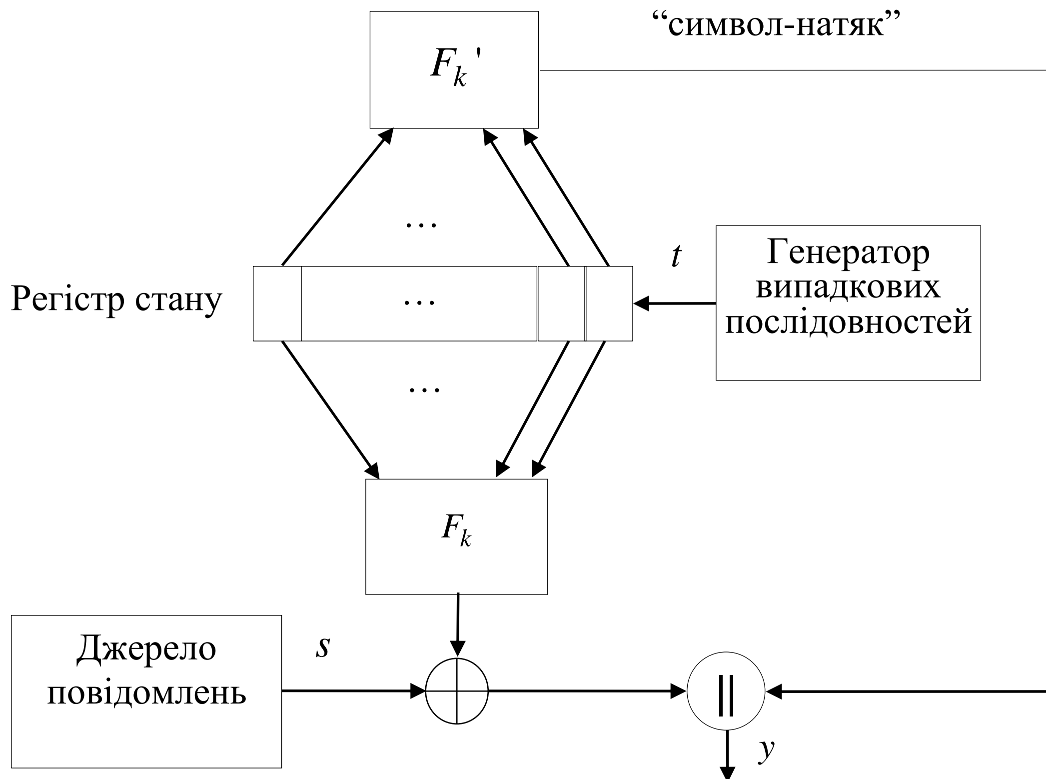


Рис. 1.14. Схема РПШ, побудована за методом 2.3.

Зашифровування кожного символу повідомлення здійснюється шляхом його додавання за модулем 2 до значення шифрувального перетворення F_k, яке залежить від поточного стану. З метою однозначного розшифровування повідомлення y законним одержувачем відправник посилає з кожним зашифрованим символом повідомлення так званий «символ-натяк», що допомагає визначати відповідний поточний стан шифросистеми. Даний символ є результатом застосування функції F_k' до поточного стану [40].

В цілому, наведені методи побудови РПШ призводять до складнореалізовних, а тому малопрактичних конструкцій шифросистем, для яких, до того ж, відсутні оцінки стійкості відносно відомих криптоаналітичних атак.

1.3.3. Системи передачі інформації каналом зв'язку з відводом. Добре відомо, що рандомізація є потужним методом захисту дискретних повідомлень, які передаються каналами зв'язку з відводом. Тут зашифровування відкритих повідомлень не відбувається, однак передбачається, що відвідний канал є "більш шумним" у порівнянні з основним каналом зв'язку. В цьому випадку метою рандомізації є створення високої ненадійності (невизначеності) повідомлення в каналі перехоплення при збереженні максимально можливої швидкості передачі в основному каналі та надійного зв'язку між законними користувачами. Вперше така модель передачі описана А. Вайнером в роботі [43], де запропонована *концепція відвідного каналу* (wire-tap channel) і обґрунтована принципова можливість безпечної передачі секретних повідомлень каналом зв'язку з відводом (рис. 1.15).

Пізніше в [44] запропонована більш загальна модель системи зв'язку, що містить відвідний канал (рис. 1.16), і отримано узагальнення основних результатів [43].

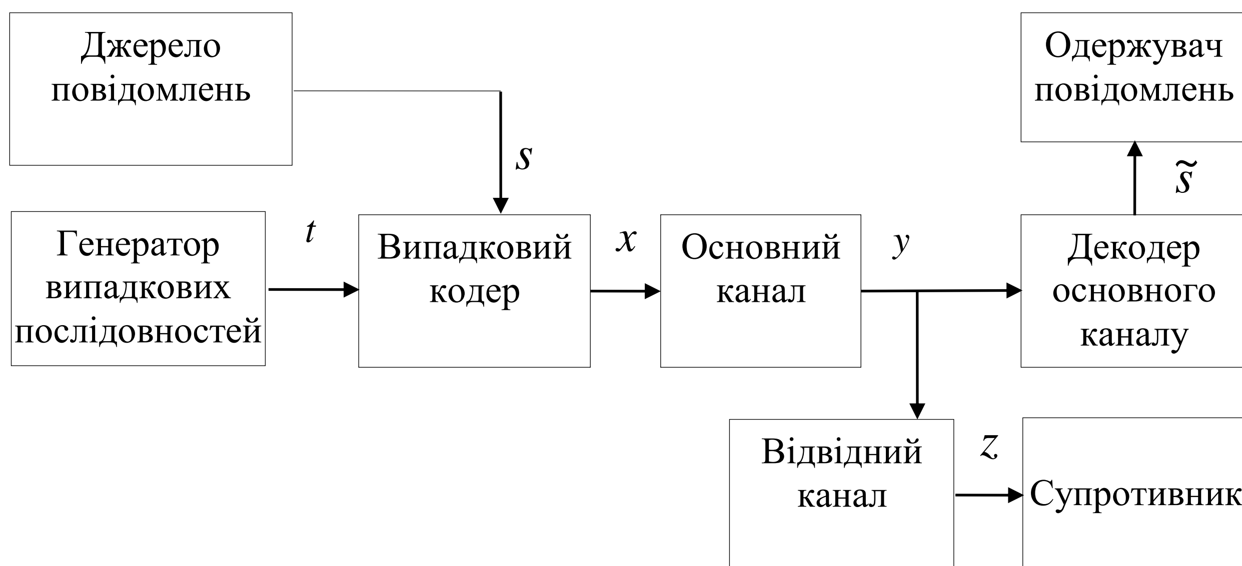


Рис. 1.15. Модель А. Вайнера системи передачі інформації каналом зв’язку з відводом

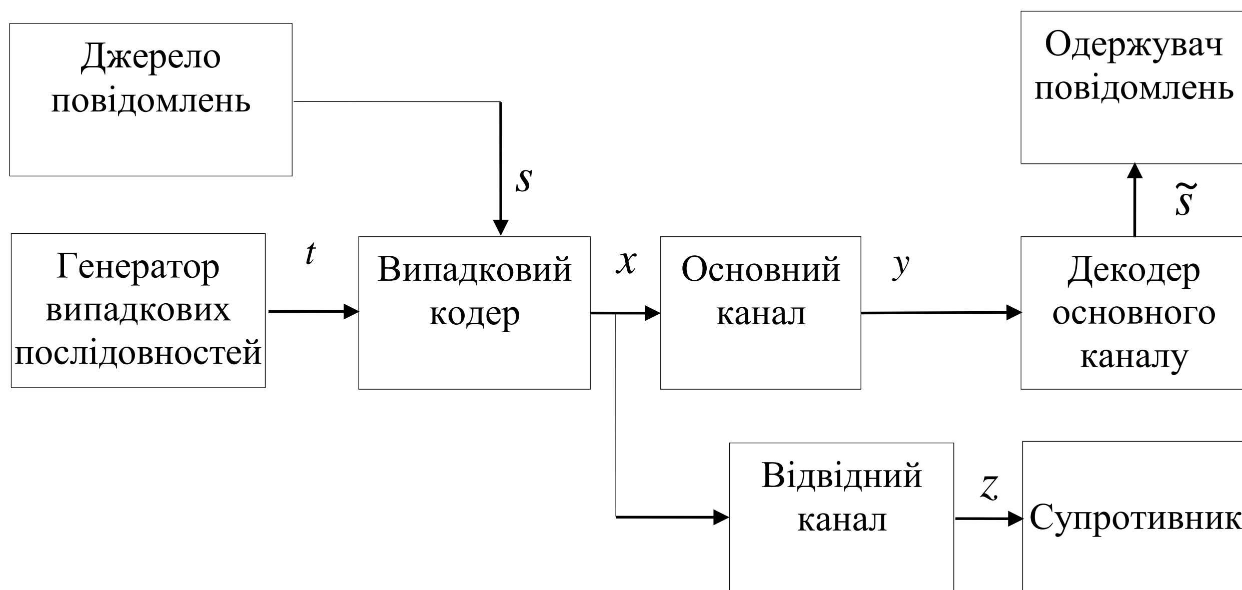


Рис. 1.16. Модель І. Чісара та Я. Кернера системи передачі інформації каналом зв’язку з відводом

Система зв’язку, зображена на рисунках 1.15 і 1.16, складається з безнадлишкового джерела і двох дискретних каналів без пам’яті з вхідним алфавітом F: основного каналу – від джерела до законного одержувача та

відвідного каналу – від джерела до супротивника. Для передачі повідомлення s, виробленого джерелом, застосовується випадкове кодування, при якому s перетворюється у повідомлення x, що вибирається випадково і рівноймовірно з відповідної підмножини множини F^n. Законний одержувач спостерігає повідомлення y на виході основного каналу зв’язку, а супротивник – повідомлення z на виході відвідного каналу.

На теперішній час концепція відвідного каналу охоплює широке коло теоретичних і прикладних задач в областях криптографії і стеганографії [54, 84 – 87], дозволяючи будувати уніфіковані математичні моделі ймовірносно-криптографічних систем і розв’язувати, певною мірою, єдиними загальними методами різні за своєю прикладною спрямованістю задачі.

В рамках класичної концепції відвідного каналу в останні роки помітний розвиток отримав напрямок, який має своєю задачею розробку математичних основ теорії кодового захисту інформації [47], що включає, зокрема, методи побудови неасимптотичних оцінок стійкості і практично ефективних алгоритмів кодування-декодування повідомлень в системах передачі інформації каналом зв’язку з відводом [46, 48, 49, 88 – 92]. До важливих, з теоретичного погляду, найбільш повно досліджених систем передачі з випадковим кодуванням відносяться системи, що складаються з ідеального основного і двійкового симетричного відвідного каналів зв’язку, в яких для реалізації випадкового кодування використовуються зрівноважені булеві відображення [48, 91 – 97] або двійкові лінійні коди [43, 44, 46 – 49, 88 – 90, 98 – 105]. Використання в системах з випадковим кодуванням нелінійних систематичних кодів, що мають “великі” дуальну відстань і швидкість передачі, а також володіють “ефективно обчислювальними” лінійними представленнями над комутативними кільцями [106 – 110] (зокрема, кодів Препарати), дозволяє за певних умов підвищити ефективність вказаних систем у зрівнянні з ефективністю аналогічних систем передачі з випадковим кодуванням лінійними блоковими кодами.

1.3.4. Рандомізовані потокові шифросистеми Міхалевича-Імаї. В роботах М. Міхалевича та Х. Імаї [59 – 62] запропоновано загальний підхід до побудови рандомізованих потокових шифросистем на основі спільного застосування процедур шифрування, завадостійкого кодування та випадкового кодування. Головною метою створення таких шифросистем є підвищення стійкості (без суттєвого зниження практичності) потокових шифрів, які використовуються на даний час в системах бездротового зв'язку, зокрема, в стандарті мобільної телефонії GSM. Іншою метою є розробка симетричних шифросистем, що мають обґрунтовану обчислювальну стійкість, яка базується на складності розв'язання певних відомих математичних задач. Як приклад, зазначимо задачу LPN (Learning from Parity with Noise), яка полягає у розв'язанні системи спотворених булевих лінійних рівнянь з випадковою рівноймовірною матрицею коефіцієнтів. Задача LPN відноситься до класу NP-повних [111], отже, для неї не відомо (та, ймовірно, не існує) поліноміальних алгоритмів. Один із перших прикладів нерандомізованих криптосистем, стійкість яких базується на складності розв'язання цієї задачі, запропоновано в [112].

В подальшому зосередимо увагу на рандомізованих потокових системах шифрування, представлених у [61, 62] та детально досліджених в [62 – 64].

Позначимо V_n множину двійкових векторів довжини n; $F_{m\times n}$ – множину $m\times n$-матриць над полем $F=\mathbf{GF}(2)$, $F^*_{m\times m}$ – групу оборотних матриць порядку m над цим полем.

Згідно з [61, 62], вхідними даними для побудови рандомізованої потокової шифросистеми Міхалевича-Імаї з параметрами $l,m,n\in\mathbf{N}$, $0<p<1/2$, де $l<m<n$, та множиною ключів K є такі (несекретні) об'єкти:

– твірна матриця G_1 двійкового лінійного $[n,m]$-коду C_1 з ефективним алгоритмом декодування (декодером) $D:V_n\to C_1$, який дозволяє надійно

виправляти помилки у двійковому симетричному каналі з імовірністю спотворення p;

– матриця $G_2 \in F^*_{m\times m}$;

– генератор гами, що виробляє за ключем $k \in K$ послідовність $f_0(k), f_1(k), \ldots$ двійкових векторів довжини n (при цьому вважається, що функції $f_i : K \to V_n$, $i = 0, 1, \ldots$, можуть залежати від загальнодоступних параметрів, наприклад, векторів ініціалізації).

Для зашифровування на ключі $k \in K$ відкритого тексту $s_0, s_1, \ldots, s_t$, де $s_i \in V_l$, $i = 0, 1, \ldots, t$, відправник генерує послідовність незалежних випадкових векторів $u_0, v_0, u_1, v_1, \ldots, u_t, v_t$, де вектор u_i має рівномірний розподіл на множині V_{m-l}, а координати вектора v_i довжини n розподілені за законом Бернуллі з імовірністю успіху p, та обчислює шифрований текст $z_0, z_1, \ldots, z_t$ за формулою

$$z_i = (s_i, u_i) G_2 G_1 \oplus f_i(k) \oplus v_i,\ i \in \overline{0, t}. \tag{1.1}$$

Законний одержувач, знаючи вектор $f_i(k)$, може швидко відновити повідомлення $(s_i, u_i) G_2$ за допомогою декодера D, а потім знайти вектор s_i, використовуючи оборотність матриці G_2.

В [62 – 64] досліджено різні варіанти побудови рандомізованих шифросистем наведеного вигляду, зокрема, з такими функціями f_i:

$$f_i(k) = kS^i,\ k \in K = V_n, \tag{1.2}$$

де S є відомою $n \times n$-матрицею над полем F;

$$f_i(k) = \alpha_i k,\ k \in K = F_{n\times n}, \tag{1.3}$$

де $\alpha_0, \alpha_1, \ldots$ є незалежними випадковими рівноймовірними булевими векторами довжини n. Зауважимо, що в останньому випадку шифротекст визначається як послідовність $(\alpha_0, z_0), (\alpha_1, z_1), \ldots, (\alpha_t, z_t)$, де вектори z_i обчислюються за формулою (1.1). Строго кажучи, рандомізовані системи шифрування цього типу не відносяться до класу потокових та запропоновані в [62] з метою вдосконалення однієї з криптосистем, стійкість яких базується на складності розв'язання задачі LPN [111].

Виходячи з умови простоти реалізації наведених систем шифрування, в [63] запропоновано задавати матриці G_1 і G_2 таким чином:

$$G_1 = \begin{pmatrix} I_{m-l} & 0 & A_1 \\ 0 & I_l & A_2 \end{pmatrix}, \; G_2 = \begin{pmatrix} 0 & I_l \\ I_{m-l} & B \end{pmatrix}, \tag{1.4}$$

де $A_1 \in F_{(m-l)\times(n-m)}$, $A_2 \in F_{l\times(n-m)}$, $B \in F_{(m-l)\times l}$, а I_l та I_{m-l} – одиничні матриці зазначеного розміру. Такий вибір матриць по суті не звужує клас шифросистем, що розглядаються, однак не є обов'язковим. В цьому випадку перетворення

$$s \mapsto (s,u)G_2G_1 = (u,\, s \oplus uB,\, sA_2 \oplus u(A_1 \oplus BA_2)),\; s \in V_l,\; u \in V_{m-l}, \tag{1.5}$$

що використовується у формулі (1.1), описує добре відому схему комбінованого випадкового кодування для каналу зв'язку з відводом, запропоновану в [43] та детально досліджену в багатьох інших публікаціях (див. роботи [113, 114] та наведені в них посилання).

Оцінювання стійкості рандомізованих шифросистем, що визначаються за допомогою співвідношення (1.1), як у теоретико-інформаційному, так і в обчислювальному сенсі, проведено в [62 – 64].

В [62] автори стверджують, що за умови (1.3) складність відновлення ключа шифросистеми, що розглядається, при проведенні атаки на основі підібраних відкритих текстів є не менше ніж складність розв’язання задачі LPN з імовірністю спотворень $1/2\cdot\left(1-(1-2p)^{(m-l)/2}\right)$. Зауважимо, що при доведенні цього результату використовується певне припущення стосовно матриці G_2G_1, яке не згадується в основній частині статті (див. доведення теореми 2 в роботі [62]).

В роботі [63] автори показують, що за умов (1.1), (1.2), (1.4) відновлення ключа шифросистеми при проведенні зазначеної вище атаки зводиться до розв’язання певної системи булевих лінійних рівнянь зі спотвореними правими частинами, де ймовірності спотворень є не менше ніж $p_w=1/2\cdot\left(1-(1-2p)^{w+1}\right)$, а w залежить певним чином від матриць G_1 та G_2. Поряд з тим, зауважимо, що наведена авторами умова $\mathrm{rank}(G^*)\geq w+1$ (разом з іншими умовами; див. [63], с. 11) не гарантує, що ймовірності спотворень у правій частині отриманої системи лінійних рівнянь обмежені знизу значенням p_w, в чому можна переконатися, аналізуючи приклад, наведений у тій самій роботі (див. [63], с. 15).

В цілому, проведений аналіз дозволяє зробити такі висновки:

1) більшість відомих методів побудови рандомізованих шифросистем зводяться до певних варіантів лінійного випадкового кодування та наступного зашифровування відкритих повідомлень;

2) при дослідженні впливу рандомізації на теоретичну стійкість захисту інформації або ключа основна увага приділяється, як правило, побудові ефективних (за різними показниками) алгоритмів рандомізації повідомлень джерела без врахування специфіки процедур наступного їх зашифровування, або сумісній побудові моделей шифрів і алгоритмів рандомізації повідомлень, що зашифровуються;

3) єдиним прикладом РПШ, які будуються на основі регулярного методу та, в принципі, можуть бути використані на практиці, є шифросистеми Міхалевича-Імаї; поряд з тим, стійкість цих шифросистем суттєво залежить від будови їх компонент і може бути менше, ніж стверджують розробники;

4) в доступних публікаціях не зазначено про наявність методів побудови обчислювально стійких рандомізованих потокових шифросистем з нелінійним випадковим кодуванням.

Наведені факти свідчать про певне протиріччя між потребами в обчислювально стійких та практичних рандомізованих потокових шифросистемах, що необхідні для забезпечення безпеки державних інформаційних ресурсів, з одного боку, та відсутністю розвинутих методів їх побудови з іншого. Зазначене протиріччя визначає напрямок, характер та окремі задачі дисертаційного дослідження.

1.4. Основні напрями та окремі задачі дисертаційного дослідження

Вище показана актуальність *наукової задачі* дисертаційної роботи, що полягає в розробці методу побудови рандомізованих потокових шифросистем з нелінійним випадковим кодуванням для забезпечення безпеки державних інформаційних ресурсів

Метою дисертаційної роботи є підвищення криптографічної стійкості потокових шифрів шляхом рандомізації джерела відкритих повідомлень для забезпечення безпеки державних інформаційних ресурсів.

Об’єктом дослідження в роботі є процес криптографічного перетворення інформації у рандомізованих потокових шифросистемах, а *предметом дослідження* – методи побудови рандомізованих потокових шифросистем з

нелінійним випадковим кодуванням, призначених для забезпечення безпеки державних інформаційних ресурсів.

У відповідності до поставленої мети, наукова задача дисертаційної роботи включає в себе ряд взаємопов'язаних окремих задач, порядок розв'язання яких визначає основні напрями дисертаційних досліджень (рис. 1.17).

Перший напрям полягає в оцінюванні обчислювальної стійкості РПШ Міхалевича-Імаї (як відносно відомих, так і нових атак), та в з'ясуванні потенційних можливостей цих шифросистем і знаходженні загальних обмежень, яким задовольняють окремі показники їх ефективності при заданих значеннях інших показників (задачі 2 – 4 на рис. 1.17). Основною задачею *другого напряму* є розробка методу побудови РПШ з нелінійним випадковим кодуванням, оцінювання та обґрунтування їх обчислювальної стійкості, а також розробка програмних реалізацій зазначених РПШ (задачі 5 – 6 на рис. 1.17).

Вирішення перелічених окремих задач дозволяє вирішити наукову задачу дисертаційної роботи та досягнути поставленої в роботі мети.

Висновки

1. Важливою задачею забезпечення інформаційної безпеки держави є розробка методів підвищення стійкості потокових шифрів без внесення змін в алгоритми шифрування шляхом застосування додаткових перетворень, які не потребують ключів, можуть бути відносно просто реалізовані та забезпечують науково обґрунтований рівень стійкості систем шифрування в цілому. Ця задача є дуже актуальною для спеціальних (військових) додатків, витоки конфіденційної інформації в яких створюють ризики для інформаційної безпеки держави.

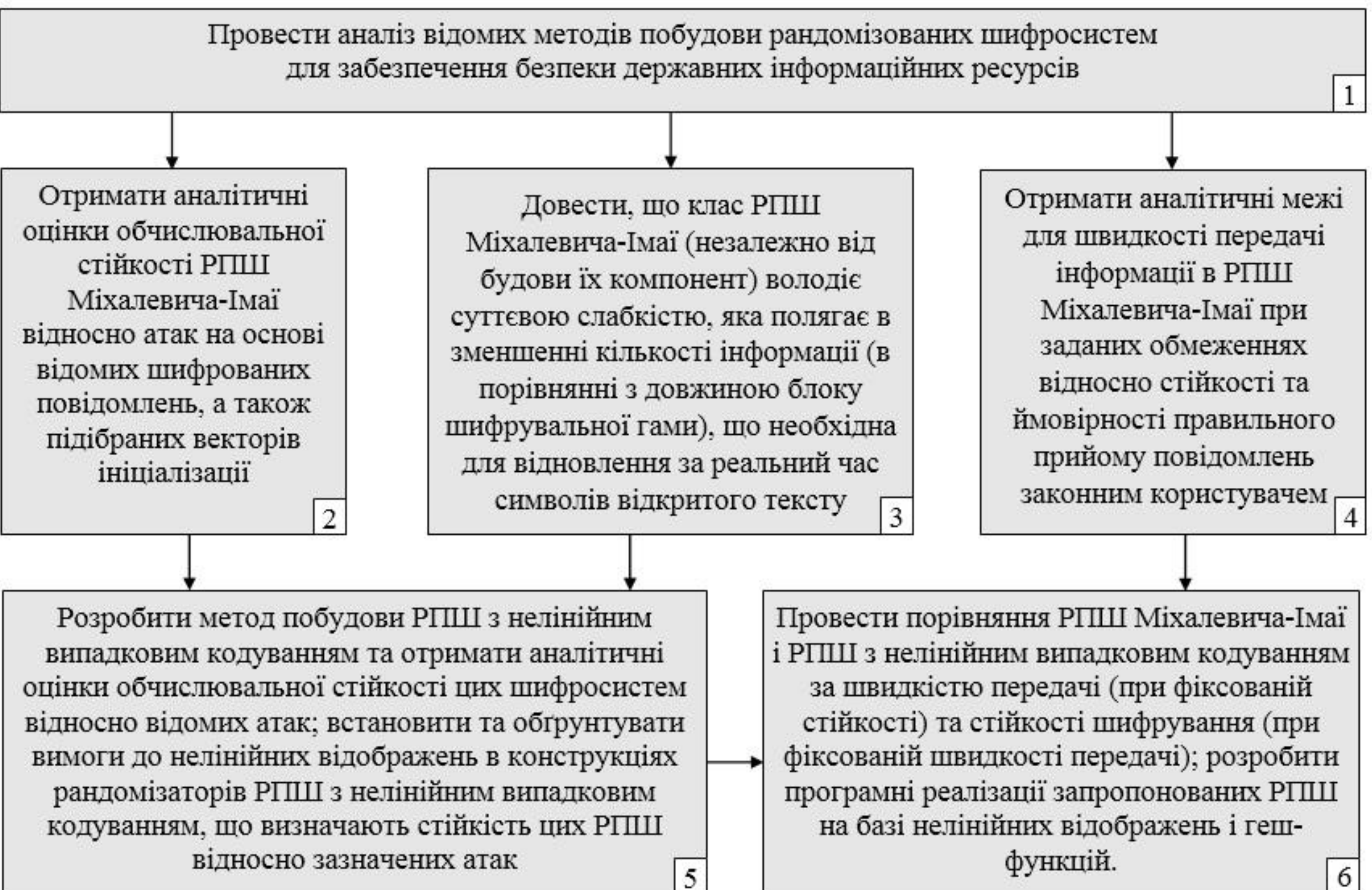


Рис. 1.17. Окремі задачі досліджень

Такими додатками є ті, в яких: часто трапляються випадки компрометації шифрувальної апаратури або алгоритму шифрування, відмови системи блокування шифратора, внаслідок чого в канал передачі може потрапити «слабка» гама шифрування; передаються короткі повідомлення (спеціальні команди чи військові накази); є невеликим навантаження на інформаційний трафік; криптографічна стійкість є важливішою за швидкість передачі інформації; є невеликою кількість абонентів; алгоритм шифрування може бути невідомим. Одним з таких загальних методів є рандомізація або випадкове кодування джерела відкритих повідомлень. Метод рандомізації є відомим достатньо давно, проте, саме для випадку потокових шифрів, його можливості досліджені не повністю.

2. Більшість відомих методів побудови рандомізованих шифросистем зводяться до певних варіантів лінійного випадкового кодування та наступного зашифровування відкритих повідомлень. Крім того, відомі методи побудови рандомізованих блокових шифросистем з нелінійним випадковим кодуванням не можуть бути безпосередньо застосовані для побудови рандомізованих потокових шифросистем через специфіку атак саме на потокові шифри.

3. Шифросистеми Міхалевича-Імаї є єдиним відомим прикладом рандомізованих потокових шифросистем, які будуються на регулярній основі та, в принципі, можуть бути використані на практиці. Рандомізатори зазначених РПШ будуються на основі двійкових лінійних перетворень, зокрема, завадостійкого кодування відкритих повідомлень лінійними кодами. Проте стійкість РПШ Міхалевича-Імаї суттєво залежить від будови їх компонент і може бути значно менше, ніж стверджують їх розробники. Деякі з цих шифросистем виявляються вразливими навіть до атак на основі відомих шифрованих повідомлень і, отже, не можуть бути використані для захисту державних інформаційних ресурсів.

4. У відповідності до поставленої мети, розв’язання наукової задачі дисертаційної роботи включає в себе дослідження за двома напрямами. Перший

напрям полягає в оцінюванні обчислювальної стійкості РПШ Міхалевича-Імаї, а також з'ясуванні потенційних можливостей цих шифросистем та знаходженні загальних обмежень, яким задовольняють окремі показники їх ефективності при заданих значеннях інших показників. Основною задачею другого напряму є розробка методу побудови РПШ з нелінійним випадковим кодуванням, оцінювання їх обчислювальної стійкості, а також розробка програмних реалізацій зазначених РПШ.

Список використаних джерел у першому розділі

1. *Проект «Концепції інформаційної безпеки України»*, Режим доступу: http://mip.gov.ua/done_img/d/30-project_08_06_15.pdf.

2. *Закон України «Про захист інформації в інформаційно-телекомунікаційних системах»*, Режим доступу: http://zakon2.rada.gov.ua/laws/show/80/94-вр.

3. Шнайер Б. *Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си*, Пер. с англ., М.: Триумф, 816 с., 2002.

4. Молдовян А.А., Молдовян Н.А., Гуц Н.Д., Изотов Б.В. *Криптография: скоростные шифры*, СПб.: БХВ-Петербург, 496 с., 2002.

5. Фергюссон Н., Шнайер Б. *Практическая криптография*, Пер. с англ., М.: "Вильямс", 424 с., 2005.

6. Харин Ю.С., Агиевич С.В. *Компьютерный практикум по математическим методам защиты информации*, Учебное пособие, Минск.: Изд-во БГУ, 2001, 191 с.

7. Харин Ю.С., Берник В.И., Матвеев Г.В. *Математические основы криптологии*, Минск.: Изд-во БГУ, 1999, 319 с.

8. Gurkaynak F., Luethi P., Bernold N., Blattman R., Goode V., Marghitola M., Kaeslin H., Felber H., Fichtner W. «Hardware evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS,

Trivium, VEST, ZK-Crypt», Retrieved October 25, 2016, URL: http://www.ecrypt.eu.org/stream/papersdir/2006/015.pdf.

9. Ekdahl P. «*On LFSR based stream ciphers – analysis and design*», Ph.D. Dissertation, LUND University, 2003, Sweden.

10. Ємець В., Мельник А., Попович Р. *Сучасна криптографія. Основні поняття*, Львів: БаК, 144 с., 2003.

11. Ekdahl P., Johansson T. «A new version of the stream cipher SNOW», *Selected Areas in Cryptography* – SAC 2002, Springer-Verlag, 2003., P.47-61.

12. Rose G., Hawkes P. «Turing: a fast stream cipher», *International Workshop on Fast Software Encryption*, 2003, P. 290-306.

13. Tasheva A., Tasheva Z., Nakov O. «Software stream cipher based on pGSSG generator», *International Journal of Cyber-Security and Digital Forensics*, 3(2), 2014, P. 11-121.

14. Dierks T., Rescorla E. «The transport layer security (TLS) protocol Version 1.2», [Electronic resource], URL: https://tools.ietf.org/html/rfv5246.

15. Potter B. «Wireless Security's Future», *IEEE Security and Privacy*, 2003, P. 68-72.

16. Stallings W. «*Cryptography and Network Security*», Pearson, 2011.

17. Aminghafari V., Hu H. «Fruit-v2: ultra-lightweight stream cipher with shorter internal state», URL: https://eprint.iacr.org/2016/355.pdf.

18. Hamann M., Krause M., Meier W. «LIZARD – A Lightweight Stream Cipher for Power-constrained Devices», *IACR Transactions on Symmetric Cryptology*, 2017, P. 45-79.

19. Armknecht F., Mikhalev V. «On lightweight stream ciphers with shorter internal states», *FSE, Lecture Notes in Computer Science*, V. 9054, 2015, P. 451-470.

20. Kashmar H., Ismail E. «BLOSTREAM: A high speed stream cipher», *Journal of Engineering Science and Technology*, V. 12, №4, 2017, P. 111-1128.

21. Hell M., Johansson T., Meier W. «Grain: a stream cipher for constraint environments», *IJWMC*, 2(1), 2007, P. 86-93.

22. Agren M., Hell M., Johansson T., Meier W. «Grain-128a: a new version of grain-128 with optional authentication», *IJWMC*, 5(1), 2011, P. 48-59.

23. Dubrova E., Hell M. «Espresso: A stream cipher for 5g wireless communication systems», *Cryptography and Communications*, 9(2), 2017, P. 273-289.

24. Fan X., Mandal K., Gong G. «WG-8: A lightweight stream cipher for resource-constrained smart devices», *ICST Trans. Security Safety*, 2(3):e4, 2015.

25. Good T., Benaissa M. «Hardware performance of estream phase-iii stream cipher candidates», *Proc. of Workshop on the State of the Art of Stream Ciphers (SACS 2008)*, 2008.

26. Mikhalev V., Armknecht F., Muller C. «On ciphers that continuously access the non-volatile key», *IACR Transactions on Symmetric Cryptology*, 2016(2), 2017, P. 52-79.

27. Babbage S., Dodd M. «The mickey stream ciphers», *New Stream Cipher Design*, *Springer*, 2008, P. 191-209.

28. De Canniere Ch., Preneel B. «Trivium specifications», *eSTREAM*, *ECRYPT Stream Cipher Project*, 2006.

29. Mentens N., Genoe J., Preneel B., Verbauwhede I. «A low-cost implementation of Trivium», *Preproceedings of SASC*, 2008, P. 197-204.

30. David M., Ranasinghe D., Larsen T. «A2u2: a stream cipher for printed electronics rfid tags», *IEEE International Conference*, RFID, 2011, P. 176-183.

31. Juels A. «RFID security and privacy: a research survey», *IEEE Journal on Selected Areas in Communications*, V. 24, 2006, P. 381-394.

32. Elbayoumy A., Shepherd S. «A high grade secure VoIP system using the tiny encryption algorithm», *7th Annual International Symposium on Advanced Radio Technologies*, *Proceedings*, 2005, P. 342-350.

33. Anderson R., Manifavas C. «Chameleon – A new kind of stream cipher» *Fast Software Encryption*, *Lecture Notes in Computer Science*, *Springer*, V. 1267, 1997, P. 107-113.

34. Fontaine C. E0 (Bluetooth). In: van Tilborg H.C.A. (eds) Encyclopedia of Cryptography and Security, Springer, 2005.

35. *The evolution of 802.11 Wireless Security*, URL: https://benton.pub/research/benton_wireless.pdf.

36. Biham E., Dunkelman O. «Cryptanalysis of the A5/1 GSM stream cipher», *Lecture Notes in Computer Science*, Indocrypt 2000, V. 1977, 2000, P. 43-51.

37. *The Secure Sockets Layer (SSL) Protocol Version 3.0* [Electronic resource], URL: https://tools.ietf.org/html/rfc6101.

38. *Microsoft Point-to-Point Encryption (MPPE) Protocol* [Electronic resource], URL: https://tools.ietf.org/html/rfc3078.

39. Сторожук А.Ю. «*Методи оцінювання та обґрунтування стійкості потокових шифрів відносно статистичних атак на основі алгебраїчно вироджених наближень булевих функцій*», Дисертація на здобуття наукового ступеня кандидата технічних наук, [Electronic resource], URL: http://er.nau.edu.ua/bitstream/NAU/25171/1/dissert_Storozhuk.pdf.

40. Rivest R.L., Sherman A.T. «Randomization Encryption Techniques», *Advances in Cryptology, CRYPTO'82, Proceedings*, Springer Verlag, 1982, P. 145-167.

41. Gunter Ch.G. «A universal algorithm for homophonic coding», *Proc. of Eurocrypt' 88. Lecture Notes in Computer Science*, Springer-Verlag, 1988, P. 405-414.

42. Jendal H.N., Kuhn Y.J.B., Massy J.L. «An information-theoretic treatment of homophonic substitution», *Proc. of Eurocrypt' 89. Lecture Notes in Computer Science*, Springer-Verlag, 1989, P. 382-394.

43. Wyner A. D. «The Wire-Tap Channel», *Bell System Techn. J.*, V. 54, № 8, 1975, P. 1355-1388.

44. Csiszar I., Korner J. «Broadcast channels with confidential messages», *IEEE Trans. Inform. Theory*, V. 24, № 3, 1978, P. 339-348.

45. Massey J. L. «An Introduction to Contemporary Cryptology», *Proc. IEEE*, V. 76, № 5, 1988, P. 533-549.

46. Коржик В. И., Яковлев В. А. «Неасимптотические оценки кодового зашумления одного канала», *Проблемы передачи информации*, Т. 17, В. 4, 1981, С. 11-18.

47. Горицкий В. М. «Вероятностная криптография в системах защиты информации: кодовая защита», *Электроника и связь*, В. 5, 1998, С. 140-145.

48. Иванов В. А. «О методе случайного кодирования», *Дискретная математика*, Т. 11, В. 3, 1999, С. 99-108.

49. Коржик В. И., Яковлев В. А. «Пропускная способность канала связи с внутренним случайным кодированием», *Проблемы передачи информации*, Т. 28, В. 4, 1992, С. 24-34.

50. Goldwasser S., Micali S. «Probabilistic encryption», *Journal of Computer and System Sciences*, V. 28, 1984, P. 270-299.

51. Штарьков Ю. М. «Некоторые теоретико-информационные задачи защиты дискретных данных», *Проблемы передачи информации*, Т. 30, В. 2, 1994, С. 49-60.

52. Штарьков Ю. М., Юхансон Т., Смитс Б. Дж. М. «О совместной стойкости защиты информации и ключа в секретных системах», *Проблемы передачи информации*, Т. 34, В. 2, 1998, С. 117-127.

53. Maurer U. M. *Provable Security in Cryptography*: Diss. ETH № 9260, 1990, 120 p.

54. Чисар И. «Почти независимость случайных величин и пропускная способность криптостойкого канала», *Проблемы передачи информации*, Т. 32, В. 1, 1996, С. 48-57.

55. Welsh D. *Codes and Cryptography*, Oxford: Clarendon Press., 1978, 256 p.

56. Алексейчук А.Н. «Математическая модель и задачи анализа стойкости вероятностно-криптографических систем в системах защиты информации», *Захист інформації*, № 3, 2001, С. 5-12.

57. Алексейчук А.Н. «О вероятности безошибочного декодирования в отводном канале с аддитивным шумом, распределенным на конечной абелевой группе», *Защита информации: сборник научных трудов Национального авиационного ун-та*, 2001, С. 9-16.

58. Алексейчук А.Н., Кривоножко Г.Е. «Оптимальная схема декодирования сообщений безызбыточного источника в групповых вероятностно-криптографических системах», *Збірник наукових праць ІПМЕ НАН України*, В. 14, 2001, С. 26-33.

59. Mihaljević M.J., Imai H. «An approach for stream cipher design based on joint computing over random and secret data», *Computing*, V. 85, No 1-2, 2009, P. 153-168.

60. Mihaljević M.J., Imai H. «A stream ciphering approach based on wiretap channel coding», *8th Central European Conference of Cryptography 2008*, Graz, Austria, July 2-4, E-Proc. (3 p.), 2008.

61. Mihaljević M.J., Imai H. «An information-theoretic and computational complexity security analysis of a randomized stream cipher model», *4th Western European Workshop on Research in Cryptology – WeWoRC 2011*, Weimar, Germany, July 20-22, Conf. Record, 2011, P. 21-25.

62. Mihaljević M.J., Imai H. «Employment of homophonic coding for improvement of certain encryption approaches based on the LPN problem”, *Symmetric Key Encryption Workshop* – SKEW 2011, Copenhagen, Denmark, Feb. 16-17, E-Proc. (17 p.), 2011.

63. Mihaljević M.J., Oggier F., Imai H. «Homophonic coding design for communication systems employing the encoding-encryption paradigm», URL: http://arXiv:1012.5895v1.

64. Oggier F., Mihaljević M.J. «An information-theoretic analysis of the security of communication systems employing the encoding-encryption paradigm», URL: http://arXiv:1008.0968v1.

65. Шестюк В.П. «Проблемы обеспечения информационной безопасности в современном мире», *Математика и безопасность информационных технологий. Материалы конференции в МГУ 28 – 29 октября 2004 г.*, М.: МЦНМО, 2005, С. 11-17.

66. Фомичев В.М. *Дискретная математика и криптология*, М.: ДИАЛОГ-МИФИ, 2003, 400 с.

67. Kholosha A.A. «Clock-controlled shift registers for key-stream generation», URL: http://eprint.iacr.org/2001/061.pdf.

68. Бабаш А.В., Шанкин Г.П. *Криптография*, М.: Солон-Р, 2002, 511с.

69. Сачков В.Н. «Вероятностные преобразователи и правильные мультиграфы», *Труды по дискретной математике*, Т. 1, 1997, С. 227-250.

70. Gollman D., Chambers W.G. «Clock-controlled shift registers: a review», *IEEE J. on Selected Areas in Communication*, V. 7, № 4, 1989, P. 525-533.

71. Nair R., Yuen H.P., Eguchi T., Kumar P. «Quantum noise randomized cipher», URL: https://arxiv.org/pdf/quant-ph/0603263.pdf.

72. Потапов В.Н. «Обзор методов неискажающего кодирования дискретных источников», *Дискретный анализ и исследование операций*, Сер. 1, Т. 6, № 1, 1999, С. 49-91.

73. Рябко Б.Я., Фионов А.Н. «Быстрый метод полной рандомизации сообщений», *Проблемы передачи информации*, Т. 33, В. 3, 1997, С. 3-14.

74. Рябко Б.Я., Мачикина Е.П. «Эффективное преобразование случайных последовательностей в равновероятные и независимые», *Проблемы передачи информации*, Т. 35, В. 2, 1999, С. 23-28.

75. Рябко Б.Я. «Просто реализуемая идеальная криптосистема», *Проблемы передачи информации*, Т. 36, В. 1, 2001, С. 90-95.

76. Алферов А.П., Зубов А.Ю., Кузьмин А.С., Черемушкин А.В. *Основы криптографии*, М.: Гелиос АРВ, 2001, 480 с.

77. McEliece R.J. «A public-key cryptosystem based on algebraic coding theory», *Deep Space Network Progress Report 42-22*, 1978, P. 114-116.

78. Алексейчук А.Н., Васюков И.В., Корнейко А.В. «Метод построения и теоретико-информационный анализ стойкости рандомизированных криптосистем с секретным ключом», *Моделювання та інформаційні технології. Збірник наукових праць ІПМЕ НАН України*, Вип. 22, К.: ІПМЕ НАН України, 2003, С. 65-73.

79. Алексейчук А.Н., Васюков И.В., Корнейко А.В. «Обоснование стойкости вероятностных моделей рандомизированных блочных шифров к методу разностного криптоанализа», *Электронное моделирование*, Т. 26, №4, 2004, С. 23-25.

80. Алексейчук А.Н. «Достаточные условия стойкости рандомизированных блочных систем шифрования относительно метода криптоанализа на основе коммутативных диаграмм», *Реєстрація, зберігання і обробка даних*, Т. 9, №2, 2007, С. 61-68.

81. Алексейчук А.Н. «Аналитические оценки теоретической стойкости рандомизированных блочных систем шифрования относительно метода разностного криптоанализа», *Захист інформації*, № 3, 2007, С. 80-88.

82. Мохор В.В., Корнейко А.В. *Синтез нелинейных разрядных преобразователей в полях Галуа для рандомизированных схем криптографической защиты информаци*, К.: ООО «Прометей», 2017, 220 с.

83. Wagner D. Towards a unifying view of block cipher cryptanalysis, *Fast Software Encryption*, *FSE'04*, Proceedings, Springer Verlag, 2004, P. 116-135.

84. Maurer U.M., Massey J.L. «Perfect local randomness in pseudo-random sequences», *Advances in Cryptology, CRYPTO'89, Proceedings*, Springer Verlag, 1990, P. 110-112.

85. Bennet C.H., Brassard G., Maurer U.M. «Generalized privacy amplifications», *IEEE Trans. Inform. Theory*, V. 41, № 6, 1995, P. 1915-1923.

86. Ahlswede R., Csiszar I. «Common randomness in information theory and cryptography – Part 1: Secret sharing», *IEEE Trans. Inform. Theory*, V. 39, № 4, 1993, P. 1121-1132.

87. Korjik V., Morales-Luna G. «Information hiding though noisy channels», *IH' 2001, Proceedings*, Springer Verlag, 2001, P. 42-50.

88. Алексейчук А.Н. «Оценки эффективности кодовой защиты дискретных сообщений с использованием линейных кодов с большим дуальным расстоянием», *Реєстрація, зберігання і обробка даних*, Т. 3, № 2, 2001, С. 99-106.

89. Алексейчук А.Н., Дроздовский Т.А., Сергиенко Ю.В. «Система передачи информации со случайным кодированием, построенная на основе кодов Рида-Соломона», *Правове, нормативне та метрологічне забезпечення системи захисту інформації в Україні*, В. 6, 2003, С. 84-89.

90. Алексейчук А.Н., Сергиенко Ю.В. «Оценки стойкости и способ реализации кодовой защиты дискретных сообщений с использованием каскадных кодов», *Электронное моделирование*, Т. 25, № 5, 2003, С. 33-44.

91. Алексейчук А.Н. «Случайное кодирование в канале связи с аддитивным шумом, распределенным на конечной абелевой группе», *Захист інформації*, № 3, 2002, С. 7-16.

92. Алексейчук А.Н. «Оптимальное случайное кодирование равновероятных сообщений в q-ичном симметричном канале», *Захист інформації*, № 4, 2002, С. 49-58.

93. Иванов В.А. «Асимптотические характеристики критериев проверки гипотез по случайно преобразованной выборке», *Труды по дискретной математике*, Т. 5, 2002, С. 61-72.

94. Иванов В.А. «Статистические методы оценки эффективности кодового зашумления», *Труды по дискретной математике*, Т. 6, 2002, С. 48-63.

95. Ошкин И.Б., Проскурин Г.В. «Нижние оценки различения подмножеств единичного куба», *Проблемы передачи информации*, Т. 30, В. 3, 1994, С. 15-22.

96. Иванов В.А. «Об эффективности методов случайного кодирования», *Обозрение прикл. промышл. матем.*, Т. 5, В. 2, 1998, С. 220-221.

97. Иванов В.А. «Вероятностные характеристики алгоритмов случайного кодирования», *Обозрение прикл. промышл. матем.*, Т. 8, В. 1, 2001, С. 187-188.

98. Osarov L.H., Wyner A.D. «Wire-tap channel II», *Bell Syst. Techn. J.*, V. 63, 1984, P. 2135-2157.

99. Thangaraj A., Dihidar S., Calderbank A.R., McLaughlin S., Merolla J.-M. «Capacity achieving codes for the wire-tap channel with applications to quantum key distribution», URL: https://arxiv.org/pdf/cs/0411003v1.pdf.

100. Thangaraj A., Dihidar S., Calderbank A.R., McLaughlin S., Merolla J.-M. «On the application of LDPC codes to a novel wire-tap channel inspired by quantum key distribution», URL: https://it.arxiv.org/pdf/cs/0411003v2.pdf.

101. Liu R., Liang Y., Poor V., Spasojevic P. «Secure nested codes for type II wiretap channels», URL: https://arxiv.org/pdf/0706.3752pdf.

102. Яковлев В.А. «Границы для оценки неопределенности в системе передачи со случайным кодированием», *Радиотехника*, № 12, 1996, С. 58-63.

103. Яковлев В.А., Коржик В.И., Синюк А.Д. «Протокол выработки ключа в канале с помехами при ограничении на сложность помехоустойчивого кода», *Материалы тезисов к докладам на межрегиональную конференцию “Информационная безопасность регионов России”*, 1999, С. 106-108.

104. Пазизин С.В. «О характеристиках случайных блоковых кодов, порождающих ошибки», *Обозрение прикл. промышл. матем.*, Т. 7, В. 2, 2000, С. 520-521.

105. Cohen G., Zemor G. «Syndrome-coding for the wire-tap channel revised», *Proc. of IEEE Inform. Theory Workshop*, Chengdu, 2006, P. 33-36.

106. Кузьмин А.С., Нечаев А.А. «Линейно представимые коды и код Кердока над произвольным полем Галуа характеристики 2», *Успехи матем. наук*, Т. 49, № 5, 1994, С. 165-166.

107. Нечаев А.А. «Код Кердока в циклической форме», *Дискретная математика*, Т. 1, В. 4, 1989, С. 123-139.

108. Hammous A.R., Kumar P.V., Calderbrank A.R., Sloane N.J.A., Sole P. «The Z_4-linearity of Kerdock, Preparata, Goethals and related codes», *Bull. Amer. Math. Soc*., V. 29, №. 2, 1993, P. 218-222.

109. Кузьмин А.С., Нечаев А.А. «Построение помехоустойчивых кодов с использованием линейных рекуррент над кольцами Галуа», *Успехи матем. наук*, Т. 47, № 5, 1992, С. 183-184.

110. Мак-Вильямс Ф.Дж., Слоэн Н.Дж.А. *Теория кодов, исправляющих ошибки*, Пер. с англ. – М.: Связь, 1979, 743 с.

111. Berlekamp E.R., McElice R.J., van Tilborg H. «On the inherent intractability of certain coding problems», *IEEE Trans. Inform. Theory*, Vol. 24, No 3, 1978, P. 384-386.

112. Gilbert H., Robshaw M.J.B., Seurin Y. «How to encrypt with the LPN problem», ICALP 2008, Part II, *Lecture Notes in Computer Science*, V. 5126, 2008, P. 679-690.

113. Alekseychuk A. N., Gryshakov S. V. «Nonlinear random coding for information transmission systems with the wire-tap», *Legal, regulatory and metrological support information security system in Ukraine*, V. 8, 2004, P. 133-140, [in Russian].

114. Thangaraj A., Dihidar S., Calderbank A.R., McLaughin S.W., Merolla J.-M. «Applications of LPDC codes to the wiretap channel", *IEEE Trans. Information Theory*, V. 53, No 8, 2007, P. 2933-2945.

РОЗДІЛ 2

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ ОБЧИСЛЮВАЛЬНОЇ СТІЙКОСТІ ТА ПРАКТИЧНОСТІ РПШ МІХАЛЕВИЧА-ІМАЇ

В попередньому розділі показано, що питання про реальну обчислювальну стійкість шифросистем Міхалевича-Імаї [1 – 4] є на сьогодні відкритим, що вимагає проведення подальших наукових досліджень у цьому напрямі.

Даний розділ присвячено докладному аналізу обчислювальної стійкості зазначених шифросистем відносно різноманітних атак (зокрема, таких, що запропоновані вперше). Показано, що стійкість цих шифросистем суттєво залежить від будови їх компонент і може бути значно менше, ніж стверджують їх розробники. На відміну від підходу до аналізу стійкості, що використовується в попередніх роботах [3 – 5], запропоновано простіші аналітичні методи, які дозволяють з’ясувати теоретико-кодовий сенс параметрів, що визначають обчислювальну стійкість цих шифросистем відносно різноманітних атак.

Виходячи з умов стійкості та практичності, вибір компонент для побудови РПШ Міхалевича-Імаї слід здійснювати з урахуванням низки жорстких обмежень, що являє собою нетривіальну задачу. У зв‘язку з цим важливою окремою задачею є з‘ясування потенційних можливостей цих шифросистем та знаходження загальних обмежень, яким задовольняють окремі показники їх ефективності при заданих значеннях інших показників. У розділі отримано аналітичні межі для швидкості передачі інформації в РПШ Міхалевича-Імаї при заданих обмеженнях щодо ймовірності правильного прийому повідомлень законним користувачем та стійкості шифрування. Отримані результати свідчать про обмежені можливості РПШ Міхалевича-Імаї з погляду сучасних вимог щодо стійкості та практичності в реальних умовах.

2.1. Формальне означення та основні показники ефективності РПШ Міхалевича-Імаї

Позначимо V_n множину булевих векторів довжини n, $F_{m\times n}$ – множину $m\times n$-матриць над полем $F = \mathbf{GF}(2)$, $F^*_{m\times m}$ – групу оборотних матриць порядку m над цим полем.

Нагадаємо (див. підрозділ 1.3), що вихідними даними для побудови рандомізованої потокової шифросистеми Міхалевича-Імаї (рис. 2.1) з параметрами $l, m, n \in \mathbf{N}$, $0 < p < 1/2$, де $l < m < n$, та множиною ключів K є такі (несекретні) об‘єкти:

– твірна матриця G_1 двійкового лінійного $[n, m]$-коду C_1 з ефективним алгоритмом декодування (декодером) $D: V_n \to C_1$, який дозволяє надійно виправляти помилки у двійковому симетричному каналі з імовірністю спотворення p;

– матриця $G_2 \in F^*_{m\times m}$;

– генератор гами, що виробляє за ключем $k \in K$ послідовність $f_0(k), f_1(k), \ldots$ двійкових векторів довжини n (при цьому вважається, що функції $f_i : K \to V_n$, $i = 0, 1, \ldots$, можуть залежати від загальнодоступних параметрів, наприклад, векторів ініціалізації).

Для зашифрування на ключі $k \in K$ відкритого тексту $s_0, s_1, \ldots, s_t$, де $s_i \in V_l$, $i = 0, 1, \ldots, t$, відправник генерує послідовність незалежних випадкових векторів $u_0, v_0, u_1, v_1, \ldots, u_t, v_t$, де вектор u_i має рівномірний розподіл на множині V_{m-l}, а координати вектора v_i довжини n розподілені за законом Бернуллі з імовірністю успіху p, та обчислює шифрований текст $z_0, z_1, \ldots, z_t$ за формулою

$$z_i = (s_i, u_i) G_2 G_1 \oplus f_i(k) \oplus v_i,\ i = 0, 1, \ldots, t. \quad (2.1)$$

Законний одержувач, знаючи вектор $f_i(k)$, може швидко відновити повідомлення $(s_i, u_i)G_2$ за допомогою декодера D, а потім знайти вектор s_i, використовуючи оборотність матриці G_2. При цьому супротивник для знаходження ключа k вимушений мати справу зі спотвореною гамою $f_i(k) \oplus (s_i, u_i)G_2G_1 \oplus v_i$, $i = 0, 1, \ldots, t$ (рис. 2.1).

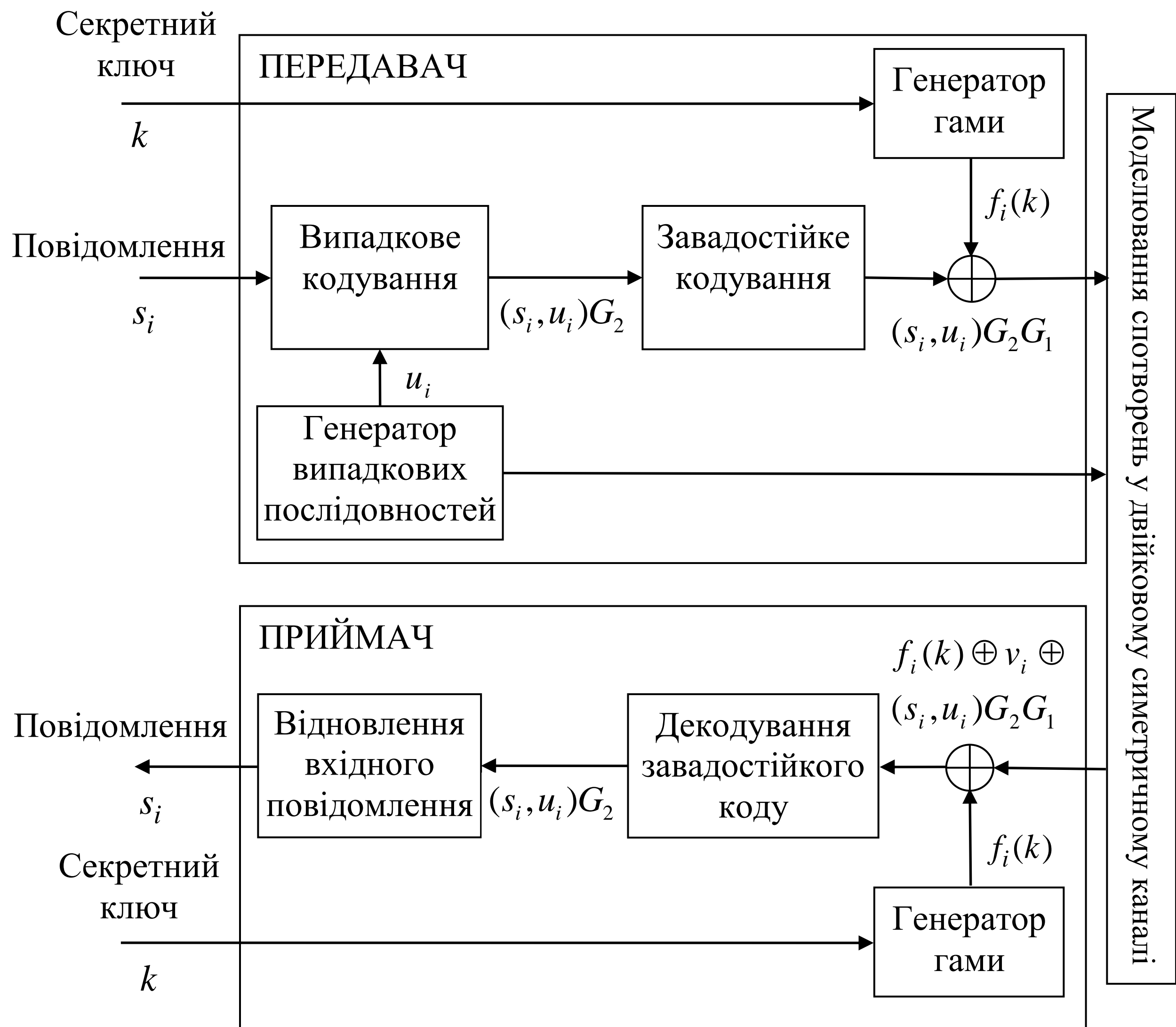


Рис. 2.1. Схема РПШ Міхалевича-Імаї

Зауважимо, що перетворення $s_i \mapsto (s_i, u_i)G_2$ у формулі (2.1) називається випадковим кодуванням повідомлення $s_i \in V_l$, а перетворення

$(s_i, u_i)G_2 \mapsto (s_i, u_i)G_2G_1$ є завадостійке кодування повідомлення $(s_i, u_i)G_2$ кодом C_1.

В [4 – 6] досліджено різні варіанти побудови РПШ наведеного вигляду, зокрема, з такими функціями f_i:

$$f_i(k) = kS^i,\ k \in K = V_n, \tag{2.2}$$

де S є відомою $n \times n$-матрицею над полем F;

$$f_i(k) = \alpha_i k,\ k \in K = F_{n \times n}, \tag{2.3}$$

де $\alpha_0, \alpha_1, \ldots$ є незалежними випадковими рівноймовірними булевими векторами довжини n. Зауважимо, що в останньому випадку шифротекст визначається як послідовність $(\alpha_0, z_0), (\alpha_1, z_1), \ldots, (\alpha_t, z_t)$, де вектори z_i обчислюються за формулою (2.1).

Позначимо $M = M(G_1, G_2, p, D)$ рандомізовану потокову шифросистему Міхалевича-Імаї з параметрами l, m, n, p і декодером D, побудовану за матрицями G_1, G_2, що задовольняють зазначеним вище умовам, а також певним фіксованим генератором гами з множиною ключів K (тут і надалі вважається, що джерело відкритих повідомлень є безнадлишковим, тобто виробляє послідовність $s_0, s_1, \ldots$ незалежних випадкових рівноймовірних векторів довжини l).

Основними показниками ефективності РПШ M є такі параметри:

– швидкість передачі інформації $\rho(M) = l/n$;

– ймовірність $p_e = \mathbf{P}\{D((s_i, u_i)G_2G_1 \oplus v_i) \neq (s_i, u_i)G_2G_1\}$ помилкового декодування повідомлення $z_i \oplus f_i(k)$ законним одержувачем, $i = 0, 1, \ldots$;

– обчислювальна складність декодера D.

Зауважимо, що в силу припущення про безнадлишковість джерела відкритих повідомлень та оборотність матриці G_2 ймовірність p_e є відмінною від нуля і залежить тільки від коду C_1, декодера D і ймовірності p спотворення у ДСК. Крім того, зрозуміло, що $\rho(M) < 1$.

Ефективність шифросистеми M залежить також від складності процедур випадкового і завадостійкого кодування вхідних повідомлень. Для зменшення складності кодування в [5] запропоновано задавати матриці G_1 і G_2 у вигляді

$$G_1 = \begin{pmatrix} I_{m-l} & 0 & A_1 \\ 0 & I_l & A_2 \end{pmatrix}, \ G_2 = \begin{pmatrix} 0 & I_l \\ I_{m-l} & B \end{pmatrix} \tag{2.4}$$

де $A_1 \in F_{(m-l)\times(n-m)}$, $A_2 \in F_{l\times(n-m)}$, $B \in F_{(m-l)\times l}$, а I_l та I_{m-l} – одиничні матриці зазначеного розміру. Такий вибір матриць по суті не звужує клас шифросистем, що розглядаються, однак не є обов‘язковим. В цьому випадку перетворення

$$s \mapsto (s,u)G_2G_1 = (u,\ s \oplus uB,\ sA_2 \oplus u(A_1 \oplus BA_2)),\ s \in V_l,\ u \in V_{m-l}, \tag{2.5}$$

що використовується у формулі (2.1), описує відому схему комбінованого випадкового кодування для каналу зв’язку з відводом (wire-tap channel), запропоновану в [7] та детально досліджену в багатьох інших публікаціях (див. підрозділ 1.3).

2.2. Обчислювальна стійкість РПШ Міхалевича-Імаї

Розглянемо довільну шифросистему $M = M(G_1, G_2, p, D)$. Як і вище, позначимо C_1 лінійний код, породжений рядками матриці G_1.

Покладемо $C_0 = \{(0,u)G_2G_1 : u \in V_{m-l}\}$. Зрозуміло, що C_0 є $[n, m-l]$-підкодом коду C_1 (див., наприклад, [8, 9]). Позначимо $d_0^{\perp}$ та $d_1^{\perp}$ дуальні відстані кодів C_0 та C_1 відповідно, а $C_0^{\perp}$ і $C_1^{\perp}$ – коди, дуальні до C_0 і C_1 відповідно. Зауважимо, що співвідношення $C_0 \subseteq C_1$ тягне таку нерівність:

$$d_0^{\perp} \le d_1^{\perp}. \tag{2.6}$$

Нарешті, для будь-якого $w = 0, 1, \ldots$ позначимо $p_w = 1/2 \cdot \left(1-(1-2p)^{w+1}\right)$.

2.2.1. Обчислювальна стійкість РПШ Міхалевича-Імаї відносно атак на основі відомих шифрованих повідомлень. Розглянемо систему рівнянь (2.1) та припустимо, що супротивник має доступ лише до шифротексту $z_0, z_1, \ldots, z_t$. Перш за все, покажемо, що для відновлення символів відкритого тексту супротивнику не треба мати повну інформацію про секретний ключ.

Твердження 2.1. Нехай H – довільна перевірочна матриця коду C_0. Тоді для будь-яких $i = 0, 1, \ldots$ та $k \in K$ супротивник може відновити (за реальний час) вектор s_i за відомими значеннями z_i та $\varphi_i(k) = f_i(k)H^T$.

Доведення. Нехай $a \in V_n$ – довільний вектор, що задовольняє умові $\varphi_i(k) = aH^T$. Тоді $g = a \oplus f_i(k) \in C_0$ і на підставі формули (2.1) $z_i \oplus a = (s_i, u_i)G_2G_1 \oplus g \oplus v_i$, де слово $(s_i, u_i)G_2G_1 \oplus g$ належить коду C_1. Отже, супротивник може відновити це слово, а також вектор v_i, застосовуючи декодер D коду C_1 до спотвореного слова $z_i \oplus a$. Тепер, знаючи v_i, супротивник може знайти вектор

$$(z_i \oplus a \oplus v_i)H^T = (s_i, u_i)G_2G_1H^T \oplus gH^T = (s_i, u_i)G_2G_1H^T.$$

Позначимо G_1' та G_1'' підматриці матриці G_1, які містяться в її перших $m-l$ та останніх l рядках відповідно. На підставі формули (2.4) отримаємо, що

$$(s_i, u_i)G_2G_1 = (u_i, s_i \oplus u_i B)G_1 = u_i G_1' \oplus (s_i \oplus u_i B)G_1''.$$

Далі, оскільки H є перевірочною матрицею коду $C_0 = \{uG_1' \oplus uBG_1'' : u \in V_{m-l}\}$, виконується рівність $G_1'H^T = BG_1''H^T$. Отже,

$$(s_i, u_i)G_2G_1H^T = (u_i G_1' \oplus (s_i \oplus u_i B)G_1'')H^T = s_i G_1'' H^T.$$

Таким чином, за відомим вектором $(s_i, u_i)G_2G_1H^T$ супротивник може знайти вектор s_i, розв'язуючи систему лінійних рівнянь $x\,G_1''H^T = s_i G_1'' H^T$ відносно невідомого $x \in V_l$.

Помітимо зараз, що ця система має єдиний розв'язок (який дорівнює s_i). Дійсно, в протилежному випадку існує ненульовий вектор $x \in V_l$ такий, що $x\,G_1''H^T = 0$. Але тоді $x\,G_1'' \in C_0$ і, отже, існує вектор $u \in V_{m-l}$ такий, що $x\,G_1'' = uG_1' \oplus uBG_1''$. Звідси випливає, що $u = 0$, $uB \oplus x = 0$, оскільки рядки матриці G_1 є лінійно незалежними (див. співвідношення (2.4)). Отже, $x = 0$, що суперечить зробленому вище припущенню.

Таким чином, отримаємо такий алгоритм відновлення вектора s_i за відомими значеннями z_i та $\varphi_i(k) = f_i(k)H^T$:

1. Знайти довільний вектор $a \in V_n$ такий, що $\varphi_i(k) = aH^T$ (наприклад, за допомогою алгоритму Гаусса).

2. Відновити вектор v_i, застосовуючи декодер D коду C_1 до спотвореного кодового слова $z_i \oplus a$.

3. Відновити вектор s_i як єдиний розв’язок $x \in V_l$ системи лінійних рівнянь $x\,G_1'' H^T = (z_i \oplus a \oplus v_i) H^T$.

Твердження доведено.

Покажемо зараз, що відновлення ключа за відомим шифрованим текстом можна звести до розв’язання певної системи булевих лінійних рівнянь зі спотвореними правими частинами, де ймовірності спотворень дорівнюють $p_{d_1^{\perp}-1}$.

Твердження 2.2. За умови (2.1) складність відновлення ключа шифросистеми при проведенні атаки на основі шифрованого повідомлення обмежена зверху складністю розв’язання системи рівнянь

$$z_i h^T = f_i(k) h^T \oplus \xi_i\,,\ i = 0, 1, \ldots, t\,, \tag{2.7}$$

де $h \in C_1^{\perp}$ – довільне слово ваги $d_1^{\perp}$, $\xi_0, \xi_1, \ldots, \xi_t$ – незалежні випадкові величини, розподілені за законом

$$\mathbf{P}(\xi_i = 1) = 1 - \mathbf{P}(\xi_i = 0) = p_{d_1^{\perp}-1},\ i = 0, 1, \ldots, t\,. \tag{2.8}$$

Доведення. Оскільки $h \in C_1^{\perp}$, то $(s_i, u_i) G_2 G_1 h^T = 0$ для будь-яких $s_i \in V_l$, $u_i \in V_{m-l}$. Звідси на підставі формули (2.1) випливає, що $z_i h^T = f_i(k) h^T \oplus v_i h^T$. При цьому, оскільки координати випадкового вектора v_i є незалежними та приймають значення 0 і 1 з ймовірностями $1-p$ і p відповідно, то $\mathbf{P}(v_i h^T = 1) = 1 - \mathbf{P}(v_i h^T = 0) = p_{wt(h)-1}$ (див, наприклад, [9], лема 9.49). Отже, твердження доведено.

Зауважимо, що в деяких випадках секретний ключ k можна однозначно відновити з системи рівнянь (2.7) з меншою складністю в порівнянні зі

складністю розв'язання задачі LPN з тими самими числами невідомих і ймовірністю спотворень вигляду (2.8). Наприклад, якщо функції f_i визначаються за формулою (2.2), то складність розв'язання системи рівнянь (2.7) залежить від алгебраїчних властивостей набору векторів $S^i h^T$, $i = 0, 1, \ldots, t$. Якщо для цих векторів існує достатньо багато перевірочних співвідношень малої ваги, то зазначену систему рівнянь можна ефективно розв'язати за допомогою відомих алгоритмів, які використовуються при побудові швидких кореляційних атак (див., наприклад, [10]).

2.2.2. Обчислювальна стійкість РПШ Міхалевича-Імаї відносно атаки на основі підібраних відкритих повідомлень. Розглянемо зараз атаку, описану в [4, 5], де припускається, що супротивник може зашифровувати на тому ж самому (невідомому) ключі повідомлення $s_i = 0$, отримуючи при цьому повідомлення z_i вигляду (2.1), $i = 0, 1, \ldots, t$. Зрозуміло, що міркування, використані вище для побудови атаки на основі шифрованого повідомлення, є застосовними і в цьому випадку, лише з тією відмінністю, що замість коду C_1 треба використовувати його підкод C_0.

Зокрема, справедливе таке твердження.

Твердження 2.3. За умов (2.1), (2.2) складність відновлення ключа шифросистеми при проведенні атаки на основі підібраних відкритих повідомлень обмежена зверху складністю розв'язання системи рівнянь

$$z_i h^T = k(S^i h^T) \oplus \xi_i,\ i = 0, 1, \ldots, t, \qquad (2.9)$$

де $h \in C_0^{\perp}$ – довільне слово ваги $d_0^{\perp}$, $\xi_0, \xi_1, \ldots, \xi_t$ – незалежні випадкові величини, розподілені за законом

$$\mathbf{P}(\xi_i = 1) = 1 - \mathbf{P}(\xi_i = 0) = p_{d_0^{\perp} - 1},\ i = 0, 1, \ldots, t. \qquad (2.10)$$

Зауважимо, що в роботі [5] для відновлення ключа за допомогою зазначеної атаки також формується певна система булевих лінійних рівнянь зі спотвореними правими частинами. Ця система рівнянь відрізняється від системи (2.9) і має більш складний вигляд. Більш того, ймовірності спотворень у її правій частині є не менше (але можуть бути й більше) ніж значення (2.10) (див. [5], с. 11, 15).

Таким чином, згідно з твердженням 2.3 стійкість РПШ Міхалевича-Імаї відносно атаки на основі підібраних відкритих повідомлень визначається дульною відстанню коду C_0 і може бути значно менше, ніж стверджується в [3 – 5]. Наприклад, якщо функції f_i визначаються за формулою (2.2), то складність розв’язання системи рівнянь (2.9) залежить від алгебраїчних властивостей набору векторів $S^i h^T$, $i = 0, 1, \ldots, t$. Якщо для цих векторів існує достатньо багато перевірочних співвідношень малої ваги, то зазначену систему рівнянь можна ефективно розв’язати за допомогою відомих алгоритмів, які використовуються при побудові швидких кореляційних атак (див., наприклад, [10]).

2.2.3. Обчислювальна стійкість РПШ відносно атаки на основі підібраних векторів ініціалізації. У випадку, коли супротивник має доступ до шифрувального пристрою та може на свій розсуд вибирати загальнодоступні параметри, наприклад, вектори ініціалізації, від яких залежать функції f_i, він може провести на шифросистему більш потужну атаку, виконуючи такий алгоритм.

Алгоритм 2.1.

1. Вибрати слово $h \in C_0^{\perp} \setminus \{0\}$ ваги $d_0^{\perp}$.

2. Подати t разів на вхід шифросистеми з невідомим фіксованим ключем k повідомлення $s_0 = 0$ і знайти (для вибраного $i = 0, 1, \ldots$) шифровані повідомлення

$$z^{(j)} = (0, u^{(j)}) G_2 G_1 \oplus f_i(k) \oplus v^{(j)},\ j \in \overline{1,t}, \tag{2.11}$$

де $u^{(0)}, v^{(0)}, u^{(1)}, v^{(1)}, \ldots$ – незалежні випадкові вектори, розподілені за законами

$\mathbf{P}\{u^{(j)} = u\} = 2^{-(m-l)}$, $\mathbf{P}\{v^{(j)} = v\} = p^{wt(v)}(1-p)^{n-wt(v)}$, $u \in V_{m-l}$, $v \in V_n$ (тут і до

кінця даного розділу $wt(v)$ позначає вагу Гемінга довільного вектора v);

3. Обчислити

$$z^{(j)} h^T = f_i(k) h^T \oplus v^{(j)} h^T,\ j \in \overline{1,t}, \tag{2.12}$$

і відновити значення $f_i(k) h^T$ методом максимуму правдоподібності.

Зауважимо, що таке багатократне зашифровування не надає додаткової інформації про ключ, якщо використовується звичайний (нерандомізований) шифр. Проте у випадку рандомізованої шифросистеми наведена атака виявляється найбільш потужною і у багатьох випадках може бути реалізована за реальний час.

Має місце наступне твердження.

Твердження 2.4. Нехай H – довільна перевірочна матриця коду C_0, $d(H) = \max\limits_{1 \le r \le n-m+l} wt(H_r)$, де H_r – r-й рядок матриці H. Тоді для будь-яких $i = 0, 1, \ldots,$ $k \in K$ та $0 < \delta < 1$ супротивник може відновити значення $\varphi_i(k) = f_i(k) H^T$ з імовірністю не менше $1 - \delta$ за $O(nt \log t)$ двійкових операцій, використовуючи

$$t = \left\lceil 1/2 \cdot (1 - 2p)^{-2d(H)} \ln(\delta^{-1}(n - m + l)) \right\rceil \tag{2.13}$$

довільних рівнянь системи (2.11).

Доведення. На підставі формули (2.11) для будь-якого $r = 1, 2, \ldots, n-m+l$ справедливі такі рівності:

$$z^{(j)} H_r^T = f_i(k) H_r^T \oplus \xi_{j,r}, \; j = 0, 1, \ldots,$$

де $\xi_{j,r}$ є незалежними випадковими величинами, розподіленими за законом

$$\mathbf{P}(\xi_{j,r} = 1) = 1 - \mathbf{P}(\xi_{j,r} = 0) = 1/2 \cdot \left(1 - (1-2p)^{wt(H_r)}\right), \; j = 0, 1, \ldots.$$

Нехай для відновлення значення $f_i(k) H_r^T$ використовується мажоритарне правило, тобто $f_i(k) H_r^T$ покладається рівним 0 тоді й тільки тоді, коли $\sum_{j=1}^{t} z^{(j)} H_r^T < t/2$. В цьому випадку, спираючись на нерівність Чернова (див., наприклад, [11], с. 300), можна оцінити ймовірність помилки таким чином:

$$\mathbf{P}\left(\sum_{j=1}^{t} \xi_{j,r} \geq t/2 \right) = \mathbf{P}\left(t^{-1} \sum_{j=1}^{t} \xi_{j,r} - 1/2 \cdot \left(1 - (1-2p)^{wt(H_r)}\right) \geq (1-2p)^{wt(H_r)} \right) \leq$$

$$\leq \exp\{-2t(1-2p)^{2wt(H_r)}\} \leq \exp\{-2t(1-2p)^{2d(H)}\}.$$

Отже, ймовірність того, що всі значення $f_i(k) H_r^T$, $r = 1, 2, \ldots, n-m+l$, будуть відновлені коректно, обмежена знизу числом $1 - (n-m+l)\exp\{-2t(1-2p)^{2d(H)}\} \geq 1 - \delta$, де остання нерівність випливає з означення параметра t. Нарешті, зрозуміло, що складність відновлення усіх

значень $f_i(k)H_r^T$, $r = 1, 2, ..., n - m + l$, з використанням мажоритарного правила складає $O(nt \log t)$ двійкових операції.

Твердження доведено.

На рис. 2.2 наведені залежності двійкового логарифму обчислювальної складності T_1 описаної атаки на РПШ Міхалевича-Імаї, побудовані на базі наступних лінійних $[n, m-l, d_0^{\perp}]$ кодів C_0: $[255, 205, 74]$, $[255, 185, 62]$ і $[255, 135, 40]$ [12]. Для всіх трьох випадків ймовірність успіху атаки дорівнює 0,95 ($\delta = 0{,}05$), параметр $d(H) = d_0^{\perp}$.

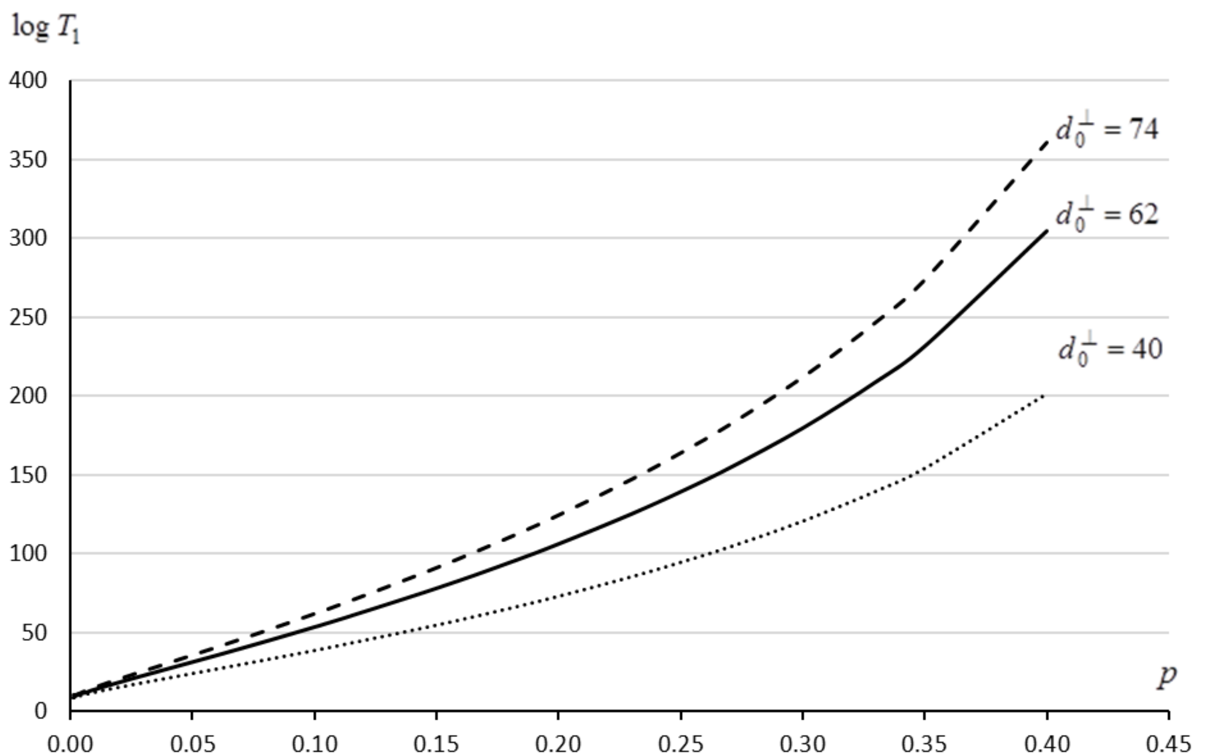


Рис. 2.2. Верхні оцінки обчислювальної складності атаки на РПШ Міхалевича-Імаї на основі підібраних векторів ініціалізації

Як видно з рисунку, складність відновлення вектора $\varphi_i(k) = f_i(k)H^T$ майже пропорційна кількості t рівнянь у системі зі спотвореними правими

частинами, де ймовірність спотворення визначається максимальною вагою стовпців перевірочної матриці коду C_0. Зауважимо, що, знаючи вектор $\varphi_i(k)$, супротивник може використати його для проведення атаки на основі відомих шифрованих повідомлень як зазначено в п. 2.2.1.

Отримаємо зараз нижню межу *інформаційної складності* наведеної атаки, тобто мінімального обсягу матеріалу, необхідного для успішного відновлення значення $f_i(k)h^T$ за набором (2.12), де h є довільним ненульовим словом коду $C_0^{\perp}$.

Твердження 2.5. Для відновлення значення $f_i(k)h^T$ з системи рівнянь (2.12) з імовірністю $1/2+\theta$, $0<\theta<1/2$, необхідно не менше

$$t_\theta = 1/4\cdot\theta^2(1-2p)^{-2d_0^{\perp}} \tag{2.14}$$

рівнянь.

Доведення. Покладемо $a = f_i(k)h^T$, $\xi_j = z^{(j)}h^T = a\oplus v^{(j)}h^T$, $j\in\overline{1,t}$. На підставі означення вектора h та випадкових векторів $u^{(0)}, v^{(0)}, u^{(1)}, v^{(1)}, \ldots$ послідовність $\xi = \xi_1, \xi_2, \ldots, \xi_t$ є схемою Бернуллі з параметрами (t, p_a), де $p_0 = 1/2\cdot\left(1-(1-2p)^{d_0^{\perp}}\right)$, $p_1 = 1-p_0$. При цьому задача відновлення числа a за набором його спотворених значень (2.12) рівносильна перевірці наступних двох гіпотез:

$$H_0\colon \mathbf{P}\{\xi_1=1\}=p_0;\ H_1\colon \mathbf{P}\{\xi_1=1\}=p_1.$$

Для будь-якого $v\in V_t$ позначимо $\mathbf{P}_0(v)=\mathbf{P}\{\xi=v\mid H_0\}$, $\mathbf{P}_1(v)=\mathbf{P}\{\xi=v\mid H_1\}$. Розглянемо довільний критерій для перевірки гіпотез H_0 і H_1, який базується на критичній множині A. Нагадаємо, що ймовірності помилок першого і другого роду вказаного критерію визначаються за формулами $\alpha=\sum_{v\in A}\mathbf{P}_0(v)$ і

$\beta = 1 - \sum_{v \in A} \mathbf{P}_1(v)$ відповідно. Для доведення твердження необхідно переконатися у справедливості такого співвідношення:

$$1/2 \cdot (\alpha + \beta) \le 1/2 - \theta \;\Rightarrow\; t \ge t_\theta .$$

Скористаємось лемою 15 в [13], згідно з якою

$$\max_{A \subseteq V_t} \left| \sum_{v \in A} (q^{wt(v)} (1-q)^{t - wt(v)} - 2^{-t}) \right| \le 2\sqrt{t}\, |2q - 1|, \; 0 \le q \le 1 . \tag{2.15}$$

Справедливі співвідношення

$$2\theta \le 1 - (\alpha + \beta) = \sum_{v \in A} (\mathbf{P}_1(v) - \mathbf{P}_0(v)) \le \left| \sum_{v \in A} (\mathbf{P}_0(v) - 2^{-t}) \right| + \left| \sum_{v \in A} (\mathbf{P}_1(v) - 2^{-t}) \right|,$$

$$\mathbf{P}_0(v) = p_0{}^{wt(v)} (1 - p_0)^{t - wt(v)}, \; \mathbf{P}_1(v) = p_1{}^{wt(v)} (1 - p_1)^{t - wt(v)},$$

з яких на підставі формули (2.15) і рівності $p_1 = 1 - p_0$ випливає, що

$$2\theta \le 2\sqrt{t}\, |2p_0 - 1| + 2\sqrt{t}\, |2p_1 - 1| = 4\sqrt{t}(1 - 2p_0) .$$

Отже, $t \ge 1/4 \cdot \theta^2 (1 - 2p_0)^{-2} = t_\theta$, що і треба було довести.

На рис. 2.3 зображені графіки верхньої та нижньої меж параметра $\log t$ як функцій ймовірності спотворення в ДСК, розраховані за формулами (2.13) (при $n - m + l = 1$, $d(H) = d_0^{\perp}$) та (2.14) відповідно. Ймовірність успіху атаки вважається рівною 0,95 ($\theta = 0{,}45$, $\delta = 0{,}05$). Побудовані криві відповідають

шифросистемам, побудованим на базі наступних лінійних $[n, m-l, d_0^{\perp}]$ кодів C_0: $[255, 205, 74]$, $[255, 185, 62]$ і $[255, 135, 40]$ [12].

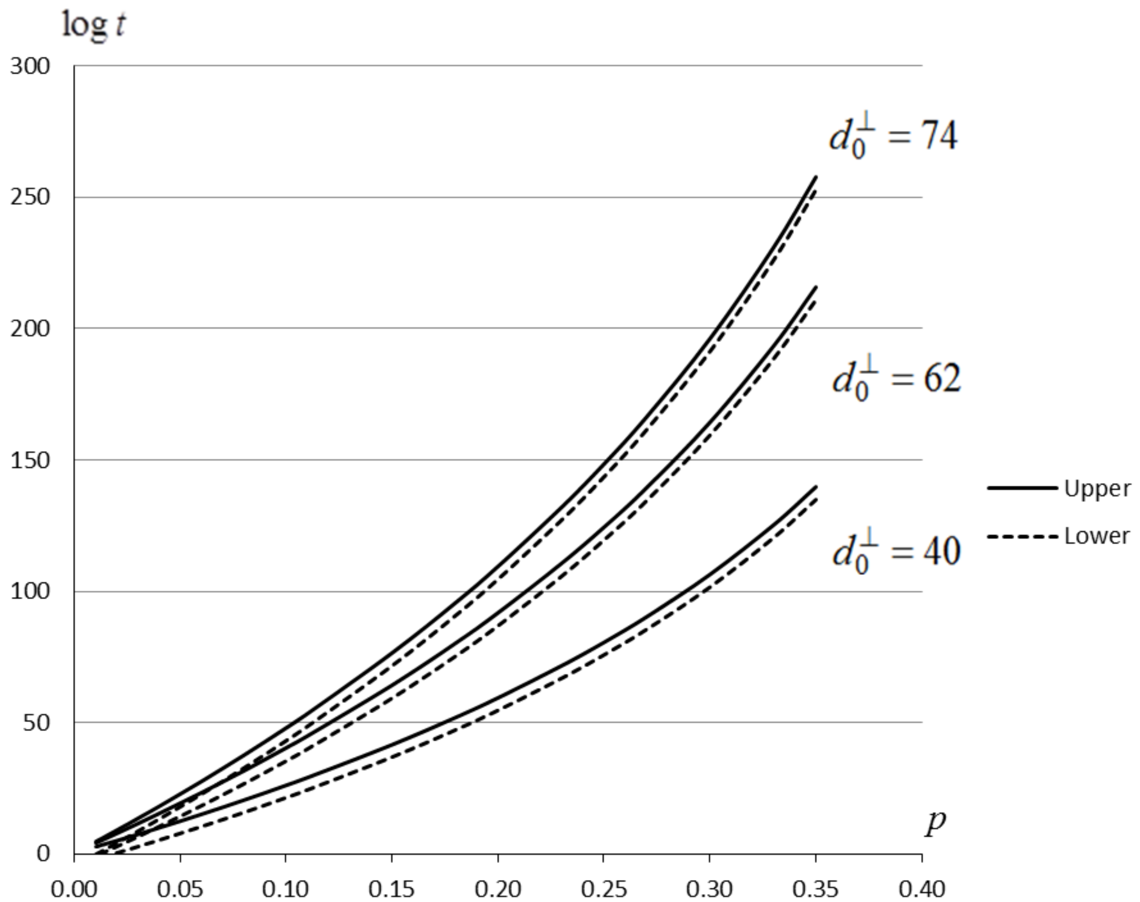


Рис. 2.3. Оцінки обсягу матеріалу, необхідного для успішного відновлення одного біта інформації про ключ РПШ Міхалевича-Імаї

Як видно із залежностей на рисунку, інформаційна складність атаки зростає як зі збільшенням ймовірності p спотворення в ДСК, так і зі збільшенням дуальної відстані $d_0^{\perp}$ коду. Так, у випадку $p = 0{,}10$ параметр t знаходиться у межах від 2^{21} до 2^{26} при $d_0^{\perp} = 40$ та у межах від 2^{43} до 2^{48} при $d_0^{\perp} = 74$. Зі збільшенням ймовірності спотворень до $p = 0{,}20$ нижня та верхня

межі інформаційної складності атаки становлять 2^{55} і 2^{59} при $d_0^{\perp} = 40$ та 2^{105} і 2^{110} при $d_0^{\perp} = 74$.

Зауважимо, що атаки на основі підібраних векторів ініціалізації не розглядаються в [2, 3, 5], проте їх необхідно враховувати, оцінюючи стійкість рандомізованих потокових шифросистем, побудованих на основі реальних генераторів гами, що використовуються в сучасних потокових шифрах.

В цілому, результати проведених досліджень свідчать про те, що обчислювальна стійкість РПШ Міхалевича-Імаї суттєво залежить від вибору матриць G_1, G_2 та функцій f_i, що визначаються відповідним генератором гами, і може бути помітно менше, ніж стверджується в [3 – 5]. Зокрема, деякі з зазначених РПШ є вразливими навіть до атак на основі відомих шифрованих повідомлень. Таким чином, для побудови обґрунтовано стійких РПШ Міхалевича-Імаї потрібно дуже ретельно вибирати матриці G_1 та G_2, що виявляється непростою задачею.

2.3. Аналітичні межі для швидкості передачі інформації в РПШ Міхалевича-Імаї при заданих обмеженнях відносно стійкості та ймовірності правильного прийому повідомлень законним користувачем

Нагадаємо, що швидкістю передачі двійкового лінійного $[n, k]$-коду називається число k/n. Для будь-якого натурального $n > 1$ і $0 < \delta < 1$ позначимо $R_n(\delta)$ найбільшу швидкість передачі двійкових лінійних кодів довжини n з мінімальною відстанню $d \geq \delta n$. В теорії кодування відомо низку верхніх меж параметру $R_n(\delta)$ (як точних, так і асимптотичних при $n \to \infty$, $\delta = \mathrm{const}$) [8, 11, 12]. Наведемо тут дві такі межі, необхідні в подальшому.

Лема 2.1. Для будь-якого натурального $n > 1$ справедливі нерівності

$$R_n(\delta) \le -1/n \cdot \log\left(1-(2\delta)^{-1}\right), \tag{2.16}$$

якщо $1/2 < \delta < 1$;

$$R_n(\delta) \le 1 - H_2\left(1/2 \cdot (1-\sqrt{1-2\delta+2/n}) - 1/n\right) + \log(n\sqrt{n})/n, \tag{2.17}$$

якщо $1/n < \delta \le 1/2$, де $H_2(x) = -x\log x - (1-x)\log(1-x)$, $0 < x < 1$.

Доведення. Формула (2.16) витікає з відомої межі Плоткіна для мінімальної відстані d двійкового лінійного $[n, k]$-коду (див., наприклад, [12], с. 40):

$$d \le n/2 \cdot \frac{2^k}{2^k - 1},$$

а формула (2.17) – з межі Бассалиго-Елайєса [11], с. 270: якщо $2(d-1) \le n$, то

$$k/n \le 1/n \cdot \log\left(d2^n \left(\sum_{i=0}^{c} \binom{n}{i}\right)^{-1}\right),$$

де

$$c = \left\lfloor n/2 \cdot \left(1 - \sqrt{1 - \frac{2(d-1)}{n}}\right)\right\rfloor,$$

і оцінок для суми біноміальних коефіцієнтів [8], с.302:

$$\sum_{i=0}^{c} \binom{n}{i} \ge \binom{n}{c} \ge \frac{2^{nH_2(c/n)}}{\sqrt{8n\,(1-c/n)\,c/n}} \ge \frac{2^{nH_2(c/n)}}{2\sqrt{n}}.$$

Лему доведено.

Наступне допоміжне твердження є окремим випадком відомого результату про зв‘язок між швидкістю передачі, пропускною здатністю каналу і ймовірністю помилкового декодування [14], с. 223.

Лема 2.2. Нехай C_1 – двійковий лінійний $[n, m]$-код, що використовується для передачі випадкових рівноймовірних повідомлень у ДСК з імовірністю спотворення $0 < p < 1/2$, D – довільний декодер коду C_1 з імовірністю помилкового декодування p_e такий, що $p_e + H_2(p_e) < 1$. Тоді

$$m/n \le \frac{1 - H_2(p)}{1 - p_e - H_2(p_e)}.$$

Отримаємо зараз верхні межі швидкості передачі інформації у шифросистемах Міхалевича-Імаї.

Твердження 2.6. Нехай $M = M(G_1, G_2, p, D)$ є РПШ Міхалевича-Імаї з параметрами l, m, n, p така, що $t_\theta \ge t \ge 1$, де $0 < \theta < 1/2$ і t_θ визначається за формулою (2.14). Тоді

$$\lambda_\theta(t, p) \overset{\text{def}}{=} -\frac{\log(4\theta^{-2} t)}{2n \log(1 - 2p)} \in (0, 1) \tag{2.18}$$

та

$$\rho(M) \le m/n - (1 - R_n(\lambda_\theta(t, p)). \tag{2.19}$$

Зокрема, якщо ймовірність p_e помилкового декодування повідомлень декодером D така, що $p_e + H_2(p_e) < 1$, то

$$\rho(M) \le \frac{1 - H_2(p)}{1 - p_e - H_2(p_e)} - 1 - 1/n \cdot \log\left(1 - (2\lambda_\theta(t, p))^{-1}\right), \tag{2.20}$$

якщо $\lambda_\theta(t, p) > 1/2$;

$$\rho(M) \le \frac{1 - H_2(p)}{1 - p_e - H_2(p_e)} -$$

$$- H_2\left(1/2 \cdot (1 - \sqrt{1 - 2\lambda_\theta(t,p) + 2/n}) - 1/n\right) + \log(n\sqrt{n})/n, \qquad (2.21)$$

якщо $1/n < \lambda_\theta(t,p) \le 1/2$.

Доведення. Розглянемо код $C_0 = \{(0,u)G_2G_1 : u \in V_{m-l}\}$, швидкість передачі якого дорівнює $1 - (m-l)/n$. З умови $t_\theta \ge t$ і формули (2.14) випливає, що мінімальна відстань цього коду задовольняє нерівності $d_0^\perp \ge n\lambda_\theta(t,p)$. Оскільки при цьому $t \ge 1$ і $d_0^\perp < n$ (в протилежному випадку виконується рівність $n - (m-l) = 1$, яка суперечить умові $m < n$), то є справедливим співвідношення (2.18).

Далі, згідно з означенням функції R_n, швидкість передачі коду C_0 не перевищує числа $R_n(\lambda_\theta(t,p))$, звідки безпосередньо випливає нерівність (2.19). Нарешті, формули (2.20), (2.21) випливають з формул (2.16), (2.17) і леми 2.2.

Твердження доведено.

Зауважимо, що нерівності (2.19) – (2.21) є справедливими для будь-яких РПШ Міхалевича-Імаї, стійкість яких відносно атаки, описаної в п. 2.2.3, складає не менше ніж t операцій (зашифровування відкритого повідомлення $s_0 = 0$), а ймовірність правильного прийому повідомлень законним користувачем обмежена знизу числом $1 - p_e$, де $p_e + H_2(p_e) < 1$.

На рис. 2.4 зображені графіки верхніх меж параметра $\rho(M)$ (як функцій параметра p), побудовані з використанням співвідношень (2.20), (2.21) при $n = 512$, $p_e = 10^{-8}$, $\theta = 0{,}45$ і $t \in \{2^{20}, 2^{40}, 2^{60}\}$. Як видно із залежностей на рисунку, при $t = 2^{20}$ максимальна швидкість передачі інформації не перевищує $0{,}36$ а при $t = 2^{60}$ – дорівнює $0{,}08$. Таким чином, для забезпечення стійкості

порядку 2^{60} операцій необхідно вибирати довжину відкритих повідомлень $l \le \lfloor 0{,}08n \rfloor = 40$ бітів (якими б не були інші компоненти шифросистеми).

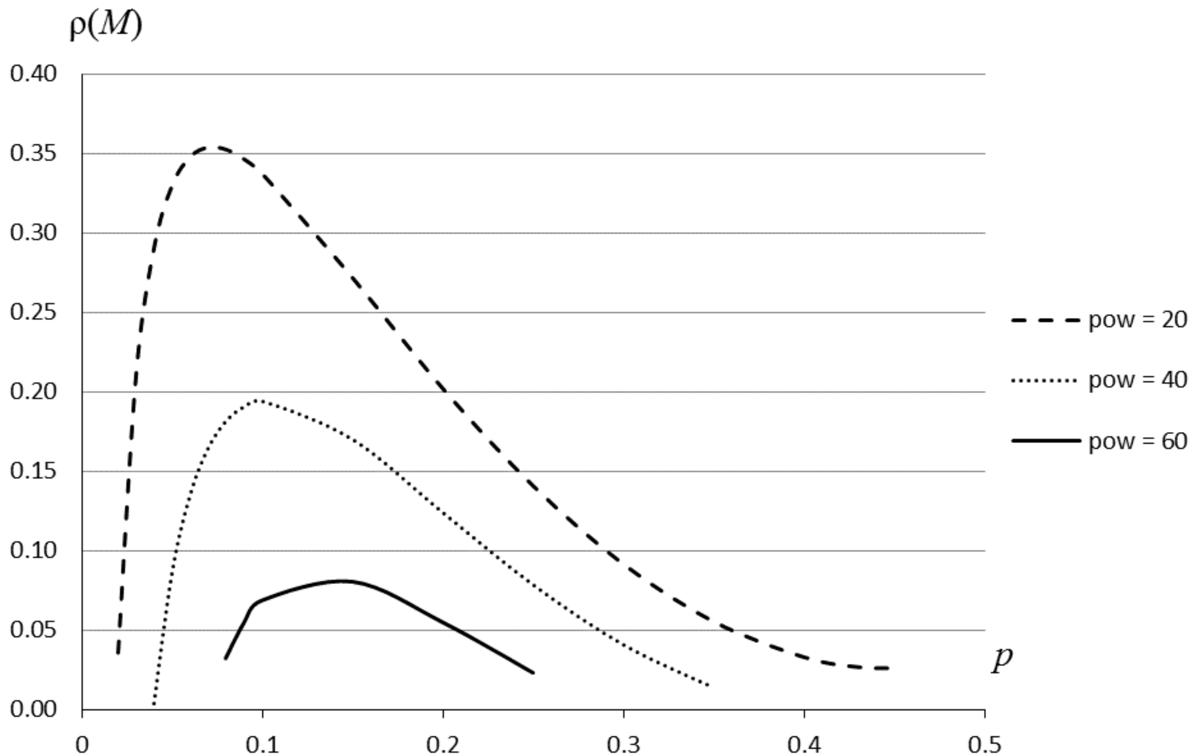


Рис. 2.4. Залежності верхніх оцінок швидкості передачі інформації у РПШ Міхалевича-Імаї від імовірності спотворення у ДСК

Зауважимо також, що при $t > 2^{79}$ і зазначених вище значеннях n, p_e, θ вирази у правих частинах нерівностей (2.20), (2.21) є від‘ємними для усіх p, які задовольняють умові (2.18), що свідчить про відсутність РПШ Міхалевича-Імаї, які володіють зазначеною стійкістю. Крім того, при передачі повідомлень зі швидкістю 0,5 (або вище) максимальне значення стійкості шифросистеми не перевищує $t = 2^{8,37}$ операцій, що свідчить про її вразливість до атаки з п. 2.2.3.

На завершення доведемо твердження, яке встановлює нижню межу для швидкості передачі, за якої існують РПШ Міхалевича-Імаї з заданою стійкістю.

Цей результат є аналогічним відомій межі Варшамова-Гілберта (див., наприклад, [12], с. 44).

Твердження 2.7. Нехай C_1 – двійковий лінійний $[n,m]$-код з твірною матрицею G_1 і дуальною відстанню $d_1^{\perp} \le n/2$; $0 < p < 1/2$, $t \ge 1$, $0 < \theta < 1/2$ і $l \in \mathbf{N}$ такі, що

$$l/n < m/n - H_2(\lambda_\theta(t,p)), \tag{2.22}$$

де

$$\lambda_\theta(t,p) = -\frac{\log(4\theta^{-2}t)}{2n\log(1-2p)} < d_1^{\perp} n^{-1}. \tag{2.23}$$

Тоді існує матриця G_2 вигляду (2.4) така, що $\rho\big(M(G_1,G_2,p,D)\big) \ge l/n$ і $t_\theta \ge t$ (яким би не був декодер $D: V_n \to C_1$).

Доведення. Достатньо переконатися в тому, що при випадковому рівноймовірному виборі матриці B дуальна відстань коду $C_0 = \{(0,u)G_2G_1 :\ u \in V_{m-l}\}$, що відповідає випадковій матриці G_2 виду (2.4), задовольняє умові $d_0^{\perp} \ge n\lambda_\theta(t,p)$ з додатною ймовірністю.

Запишемо матрицю G_1 у вигляді $G_1 = \begin{pmatrix} G_1' \\ G_1'' \end{pmatrix}$, де $G_1' \in F_{(m-l)\times n}$, $G_1'' \in F_{l\times n}$. За означенням код C_0 складається з усіх слів вигляду $(0,u)G_2G_1 = (u,uB)\begin{pmatrix} G_1' \\ G_1'' \end{pmatrix} = uG_1' \oplus uBG_1''$, де $u \in V_{m-l}$.

Припустимо, що $d_0^{\perp} < n\lambda_\theta(t,p)$; тоді існує ненульовий вектор $x \in V_n$ ваги $wt(x) \le \lfloor n\lambda_\theta(t,p) \rfloor$, ортогональний коду C_0, тобто такий, що задовольняє умові

$$B(G_1''x) = G_1'x. \tag{2.24}$$

Помітимо, що $G_1''x \neq 0$, оскільки в протилежному випадку $G_1''x = 0$, $G_1'x = 0$ і, значить, $x \in C_1^{\perp} \setminus \{0\}$, звідки на підставі формули (2.23) витікає, що $wt(x) \geq d_1^{\perp} > n\lambda_{\theta}(t, p)$. Відповідно, для будь-якого фіксованого $x \in V_n \setminus \{0\}$ ймовірність події (2.24) дорівнює $2^{-(m-l)}$. Звідси, використовуючи нерівність Чернова (див., наприклад, [11], с. 300) і формулу (2.22), отримаємо, що

$$\mathbf{P}\{d_0^{\perp} < n\lambda_{\theta}(t,p)\} \leq \sum_{x \in V_n : 1 \leq wt(x) \leq \lfloor n\lambda_{\theta}(t,p) \rfloor} 2^{-(m-l)} =$$

$$= 2^{-(m-l)} \sum_{i=1}^{\lfloor n\lambda_{\theta}(t,p) \rfloor} \binom{n}{i} \leq 2^{-(m-l)} 2^{nH_2(n\lambda_{\theta}(t,p))} < 1.$$

Таким чином, справедлива нерівність $\mathbf{P}\{d_0^{\perp} \geq n\lambda_{\theta}(t,p)\} > 0$, що й треба було довести.

Зауважимо, що твердження 2.7, як і межа Варшамова-Гілберта, містить лише достатні умови існування шуканих об‘єктів (шифросистем Міхалевича-Імаї з заданими стійкістю та швидкістю передачі) без зазначення ефективного способу їх побудови.

Висновки

1. Обчислювальна стійкість РПШ Міхалевича-Імаї (як відносно відомої атаки [4, 5], так і нових атак, викладених у розділі) суттєво залежить від вибору матриць G_1, G_2 та функцій f_i, які визначаються відповідним генератором гами, і може бути помітно менше, ніж стверджується в [3 – 5]. Зокрема, деякі з зазначених РПШ є вразливими навіть до атак на основі відомих шифрованих повідомлень.

2. Вплив генератора гами на стійкість РПШ Міхалевича-Імаї проявляється в тому, що системи булевих лінійних рівнянь зі спотвореними правими частинами, до розв'язання яких зводиться відновлення ключів, можуть мати дуже спеціальний вигляд та розв'язуватися суттєво швидше, ніж системи загального вигляду з тими самими числами невідомих та ймовірністю спотворень. Останній параметр залежить від дуальної відстані коду C_0, належний вибір якого (для заданого коду C_1), з урахуванням обмеження (2.6), є нетривіальною задачею. (Підкреслимо, що критерії вибору матриці G_2, наведені в [5], с. 11 та с. 15, не гарантують заявленого рівня стійкості).

3. Суттєва слабкість класу РПШ Міхалевича-Імаї в цілому полягає в зменшенні кількості інформації (в порівнянні з довжиною блоку шифрувальної гами), що необхідна для відновлення за реальний час символів відкритого тексту (див. твердження 2.1). Зазначена властивість є наслідком спільного застосування випадкового і завадостійкого кодування повідомлень лінійними кодами та непритаманна аналогічним за будовою шифросистемам [15], де випадкове кодування не використовується.

4. Розроблено атаку на РПШ Міхалевича-Імаї на основі підібраних векторів ініціалізації, що має обчислювальну складність, яка залежить лінійно від довжини кодового слова та субквадратично від обсягу матеріалу, що використовується (див. твердження 2.4). Для шифросистем, побудованих на базі лінійних кодів C_0 з параметрами $n = 255$, $m - l = 185$, $d_0^{\perp} = 62$ складність атаки (з імовірністю успіху $0{,}95$) не перевищує $2^{18{,}86}$ двійкових операцій при $p = 0{,}02$ та $2^{53{,}84}$ двійкових операцій при $p = 0{,}10$.

5. Отримані аналітичні межі для швидкості передачі інформації в РПШ Міхалевича-Імаї дозволяють з'ясувати їх потенційні можливості та визначити загальні обмеження, яким задовольняють окремі показники їх ефективності при заданих значеннях інших показників. Застосування отриманих меж до РПШ з параметром $n = 512$ показує, що їх стійкість не перевищує 2^{79} операцій (якими

б не були їх компоненти). При передачі повідомлень зі швидкістю 0,36 максимальне значення стійкості шифросистеми не перевищує 2^{20} операцій, а збільшення швидкості до 0,5 призводить до втрати стійкості. Зазначені факти свідчать про обмежені можливості РПШ Міхалевича-Імаї з погляду сучасних вимог щодо стійкості та практичності в реальних умовах.

Список використаних джерел у другому розділі

1. Mihaljević M.J., Imai H. «A stream ciphering approach based on wiretap channel coding», 8th Central European Conference of Cryptography 2008, Graz, Austria, July 2-4, E-Proc. (3 p.), 2008.

2. Mihaljević M.J., Imai H. «An approach for stream cipher design based on joint computing over random and secret data», *Computing*, V. 85, No 1-2, PP. 153-168, June 2009.

3. Mihaljević M.J., Imai H. «An information-theoretic and computational complexity security analysis of a randomized stream cipher model», *4th Western European Workshop on Research in Cryptology – WeWoRC 2011*, Weimar, Germany, July 20-22, Conf. Record, 2011, P. 21-25.

4. Mihaljević M.J., Imai H. «Employment of homophonic coding for improvement of certain encryption approaches based on the LPN problem», *Symmetric Key Encryption Workshop* – SKEW 2011, Copenhagen, Denmark, Feb. 16-17, E-Proc. (17 p.), 2011.

5. Mihaljević M.J., Oggier F., Imai H. «Homophonic coding design for communication systems employing the encoding-encryption paradigm», URL: http://arXiv:1012.5895v1.

6. Oggier F., Mihaljević M.J. «An information-theoretic analysis of the security of communication systems employing the encoding-encryption paradigm», URL: http://arXiv:1008.0968v1.

7. Wyner A.D. «The wire-tap channel», *Bell. Systems Technical Journal*, V. 54, 1975, P. 1355-1387.

8. MacWilliams F.J., Sloane N.J.A. «*The theory of error-correcting codes*», North-Holland, 1977.

9. Логачев О.А., Сальников А.А., Ященко В.В. «*Булевы функции в теории кодирования и криптологии*», М.:МЦНМО, 2004, 470с.

10. Canteaut A. «Fast correlation attacks against stream ciphers and related open problems», *The 2005 IEEE Information Theory Workshop on Theory and Practice in Information-Theoretic Security* – ITW 2005, E-Proc. (6 p.), Awaji Island, Japan, Oct. 2005.

11. Васильев Ю.Л., Ветухновский Ф.Я., Глаголев В.В. и др. «*Дискретная математика и математические вопросы кибернетики*», Т. 1, М.: Наука, 1974, 311 с.

12. Влэдуц С.Г., Ногин Д.Ю., Цфасман М.А. «*Алгеброгеометрические коды. Основные понятия*», М.: МЦНМО, 2003, 504 с.

13. Vaudenay S. «Decorrelation: a theory for block cipher security», *Journal of Cryptology*, V. 16, No. 4, 2003, P. 249-286.

14. Фано Р. «*Передача информации. Статистическая теория связи*», М.: Мир, 1966, 438 с.

15. Gilbert H., Robshaw M.J.B., Seurin Y. «How to encrypt with the LPN problem», *ICALP 2008*, *Part II*, *Lecture Notes in Computer Science*, V. 5126, 2008, P. 679-690.

РОЗДІЛ 3

МЕТОД ПОБУДОВИ РАНДОМІЗОВАНИХ ПОТОКОВИХ ШИФРОСИСТЕМ З НЕЛІНІЙНИМ ВИПАДКОВИМ КОДУВАННЯМ

Результати попереднього розділу показують, що обчислювальна стійкість рандомізованих потокових шифросистем Міхалевича-Імаї суттєво залежить від будови їх компонент і може бути значно менше, ніж стверджують їх розробники. Більш того, отримані аналітичні межі для швидкості передачі інформації в РПШ Міхалевича-Імаї свідчать про їх обмежені можливості з погляду сучасних вимог щодо стійкості та практичності в реальних умовах.

В даному розділі викладено альтернативний метод побудови РПШ з підвищеною стійкістю, сутність якого полягає в застосуванні для випадкового кодування нелінійних відображень або безключевих геш-функцій.

На відміну від РПШ Міхалевича-Імаї [1 – 5], де використовуються тільки лінійні перетворення над полем з двох елементів, зокрема, завадостійке кодування вхідних повідомлень лінійними кодами, запропонований метод надає більше можливостей для побудови обчислювально стійких РПШ за рахунок розширення класу перетворень, що використовуються в конструкції рандомізатора. Крім того, на відміну від рандомізованих блокових шифросистем [6, 7], до нелінійних перетворень у рандомізаторах потокових шифросистем висуваються дещо інші вимоги, пов’язані зі специфікою атак на потокові шифри. Це обумовлює відмінності між критеріями вибору нелінійних перетворень для побудови рандомізаторів блокових та потокових шифросистем відповідно, зокрема, застосування в ролі таких перетворень геш-функцій. У розділі отримано аналітичні оцінки обчислювальної стійкості рандомізованих потокових шифросистем з нелінійним випадковим кодуванням відносно низки атак, зокрема, найбільш потужних (на сьогодні) атак на основі підібраних

векторів ініціалізації. Наведено також рекомендації щодо вибору компонентів для побудови рандомізаторів зазначених шифросистем.

3.1 Формальне означення рандомізованих потокових шифросистем з нелінійним випадковим кодуванням

За означенням вхідними даними для побудови рандомізованої потокової шифросистеми з нелінійним випадковим кодуванням з параметрами $l, m \in \mathbf{N}$, де $l < m$, та множиною ключів K є такі об‘єкти:

– відображення $\phi : V_{m-l} \to V_l$;

– комутативна групова операція $*$ на множині V_m;

– підстановочна матриця P порядку m;

– генератор гами, який виробляє за ключем $k \in K$ послідовність $f_0(k), f_1(k), \ldots$ булевих векторів довжини m.

Аналогічно шифросистемам Міхалевича-Імаї (див. підрозділ 2.1) вважається, що функції $f_i : K \to V_m$, $i = 0, 1, \ldots$, можуть залежати від загальнодоступних параметрів (векторів ініціалізації).

Для зашифрування на ключі $k \in K$ відкритого тексту $s_0, s_1, \ldots, s_t$, де $s_i \in V_l$, $i = 0, 1, \ldots, t$, відправник генерує послідовність незалежних випадкових рівноймовірних векторів $u_0, u_1, \ldots, u_t$ довжини $m - l$ та обчислює шифрований текст $z_0, z_1, \ldots, z_t$ за формулою

$$z_i = (u_i, s_i \oplus \phi(u_i))P * f_i(k),\ i = 0, 1, \ldots, t. \tag{3.1}$$

Законний одержувач, маючи вектор $f_i(k)$, може обчислити повідомлення $(z_{1,i}, z_{2,i}) = z_i *^{-1} f_i(k)$, де $z_{1,i} \in V_{m-l}$, $z_{2,i} \in V_l$, а операція $*^{-1}$ визначається

таким співвідношенням: $x = y *^{-1} z \Leftrightarrow y = x * z$, $x, y, z \in V_m$. Після цього він може відновити повідомлення s_i за формулою $s_i = \phi(z_{1,i}) \oplus z_{2,i}$ (рис. 3.1). При цьому супротивник для знаходження ключа k вимушений мати справу зі спотвореною гамою $(u_i, s_i \oplus \phi(u_i))P * f_i(k)$, $i = 0, 1, ..., t$.

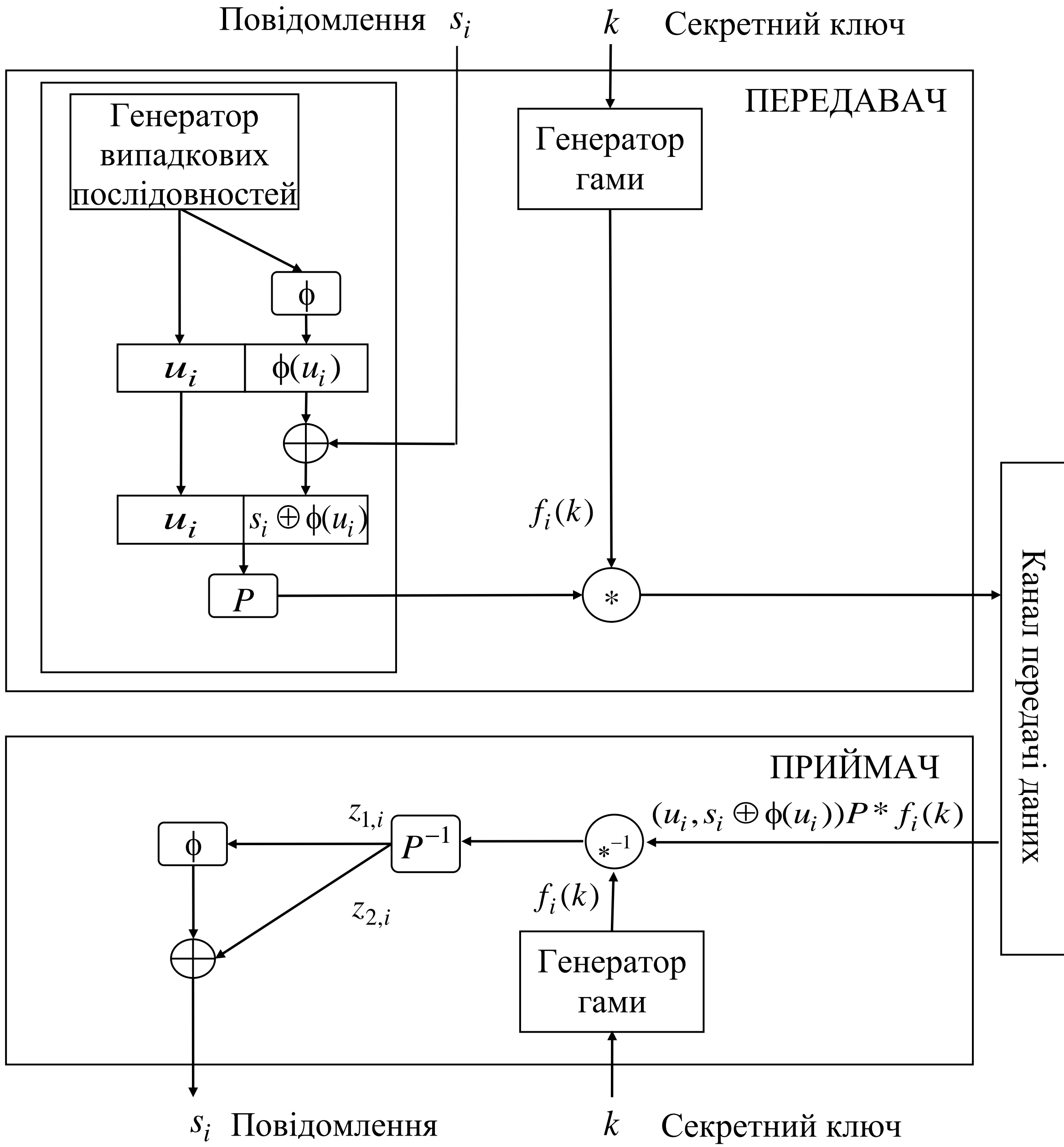


Рис. 3.1. Схема РПШ з нелінійним випадковим кодуванням

Зауважимо, що вхідні дані $\phi, *, P$ слід вибирати з урахуванням вимог як до криптографічної стійкості, так і ефективності реалізації перетворень вигляду (3.1).

Наприклад, виходячи з останньої вимоги, можна покласти $a * b = (a+b) \bmod 2^m$, де довільні вектори $a, b \in V_m$ ототожнюються з відповідними числами з множини $\{0, 1, ..., 2^m - 1\}$, та визначити P як матрицю оператора циклічного зсуву на певну кількість розрядів.

Відображення ϕ слід вибирати більш ретельно, оскільки його властивості (як показано нижче) суттєвим чином впливають на стійкість шифросистеми, що розглядається.

Пропонується використовувати один із двох загальних підходів:

1) застосовувати в ролі ϕ нелінійне відображення множини V_l (при $m = 2l$) з гарними криптографічними властивостями на зразок тих, що використовуються в сучасних блокових шифрах;

2) застосовувати в ролі ϕ безключову геш-функцію (таку як Keccak [8] або "Купина" [9]).

Враховуючи той факт, що стійка геш-функція достатньо добре імітує випадкове відображення (в даному випадку множини V_{m-l} в множину V_l), останній варіант видається більш переважним з погляду забезпечення належної стійкості рандомізованої шифросистеми.

Таким чином, застосування нелінійних відображень зазначеного вище вигляду при побудові рандомізаторів є головною відмінністю запропонованих шифросистем від шифросистем Міхалевича-Імаї, де використовуються тільки лінійні перетворення над полем з двох елементів, зокрема, завадостійке кодування вхідних повідомлень лінійними кодами. (Зауважимо, що зазначений вид кодування взагалі не застосовується у перетвореннях вигляду (3.1)).

Зазначимо також, що на відміну від рандомізованих блокових шифросистем [6, 7], до нелінійних перетворень в конструкції рандомізатора

потокової шифросистеми висуваються дещо інші вимоги, пов'язані зі специфікою атак на потокові шифри. Це обумовлює відмінності між критеріями вибору нелінійних перетворень для побудови рандомізаторів блокових та потокових шифросистем відповідно, зокрема, застосування в ролі таких перетворень геш-функцій.

3.2. Обчислювальна стійкість РПШ з нелінійним випадковим кодуванням

3.2.1. Обчислювальна стійкість РПШ з нелінійним випадковим кодуванням відносно атак на основі відомих шифрованих повідомлень. Розглянемо важливий окремий випадок, в якому операція $*$ у формулі (3.1) співпадає з операцією $\oplus$ покоординатного додавання за модулем 2, а матриця P є тотожною порядку m. Припустимо також, що джерело відкритих повідомлень є безнадлишковим, тобто виробляє послідовність незалежних рівноймовірних двійкових векторів довжини l, а генератор гами виробляє послідовність незалежних, але нерівноймовірних символів гами, розподілених за законом

$$\mathbf{P}(f_i(k) = a) = (1-p)^{m-\|a\|} p^{\|a\|},\ a \in V_m, \tag{3.2}$$

де p (відповідно $1-p$) є ймовірність появи одиничного (відповідно нульового) знаку гами, $0 < p < 1/2$, $\|a\|$ є вагою Гемінга довільного вектора $a \in V_m$.

Метою атаки на РПШ на основі відомого шифрованого повідомлення є відновлення (для деякого $i = 0, 1, \ldots, t$) символу відкритого тексту s_i за символом шифротексту z_i вигляду (3.1). Зауважимо, що неідеальність генератора гами (див. рівність (3.2)) є необхідною умовою для успішного проведення такої атаки.

Будь-який алгоритм (процедура або схема) відновлення символу відкритого тексту за відповідним символом шифрованого задається довільним відображенням $\delta: V_m \to V_l$. Ймовірність правильного відновлення символу s_i за символом z_i визначається за формулою $\pi(\delta) = \mathbf{P}(\delta(z_i) = s_i)$, де $\mathbf{P}$ позначає сумісний розподіл ймовірностей незалежних випадкових векторів s_i та u_i (див. формулу (3.1)). Відображення $\delta^*: V_m \to V_l$ називається *оптимальною схемою дешифрування*, якщо $\pi(\delta^*) \geq \pi(\delta)$ для будь-якого $\delta: V_m \to V_l$.

З метою оцінювання ймовірності $\pi(\delta^*)$, яка характеризує стійкість РПШ з нелінійним випадковим кодуванням відносно атаки на основі відомого шифрованого повідомлення, введемо декілька додаткових позначень.

Для будь-якого $s \in V_l$ позначимо $C_s = \{(u, \phi(u) \oplus s) : u \in V_{m-l}\}$. Зрозуміло, що множини C_s, $s \in V_l$, попарно не перетинаються, а їх об'єднання співпадає з множиною V_m. Більш того, кожна множина C_s отримується шляхом зсуву множини C_0, яка є систематичним нелінійним двійковим кодом, що задається відображенням $\phi: C_s = C_0 \oplus (0, s)$, де $C_0 = \{(u, \phi(u)) : u \in V_{m-l}\}$, а $(0, s)$ позначає вектор довжини m, перші $m-l$ координат якого дорівнюють нулю.

Для коду $C = C_0$ позначимо

$$B_i'(C) = 2^{-2(m-l)} \sum_{\substack{x \in V_m: \\ \|x\| = i}} \left| \hat{I}_C(x) \right|^2, \; i = 0, 1, \ldots, m, \tag{3.3}$$

де I_C – індикатор множини C ($I_C(x) = 1$, якщо $x \in C$; $I_C(x) = 0$ – у протилежному випадку), а $\hat{I}_C(x) = \sum_{y \in C} (-1)^{xy}$, $x \in V_m$ є перетворенням Фур'є функції I_C. Набір чисел (3.3) називається *дуальним спектром відстаней* коду C [10]. Безпосередньо з рівності (3.3) випливає,

що $B'_0(C)=1$. При цьому найменше натуральне число $d'=d'(C)$, яке задовольняє умові $B'_{d'}(C)\neq 0$, називається *дуальною відстанню* (нелінійного) коду C [10].

Твердження 3.1. За умови (3.2) ймовірність правильного відновлення символу відкритого тексту за символом шифротексту РПШ, що описується співвідношенням (3.1), за допомогою оптимальної схеми дешифрування $\delta^*: V_m \to V_l$ задовольняє нерівностям

$$2^{-l} \le \pi(\delta^*) \le 2^{-l} + (1-2p)^{d'}, \tag{3.4}$$

де d' є дуальною відстанню коду $C=\{(u,\phi(u)): u\in V_{m-l}\}$.

Доведення. Розглянемо відображення $\sigma(x_1,x_2)=\phi(x_1)\oplus x_2$, $x_1\in V_{m-l}$, $x_2\in V_l$ та скористаємося формулою

$$\pi(\delta) = 2^{-2m}\sum_{s\in V_l}\sum_{a\in V_m}(1-2p)^{\|a\|}\,\hat{I}_{\sigma,s}(a)\,\hat{I}_{\delta,s}(a) \tag{3.5}$$

де $\delta: V_m \to V_l$, $I_{\sigma,s}$ та $I_{\delta,s}$ є індикатори множин $\sigma^{-1}(s)$ та $\delta^{-1}(s)$ відповідно, $s\in V_l$, а $\hat{I}_{\sigma,s}$, $\hat{I}_{\delta,s}$ є перетворення Фур'є зазначених функцій [11].

Помітимо, що на підставі оптимальності схеми δ^* та означення перетворення Фур'є

$$\pi(\delta^*) \ge \pi(\sigma) = 2^{-2m}\sum_{s\in V_l}\sum_{a\in V_m}(1-2p)^{\|a\|}\left|\hat{I}_{\sigma,s}(a)\right|^2 \ge$$

$$\geq 2^{-2n} \sum_{s \in V_l} \left| \hat{I}_{\sigma,s}(0) \right|^2 = 2^{-2m}\ 2^l\ 2^{2(m-l)} = 2^{-l},$$

звідки випливає нижня межа (3.4).

Для доведення верхньої межі виділимо у внутрішній сумі у правій частині рівності (3.5) доданок, який відповідає значенню $a = 0$ та скористаємося рівностями $\hat{I}_{\sigma,s}(a) = 0$, $1 \leq \|a\| \leq d' - 1$, що випливають з формули (3.3) та означення дуальної відстані коду C. В результаті отримаємо, що

$$\pi(\delta) = 2^{-2m} \sum_{s \in V_l} \hat{I}_{\sigma,s}(0)\, \hat{I}_{\delta,s}(0) + 2^{-2m} \sum_{s \in V_l} \sum_{\substack{a \in V_m: \\ \|a\| \geq d'}} (1-2p)^{\|a\|}\, \hat{I}_{\sigma,s}(a)\, \hat{I}_{\delta,s}(a) =$$

$$= 2^{-l} + 2^{-2m} \sum_{s \in V_l} \sum_{\substack{a \in V_m: \\ \|a\| \geq d'}} ((1-2p)^{\frac{1}{2}\|a\|}\, \hat{I}_{\sigma,s}(a))\, ((1-2p)^{\frac{1}{2}\|a\|}\, \hat{I}_{\delta,s}(a)),$$

звідки в силу нерівності Коши-Буняковського випливає, що

$$\pi(\delta) \leq 2^{-l} + 2^{-2m} \left(\sum_{s \in V_l} \sum_{\substack{a \in V_m: \\ \|a\| \geq d'}} (1-2p)^{\frac{1}{2}\|a\|}\, \hat{I}_{\sigma,s}(a) \right)^{\frac{1}{2}} \left(\sum_{s \in V_l} \sum_{\substack{a \in V_m: \\ \|a\| \geq d'}} (1-2p)^{\frac{1}{2}\|a\|}\, \hat{I}_{\sigma,s}(a) \right)^{\frac{1}{2}}. \tag{3.6}$$

Використовуючи рівність Парсеваля [11], оцінимо зверху другий співмножник у виразі (3.6) таким чином:

$$(\sum_{s\in V_l}\sum_{\substack{a\in V_m:\\ \|a\|\geq d'}}(1-2p)^{\|a\|}\left|\hat{I}_{\delta,s}(a)\right|^2)^{1/2} \leq (1-2p)^{d'/2}(\sum_{s\in V_l}\sum_{\substack{a\in V_m:\\ \|a\|\geq d'}}\left|\hat{I}_{\delta,s}(a)\right|^2)^{1/2} \leq$$

$$\leq (1-2p)^{d'/2}(\sum_{s\in V_l}\sum_{a\in V_m}\left|\hat{I}_{\delta,s}(a)\right|^2)^{1/2} \leq$$

$$\leq (1-2p)^{d'/2}(2^m\sum_{s\in V_l}\sum_{a\in V_m}\left|I_{\delta,s}(a)\right|^2)^{1/2} = 2^m(1-2p)^{d'/2}.$$

Аналогічно отримаємо нерівність

$$(\sum_{s\in V_l}\sum_{\substack{a\in V_m:\\ \|a\|\geq d'}}(1-2p)^{\|a\|}\left|\hat{I}_{\sigma,s}(a)\right|^2)^{1/2} \leq 2^m(1-2p)^{d'/2}.$$

Підставляючи наведені оцінки у формулу (3.6), отримаємо, що для будь-якого $\delta: V_m \to V_l$ має місце нерівність $\pi(\delta) \leq 2^{-l} + (1-2p)^{d'}$. Отже, справедлива верхня межа (3.4).

Твердження доведено.

Безпосередньо з твердження 3.1 випливає такий результат.

Наслідок 3.1. Найменший обсяг t матеріалу, необхідного для успішного знаходження за шифротекстом $z_0, z_1, \ldots, z_{t-1}$ хоча б одного символу s_i відкритого тексту ($i = 0, 1, \ldots, t-1$) за допомогою довільної схеми дешифрування $\delta: V_m \to V_l$ є не менше ніж

$$t_* = \frac{1}{2^{-l} + (1-2p)^{d'}}. \tag{3.7}$$

Таким чином, за наведених вище умов стійкість РПШ з нелінійним випадковим кодуванням відносно атаки на основі відомого шифрованого повідомлення визначається дуальною відстанню нелінійного коду C, який будується за відображенням ϕ. Зі збільшенням дуальної відстані коду складність атаки на шифросистему збільшується та наближається до складності повного перебору усіх двійкових векторів довжини l.

Як приклад застосування наведених результатів, розглянемо послідовність кодів Препарати (Π_r: $r = 4, 6, \ldots$). Для кожного парного $r \geq 4$ код Π_r має довжину $m = 2^r$, потужність 2^{m-2r}, дуальну відстань $d' = 2^{r-1} - 2^{(r-2)/2}$ і є нелінійним систематичним кодом [10], с. 454.

Позначимо $l = 2r$ та задамо відображення $\phi_r : V_{m-l} \to V_l$ за правилом: $\forall u \in V_{m-l}$: $\phi_r(u) = v \Leftrightarrow (u, v) \in \Pi_r$. Вважаючи у формулі (3.1) $\phi = \phi_r$, $* = \oplus$, $P = I_m$, отримаємо РПШ з нелінійним кодуванням, побудовану на базі коду Препарати Π_r. Зауважимо, що оскільки Π_r є $\mathbf{Z}_4$-лінійним кодом [12 – 14], то для кожного $u \in V_{m-l}$ значення $\phi_r(u)$ можна обчислити за $O(m^2)$ двійкових операцій.

На рис. 3.2 зображені графіки параметра $\log(t_*)$ як функцій ймовірності p появи одиничного знаку гами, обчислених за формулою (3.7) для РПШ з нелінійним випадковим кодуванням, побудованих на базі кодів Π_r при $r = 8, 10, 12$.

Як видно із залежностей на рисунку, зі збільшенням ймовірності p появи одиничного знаку гами або довжини m кодового слова обсяг матеріалу, необхідного для відновлення хоча б одного символу відкритого тексту, швидко зростає. Зокрема, при $p \geq 0{,}07$ значення (3.7) майже не відрізняється від числа 2^l, де $l = 2r$ є довжиною окремого символу відкритого тексту.

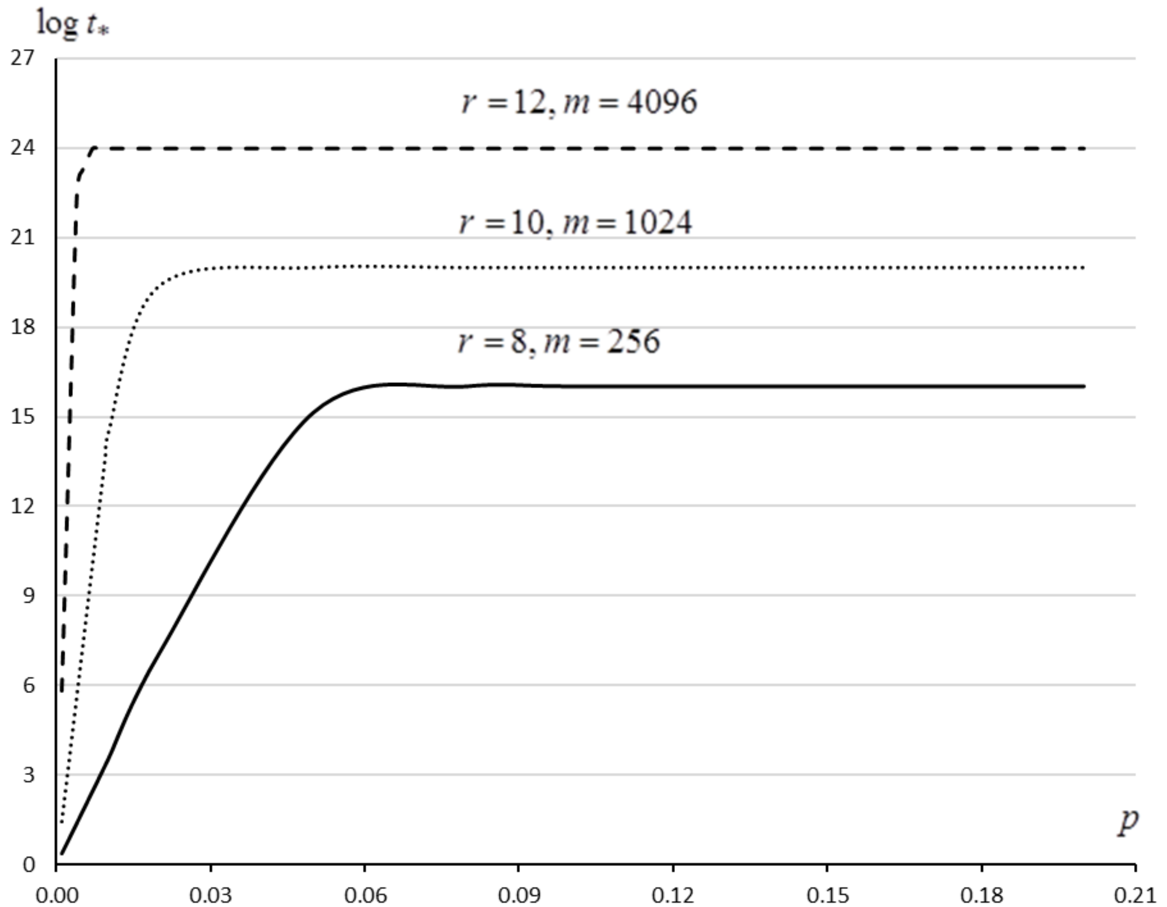


Рис. 3.2. Нижні межі стійкості РПШ з нелінійним випадковим кодуванням, побудованих на базі кодів Препарати

Таким чином, навіть у випадку неякісного генератора гами запропонований метод дозволяє забезпечити потрібний рівень стійкості рандомізованої потокової шифросистеми відносно атаки на основі відомого шифрованого повідомлення.

3.2.2. Обчислювальна стійкість РПШ з нелінійним випадковим кодуванням відносно атаки на основі підібраних векторів ініціалізації в загальному випадку. Розглянемо одну з найбільш потужних атак на рандомізовані потокові шифросистеми, описану в п. 2.2.3, коли супротивник має доступ до шифрувального оракулу з невідомим

(вибраним випадково та рівноймовірно з множини K) ключем k та може вибирати на свій розсуд вектори ініціалізації, які визначають функції f_i, $i = 0, 1, \ldots$. Метою цієї атаки є відновлення вектора $f_i(k)$ (для деякого фіксованого i) з набору повідомлень, отриманих в результаті зашифровування t разів того ж самого повідомлення $s_i = 0$ при однаковому векторі ініціалізації. В цьому випадку супротивник може отримати рівняння вигляду $(u_j, \phi(u_j))P * f_i(k) = y_j$, $j = \overline{1, t}$, де значення $y_1, y_2, \ldots, y_t$ є відомими, а значення $f_i(k)$, $u_1, \ldots, u_t$ невідомі.

З метою оцінювання стійкості РПШ з нелінійним випадковим кодуванням відносно зазначеної атаки розглянемо більш загальну задачу.

Нехай задано систему випадкових рівнянь

$$\xi_j + x = y_j, \; j = \overline{1, t} \tag{3.8}$$

над скінченною абелевою групою $(G, +)$, де $\xi_1, \xi_2, \ldots, \xi_t$ – незалежні випадкові величини з рівномірним розподілом на множині $M \subseteq G$, а $y_j = x_0 + \xi_j$ є результатом підстановки в j-е рівняння системи невідомого елемента $x_0 \in G$, $j = 1, 2, \ldots, t$. Необхідно відновити цей елемент за відомими значеннями $y_1, y_2, \ldots, y_t$ і M.

Зрозуміло, що відновлення вектора $f_i(k)$ при проведенні зазначеної вище атаки зводиться до розв‘язання сформульованої вище задачі, якщо покласти

$$(G, +) = (V_m, *), \; x_0 = f_i(k), \; M = \{(u, \phi(u))P : \; u \in V_{m-l}\}. \tag{3.9}$$

Зрозуміло також, що множина всіх розв’язків системи рівнянь (3.8) дорівнює $\bigcap_{j=1}^{t} (y_j - M)$ і містить, принаймні, один елемент, що дорівнює x_0.

Для будь-якого $x \in G$ позначимо $S_x(\xi) = \bigcap_{j=1}^{t} (x + \xi_j - M)$, де $\xi = (\xi_1, \xi_2, ..., \xi_t)$. Тоді $S_x(\xi)$ є перетином незалежних випадкових множин, розподілених за законом

$$\mathbf{P}\{x + \xi_j - M = A\} = \frac{|\{y \in M : A = x + y - M\}|}{|M|}, \; A \subseteq G.$$

Припустимо, що для знаходження істинного розв'язку x_0 системи рівнянь (3.8) супротивник використовує найбільш природний алгоритм.

Алгоритм 3.1.

Перебрати усі значення ξ_1 та перевірити умову $y_1 - \xi_1 \in \bigcap_{j=2}^{t} (y_j - M)$ (вважається, що перебір здійснюється до першого успіху).

Оцінимо часову складність алгоритму 3.1. Припустимо, що додавання двох елементів $a, b \in G$ та перевірка умови $a \in M$ для будь-якого $a \in G$ займає постійний час.

Позначимо

$$d_M(a) = \frac{|\{y \in M : y + a \in M\}|}{|M|}, \; a \in G, \tag{3.10}$$

$$d_M = \max\{d_M(a) : a \in G \setminus \{0\}\}. \tag{3.11}$$

Твердження 3.2. Нехай $d_M < 1$. Тоді для будь-якого $0 < \delta < 1$ і

$$t = \left\lceil \frac{\log(\delta^{-1} |M|)}{\log({d_M}^{-1})} \right\rceil + 1$$

істинний розв'язок x_0 системи випадкових рівнянь (3.8) можна знайти з імовірністю не менше $1-\delta$ за $O(|M|t)$ операцій.

Доведення. Достатньо переконатися, що ймовірність p_e помилкового відновлення елемента x_0 за допомогою алгоритму 3.1 задовольняє нерівності

$$p_e \le |M| {d_M}^{t-1}. \qquad (3.12)$$

Дійсно, якщо алгоритм 3.1 припускається помилки, існує елемент $x \in G \setminus \{x_0\}$, який належить множині $S_{x_0}(\xi) = \bigcap_{j=1}^{t}(x_0 + \xi_j - M)$. Отже, оскільки $\xi_1, \xi_2, \ldots, \xi_t$ є незалежними випадковими величинами з рівномірним розподілом на множині M , то отримаємо

$$p_e \le \sum_{x \ne x_0} \mathbf{P}\{x \in S_{x_0}(\xi)\} = \sum_{x \ne x_0} \Big(|M|^{-1} \cdot |\{y \in M : y + x_0 - x \in M\}| \Big)^t .$$

Звідси, використовуючи формули (3.10), (3.11) та позначення $I_M(z)$, $z \in G$ для індикатора множини M , отримаємо, що

$$p_e \le \sum_{x \ne x_0} \big(d_M(x_0 - x)\big)^t \le {d_M}^{t-1} \sum_{x \ne x_0} d_M(x_0 - x) =$$

$$= {d_M}^{t-1} |M|^{-1} \sum_{a \ne 0} \sum_{z \in G} I_M(z) I_M(z+a) = {d_M}^{t-1} |M|^{-1} \sum_{z \in G} I_M(z) \sum_{a \ne 0} I_M(z+a) =$$

$$= {d_M}^{t-1}(|M| - 1) \le |M| {d_M}^{t-1} .$$

Таким чином, нерівність (3.12), а поряд з нею й твердження, доведені.

Наслідок 3.2. Нехай виконується умова (3.9) і $d_M = 2^{c-(m-l)}$, де $c = \text{const}$. Тоді істинний розв'язок x_0 системи рівнянь (3.8) може бути знайдений з імовірністю не менше $1-\delta$ за $O\left(2^{m-l}\left(1+\frac{\log\delta^{-1}}{m-l}\right)\right)$ операцій при $\delta \to 0$ та $m-l \to \infty$.

Відмітимо, що в найгіршому випадку алгоритм 3.1 вимагає перебору всіх $(m-l)$-вимірних булевих векторів, тому він стає практично незастосовним, наприклад, при $m-l > 64$.

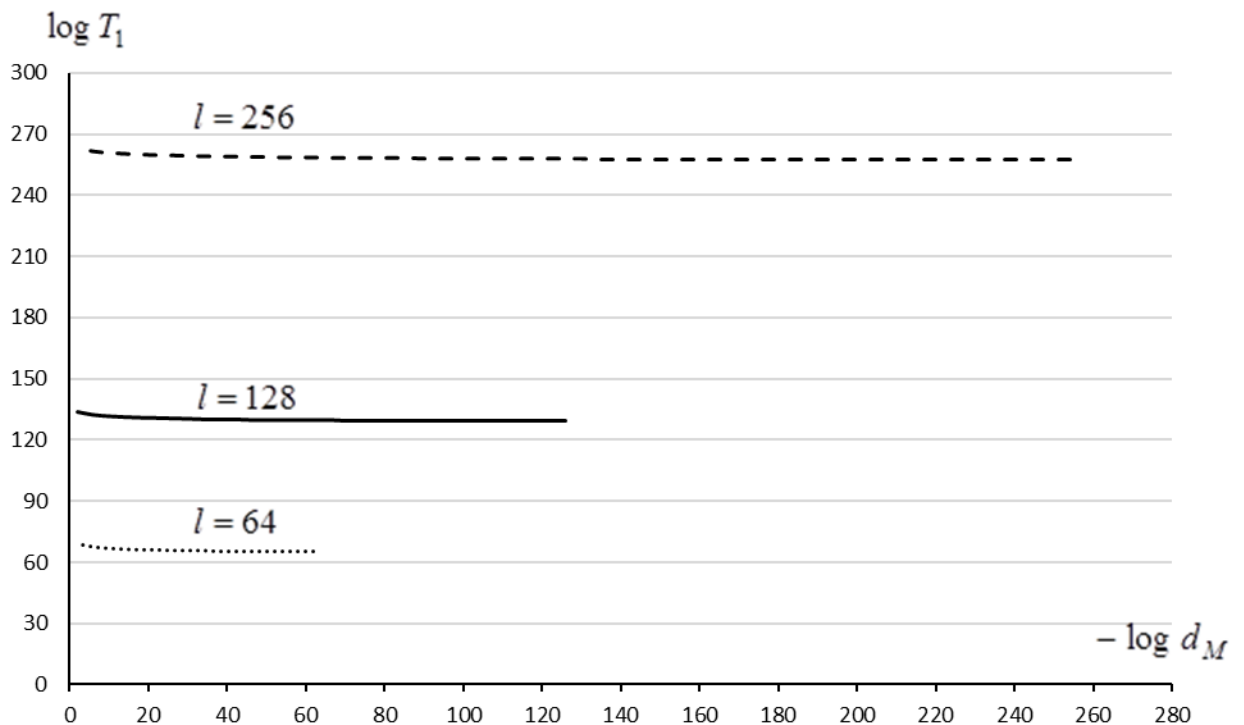


Рис. 3.3. Верхні оцінки обчислювальної складності атаки на РПШ з нелінійним випадковим кодуванням на основі підібраних векторів ініціалізації ($\delta = 0{,}05$)

На рис. 3.3 зображено графіки верхніх меж двійкового логарифму обчислювальної складності T_1 описаної атаки на рандомізовані потокові шифросистеми з параметрами $l \in \{64, 128, 256\}$, $m = 2l$, побудовані на базі

підстановок $\phi: V_l \to V_l$ (при цьому $(G,+)=(V_m,\oplus)$, $x_0 = f_i(k)$, $M = \{(u,\phi(u)):\ u \in V_{m-l}\}$; відомо [15], що в цьому випадку значення (3.11) змінюється від 1 до 2^{2-l}).

Як видно з рисунку, складність відновлення вектора $f_i(k)$ пропорційна числу 2^l і в кожному з трьох випадків дуже повільно спадає зі зменшенням параметра d_M. Так, при $l=128$ обчислювальна стійкість РПШ відносно наведеної атаки дорівнює 2^{134}, якщо $d_M = 2^{-2}$ і 2^{130}, якщо $d_M = 2^{-120}$. Це свідчить про можливість забезпечення гарантованої стійкості РПШ з нелінійним випадковим кодуванням відносно цієї атаки шляхом належного вибору підстановки ϕ.

3.2.3. Обчислювальна стійкість РПШ з нелінійним випадковим кодуванням відносно атаки на основі підібраних векторів ініціалізації в окремому випадку. Розглянемо важливий окремий випадок, коли в системі рівнянь (3.8)

$$(G,+)=(V_m,\oplus),\ x_0 = f_i(k),\ M = \{(u,\phi(u)):\ u \in V_{m-l}\}. \tag{3.13}$$

Зауважимо, що в цьому випадку параметр (3.11) співпадає з величиною

$$D_\phi = \max_{\alpha \in V_{m-l} \setminus \{0\},\, \beta \in V_l} \{2^{-(m-l)}\,|\{z \in V_{m-l} : \phi(z \oplus \alpha) \oplus \phi(z) = \beta\}|\}, \tag{3.14}$$

яка характеризує стійкість відображення ϕ відносно різницевого криптоаналізу (див., наприклад, [16]).

В зазначеному випадку для розв‘язання системи рівнянь (3.8) можна застосувати інший, іноді більш ефективний в порівнянні з наведеним вище

метод, що є близьким до лінійного криптоаналізу та кореляційних атак на РПШ Міхалевича-Імаї (див., наприклад, [3 – 5, 17] та п. 2.2.3).

Для будь-якого $n \in \mathbf{N}$, $a = (a_1, \dots, a_n)$, $b = (b_1, \dots, b_n) \in V_n$ позначимо $ab = a_1 b_1 \oplus \cdots \oplus a_n b_n$. Покладемо

$$l_\phi(a,b) = 2^{-(m-l)} \, |\{z \in V_{m-l} : az \neq b\phi(z)|, \; a \in V_{m-l}, \; b \in V_l;$$

$$L_\phi = \max_{a \in V_{m-l}, \, b \in V_l \setminus \{0\}} \{|1 - 2 l_\varphi(a,b)|\}. \tag{3.15}$$

Тоді за умови (3.13) відновити істинний розв‘язок x_0 системи рівнянь (3.8) можна за допомогою наступного алгоритму.

Алгоритм 3.2.

1. Вибрати m лінійно незалежних векторів (a_r, b_r), де $a_r \in V_{m-l}$, $b_r \in V_l \setminus \{0\}$ таких, що $l_\phi(a_r, b_r) \neq 1/2$, $r \in \overline{1, m}$.

2. Для кожного $r \in \overline{1, m}$ отримати з (3.8) систему рівнянь

$$(a_r u_j \oplus b_r \phi(u_j)) \oplus (a_r, b_r) x_0 = (a_r, b_r) y_j, \; j = \overline{1, t} \tag{3.16}$$

і відновити величину $(a_r, b_r) x_0$, використовуючи мажоритарне правило:

– якщо $l_\phi(a_r, b_r) < 1/2$, то

$$(a_r, b_r) x_0 \stackrel{\text{def}}{=} 0 \;\Leftrightarrow\; \sum_{j=1}^{t} (a_r, b_r) y_j < t/2;$$

– якщо $l_\phi(a_r, b_r) > 1/2$, то

$$(a_r, b_r)x_0 \overset{\text{def}}{=} 0 \Leftrightarrow \sum_{j=1}^{t}(a_r, b_r)y_j > t/2.$$

3. Знайти вектор x_0 з отриманих величин $(a_r, b_r)x_0$, використовуючи алгоритм Гаусса.

Зауважимо, що вибір векторів (a_1, b_1), …, (a_m, b_m) на кроці 1 та обернення складеної з них матриці на кроці 3 алгоритму 3.2 виконується одноразово, на етапі передобчислень. Тому часова складність цього алгоритму фактично визначається часом виконання кроку 2.

Наступне твердження встановлює верхню оцінку обчислювальної складності описаної атаки на РПШ з нелінійним випадковим кодуванням.

Твердження 3.3. За умови (3.13) супротивник може відновити на кроці 2 алгоритму 3.2 всі значення $(a_r, b_r)x_0$, $r = \overline{1, m}$, з імовірністю не менше $1-\delta$, $0<\delta<1$, використовуючи $O(mt\log t)$ двійкових операцій і

$$t = \left\lceil 1/2 \cdot \max_{1\le r\le m}\{|1-2l_\phi(a_r, b_r)|^{-2}\}\ln(\delta^{-1}m)\right\rceil \tag{3.17}$$

довільних рівнянь системи (3.8).

Доведення. На підставі формули (3.16) для будь-якого $r = 1, 2, \ldots, m$ справедливі такі рівності:

$$(a_r, b_r)y_j = (a_r, b_r)x_0 \oplus \varsigma_{j,r},\ j = \overline{1, t},$$

де $\varsigma_{j,r}$ є незалежними випадковими величинами, розподіленими за законом

$$\mathbf{P}(\varsigma_{j,r} = 1) = 1 - \mathbf{P}(\varsigma_{j,r} = 0) = 1/2 \cdot \left(1 - |1-2l_\phi(a_r, b_r)|\right),\ j = \overline{1, t}$$

Нехай $l_{\phi}(a_r,b_r)<1/2$. Оскільки для відновлення значення $(a_r,b_r)x_0$ використовується мажоритарне правило, визначене на кроці 2 алгоритму 3.2, то, спираючись на нерівність Чернова (див., наприклад, [18], с. 300), можна оцінити ймовірність помилки таким чином:

$$\mathbf{P}\left(\sum_{j=1}^{t}\varsigma_{j,r}\geq t/2\right)=\mathbf{P}\left(t^{-1}\sum_{j=1}^{t}\varsigma_{j,r}-1/2\cdot\left(1-|1-2l_{\phi}(a_r,b_r)|\right)\geq|1-2l_{\phi}(a_r,b_r)|\right)\leq$$

$$\leq\exp\{-2t(1-2l_{\phi}(a_r,b_r))^2\}\leq\exp\{-2t\cdot\max_{1\leq r\leq m}\{|1-2l_{\phi}(a_r,b_r)|^2\}\}.$$

Аналогічна оцінка ймовірності помилки отримується у випадку $l_{\phi}(a_r,b_r)>1/2$.

Отже, ймовірність того, що всі значення $(a_r,b_r)x_0$, $r=1,2,\ldots,m$, відновлені коректно, обмежена знизу числом

$$1-m\exp\{-2t\cdot\max_{1\leq r\leq m}\{|1-2l_{\phi}(a_r,b_r)|^2\}\geq 1-\delta,$$

де остання нерівність випливає з означення параметра t. Нарешті, зрозуміло, що складність відновлення усіх значень, $r=1,2,\ldots,m$, з використанням мажоритарного правила складає $O(mt\log t)$ двійкових операції.

Твердження доведено.

На рис. 3.4 зображено графіки верхніх меж двійкового логарифму обчислювальної складності T_2 описаної атаки на рандомізовані потокові шифросистеми з параметрами $l\in\{64,128,256\}$, $m=2l$, побудовані на базі підстановок $\phi:V_l\to V_l$ (відомо [15], що в цьому випадку значення (3.15) змінюється від 1 до $2^{\frac{2-l}{2}}$).

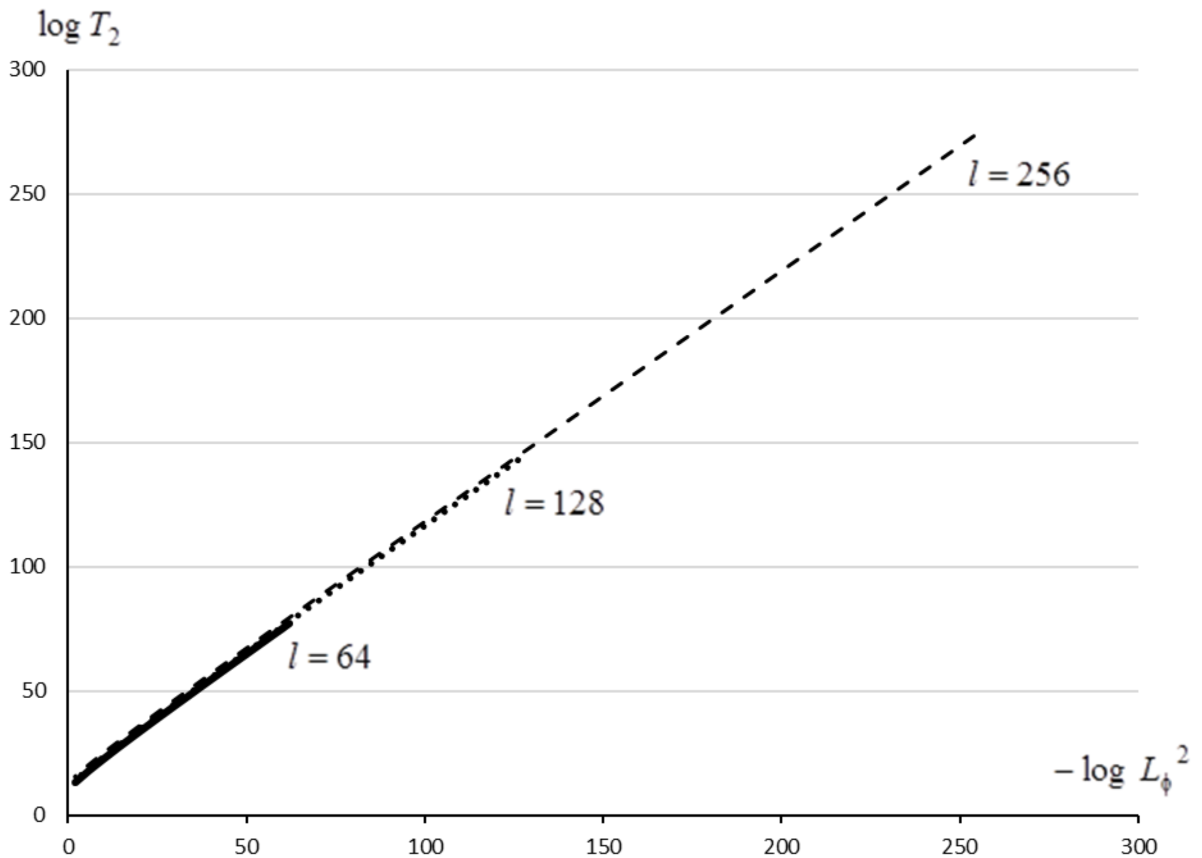


Рис. 3.4. Верхні оцінки обчислювальної складності атаки на РПШ з нелінійним випадковим кодуванням на основі підібраних векторів ініціалізації ($\delta = 0{,}05$)

Як видно з рисунку, складність відновлення значень $(a_r, b_r)x_0$, $r = \overline{1, m}$ є прямо пропорційною числу 2^l і обернено пропорційною параметру $L_\phi{}^2$. Так, у випадку $l = 128$ стійкість шифросистеми складає 2^{20}, якщо $L_\phi{}^2 = 2^{-6}$ і 2^{127}, якщо $L_\phi{}^2 = 2^{-122}$.

Доведення наступного твердження майже дослівно повторює доведення твердження 2.5.

Твердження 3.4. Для відновлення значення $(a_r, b_r)x_0$ з системи рівнянь (3.16) з імовірністю $1/2 + \theta$, $0 < \theta < 1/2$, необхідно не менше ніж

$$t_\theta = 1/4 \cdot \theta^2 \cdot L_\phi^{-2} \qquad (3.18)$$

рівнянь.

Отже, на підставі твердження 3.4 алгоритм 3.2 стає практично незастосовним, якщо значення (3.15) є достатньо малим (скажімо, $L_\phi \le 2^{-32}$).

На рис. 3.5 зображено графіки верхньої та нижньої меж параметра $\log t$ (як функцій параметра $-\log L_\varphi{}^2$), розраховані з використанням співвідношень (3.17) і (3.18) для шифросистем, побудованих на базі підстановок $\phi : V_l \to V_l$ при $l \in \{128, 256\}$, $m = 2l$, $\theta = 0{,}45$ ($\delta = 0{,}05$).

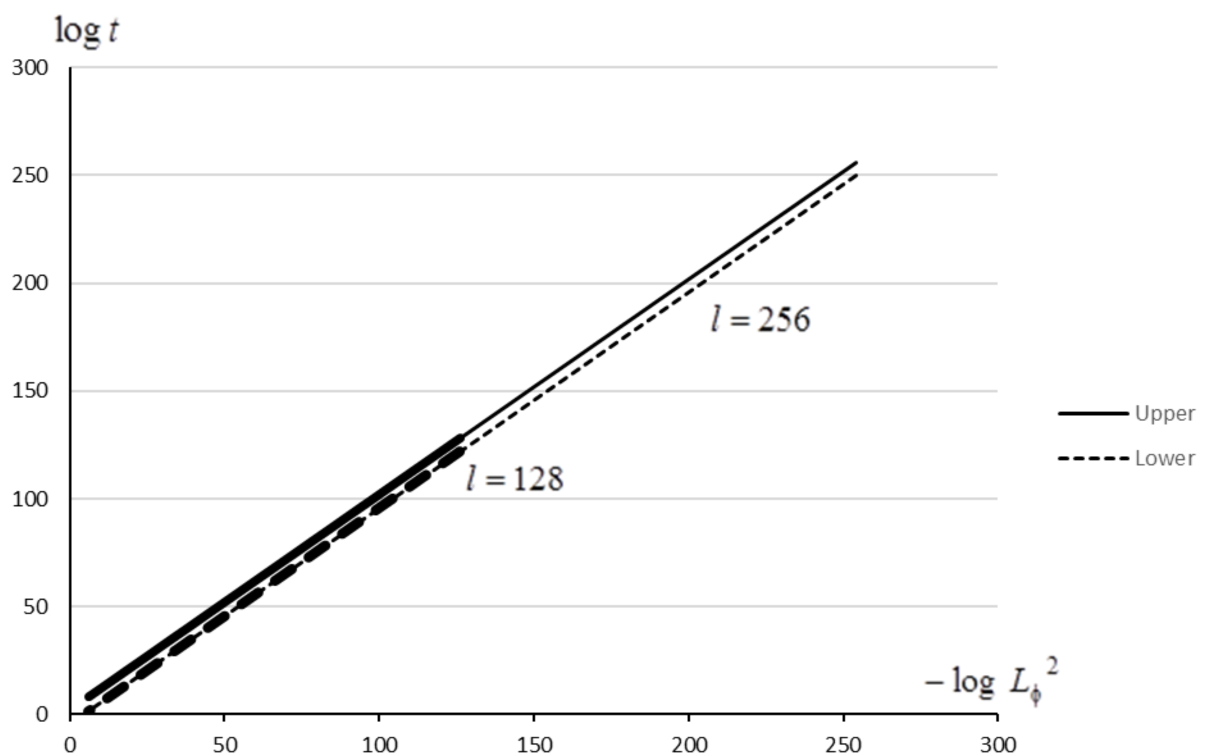


Рис. 3.5. Оцінки обсягу матеріалу, необхідного для успішного відновлення одного біта інформації про ключ РПШ з нелінійним випадковим кодуванням

Як видно із залежностей на рисунку, інформаційна складність описаної атаки є прямо пропорційною числу 2^l і обернено пропорційною параметру $L_\phi{}^2$.

Так, у випадку $l = 256$ параметр t знаходиться у межах від 2^{26} до 2^{32}, якщо ${L_\phi}^2 = 2^{-30}$ та у межах від 2^{246} до 2^{252}, якщо ${L_\phi}^2 = 2^{-250}$.

На завершення розглянемо ще один можливий підхід до розв‘язання системи рівнянь (3.8) за умови (3.13), оснований на ідеях алгебраїчного криптоаналізу.

Позначимо $x = (x_1, x_2)$, $\xi_j = (u_j, \phi(u_j))$, $y_j = (\alpha_j, \beta_j)$, де $x_1, u_j, \alpha_j \in V_{m-l}$, $x_2, \beta_j \in V_l$, $j = \overline{1, t}$. Тоді система (3.8) є рівносильною системі рівнянь:

$$x_1 \oplus u_j = \alpha_j,\ x_2 \oplus \phi(u_j) = \beta_j,\ j = \overline{1, t},$$

яка може бути записана таким чином:

$$\phi(z \oplus (\alpha_1 \oplus \alpha_j)) \oplus \phi(z) = \beta_1 \oplus \beta_j,\ j = \overline{2, t}, \tag{3.19}$$

$$x_1 = \alpha_1 \oplus z,\ x_2 = \phi(z) \oplus \beta_1,\ u_j = \alpha_j \oplus \alpha_1 \oplus z,\ j = \overline{1, t}.$$

Безпосередньо з наведених співвідношень випливає наступне твердження.

Твердження 3.5. За умови (3.13) обчислювальна стійкість шифросистеми, що розглядається, обмежена зверху часовою складністю розв‘язання системи рівнянь (3.19) для довільних (відомих) векторів α_j, β_j, $j = \overline{1, t}$.

Існує багато сімейств булевих відображень з малими значеннями параметрів (3.14) і (3.15) (див., наприклад, [16, 19]). Проте не усі з них гарантують високу складність розв‘язання систем рівнянь вигляду (3.19).

Як приклад, розглянемо відображення $\phi(x) = x^{2^l - 2}$, яке широко застосовується в сучасних блокових шифрах. Відомо, що $D_\phi = {L_\phi}^2 = 2^{2-l}$, але кожне окреме рівняння системи (3.19) має щонайбільше чотири розв‘язки, які

можуть бути знайдені за реальний час [20]. Таким чином, маючи невелику (а саме, $t = \left\lceil \frac{l + \log \delta^{-1}}{\log(D_{\phi}^{-1})} \right\rceil + 1 = \left\lceil \frac{l + \log \delta^{-1}}{l - 2} \right\rceil + 1$) кількість рівнянь в системі (3.8), можна знайти за реальний час (однозначно з імовірністю не менше $1 - \delta$, $0 < \delta < 1$) її розв‘язок x_0, розв‘язуючи систему (3.19).

Поряд з тим, задача розв'язання системи рівнянь (3.19) для довільного відображення $\phi : V_{m-l} \to V_l$ виявляється обчислювально складною. Більш того, невідомо ефективних алгоритмів розв'язання таких систем рівнянь для будь-якої з сучасних обчислювально стійких геш-функцій. Видається дуже правдоподібним, що існування таких алгоритмів може бути небажаною властивістю, яка дозволить ефективно відрізнити геш-функцію від суто випадкового відображення.

В цілому, викладені наукові результати встановлюють умови обчислювальної стійкості запропонованих РПШ відносно відомих атак та свідчать про можливість практичної побудови зазначених шифросистем на основі нелінійних відображень чи безключових геш-функцій.

Висновки

1. Основним науковим результатом розділу є запропонований метод побудови рандомізованих потокових шифросистем з нелінійним випадковим кодуванням, сутність якого полягає в застосуванні для випадкового кодування нелінійних відображень або безключевих геш-функцій. На відміну від РПШ Міхалевича-Імаї [1 – 5], де використовуються тільки лінійні перетворення над полем з двох елементів, зокрема, завадостійке кодування вхідних повідомлень лінійними кодами, запропонований метод надає більше можливостей для побудови обчислювально стійких рандомізованих потокових шифросистем за

рахунок розширення класу перетворень, що використовуються в конструкції рандомізатора. Крім того, на відміну від рандомізованих блокових шифросистем [6, 7], до нелінійних перетворень в конструкції рандомізатора потокової шифросистеми висуваються дещо інші вимоги, пов'язані зі специфікою атак на потокові шифри. Це обумовлює відмінності між критеріями вибору нелінійних перетворень для побудови рандомізаторів блокових та потокових шифросистем відповідно, зокрема, застосування в ролі таких перетворень геш-функцій.

2. Обчислювальна стійкість РПШ з нелінійним випадковим кодуванням відносно атаки на основі відомого шифрованого повідомлення визначається дуальною відстанню d' нелінійного коду, який будується за нелінійним відображенням в конструкції рандомізотора шифросистеми. Зі збільшенням дуальної відстані коду складність атаки на шифросистему збільшується та наближається до складності повного перебору усіх вхідних повідомлень довжини l (див. наслідок 3.1). Зокрема, для шифросистем, побудованих на базі кодів Препарати, при $l = 20$, $d' = 496$ обчислювальна стійкість змінюється від $2^{7,19}$ (при $p = 0{,}005$) до 2^{20} (при $p \geq 0{,}05$). Якщо $l = 24$, $d' = 2016$, то стійкість відповідної шифросистеми дорівнює $2^{23,26}$ при $p = 0{,}005$ та 2^{24} при $p \geq 0{,}008$.

3. Обчислювальна стійкість запропонованих РПШ відносно найбільш потужних (з відомих сьогодні) атак на основі підібраних векторів ініціалізації визначається такими властивостями відображення $\phi : V_{m-l} \to V_l$:

– велике значення параметра $m - l$ для протидії перебірній атаці (див. алгоритм 3.1);

– мале значення параметра (3.15) для протидії атакам лінійного типу (див. алгоритм 3.2);

– велика часова складність розв‘язання системи рівнянь (3.19).

На відміну від РПШ Міхалевича-Імаї, що є вразливими до кореляційних атак (або малопрактичними; див. підрозділи 2.2, 2.3), запропоновані

шифросистеми є обчислювально стійкими відносно атаки на основі підібраних векторів ініціалізації, якщо відображення ϕ характеризується малим значенням максимального елемента таблиці лінійних апроксимацій (твердження 3.4). При цьому для підвищення практичності РПШ величина l повинна бути також достатньо великою (наприклад, умова $m-l=l=128$ забезпечує швидкість передачі інформації $l/m=1/2$ незалежно від вибору відображення ϕ).

4. Для шифросистем, побудованих на базі відображень $\phi: V_{m-l} \to V_l$ з параметрами $l=128$, $m=256$, обчислювальна стійкість відносно перебірної атаки (алгоритм 3.1) складає 2^{134} операцій, якщо параметр (3.14) дорівнює 2^{-2} та 2^{130} операцій, якщо цей параметр дорівнює 2^{-120}. Якщо при цьому $L_\phi = 2^{-61}$, то обчислювальна стійкість зазначених шифросистем відносно атаки лінійного типу (алгоритм 3.2) складає 2^{127}.

5. Для протидії відомим, а також іншим можливим атакам відображення ϕ повинно мати властивості, аналогічні властивостям випадкового рівноймовірного відображення множини V_{m-l} в множину V_l. З цієї точки зору природно вибирати в ролі ϕ одну з сучасних безключових геш-функцій. Можливість практичної побудови зазначених РПШ та оцінювання ефективності їх реалізації є предметом наступного розділу.

Список використаних джерел у третьому розділі

1. Mihaljević M.J., Imai H. «A stream ciphering approach based on wiretap channel coding», 8th Central European Conference of Cryptography 2008, Graz, Austria, July 2-4, E-Proc. (3 p.), 2008.

2. Mihaljević M.J., Imai H. «An approach for stream cipher design based on joint computing over random and secret data», *Computing*, V. 85, No 1-2, PP. 153-168, June 2009.

3. Mihaljević M.J., Imai H. «An information-theoretic and computational complexity security analysis of a randomized stream cipher model», *4th Western*

European Workshop on Research in Cryptology – WeWoRC 2011, Weimar, Germany, July 20-22, Conf. Record, 2011, P. 21-25.

4. Mihaljević M.J., Imai H. «Employment of homophonic coding for improvement of certain encryption approaches based on the LPN problem», *Symmetric Key Encryption Workshop* – SKEW 2011, Copenhagen, Denmark, Feb. 16-17, E-Proc. (17 p.), 2011.

5. Mihaljević M.J., Oggier F., Imai H. «Homophonic coding design for communication systems employing the encoding-encryption paradigm», URL: http://arXiv:1012.5895v1.

6. Алексейчук А.Н. «Аналитические границы параметров, определяющие доказуемую стойкость блочных шифров относительно дифференциального криптоанализа», *Захист інформації*, №2, 2007, С. 12-23.

7. Алексейчук А.Н. «Достаточные условия стойкости рандомизированных блочных систем шифрования относительно метода криптоанализа на основе коммутативных диаграмм», *Реєстрація, зберігання і обробка даних*, Т. 9, №2, 2007, С. 61-68.

8. «ECRYPT II: Final hash function status report», URL: http://www.ecrypt.eu.org/ecrypt2/documents/D.SYM.11update.pdf.

9. «A new standard of Ukraine: The Kupyna hash function», URL: http://eprint.iacr.org/2015/885.pdf.

10. MacWilliams F.J., Sloane N.J.A. «*The theory of error-correcting codes*», North-Holland, 1977.

11. Зиновьев В.А., Эриксон Т. «О Фурье-инвариантных разбиениях конечных абелевых групп и тождестве Мак-Вильямс для групповых кодов», *Проблемы передачи информации*, Т. 32, В. 1, 1996, С. 137-143.

12. Hammous A.R., Kumar P.V., Calderbank A.R., Sloane N.J.A., Sole P. «The $\mathbf{Z}_4$-linearity of Kerdock, Preparata, Goethals and related codes», *Bull. Amer. Math. Soc.*, V. 29, No. 2, 1993, P. 218-222.

13. Нечаев А.А. «Код Кердока в циклической форме», *Дискретная математика*, Т. 1, В. 4, 1989, С. 123-139.

14. Кузьмин А.С., Нечаев А.А. «Построение помехоустойчивых кодов с использованием линейных рекуррент над кольцами Галуа», *Успехи математических наук*, Т. 47, №5, 1992, С. 183-184.

15. Логачев О.А., Сальников А.А., Ященко В.В. «*Булевы функции в теории кодирования и криптологии*», М.:МЦНМО, 2004, 470с.

16. Cantenaut A. «Cryptographic functions and design criteria for block ciphers», *INDOCRYPT 2001*, *LNCS 2247*, *Springer Verlag*, 2001, P. 1-16.

17. Alekseychuk A.N, Gryshakov S.V. «On the computational security of randomized stream ciphers proposed by Mihaljević and Imai», *Zakhist Inform.*, No. 4, 2014, P. 328-334.

18. Васильев Ю.Л., Ветухновский Ф.Я., Глаголев В.В. и др. «*Дискретная математика и математические вопросы кибернетики*», Т. 1, М.: Наука, 1974, 311 с.

19. Carlet C. «*Vectorial functions for cryptography*», Boolean Models and Methods in Mathematics, Computer Science and Engineering, Cambridge University Press, 2010, P. 398-469.

20. Nyberg K. «Differentially uniform mappings for cryptography», *Advances in Cryptology*, *EUROCRYPT'93*, *LNCS 765*, 1994, P. 55-64.

РОЗДІЛ 4

РЕЗУЛЬТАТИ ПОРІВНЯННЯ СТІЙКОСТІ ТА ЕФЕКТИВНОСТІ ПРОГРАМНИХ РЕАЛІЗАЦІЙ РАНДОМІЗОВАНИХ ПОТОКОВИХ ШИФРОСИСТЕМ

Даний розділ присвячено розв‘язанню прикладних задач дисертаційного дослідження, які полягають в застосуванні отриманих у попередніх розділах теоретичних результатів для порівняння стійкості та швидкості передачі РПШ Міхалевича-Імаї і РПШ з нелінійним випадковим кодуванням при однаковій довжині шифрованих повідомлень. Крім того, розроблено програмні реалізації шифросистем з нелінійним випадковим кодуванням на основі обґрунтованого вибору їх складових компонент. Показано, що побудовані шифросистеми можуть бути застосовані на практиці для зашифровування/розшифровування даних в режимі реального часу. Запропонований метод побудови РПШ є достатньо гнучким і надає можливість замінювати окремі компоненти більш сучасними або ефективними, не порушуючи роботу системи шифрування в цілому. При цьому уповільнення в часі при зашифровуванні/розшифровуванні (в порівнянні з програмною реалізацією алгоритму AES в режимі зворотного зв‘язку за виходом), що відбувається за рахунок нелінійного випадкового кодування, компенсується суттєвим підвищенням стійкості шифросистеми відносно відомих атак незалежно від криптографічних властивостей вхідного потокового шифру.

В цілому, отримані результати свідчать про помітну перевагу (як з погляду стійкості, так і практичності) побудованих РПШ в порівнянні з рандомізованими потоковими шифросистемами Міхалевича-Імаї.

4.1. Порівняння запропонованих РПШ з шифросистемами Міхалевича-Імаї за стійкістю та швидкістю передачі

4.1.1. Порівняння РПШ за швидкістю передачі при заданих обмеженнях щодо стійкості та довжини шифрованих повідомлень. Нехай задано такі параметри:

– нижня межа $t \geq 1$ інформаційної складності кореляційних атак на РПШ (див. твердження 2.5 і твердження 3.4);

– довжина $n = 2l$ шифрованих повідомлень;

– верхня межа p_e ймовірності помилкового відновлення відкритого повідомлення законним одержувачем інформації РПШ Міхалевича-Імаї (при цьому $p_e + H_2(p_e) < 1$, де $H_2(\cdot)$ – двійкова ентропійна функція; див. лему 2.2);

– число $0 < \theta < 1/2$, де $1/2 + \theta$ є ймовірністю успішного проведення кореляційної атаки на РПШ.

Треба для заданих вхідних параметрів оцінити зверху максимальну швидкість передачі РПШ Міхалевича-Імаї та порівняти її з максимальною швидкістю передачі інформації у запропонованих РПШ з нелінійним випадковим кодуванням.

Використовуючи формули (2.20) і (2.21), оцінимо максимальну швидкість передачі рандомізованих потокових шифросистем Міхалевича-Імаї таким чином:

$$\rho(M) \leq \frac{1 - H_2(p)}{1 - p_e - H_2(p_e)} - 1 - 1/n \cdot \log\left(1 - (2\lambda_\theta(t, p))^{-1}\right), \tag{4.1}$$

якщо $\lambda_\theta(t, p) > 1/2$;

$$\rho(M) \leq \frac{1 - H_2(p)}{1 - p_e - H_2(p_e)} -$$

$$-H_2\left(1/2\cdot(1-\sqrt{1-2\lambda_\theta(t,p)+2/n})-1/n\right)+\log(n\sqrt{n})/n, \qquad (4.2)$$

якщо $1/n<\lambda_\theta(t,p)\le 1/2$, де $\lambda_\theta(t,p)=-\dfrac{\log(4\theta^{-2}t)}{2n\log(1-2p)}$.

Оскільки функція від p, що визначається виразами в правих частинах нерівностей (4.1) і (4.2), має єдину точку максимуму в інтервалі $(0,1/2)$, то для знаходження цього максимуму можна скористатися методом послідовних наближень. Результати представлені в табл. 4.1.

Таблиця 4.1

Верхні межі максимальної швидкості передачі РПШ Міхалевича-Імаї при заданих обмеженнях щодо стійкості та довжини шифрованих повідомлень

($p_e=10^{-8}$, $\theta=0{,}45$)

$n=256$									
$\log t$	121	41	35	30	25	20	15	10	5
$\rho(M)$	–	0,001	0,047	0,089	0,136	0,191	0,255	0,334	0,434
$n=512$									
$\log t$	249	79	65	50	40	30	20	10	5
$\rho(M)$	–	0,004	0,056	0,131	0,189	0,258	0,347	0,463	0,544
$n=1024$									
$\log t$	505	154	120	100	80	60	40	20	10
$\rho(M)$	–	0,001	0,074	0,127	0,188	0,261	0,355	0,484	0,569

Як видно з таблиці, для кожного з трьох значень параметра n максимальна швидкість передачі РПШ Міхалевича-Імаї зростає зі зменшенням вимог до стійкості шифросистеми (при цьому знак «–» означає відсутність РПШ Міхалевича-Імаї із заданою стійкістю, оскільки отримані верхні межі

швидкості передачі є від'ємними). Перші додатні значення швидкості передачі відповідають граничному рівню стійкості, який може забезпечити відповідна РПШ Міхалевича-Імаї і є набагато меншими за швидкість РПШ з нелінійним випадковим кодуванням.

Дійсно, покажемо, що в класі запропонованих РПШ існують (практично реалізовні) рандомізовані шифросистеми, які забезпечують стійкість на рівні, зазначеному в табл. 4.1, та мають швидкість передачі $1/2$ (яка суттєво перевищує найбільшу можливу швидкість РПШ Міхалевича-Імаї при тій самій стійкості).

Згідно з методом, наведеним в розділі 3, побудуємо рандомізатор шифросистеми, використовуючи нелінійне відображення $\varphi(x) = x^{2^{l/2+l/4+1}}$, $x \in \mathbf{GF}(2^l)$. Оскільки $L_\phi{}^2 = 2^{2-l}$ [1], то на підставі твердження 3.4 інформаційна складність кореляційної атаки на РПШ з нелінійним випадковим кодуванням є не менше ніж $t_\theta = \theta^2 \cdot 2^{l-4}$. Крім того, оскільки $D_\phi = 2^{2-l}$, то часова складність атаки, наведеної в п. 3.2.2, є величиною порядку $2^l t_0$, де

$t_0 = \left\lceil \dfrac{\log(\left(1/2-\theta\right)^{-1} 2^l)}{l-2} \right\rceil + 1$ і, отже, є не менше ніж 2^l.

Таким чином, використовуючи в ролі φ зазначене вище відображення, отримаємо РПШ з нелінійним випадковим кодуванням, яка забезпечує стійкість відносно кореляційних атак, що є не менше ніж $t_\theta = \theta^2 \cdot 2^{l-4}$ та має швидкість передачі $l/2l = 1/2$. Зокрема, згідно з даними табл. 4.1, при $n = 512$ максимальна швидкість передачі РПШ Міхалевича-Імаї з рівнем стійкості 2^{79} дорівнює $0{,}004$, що у 125 разів менше швидкості передачі нелінійних РПШ (зі стійкістю 2^{249}). У випадку $n = 1024$ максимальна швидкість передачі РПШ Міхалевича-Імаї зі стійкістю 2^{154} дорівнює $0{,}001$, що у 500 разів менше швидкості передачі РПШ з нелінійним випадковим кодуванням (зі стійкістю 2^{505}).

4.1.2. **Порівняння РПШ за стійкістю при заданих обмеженнях щодо швидкості передачі та довжини шифрованих повідомлень.** Нехай задано такі параметри:

– значення швидкості передачі РПШ (дорівнює $1/2$);

– довжина $n = 2l$ шифрованих повідомлень;

– верхня межа p_e ймовірності помилкового відновлення відкритого повідомлення законним одержувачем інформації РПШ Міхалевича-Імаї (при цьому $p_e + H_2(p_e) < 1$, де H_2 – двійкова ентропійна функція; див. лему 2.2);

– число $0 < \theta < 1/2$, де $1/2 + \theta$ є ймовірністю успішного проведення кореляційної атаки на РПШ.

Треба для заданих вхідних параметрів оцінити зверху максимальну стійкість РПШ Міхалевича-Імаї та порівняти її зі стійкістю, що забезпечують при тих самих параметрах запропоновані РПШ з нелінійним випадковим кодуванням.

Використовуючи метод послідовних наближень, знайдемо найбільше значення $\log t$, для якого функція від p, що визначається виразами в правих частинах нерівностей (4.1) і (4.2), приймає значення не менше ніж $1/2$. Для обчислення стійкості РПШ з нелінійним випадковим кодуванням використаємо нижню межу $t_\theta = \theta^2 \cdot 2^{l-4}$, наведену в п. 4.1.1. Результати представлені в табл. 4.2.

Як видно з таблиці, в кожному з трьох випадків стійкість РПШ з нелінійним випадковим кодуванням є суттєво вище стійкості РПШ Міхалевича-Імаї, які виявляються практично нестійкими при зазначеній швидкості передачі $1/2$. Зокрема, при $n = 512$ і $n = 1024$ стійкість РПШ з нелінійним випадковим кодуванням вища за стійкість РПШ Міхалевича-Імаї у 2^{242} і 2^{487} разів відповідно.

Таблиця 4.2

Межі максимальної стійкості РПШ при заданих обмеженнях
щодо швидкості передачі ($l/n = 1/2$) та довжини шифрованих повідомлень
($p_e = 10^{-8}$, $\theta = 0{,}45$)

Довжина n шифрованого повідомлення	Двійковий логарифм верхньої межі інформаційної складності атаки на РПШ Міхалевича-Імаї	Двійковий логарифм нижньої межі інформаційної складності атаки на РПШ з нелінійним випадковим кодуванням
256	2	121
512	7	249
1024	18	505

4.2. Обґрунтування вибору компонент для побудови РПШ з нелінійним випадковим кодуванням

Розглянемо задачу практичної побудови конкретних варіантів РПШ з нелінійним випадковим кодуванням згідно з методом, запропонованим у попередньому розділі. З цією метою проведемо аналіз щодо вибору кожної складової компоненти у відповідності із зазначеними в розділі 3 вимогами до криптографічної стійкості відносно відомих атак та ефективності реалізації.

Нагадаємо, що за означенням (див. підрозділ 3.1), вхідними даними для побудови РПШ з нелінійним випадковим кодуванням є такі об‘єкти:

– відображення $\phi : V_{m-l} \to V_l$;

– комутативна групова операція $*$ на множині V_m;

– підстановочна матриця P порядку m;

– генератор гами, який виробляє за ключем $k \in K$ послідовність $f_0(k), f_1(k), \ldots$ булевих векторів довжини m.

Також до складу рандомізатора шифросистеми входить генератор випадкових послідовностей.

Надалі, виходячи з вимог практичності, вважатимемо, що $*$ є операцією покоординатного додавання за модулем 2, а P є тотожною матрицею порядку m. Пропозиції щодо вибору решти об‘єктів наведено нижче.

4.2.1. Вибір відображення ϕ. Як зазначено у розділі 3, для протидії відомим, а також іншим можливим атакам в ролі відображення $\phi : V_{m-l} \to V_l$ природно вибрати безключову геш-функцію. Таке рішення можна пояснити наступним чином.

Одним із розповсюджених на сьогодні підходів до обґрунтування стійкості шифросистем є використання припущень, притаманних ідеалізованим моделям. Широко поширеним на практиці прикладом такої моделі є *модель випадкового оракулу*, яка постулює існування загальнодоступної випадково вибраної безключової функції H, значення якої можна обчислити тільки шляхом надіслання запиту до оракулу [2]. Випадковий оракул можна представити у вигляді «чорної скрині», яка повертає значення $H(x)$ при отриманні на вхід аргумента x. Вважається, що найбільш близькою практичною реалізацією випадкового оракулу є криптографічна стійка безключова геш-функція. Модель випадкового оракулу дозволяє будувати значно більш ефективні шифросистеми, ніж ті, що будуються за стандартних припущень (при заданій стійкості). При цьому на сьогодні не відомо будь-яких реалізовних атак на шифросистеми, які є обґрунтовано стійкими в моделі випадкового оракулу [2].

Обґрунтування стійкості певної шифросистеми в моделі випадкового оракулу свідчить про правильність її побудови в тому сенсі, що єдиними її можливими слабкостями є слабкості геш-функції, на якій базується реалізація випадкового оракулу. Відповідно вибір криптографічно стійкої геш-функції є певною гарантією стійкості такої шифросистеми. Більш того, у разі появи

успішної атаки на шифросистему відбувається заміна діючої геш-функції на більш стійкий варіант.

Зауважимо, що на підставі рівності (3.1) запропоновані РПШ з нелінійним випадковим кодуванням є безумовно стійкими відносно будь-яких атак на основі підібраних векторів ініціалізації в моделі випадкового оракула. Дійсно, якщо $\phi : V_{m-l} \to V_l$ є випадковим рівноймовірним відображенням, то для будь-якого фіксованого (відомого) відкритого тексту $s_0, s_1, \ldots$ випадкові вектори $(u_i, s_i \oplus \phi(u_i))$, $i = 0, 1, \ldots$ є незалежними в сукупності та мають рівномірний розподіл ймовірностей на множині V_m. В цьому випадку шифротекст $z_0, z_1, \ldots$ є результатом накладання випадкової рівноймовірної "гами" вигляду $(u_0, s_0 \oplus \phi(u_0))$, $(u_1, s_1 \oplus \phi(u_1))$, … на послідовність $f_0(k), f_1(k), \ldots$, причому для зашифровування кожного нового відкритого тексту використовується нова суто рівноймовірна "гама". Звідси випливає, що кількість інформації про невідому послідовність $f_0(k), f_1(k), \ldots$ (а, отже, і про ключ k), яка міститься у будь-якій сукупності відкритих та відповідних їм шифрованих текстів, дорівнює нулю, тобто РПШ, що розглядається, є безумовно стійкою відносно атак на основі підібраних векторів ініціалізації.

Таким чином, на підставі зазначеного пропонується використовувати в ролі відображення ϕ одну з відомих обчислювально стійких геш-функцій, а саме, "Купину" або Keccak.

Ітеративна геш-функція "Купина" введена в дію у 2015 році як національний стандарт України ДСТУ 7564:2014 "Інформаційні технології. Криптографічний захист інформації. Функція гешування" [3]. Стандартом визначено як саму геш-функцію, так і її додатковий режим використання – генерацію коду автентифікації повідомлень. Ця функція прийшла на заміну застарілому алгоритму ГОСТ 34.311-95 [4], програмна реалізація якого є значно повільнішою в порівнянні з сучасними рішеннями. Крім того, на геш-функцію ГОСТ 34.311-95 відомі атаки на основі колізій [5], які є ефективнішими за атаку

повного перебору. Основними вимогами до нової геш-функції є високий рівень стійкості відносно існуючих атак та висока швидкодія програмної реалізації на 64-бітових процесорах загального призначення.

Структурно геш-функція “Купина” (рис. 4.1) є подібною до геш-функції Groestl [6], що є фіналістом конкурсу NIST на новий стандарт алгоритму гешування SHA-3, але відрізняється від нього у певних деталях раундових перетворень: деякі раундові константи “Купини” додаються за модулем 2^{64} замість порозрядного додавання за модулем 2, що використовується у Groestl. Така відмінність запобігає безпосередньому застосуванню нещодавно винайдених rebound-атак на функцію стиснення AES-подібних геш-конструкцій [7 – 9]. Підстановки, що використовуються в алгоритмі “Купина”, побудовані за допомогою перетворень, застосованих у блоковому шифрі “Калина”, визначеному як український стандарт ДСТУ 7624:2014 [10]. “Купина” дозволяє виробляти геш-значення довжиною від 8 до 512 бітів (з кроком 8 бітів). Варіант функції, який повертає n бітів, позначається “Купина-n”. Для практичних цілей рекомендовано використовувати наступні варіанти функцій: “Купина-256”, “Купина-384” та “Купина-512” [11].

В [3] зазначено, що різницеві атаки, а також rebound-атаки на цю геш-функцію є неефективними вже після чотирьох ітерацій. В результаті проведення незалежного криптоаналізу [12, 13] також не виявлено вразливостей геш-функції “Купина”. Зокрема, в [12] з використанням rebound-атаки представлена атака на основі колізій на 5 раундів геш-функції “Купина-256”. Часова і ємнісна складності цієї атаки дорівнюють 2^{120} і 2^{64} відповідно. Більш того, побудовано модифіковану версію атаки “зустріч посередині” для отримання псевдо-прообразу на 6 раундів “Купини-256” і на 8 раундів “Купини-512”. Часова і ємнісна складності цих атак складають відповідно $2^{250,33}$ і $2^{250,33}$ для “Купини-256”, а також $2^{498,33}$ і $2^{498,33}$ для “Купини-512”.

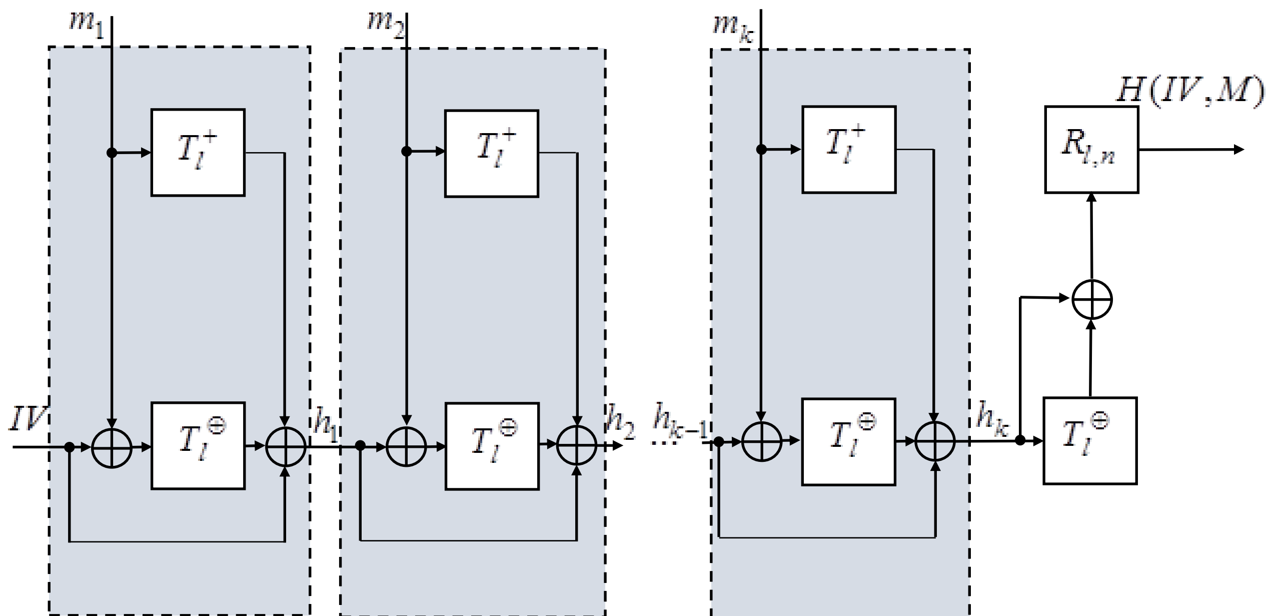


Рис. 4.1. Схема геш-функції “Купина” [3]

В [13] описані атаки колізій на функцію стиснення “Купини-256” для 6 раундів зі складністю 2^{70} і для 7 раундів зі складністю $2^{125,8}$. До того ж, з використанням rebound-атаки проведено атаку на основі колізій на редуковану версію власно геш-функції, складність якої для 4 і 5 раундів “Купини-256” дорівнює 2^{67} і 2^{120} відповідно.

Таким чином, геш-функція “Купина” є на сьогодні обчислювально стійкою відносно усіх відомих атак та може використовуватися в ролі відображення ϕ в конструкції рандомізатора РПШ з нелінійним випадковим кодуванням.

Розглянемо зараз інший можливий варіант для застосування в ролі відображення ϕ – геш-функцію Keccak [14], яка є переможцем конкурсу на стандарт нового геш-алгоритму SHA-3, що проводився Національним інститутом стандартів і технологій США NIST упродовж 2007 – 2012 років. Новий алгоритм затверджено у 2015 році та опубліковано як стандарт США FIPS 202 [15]. Геш-функція Keccak реалізована на базі конструкції, що називається “криптографічна губка” (рис. 4.2). Дана конструкція вперше представлена у 2007 році на симпозіумі ECRYPT Hash Function як альтернатива

традиційній конструкції Меркля-Дамгорда [16] і являє собою відображення вхідних даних довільної довжини у вихідні дані також довільної довжини.

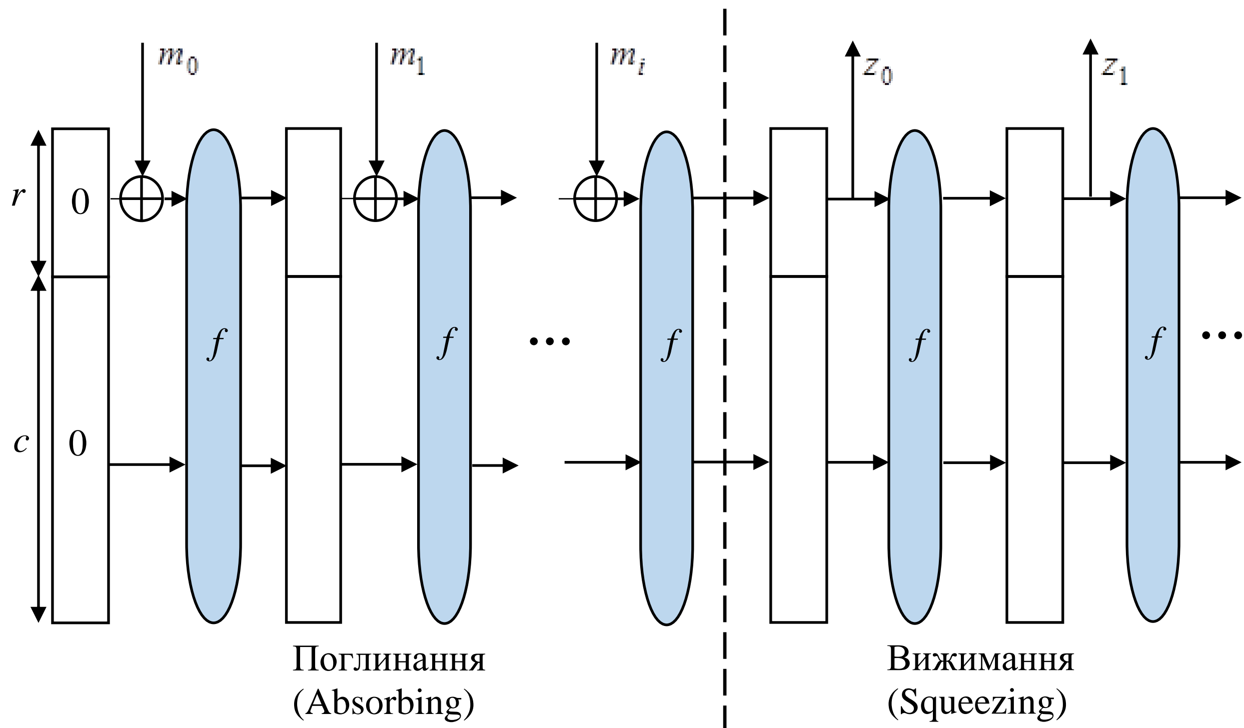


Рис. 4.2. Конструкція "криптографічна губка" [16]

Як видно з рис. 4.2, основу конструкції геш-функції Keccak складає багатораундова безключова псевдовипадкова функція f, що виконує перемішування внутрішнього стану розрядністю $b = r + c$, де r називається бітовою швидкістю, а c – потужністю. На початковому етапі, як і в конструкції Меркля-Дамгорда, вхідні дані розширюються у відповідності із заданим алгоритмом (до довжини, кратної r), після чого розбиваються на блоки m_i по r бітів кожний. Далі відбувається процедура ініціалізації внутрішнього стану нульовим значенням [15].

Процедура вироблення геш-значення складається із двох етапів. Під час першого етапу, який називається "поглинанням" (absorbing), r-бітові блоки повідомлення m_i додаються за модулем 2 до r бітів внутрішнього стану –

результату перетворення f. На другому етапі, який називається “вижиманням” (squeezing), відбувається формування вихідного геш-значення. При цьому результат першого етапу довжини b знову поступає на вхід перетворення f, з виходу якого зчитуються тільки перші r бітів. Даний процес повторюється кінцеву кількість разів, на кожному з яких новий r-бітовий блок z_i доповнює попередні блоки до отримання потрібної довжини геш-значення.

Для прийнятої як стандарт геш-функції Keccak довжина внутрішнього стану дорівнює 1600 бітів (при $r = 576$ бітів та $c = 1024$ бітів) з розміром геш-значення у 512 бітів. Кількість раундів у перетворенні f дорівнює 24. Крім того, розробниками рекомендовано до використання додаткові режими:

- $r = 1152$ бітів, $c = 448$ бітів – для вироблення геш-значення довжиною 224 біти;

- $r = 1088$ бітів, $c = 512$ бітів – для вироблення геш-значення довжиною 256 бітів (цей режим застосовано для програмної реалізації РПШ);

- $r = 1152$ бітів, $c = 448$ бітів – для вироблення геш-значення довжиною 384 біти.

При використанні геш-функції Keccak в ролі відображення $\phi : V_{m-l} \to V_l$ для побудови РПШ з нелінійним випадковим кодуванням параметр l дорівнює довжині геш-значення, а параметр $m = 2l$.

Результати проведених під час конкурсу та після нього незалежних криптоаналітичних досліджень підтверджують практичну стійкість геш-функції Keccak. Так, складність розрізнювальної атаки на всі 24 раунди перетворення f в 2010 р. складала 2^{1590} [17] і покращена в 2011 р. до 2^{1579} [18]. В травні 2014 р. на сайті конкурсу, присвяченого результатам криптоаналізу Keccak, заявлено про можливість застосування кубічної атаки на 6 раундів геш-функції (з параметрами $r = 1024$ і $c = 576$) [19]. В режимі потокового шифру процедура відновлення 128-бітового ключа має складність 2^{36}, а в режимі вироблення імітовставки складність відновлення 80-бітового ключа дорівнює 2^{35}. В травні

2016 року з‘явилося повідомлення [20] про знаходження колізії для 5 раундів Keccak з параметрами $r=1440$, $c=160$, а в грудні 2016 року для редукованого до 4 раундів Keccak з тими самими параметрами заявлено про знаходження прообразу для 80-бітового геш-значення [20]. Останній результат датовано березнем 2017 р. [21], де йдеться про знаходження колізії вже для 6 раундів Keccak з параметрами $r=1440$ і $c=160$. Зазначені результати свідчать про практичну стійкість цієї геш-функції відносно усіх відомих сьогодні атак.

Поряд з розглянутими геш-функціями, альтернативним варіантом є застосування в ролі $\phi: V_{m-l} \to V_l$ нелінійного відображення з обґрунтованими криптографічними властивостями, які зазначені в розділі 3:

– велике значення параметра $m-l$ для протидії перебірній атаці (див. алгоритм 3.1);

– мале значення параметра (3.15) для протидії атакам лінійного типу (див. алгоритм 3.2);

– велика часова складність розв‘язання системи рівнянь (3.16).

Крім того, для підвищення практичності РПШ величина l повинна бути також достатньо великою.

На сьогодні відомо чимало видів відображень (які задаються поліномами над полем $\mathbf{GF}(2^l)$ при $m=2l$), що задовольняють першим двом з наведених вимог, наприклад, $\varphi(x)=\sum_{i=0}^{2^l-3} x^i$, $\varphi(x)=x^{2^l-2}$, $\varphi(x)=x^{2^{l/2+l/4+1}}$, $x \in \mathbf{GF}(2^l)$. Для кожного з них справедлива рівність ${L_\phi}^2 = 2^{2-l}$ [1, 22, 23], що свідчить про малість параметра (3.15) при $l \geq 256$. Зауважимо, що друге з наведених відображень не задовольняє останній вимозі [23], однак питання про те, чи стосується це решти відображень потребує окремого дослідження. Останнє з наведених відображень використано у програмній реалізації РПШ з нелінійним випадковим кодуванням.

4.2.2. Вибір генератора гами. В якості генератора гами пропонується використовувати криптографічний алгоритм SNOW 2.0 [24], що являє собою слово-орієнтований синхронний потоковий шифр (з довжиною слова 32 біти), запропонований у 2002 р. як альтернатива попередньої (більш слабкої) версії – SNOW. На сьогодні цей шифр є стандартизованим в ISO/IEC [25] і розглядається як прототип майбутнього національного стандарту потокового шифрування України.

SNOW 2.0 (рис. 4.3) являє собою один з найбільш швидких програмно- і апаратно-орієнтованих потокових шифрів. Висока швидкодія досягається використанням мінімального набору операцій (додавання слів за модулями 2 та 2^{32} відповідно, байтового зсуву слова та звернення до таблиці), які є доступними в сучасних процесорах. Довжина ключа SNOW 2.0 становить 128 або 256 бітів (4 або 8 слів відповідно), а довжина вектору ініціалізації дорівнює 128 бітів (4 слова).

Стійкість шифру SNOW 2.0 до алгебраїчних атак досліджено в [26], де показано, що модифікована версія SNOW 2.0, в якій додавання за модулем 2^{32} замінені покоординатними булевими додаваннями, є дуже вразливою до алгебраїчної атаки. Часова складність такої атаки дорівнює 2^{50} операцій і потребує не більше 1000 слів. Крім того, показано, що SNOW 2.0 може бути описаний за допомогою системи перевизначених квадратичних рівнянь, яку в принципі можна розв’язати, використовуючи приблизно 17 знаків гами.

На повну версію шифру SNOW 2.0 відомі ефективні атаки зі зв‘язаними ключами (related key attacks), які базуються на існуванні ключів, еквівалентних із затримкою [27]. У роботі [28] представлена розрізнювальна атака на SNOW 2.0 з використанням лінійного маскувального методу [29]. Ця атака потребує 2^{225} слів (2^{230} бітів) і 2^{225} операцій зашифровування для розрізнення виходу шифру SNOW 2.0 від суто випадкової послідовності. Зазначений результат покращено у [30], де показано,

що для SNOW 2.0 існує лінійний розрізнювач, який потребує 2^{174} слів (2^{179} бітів) і 2^{174} операцій зашифровування.

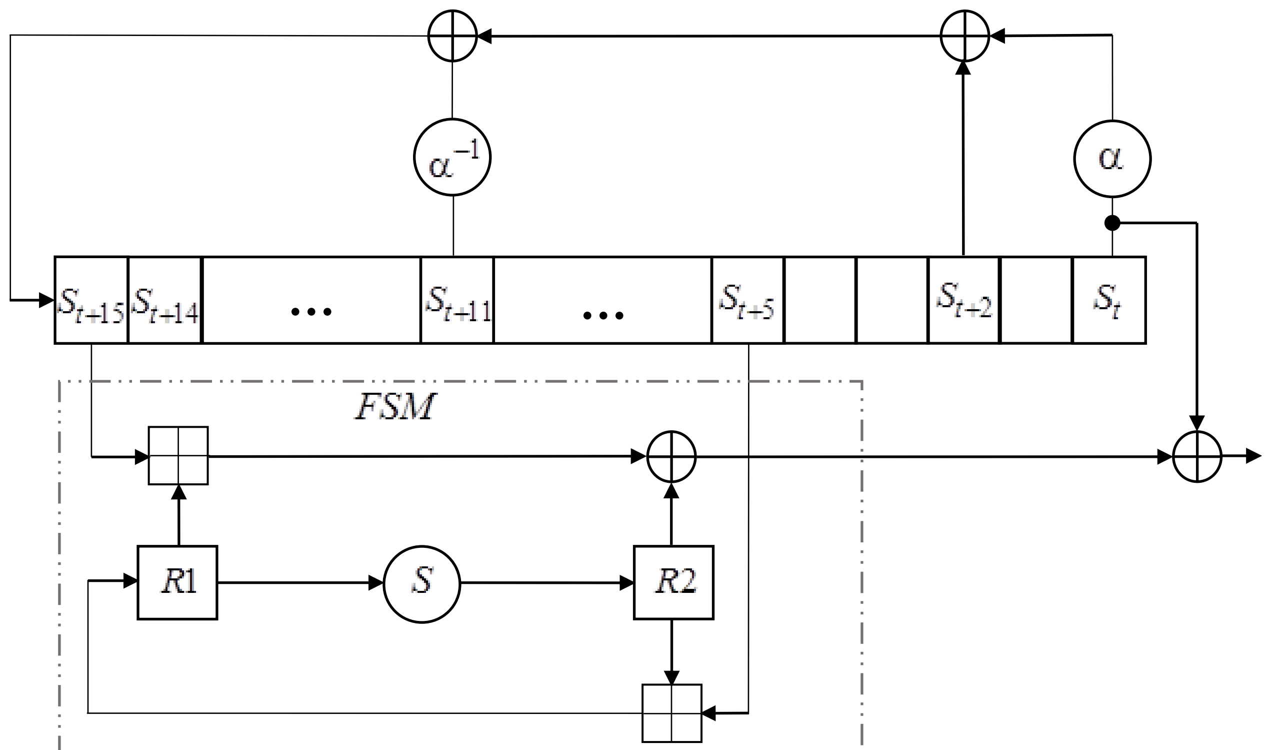


Рис. 4.3. Схема генератора гами потокового шифру SNOW 2.0 [25]

4.2.3. Вибір генератора випадкових послідовностей. Для підвищення ефективності програмних реалізацій РПШ з нелінійним випадковим кодуванням будемо моделювати генератор випадкових послідовностей за допомогою генератора псевдовипадкових чисел, в ролі якого виберемо алгоритм ISAAC (Indirection, Shift, Accumulate, Add and Count) [31].

Цей алгоритм спроектовано у відповідності до наступних вимог:

- криптографічна стійкість відносно усіх відомих атак;
- неможливість отримання внутрішнього стану за наявними вихідними результатами;
- відсутність коротких циклів;

- відсутність будь-яких тенденцій розподілу бітів на всьому циклі;
- упорядковані стани повинні швидко ставати хаотичними.

На відміну від багатьох генераторів псевдовипадкових чисел, алгоритм ISAAC розроблено без використання лінійних регістрів зсуву. Середня кількість машинних інструкцій, потрібних для отримання 32-бітового значення дорівнює 18,75, а 64-бітова версія ISAAC вимагає 19 інструкцій для отримання одного 64-бітового значення.

Цей генератор відносять до криптографічно стійких генераторів псевдовипадкових чисел, про що свідчать результати наявних робіт стосовно його криптоаналізу. Так, в [32] описано атаку на основі відомих відкритих текстів, з допомогою якої можна знайти початковий стан генератора за невеликим сегментом вихідних даних. Складність такої атаки дорівнює $4{,}67 \cdot 10^{1240}$, в той час як складність повного перебору становить $5{,}91 \cdot 10^{2446}$. В роботі [33] описано чотири множини «слабких» початкових станів, які можуть перетинатися між собою. Слабкими вважаються такі стани, для яких частина елементів є випадковими, а решта елементів співпадають між собою. Зважаючи на це, в [33] запропоновано модифіковану версію ISAAC, а саме, ISAAC+.

Програмні реалізації алгоритму ISAAC працюють достатньо швидко і надійно, тому цей генератор псевдовипадкових чисел використовується, наприклад, в Unix-утиліті *shred* [34] для зашифровування перезаписаних даних. Цей генератор реалізовано також в одній з найбільш розповсюджених бібліотек мови програмування Java – Apache Commons Math [35].

Таким чином, обрані вище компоненти відповідають зазначеним в розділі 3 вимогам до криптографічної стійкості відносно відомих атак та ефективності реалізації, і можуть бути використані для практичної побудови конкретних варіантів РПШ з нелінійним випадковим кодуванням.

4.3. Результати дослідження ефективності програмних реалізацій РПШ з нелінійним випадковим кодуванням

З метою оцінки на практиці ефективності РПШ з нелінійним випадковим кодуванням автором дисертації розроблено та програмно реалізовано конкретні варіанти шифросистем і наведено результати порівняння їх роботи. Показано, що побудовані шифросистеми можуть бути застосовані на практиці для зашифровування/розшифровування даних в режимі реального часу. Реалізації зазначених шифросистем виконано у таких варіантах (ГЕНЕРАТОР ВИПАДКОВИХ ЧИСЕЛ – ВІДОБРАЖЕННЯ ϕ – ГЕНЕРАТОР ГАМИ):

1) “Isaac – Keccak – Snow 2.0” (IKCS);

2) “Isaac – “Купина” – Snow 2.0” (IKPS);

3) “Isaac – NonLinearMap – Snow 2.0” (INLS), де елемент NonLinearMap означає нелінійне відображення $\varphi(x) = x^{2^{l/2+l/4+1}}$, $x \in \mathbf{GF}(2^l)$.

В кожному з трьох варіантів реалізацій вибрано такі значення параметрів: $l = 256$, $m = 512$.

Криптографічна схема першого варіанту реалізації РПШ з нелінійним випадковим кодуванням наведена на рис. 4.4. Решта варіантів будуються аналогічним чином. Крім того, з використанням криптографічного пакету OpenSSL (з відкритим програмним кодом) [36] реалізовано алгоритм AES у режимі зворотного зв‘язку за виходом. Вихідні тексти комп’ютерних програм для всіх зазначених реалізацій наведено в додатках.

Для перевірки коректності реалізації шифросистем кожен із складових елементів було попередньо відлагоджено на тестових наборах вхідних/вихідних даних, що представлені на відповідних web-сторінках з описами цих алгоритмів.

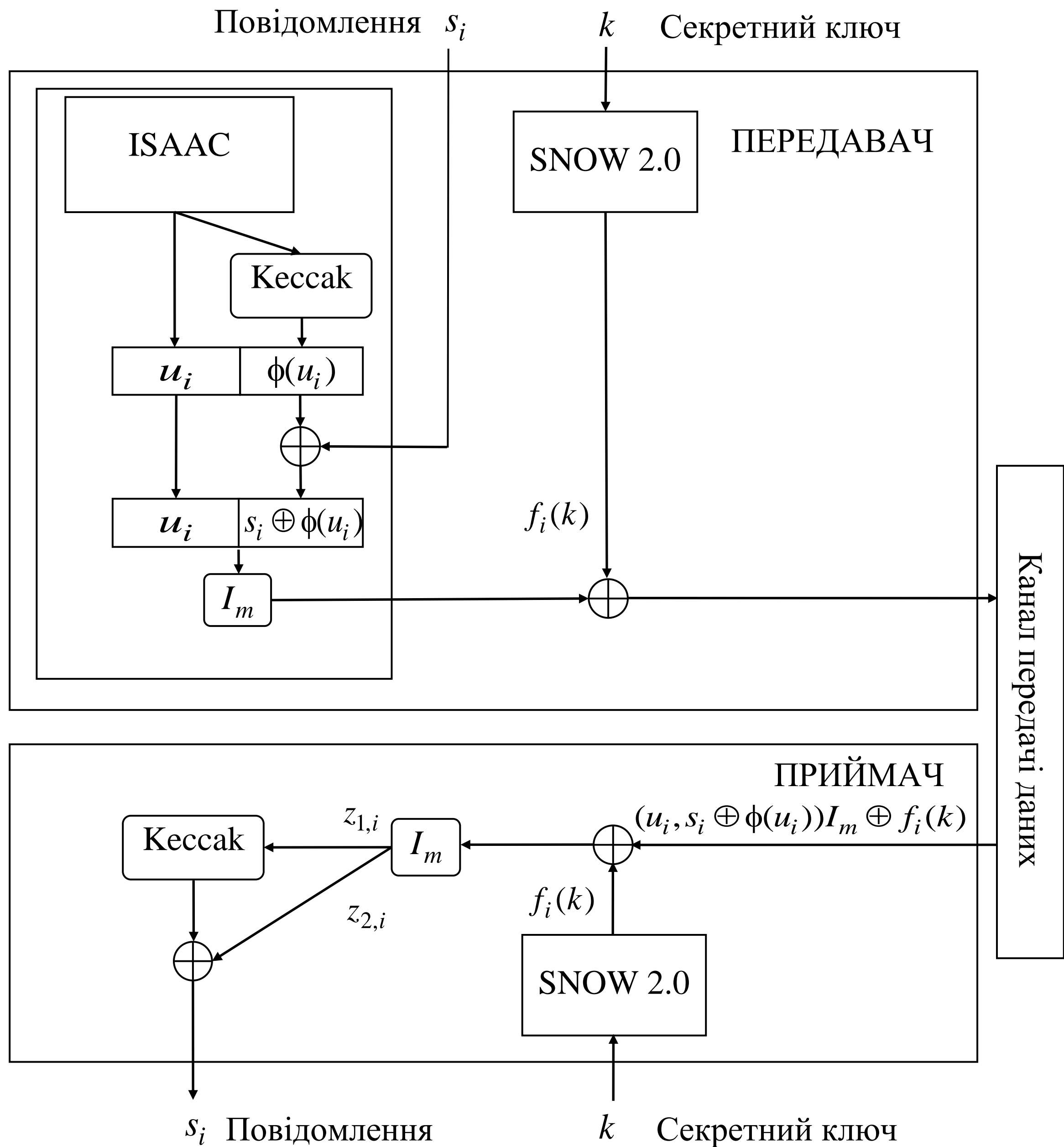


Рис. 4.4. Схема РПШ з нелінійним випадковим кодуванням IKCS

Час виконання процедур зашифровування/розшифровування повідомлень з використанням розроблених реалізацій РПШ з нелінійним випадковим кодуванням наведено в табл. 4.3. Всі реалізації виконувались на обчислювальній системі з процесором Intel(R) Core(TM) i3-6100, 3.7GHz та обсягом оперативної пам’яті 4 ГБ на базі 64-розрядної ОС Windows 7 Service

Pack 1. Мова програмування – C++. Середовище розробки – Microsoft Visual Studio 2013 (Release версія).

Таблиця 4.3

Типовий приклад, що ілюструє час виконання процедур зашифровування/розшифровування для РПШ з нелінійним випадковим кодуванням та алгоритму AES

Довжина файлу даних	IKCS	IKPS	INLS	AES
Файл 1 – 3454 б	0,011 сек.	0,152 сек.	0,735 сек.	0,001 сек.
Файл 2 – 180 Кб	0,106 сек.	6,905 сек.	32,160 сек.	0,016 сек.
Файл 3 – 1162 Кб	0,526 сек.	45,122 сек.	208,228 сек.	0,023 сек.

Як видно з таблиці, найбільш ефективною серед рандомізованих шифросистем за часом виконання процедур зашифровування/розшифровування виявляється РПШ IKCS, в якій відображення ϕ реалізовано геш-функцією Keccak. Так, ефективність цієї шифросистеми (при зашифровуванні/розшифровуванні файлу 2 розміром 180 Кб) в 65 разів перевищує ефективність шифросистеми IKPS та в 303 рази ефективність шифросистеми INLS. Для файлу 3 розміром 1162 Кб ефективність шифросистеми IKCS у порівнянні з шифросистемами IKPS та INLS ще вища, у 86 та 395 разів відповідно. Це свідчить про необхідність ретельного підбору компонентів для реалізації конкретного варіанту РПШ з нелінійним випадковим кодуванням. Відмітимо, що система IKPS (на базі геш-функції “Купина”) також є достатньо практичною і може бути використана для захисту в режимі реального часу файлів даних невеликого розміру (наприклад, текстових файлів). Висока ефективність реалізації алгоритму AES з допомогою криптографічного пакету OpenSSL [36] пов’язана, певно, з дуже гарною програмною оптимізацією вбудованих функцій.

Зауважимо, що уповільнення в часі при зашифровуванні/розшифровуванні (в порівнянні з програмною реалізацією алгоритму AES в режимі зворотного зв‘язку за виходом), яке відбувається за рахунок нелінійного випадкового кодування, компенсується суттєвим підвищенням стійкості шифросистем відносно відомих атак незалежно від криптографічних властивостей вхідного потокового шифру (у даному випадку SNOW 2.0). Так, РПШ IKCS та IKPS є безумовно стійкими в моделі випадкового оракула та забезпечують практичну стійкість на рівні 2^{256} операцій (за умови практичної стійкості геш-функцій Keccak та “Купина”). Стійкість РПШ INLS відносно найбільш потужної на сьогодні атаки на основі підібраних векторів ініціалізації є не менше ніж 2^{249} операцій (див. табл. 4.2).

В цілому, отримані результати свідчать про помітну перевагу (як з погляду стійкості, так і практичності) побудованих РПШ в порівнянні з рандомізованими потоковими шифросистемами Міхалевича-Імаї.

Висновки

1. При фіксованій стійкості та однаковій довжині n шифрованого повідомлення побудовані рандомізовані шифросистеми з нелінійним випадковим кодуванням мають значно більшу швидкість передачі в порівнянні з РПШ Міхалевича-Імаї. Так, при $n = 512$ максимальна швидкість передачі РПШ Міхалевича-Імаї зі стійкістю 2^{79} дорівнює $0{,}004$, що у 125 разів менше швидкості передачі нелінійних РПШ (зі стійкістю 2^{249}). У випадку $n = 1024$ максимальна швидкість передачі РПШ Міхалевича-Імаї зі стійкістю 2^{154} дорівнює $0{,}001$, що у 500 разів менше швидкості передачі РПШ з нелінійним випадковим кодуванням (зі стійкістю 2^{505}); див. табл. 4.1.

2. Результати порівняння стійкості двох зазначених видів шифросистем при фіксованій швидкості передачі (що дорівнює $1/2$) також свідчать про суттєву перевагу РПШ з нелінійним випадковим кодуванням над РПШ Міхалевича-Імаї, які виявляються практично нестійкими при зазначеній швидкості. Зокрема, при $n=512$ і $n=1024$ стійкість РПШ з нелінійним випадковим кодуванням вища за стійкість РПШ Міхалевича-Імаї у 2^{242} і 2^{487} разів відповідно (див. табл. 4.2).

3. Аналіз вибору компонент для практичної побудови конкретних варіантів РПШ з нелінійним випадковим кодуванням показує, що в ролі нелінійного відображення в конструкції рандомізотора доцільно використовувати одну з відомих обчислювально стійких геш-функцій, наприклад, “Купину” або Keccak. Побудовані таким чином РПШ є безумовно стійкими відносно будь-яких атак на основі підібраних векторів ініціалізації в моделі випадкового оракула та забезпечують (на сьогодні) практичну стійкість на рівні 2^l операцій за умови практичної стійкості геш-функцій Keccak-l та “Купина”-l.

4. Розроблені програмні реалізації трьох варіантів РПШ з нелінійним випадковим кодуванням дозволяють здійснювати на практиці процедури зашифровування/розшифровування даних в режимі реального часу. При цьому найбільш ефективною за часом виконання є шифросистема IKCS (ISAAC – Keccak – Snow 2.0). Зокрема, час зашифровування/розшифровування файлів розміром 180 Кб за допомогою цієї шифросистеми є в 65 та 303 рази менше часу зашифровування/розшифровування файлів такого ж розміру за допомогою шифросистем IKPS та INLS відповідно (табл. 4.3).

Список використаних джерел у четвертому розділі

1. Bracken C., Leander G. «A highly nonlinear differentially 4 uniform power mapping that permutes fields of even degree», Finite Fields and Their Applications, V. 16, No. 4, 2010, P. 231-242.

2. Katz J., Lindell Y. «*Introduction to modern cryptography*», CRC PRESS, 2007.

3. Олійников Р., Горбенко І., Казимиров О., Руженцев В., Бойко А., Кузнєцов О., Горбенко Ю., Долгов В., Дирда О., Пушкарьов А. «*ДСТУ 7564:2014. Національний стандарт України. Інформаційні технології. Криптографічний захист інформації. Функція гешування*», 2015.

4. *ГОСТ 34.311 – 95. Інформаційна технологія. Криптографічний захист інформації. Функція хешування*, Введ. 1995-01-01, К: Держспоживстандарт, 1994, 23 с.

5. Mendel F., Pramstaller N., Rechberger C., Kontak M., Szmidt J. «Cryptanalysis of the GOST hash function», *Advances in Cryptology – CRYPTO 2008*, *LNCS*, V. 5157, 2008, P. 162-178.

6. Gauravaram P., Knudsen L., Matusiewicz K., Mendel F., Rechberger C., Schlaffer M., Thomsen S. «Groestl – a SHA-3 candidate», *Submission to NIST*, 2009, URL: https://www.groestl.info.

7. Mendel F., Rechberger C., Schlaffer M., Thomsen S. «The rebound attack: Cryptanalysis of reduced Whirlpool and Groestl», *Fast Software Encryption – FSE 2009*, *LNCS*, V. 5665, 2009, P. 260-276.

8. Gilbert H., Peyrin T. «Super-Sbox crypanalysis: Improved attacks for AES-like permutations», *Fast Software Encryption – FSE 2010*, *LNCS*, V. 6147, 2010, P. 365-383.

9. Jean J., Naya-Plasencia M., Peyrin T. «Improved rebound attack on the finalist Groestl», *Fast Software Encryption – FSE 2012*, *LNCS*, V. 7549, 2012, P. 110-126.

10. Олійников Р., Горбенко І., Казимиров О., Руженцев В., Кузнєцов О., Горбенко Ю., Бойко А., Долгов В., Дирда О., Пушкарьов А., Мордвинов Р., Кайдалов Д. «*ДСТУ 7624:2014. Національний стандарт України. Інформаційні технології. Криптографічний захист інформації. Алгоритм симетричного блокового перетворення*», 2015.

11. Oliynykov R., Gorbenko I., Kazymyrov O., Ruzhentsev V., Kuznetsov O., Gorbenko Y., Boiko A., Dyrda O., Dolgov V., Pushkaryov A. «A new standard of Ukraine: The Kupyna hash function», *Cryptology ePrint Archive*, 2015, URL: https://eprint.iacr.org/2015/885.

12. Zou J., Dong L. «Cryptanalysis of the round-reduced Kupyna hash function», *Cryptology ePrint Archive*, 2015, URL: http://eprint.iacr.org/2015/959.pdf.

13. Dobrauning C., Eichlseder M., Mendel F. «Analysis of the Kupyna-256 hash function», *Fast Software Encryption – FSE 2016*, *LNCS*, V. 9783, 2016, P. 575-590.

14. Bertoni G., Daemen J., Peeters M., Van Assche G. «Keccak sponge function family main document», URL: https://keccak.team/obsolete/Keccak-main-1.0.pdf.

15. National Institute of Standards and Technology «SHA-3 Standard: Permutation-based hash and extendable-output functions», URL: https://csrc.nist.gov/Projects/Hash-Functions/SHA-3-Project/SHA-3-Standardization.

16. Bertoni G., Daemen J., Peeters M., Van Assche G., Van Keer R. «The sponge and duplex constructions», URL: https://keccak.team/sponge_duplex.html.

17. Boura C., Canteaut A., De Canniere C. «Higher order differential properties of Keccak and Luffa», *Fast Software Encryption – FSE 2011*, *LNCS*, V. 6733, 2011, P. 252-269.

18. Duan M., Lai X. «Improved zero-sum distinguisher for full round Keccak-f permutation», *Chinese Science Bulletin*, V. 57, No. 6, 2012, P. 694-697.

19. Dinur I., Morawiecki P., Pieprzyk J., Srebtny M., Straus M. «Practical complexity cube attacks on round-reduced Keccak sponge function», *Cryptology ePrint Archive*, 2014, URL: https://eprint.iacr.org/2014/259.pdf.

20. Guo J., Liu M., Song L., Qiao K. «The Keccak crunchy crypto collision and pre-image contest», 2016, URL: https://keccak.team/crunchy_contest.html.

21. Mella S., Daemen J., Van Assche G. «New techniques for trail bounds and application to differential trails in Keccak», *IACR Transactions on Symmetric Cryptology*, V. 2017, No. 1, 2017, P. 329-357.

22. Yu Y., Wang M., Li Y. «Constructing low differential uniformity functions from known ones», *Chinese Journal of Electronics*, V. 22, No. 3, 2013, P. 495-499.

23. Nyberg K. «Differentially uniform mappings for cryptography», *Advances in Cryptography-EUROCRYPT'93*, *LNCS*, V. 765, 1994, P. 55-64.

24. Ekdahl P., Johansson T. «A new version of the stream cipher SNOW», *Selected Areas in Cryptography – SAC 2002*, LNCS 2295, 2002, P. 47-61.

25. ISO/IEC 18033-4: 2011(E). *Information technology – Security techniques – Encryption algorithm – Part 4: Stream ciphers*, 2011, 92 p.

26. Billet O., Gilbert H. «Resistance of SNOW 2.0 against algebraic attacks», *CT – RSA 2005*, V. 3376, 2005, P. 19-28.

27. Kircanski A., Youssef A. «On the sliding property of SNOW 3G and SNOW 2.0», *IET Information Security*, V. 5, No. 4, 2011, P. 199-206.

28. Watanabe D., Biryukov A., De Canniere C. «A distinguishing attack of SNOW 2.0 with linear masking method», *Selected Areas in Cryptography – SAC 2003*, V. 3006, 2004, P. 222-233.

29. Coppersmith D., Halevi S., Jutla C. «Cryptanalysis of stream ciphers with linear masking», *Advances in Cryptology*, *CRYPTO 2002*, LNCS 2442, 2002, P. 515-532.

30. Nyberg K., Wallen J. «Improved linear distinguishers for SNOW 2.0», *Fast Software Encryption – FSE 2006*, V. 4047, 2006, P. 144-162.

31. Jenkins R.J. «ISAAC», *Fast Software Encryption – FSE 1996*, V. 1039, 1996, P. 41-49.

32. Pudovkina M. «A known plaintext attack on the ISAAC keystream generator», *Cryptology ePrint Archive*, 2001, URL: https://eprint.iacr.org/2001/049.pdf.

33. Aumasson J.-P. «On the pseudo-random generator ISAAC», *Cryptology ePrint Archive*, 2006, URL: https://eprint.iacr.org/2006/438.pdf.

34. Richmond G. «Shred and secure-delete: tools for wiping files, partitions and disks in GNU/Linux», *Free Software Magazine*, 2008, URL:

http://freesoftwaremagazine.com/articles/shred_and_secure_delete_tools_wiping_files_partitions_and_disks_gnu_linux.

35. «Commons Math: The Apache Commons Mathematics Library», URL: http://commons.apache.org/proper/commons-math.

36. «OpenSSL. Cryptography and SSL/TLS Toolkit», URL: https://www.openssl.org.

ВИСНОВКИ

З огляду на зростання рівня зовнішніх загроз національній безпеці та суверенітету України, особливої гостроти набувають задачі забезпечення інформаційної безпеки держави. Одним із основних напрямків вирішення цих задач є створення нових та удосконалення існуючих методів криптографічного захисту інформації, спрямованих на забезпечення конфіденційності, цілісності, справжності та доступності інформації. Широку розповсюдженість для захисту інформації в спеціальних інформаційно-телекомунікаційних системах отримали потокові шифри. Це обумовлено, головним чином, надвисокою швидкістю потокового шифрування на канальному, мережевому та транспортному рівнях, а також придатністю потокових шифрів для застосування у пристроях з обмеженими обчислювальними ресурсами або з малим споживанням електроенергії.

Неухильний розвиток інформаційних технологій поряд з нарощуванням потужності обчислювальних засобів та появою нових (або підсиленням відомих) методів криптоаналізу створює потенційну загрозу для спеціальних (військових) додатків, в яких використовуються потокові шифри. Оскільки швидка заміна алгоритму шифрування, криптографічні слабкості якого виявлені на етапі його експлуатації (як правило, через багато років після його створення) є практично неможливою, видається доцільним створення методів підвищення стійкості потокових шифрів без внесення змін в алгоритми шифрування шляхом застосування додаткових перетворень, які не потребують ключів, можуть бути відносно просто реалізовані та забезпечують науково обґрунтований рівень стійкості систем шифрування в цілому.

Одним з таких загальних методів є рандомізація або випадкове кодування джерела відкритих повідомлень. Єдиним відомим прикладом рандомізованих потокових шифросистем (РПШ), які будуються на регулярній основі та, в

принципі, можуть бути використані на практиці, є шифросистеми Міхалевича-Імаї. Проте стійкість РПШ Міхалевича-Імаї суттєво залежить від будови їх компонент і може бути значно менше, ніж стверджують їх розробники. Деякі з цих шифросистем виявляються вразливими навіть до атак на основі відомих шифрованих повідомлень і, отже, не можуть бути використані для захисту державних інформаційних ресурсів.

В дисертаційній роботі вирішено **актуальну наукову задачу** розробки методу побудови рандомізованих потокових шифросистем з нелінійним випадковим кодуванням для забезпечення безпеки державних інформаційних ресурсів.

Для вирішення поставленої наукової задачі **використано методи** лінійної алгебри, теорії ймовірностей, теорії інформації, теорії складності обчислень, теорії кодування та математичної статистики. Чисельні розрахунки на обчислювальній системі виконувалися з використанням середовища розробки Microsoft Visual Studio 2013 (мова програмування C++) з процесором Intel(R) Core(TM) i3-6100 CPU @ 3.7GHz та обсягом оперативної пам’яті 4 ГБ на базі 64-розрядної Windows 7 Service Pack 1.

Основні наукові та практичні результати, отримані в дисертації.

1. *Вперше* отримано аналітичні оцінки параметрів, що визначають стійкість РПШ Міхалевича-Імаї відносно атак на основі відомих шифрованих повідомлень, а також підібраних векторів ініціалізації. Отримані оцінки *дозволяють* з’ясувати теоретико-кодовий сенс параметрів, які визначають обчислювальну стійкість цих шифросистем, а також встановити, що їх стійкість може бути значно менше, ніж стверджують їх розробники, що досягається *за рахунок* розширення можливостей супротивника при проведенні зазначених атак.

2. *Вперше* доведено, що клас РПШ Міхалевича-Імаї (незалежно від будови їх компонент) володіє суттєвою слабкістю, яка полягає в зменшенні кількості інформації (в порівнянні з довжиною блоку шифрувальної гами), що необхідна

для відновлення за реальний час символів відкритого тексту. Зазначена властивість *дозволяє* зробити практично важливий висновок про те, що для відновлення символів відкритого тексту супротивнику достатньо мати лише часткову інформацію про секретний ключ РПШ, що відбувається *за рахунок* спільного застосування випадкового і завадостійкого кодування повідомлень лінійними кодами.

3. *Вперше* отримано аналітичні межі для швидкості передачі інформації в РПШ Міхалевича-Імаї при заданих обмеженнях щодо ймовірності правильного прийому повідомлень законним користувачем та стійкості шифрування. Зазначені межі, *за рахунок* застосування оцінок Плоткіна та Бассалиго-Елайєса для швидкості передачі лінійних кодів, *дозволяють* зробити науково обґрунтований висновок про обмежені можливості РПШ Міхалевича-Імаї з погляду сучасних вимог щодо стійкості та практичності в реальних умовах.

4. *Отримав подальший розвиток* метод побудови РПШ, який, *на відміну від раніше відомих*, базується на застосуванні для випадкового кодування нелінійних відображень або безключевих геш-функцій та *дозволяє* збільшити стійкість в порівнянні з РПШ Міхалевича-Імаї *за рахунок* розширення класу перетворень, які використовуються в конструкції рандомізатора.

Зокрема, застосування в ролі нелінійного відображення геш-функцій "Купина" або Keccak дозволяє будувати шифросистеми, що є безумовно стійкими відносно будь-яких атак на основі підібраних векторів ініціалізації в моделі випадкового оракулу.

Достовірність результатів дисертаційної роботи забезпечується адекватністю припущень, які лежать в основі проведених наукових досліджень, а також коректним застосуванням відомих математичних методів. Результати проведених чисельних розрахунків узгоджуються з отриманими теоретичними висновками.

Значення наукових результатів дисертації для теорії полягає в тому, що вони утворюють наукову основу для вирішення задач побудови рандомізованих

потокових шифросистем з нелінійним випадковим кодуванням, призначених для забезпечення безпеки державних інформаційних ресурсів.

Практичне значення роботи. Розроблено програмні реалізації рандомізованих потокових шифросистем з нелінійним випадковим кодуванням на основі нелінійних відображень чи безключевих геш-функцій, що є більш стійкими (у 2^{242} і більше разів) і більш швидкісними (у 125 і більше разів) в порівнянні з раніше відомими РПШ при однаковій довжині вихідного повідомлення. Розроблені реалізації РПШ дозволяють здійснювати процедури зашифровування/розшифровування даних в режимі реального часу та можуть бути використані на практиці у спеціальних (військових) додатках, витоки конфіденційної інформації в яких створюють ризики для інформаційної безпеки держави.

Висновки та рекомендації по науковому та практичному використанню наукових результатів. Отримані в дисертаційній роботі нові наукові та практичні результати мають універсальний характер і можуть бути використані як при побудові, так і при дослідженні стійкості рандомізованих потокових шифросистем.

1. Запропонована атака на РПШ Міхалевича-Імаї на основі підібраних векторів ініціалізації має обчислювальну складність, яка залежить лінійно від довжини кодового слова та субквадратично від обсягу матеріалу, що використовується (твердження 2.4). Для шифросистем, побудованих на базі лінійних кодів C_0 з параметрами $n=255$, $m-l=185$, $d_0^{\perp}=62$ складність атаки (з імовірністю успіху $0{,}95$) не перевищує $2^{18{,}86}$ двійкових операцій при $p=0{,}02$ та $2^{53{,}84}$ двійкових операцій при $p=0{,}10$.

2. Вплив генератора гами на стійкість РПШ Міхалевича-Імаї проявляється в тому, що системи булевих лінійних рівнянь зі спотвореними правими частинами, до розв’язання яких зводиться відновлення ключів, можуть мати дуже спеціальний вигляд та розв’язуватися суттєво швидше, ніж системи

загального вигляду з тими самими числами невідомих та ймовірністю спотворень. Останній параметр залежить від дуальної відстані коду C_0, належний вибір якого (для заданого коду C_1), з урахуванням обмеження (2.6), є нетривіальною задачею.

3. Обчислювальна стійкість РПШ з нелінійним випадковим кодуванням відносно атаки на основі відомого шифрованого повідомлення визначається дуальною відстанню d' нелінійного коду, який будується за нелінійним відображенням в конструкції рандомізотора шифросистеми. Зі збільшенням дуальної відстані коду складність атаки на шифросистему збільшується та наближається до складності повного перебору усіх вхідних повідомлень довжини l (наслідок 3.1). Зокрема, для шифросистем, побудованих на базі кодів Препарати, при $l = 20$, $d' = 496$ обчислювальна стійкість змінюється від $2^{7,19}$ (при $p = 0,005$) до 2^{20} (при $p \geq 0,05$). Якщо $l = 24$, $d' = 2016$, то стійкість відповідної шифросистеми дорівнює $2^{23,26}$ при $p = 0,005$ та 2^{24} при $p \geq 0,008$.

4. Обчислювальна стійкість запропонованих РПШ відносно найбільш потужних (з відомих сьогодні) атак на основі підібраних векторів ініціалізації визначається такими властивостями відображення $\phi : V_{m-l} \to V_l$:

– велике значення параметра $m - l$ для протидії перебірній атаці (алгоритм 3.1);

– мале значення параметра (3.15) для протидії атакам лінійного типу (алгоритм 3.2);

– велика часова складність розв‘язання системи рівнянь (3.19).

На відміну від РПШ Міхалевича-Імаї, що є вразливими до кореляційних атак (або малопрактичними; підрозділи 2.2, 2.3), запропоновані шифросистеми є обчислювально стійкими відносно атаки на основі підібраних векторів ініціалізації, якщо відображення ϕ характеризується малим значенням максимального елемента таблиці лінійних апроксимацій

(твердження 3.4). При цьому для підвищення практичності РПШ величина l повинна бути також достатньо великою (наприклад, умова $m-l=l=128$ забезпечує швидкість передачі інформації $l/m=1/2$ незалежно від вибору відображення ϕ).

5. Для шифросистем, побудованих на базі відображень $\phi: V_{m-l} \to V_l$ з параметрами $l=128$, $m=256$, обчислювальна стійкість відносно перебірної атаки (алгоритм 3.1) складає 2^{134} операцій, якщо параметр (3.14) дорівнює 2^{-2} та 2^{130} операцій, якщо цей параметр дорівнює 2^{-120}. Якщо при цьому $L_{\phi}=2^{-61}$, то обчислювальна стійкість зазначених шифросистем відносно атаки лінійного типу (алгоритм 3.2) складає 2^{127}.

6. При фіксованій стійкості та однакій довжині n шифрованого повідомлення побудовані рандомізовані потокові шифросистеми з нелінійним випадковим кодуванням мають значно більшу швидкість передачі в порівнянні з РПШ Міхалевича-Імаї. Так, при $n=512$ максимальна швидкість передачі РПШ Міхалевича-Імаї зі стійкістю 2^{79} дорівнює $0{,}004$, що у 125 разів менше швидкості передачі нелінійних РПШ (зі стійкістю 2^{249}). У випадку $n=1024$ максимальна швидкість передачі РПШ Міхалевича-Імаї зі стійкістю 2^{154} дорівнює $0{,}001$, що у 500 разів менше швидкості передачі РПШ з нелінійним випадковим кодуванням (зі стійкістю 2^{505}) (табл. 4.1).

7. Результати порівняння стійкості двох зазначених видів шифросистем при фіксованій швидкості передачі (що дорівнює $1/2$) також свідчать про суттєву перевагу РПШ з нелінійним випадковим кодуванням над РПШ Міхалевича-Імаї, які виявляються практично нестійкими при зазначеній швидкості. Зокрема, при $n=512$ і $n=1024$ стійкість РПШ з нелінійним випадковим кодуванням вища за стійкість РПШ Міхалевича-Імаї у 2^{242} і 2^{487} разів відповідно (табл. 4.2).

8. Аналіз вибору компонент для практичної побудови конкретних варіантів РПШ з нелінійним випадковим кодуванням показує, що в ролі нелінійного відображення в конструкції рандомізотора доцільно використовувати одну з відомих обчислювально стійких геш-функцій, наприклад, "Купину" або Keccak. Побудовані таким чином РПШ є безумовно стійкими відносно будь-яких атак на основі підібраних векторів ініціалізації в моделі випадкового оракула та забезпечують (на сьогодні) практичну стійкість на рівні 2^l операцій за умови практичної стійкості геш-функцій Keccak-l та "Купина"-l.

9. Розроблені програмні реалізації трьох варіантів РПШ з нелінійним випадковим кодуванням дозволяють здійснювати на практиці процедури зашифровування/розшифровування даних в режимі реального часу. При цьому найбільш ефективною за часом виконання є шифросистема IKCS (ISAAC – Keccak – Snow 2.0). Зокрема, час зашифровування/розшифровування файлів розміром 180 Кб за допомогою цієї шифросистеми є в 65 та 303 рази менше часу зашифровування/розшифровування файлів такого ж розміру за допомогою шифросистем IKPS та INLS відповідно.

10. Основні наукові та практичні результати реалізовані в НДР "Кета" та в науково-технічних розробках ЗАО "Інститут інформаційних технологій". Вони також можуть бути використані у спеціальних (військових) додатках, витоки конфіденційної інформації в яких створюють ризики для інформаційної безпеки держави. Такими додатками є ті, в яких: часто трапляються випадки компрометації шифрувальної апаратури або алгоритму шифрування, відмови системи блокування шифратора, внаслідок чого в канал передачі може потрапити «слабка» гама шифрування; передаються короткі повідомлення (спеціальні команди чи військові накази); є невеликим навантаження на інформаційний трафік; криптографічна стійкість є важливішою за швидкість передачі інформації; є невеликою кількість абонентів; алгоритм шифрування може бути невідомим. Подальший

розвиток наукових ідей та методів, які лежать в основі дисертаційного дослідження, є актуальним напрямом в галузі забезпечення інформаційної безпеки держави.

ДОДАТКИ

Додаток А

Програмний код реалізації РПШ з нелінійним випадковим кодуванням (варіант IKCS: “Isaac – Keccak – Snow 2.0”)

Програмна реалізація РПШ IKCS виконана на ПК з процесором Intel(R) Core(TM) i3-6100, 3.7GHz та обсягом оперативної пам’яті 4 ГБ на базі 64-розрядної ОС Windows 7 Service Pack 1. Мова програмування – C++. Середовище розробки – Microsoft Visual Studio 2013.

```
// Файл Realization_1.cpp

#include "stdafx.h"
#include "General.h"
#include "IKS2.h"
#include <time.h>


int main(int argc, char* argv[])
{
	General run;

	IKS2 iks2;

	int flen;

	clock_t start, finish;

	double duration;

	BYTE *in_file_buf;

	FILE *fp = fopen(argv[1], "rb");

	if(fp == NULL)
	{
		printf("ERROR!!! The test.to file was not opened!!! \n");

		return 0;
	}

	flen = run.get_file_len(fp);

	in_file_buf = new BYTE [flen];    //buffer for the input file

	memset(in_file_buf, 0, flen);
```

```
	int read_bytes = fread(in_file_buf, 1, flen, fp);

	if(flen != read_bytes)
	{
		printf("\n ERROR!!! It wasn't read all bytes\n ");

		return 0;
	}

	start = clock();

	iks2.EncrDecr(flen, in_file_buf, argv[1]); // read by 32 bytes

	finish = clock();

	duration = (double)(finish - start) / CLOCKS_PER_SEC;

	printf("\n elapsed time = %f seconds", duration);

	fclose(fp);

	return 0;
}


// Файл IKS2.cpp

#include "StdAfx.h"
#include "IKS2.h"
#include "Isaac64.h"
#include "Keccak.h"
#include "General.h"
#include "Snow.h"

IKS2::IKS2(void)
{
}

IKS2::~IKS2(void)
{
}

void IKS2::EncrDecr(int len, BYTE* plain_txt, char *fto)
{
	char fts[25] = { 0 };
	char fdec[25] = { 0 };

	Snow snow_encr, snow_decr;

	Isaac64 isaac;

	Keccak keccak;

	General run;

	int remLen = len % 32;

	// make new name for encrypted and decrypted files
	run.NewName(fto, fts, ".iks");

	run.NewName(fto, fdec, ".iks_dec");

	BYTE vec[32], tmp, plain_byte[32];
```

```
	BYTE *hash;

	Init(len); //allocate memory for encrypted and decrypted text

	__int64 *u64;

	// 1) ISAAC: output = 256 8-bytes numbers (__int64), total = 2048 bytes
	int cycles = (len / 2048) + 1;

	BYTE *u = new BYTE[256 * 8 * cycles];

	for (int cycl = 0; cycl < cycles; cycl++)
	{
		u64 = isaac.DoIsaac64();

		for (int i = 0; i < 256; i++)  // 256 * 8 bytes = 2048 bytes
		{
			for (int j = 0; j < 8; j++)
			{
				u[cycl + i * 8 + j] = ((u64[i] >> (j * 8)) & 0xff);
			}
		}
	}

	// 2)
	Snow::Ecrypt_ctx *ctx = new Snow::Ecrypt_ctx;

	BYTE snow_key[32] = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef, 0x12, 0x34,
0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0,
                                                0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff, 0x33 },
	      snow_IV[16]  = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef, 0x12, 0x34,
0x56, 0x78, 0x9a, 0xbc, 0xcd, 0xef };

	snow_encr.Ecrypt_init(ctx, snow_key, 256, 128, snow_IV);

	// do enciphering (m - l = l = 256 bits)
	// read the plain text in 32 bytes' blocks
	for (int i = 0; i < len / 32; i++)
	{
		hash = keccak.DoKeccak(32, u + i * 32); // compute 32 bytes hash code from
random vector u

		// fill the first 32 bytes of the output vector by pseudorandom bytes u (m =
512 bits: (m - l, l))
		for (int y = 0; y < 32; y++)
			vec_m[y] = u[i * 32 + y];

		// the second part of the output is s_i ^ fi(u_i)
		for (int j = 0; j < 32; j++)
			vec[j] = plain_txt[i * 32 + j] ^ hash[j];

		memcpy(vec_m + 32, vec, 32);

		// the result vector goes for enciphering
		snow_encr.Ecrypt_process_bytes(0, ctx, vec_m, enc_out + i * 64, 64);

	}

	// process the last block which is smaller than 32 bytes
	if (remLen)
	{
```

```
		hash = keccak.DoKeccak(32, u + (len / 32) * 32);

		for (int y = 0; y < 32; y++)
			vec_m[y] = u[(len / 32) * 32 + y];

		for (int j = 0; j < remLen; j++)
			vec[j] = plain_txt[(len / 32) * 32 + j] ^ hash[j];

		memcpy(vec_m + 32, vec, remLen);

		snow_encr.Ecrypt_process_bytes(0, ctx, vec_m, enc_out + (len / 32) * 64, 32 + 
remLen);
	}

	run.WritetoFile(fts, enc_out, len * 2 - (32 - (len % 32)));  // write cipher text in 
file

	// D E C R Y P T I O N
	FILE *fout = fopen(fdec, "wb");

	// do INITIALIZATION
	snow_decr.Ecrypt_init(ctx, snow_key, 256, 128, snow_IV);

	for (int i = 0; i < len / 32; i++)
	{
		//do deciphering
		snow_decr.Ecrypt_process_bytes(1, ctx, enc_out + i * 64, dec_out + i * 64, 
64);

		hash = keccak.DoKeccak(32, dec_out + i * 64);

		for (int j = 0; j < 32; j++)
		{
			plain_byte[j] = hash[j] ^ dec_out[(i * 64 + 32) + j];

			fprintf(fout, "%c", plain_byte[j]);
		}
	}

	// decipher the last block
	if (remLen)
	{
		int bound = len / 32;

		snow_decr.Ecrypt_process_bytes(1, ctx, enc_out + bound * 64, dec_out + bound * 
64, 32 + remLen);

		hash = keccak.DoKeccak(32, dec_out + bound * 64);

		for (int j = 0; j < remLen; j++)
		{
			plain_byte[j] = hash[j] ^ dec_out[(bound * 64 + 32) + j];

			fprintf(fout, "%c", plain_byte[j]);
		}
	}

	fcloseall();
	delete[] u;
}

void IKS2::Init(int len)
{
```

```
        enc_out = new BYTE [len * 2];

        dec_out = new BYTE [len * 2];

        memset(enc_out, 0, sizeof(enc_out));

        memset(dec_out, 0, sizeof(dec_out));

}


// Файл IKS2.h

#pragma once
class IKS2
{
public:
        IKS2(void);
        ~IKS2(void);

        BYTE *enc_out; // encrypted text

        BYTE *dec_out; // decrypted text

        BYTE vec_m[64];

        void EncrDecr(int len, BYTE* plain_txt, char *fto);

        void Init(int len);
};


// Файл General.cpp

#include "StdAfx.h"
#include "General.h"

General::General(void)
{
}

General::~General(void)
{
}

void General::hex_print(const void* pv, int len)
{
        const BYTE * p = (const BYTE*)pv;

        if (NULL == pv)
                printf("NULL");
        else
        {
                for (int i = 0; i < len; ++i)
                {
                        if (i % 16 == 0)
                                printf("\n");
                        printf("%02X ", *p++);
                }
        }
        printf("\n");
}
```

```
int General::get_file_len(FILE* fp)
{
	int len;

	fseek(fp, 0, SEEK_END);

	len = ftell(fp);

	fseek(fp, 0, SEEK_SET);

	return len;
}

void General::NewName(char* fn1, char* fn2, const char* _ext)
{
	memset(fn2, 0, sizeof(fn2));

	strcpy(fn2, fn1);

	char *p = strchr(fn2, '.');

	if(p)
		strcpy(p, _ext);
	else
		strcat(fn2, _ext);
}

void General::WritetoFile(char* fname, BYTE *buf, int len)
{
	FILE *fout;

	if((fout = fopen(fname, "wb")) == NULL)
	{
		perror("open failed");
		printf("\n ERROR open file %s to write!!!", fname);
	}

	fwrite(buf, 1, len, fout);
	fclose(fout);
}


// Файл General.h

#pragma once

class General
{
public:
	General(void);
	~General(void);

	static void hex_print(const void* pv, int len);
	int get_file_len(FILE* fp);
	void NewName(char* fn1, char* fn2, const char* _ext);
	void WritetoFile(char* fname, BYTE *buf, int len);
};


// Файл Isaac64.cpp
#include "StdAfx.h"
#include "Isaac64.h"
```

```
Isaac64::Isaac64(void)
{
}

Isaac64::~Isaac64(void)
{
}

void Isaac64::RandInit(int flag)
{
	__int64 a, b, c, d, e, f, g, h;

	aa = bb = cc = (__int64)0;
	a = b = c = d = e = f = g = h = 0x9e3779b97f4a7c13LL;
	for (int i = 0; i < 4; ++i)
	{
		mix(a, b, c, d, e, f, g, h);
	}

	for (int i = 0; i < RANDSIZ; i += 8)
	{
		if (flag)
		{
			a += randrsl[i];
			b += randrsl[i + 1];
			c += randrsl[i + 2];
			d += randrsl[i + 3];
			e += randrsl[i + 4];
			f += randrsl[i + 5];
			g += randrsl[i + 6];
			h += randrsl[i + 7];
		}

		mix(a, b, c, d, e, f, g, h);

		mm[i] = a;
		mm[i + 1] = b;
		mm[i + 2] = c;
		mm[i + 3] = d;
		mm[i + 4] = e;
		mm[i + 5] = f;
		mm[i + 6] = g;
		mm[i + 7] = h;
	}

	if (flag)
	{
		for (int i = 0; i < RANDSIZ; i += 8)
		{
			a += mm[i];
			b += mm[i + 1];
			c += mm[i + 2];
			d += mm[i + 3];
			e += mm[i + 4];
			f += mm[i + 5];
			g += mm[i + 6];
			h += mm[i + 7];

			mix(a, b, c, d, e, f, g, h);

			mm[i] = a;
			mm[i + 1] = b;
			mm[i + 2] = c;
```

```
				mm[i + 3] = d;
				mm[i + 4] = e;
				mm[i + 5] = f;
				mm[i + 6] = g;
				mm[i + 7] = h;
			}
		}

		RunIsaac64();
		randcnt = RANDSIZ;
}

void Isaac64::RunIsaac64()
{
		register __int64 a, b, x, y, *m, *m2, *r, *mend;
		m = mm;
		r = randrsl;
		a = aa;
		b = bb + (++cc);

		for (m = mm, mend = m2 = m + (RANDSIZ / 2); m < mend;)
		{
			rngstep(~(a ^ (a << 21)), a, b, mm, m, m2, r, x);
			rngstep(a ^ (a >> 5), a, b, mm, m, m2, r, x);
			rngstep(a ^ (a << 12), a, b, mm, m, m2, r, x);
			rngstep(a ^ (a >> 33), a, b, mm, m, m2, r, x);
		}

		for (m2 = mm; m2<mend;)
		{
			rngstep(~(a ^ (a << 21)), a, b, mm, m, m2, r, x);
			rngstep(a ^ (a >> 5), a, b, mm, m, m2, r, x);
			rngstep(a ^ (a << 12), a, b, mm, m, m2, r, x);
			rngstep(a ^ (a >> 33), a, b, mm, m, m2, r, x);
		}

		bb = b; aa = a;
}

__int64* Isaac64::DoIsaac64(void)
{
		aa = bb = cc = 0;
		for (int i = 0; i < RANDSIZ; ++i)
			mm[i] = 0;

		RandInit(TRUE);

		RunIsaac64();

		return randrsl;
}


// Файл Isaac64.h

#pragma once

#define mix(a, b, c, d, e, f, g, h) \
{ \
		a -= e;  f ^= h >> 9;   h += a; \
		b -= f;  g ^= a << 9;   a += b; \
		c -= g;  h ^= b >> 23;  b += c; \
		d -= h;  a ^= c << 15;  c += d; \
```

```
	e -= a;  b ^= d >> 14;  d += e; \
	f -= b;  c ^= e << 20;  e += f; \
	g -= c;  d ^= f >> 17;  f += g; \
	h -= d;  e ^= g << 14;  g += h; \
}

#define ind(mm, x)  (*(__int64 *)((ub1 *)(mm) + ((x) & ((RANDSIZ - 1) << 3))))

#define rngstep(mix, a, b, mm, m, m2, r, x) \
{ \
	x = *m;  \
	a = (mix)+*(m2++); \
	*(m++) = y = ind(mm, x) + a + b; \
	*(r++) = b = ind(mm, y >> RANDSIZL) + x; \
}

static __int64 mm[RANDSIZ];
static __int64 aa, bb, cc;

class Isaac64
{
public:
	Isaac64(void);
	~Isaac64(void);

	void RandInit(int flag);
	__int64 randrsl[RANDSIZ];
	__int64 randcnt;
	void RunIsaac64();
	__int64* DoIsaac64(void);
};


// Файл Keccak.cpp

#include "stdafx.h"
#include "Keccak.h"
#include "General.h"

Keccak::Keccak(void)
{
}

Keccak::~Keccak(void)
{
}

const uint64 Keccak::keccakf_rndc[24] =
{
	0x0000000000000001, 0x0000000000008082, 0x800000000000808a,
	0x8000000080008000, 0x000000000000808b, 0x0000000080000001,
	0x8000000080008081, 0x8000000000008009, 0x000000000000008a,
	0x0000000000000088, 0x0000000080008009, 0x000000008000000a,
	0x000000008000808b, 0x800000000000008b, 0x8000000000008089,
	0x8000000000008003, 0x8000000000008002, 0x8000000000000080,
	0x000000000000800a, 0x800000008000000a, 0x8000000080008081,
	0x8000000000008080, 0x0000000080000001, 0x8000000080008008
};
const int Keccak::keccakf_rotc[24] =
{
	1,  3,   6, 10, 15, 21, 28, 36, 45, 55,  2, 14,
	27, 41, 56,  8, 25, 43, 62, 18, 39, 61, 20, 44
};
```

```
const int Keccak::keccakf_piln[24] =
{
	10,  7, 11, 17, 18, 3,  5, 16,  8, 21, 24, 4,
	15, 23, 19, 13, 12, 2, 20, 14, 22,  9, 6,  1
};

const unsigned int Keccak::block_size_bytes = 136;

void Keccak::keccakf(uint64 st[25], int rounds)
{
	int i, j, round;
	uint64 t, bc[5];

	for (round = 0; round < rounds; round++)
	{
		// Theta
		for (i = 0; i < 5; i++)
			bc[i] = st[i] ^ st[i + 5] ^ st[i + 10] ^ st[i + 15] ^ st[i + 20];

		for (i = 0; i < 5; i++)
		{
			t = bc[(i + 4) % 5] ^ ROTL64(bc[(i + 1) % 5], 1);
			for (j = 0; j < 25; j += 5)
				st[j + i] ^= t;
		}

		// Rho Pi
		t = st[1];

		for (i = 0; i < 24; i++)
		{
			j = keccakf_piln[i];
			bc[0] = st[j];
			st[j] = ROTL64(t, keccakf_rotc[i]);
			t = bc[0];
		}

		//  Chi
		for (j = 0; j < 25; j += 5)
		{
			for (i = 0; i < 5; i++)
				bc[i] = st[j + i];

			for (i = 0; i < 5; i++)
				st[j + i] ^= (~bc[(i + 1) % 5]) & bc[(i + 2) % 5];
		}

		//  Iota
		st[0] ^= keccakf_rndc[round];
	}
}

// compute a keccak hash (md) of given byte length from "in"
// :136: bytes input length for 32 bytes (256 bits) hash

int Keccak::keccak_hash_block(const BYTE *in, int inlen, int mdlen)
{
	int i, rsiz, rsizw;

	rsiz = 200 - 2 * mdlen;

	rsizw = rsiz / 8;
```

```
	for (i = 0; i < rsizw; i++)
		state[i] ^= ((uint64 *)in)[i];

	keccakf(state, KECCAK_ROUNDS);

	return 0;
}

// compute final hash
int Keccak::keccak_final(const BYTE *in, int inlen, int mdlen)
{
	BYTE temp[144];

	int i, rsiz, rsizw;

	rsiz = 200 - (mdlen << 1);

	rsizw = rsiz / 8;

	memcpy(temp, in, inlen);

	temp[inlen++] = 1;

	memset(temp + inlen, 0, rsiz - inlen);

	temp[rsiz - 1] |= 0x80;

	for (i = 0; i < rsizw; i++)
		state[i] ^= ((uint64 *)temp)[i];

	keccakf(state, KECCAK_ROUNDS);

	return 0;
}

void Keccak::get_hash_bytes(BYTE *hash, int hashlen)
{
	memcpy(hash, state, hashlen);
}

unsigned char* Keccak::DoKeccak(int len, BYTE* buf)
{
	General run;

	int i;

	unsigned int block_size_bytes = get_block_size_bytes();

	unsigned int bytes_read;

	InitState();

	for(i = 0; (i + block_size_bytes) < len; i += block_size_bytes)
		keccak_hash_block(buf + i, block_size_bytes, 32);

	keccak_final(buf + i, len - i, 32);

	get_hash_bytes(hash, 32);

	return hash;
}
void Keccak::InitState(void)
{
```

```
	memset(state, 0, sizeof(state));

}


// Файл Keccak.h

#pragma once

#define KECCAK_ROUNDS 24
#define ROTL64(x, y) (((x) << (y)) | ((x) >> (64 - (y))))

class Keccak
{
private:
	static const uint64 keccakf_rndc[24];

	static const int keccakf_rotc[24];

	static const int keccakf_piln[24];

	uint64 state[25];

	static const unsigned int block_size_bytes;

	unsigned char buffer[64];
	unsigned char hash[32];

public:
	Keccak(void);
	~Keccak(void);

	// update the state
	void keccakf(uint64 st[25], int norounds);

	// compute a keccak hash (md) of given byte length from "in"
	int keccak_hash_block(const BYTE *in, int inlen, int mdlen);

	// compute final hash
	int keccak_final(const BYTE *in, int inlen, int mdlen);

	void get_hash_bytes(BYTE *hash, int hashlen);

	unsigned int get_block_size_bytes()
	{
		return block_size_bytes;
	}
	unsigned char* DoKeccak(int len, BYTE* buf);

	void InitState(void);
};

// Файл Snow.cpp

#include "stdafx.h"
#include "Snow.h"
#include "snowtab.h"
#include "General.h"

Snow::Snow()
{
}
```

```
Snow::~Snow()
{
}

/* Key and message independent initialization. This function will be
*  called once when the program starts (e.g., to build expanded S-box
*  tables).
*/
void Snow::Ecrypt_init(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize, const
BYTE* iv)
{
	Ecrypt_keysetup(ctx, key, keysize, ivsize);

	Ecrypt_ivsetup(ctx, iv);
}

/* Key setup. It is the user's responsibility to select the values of
*  keysize and ivsize from the set of supported values specified
*  above.
*/
void Snow::Ecrypt_keysetup(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize)
{
	for (int i = 0; i < keysize / 8; ++i)
		ctx->key[i] = key[i];

	ctx->keysize = keysize;
}

/* IV setup. After having called ECRYPT_keysetup(), the user is
*  allowed to call ECRYPT_ivsetup() different times in order to
*  encrypt/decrypt different messages with the same key but different
*  IV's.
*/
void Snow::Ecrypt_ivsetup(Ecrypt_ctx* ctx, const BYTE* iv)
{
	snow_loadkey_fast(ctx, U8TO32_LITTLE(iv), U8TO32_LITTLE(iv + 4), U8TO32_LITTLE(iv +
8), U8TO32_LITTLE(iv + 12));
}


void Snow::Ecrypt_process_bytes(int action, Ecrypt_ctx* ctx, const BYTE* input, BYTE*
output, u32 msglen)
{
	u32 i;
	u32 keystream[16];
	u32 tmp;

	for (; msglen >= 64; msglen -= 64, input += 64, output += 64)
	{
		snow_keystream_fast(ctx, keystream); // keystream = 64 bytes

		for (i = 0; i < 16; ++i)
		{
			tmp = ((u32*)input)[i];

			((u32*)output)[i] = tmp ^ U32TO32_LITTLE(keystream[i]);
		}
	}

	if (msglen > 0)
	{
		snow_keystream_fast(ctx, keystream);
```

```
		for (i = 0; i < msglen; i++)
			output[i] = input[i] ^ ((BYTE*)keystream)[i];
	}
}


/* Function:  snow_loadkey_fast
*
* Synopsis:
*   Loads the key material and performs the initial mixing.
*
* Returns: void
*
* Assumptions:
*   keysize is either 128 or 256.
*   key is of proper length, for keysize=128, key is of lenght 16 bytes
*      and for keysize=256, key is of length 32 bytes.
*   key is given in big endian format,
*   For 128 bit key:
*        key[0]-> msb of k_3
*         ...
*        key[3]-> lsb of k_3
*         ...
*        key[12]-> msb of k_0
*         ...
*        key[15]-> lsb of k_0
*
*   For 256 bit key:
*        key[0]-> msb of k_7
*          ...
*        key[3]-> lsb of k_7
*          ...
*        key[28]-> msb of k_0
*          ...
*        key[31]-> lsb of k_0
*/

void Snow::snow_loadkey_fast(Ecrypt_ctx* ctx, u32 IV3, u32 IV2, u32 IV1, u32 IV0)
{
	int i;

	if (ctx->keysize == 128)
	{
		ctx->s15 = (((u32)*(ctx->key + 0)) << 24) | (((u32)*(ctx->key + 1)) << 16) |
			(((u32)*(ctx->key + 2)) << 8) | (((u32)*(ctx->key + 3)));
		ctx->s14 = (((u32)*(ctx->key + 4)) << 24) | (((u32)*(ctx->key + 5)) << 16) |
			(((u32)*(ctx->key + 6)) << 8) | (((u32)*(ctx->key + 7)));
		ctx->s13 = (((u32)*(ctx->key + 8)) << 24) | (((u32)*(ctx->key + 9)) << 16) |
			(((u32)*(ctx->key + 10)) << 8) | (((u32)*(ctx->key + 11)));
		ctx->s12 = (((u32)*(ctx->key + 12)) << 24) | (((u32)*(ctx->key + 13)) << 16) |
			(((u32)*(ctx->key + 14)) << 8) | (((u32)*(ctx->key + 15)));
		ctx->s11 = ~ctx->s15;  /* bitwise inverse */
		ctx->s10 = ~ctx->s14;
		ctx->s9 = ~ctx->s13;
		ctx->s8 = ~ctx->s12;
		ctx->s7 = ctx->s15;    /* just copy */
		ctx->s6 = ctx->s14;
		ctx->s5 = ctx->s13;
		ctx->s4 = ctx->s12;
		ctx->s3 = ~ctx->s15;   /* bitwise inverse */
		ctx->s2 = ~ctx->s14;
		ctx->s1 = ~ctx->s13;
		ctx->s0 = ~ctx->s12;
```

```
	}
	else
	{
		// assume keysize=256
		ctx->s15 = (((u32)*(ctx->key + 0)) << 24) | (((u32)*(ctx->key + 1)) << 16) |
			(((u32)*(ctx->key + 2)) << 8) | (((u32)*(ctx->key + 3)));
		ctx->s14 = (((u32)*(ctx->key + 4)) << 24) | (((u32)*(ctx->key + 5)) << 16) |
			(((u32)*(ctx->key + 6)) << 8) | (((u32)*(ctx->key + 7)));
		ctx->s13 = (((u32)*(ctx->key + 8)) << 24) | (((u32)*(ctx->key + 9)) << 16) |
			(((u32)*(ctx->key + 10)) << 8) | (((u32)*(ctx->key + 11)));
		ctx->s12 = (((u32)*(ctx->key + 12)) << 24) | (((u32)*(ctx->key + 13)) << 16) |
			(((u32)*(ctx->key + 14)) << 8) | (((u32)*(ctx->key + 15)));
		ctx->s11 = (((u32)*(ctx->key + 16)) << 24) | (((u32)*(ctx->key + 17)) << 16) |
			(((u32)*(ctx->key + 18)) << 8) | (((u32)*(ctx->key + 19)));
		ctx->s10 = (((u32)*(ctx->key + 20)) << 24) | (((u32)*(ctx->key + 21)) << 16) |
			(((u32)*(ctx->key + 22)) << 8) | (((u32)*(ctx->key + 23)));
		ctx->s9 = (((u32)*(ctx->key + 24)) << 24) | (((u32)*(ctx->key + 25)) << 16) |
			(((u32)*(ctx->key + 26)) << 8) | (((u32)*(ctx->key + 27)));
		ctx->s8 = (((u32)*(ctx->key + 28)) << 24) | (((u32)*(ctx->key + 29)) << 16) |
			(((u32)*(ctx->key + 30)) << 8) | (((u32)*(ctx->key + 31)));
		ctx->s7 = ~ctx->s15; /* bitwise inverse */
		ctx->s6 = ~ctx->s14;
		ctx->s5 = ~ctx->s13;
		ctx->s4 = ~ctx->s12;
		ctx->s3 = ~ctx->s11;
		ctx->s2 = ~ctx->s10;
		ctx->s1 = ~ctx->s9;
		ctx->s0 = ~ctx->s8;
	}

	// XOR IV values
	ctx->s15 ^= IV0;
	ctx->s12 ^= IV1;
	ctx->s10 ^= IV2;
	ctx->s9 ^= IV3;


	ctx->r1 = 0;

	ctx->r2 = 0;

	/* Do 32 initial clockings */
	for (i = 0; i < 2; i++)
	{
		u32 outfrom_fsm, fsmtmp;

		outfrom_fsm = (ctx->r1 + ctx->s15) ^ ctx->r2;
		ctx->s0 = a_mul(ctx->s0) ^ ctx->s2 ^ ainv_mul(ctx->s11) ^ outfrom_fsm;
		fsmtmp = ctx->r2 + ctx->s5;
		ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
		ctx->r1 = fsmtmp;

		outfrom_fsm = (ctx->r1 + ctx->s0) ^ ctx->r2;
		ctx->s1 = a_mul(ctx->s1) ^ ctx->s3 ^ainv_mul(ctx->s12) ^ outfrom_fsm;
		fsmtmp = ctx->r2 + ctx->s6;
		ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
		ctx->r1 = fsmtmp;

		outfrom_fsm = (ctx->r1 + ctx->s1) ^ ctx->r2;
		ctx->s2 = a_mul(ctx->s2) ^ ctx->s4 ^ainv_mul(ctx->s13) ^ outfrom_fsm;
		fsmtmp = ctx->r2 + ctx->s7;
```

```
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s2) ^ ctx->r2;
            ctx->s3 = a_mul(ctx->s3) ^ ctx->s5 ^ainv_mul(ctx->s14) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s8;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s3) ^ ctx->r2;
            ctx->s4 = a_mul(ctx->s4) ^ ctx->s6 ^ainv_mul(ctx->s15) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s9;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s4) ^ ctx->r2;
            ctx->s5 = a_mul(ctx->s5) ^ ctx->s7 ^ainv_mul(ctx->s0) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s10;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s5) ^ ctx->r2;
            ctx->s6 = a_mul(ctx->s6) ^ ctx->s8 ^ainv_mul(ctx->s1) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s11;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s6) ^ ctx->r2;
            ctx->s7 = a_mul(ctx->s7) ^ ctx->s9 ^ainv_mul(ctx->s2) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s12;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s7) ^ ctx->r2;
            ctx->s8 = a_mul(ctx->s8) ^ ctx->s10 ^ainv_mul(ctx->s3) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s13;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s8) ^ ctx->r2;
            ctx->s9 = a_mul(ctx->s9) ^ ctx->s11 ^ainv_mul(ctx->s4) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s14;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s9) ^ ctx->r2;
            ctx->s10 = a_mul(ctx->s10) ^ ctx->s12 ^ainv_mul(ctx->s5) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s15;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s10) ^ ctx->r2;
            ctx->s11 = a_mul(ctx->s11) ^ ctx->s13 ^ainv_mul(ctx->s6) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s0;
```

```
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s11) ^ ctx->r2;
            ctx->s12 = a_mul(ctx->s12) ^ ctx->s14 ^ainv_mul(ctx->s7) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s1;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s12) ^ ctx->r2;
            ctx->s13 = a_mul(ctx->s13) ^ ctx->s15 ^ainv_mul(ctx->s8) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s2;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s13) ^ ctx->r2;
            ctx->s14 = a_mul(ctx->s14) ^ ctx->s0 ^ainv_mul(ctx->s9) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s3;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s14) ^ ctx->r2;
            ctx->s15 = a_mul(ctx->s15) ^ ctx->s1 ^ainv_mul(ctx->s10) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s4;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

      }
}

/* Function: snow_keystream_fast
*
*  Synopsis:
*  Clocks the cipher 16 times and returns 16 words of keystream symbols
*  in keystream_block.
*
*/

void Snow::snow_keystream_fast(Ecrypt_ctx* ctx, u32* keystream_block)
{
      u32 fsmtmp;

      ctx->s0 = a_mul(ctx->s0) ^ ctx->s2 ^ainv_mul(ctx->s11);
      fsmtmp = ctx->r2 + ctx->s5;
      ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
      ctx->r1 = fsmtmp;
      keystream_block[0] = (ctx->r1 + ctx->s0) ^ ctx->r2^ ctx->s1;

      ctx->s1 = a_mul(ctx->s1) ^ ctx->s3 ^ainv_mul(ctx->s12);
      fsmtmp = ctx->r2 + ctx->s6;
      ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
      ctx->r1 = fsmtmp;
      keystream_block[1] = (ctx->r1 + ctx->s1) ^ ctx->r2^ ctx->s2;

      ctx->s2 = a_mul(ctx->s2) ^ ctx->s4 ^ainv_mul(ctx->s13);
      fsmtmp = ctx->r2 + ctx->s7;
```

```
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[2] = (ctx->r1 + ctx->s2) ^ ctx->r2^ ctx->s3;

        ctx->s3 = a_mul(ctx->s3) ^ ctx->s5 ^ainv_mul(ctx->s14);
        fsmtmp = ctx->r2 + ctx->s8;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[3] = (ctx->r1 + ctx->s3) ^ ctx->r2^ ctx->s4;

        ctx->s4 = a_mul(ctx->s4) ^ ctx->s6 ^ainv_mul(ctx->s15);
        fsmtmp = ctx->r2 + ctx->s9;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[4] = (ctx->r1 + ctx->s4) ^ ctx->r2^ ctx->s5;

        ctx->s5 = a_mul(ctx->s5) ^ ctx->s7 ^ainv_mul(ctx->s0);
        fsmtmp = ctx->r2 + ctx->s10;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[5] = (ctx->r1 + ctx->s5) ^ ctx->r2^ ctx->s6;

        ctx->s6 = a_mul(ctx->s6) ^ ctx->s8 ^ainv_mul(ctx->s1);
        fsmtmp = ctx->r2 + ctx->s11;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[6] = (ctx->r1 + ctx->s6) ^ ctx->r2^ ctx->s7;

        ctx->s7 = a_mul(ctx->s7) ^ ctx->s9 ^ainv_mul(ctx->s2);
        fsmtmp = ctx->r2 + ctx->s12;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[7] = (ctx->r1 + ctx->s7) ^ ctx->r2^ ctx->s8;

        ctx->s8 = a_mul(ctx->s8) ^ ctx->s10 ^ainv_mul(ctx->s3);
        fsmtmp = ctx->r2 + ctx->s13;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[8] = (ctx->r1 + ctx->s8) ^ ctx->r2^ ctx->s9;

        ctx->s9 = a_mul(ctx->s9) ^ ctx->s11 ^ainv_mul(ctx->s4);
        fsmtmp = ctx->r2 + ctx->s14;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[9] = (ctx->r1 + ctx->s9) ^ ctx->r2^ ctx->s10;

        ctx->s10 = a_mul(ctx->s10) ^ ctx->s12 ^ainv_mul(ctx->s5);
        fsmtmp = ctx->r2 + ctx->s15;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[10] = (ctx->r1 + ctx->s10) ^ ctx->r2^ ctx->s11;

        ctx->s11 = a_mul(ctx->s11) ^ ctx->s13 ^ainv_mul(ctx->s6);
        fsmtmp = ctx->r2 + ctx->s0;
```

```
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[11] = (ctx->r1 + ctx->s11) ^ ctx->r2^ ctx->s12;

	ctx->s12 = a_mul(ctx->s12) ^ ctx->s14 ^ainv_mul(ctx->s7);
	fsmtmp = ctx->r2 + ctx->s1;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[12] = (ctx->r1 + ctx->s12) ^ ctx->r2^ ctx->s13;

	ctx->s13 = a_mul(ctx->s13) ^ ctx->s15 ^ainv_mul(ctx->s8);
	fsmtmp = ctx->r2 + ctx->s2;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[13] = (ctx->r1 + ctx->s13) ^ ctx->r2^ ctx->s14;

	ctx->s14 = a_mul(ctx->s14) ^ ctx->s0 ^ainv_mul(ctx->s9);
	fsmtmp = ctx->r2 + ctx->s3;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[14] = (ctx->r1 + ctx->s14) ^ ctx->r2^ ctx->s15;

	ctx->s15 = a_mul(ctx->s15) ^ ctx->s1 ^ainv_mul(ctx->s10);
	fsmtmp = ctx->r2 + ctx->s4;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[15] = (ctx->r1 + ctx->s15) ^ ctx->r2^ ctx->s0;

}


// Файл Snow.h

#pragma once

class Snow
{
public:
	Snow();

	~Snow();

	typedef struct
	{
		u32 keysize; // keysize = 128 or 256 bits, init vector = 128 bits

		BYTE key[32];

		u32 s15, s14, s13, s12, s11, s10, s9, s8, s7, s6, s5, s4, s3, s2, s1, s0;

		u32 r1, r2;
	}
	Ecrypt_ctx;

	void Ecrypt_init(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize, const 
BYTE* iv);

	void Ecrypt_keysetup(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize);
```

```
        void Ecrypt_ivsetup(Ecrypt_ctx* ctx, const BYTE* iv);

        void Ecrypt_process_bytes(
                int action,                                 /* 0 = encrypt; 1 = decrypt; */
                Ecrypt_ctx* ctx,
                const BYTE* input,
                BYTE* output,
                u32 msglen);                      /* Message length in bytes. */

        void snow_loadkey_fast(Ecrypt_ctx* ctx, u32 IV3, u32 IV2, u32 IV1, u32 IV0);

        void snow_keystream_fast(Ecrypt_ctx* ctx, u32* keystream_block);
};
```

Додаток Б

Програмний код реалізації РПШ з нелінійним випадковим кодуванням (варіант IKPS: “Isaac – Kupyna – Snow 2.0”)

Програмна реалізація РПШ IKPS виконана на ПК з процесором Intel(R) Core(TM) i3-6100, 3.7GHz та обсягом оперативної пам’яті 4 ГБ на базі 64-розрядної ОС Windows 7 Service Pack 1. Мова програмування – C++. Середовище розробки – Microsoft Visual Studio 2013.

```
// Файл Realization_2.cpp

#include "stdafx.h"
#include "General.h"
#include "IKS2.h"
#include "MihImai.h"
#include <time.h>

int main(int argc, char* argv[])
{
	General run;

	IKS2 iks2;

	int flen;

	clock_t start, finish;

	double duration;

	BYTE *in_file_buf;

	FILE *fp = fopen(argv[1], "rb");

	if(fp == NULL)
	{
		printf("ERROR!!! The test.to file was not opened!!! \n");

		return 0;
	}

	flen = run.get_file_len(fp);

	in_file_buf = new BYTE [flen];    //buffer for the input file

	memset(in_file_buf, 0, flen);

	int read_bytes = fread(in_file_buf, 1, flen, fp);

	if(flen != read_bytes)
	{
		printf("\n ERROR!!! It wasn't read all bytes\n ");
```

```
                return 0;
        }

        start = clock();

        iks2.EncrDecr(flen, in_file_buf, argv[1]); // read by 32 bytes

        finish = clock();

        duration = (double)(finish - start) / CLOCKS_PER_SEC;

        printf("\n elapsed time = %f seconds", duration);
        fclose(fp);
        return 0;
}

// Файл General.cpp

#include "StdAfx.h"
#include "General.h"
General::General(void)
{
}

General::~General(void)
{
}

void General::hex_print(const void* pv, int len)
{
        const BYTE * p = (const BYTE*)pv;

        if (NULL == pv)
                printf("NULL");
        else
        {
                for (int i = 0; i < len; ++i)
                {
                        if (i % 16 == 0)
                                printf("\n");

                        printf("%02X ", *p++);
                }
        }
        printf("\n");
}

int General::get_file_len(FILE* fp)
{
        int len;

        fseek(fp, 0, SEEK_END);

        len = ftell(fp);

        fseek(fp, 0, SEEK_SET);

        return len;
}
void General::NewName(char* fn1, char* fn2, const char* _ext)
{
        memset(fn2, 0, sizeof(fn2));
```

```
        strcpy(fn2, fn1);

        char *p = strchr(fn2, '.');

        if(p)
              strcpy(p, _ext);
        else
              strcat(fn2, _ext);
}

void General::WritetoFile(char* fname, BYTE *buf, int len)
{
        FILE *fout;

        if((fout = fopen(fname, "wb")) == NULL)
        {
              perror("open failed");
              printf("\n ERROR open file %s to write!!!", fname);
        }

        fwrite(buf, 1, len, fout);

        fclose(fout);
}


// Файл General.h

#pragma once
class General
{
public:
        General(void);
        ~General(void);
        static void hex_print(const void* pv, int len);

        int get_file_len(FILE* fp);

        void NewName(char* fn1, char* fn2, const char* _ext);

        void WritetoFile(char* fname, BYTE *buf, int len);
};


// Файл IKS2.cpp

#include "StdAfx.h"
#include "IKS2.h"
#include "Isaac64.h"
#include "General.h"
#include "Snow.h"
#include "Kupyna.h"

IKS2::IKS2(void)
{
}

IKS2::~IKS2(void)
{
}
void IKS2::EncrDecr(int len, BYTE* plain_txt, char *fto)
{
        char fts[25] = { 0 };
```

```
	char fdec[25] = { 0 };

	Snow snow_encr, snow_decr;

	Isaac64 isaac;

	Kupyna kupyna;

	General run;

	int remLen = len % 32;

	// make new name for encrypted and decrypted files
	run.NewName(fto, fts, ".ikps");

	run.NewName(fto, fdec, ".ikps_dec");

	BYTE vec[32], tmp, plain_byte[32];

	BYTE *hash, hash_code[32];

	Init(len); //allocate memory for encrypted and decrypted text

	__int64 *u64;

	// 1) ISAAC: output = 256 8-bytes numbers (__int64), total = 2048 bytes
	int cycles = (len / 2048) + 1;

	BYTE *u = new BYTE[256 * 8 * cycles];

	for (int cycl = 0; cycl < cycles; cycl++)
	{
		u64 = isaac.DoIsaac64();

		for (int i = 0; i < 256; i++)  // 256 * 8 bytes = 2048 bytes
		{
			for (int j = 0; j < 8; j++)
			{
				u[cycl + i * 8 + j] = ((u64[i] >> (j * 8)) & 0xff);
			}
		}
	}

	// 2)
	Snow::Ecrypt_ctx *ctx = new Snow::Ecrypt_ctx;

	Kupyna::kupyna_t *ctx2 = new Kupyna::kupyna_t;

	BYTE snow_key[32] = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef, 0x12, 0x34,
0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0,
                                            0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff, 0x33 },
              snow_IV[16]  = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef, 0x12, 0x34,
0x56, 0x78, 0x9a, 0xbc, 0xcd, 0xef };

	snow_encr.Ecrypt_init(ctx, snow_key, 256, 128, snow_IV);

	kupyna.KupynaInit(256, ctx2);

	// do enciphering ( m - l = l = 256 bits)
	// read the plain text in 32 bytes' blocks

	for (int i = 0; i < len / 32; i++)
```

```
	{
		kupyna.KupynaHash(ctx2, u + i * 32, 32, hash_code);

		// fill the first 32 bytes of the output vector by pseudorandom bytes u (m = 
512 bits: (m - l, l))
		for (int y = 0; y < 32; y++)
			vec_m[y] = u[i * 32 + y];

		// the second part of the output is s_i ^ fi(u_i)
		for (int j = 0; j < 32; j++)
			vec[j] = plain_txt[i * 32 + j] ^ hash_code[j];

		memcpy(vec_m + 32, vec, 32);

		// the result vector goes for enciphering
		snow_encr.Ecrypt_process_bytes(0, ctx, vec_m, enc_out + i * 64, 64);
	}

	// process the last block which is smaller than 32 bytes
	if (remLen)
	{
		kupyna.KupynaHash(ctx2, u + (len / 32) * 32, 32, hash_code);

		for (int y = 0; y < 32; y++)
			vec_m[y] = u[(len / 32) * 32 + y];

		for (int j = 0; j < remLen; j++)
			vec[j] = plain_txt[(len / 32) * 32 + j] ^ hash_code[j];

		memcpy(vec_m + 32, vec, remLen);
		snow_encr.Ecrypt_process_bytes(0, ctx, vec_m, enc_out + (len / 32) * 64, 32 + 
remLen);
	}

	run.WritetoFile(fts, enc_out, len * 2 - (32 - (len % 32)));  // write cipher text in 
file

	// D E C R Y P T I O N
	FILE *fout = fopen(fdec, "wb");

	// do INITIALIZATION
	snow_decr.Ecrypt_init(ctx, snow_key, 256, 128, snow_IV);

	for (int i = 0; i < len / 32; i++)
	{
		//do deciphering
		snow_decr.Ecrypt_process_bytes(1,ctx, enc_out + i * 64, dec_out + i * 64, 64);

		kupyna.KupynaHash(ctx2, dec_out + i * 64, 32, hash_code);

		for (int j = 0; j < 32; j++)
		{
			plain_byte[j] = hash_code[j] ^ dec_out[(i * 64 + 32) + j];

			fprintf(fout, "%c", plain_byte[j]);
		}
	}
	// decipher the last block
	if (remLen)
	{
		int bound = len / 32;
```

```
            snow_decr.Ecrypt_process_bytes(1, ctx, enc_out + bound * 64, dec_out + bound *
64, 32 + remLen);

            kupyna.KupynaHash(ctx2, dec_out + bound * 64, 32, hash_code);

            for (int j = 0; j < remLen; j++)
            {
                  plain_byte[j] = hash_code[j] ^ dec_out[(bound * 64 + 32) + j];

                  fprintf(fout, "%c", plain_byte[j]);
            }
      }

      fcloseall();

      delete[] u;
}

void IKS2::Init(int len)
{
      enc_out = new BYTE [len * 2];
      dec_out = new BYTE [len * 2];
      memset(enc_out, 0, sizeof(enc_out));
      memset(dec_out, 0, sizeof(dec_out));
}


// Файл IKS2.h

#pragma once

class IKS2
{
public:
      IKS2(void);
      ~IKS2(void);

      BYTE *enc_out; // encrypted text

      BYTE *dec_out; // decrypted text

      BYTE vec_m[64];

      void EncrDecr(int len, BYTE* plain_txt, char *fto);

      void Init(int len);
};


// Файл Isaac64.cpp

#include "StdAfx.h"
#include "Isaac64.h"


Isaac64::Isaac64(void)
{
}

Isaac64::~Isaac64(void)
{
}
```

```
void Isaac64::RandInit(int flag)
{
	__int64 a, b, c, d, e, f, g, h;

	aa = bb = cc = (__int64)0;

	a = b = c = d = e = f = g = h = 0x9e3779b97f4a7c13LL;

	for (int i = 0; i < 4; ++i)
	{
		mix(a, b, c, d, e, f, g, h);
	}

	for (int i = 0; i < RANDSIZ; i += 8)
	{
		if (flag)
		{
			a += randrsl[i];
			b += randrsl[i + 1];
			c += randrsl[i + 2];
			d += randrsl[i + 3];
			e += randrsl[i + 4];
			f += randrsl[i + 5];
			g += randrsl[i + 6];
			h += randrsl[i + 7];
		}
		mix(a, b, c, d, e, f, g, h);

		mm[i] = a;
		mm[i + 1] = b;
		mm[i + 2] = c;
		mm[i + 3] = d;
		mm[i + 4] = e;
		mm[i + 5] = f;
		mm[i + 6] = g;
		mm[i + 7] = h;
	}

	if (flag)
	{
		for (int i = 0; i < RANDSIZ; i += 8)
		{
			a += mm[i];
			b += mm[i + 1];
			c += mm[i + 2];
			d += mm[i + 3];
			e += mm[i + 4];
			f += mm[i + 5];
			g += mm[i + 6];
			h += mm[i + 7];

			mix(a, b, c, d, e, f, g, h);

			mm[i] = a;
			mm[i + 1] = b;
			mm[i + 2] = c;
			mm[i + 3] = d;
			mm[i + 4] = e;
			mm[i + 5] = f;
			mm[i + 6] = g;
			mm[i + 7] = h;
		}
	}
```

```
	RunIsaac64();

	randcnt = RANDSIZ;
}

void Isaac64::RunIsaac64()
{
	register __int64 a, b, x, y, *m, *m2, *r, *mend;
	m = mm;
	r = randrsl;
	a = aa;
	b = bb + (++cc);

	for (m = mm, mend = m2 = m + (RANDSIZ / 2); m < mend;)
	{
		rngstep(~(a ^ (a << 21)), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a >> 5), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a << 12), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a >> 33), a, b, mm, m, m2, r, x);
	}
	for (m2 = mm; m2<mend;)
	{
		rngstep(~(a ^ (a << 21)), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a >> 5), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a << 12), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a >> 33), a, b, mm, m, m2, r, x);
	}
	bb = b; aa = a;
}

__int64* Isaac64::DoIsaac64(void)
{
	aa = bb = cc = 0;

	for (int i = 0; i < RANDSIZ; ++i)
		mm[i] = 0;

	RandInit(TRUE);

	RunIsaac64();

	return randrsl;
}


// Файл Isaac64.h

#pragma once
#define mix(a, b, c, d, e, f, g, h) \
{ \
	a -= e;  f ^= h >> 9;   h += a; \
	b -= f;  g ^= a << 9;   a += b; \
	c -= g;  h ^= b >> 23;  b += c; \
	d -= h;  a ^= c << 15;  c += d; \
	e -= a;  b ^= d >> 14;  d += e; \
	f -= b;  c ^= e << 20;  e += f; \
	g -= c;  d ^= f >> 17;  f += g; \
	h -= d;  e ^= g << 14;  g += h; \
}

#define ind(mm, x)  (*(__int64 *)((ub1 *)(mm) + ((x) & ((RANDSIZ - 1) << 3))))
#define rngstep(mix, a, b, mm, m, m2, r, x) \
```

```
{ \
	x = *m;  \
	a = (mix)+*(m2++); \
	*(m++) = y = ind(mm, x) + a + b; \
	*(r++) = b = ind(mm, y >> RANDSIZL) + x; \
}

static __int64 mm[RANDSIZ];
static __int64 aa, bb, cc;

class Isaac64
{
public:
	Isaac64(void);
	~Isaac64(void);

	void RandInit(int flag);

	__int64 randrsl[RANDSIZ];

	__int64 randcnt;

	void RunIsaac64();

	__int64* DoIsaac64(void);
};

// Файл Kupyna.cpp

#include "stdafx.h"
#include "Kupyna.h"

Kupyna::Kupyna()
{
	uint8_t mds_matrix[8][8] = {
		{ 0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04 },
		{ 0x04, 0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07 },
		{ 0x07, 0x04, 0x01, 0x01, 0x05, 0x01, 0x08, 0x06 },
		{ 0x06, 0x07, 0x04, 0x01, 0x01, 0x05, 0x01, 0x08 },
		{ 0x08, 0x06, 0x07, 0x04, 0x01, 0x01, 0x05, 0x01 },
		{ 0x01, 0x08, 0x06, 0x07, 0x04, 0x01, 0x01, 0x05 },
		{ 0x05, 0x01, 0x08, 0x06, 0x07, 0x04, 0x01, 0x01 },
		{ 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04, 0x01 }
	};

	uint8_t sboxes[4][256] = {
		{
			0xa8, 0x43, 0x5f, 0x06, 0x6b, 0x75, 0x6c, 0x59, 0x71, 0xdf, 0x87, 0x95,
0x17, 0xf0, 0xd8, 0x09,
			0x6d, 0xf3, 0x1d, 0xcb, 0xc9, 0x4d, 0x2c, 0xaf, 0x79, 0xe0, 0x97, 0xfd,
0x6f, 0x4b, 0x45, 0x39,
			0x3e, 0xdd, 0xa3, 0x4f, 0xb4, 0xb6, 0x9a, 0x0e, 0x1f, 0xbf, 0x15, 0xe1,
0x49, 0xd2, 0x93, 0xc6,
			0x92, 0x72, 0x9e, 0x61, 0xd1, 0x63, 0xfa, 0xee, 0xf4, 0x19, 0xd5, 0xad,
0x58, 0xa4, 0xbb, 0xa1,
			0xdc, 0xf2, 0x83, 0x37, 0x42, 0xe4, 0x7a, 0x32, 0x9c, 0xcc, 0xab, 0x4a,
0x8f, 0x6e, 0x04, 0x27,
			0x2e, 0xe7, 0xe2, 0x5a, 0x96, 0x16, 0x23, 0x2b, 0xc2, 0x65, 0x66, 0x0f,
0xbc, 0xa9, 0x47, 0x41,
			0x34, 0x48, 0xfc, 0xb7, 0x6a, 0x88, 0xa5, 0x53, 0x86, 0xf9, 0x5b, 0xdb,
0x38, 0x7b, 0xc3, 0x1e,
			0x22, 0x33, 0x24, 0x28, 0x36, 0xc7, 0xb2, 0x3b, 0x8e, 0x77, 0xba, 0xf5,
0x14, 0x9f, 0x08, 0x55,
```

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

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

Kupyna::~Kupyna()
{
}
```

```
int Kupyna::KupynaInit(size_t hash_nbits, kupyna_t* ctx)
{
	if ((hash_nbits % 8 != 0) || (hash_nbits > 512))
	{
		return -1;
	}
	if (hash_nbits <= 256) {
		ctx->rounds = NR_512;
		ctx->columns = NB_512;
		ctx->nbytes = STATE_BYTE_SIZE_512;
	}
	else {
		ctx->rounds = NR_1024;
		ctx->columns = NB_1024;
		ctx->nbytes = STATE_BYTE_SIZE_1024;
	}

	ctx->hash_nbits = hash_nbits;

	memset(ctx->state, 0, ctx->nbytes);

	// Set init value according to the specification.
	ctx->state[0][0] = ctx->nbytes;

	return 0;
}

void Kupyna::SubBytes(uint8_t state[NB_1024][ROWS], int columns)
{
	int i, j;

	uint8_t temp[NB_1024];

	for (i = 0; i < ROWS; ++i)
	{
		for (j = 0; j < columns; ++j)
		{
			state[j][i] = sboxes[i % 4][state[j][i]];
		}
	}
}

void Kupyna::ShiftBytes(uint8_t state[NB_1024][ROWS], int columns)
{
	int i, j;

	uint8_t temp[NB_1024];

	int shift = -1;

	for (i = 0; i < ROWS; ++i)
	{
		if ((i == ROWS - 1) && (columns == NB_1024)) {
			shift = 11;
		}
		else {
			++shift;
		}
		for (j = 0; j < columns; ++j) {
			temp[(j + shift) % columns] = state[j][i];
		}
		for (j = 0; j < columns; ++j) {
```

```
                state[j][i] = temp[j];
            }
        }
}

uint8_t Kupyna::MultiplyGF(uint8_t x, uint8_t y)
{
        int i;

        uint8_t r = 0;

        uint8_t hbit = 0;

        for (i = 0; i < BITS_IN_BYTE; ++i)
        {
                if ((y & 0x1) == 1)
                        r ^= x;
                hbit = x & 0x80;
                x <<= 1;
                if (hbit == 0x80)
                        x ^= REDUCTION_POLYNOMIAL;
                y >>= 1;
        }
        return r;
}

void Kupyna::MixColumns(uint8_t state[NB_1024][ROWS], int columns)
{
        int i, row, col, b;

        uint8_t product;

        uint8_t result[ROWS];

        for (col = 0; col < columns; ++col)
        {
                memset(result, ROWS, 0);
                for (row = ROWS - 1; row >= 0; --row)
                {
                        product = 0;
                        for (b = ROWS - 1; b >= 0; --b) {
                                product ^= MultiplyGF(state[col][b], mds_matrix[row][b]);
                        }
                        result[row] = product;
                }
                for (i = 0; i < ROWS; ++i) {
                        state[col][i] = result[i];
                }
        }
}

void Kupyna::AddRoundConstantP(uint8_t state[NB_1024][ROWS], int columns, int round) {
        int i;

        for (i = 0; i < columns; ++i) {
                state[i][0] ^= (i * 0x10) ^ round;
        }
}

void Kupyna::AddRoundConstantQ(uint8_t state[NB_1024][ROWS], int columns, int round) {
        int j;

        uint64_t* s = (uint64_t*)state;
```

```
		for (j = 0; j < columns; ++j) {
			s[j] = s[j] + (0x00F0F0F0F0F0F0F3ULL ^
				((((columns - j - 1) * 0x10ULL) ^ round) << (7 * 8)));
		}
	}

	void Kupyna::P(kupyna_t* ctx, uint8_t state[NB_1024][ROWS]) {
		int i;
		for (i = 0; i < ctx->rounds; ++i) {
			AddRoundConstantP(state, ctx->columns, i);

			SubBytes(state, ctx->columns);

			ShiftBytes(state, ctx->columns);

			MixColumns(state, ctx->columns);
		}
	}

	void Kupyna::Q(kupyna_t* ctx, uint8_t state[NB_1024][ROWS]) {
		int i;
		for (i = 0; i < ctx->rounds; ++i) {
			AddRoundConstantQ(state, ctx->columns, i);

			SubBytes(state, ctx->columns);

			ShiftBytes(state, ctx->columns);

			MixColumns(state, ctx->columns);
		}
	}

	int Kupyna::Pad(kupyna_t* ctx, uint8_t* data, size_t msg_nbits) {
		int i;
		int mask;
		int pad_bit;
		int extra_bits;
		int zero_nbytes;
		size_t msg_nbytes = msg_nbits / BITS_IN_BYTE;
		size_t nblocks = msg_nbytes / ctx->nbytes;
		ctx->pad_nbytes = msg_nbytes - (nblocks * ctx->nbytes);
		ctx->data_nbytes = msg_nbytes - ctx->pad_nbytes;
		uint8_t* pad_start = data + ctx->data_nbytes;
		extra_bits = msg_nbits % BITS_IN_BYTE;
		if (extra_bits) {
			ctx->pad_nbytes += 1;
		}
		memcpy(ctx->padding, pad_start, ctx->pad_nbytes);
		extra_bits = msg_nbits % BITS_IN_BYTE;
		if (extra_bits) {
			mask = ~(0xFF >> (extra_bits));
			pad_bit = 1 << (7 - extra_bits);
			ctx->padding[ctx->pad_nbytes - 1] = (ctx->padding[ctx->pad_nbytes - 1] & mask)
	| pad_bit;
		}
		else {
			ctx->padding[ctx->pad_nbytes] = 0x80;
			ctx->pad_nbytes += 1;
		}
		zero_nbytes = ((-msg_nbits - 97) % (ctx->nbytes * BITS_IN_BYTE)) / BITS_IN_BYTE;
		memset(ctx->padding + ctx->pad_nbytes, 0, zero_nbytes);
		ctx->pad_nbytes += zero_nbytes;
		for (i = 0; i < (96 / 8); ++i, ++ctx->pad_nbytes) {
```

```
        if (i < sizeof(size_t)) {
            ctx->padding[ctx->pad_nbytes] = (msg_nbits >> (i * 8)) & 0xFF;
        }
        else {
            ctx->padding[ctx->pad_nbytes] = 0;
        }
    }
    return 0;
}

void Kupyna::Digest(kupyna_t* ctx, uint8_t* data) {
    int b, i, j;
    uint8_t temp1[NB_1024][ROWS];
    uint8_t temp2[NB_1024][ROWS];
    for (b = 0; b < ctx->data_nbytes; b += ctx->nbytes) {
        for (i = 0; i < ROWS; ++i) {
            for (j = 0; j < ctx->columns; ++j) {
                temp1[j][i] = ctx->state[j][i] ^ data[b + j * ROWS + i];
                temp2[j][i] = data[b + j * ROWS + i];
            }
        }
        P(ctx, temp1);

        Q(ctx, temp2);

        for (i = 0; i < ROWS; ++i) {
            for (j = 0; j < ctx->columns; ++j) {
                ctx->state[j][i] ^= temp1[j][i] ^ temp2[j][i];
            }
        }
    }
    /* Process extra bytes in padding. */
    for (b = 0; b < ctx->pad_nbytes; b += ctx->nbytes) {
        for (i = 0; i < ROWS; ++i) {
            for (j = 0; j < ctx->columns; ++j) {
                temp1[j][i] = ctx->state[j][i] ^ ctx->padding[b + j * ROWS + i];
                temp2[j][i] = ctx->padding[b + j * ROWS + i];
            }
        }
        P(ctx, temp1);

        Q(ctx, temp2);

        for (i = 0; i < ROWS; ++i) {
            for (j = 0; j < ctx->columns; ++j) {
                ctx->state[j][i] ^= temp1[j][i] ^ temp2[j][i];
            }
        }
    }
}

void Kupyna::Trunc(kupyna_t* ctx, uint8_t* hash_code) {
    int i;
    size_t hash_nbytes = ctx->hash_nbits / BITS_IN_BYTE;

    memcpy(hash_code, (uint8_t*)ctx->state + ctx->nbytes - hash_nbytes, hash_nbytes);
}

void Kupyna::OutputTransformation(kupyna_t* ctx, uint8_t* hash_code) {
    int i, j;

    uint8_t temp[NB_1024][ROWS];
```

```
	memcpy(temp, ctx->state, ROWS * NB_1024);

	P(ctx, temp);

	for (i = 0; i < ROWS; ++i) {
		for (j = 0; j < ctx->columns; ++j) {
			ctx->state[j][i] ^= temp[j][i];
		}
	}
	Trunc(ctx, hash_code);
}

void Kupyna::KupynaHash(kupyna_t* ctx, uint8_t* data, size_t msg_bit_len, uint8_t*
hash_code) {
	/* Reinitialize internal state. */
	memset(ctx->state, 0, ctx->nbytes);
	ctx->state[0][0] = ctx->nbytes;

	Pad(ctx, data, msg_bit_len);
	Digest(ctx, data);
	OutputTransformation(ctx, hash_code);
}


// Файл Kupyna.h

#pragma once

#define ROWS 8
#define NB_512 8  ///< Number of 8-byte words in state for <=256-bit hash code.
#define NB_1024 16  ///< Number of 8-byte words in state for <=512-bit hash code.
#define STATE_BYTE_SIZE_512 (ROWS * NB_512)
#define STATE_BYTE_SIZE_1024 (ROWS * NB_1024)
#define NR_512 10  ///< Number of rounds for 512-bit state.
#define NR_1024 14  ///< Number of rounds for 1024-bit state.
#define REDUCTION_POLYNOMIAL 0x011d  /* x^8 + x^4 + x^3 + x^2 + 1 */
#define BITS_IN_WORD 64
#define BITS_IN_BYTE 8

class Kupyna
{
public:
	Kupyna();
	~Kupyna();

	typedef struct
	{
		uint8_t state[NB_1024][ROWS];  ///< Hash function internal state (of maximum
possible size to fit for all modes of operation).
		size_t nbytes;  ///< Number of bytes currently located in state.
		size_t data_nbytes;  ///< Number of bytes in input data sequence.
		uint8_t padding[STATE_BYTE_SIZE_1024 * 2];  ///< Space for extra bytes and
padding.
		size_t pad_nbytes;  ///< Number of bytes currently located in padding buffer.
		size_t hash_nbits;  ///< Hash code bit length.
		int columns;  ///< Number of columns (8-byte vectors) located in internal
state.
		int rounds;  ///< Number of rounds for current mode of operation.
	} kupyna_t;

	uint8_t mds_matrix[8][8];
	uint8_t sboxes[4][256];
	int KupynaInit(size_t hash_nbits, kupyna_t* ctx);
```

```
	void SubBytes(uint8_t state[NB_1024][ROWS], int columns);
	void ShiftBytes(uint8_t state[NB_1024][ROWS], int columns);
	uint8_t MultiplyGF(uint8_t x, uint8_t y);
	void MixColumns(uint8_t state[NB_1024][ROWS], int columns);
	void AddRoundConstantP(uint8_t state[NB_1024][ROWS], int columns, int round);
	void AddRoundConstantQ(uint8_t state[NB_1024][ROWS], int columns, int round);
	void P(kupyna_t* ctx, uint8_t state[NB_1024][ROWS]);
	void Q(kupyna_t* ctx, uint8_t state[NB_1024][ROWS]);
	int Pad(kupyna_t* ctx, uint8_t* data, size_t msg_nbits);
	void Digest(kupyna_t* ctx, uint8_t* data);
	void Trunc(kupyna_t* ctx, uint8_t* hash_code);
	void OutputTransformation(kupyna_t* ctx, uint8_t* hash_code);
	void KupynaHash(kupyna_t* ctx, uint8_t* data, size_t msg_bit_len, uint8_t*
hash_code);

};


// Файл Snow.cpp

#include "stdafx.h"
#include "Snow.h"
#include "snowtab.h"
#include "General.h"

Snow::Snow()
{
}

Snow::~Snow()
{
}

/* Key and message independent initialization. This function will be
* called once when the program starts (e.g., to build expanded S-box
* tables).
*/
void Snow::Ecrypt_init(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize, const
BYTE* iv)
{
	Ecrypt_keysetup(ctx, key, keysize, ivsize);

	Ecrypt_ivsetup(ctx, iv);
}

/*
* Key setup. It is the user's responsibility to select the values of
* keysize and ivsize from the set of supported values specified
* above.
*/
void Snow::Ecrypt_keysetup(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize)
{
	for (int i = 0; i < keysize / 8; ++i)
		ctx->key[i] = key[i];

	ctx->keysize = keysize;
}

/* IV setup. After having called ECRYPT_keysetup(), the user is
* allowed to call ECRYPT_ivsetup() different times in order to
* encrypt/decrypt different messages with the same key but different
* IV's. */
void Snow::Ecrypt_ivsetup(Ecrypt_ctx* ctx, const BYTE* iv)
```

```
{
	snow_loadkey_fast(ctx, U8TO32_LITTLE(iv), U8TO32_LITTLE(iv + 4), U8TO32_LITTLE(iv + 
8), U8TO32_LITTLE(iv + 12));
}


void Snow::Ecrypt_process_bytes(int action, Ecrypt_ctx* ctx, const BYTE* input, BYTE* 
output, u32 msglen)
{
	u32 i;
	u32 keystream[16];
	u32 tmp;

	for (; msglen >= 64; msglen -= 64, input += 64, output += 64)
	{
		snow_keystream_fast(ctx, keystream); // keystream = 64 bytes

		for (i = 0; i < 16; ++i)
		{
			tmp = ((u32*)input)[i];
			((u32*)output)[i] = tmp ^ U32TO32_LITTLE(keystream[i]);
		}

	}

	if (msglen > 0)
	{
		snow_keystream_fast(ctx, keystream);

		for (i = 0; i < msglen; i++)
			output[i] = input[i] ^ ((BYTE*)keystream)[i];
	}
}


// Function:  snow_loadkey_fast
void Snow::snow_loadkey_fast(Ecrypt_ctx* ctx, u32 IV3, u32 IV2, u32 IV1, u32 IV0)
{
	int i;

	if (ctx->keysize == 128)
	{
		ctx->s15 = (((u32)*(ctx->key + 0)) << 24) | (((u32)*(ctx->key + 1)) << 16) |
			(((u32)*(ctx->key + 2)) << 8) | (((u32)*(ctx->key + 3)));
		ctx->s14 = (((u32)*(ctx->key + 4)) << 24) | (((u32)*(ctx->key + 5)) << 16) |
			(((u32)*(ctx->key + 6)) << 8) | (((u32)*(ctx->key + 7)));
		ctx->s13 = (((u32)*(ctx->key + 8)) << 24) | (((u32)*(ctx->key + 9)) << 16) |
			(((u32)*(ctx->key + 10)) << 8) | (((u32)*(ctx->key + 11)));
		ctx->s12 = (((u32)*(ctx->key + 12)) << 24) | (((u32)*(ctx->key + 13)) << 16) |
			(((u32)*(ctx->key + 14)) << 8) | (((u32)*(ctx->key + 15)));
		ctx->s11 = ~ctx->s15;  /* bitwise inverse */
		ctx->s10 = ~ctx->s14;
		ctx->s9 = ~ctx->s13;
		ctx->s8 = ~ctx->s12;
		ctx->s7 = ctx->s15;    /* just copy */
		ctx->s6 = ctx->s14;
		ctx->s5 = ctx->s13;
		ctx->s4 = ctx->s12;
		ctx->s3 = ~ctx->s15;   /* bitwise inverse */
		ctx->s2 = ~ctx->s14;
		ctx->s1 = ~ctx->s13;
		ctx->s0 = ~ctx->s12;
	}
```

```
	else
	{
		// assume keysize=256
		ctx->s15 = (((u32)*(ctx->key + 0)) << 24) | (((u32)*(ctx->key + 1)) << 16) |
			(((u32)*(ctx->key + 2)) << 8) | (((u32)*(ctx->key + 3)));
		ctx->s14 = (((u32)*(ctx->key + 4)) << 24) | (((u32)*(ctx->key + 5)) << 16) |
			(((u32)*(ctx->key + 6)) << 8) | (((u32)*(ctx->key + 7)));
		ctx->s13 = (((u32)*(ctx->key + 8)) << 24) | (((u32)*(ctx->key + 9)) << 16) |
			(((u32)*(ctx->key + 10)) << 8) | (((u32)*(ctx->key + 11)));
		ctx->s12 = (((u32)*(ctx->key + 12)) << 24) | (((u32)*(ctx->key + 13)) << 16) |
			(((u32)*(ctx->key + 14)) << 8) | (((u32)*(ctx->key + 15)));
		ctx->s11 = (((u32)*(ctx->key + 16)) << 24) | (((u32)*(ctx->key + 17)) << 16) |
			(((u32)*(ctx->key + 18)) << 8) | (((u32)*(ctx->key + 19)));
		ctx->s10 = (((u32)*(ctx->key + 20)) << 24) | (((u32)*(ctx->key + 21)) << 16) |
			(((u32)*(ctx->key + 22)) << 8) | (((u32)*(ctx->key + 23)));
		ctx->s9 = (((u32)*(ctx->key + 24)) << 24) | (((u32)*(ctx->key + 25)) << 16) |
			(((u32)*(ctx->key + 26)) << 8) | (((u32)*(ctx->key + 27)));
		ctx->s8 = (((u32)*(ctx->key + 28)) << 24) | (((u32)*(ctx->key + 29)) << 16) |
			(((u32)*(ctx->key + 30)) << 8) | (((u32)*(ctx->key + 31)));
		ctx->s7 = ~ctx->s15; /* bitwise inverse */
		ctx->s6 = ~ctx->s14;
		ctx->s5 = ~ctx->s13;
		ctx->s4 = ~ctx->s12;
		ctx->s3 = ~ctx->s11;
		ctx->s2 = ~ctx->s10;
		ctx->s1 = ~ctx->s9;
		ctx->s0 = ~ctx->s8;
	}

	// XOR IV values
	ctx->s15 ^= IV0;
	ctx->s12 ^= IV1;
	ctx->s10 ^= IV2;
	ctx->s9 ^= IV3;

	ctx->r1 = 0;
	ctx->r2 = 0;

	// Do 32 initial clockings
	for (i = 0; i < 2; i++)
	{
		u32 outfrom_fsm, fsmtmp;

		outfrom_fsm = (ctx->r1 + ctx->s15) ^ ctx->r2;
		ctx->s0 = a_mul(ctx->s0) ^ ctx->s2 ^ ainv_mul(ctx->s11) ^ outfrom_fsm;
		fsmtmp = ctx->r2 + ctx->s5;
		ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
		ctx->r1 = fsmtmp;

		outfrom_fsm = (ctx->r1 + ctx->s0) ^ ctx->r2;
		ctx->s1 = a_mul(ctx->s1) ^ ctx->s3 ^ainv_mul(ctx->s12) ^ outfrom_fsm;
		fsmtmp = ctx->r2 + ctx->s6;
		ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
		ctx->r1 = fsmtmp;

		outfrom_fsm = (ctx->r1 + ctx->s1) ^ ctx->r2;
		ctx->s2 = a_mul(ctx->s2) ^ ctx->s4 ^ainv_mul(ctx->s13) ^ outfrom_fsm;
		fsmtmp = ctx->r2 + ctx->s7;
		ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
		ctx->r1 = fsmtmp;
```

```
            outfrom_fsm = (ctx->r1 + ctx->s2) ^ ctx->r2;
            ctx->s3 = a_mul(ctx->s3) ^ ctx->s5 ^ainv_mul(ctx->s14) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s8;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s3) ^ ctx->r2;
            ctx->s4 = a_mul(ctx->s4) ^ ctx->s6 ^ainv_mul(ctx->s15) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s9;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s4) ^ ctx->r2;
            ctx->s5 = a_mul(ctx->s5) ^ ctx->s7 ^ainv_mul(ctx->s0) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s10;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s5) ^ ctx->r2;
            ctx->s6 = a_mul(ctx->s6) ^ ctx->s8 ^ainv_mul(ctx->s1) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s11;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s6) ^ ctx->r2;
            ctx->s7 = a_mul(ctx->s7) ^ ctx->s9 ^ainv_mul(ctx->s2) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s12;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s7) ^ ctx->r2;
            ctx->s8 = a_mul(ctx->s8) ^ ctx->s10 ^ainv_mul(ctx->s3) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s13;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s8) ^ ctx->r2;
            ctx->s9 = a_mul(ctx->s9) ^ ctx->s11 ^ainv_mul(ctx->s4) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s14;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s9) ^ ctx->r2;
            ctx->s10 = a_mul(ctx->s10) ^ ctx->s12 ^ainv_mul(ctx->s5) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s15;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s10) ^ ctx->r2;
            ctx->s11 = a_mul(ctx->s11) ^ ctx->s13 ^ainv_mul(ctx->s6) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s0;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;
```

```
            outfrom_fsm = (ctx->r1 + ctx->s11) ^ ctx->r2;
            ctx->s12 = a_mul(ctx->s12) ^ ctx->s14 ^ainv_mul(ctx->s7) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s1;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s12) ^ ctx->r2;
            ctx->s13 = a_mul(ctx->s13) ^ ctx->s15 ^ainv_mul(ctx->s8) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s2;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s13) ^ ctx->r2;
            ctx->s14 = a_mul(ctx->s14) ^ ctx->s0 ^ainv_mul(ctx->s9) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s3;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s14) ^ ctx->r2;
            ctx->s15 = a_mul(ctx->s15) ^ ctx->s1 ^ainv_mul(ctx->s10) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s4;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;
      }
}

// Function: snow_keystream_fast

void Snow::snow_keystream_fast(Ecrypt_ctx* ctx, u32* keystream_block)
{
      u32 fsmtmp;

      ctx->s0 = a_mul(ctx->s0) ^ ctx->s2 ^ainv_mul(ctx->s11);
      fsmtmp = ctx->r2 + ctx->s5;
      ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
      ctx->r1 = fsmtmp;
      keystream_block[0] = (ctx->r1 + ctx->s0) ^ ctx->r2^ ctx->s1;

      ctx->s1 = a_mul(ctx->s1) ^ ctx->s3 ^ainv_mul(ctx->s12);
      fsmtmp = ctx->r2 + ctx->s6;
      ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
      ctx->r1 = fsmtmp;
      keystream_block[1] = (ctx->r1 + ctx->s1) ^ ctx->r2^ ctx->s2;

      ctx->s2 = a_mul(ctx->s2) ^ ctx->s4 ^ainv_mul(ctx->s13);
      fsmtmp = ctx->r2 + ctx->s7;
      ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
      ctx->r1 = fsmtmp;
      keystream_block[2] = (ctx->r1 + ctx->s2) ^ ctx->r2^ ctx->s3;

      ctx->s3 = a_mul(ctx->s3) ^ ctx->s5 ^ainv_mul(ctx->s14);
      fsmtmp = ctx->r2 + ctx->s8;
      ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
      ctx->r1 = fsmtmp;
```

```
	keystream_block[3] = (ctx->r1 + ctx->s3) ^ ctx->r2^ ctx->s4;

	ctx->s4 = a_mul(ctx->s4) ^ ctx->s6 ^ainv_mul(ctx->s15);
	fsmtmp = ctx->r2 + ctx->s9;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[4] = (ctx->r1 + ctx->s4) ^ ctx->r2^ ctx->s5;

	ctx->s5 = a_mul(ctx->s5) ^ ctx->s7 ^ainv_mul(ctx->s0);
	fsmtmp = ctx->r2 + ctx->s10;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[5] = (ctx->r1 + ctx->s5) ^ ctx->r2^ ctx->s6;

	ctx->s6 = a_mul(ctx->s6) ^ ctx->s8 ^ainv_mul(ctx->s1);
	fsmtmp = ctx->r2 + ctx->s11;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[6] = (ctx->r1 + ctx->s6) ^ ctx->r2^ ctx->s7;

	ctx->s7 = a_mul(ctx->s7) ^ ctx->s9 ^ainv_mul(ctx->s2);
	fsmtmp = ctx->r2 + ctx->s12;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[7] = (ctx->r1 + ctx->s7) ^ ctx->r2^ ctx->s8;

	ctx->s8 = a_mul(ctx->s8) ^ ctx->s10 ^ainv_mul(ctx->s3);
	fsmtmp = ctx->r2 + ctx->s13;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[8] = (ctx->r1 + ctx->s8) ^ ctx->r2^ ctx->s9;

	ctx->s9 = a_mul(ctx->s9) ^ ctx->s11 ^ainv_mul(ctx->s4);
	fsmtmp = ctx->r2 + ctx->s14;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[9] = (ctx->r1 + ctx->s9) ^ ctx->r2^ ctx->s10;

	ctx->s10 = a_mul(ctx->s10) ^ ctx->s12 ^ainv_mul(ctx->s5);
	fsmtmp = ctx->r2 + ctx->s15;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[10] = (ctx->r1 + ctx->s10) ^ ctx->r2^ ctx->s11;

	ctx->s11 = a_mul(ctx->s11) ^ ctx->s13 ^ainv_mul(ctx->s6);
	fsmtmp = ctx->r2 + ctx->s0;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[11] = (ctx->r1 + ctx->s11) ^ ctx->r2^ ctx->s12;

	ctx->s12 = a_mul(ctx->s12) ^ ctx->s14 ^ainv_mul(ctx->s7);
	fsmtmp = ctx->r2 + ctx->s1;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
```

```
	keystream_block[12] = (ctx->r1 + ctx->s12) ^ ctx->r2^ ctx->s13;

	ctx->s13 = a_mul(ctx->s13) ^ ctx->s15 ^ainv_mul(ctx->s8);
	fsmtmp = ctx->r2 + ctx->s2;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[13] = (ctx->r1 + ctx->s13) ^ ctx->r2^ ctx->s14;

	ctx->s14 = a_mul(ctx->s14) ^ ctx->s0 ^ainv_mul(ctx->s9);
	fsmtmp = ctx->r2 + ctx->s3;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[14] = (ctx->r1 + ctx->s14) ^ ctx->r2^ ctx->s15;

	ctx->s15 = a_mul(ctx->s15) ^ ctx->s1 ^ainv_mul(ctx->s10);
	fsmtmp = ctx->r2 + ctx->s4;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2, 
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[15] = (ctx->r1 + ctx->s15) ^ ctx->r2^ ctx->s0;
}


// Файл Snow.h

#pragma once
class Snow
{
public:
	Snow();
	~Snow();

	typedef struct
	{
		u32 keysize; // keysize = 128 or 256 bits, init vector = 128 bits
		BYTE key[32];

		u32 s15, s14, s13, s12, s11, s10, s9, s8, s7, s6, s5, s4, s3, s2, s1, s0;
		u32 r1, r2;
	} Ecrypt_ctx;

	void Ecrypt_init(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize, const 
BYTE* iv);

	void Ecrypt_keysetup(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize);

	void Ecrypt_ivsetup(Ecrypt_ctx* ctx, const BYTE* iv);

	void Ecrypt_process_bytes(
		int action,                              /* 0 = encrypt; 1 = decrypt; */
		Ecrypt_ctx* ctx,
		const BYTE* input,
		BYTE* output,
		u32 msglen);                      /* Message length in bytes. */

	void snow_loadkey_fast(Ecrypt_ctx* ctx, u32 IV3, u32 IV2, u32 IV1, u32 IV0);

	void snow_keystream_fast(Ecrypt_ctx* ctx, u32* keystream_block);
};
```

Додаток В

Програмний код реалізації РПШ з нелінійним випадковим кодуванням (варіант INLS: “Isaac – NonLinearMap – Snow 2.0”)

Програмна реалізація РПШ INLS виконана на ПК з процесором Intel(R) Core(TM) i3-6100, 3.7GHz та обсягом оперативної пам’яті 4 ГБ на базі 64-розрядної ОС Windows 7 Service Pack 1. Мова програмування – C++. Середовище розробки – Microsoft Visual Studio 2013.

```
// Файл Realization_3.cpp

#include "stdafx.h"

#include "General.h"
#include "IKS2.h"
#include "MihImai.h"
#include <time.h>
#include "Kupyna.h"

int main(int argc, char* argv[])
{
	General run;

	IKS2 iks2;

	int flen;

	clock_t start, finish;

	double duration;

	BYTE *in_file_buf;

	FILE *fp = fopen(argv[1], "rb");

	if(fp == NULL)
	{
		printf("ERROR!!! The test.to file was not opened!!! \n");

		return 0;
	}

	flen = run.get_file_len(fp);

	in_file_buf = new BYTE [flen];    //buffer for the input file

	memset(in_file_buf, 0, flen);

	int read_bytes = fread(in_file_buf, 1, flen, fp);

	if(flen != read_bytes)
```

```
	{
		printf("\n ERROR!!! It wasn't read all bytes\n ");
		return 0;
	}

	start = clock();

	iks2.EncrDecr(flen, in_file_buf, argv[1]); // read by 32 bytes

	finish = clock();

	duration = (double)(finish - start) / CLOCKS_PER_SEC;

	printf("\n elapsed time = %f seconds", duration);

	fclose(fp);

	return 0;
}


// Файл General.cpp

#include "StdAfx.h"
#include "General.h"

General::General(void)
{
}

General::~General(void)
{
}
void General::hex_print(const void* pv, int len)
{
	const BYTE * p = (const BYTE*)pv;
	if (NULL == pv)
		printf("NULL");
	else
	{
		for (int i = 0; i < len; ++i)
		{
			if (i % 16 == 0)
				printf("\n");
			printf("%02X ", *p++);
		}
	}
	printf("\n");
}

int General::get_file_len(FILE* fp)
{
	int len;

	fseek(fp, 0, SEEK_END);

	len = ftell(fp);

	fseek(fp, 0, SEEK_SET);

	return len;
}
```

```
void General::NewName(char* fn1, char* fn2, const char* _ext)
{
	memset(fn2, 0, sizeof(fn2));

	strcpy(fn2, fn1);

	char *p = strchr(fn2, '.');

	if(p)
		strcpy(p, _ext);
	else
		strcat(fn2, _ext);
}

void General::WritetoFile(char* fname, BYTE *buf, int len)
{
	FILE *fout;

	if((fout = fopen(fname, "wb")) == NULL)
	{
		perror("open failed");

		printf("\n ERROR open file %s to write!!!", fname);
	}

	fwrite(buf, 1, len, fout);

	fclose(fout);
}


// Файл General.h

#pragma once

class General
{
public:
	General(void);
	~General(void);

	static void hex_print(const void* pv, int len);

	int get_file_len(FILE* fp);

	void NewName(char* fn1, char* fn2, const char* _ext);

	void WritetoFile(char* fname, BYTE *buf, int len);
};


// Файл IKS2.cpp

#include "StdAfx.h"
#include "IKS2.h"
#include "Isaac64.h"
#include "Keccak.h"
#include "General.h"
#include "Snow.h"
#include "Kupyna.h"
IKS2::IKS2(void)
{
}
```

```
IKS2::~IKS2(void)
{
}

void IKS2::EncrDecr(int len, BYTE* plain_txt, char *fto)
{
	char fts[25] = { 0 };

	char fdec[25] = { 0 };

	Snow snow_encr, snow_decr;

	Isaac64 isaac;

	General run;

	int POWER = 193;

	int remLen = len % 32;

	// make new name for encrypted and decrypted files
	run.NewName(fto, fts, ".ikps");

	run.NewName(fto, fdec, ".ikps_dec");

	BYTE vec[32], tmp, plain_byte[32];

	BYTE *hash, hash_code[32], save[32];

	memset(hash_code, 0, sizeof(hash_code));

	memset(save, 0, sizeof(save));

	Init(len); //allocate memory for encrypted and decrypted text

	__int64 *u64;

	// 1) ISAAC: output = 256 8-bytes numbers (__int64), total = 2048 bytes
	int cycles = (len / 2048) + 1;

	BYTE *u = new BYTE[256 * 8 * cycles];

	for (int cycl = 0; cycl < cycles; cycl++)
	{
		u64 = isaac.DoIsaac64();

		for (int i = 0; i < 256; i++)  // 256 * 8 bytes = 2048 bytes
		{
			for (int j = 0; j < 8; j++)
			{
				u[cycl + i * 8 + j] = ((u64[i] >> (j * 8)) & 0xff);
			}
		}
	}

	// 2)
	Snow::Ecrypt_ctx *ctx = new Snow::Ecrypt_ctx;

	BYTE snow_key[32] = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef, 0x12, 0x34, 
0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0,
					   0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8, 
0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff, 0x33 },
```

```
            snow_IV[16]  = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef, 0x12, 0x34,
0x56, 0x78, 0x9a, 0xbc, 0xcd, 0xef };

       snow_encr.Ecrypt_init(ctx, snow_key, 256, 128, snow_IV);


       // do enciphering ( m - l = l = 256 bits)
       // read the plain text in 32 bytes' blocks
       for (int i = 0; i < len / 32; i++)
       {
              printf("\r %d from %d", i, len / 32);

              memcpy(save, &u[i * 32], 32);

             NonLinPerm(save, hash_code, POWER);

        // fill the first 32 bytes of the output vector by pseudorandom bytes u (m = 512
bits: (m - l, l))
              for (int y = 0; y < 32; y++)
                     vec_m[y] = u[i * 32 + y];

              // the second part of the output is s_i ^ fi(u_i)
              for (int j = 0; j < 32; j++)
                     vec[j] = plain_txt[i * 32 + j] ^ hash_code[j];

              memcpy(vec_m + 32, vec, 32);

              // the result vector goes for enciphering
              snow_encr.Ecrypt_process_bytes(0, ctx, vec_m, enc_out + i * 64, 64);

       }

       // process the last block which is smaller than 32 bytes
       if (remLen)
       {
              printf("----- REMLEN\n\n");

              NonLinPerm(&u[(len / 32) * 32], hash_code, POWER);

              for (int y = 0; y < 32; y++)
                     vec_m[y] = u[(len / 32) * 32 + y];

              for (int j = 0; j < remLen; j++)
                     vec[j] = plain_txt[(len / 32) * 32 + j] ^ hash_code[j];

              memcpy(vec_m + 32, vec, remLen);

              snow_encr.Ecrypt_process_bytes(0, ctx, vec_m, enc_out + (len / 32) * 64, 32 +
remLen);

       }

       run.WritetoFile(fts, enc_out, len * 2 - (32 - (len % 32)));  // write cipher text in
file

      // D E C R Y P T I O N
      FILE *fout = fopen(fdec, "wb");

      printf("\n\n DECRYPTED\n\n");

      // do INITIALIZATION
      snow_decr.Ecrypt_init(ctx, snow_key, 256, 128, snow_IV);
```

```
	for (int i = 0; i < len / 32; i++)
	{
		printf("\r %d from %d", i, len/32);

		//do deciphering
		snow_decr.Ecrypt_process_bytes(1, ctx, enc_out + i * 64, dec_out + i * 64,
64);

	 memcpy(save, &dec_out[i * 64], 32);

	 NonLinPerm(save, hash_code, POWER);

	 for (int j = 0; j < 32; j++)
		{
			plain_byte[j] = hash_code[j] ^ dec_out[(i * 64 + 32) + j];

			fprintf(fout, "%c", plain_byte[j]);
		}
	}



	printf("\n\n REMLEN\n\n");
	// decipher the last block
	if (remLen)
	{
		int bound = len / 32;

		snow_decr.Ecrypt_process_bytes(1, ctx, enc_out + bound * 64, dec_out + bound *
64, 32 + remLen);

	 NonLinPerm(dec_out + bound * 64, hash_code, POWER);

	 for (int j = 0; j < remLen; j++)
		{
			plain_byte[j] = hash_code[j] ^ dec_out[(bound * 64 + 32) + j];

			fprintf(fout, "%c", plain_byte[j]);
		}
	}

	fcloseall();

	delete[] u;
}

void IKS2::Init(int len)
{
	enc_out = new BYTE [len * 2];

	dec_out = new BYTE [len * 2];

	memset(enc_out, 0, sizeof(enc_out));

	memset(dec_out, 0, sizeof(dec_out));

	memset(irreduc_pol, 0, sizeof(irreduc_pol));

	irreduc_pol[0]  = 0x25;
	irreduc_pol[1]  = 0x04;
	irreduc_pol[33] = 0x01;
}
```

```
BYTE* IKS2::MultGF256(BYTE* x, BYTE* res)
{
	BYTE hbit = 0;

	BYTE *y = new BYTE[32];

	memcpy(y, x, 32);

	BYTE save;

	BYTE byte32[32];

	memset(byte32, 0, sizeof(byte32));

	for (int i = 0; i < 32; i++)
	{
	 for (int j = 0; j < BITS_IN_BYTE; ++j)
		{
			//1.------------------------------------
			if ((y[0] & 0x01) == 1)
			{
				for (int k = 0; k < 32; k++)
					res[k] ^= x[k];
			}

			//2.-------------------------------------
			hbit = x[31] & 0x80;

			//3.---------------------------------------
			//shift left the whole 32-bytes register x[32]
			for (int k = 31; k > 0; k--)
			{
				save = (x[k] << 1) ^ ((x[k - 1] & 0x80) >> 7);
				byte32[k] = save;
			}
			byte32[0] = x[0] << 1;

			memcpy(x, byte32, sizeof(byte32));

			//4.---------------------------------------
			if (hbit == 0x80)
			{
				for (int k = 0; k < 32; k++)
					x[k] ^= irreduc_pol[k];
			}

			memset(byte32, 0, sizeof(byte32));

			//5.---------------------------------------------
			// shift right the whole 32-bytes register y[32]
			for (int k = 0; k < 31; k++)
			{
				save = (y[k] >> 1) ^ ((y[k + 1] & 0x01) << 7);
				byte32[k] = save;
			}
			byte32[31] = y[31] >> 1;

			memcpy(y, byte32, sizeof(byte32));
		}
	}

	 delete[] y;
```

```
    return res;
}

BYTE* IKS2::NonLinPerm(BYTE* x, BYTE* res, int POWER)
{
       BYTE *y = new BYTE[32];

       memcpy(y, x, 32);

       memset(res, 0, 32);

       for (int i = 0; i < POWER; i++)
       {
              MultGF256(y, res);

        memcpy(y, res, sizeof(y));
       }

       delete[] y;

       return res;
}

// Файл IKS2.h

#pragma once

class IKS2
{
public:
       IKS2(void);
       ~IKS2(void);

       BYTE *enc_out; // encrypted text

       BYTE *dec_out; // decrypted text

       BYTE irreduc_pol[33];

       BYTE vec_m[64];

       void EncrDecr(int len, BYTE* plain_txt, char *fto);

       void Init(int len);

       BYTE* MultGF256(BYTE* x, BYTE* res);

       BYTE* NonLinPerm(BYTE* x, BYTE* res, int POWER);
};

// Файл Isaac64.cpp

#include "StdAfx.h"
#include "Isaac64.h"


Isaac64::Isaac64(void)
{
}

Isaac64::~Isaac64(void)
{
```

```
}

void Isaac64::RandInit(int flag)
{
	__int64 a, b, c, d, e, f, g, h;

	aa = bb = cc = (__int64)0;
	a = b = c = d = e = f = g = h = 0x9e3779b97f4a7c13LL;

	for (int i = 0; i < 4; ++i)
		mix(a, b, c, d, e, f, g, h);
	for (int i = 0; i < RANDSIZ; i += 8)
	{
		if (flag)
		{
			a += randrsl[i];
			b += randrsl[i + 1];
			c += randrsl[i + 2];
			d += randrsl[i + 3];
			e += randrsl[i + 4];
			f += randrsl[i + 5];
			g += randrsl[i + 6];
			h += randrsl[i + 7];
		}
		mix(a, b, c, d, e, f, g, h);

		mm[i] = a;
		mm[i + 1] = b;
		mm[i + 2] = c;
		mm[i + 3] = d;
		mm[i + 4] = e;
		mm[i + 5] = f;
		mm[i + 6] = g;
		mm[i + 7] = h;
	}

	if (flag)
	{
		for (int i = 0; i < RANDSIZ; i += 8)
		{
			a += mm[i];
			b += mm[i + 1];
			c += mm[i + 2];
			d += mm[i + 3];
			e += mm[i + 4];
			f += mm[i + 5];
			g += mm[i + 6];
			h += mm[i + 7];

			mix(a, b, c, d, e, f, g, h);

			mm[i] = a;
			mm[i + 1] = b;
			mm[i + 2] = c;
			mm[i + 3] = d;
			mm[i + 4] = e;
			mm[i + 5] = f;
			mm[i + 6] = g;
			mm[i + 7] = h;
		}
	}

	RunIsaac64();
```

```
	randcnt = RANDSIZ;
}


void Isaac64::RunIsaac64()
{
	register __int64 a, b, x, y, *m, *m2, *r, *mend;
	m = mm;
	r = randrsl;
	a = aa;
	b = bb + (++cc);

	for (m = mm, mend = m2 = m + (RANDSIZ / 2); m < mend;)
	{
		rngstep(~(a ^ (a << 21)), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a >> 5), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a << 12), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a >> 33), a, b, mm, m, m2, r, x);
	}
	for (m2 = mm; m2<mend;)
	{
		rngstep(~(a ^ (a << 21)), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a >> 5), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a << 12), a, b, mm, m, m2, r, x);
		rngstep(a ^ (a >> 33), a, b, mm, m, m2, r, x);
	}
	bb = b; aa = a;
}

__int64* Isaac64::DoIsaac64(void)
{
	aa = bb = cc = 0;
	for (int i = 0; i < RANDSIZ; ++i)
		mm[i] = 0;

	RandInit(TRUE);

	RunIsaac64();

	return randrsl;
}


// Файл Isaac64.h

#pragma once

#define mix(a, b, c, d, e, f, g, h) \
{ \
	a -= e;  f ^= h >> 9;   h += a; \
	b -= f;  g ^= a << 9;   a += b; \
	c -= g;  h ^= b >> 23;  b += c; \
	d -= h;  a ^= c << 15;  c += d; \
	e -= a;  b ^= d >> 14;  d += e; \
	f -= b;  c ^= e << 20;  e += f; \
	g -= c;  d ^= f >> 17;  f += g; \
	h -= d;  e ^= g << 14;  g += h; \
}

#define ind(mm, x)  (*(__int64 *)((ub1 *)(mm) + ((x) & ((RANDSIZ - 1) << 3))))
#define rngstep(mix, a, b, mm, m, m2, r, x) \
{ \
	x = *m;  \
```

```
        a = (mix)+*(m2++); \
        *(m++) = y = ind(mm, x) + a + b; \
        *(r++) = b = ind(mm, y >> RANDSIZL) + x; \
}

static __int64 mm[RANDSIZ];
static __int64 aa, bb, cc;
class Isaac64
{
public:
        Isaac64(void);
        ~Isaac64(void);

        void RandInit(int flag);
        __int64 randrsl[RANDSIZ];
        __int64 randcnt;

        void RunIsaac64();
        __int64* DoIsaac64(void);
};


// Файл Snow.cpp

#include "stdafx.h"
#include "Snow.h"
#include "snowtab.h"
#include "General.h"

Snow::Snow()
{
}

Snow::~Snow()
{
}

/* Key and message independent initialization. This function will be
* called once when the program starts (e.g., to build expanded S-box
* tables).
*/
void Snow::Ecrypt_init(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize, const
BYTE* iv)
{
        Ecrypt_keysetup(ctx, key, keysize, ivsize);

        Ecrypt_ivsetup(ctx, iv);
}

/*
* Key setup. It is the user's responsibility to select the values of
* keysize and ivsize from the set of supported values specified
* above.
*/
void Snow::Ecrypt_keysetup(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize)
{
        for (int i = 0; i < keysize / 8; ++i)
                ctx->key[i] = key[i];

        ctx->keysize = keysize;
}

/* IV setup. After having called ECRYPT_keysetup(), the user is
```

```
*  allowed to call ECRYPT_ivsetup() different times in order to
*  encrypt/decrypt different messages with the same key but different
*  IV's./
void Snow::Ecrypt_ivsetup(Ecrypt_ctx* ctx, const BYTE* iv)
{
       snow_loadkey_fast(ctx, U8TO32_LITTLE(iv), U8TO32_LITTLE(iv + 4), U8TO32_LITTLE(iv +
8), U8TO32_LITTLE(iv + 12));
}


void Snow::Ecrypt_process_bytes(int action, Ecrypt_ctx* ctx, const BYTE* input, BYTE*
output, u32 msglen)
{
       u32 i;

       u32 keystream[16];

       u32 tmp;

       for (; msglen >= 64; msglen -= 64, input += 64, output += 64)
       {
              snow_keystream_fast(ctx, keystream); // keystream = 64 bytes

              for (i = 0; i < 16; ++i)
              {
                     tmp = ((u32*)input)[i];

                     ((u32*)output)[i] = tmp ^ U32TO32_LITTLE(keystream[i]);
              }

       }

       if (msglen > 0)
       {
              snow_keystream_fast(ctx, keystream);

              for (i = 0; i < msglen; i++)
                     output[i] = input[i] ^ ((BYTE*)keystream)[i];
       }
}


/*
* Function:  snow_loadkey_fast
*
*/
void Snow::snow_loadkey_fast(Ecrypt_ctx* ctx, u32 IV3, u32 IV2, u32 IV1, u32 IV0)
{
       int i;

       if (ctx->keysize == 128)
       {
              ctx->s15 = (((u32)*(ctx->key + 0)) << 24) | (((u32)*(ctx->key + 1)) << 16) |
                     (((u32)*(ctx->key + 2)) << 8) | (((u32)*(ctx->key + 3)));
              ctx->s14 = (((u32)*(ctx->key + 4)) << 24) | (((u32)*(ctx->key + 5)) << 16) |
                     (((u32)*(ctx->key + 6)) << 8) | (((u32)*(ctx->key + 7)));
              ctx->s13 = (((u32)*(ctx->key + 8)) << 24) | (((u32)*(ctx->key + 9)) << 16) |
                     (((u32)*(ctx->key + 10)) << 8) | (((u32)*(ctx->key + 11)));
              ctx->s12 = (((u32)*(ctx->key + 12)) << 24) | (((u32)*(ctx->key + 13)) << 16) |
                     (((u32)*(ctx->key + 14)) << 8) | (((u32)*(ctx->key + 15)));
              ctx->s11 = ~ctx->s15;  /* bitwise inverse */
              ctx->s10 = ~ctx->s14;
              ctx->s9 = ~ctx->s13;
```

```
                ctx->s8 = ~ctx->s12;
                ctx->s7 = ctx->s15;    /* just copy */
                ctx->s6 = ctx->s14;
                ctx->s5 = ctx->s13;
                ctx->s4 = ctx->s12;
                ctx->s3 = ~ctx->s15;   /* bitwise inverse */
                ctx->s2 = ~ctx->s14;
                ctx->s1 = ~ctx->s13;
                ctx->s0 = ~ctx->s12;
        }
        else
        {  /* assume keysize=256 */
                ctx->s15 = (((u32)*(ctx->key + 0)) << 24) | (((u32)*(ctx->key + 1)) << 16) |
                        (((u32)*(ctx->key + 2)) << 8) | (((u32)*(ctx->key + 3)));
                ctx->s14 = (((u32)*(ctx->key + 4)) << 24) | (((u32)*(ctx->key + 5)) << 16) |
                        (((u32)*(ctx->key + 6)) << 8) | (((u32)*(ctx->key + 7)));
                ctx->s13 = (((u32)*(ctx->key + 8)) << 24) | (((u32)*(ctx->key + 9)) << 16) |
                        (((u32)*(ctx->key + 10)) << 8) | (((u32)*(ctx->key + 11)));
                ctx->s12 = (((u32)*(ctx->key + 12)) << 24) | (((u32)*(ctx->key + 13)) << 16) |
                        (((u32)*(ctx->key + 14)) << 8) | (((u32)*(ctx->key + 15)));
                ctx->s11 = (((u32)*(ctx->key + 16)) << 24) | (((u32)*(ctx->key + 17)) << 16) |
                        (((u32)*(ctx->key + 18)) << 8) | (((u32)*(ctx->key + 19)));
                ctx->s10 = (((u32)*(ctx->key + 20)) << 24) | (((u32)*(ctx->key + 21)) << 16) |
                        (((u32)*(ctx->key + 22)) << 8) | (((u32)*(ctx->key + 23)));
                ctx->s9 = (((u32)*(ctx->key + 24)) << 24) | (((u32)*(ctx->key + 25)) << 16) |
                        (((u32)*(ctx->key + 26)) << 8) | (((u32)*(ctx->key + 27)));
                ctx->s8 = (((u32)*(ctx->key + 28)) << 24) | (((u32)*(ctx->key + 29)) << 16) |
                        (((u32)*(ctx->key + 30)) << 8) | (((u32)*(ctx->key + 31)));
                ctx->s7 = ~ctx->s15; /* bitwise inverse */
                ctx->s6 = ~ctx->s14;
                ctx->s5 = ~ctx->s13;
                ctx->s4 = ~ctx->s12;
                ctx->s3 = ~ctx->s11;
                ctx->s2 = ~ctx->s10;
                ctx->s1 = ~ctx->s9;
                ctx->s0 = ~ctx->s8;
        }

        /* XOR IV values */
        ctx->s15 ^= IV0;
        ctx->s12 ^= IV1;
        ctx->s10 ^= IV2;
        ctx->s9 ^= IV3;


        ctx->r1 = 0;
        ctx->r2 = 0;

        /* Do 32 initial clockings */
        for (i = 0; i < 2; i++)
        {
                u32 outfrom_fsm, fsmtmp;

                outfrom_fsm = (ctx->r1 + ctx->s15) ^ ctx->r2;
                ctx->s0 = a_mul(ctx->s0) ^ ctx->s2 ^ ainv_mul(ctx->s11) ^ outfrom_fsm;
                fsmtmp = ctx->r2 + ctx->s5;
                ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
                ctx->r1 = fsmtmp;

                outfrom_fsm = (ctx->r1 + ctx->s0) ^ ctx->r2;
                ctx->s1 = a_mul(ctx->s1) ^ ctx->s3 ^ainv_mul(ctx->s12) ^ outfrom_fsm;
                fsmtmp = ctx->r2 + ctx->s6;
```

```
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s1) ^ ctx->r2;
            ctx->s2 = a_mul(ctx->s2) ^ ctx->s4 ^ainv_mul(ctx->s13) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s7;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s2) ^ ctx->r2;
            ctx->s3 = a_mul(ctx->s3) ^ ctx->s5 ^ainv_mul(ctx->s14) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s8;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s3) ^ ctx->r2;
            ctx->s4 = a_mul(ctx->s4) ^ ctx->s6 ^ainv_mul(ctx->s15) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s9;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s4) ^ ctx->r2;
            ctx->s5 = a_mul(ctx->s5) ^ ctx->s7 ^ainv_mul(ctx->s0) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s10;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s5) ^ ctx->r2;
            ctx->s6 = a_mul(ctx->s6) ^ ctx->s8 ^ainv_mul(ctx->s1) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s11;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s6) ^ ctx->r2;
            ctx->s7 = a_mul(ctx->s7) ^ ctx->s9 ^ainv_mul(ctx->s2) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s12;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s7) ^ ctx->r2;
            ctx->s8 = a_mul(ctx->s8) ^ ctx->s10 ^ainv_mul(ctx->s3) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s13;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s8) ^ ctx->r2;
            ctx->s9 = a_mul(ctx->s9) ^ ctx->s11 ^ainv_mul(ctx->s4) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s14;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s9) ^ ctx->r2;
            ctx->s10 = a_mul(ctx->s10) ^ ctx->s12 ^ainv_mul(ctx->s5) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s15;
```

```
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;
            outfrom_fsm = (ctx->r1 + ctx->s10) ^ ctx->r2;
            ctx->s11 = a_mul(ctx->s11) ^ ctx->s13 ^ainv_mul(ctx->s6) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s0;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s11) ^ ctx->r2;
            ctx->s12 = a_mul(ctx->s12) ^ ctx->s14 ^ainv_mul(ctx->s7) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s1;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s12) ^ ctx->r2;
            ctx->s13 = a_mul(ctx->s13) ^ ctx->s15 ^ainv_mul(ctx->s8) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s2;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s13) ^ ctx->r2;
            ctx->s14 = a_mul(ctx->s14) ^ ctx->s0 ^ainv_mul(ctx->s9) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s3;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;

            outfrom_fsm = (ctx->r1 + ctx->s14) ^ ctx->r2;
            ctx->s15 = a_mul(ctx->s15) ^ ctx->s1 ^ainv_mul(ctx->s10) ^ outfrom_fsm;
            fsmtmp = ctx->r2 + ctx->s4;
            ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^
snow_T2[byte(2, ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
            ctx->r1 = fsmtmp;
      }
}

/*
* Function: snow_keystream_fast
*
* Synopsis :
*Clocks the cipher 16 times and returns 16 words of keystream symbols
*   in keystream_block.
*
* Returns : void
*
*/
void Snow::snow_keystream_fast(Ecrypt_ctx* ctx, u32* keystream_block)
{
      u32 fsmtmp;

      ctx->s0 = a_mul(ctx->s0) ^ ctx->s2 ^ainv_mul(ctx->s11);

      fsmtmp = ctx->r2 + ctx->s5;

      ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];

      ctx->r1 = fsmtmp;
```

```
        keystream_block[0] = (ctx->r1 + ctx->s0) ^ ctx->r2^ ctx->s1;

        ctx->s1 = a_mul(ctx->s1) ^ ctx->s3 ^ainv_mul(ctx->s12);
        fsmtmp = ctx->r2 + ctx->s6;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[1] = (ctx->r1 + ctx->s1) ^ ctx->r2^ ctx->s2;

        ctx->s2 = a_mul(ctx->s2) ^ ctx->s4 ^ainv_mul(ctx->s13);
        fsmtmp = ctx->r2 + ctx->s7;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[2] = (ctx->r1 + ctx->s2) ^ ctx->r2^ ctx->s3;

        ctx->s3 = a_mul(ctx->s3) ^ ctx->s5 ^ainv_mul(ctx->s14);
        fsmtmp = ctx->r2 + ctx->s8;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[3] = (ctx->r1 + ctx->s3) ^ ctx->r2^ ctx->s4;

        ctx->s4 = a_mul(ctx->s4) ^ ctx->s6 ^ainv_mul(ctx->s15);
        fsmtmp = ctx->r2 + ctx->s9;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[4] = (ctx->r1 + ctx->s4) ^ ctx->r2^ ctx->s5;

        ctx->s5 = a_mul(ctx->s5) ^ ctx->s7 ^ainv_mul(ctx->s0);
        fsmtmp = ctx->r2 + ctx->s10;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[5] = (ctx->r1 + ctx->s5) ^ ctx->r2^ ctx->s6;

        ctx->s6 = a_mul(ctx->s6) ^ ctx->s8 ^ainv_mul(ctx->s1);
        fsmtmp = ctx->r2 + ctx->s11;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[6] = (ctx->r1 + ctx->s6) ^ ctx->r2^ ctx->s7;

        ctx->s7 = a_mul(ctx->s7) ^ ctx->s9 ^ainv_mul(ctx->s2);
        fsmtmp = ctx->r2 + ctx->s12;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[7] = (ctx->r1 + ctx->s7) ^ ctx->r2^ ctx->s8;

        ctx->s8 = a_mul(ctx->s8) ^ ctx->s10 ^ainv_mul(ctx->s3);
        fsmtmp = ctx->r2 + ctx->s13;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
        keystream_block[8] = (ctx->r1 + ctx->s8) ^ ctx->r2^ ctx->s9;

        ctx->s9 = a_mul(ctx->s9) ^ ctx->s11 ^ainv_mul(ctx->s4);
        fsmtmp = ctx->r2 + ctx->s14;
        ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
        ctx->r1 = fsmtmp;
```

```
	keystream_block[9] = (ctx->r1 + ctx->s9) ^ ctx->r2^ ctx->s10;

	ctx->s10 = a_mul(ctx->s10) ^ ctx->s12 ^ainv_mul(ctx->s5);
	fsmtmp = ctx->r2 + ctx->s15;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[10] = (ctx->r1 + ctx->s10) ^ ctx->r2^ ctx->s11;

	ctx->s11 = a_mul(ctx->s11) ^ ctx->s13 ^ainv_mul(ctx->s6);
	fsmtmp = ctx->r2 + ctx->s0;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[11] = (ctx->r1 + ctx->s11) ^ ctx->r2^ ctx->s12;

	ctx->s12 = a_mul(ctx->s12) ^ ctx->s14 ^ainv_mul(ctx->s7);
	fsmtmp = ctx->r2 + ctx->s1;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[12] = (ctx->r1 + ctx->s12) ^ ctx->r2^ ctx->s13;

	ctx->s13 = a_mul(ctx->s13) ^ ctx->s15 ^ainv_mul(ctx->s8);
	fsmtmp = ctx->r2 + ctx->s2;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[13] = (ctx->r1 + ctx->s13) ^ ctx->r2^ ctx->s14;

	ctx->s14 = a_mul(ctx->s14) ^ ctx->s0 ^ainv_mul(ctx->s9);
	fsmtmp = ctx->r2 + ctx->s3;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[14] = (ctx->r1 + ctx->s14) ^ ctx->r2^ ctx->s15;

	ctx->s15 = a_mul(ctx->s15) ^ ctx->s1 ^ainv_mul(ctx->s10);
	fsmtmp = ctx->r2 + ctx->s4;
	ctx->r2 = snow_T0[byte(0, ctx->r1)] ^ snow_T1[byte(1, ctx->r1)] ^ snow_T2[byte(2,
ctx->r1)] ^ snow_T3[byte(3, ctx->r1)];
	ctx->r1 = fsmtmp;
	keystream_block[15] = (ctx->r1 + ctx->s15) ^ ctx->r2^ ctx->s0;
}


// Файл Snow.h

#pragma once
class Snow
{
public:
	Snow();

	~Snow();

	typedef struct
	{
		u32 keysize; // keysize = 128 or 256 bits, init vector = 128 bits

		BYTE key[32];

		u32 s15, s14, s13, s12, s11, s10, s9, s8, s7, s6, s5, s4, s3, s2, s1, s0;
```

```
        u32 r1, r2;
    } Ecrypt_ctx;

    void Ecrypt_init(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize, const
BYTE* iv);

    void Ecrypt_keysetup(Ecrypt_ctx* ctx, const BYTE* key, u32 keysize, u32 ivsize);

    void Ecrypt_ivsetup(Ecrypt_ctx* ctx, const BYTE* iv);

    void Ecrypt_process_bytes(
        int action,                               /* 0 = encrypt; 1 = decrypt; */
        Ecrypt_ctx* ctx,
        const BYTE* input,
        BYTE* output,
        u32 msglen);                      /* Message length in bytes. */

    void snow_loadkey_fast(Ecrypt_ctx* ctx, u32 IV3, u32 IV2, u32 IV1, u32 IV0);

    void snow_keystream_fast(Ecrypt_ctx* ctx, u32* keystream_block);
   };
```

Додаток Д

Програмний код реалізації симетричного блокового алгоритму шифрування AES (в режимі зворотного зв’язку за виходом)

Програмна реалізація алгоритму AES виконана на ПК з процесором Intel(R) Core(TM) i3-6100, 3.7GHz та обсягом оперативної пам’яті 4 ГБ на базі 64-розрядної ОС Windows 7 Service Pack 1. Мова програмування – С++. Середовище розробки – Microsoft Visual Studio 2013.

```
// Файл Realization_4.cpp

#include "stdafx.h"
#include "General.h"
#include "AES_alg.h"
#include <time.h>

int main(int argc, char* argv[])
{
	General run;
	AES_alg aes;

	int flen;

	clock_t start, finish;
	double duration;

	BYTE *in_file_buf;

	FILE *fp = fopen(argv[1], "rb");

	if(fp == NULL)
	{
		printf("ERROR!!! The test.to file was not opened!!! \n");
		return 0;
	}

	flen = run.get_file_len(fp);

	//buffer for the input file
	in_file_buf = new BYTE [flen];

	memset(in_file_buf, 0, flen);

	int read_bytes = fread(in_file_buf, 1, flen, fp);

	if(flen != read_bytes)
	{
		printf("\n ERROR!!! It wasn't read all bytes\n ");
		return 0;
	}
```

```
        start = clock();

        aes.DoAES(flen, in_file_buf, argv[1]);

        finish = clock();

        duration = (double)(finish - start) / CLOCKS_PER_SEC;

        printf("\n elapsed time = %f seconds", duration);

        fclose(fp);

        return 0;
}

// Файл General.cpp

#include "StdAfx.h"
#include "General.h"

General::General(void)
{
}

General::~General(void)
{
}

void General::hex_print(const void* pv, int len)
{
        const BYTE * p = (const BYTE*)pv;

        if (NULL == pv)
               printf("NULL");
        else
        {
               for (int i = 0; i < len; ++i)
               {
                      if (i % 16 == 0)
                             printf("\n");
                      printf("%02X ", *p++);
               }
        }
        printf("\n");
}

int General::get_file_len(FILE* fp)
{
        int len;

        fseek(fp, 0, SEEK_END);

        len = ftell(fp);

        fseek(fp, 0, SEEK_SET);

        return len;
}
void General::NewName(char* fn1, char* fn2, const char* _ext)
{
        memset(fn2, 0, sizeof(fn2));

        strcpy(fn2, fn1);
```

```
	char *p = strchr(fn2, '.');

	if(p)
		strcpy(p, _ext);
	else
		strcat(fn2, _ext);
}

void General::WritetoFile(char* fname, BYTE *buf, int len)
{
	FILE *fout;

	if((fout = fopen(fname, "wb")) == NULL)
	{
		perror("open failed");
		printf("\n ERROR open file %s to write!!!", fname);
	}
	fwrite(buf, 1, len, fout);

	fclose(fout);
}


// Файл General.h

#pragma once

class General
{
public:
	General(void);
	~General(void);

	static void hex_print(const void* pv, int len);

	int get_file_len(FILE* fp);

	void NewName(char* fn1, char* fn2, const char* _ext);

	void WritetoFile(char* fname, BYTE *buf, int len);
};


// Файл AES_alg.cpp

#include "StdAfx.h"
#include "AES_alg.h"
#include "General.h"

AES_alg::AES_alg(void)
{
}

AES_alg::~AES_alg(void)
{
}

int AES_alg::GenerateKey(BYTE* buf)
{
	if (!RAND_bytes(buf, KEYLEN / 8))
	{
		printf("ERROR!!! The encryption key was not generated!\n");
```

```
                return 0;
        }
        else
                return 1;
}

int AES_alg::GenerateIV(BYTE* buf)
{
        if (!RAND_bytes(buf, AES_BLOCK_SIZE))
        {
                printf("ERROR!!! The IV was not generated!\n");
                return 0;
        }
        else
                return 1;
}

void AES_alg::InitAES(int len)
{
        memset(key, 0, KEYLEN / 8);

        enc_out = new BYTE [len];

        dec_out = new BYTE [len];

        memset(enc_out, 0, sizeof(enc_out));

        memset(dec_out, 0, sizeof(dec_out));
}

void AES_alg::DoAES(int len, BYTE* plain_txt, char *fto)
{
        General run;

        char fts[15] = {0};
        char fdec[15] = {0};

        run.NewName(fto, fts, ".aes");
        run.NewName(fto, fdec, ".aes_dec");

        int num = 0;

        InitAES(len);

        GenerateKey(key);

        GenerateIV(iv_enc);

        memcpy(iv_dec, iv_enc, AES_BLOCK_SIZE);

        AES_KEY enc_key;

        AES_set_encrypt_key(key, KEYLEN, &enc_key);

        AES_ofb128_encrypt(plain_txt, enc_out, len, &enc_key, iv_enc, &num);

        num = 0;

        AES_ofb128_encrypt(enc_out, dec_out, len, &enc_key, iv_dec, &num);

        run.WritetoFile(fts, enc_out, len);

        run.WritetoFile(fdec, dec_out, len);
```

```
	delete [] enc_out;

	delete [] dec_out;
}

// Файл AES_alg.h

#pragma once
class AES_alg
{
public:
	AES_alg(void);
	~AES_alg(void);

private:

	BYTE *enc_out;

	BYTE *dec_out;

	BYTE iv_enc[AES_BLOCK_SIZE], iv_dec[AES_BLOCK_SIZE];

	static const int KEYLEN = 128;

	BYTE key[KEYLEN / 8];

public:
	int GenerateKey(BYTE* buf);

	int GenerateIV(BYTE* buf);

	void InitAES(int len);

	void DoAES(int len, BYTE* plain_txt, char *to);
};
```

“ЗАТВЕРДЖУЮ”

Виконавчий директор АТ «ІІТ»

_________ В.Д. Кравченко

«__» ____ 2016 р.

АКТ
впровадження результатів дисертаційних досліджень

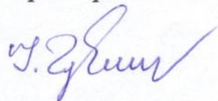
Гришакова Сергія Володимировича
у Приватному акціонерному товаристві «Інститут інформаційних технологій»

Комісія у складі голови комісії, головного конструктора, доктора технічних наук, професора Горбенка І.Д. і членів комісії, заступника головного конструктора з БС, кандидата технічних наук, професора Качко О.Г., і начальника відділу АЗЗІ, кандидата технічних наук Бобуха В.А. встановила, що у Приватному акціонерному товаристві «Інститут інформаційних технологій» впроваджені наступні результати, які одержані Гришаковим Сергійом Володимировичем у процесі виконання дисертаційних досліджень.

1. Криптоаналітичні атаки на шифросистеми Міхалевича-Імаї, відповідно, на основі шифрованих, відкритих та підібраних відкритих повідомлень і векторів ініціалізації.
2. Аналітичні межі для швидкості передачі інформації у шифросистемах Міхалевича-Імаї при заданих обмеженнях відносно їх стійкості та ймовірності правильного прийому повідомлень законним користувачем.
3. Метод побудови рандомізованих потокових шифросистем з нелінійним випадковим кодуванням.

Ефект від впровадження зазначених наукових результатів полягає в тому, що вони дозволяють:

– запропонувати більш ефективні в порівнянні з раніше відомими криптоаналітичні атаки на рандомізовані потокові шифросистеми з лінійним випадковим кодуванням (шифросистеми Міхалевича-Імаї);
– підсилити відомі необхідні умови стійкості зазначених шифросистем відносно відомих статистичних атак;
– з‘ясувати потенційні можливості (з погляду стійкості та практичності) шифросистем Міхалевича-Імаї і визначити загальні обмеження, яким задовольняють окремі показники їх ефективності при заданих значеннях інших показників;
– надати розробникові більше можливостей для побудови обчислювально стійких рандомізованих потокових шифросистем за рахунок розширення класу перетворень, що використовуються в конструкції рандомізатора;
– будувати на практиці обґрунтовано стійки та практичні рандомізовані потокові шифросистеми на основі нелінійних бієктивних перетворень чи безключевих геш-функцій.

Голова комісії, д.т.н., проф.  І.Д. Горбенко

Члени комісії:

к.т.н., проф.. О.Г. Качко

к.т.н. В.А. Бобух

Прим. № 1

"ЗАТВЕРДЖУЮ"
Заступник директора департаменту –
начальник управління
Служби зовнішньої розвідки України

Романович К.О.

_____________ 2016 року

АКТ

впровадження результатів досліджень дисертаційної роботи
Гришакова Сергія Володимировича в науково-дослідній роботі "Дослідження та застосування методів криптографічного аналізу важкозворотних перетворень у сучасних криптографічних системах захисту інформації з урахуванням додаткових даних" (шифр "Кета")

Комісія у складі голови комісії Сединкіна С.К. та членів комісії: Черевко Ю.П., Гудзенко С.В. з'ясувала, що в Службі зовнішньої розвідки України в результаті виконання науково-дослідної роботи "Дослідження та застосування методів криптографічного аналізу важкозворотних перетворень у сучасних криптографічних системах захисту інформації з урахуванням додаткових даних" (шифр "Кета") вперше впроваджено отримані Гришаковим Сергієм Володимировичем такі наукові результати:

1. Криптоаналітичні атаки на шифросистеми Міхалевича-Імаї, відповідно, на основі шифрованих, відкритих та підібраних відкритих повідомлень і векторів ініціалізації.
2. Метод побудови рандомізаторів для шифросистем Міхалевича-Імаї при заданих обмеженнях щодо обчислювальної стійкості та ефективності реалізації.
3. Метод побудови рандомізованих потокових шифросистем з нелінійним випадковим кодуванням.

Ефект від впровадження зазначених наукових результатів полягає в тому, що вони дозволяють:

– оцінювати ефективність програмних реалізацій рандомізованих потокових шифросистем з нелінійним випадковим кодуванням і шифросистем Міхалевича-Імаї при різних об'ємах вхідних даних;

– будувати на практиці рандомізовані потокові шифросистеми на основі нелінійних відображень або безключових геш-функцій, що є більш стійкими (у 2^{242} і більше разів) і більш швидкісними (у 125 і більше разів) за шифросистеми Міхалевича-Імаї при однаковій довжині вхідного повідомлення;

– програмно реалізувати шифросистеми Міхалевича-Імаї на базі кодів БЧХ, що є обчислювально стійкими відносно атак на основі підібраних відкритих повідомлень (за умови належної якості генератора випадкових послідовностей, що використовується).

Голова комісії: _____________ Сединкін С.К.

Члени комісії:
к.т.н. _____________ Черевко Ю.П.

к.ф.-м.н. _____________ Гудзенко С.В.

"14" 09 2016 року