**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ**

Інститут аерокосмічних систем управління

Факультет електроніки

Кафедра радіоелектроніки

На правах рукопису

Ткаченко Павло Леонідович

УДК 519.725

Синтез і аналіз ефективності модифікованого алгоритму завадостійкого коду Ріда-Соломона

05.13.21 – Системи захисту інформації

Кваліфікаційна робота на здобуття наукового ступеня

кандидата технічних наук НАУ

Науковий керівник

Білецький Анатолій Якович

доктор технічних наук, професор

кафедри радіоелектроніки

Київ 2010

# ЗМІСТ

ВСТУП 4

ПЕРЕЛІК СКОРОЧЕНЬ 9

1. **Аналіз стану завадостійкого кодування криптографічної інформації 10**
   1. Аналіз існуючих алгоритмів завадостійких кодів 10
   2. Про RAID системи 12
   3. Про алгоритм коду Ріда-Соломона 13
   4. Завадостійке кодування Ріда-Соломона у RAID - подібних системах 15
   5. Висновки 17
2. **Математичні основи завадостійкого кодування та захисту інформації 18**
   1. Матричні обчислення в RAID-подібних системах 18
      1. Розрахунок матриці Вандермонда 18
      2. Створення томів для відновлення 20
      3. Використання томів для відновлення 20
      4. Пошук оберненої матриці методом Жорданових винятків 22
      5. Важливе доповнення до розрахунку матриці кодування 28
   2. Оптимізація матричних обчислень 32
   3. Криптографічна складова системи – RSB алгоритм 33
      1. Синтез алгоритму симетричного блокового RSB шифрування 35
      2. Загальна характеристика RSB алгоритму 36
      3. RSB криптографічні примітиви 42
         * 1. Стохастична кругова прокрутка блоку 42
           2. Ковзне кодування 43
           3. Нелінійна заміна байтів 48
           4. Стохастична перестановка елементів блоку 50
   4. Висновки 51
3. **Синтез та аналіз ефективності завадостійкого кодування та криптографічного захисту 52**
   1. Арифметика полів Галуа 52
      1. Додавання / віднімання 54
      2. Множення 54
      3. Ділення 58
   2. Практична реалізація арифметики полів Галуа на С # 59
   3. Опис архітектури програми «RS Кодер-декодер» 59
      1. Діаграма класів 60
      2. Інтерфейс програми «RS Кодер-декодер» 62
   4. Логіка роботи програми «RS Кодер-декодер» 62
      1. Процес кодування 62
      2. Процес декодування 63
      3. Процес лікування набору томів 63
      4. Процес тестування набору томів 64
   5. Програмний комплекс з дослідження статистичних властивостей симетричного блочного RSB криптографічного алгоритму 64
   6. Опис архітектури програми «RSB32 2.0» 69
      1. Інтерфейс програми «RSB32 2.0» 69
   7. Логіка роботи програми «RSB32 2.0» 70
      1. Процес кодування 70
      2. Процес декодування 70
   8. Висновки 71

**ВИСНОВКИ 72**

**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 73**

**ДОДАТКИ 77**

# ВСТУП

*Актуальність*. Зі зростанням використання електроніки і комп'ютерів, зростає потреба у швидкій і надійної передачі інформації по каналах зв'язку. У будь-якому каналі зв'язку присутні шуми - сигнали, які можуть спотворювати інформацію, що передається. З цими спотвореннями можна боротися, перетворюючи інформацію, що передається за допомогою коду, який буде виявляти, і виправляти помилки. В основі роботи всіх кодів лежить модифікування вихідних даних шляхом додавання деякої надлишкової інформації, яка дозволяє виявляти і виправляти помилки, які могли виникнути при передачі кодованої інформації по зашумленими каналу зв'язку.

У 1969 році, за допомогою штучних супутників Mariner 6 і Mariner 7, було отримано близько двохсот фотографій Марса. Кожна фотографія складалася з 658240 восьми бітних пікселів. Таким чином, для кожної фотографії було потрібно близько 5-ти мільйонів біт інформації. Ці біти були кодовані кодом, що виправляють помилки, і передані зі швидкістю 16200 біт в секунду на Землю, де вони були успішно декодувати.

Історія кодування, що контролює помилки, почалася в 1948р. публікацією знаменитої статті Клода Шенона. Шенон показав, що з кожним каналом пов'язано вимірюється в бітах в секунду і зване пропускною спроможністю каналу число C, що має таке значення. Якщо необхідна від системи зв'язку швидкість передачі інформації R (яка вимірюється в бітах секунду) менше C, то, використовуючи коди, які контролюють помилки, для даного каналу можна побудувати таку систему зв'язку, що ймовірність помилки на виході буде як завгодно мала.

У п'ятдесяті роки багато зусиль було витрачено на спроби побудови в явному вигляді класів кодів, що дозволяють отримати обіцяну як завгодно малу ймовірність помилки, але результати були мізерними. У наступному десятилітті цій захоплюючій завдання приділялося менше уваги, замість цього дослідники кодів зробили атаку за кількома основними напрямками.

Один з напрямків носило чисто алгебраїчний характер і переважно розглядало блокові коди. Перші блокові коди були введені в 1950 р., коли Хеммінга описав клас блокових кодів, що виправляють поодинокі помилки. Незважаючи на посилені дослідження, до кінця п'ятдесятих років не було побудовано кращого класу кодів. Основний зсув стався, коли Боуз і Рой-Чоудхурі в 1960 р. і Хоквінгем в 1959 р. знайшли великий клас кодів, що виправляють кратні помилки (коди БЧХ).

Але клас кодів що виправляють кратні помилки не є досить потужним для виправлення пакетних помилок, що часто виникають у каналах зв’язку, тому є виправданим рішення розглядати код Ріда-Соломона для проектування програмного комплексу, а методом проектування обрати розробку програми що працюватиме по алгоритму Ріда-Соломона.

*Зв'язок роботи з науковими програмами, планами, темами.* Кваліфікаційна робота виконана в рамках держбюджетних НДР: 388-ДБ07 «Розробка та впровадження новітніх технологій стиску та захисту сигналів та зображень в радіоелектронних системах і комплексах» (№0107U002666), 669-ДБ10 «Розроблення та впровадження програмних засобів захисту інформації від несанкціонованого доступу в електронних системах документообігу у вищих навальних закладах України» (№0110U000222).

*Мета і задачі дослуження.* Метою роботи є розробка програмної частини для цифрового апаратно-програмного комплексу завадостійкої системи прийомо-передачі криптографічних даних що дасть змогу, при впровадженні даної системи, отримати досить стійку систему захищену як від завад та помилок у обмінюваній інформації але й забезпечити конфіденційність цієї інформації.

*Завдання роботи.* Завданням даної роботиє розробка цифрового програмно-апаратного комплексу завадостійкої системи прийомо-передачі криптографічних даних а саме алгоритм завадостійкого кодування-декодування.

*Об’єкт дослідження.*Об'єктом роботи є процес побудови завадостійкого кодування по алгоритму Ріда-Соломона з елементами RAID системи та криптографічною складовою.

*Предмет дослідження.*Предметомдослідження є різні методи завадостійкого кодування інформації з єлементами криптографічного захисту. Під час дослідження методів завадостійкого кодування використовувалися методи кодування за допомогою алгоритму Ріда-Соломона, RAID метод кодування та метод CRC кодів. Було досліджено методики блокового, циклічного кодування, кодів Хеммінга та БЧХ коди.

*Методи дослідження.* Для вирішення даного завдання було використано наступні методи:

1. Метод матричних обчислень в RAID-подібних системах.
2. Метод розрахунку матриці Вандермонда.
3. Пошуку оберненої матриці методом Жорданових винятків.
4. Метод оптимізації матричних обчислень.
5. У елементах криптографічного захисту використовувалися методи стохастичної кругової прокрутки блоку, ковзного кодування, нелінійної заміни байтів та стохастичної перестановки.

*Наукова новизна.* Наукова новизна полягає в способі захисту інформації від помилок та завад, полягає у поєднанні досить потужних методик (RAID системи, алгоритм Ріда-Соломона) боротьби з цими проблемами, що є до речі досить новим підходом до вирішення цих проблем.

*Практичне значення отриманих результатів.* Практичне значення отриманих результатів можна виразити через визначення: комп'ютерний захист - це постійна боротьба з «дурістю» користувачів й інтелектом хакерів. Навіть хакери найчастіше використають саме некомпетентність і недбалість обслуговуючого персоналу й саме останнім можна вважати головною загрозою безпеки. Одна із проблем подібного роду - так звані слабкі паролі. Користувачі для кращого запам'ятовування вибирають паролі, що легко вгадати. Причому проконтролювати складність пароля неможливо. Інша проблема - зневага вимогами безпеки. Наприклад, небезпечно використати неперевірене програмне забезпечення. Звичайно користувач сам "запрошує" у систему віруси й "троянських коней". Крім того, багато неприємностей може принести неправильно набрана команда.

Кращий захист від нападу - не допускати його. Навчання користувачів правилам безпеки мережі може запобігти нападам. Захист інформації містить у собі крім технічних мір ще й навчання або правильний підбор обслуговуючого персоналу.

Захист інформації не обмежується технічними методами. Проблема є значно ширшою. Основний недолік захисту - люди, і тому надійність системи безпеки залежить в основному від відношення до неї. Крім цього, захист повинен постійно вдосконалюватися разом з розвитком комп'ютерної мережі.

У цей час узагальнена теорія безпеки інформації поки не створена. Застосовувані на практиці підходи й засоби нерідко страждають істотними недоліками й не мають оголошену надійність. Тому необхідно володіти достатньою підготовкою й кваліфіковано орієнтуватися у всьому спектрі питань забезпечення інформаційної безпеки, розуміючи їх комплексний і взаємообумовлений характер

А захист від помилок – це боротьба з природними впливами, що загрожують інформації у будь-яких методах її зберігання та обміну оскільки «ідеального» пристрою не існує, а тому потрібно розробляти системи та алгоритми які будуть виправляти цей недолік апаратних комплексів.

*Особистий внесок здобувача.*В роботі [30] удосконалено алгоритм установки ключів, що дозволяє використовувати даний алгоритм у незахищених комп’ютерних мережах оскільки забезпечує захист від завад при передачі ключа по каналах зв’язку. В роботі [32] розроблено алгоритм синтезу утворюючих та перевірочних матриць для (*n, k*) – кодів Хемінга, а також вирішена задача локалізації та усунення одиничної помилки у любому розряді коду, причому розроблені алгоритми допускають досить просту як програмну так і апаратну реалізацію.

*Апробація результатів роботи.* Апробація отриманих результатів була проведена під час багатьох виставок та конференцій, а саме конференції «Політ» за 2007, 2008 роки та в цих же роках у виставках студентських талантів інституту ІЕСУ НАУ. Також основні положення роботи доповідались на науково-практичній конференції «Захист в інформаційно-комунікаційних системах» (НАУ, травень 2009) та студентській науково-технічній конференції ІНТ (НАУ, листопад 2009).

*Публікації*. За темою кваліфікаційної роботи опубліковано 3 друковані праці (дві статті та одні тези доповідей на конференціях).

*Структура кваліфікаційної роботи.* Робота складається із вступу, списку скорочень, трьох розділів, висновків. Обсяг роботи складає 104 сторінок, робота містить 22 рисунки, 19 таблиць, 54 використаних джерел, 6 додатків.

# ПЕРЕЛІК СКОРОЧЕНЬ

AES англ. Advanced Encryption Standard - розширений стандарт шифрування

CBC англ Ciphertext Block Chaining - зчеплення блоків шифротексту

CRC англ. Cyclic Redundancy Check - циклічна надлишкова перевірка

DES англ. Data Encryption Standard - стандарт шифрування даних

GF англ. Galois Field – поля Галуа

OSI англ. Open Systems Interconnection Reference Model - модель взаємодії відкритих систем

RAID англ. Redundant array of independent/inexpensive disks - надлишковий масив незалежних/недорогих дисків

RC англ. Rоund Code – Раундовий код

RSB англ. Rоund Step Blok

RK англ. Rоund Kay – Раундовий ключ

XOR англ. Exclusive OR – Виключаюче АБО

БЧХ Боуз Рой-Чоудхурі

# РОЗДІЛ 1

## Аналіз стану завадостійкого кодування криптографічної інформації

1.1 Аналіз існуючих алгоритмів завадостійких кодів

Виявлення помилок в техніці зв'язку - дія, спрямована на контроль цілісності даних при записі/відтворенні інформації або при її передачі по лініях зв'язку. Виправлення помилок (корекція помилок) - процедура відновлення інформації після читання її з пристрою зберігання або каналу зв'язку.

Для виявлення помилок використовують коди виявлення помилок, для виправлення - коригуючі коди (коди, що виправляють помилки, коди з корекцією помилок, завадостійкі коди).

У процесі зберігання даних і передачі інформації по мережах зв'язку неминуче виникають помилки. Контроль цілісності даних і виправлення помилок - важливі завдання на багатьох рівнях роботи з інформацією (зокрема, фізичному, канальному, транспортному рівнях моделі OSI).

У системах зв'язку можливі кілька стратегій боротьби з помилками:

* виявлення помилок у блоках даних і автоматичний запит повторної передачі пошкоджених блоків - цей підхід застосовується в основному на канальному і транспортному рівнях;
* виявлення помилок у блоках даних і відкидання пошкоджених блоків - такий підхід іноді застосовується в системах потокового мультимедіа, де важлива затримка передачі і немає часу на повторну передачу;
* виправлення помилок застосовується на фізичному рівні (а також програмному рівні).

Коригуючі коди - коди, що служать для виявлення або виправлення помилок,що виникають при передачі інформації під впливом перешкод, а також при її зберіганні.

Для цього при записі (передачі) у корисні дані додають спеціальним чином структуровану надлишкову інформацію (контрольне число), а при читанні (прийомі) її використовують для того, щоб виявити або виправити помилки. Природно, що кількість помилок, яку можна виправити, обмежений і залежить від конкретного застосовуваного коду.

З кодами, що виправляють помилки, тісно пов'язані коди виявлення помилок. На відміну від першого, останні можуть тільки встановити факт наявності помилки в переданих даних, але не виправити її.

Насправді, використовувані коди виявлення помилок належать до тих же класів кодів, що і коди, що виправляють помилки. Фактично, будь-який код, що виправляє помилки, може бути також використаний для виявлення помилок (при цьому він буде здатний виявити більше число помилок, ніж був здатен виправити).

За способом роботи з даними коди, що виправляють помилки поділяються на блокові, що ділять інформацію на фрагменти постійної довжини і оброблюють кожен з них окремо, та сверточні, що працюють з даними як з безперервним потоком. Перелічимо основні типи кодів:

* Блокові коди
* Лінійні коди загального вигляду (Коди Хеммінга)
* Лінійні циклічні коди
* Коди CRC
* Коди БЧХ
* Коди корекції помилок Ріда - Соломона
* Сверточні коди

Практично всі використовувані коди є лінійними. Це пов'язано з тим, що нелінійні коди значно складніше досліджувати, і для них важко забезпечити прийнятну легкість кодування та декодування.

Коди CRC є систематичними кодами, призначеними не для виправлення помилок, а для їх виявлення. Вони використовують спосіб систематичного кодування: «контрольна сума» обчислюється шляхом ділення *xn - ku(x)* на *g(x)*. З огляду на те, що виправлення помилок не потрібно, перевірка правильності передачі може проводитися точно так само.

Коди Боуз - Чоудхурі - Хоквінгема (БЧХ) є підкласом циклічних кодів. Їх відміна властивість - можливість побудови коду БЧХ з мінімальною відстанню не менше заданого. Це важливо, тому що, взагалі кажучи, визначення мінімальної відстані коду є дуже складне завдання.

Коди Ріда - Соломона - Недвійкова циклічні коди, що дозволяють виправляти помилки в блоках даних. Елементами кодового вектора є не біти, а групи бітів (блоки). Дуже поширені коди Ріда - Соломона, що працюють з байтами (октетами). Код Ріда - Соломона є окремим випадком БЧХ-коду.

Сверточні коди ефективно працюють в каналі з білим шумом, але погано справляються з пакетами помилок. Більш того, якщо декодер помиляється, на його виході завжди виникає пакет помилок.

Хоча блокові коди, як правило, добре справляються з рідкісними, але великими пачками помилок, їх ефективність при частих, але невеликих помилки, менш висока.

Коди, що виправляють помилки, застосовуються:

* в системах цифрового зв'язку, у тому числі: супутникового, радіорелейного, стільникового, передачі даних по телефонних каналах.
* в системах зберігання інформації, у тому числі магнітних і оптичних.

Коди, що виявляють помилки, застосовуються в мережевих протоколах різних рівнів.

### 1.2 Про RAID системи

RAID дисковий масив - це набір дискових пристроїв, що працюють разом, щоб підвищити швидкість і надійність системи вводу / виводу. Цим набором пристроїв управляє спеціальний RAID-контролер (контролер масиву), який забезпечує функції розміщення даних по масиву, а для решти всієї системи дозволяє представляти весь масив як один логічний пристрій вводу / виводу.

Масиви також можуть забезпечувати надмірне зберігання даних, з тим, щоб дані не були втрачені у разі виходу з ладу одного з дисків. Залежно від рівня RAID, проводиться або дзеркалювання або розподіл даних по дисках.

Але реалізації RAID можна застосовувати не тільки апаратні засоби, але і повністю програмні компоненти (драйвери). Наприклад, у системах на ядрі Linux існують спеціальні модулі ядра, а керувати RAID-пристроями в GNU/Linux можна за допомогою утиліти mdadm. Програмний RAID має свої переваги і недоліки. З одного боку, він нічого не вартий (на відміну від апаратних RAID-контролерів, ціна яких починається від $250). З іншого боку, програмний RAID використовує ресурси центрального процесора, і в моменти пікового навантаження на дискову систему процесор може значну частину потужності витрачати на обслуговування RAID-пристроїв.

Тому для підвищення завадостійких властивостей проектуємої системи було вирішено покласти в основу алгоритму кодування саме RAID принцип кодування.

### 1.3 Про алгоритм коду Ріда-Соломона

Код Ріда - Соломона був винайдений в 1960 році співробітниками лабораторії Лінкольна Массачуссетського технологічного інституту Ірвіном Рідом (англ.) і Густавом Соломоном (англ.). Ідея використання цього коду була представлена в статті «Polynomial Codes over Certain Finite Fields». Перше застосування код Ріда - Соломона отримав в 1982 році в серійному випуску компакт-дисків. Ефективний алгоритм декодування був запропонований у 1969 році Елвіном Берлекемпом (англ.) і Джеймс Мессі (алгоритм Берлекемпа - Мессі).

Коди Ріда - Соломона - Недвійкові циклічні коди, що дозволяють виправляти помилки в блоках даних. Елементами кодового вектора є не біти, а групи бітів (блоки). Дуже поширені коди Ріда - Соломона, що працюють з байтами (октетами). Код Ріда - Соломона є окремим випадком БЧХ-коду.

Код Ріда - Соломона є одним з найбільш потужних кодів, що виправляють багаторазові пакети помилок. Застосовується в каналах, де пакети помилок можуть утворюватися настільки часто, що їх вже не можна виправляти за допомогою кодів, що виправляють поодинокі помилки.

В даний час широко використовується в системах відновлення даних з компакт-дисків, при створенні архівів з інформацією для відновлення у випадку ушкоджень, в завадостійкому коді

Кодування за допомогою коду Ріда - Соломона може бути реалізовано двома способами: систематичним і несистематичні

При несистематичної кодуванні інформаційне слово множиться на якийсь неприводимий поліном в полі Галуа. Отримане закодоване слово повністю відрізняється від вихідного і для добування інформаційного слова потрібно виконати операцію декодування і вже потім можна перевірити дані на вміст помилок. Таке кодування вимагає великих витрат ресурсів лише на отримання інформаційних даних, при цьому вони можуть бути без помилок.

При систематичному кодуванні до інформаційного блоку з *k* символів приписуються *2t* перевірочних символів, при обчисленні кожного перевірочного символу використовуються всі *k* символів вихідного блоку. У цьому випадку немає витрат ресурсів при отриманні вихідного блоку, якщо інформаційне слово не містить помилок, але кодер/декодер повинен виконати *k (n - k)* операцій додавання і множення для створення перевірочних символів. Крім того, тому що всі операції проводяться в полі Галуа, то самі операції кодування/декодування вимагають багато ресурсів і часу. Швидкий алгоритм декодування, заснований на швидкому перетворенні Фур'є, виконується за час порядку *lnn2*.

Виходячи з цього саме алгоритм Ріда - Соломона покладено у основу роботи, оскільки цей алгоритм якнайкраще підходить для виправлення пакетних помилок у каналах зв’язку.

### 1.4 Завадостійке кодування Ріда - Соломона у RAID - подібних системах

Під RAID-подібними системами будемо розуміти такі програмно-апаратні рішення, які передбачають збиткове подання даних в наступному вигляді:

*N+M*; (1.1)

де *N* - кількість основних пристроїв, призначених для зберігання даних користувача; *M* - кількість пристроїв, що зберігають збиткові дані.

При вищевказаній схемі поданні даних має виконуватися наступна нерівність:

*N+M<2* розрядність кодера *-1*; (1.2)

Нерівність (1.2) вказує на те, що загальна кількість пристроїв з даними і збитковістю в RAID-подібних системах не може перевищувати розмір поля. У разі прикладного застосування, замість терміна пристрій може застосовуватися термін Том[[1]](#footnote-1). Особливості RAID - подібних систем припускають, що всі томи мають однаковий розмір. Схема кодування виглядає таким чином: якщо всі томи представити у вигляді стовпців таблиці, то кодування буде відбуватися рядок за рядком. Іншими словами, рядок даних такої таблиці міститиме спочатку байти основних томів, а, відтак, збиткові дані, що обчислюються на підставі основних томів.

Кожен том для відновлення зможе замінити собою один з загублених основних томів. Пояснимо дану схему кодування на наступному прикладі: нехай є два призначених для користувача файли: «Основний том № 1» та «Основний том № 2», що містять такі дані: (2,4,8,16) та (4,3,2,1 ). Уявімо дану конфігурацію у вигляді таблиці 1.1.

Таблиця 1.1.

Схема подання даних в RAID - подібному форматі

|  |  |  |
| --- | --- | --- |
| Основний том №1 | Основний том №2 | Том для відновлення |
| 2 | 4 | ? |
| 4 | 3 | ? |
| 8 | 2 | ? |
| 16 | 1 | ? |

Як видно з таблиці 1.1, дані «Том для відновлення» не визначені. У випадку двох основних томів, навіть у світлі використання кодування Ріда-Соломона, як і в «дорослих» системах, для отримання даних збиткового тому повинна застосовуватися операція XOR[[2]](#footnote-2) (побітове додавання по модулі два). Обчислимо дані тому для відновлення і помістимо результати в таблицю 1.2.

Таблиця 1.2.

Результат кодування даних в RAID - подібному форматі

|  |  |  |
| --- | --- | --- |
| Основний том №1 | Основний том №2 | Том для відновлення |
| 2 | 4 | 6 |
| 4 | 3 | **7** |
| 8 | 2 | 10 |
| 16 | 1 | 17 |

Припустимо тепер, що один з основних томів втрачено (пошкоджено). Використовуючи дані, що залишилися від основного тому та тому для відновлення (застосовуючи операцію XOR) Ви зможете отримати оригінальні дані пошкодженого тому.

### 1.5 Висновки

Підсумком до винонаного аналізу стану завадостійкого кодування криптографічної інформації можна вважати отримання інформації про досить недостатню можливість виправлення помилок у великих блоках пошкодженої інформації такими класами кодів як блокові коди, лінійні, циклічні коди та коди CRC. Але це не говорить про те що для вирішення серйозних задач захисту інформації від пошкодження завадами та помилками не можна використовувати єлементи вищезгаданих алгоритмів. Аналіз цих питань призвів до виникнення ідеї поєднання досить потужного коду Ріда-Соломона з єлементами менш дієвих алгоритмів, що дозволяє підвищити здатність виправлення більшої кількості помилок без значного падіння швидкодії роботи алгоритму.

# РОЗДІЛ 2

## Математичні основи завадостійкого кодування та захисту інформації

### 2.1 Матричні обчислення в RAID-подібних системах

Якщо кількість основних томів перевищує 2, то потрібно перейти від конкретного випадку теорії до загального. У даному розділі мова піде про матричну арифметику в контексті завадостійкого кодування.

#### 2.1.1 Розрахунок матриці Вандермонда

Перш ніж приступати до опису процесу обчислення томів для відновлення на основі даних основних томів, розглянемо матрицю Вандермонда. Кожен елемент такої матриці може бути обчислений за такою формулою:

*F(i, j) = i j-1*; (2.1)

де i - номер рядка, в якій знаходиться обчислюваний елемент; j - номер стовпця, в якому знаходиться обчислюваний елемент.

Головною властивістю матриці, що заповнюється таким чином, є те, що вона ніколи не є виродження, тобто завжди має зворотну матрицю. Якщо матрицю Вандермонда помножити на вектор-стовпець зрізу[[3]](#footnote-3) даних основних томів, то результатом операції стане вектор-стовпець зрізу даних томів для відновлення.

Далі усі матричні операції RAID-систем можуть бути застосовні як для звичайної арифметики, що використовується у представлених прикладах в середовищі MathCAD, так і для арифметики поля Галуа (табл. 2.1).

Таблиця 2.1.

Функція, що реалізує розрахунок матриці Вандермонда

Зазначаємо, що індекси масивів починаються з "1":

ORIGIN := 1

Задаємо функцію для обчислення елементів матриці Вандермонда:

F(i, j) := i j- 1

Задаємо кількість основних томів:

n := 4

Задаємо кількість томів для відновлення:

m := 3

Функция расчета матрицы Вандермонда:

GetF(N, M) T :=

Розраховуємо матрицю і виводимо її на екран …

F := GetF(n,m)

F =

При необхідності, кількість рядків в матриці Вандермонда може перевищувати кількість стовпців, що буде означати кодування з збитковістю більше 100%.

#### 2.1.2 Створення томів для відновлення

Приклад арифметики створення томів в наведено у табл.. 2.2. Вектор "R" - три томи для відновлення. Кожен тому для відновлення може замінити собою, в процесі відновлення даних, один загублений основний том.

Таблиця 2.2.

Розрахунок зрізу даних томів для відновлення

Задаємо вектор-стовпець вхідних даних:

D :=

Отримуємо вектор-стовпець даних для відновлення "R", ( F·D = R ):

R := F·D

R =

#### 2.1.3 Використання томів для відновлення

Аналогічним чином наведемо арифметику використання томів для відновлення (табл. 2.3)

Таблиця 2.3.

Формування дисперсної матриці "A"

I := D =

F = R =

A := stack(I, F) B := stack(D, R)

A = B =

Кожен рядок одиничної матриці I відповідає за відповідний основний тому, причому номер рядка або стовпця, в якій знаходиться одиниця, що відповідає номеру томи. Кожен рядок матриці F відповідає за відповідний тому для відновлення. Разом вони представляють «дисперсну» матрицю A.

Кожен рядок вектора-стовпця R - дані відповідного томи для відновлення. У коді MathCAD представлений один зріз, але у випадку реальних файлів мається на увазі безліч ітерацій обробки, і, як наслідок, безліч зрізів, з яких будуть формуватися генеруються файли для відновлення.

Кожній рядку дисперсної матриці A відповідає рядок матриці B, що складається з даних основних томів D і даних томів для відновлення R.

Після виконання завадостійкого кодування, до сукупності основних томів додаються томи для відновлення. Разом вони представляють систему з надмірністю (сукупність матриць "A" і "B"). Кожен втрачений основний тому чи тому для відновлення означає видалення відповідного рядка з обох матриць "A" і "B". Припустимо, що первісна сукупність матриць "A" і "B" була змінена в результаті втрати даних, а частина даних, можливо, була відкинута нами за непотрібністю (адже потрібно одержати квадратну матрицю, отже, потрібно відібрати лише m-строк з (n + m ), і створити нову сукупність "A\_", "B\_"). Для того, щоб запустити процес відновлення даних, необхідно і достатньо, щоб дисперсна матриця "A\_" була квадратної (конденсована дисперсна матриця), тому що це є умовою для знаходження її зворотній форми. Якщо в сформовану з "залишків" вихідної сукупності матрицю "A\_" додається рядок, що відповідає основному тому, то вона повинна розміщуватися на "своє" місце, так, щоб єдина одиниця в її рядку виявилася на головній діагоналі "A\_". При додаванні рядка в матрицю "A\_" з "A" відбувається автоматичне додавання в "B\_" пов'язаного з нею елемента з "B". Потім, коли матриці "A\_" і "B\_" набрані із "залишків", слід виконати наступні обчислення:

D=A\_-1 ·B\_; (2.2)

Перевіримо на практиці зазначені рекомендації (табл. 2.4.):

Таблиця 2.4.

Обчислення зрізу даних основних томів на підставі даних зворотного конденсованої дисперсної матриці остаточного набору томів

A\_ := B\_ :=

D := A\_-1 · B\_ D\_ =

Зріз даних основних томів обчислений коректно.

#### 2.1.4 Пошук оберненої матриці методом Жорданових винятків

Система лінійних алгебраїчних рівнянь – це система рівнянь виду:

(2.3)

Тут *x1 ,..., xn* – невідомі, а *aij ,bi ,* *i* = 1,...,*m* , *j* = 1,...,*n* – задані коефіцієнти, які прийнято записувати у вигляді матриць:

Дуже часто в обговоренні таких систем рівнянь використовують матрицю

яку називають розширеною матрицею системи; матриця *А* називається матрицею системи, а матриця *B* – матрицею вільних членів.

Рішення системи (2.3) – це такий набір чисел *h1, h2, …, hn*, що при підстановці

*xi= hi*, *i = 1,2 ,..., n*, всі рівності в (2.3) стають тотожності.

Система (2.3) називається сумісною, якщо в неї є хоч одне рішення, і несумісною, якщо жодного рішення у неї немає.

Система (2.3) називається визначеною, якщо в неї є рішення і до того ж тільки одне.

Нехай *А* і *В* – дві матриці такого вигляду:

звернемо увагу на те, що в першій з них стовпців стільки ж, скільки в другій з них – Рядків. У цьому випадку можна побудувати нову матрицю *С*,

,

звану добутком *А* на *В* (пишуть *С = А ∙ В*), за правилом

,

де *i = 1,..., m* та *j = 1,..., r*. Серед матриць прийнято виділяти і позначати буквою *E* наступну матрицю, яка називається одиничною, -

Одинична матриця завжди квадратна, за її діагоналі стоїть 1, а всі інші елементи – нулі, тобто якщо *Е = (еij), i, j = 1 ,..., n*, то

Кажуть, що квадратна матриця А і квадратна матриця У того ж розміру є зворотними по відношенню один до одного, якщо виконано хоч одне з матричних рівності – або *А ∙ В =* Е або *В ∙ А = Е*. Можна перевірити, що якщо одне з цих матричних рівності виконується, то виконується та інше. Про всяк випадок пояснимо, що слова «матрична рівність» означають рівність по елементне, тобто якщо має місце рівність матриць *U = V*, то це означає, що у них рівні кількості рядків і рівні кількості стовпців і при цьому uij = vij для всіх *і* та *j*. Зокрема, система лінійних алгебраїчних рівнянь (2.3) може бути записана у вигляді *A ∙ X = B*, де

Неважко помітити, що якщо для квадратної матриці *А* потрібно знайти зворотну матрицю *В*, то треба вирішити *n* систем лінійних алгебраїчних рівнянь з однією і тією ж матрицею: запишемо матричне рівність *А ∙ В = Е* поелементно (тут все вийде *n2* рівності), причому будемо послідовно прирівнювати стовпці - перший стовпець лівій частині до на одну колонку правій частині , другий стовпчик лівій частині на дві колонки правій частині і т.д. –

З цього спостереження випливає, що знайти обернену матрицю можна вирішуючи одночасно n систем лінійних алгебраїчних рівнянь з однією і тією ж матрицею А;

Приклад. Потрібно знайти матрицю, зворотну по відношенню до матриці зазначеній у табл 2.5, а зворотну матрицю позначимо у табл. 2.6. Робочу таблицю, враховуючи, що належить вирішувати п'ять систем лінійних алгебраїчних рівнянь з однією і тією ж матрицею коефіцієнтів, запишемо у табл. 2.7.

Таблиця 2.5.

Вихідна матриця

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 1 | 0 | 1 |
| 0 | 1 | 3 | 2 | 1 |
| 0 | 0 | 1 | 2 | 1 |
| 0 | 0 | 0 | 1 | 5 |
| 0 | 0 | 0 | 0 | 2 |

Таблиця 2.6.

Зворотна матриця

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *x*11 | *x*12 | *x*13 | *x*14 | *x*15 |
| *x*21 | *x*22 | *x*23 | *x*24 | *x*25 |
| *x*31 | *x*32 | *x*33 | *x*34 | *x*35 |
| *x*41 | *x*42 | *x*43 | *x*44 | *x*45 |
| *x*51 | *x*52 | *x*53 | *x*54 | *x*55 |

Таблиця 2.7.

Робоча таблиця п’яти систем рівнянь

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | *x*11 | *x*12 | *x*13 | *x*14 | *x*15 |
|  |  |  |  |  | *x*21 | *x*22 | *x*23 | *x*24 | *x*25 |
|  |  |  |  |  | *x*31 | *x*32 | *x*33 | *x*34 | *x*35 |
|  |  |  |  |  | *x*41 | *x*42 | *x*43 | *x*44 | *x*45 |
|  |  |  |  |  | *x*51 | *x*52 | *x*53 | *x*54 | *x*55 |
| 1 | 0 | 0 | 0 | 0 | 1 | 2 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 3 | 2 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 2 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 5 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 |

Тут зашифровані відразу п'ять систем рівнянь з однією і тією ж матрицею; перша система рівнянь відповідає на одну колонку шуканої матриці; її невідомі - в самому верху таблиці, а вільні члени - ліворуч, в крайньому стовпці. Друга система рівнянь відповідає дві колонки; її невідомі у другому рядку зверху, а вільні члени - у другому стовпці сторінки. І так далі.

Метод Жорданових винятків працює таким чином: на кожній ітерації алгоритму вибирається дозволяє елемент ("*pivot*"), що знаходиться на головній діагоналі матриці (номер рядка та стовпця дозволяє елемента відповідає номеру ітерації), який обов'язково повинен бути ненульовим. Його значення запам'ятовується і на його місце ставиться одиниця. Далі у всіх елементів рядка, в якій знаходиться дозволяє елемент, знак змінюється на протилежний (крім, зрозуміло, самого дозволяє елементу). Потім всі елементи матриці, за винятком елементів рядка та стовпця дозволяє елемента, піддаються наступного перерахунку:

(2.4)

Заключною операцією кожної ітерації є розподіл матриці на збережений раніше елемент "*pivot*".

Кожна ітерація міняє місцями рядок і стовпчик "допоміжних" матриць *E* і *X*, а точка заміни визначається що дозволяє елементом. Таким чином, після виконання всіх ітерацій, матриці *E* і *X* повністю міняються місцями. Нижче представлений код функції знаходження оберненої матриці методом Жорданових винятків (табл. 2.8) та тестування функції JordanInv для знаходження оберненої матриці (табл. 2.9).

Таблиця 2.8.

Функція знаходження оберненої матриці методом Жорданових винятків

Таблиця 2.9.

Тестування функції JordanInv для знаходження оберненої матриці

Застосовуємо метод Жорданових винятків для знаходження оберненої матриці:

Як видно з представленого результату, метод Жорданових винятків, призначений для знаходження оберненої матриці, працює коректно.

Неважко помітити, що в процесі роботи з матрицями виходять дуже великі числа. Особливо це стосується обчислення томів для відновлення. Для забезпечення можливості коректних обчислень використовується арифметика поля Галуа. У ньому працює вся теорія чисел, поле містить числа тільки кінцевого розміру, при розподілі відсутні помилки округлення.

#### 2.1.5 Важливе доповнення до розрахунку матриці кодування

Якщо всі обчислення проводити у звичайних числах, то ніяких проблем з матричної арифметикою не виникає. Однак, в реальних системах, зважаючи на величезних чисел, що виникають у процесі роботи з матрицями, використовується спеціальна арифметика полів Галуа. По-суті вона працює не з самими числами, а з їх залишками (в рамках поля заданого розміру). Це означає, що при обчисленні різниці двох значно відрізняються один від одного чисел може виникнути такий результат, який в полях Галуа буде нулем. Природно, що на такий результат ділити не можна. У випадку алгоритму пошуку оберненої матриці це ставить нас перед фактом виявленої сингулярності. У пунктах [2.1.1](#Розрахунок_матриці_Вандермонда) та [2.1.2](#Створення_томів_для_відновлення) для кодування та декодування використовувалися рядки з матриці Вандермонда. Однак, в реальних системах, використання такого алгоритму означає наступну проблему: якщо в процесі відновлення, при підстановці відповідних рядків з матриці Вандермонда, буде порушено послідовний порядок, обернена матриця буде сингулярною. Тобто якщо для відновлення буде використаний набір томів 1,2,3,4,5 або 3,4,5,6,7 або 5,6,7,8,9 ... - то все пройде гладко. Але такі варіанти проходження томів для відновлення, ймовірно, приведуть до сингулярності: 1,2,4,5,6 або 3,5,6,7,8 або 9,8,5,6,7 ..., хоча, формально, такий порядок коректним.

Для раннього усунення сингулярності матриці відновлення, замість рядків з матриці Вандермонда, беруть рядки з дисперсної матриці, яка має стільки рядків, скільки всього томів у наборі, і стільки стовпців, скільки основних томів, тобто (*n* + *m, n*). Перші *n*-строк такої матриці містять нулі в рядку і одиницю на головній діагоналі. Наступні за ними *m*-рядків містять рядки, що замінюють в пунктах [2.1.1](#Розрахунок_матриці_Вандермонда) та [2.1.2](#Створення_томів_для_відновлення) рядки з матриці Вандермонда. Така дисперсна матриця виходить за допомогою додаткової обробки з матриці Вандермонда розмірністю (*n + m, n*), але що заповнюється за зміненою формулою:

(2.5)

де *i* - номер рядка, *j* - номер стовпця (відлік *i* і *j* починається з нуля). Крім того, під невизначеністю "00" слід розуміти "1", тобто "00 = 1".

Додаткова обробка полягає в наступному: треба послідовно пройти по всіх елементів головної діагоналі, виконуючи для кожного діагонального елемента наступну обробку: якщо поточний діагональний елемент дорівнює нулю, шукаємо в розташованих нижче рядках такий рядок, де елемент в цьому ж стовпці не дорівнює нулю і міняємо ці рядки місцями. Потім, якщо діагональний елемент не дорівнює "1", множимо весь поточний стовпець на його зворотне значення (ділимо всі елементи поточного стовпця на діагональний елемент, включаючи його самого). Таким чином, на кожній ітерації спуску по головній діагоналі після таких дій діагональний елемент стає дорівнює одиниці. Наступним дією є віднімання з усіх стовпців матриці поточного стовпця, помноженої на множник, що знаходиться на перетині рядка діагонального елемента і віднімається стовпця. Віднімання стовпців по елементне, множник, природно, застосовується до всіх елементів. Після того, як будуть опрацьовані всі елементи головної діагоналі матриці, вона стане дисперсної (перша *n*-строк міститимуть нулі, і одиницю на головній діагоналі, а останні *m*-строк будуть матрицею кодування, тобто будуть заміняти собою рядки з матриці Вандермонда). При використанні такого способу кодування можна застосовувати будь-які комбінації томів для відновлення.

Розглянемо приклад зазначеної вище обробки матриці "A":

Рядок "0" вже є обробленої рядком (складається з нулів і має одиницю на головній діагоналі). Тому переходимо на наступний діагональний елемент. У відповідності з алгоритмом: С0 = (С0 - A1, 0 ∙ С1) і С2 = (С2 - A1, 2 ∙ С1) тобто С0 = (С0 - 1 ∙ С1) і С2 = (С2 - 1 ∙ С1). У коефіцієнта Ai, j індекс i залежить від поточного діагонального елементу (його рядки), а j - від поточного стовпця, з яким працюємо (для кожного діагонального елемента виробляємо віднімання "діагонального" стовпця з усіх інших, не забуваючи про відповідний множник). В результаті отримуємо:

Все, що залишилося обробити - це рядок 2. С0 = (С0 - 3 ∙ С2) і С1 = (С1 - 2 ∙ С2). Нижче представлена дисперсна матриця, останні три рядки якої повинні використовуватися в RAID-подібному кодері / декодері Ріда-Соломона.

Існує оптимізований (альтернативний) варіант заповнення матриці кодування, що виграє у описаного вище в швидкості старту, але програє за максимальною кількістю оброблюваних томів: 32768 проти 65535. Суть методу полягає в наступному: при формуванні матриці кодування використовуються тільки ті числа-ступеня, логарифм підстави яких взаємно простий з 65535. Матриця заповнюється безпосередньо, без проміжної обробки (за формулою (2.5) у табл. 2.10). Основна умова - вибір "правильного" підстави ступеня i для кожного рядка. Даний спосіб заповнення матриці хоча і досить швидкий, однак, має суттєвий недолік: сингулярність може виникнути, якщо число томів для відновлення в точності дорівнює кількості втрачених томів даних. Якщо є перестановка рядків і хоча б один зайвий тому відновлення, то звернення матриці гарантовано.

Матриця Коші (на відміну від матриці Вандермонда) має гарантією до обігу при розмірності N \* N.

Крім матриці Вандермонда, є ще чимало іменних матриць. Для звичайних чисел матриця Вандермонда є відомим прикладом матриці, у якої кожна під матриця не вироджена. А матриця Гільберта 1, 1 / 2, 1 / 3 ... є улюбленим прикладом погано обумовленої матриці, непридатною до обчислень. У кінцевому полі ситуація може змінитися з точністю до навпаки.

Таблиця 2.10.

Заповнення матриці Коші

i:=0;

while i<m do

begin

j:=0;

while j<n do

begin

Cauchy[i,j]:= GFRec(GFExp[i+n] xor GFExp[j]);

inc(j);

end;

inc(i);

end;

Позначення:

n – кількість томів даних;

m – кількість томів для відновлення;

GFRec – зворотне значення елемента поля Галуа.

### 2.2 Оптимізація матричних обчислень

Матричні обчислення - основна складова завантаження центрального процесора при кодуванні або декодування. При збільшенні матриці на вектор з використанням арифметики поля Галуа досить розумно не обчислювати дискретний логарифм елементів матриці кожного разу, коли він потрібен, а використовувати заздалегідь про логарифмовану матрицю (втім, як і вхідний вектор). Операція XOR (додавання або віднімання в полях Галуа) використовується в оптимізованому варіанті безпосередньо, а не через клас GF16 з метою збільшення швидкості. Для того, щоб уникнути використання логічних умов в операції множення (адже логарифм нуля не існує, і потрібні перевірки) можна надійти в такий спосіб - використовувати таблицю експонент учетверенного розміру. При цьому перша її половина - це сама таблиця експонент з копією (щоб не приводити результат суми до розміру поля), а решта заповнена нулями. При цьому логарифм нуля покладається таким, щоб сума логарифмів завжди вказувала у частину таблиці експонент, заповнену нулями, якщо хоча б один з співмножників дорівнює нулю.

Існує ще один досить важливий момент в оптимізації: тип масиву матриці кодування. Будь-який двовимірний масив може бути легко і просто замінено на одновимірний, з використанням складного індексу за схемою:

array[i][j] → array[(i \* кількість стовпців) + j];

Це дає приблизно 1,5 .. 2 - разовий виграш у швидкодії.

Лістинг функцій множення прологарифмованої матриці на вектор (кодер та декодер) наведено у додатку А.

### 2.3. Криптографічна складова системи – RSB алгоритм

Відповідно до основних положень концепції Національної безпеки

України витік конфіденційної інформації вважається однією з найбільш небезпечних загроз безпеки в інформаційній сфері [1, 2]. Основним комплексним заходом щодо захисту національного інформаційного простору є побудова Національної системи конфіденційного зв'язку, в якій за допомогою криптографічних і технічних засобів захисту інформації реалізуються послуги інформаційної безпеки [3].

Основними та найбільш ефективними механізмами криптографічного захисту інформації є методи симетричних блокового та потокового криптографічного перетворення [4-6]. Поряд з високою швидкістю перетворень і простотою практичної реалізації симетричні крипто-алгоритми дозволяють забезпечити високу стійкість до різних методів криптографічного аналізу [6, 7]. У той же час, на сьогоднішній день в Україні немає національних стандартів алгоритмів блокового і поточного симетричного криптографічного перетворення інформації. Нині чинний стандарт ГОСТ 28147-89 застарів і не забезпечує виконання сучасних вимог [8-10]. З жовтня 2006 року в Україні проводиться відкритий конкурс симетричних блокових криптографічних алгоритмів для визначення перспективного криптографічного алгоритму, на основі якого в подальшому може бути розроблено національний стандарт України [11, 12].

Сучасні методи захисту інформації (шифрування) в комп'ютерних мережах представляють собою математичні перетворення (алгоритми), в яких повідомлення розглядаються як числа або алгебраїчні елементи в деякому просторі [13]. Криптографічні алгоритми відображають область «свідомих повідомлень» (вхідний текст) в область «безглуздих повідомлень» (вихідний або шифротекст, шифрограма). З позицій теорії сигналів та процесів за шифрування вхідного (корельованого, надмірного) тексту полягає в його «відбілюванні». При цьому початковий текст перетворюється у некорельовану послідовність символів (елементів) шифрограми (практично нестисливої) зі щільністю розподілу ймовірностей елементів вихідного алфавіту, що максимально наближається до рівномірної.

Для того щоб мати можливість відновити інформацію, шифрувальні перетворення повинні бути зворотними. Процес зворотного перетворення шифрограми у вихідний текст називається розшифруванням. Алгоритми шифрування (за шифрування, розшифрування) параметризуються за допомогою ключів. Сукупність алгоритмів шифрування, а також опис формату повідомлень (вхідного відкритого тексту) і простір ключів утворюють криптографічну систему (шифр). У тому випадку, коли для за шифрування і розшифрування використовується один і той же ключ (або ключ розшифрування досить легко обчислюється з ключа за шифрування), то такі криптосистеми називають криптосистемами із секретним (симетричним) ключем (або симетричними криптосистемами). Саме розробці алгоритмів такого класу симетричних криптосистем присвячена ця науково-технічна робота.

Крім розробки алгоритмів криптографічного захисту інформації передбачається також розробка алгоритмів управління ключами шифрування (генерацію та розподіл ключів між акредитованими абонентами комп'ютерної мережі).

#### 2.3.1 Синтез алгоритму симетричного блокового RSB шифрування

Абревіатура RSB походить від ключових слів Rоund, Step, Blok, підкреслюючи тим самим, що основними для крипто-алгоритму є раундові перетворення, розбиті на певну кількість кроків, а дія алгоритму здійснюється над блоками відкритого або закритого текстів. Це ітераційний блочний шифр, який дає унікальну можливість по зміні як розмірів секретних ключів, так і числа кроків (раундів) шифрування [14-15]. Відмінна особливість алгоритму полягає у використанні функції шифрування типу ковзного кодування, яка забезпечує не лише глибоке перемішування відкритого тексту, але і бере участь у формуванні блокового раундового ключа (визначення дається нижче) для чергового блоку, що шифрується. Тим самим всі перетворення, що виконуються крипто-алгоритмом, стають залежними не тільки від секретного ключа, а й від даних, що шифруються, тобто відносяться до класу «керованих операцій крипто-перетворень», або «керованих крипто-примітивів». Термін «ковзне кодування» швидше за все вперше вводиться в криптографії. Суть цієї функції криптографічного розкривається у параграфі 2.3.3.2.

#### 2.3.2 Загальна характеристика RSB алгоритму

Алгоритм передбачає три варіанти довжини блоку (для застосування в різних класах безпеки) і змінну довжину ключа шифрування для кожного варіанта довжини блоку. Структура RSB алгоритму ідентична для різних розмірів блоку, що дорівнює 128*n*, де *n* − коефіцієнт кратності довжини блоку, який може приймати значення 1, 2 або 3, тобто алгоритм підтримує блоки довжиною 128, 256 і 512 біт. Опис алгоритму наводиться в основному для довжини блоку 256 біт і відзначаються особливості реалізації шифру для інших розмірів блоку.

Основні параметри RSB алгоритму:

* Размір блоку: *N*= 128, 256 або 512 біт.
* Довжина раундового ключа - 32 біта.
* Довжина загального (крокового) ключа: *r*\*32, *r*=1,2,..
* Число кроків шифрування: *s*=1,2,…
* Загальне число раундів шифрування:*r\*s*.
* Розмір елементів ковзного кодування - 32 біта.
* Розмір елементів нелінійної заміни: 8 біт.

Узагальнена структурна схема RSB алгоритму наведена на рис. 2.1.

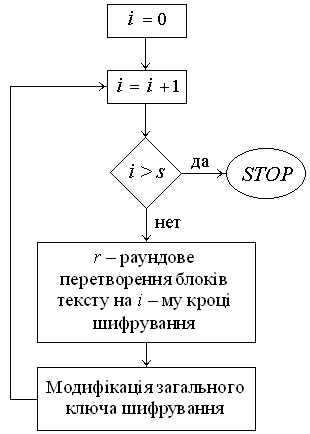


Рис. 2.1. Узагальнена структурна схема RSB алгоритму

Дана схема відображає процес перетворення текстів як для алгоритму за шифрування вхідного відкритого тексту, так і розшифрування криптограми. Тому надалі ми обмежимося в основному поясненням лише організації процесу за шифрування відкритого тексту.

Перед початком процедури за шифрування вхідний відкритий текст розбивається на блоки розміром в ***N*** біт. Якщо останній блок виявився менше обраного розміру, то він доповнюється (пробілами) до повного блоку. Назвемо такий текст розширеним файлом. Обсяг розширеного файлу в ході за шифрування не змінюється, тому обсяг шифротекста завжди буде кратним розміру блоку.

Розгорнута структурна схема RSB алгоритму в режимі за шифрування наведена на рис. 2.2.

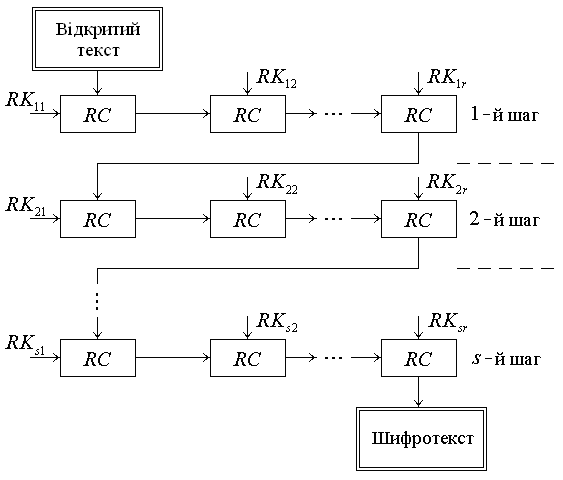


Рис. 2.2. Структурна схема RSB алгоритму в режимі за шифрування

На рисунку 2.2 використані такі позначення:

*  (*Rоund Code*) − операції за шифрування тексту раундовим ключем (підключем загального ключа);
*  − -й раундовий ключ на -му кроці за шифрування.

Таким чином, RSB алгоритм (як і більшість сучасних симетричних блокових шифрів) складається з великої кількості раундів, що повторюються. Як випливає з структурної схеми крипто-алгоритму (рис. 2.2), спочатку виконуються послідовні перетворення всіх блоків розширеного файлу раундовим ключем *RK1*, потім ключем *RK2*  і, нарешті, ключем *RKr*. На цьому закінчується обробка тексту на першому кроці за шифрування. За умови, що число кроків шифрування більше одиниці, відбувається часткове оновлення (модифікація) базових раундовий ключів за рахунок циклічного (кругового) зсуву на сім розрядів вліво загального ключа (рис. 2.3).

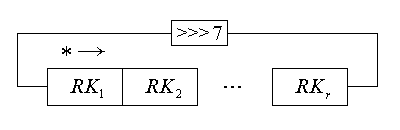


Рис. 2.3. Схема модифікації базових раундовий ключів на етапі за шифрування

Надалі процедура перетворення, що описана вище, повторюється на черговому кроці за шифрування.

Природно, що на етапі розшифрування послідовність базових раундовий ключів *RK* повинна бути інверсної по відношенню до послідовності раундовий ключів за шифрування. Узагальнена структурна схема RSB алгоритму в режимі розшифрування наведена на рис.2.4.

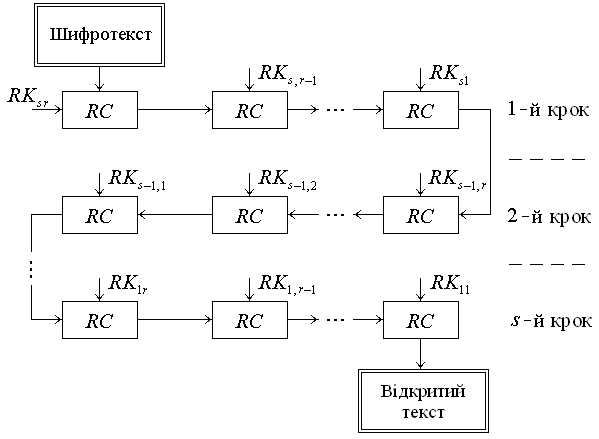


Рис. 2.4. Узагальнена структурна схема алгоритму розшифрування криптограми

Процес розшифрування починається на першому кроці під управлінням спочатку базового раундового ключа *RKsr*, потім ключа *RKs,r-1* і, нарешті, ключа *RKs1*. Перехід до чергового кроку розшифрування супроводжується модифікацією базових раундовий ключів, що здійснюється за рахунок циклічного зсуву на сім розрядів, але тепер вже вправо, загального ключа (рис. 2.5), після чого процедура перетворення повторюється на черговому кроці.

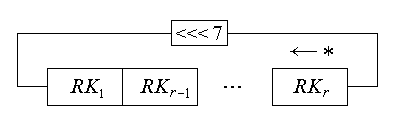


Рис. 2.5. Схема модифікації базових раундовий ключів на етапі розшифрування

Символом \* позначений стартовий базовий раундовий ключ на етапі за шифрування (рис. 2.3) та розшифрування (рис. 2.5), а стрілка вказує, в якому напрямку обираються базові раундовий ключі на черговому кроці шифрування.

Кожний раунд за шифрування RSB алгоритму включає наступну послідовність криптографічних примітивів:

* стохастична кругова прокрутка блоку, що шифруєтся;
* ковзне кодування 32-розрядних елементів блоку;
* стохастична нелінійна підстановка (заміна) байтів блоку;
* стохастична перестановка елементів блоку, розмір яких залежить від розміру блоку.

Структурна схема раундового перетворення блоку, що шифрується, в загальному вигляді представлена на рис. 2.6.

На рис. 2.6 позначені:

* «InText» − вхідний текст;
* «OutText» − вихідний текст;
* «KeyBlock» − раундовий ключ блоку, що шифрується;
* «ShiftRow» − лінійне перетворення блоку на основі циклічного зсуву (кругової прокрутки), що параметризується блочним раундовим ключем Key Block;
* «SlideCode» − лінійне перетворення блоку на основі ковзного кодування, що параметризується блочним раундовим ключем;
* «SubByte» − нелінійне перетворення байтів блоку (підстановка), що параметризується блочним раундовим ключем;
* «PermuBox» − лінійне перетворення на основі перестановки елементів блоку, яке виконується за допомогою інволютивної матриці, що обирається в залежності від стану блокового раундового ключа.

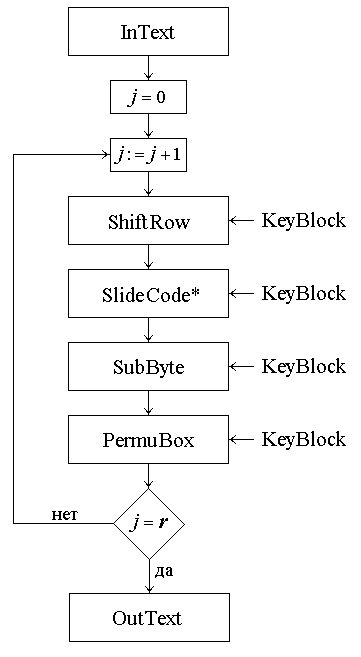


Рис. 2.6. Структурна схема раундового перетворення

Природно, що на етапі розшифрування операції перетворення, яки входять до циклу на рис. 2.6, повинні виконуватися у зворотному порядку.   
Управління примітивами здійснюється вмістом раундових ключів, структура яких наведена на рис. 2.7.

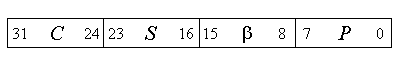


Рис. 2.7. Структура раундового ключа

Таким чином, 32-розрядний раундовий ключ складається з чотирьох восьми розрядних секторів (байтів), за допомогою яких здійснюються такі функціональні перетворення блоків тексту, що шифрується:

*  − циклічний зсув;
*  − нелінійна заміна байтів;
*  − параметризація аддитивної компоненти в операторі заміни;
*  − перестановка байтів блоку, що шифрується.

Як буде показано далі, раундові ключі, за керуванням яких здійснюється шифрування блоків, змінюються кожного разу при переході до чергового блоку, що перетворюється. Така модифікація блокових раундових ключів досягається операцією ковзного кодування тексту, що міститься в попередніх блоках. У силу зазначеної особливості 32-розрядні компоненти загального ключа шифрування (рис. 2.3 і 2.5) вище названі базовими раундовими ключами, а результат їх перетворення − функцією ковзного кодування, будемо називати блочними раундовими ключами. Для першого блоку тексту, що шифрується, блочний раундовий ключ збігається з базовим.

#### 2.3.3 RSB криптографічні примітиви

Далі наводиться більш детальний опис основних криптографічних примітивів, а саме стохастична кругова прокрутка, ковзне кодування, нелінійна заміна байтів, стохастична перестановка елементів блоку.

##### **2.3.3.1** **Стохастична кругова прокрутка блоку**

За допомогою даної операції здійснюється циклічний зсув (кругова прокрутка) блоку, що шифрується, на випадкове непарне число, яке задається восьми розрядним двійковим байтом C. Сім старших розрядів цього байту зчитуються з сектора блокового раундового ключа (розряди 31–25 на рис. 2.7), а в молодший розряд кодової комбінації примусово записується одиниця. Тим самим код, яким визначається порядок циклічного зсуву блоку, буде містити непарне число в інтервалі від 1 до 255. Схема організації циклічного зсуву, яка вище описана, використовується в шифрах для розмірів блоків, що дорівнюють 256 і 512 біт. У тому випадку, коли N=128 біт, порядок зсуву визначається сім'ю молодшими розрядами регістру, тобто є непарним числом, що належить інтервалу від 1 до 127.

##### **2.3.3.2** **Ковзне кодування**

Операція ковзного кодування над 32-розрядними елементами блоків виконує в RSB крипто-алгоритмі подвійну роль. По-перше, вона забезпечує досить глибоке перемішування тексту, що перетворюється. Мета подібного перетворення полягає в тому, щоб зробити якомога складнішою залежність між ключем і шифротекстом. *І*, по-друге, за допомогою такої операції здійснюється модифікація блокових раундовий ключів, під керуванням яких виконуються функціональні перетворення блоків тексту, починаючи з другого. У результаті такої модифікації блочний раундовий ключ *i*-го блоку (*i*>1) стає залежним як від вхідного базового раундового ключа, за керуванням якого здійснюються перетворення першого блоку тексту, так і від даних усіх попередніх (*i*-1)-х блоків, що шифруються.

У RSB шифрі реалізовані два типи ковзного кодування: ліво- і правостороннє, причому лівостороннє ковзне кодування застосовується на парних раундах, а правостороннє – на непарних раундах шифрування. Структурна схема алгоритму прямого лівостороннього ковзного кодування на етапі за шифрування першого блоку наведена на рис. 2.8.

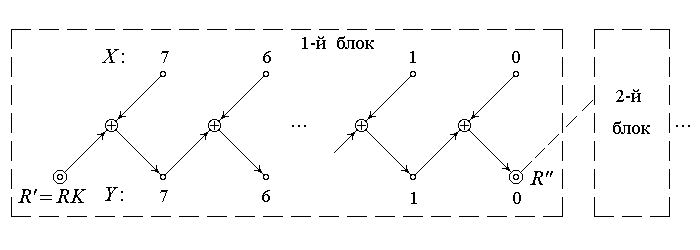


Рис. 2.8. Структурна схема алгоритму прямого лівостороннього

ковзного кодування

Нехай  () − 32-розрядні елементи (слова) шифруємого блоку, а *yi* − відповідні елементи блоку після виконання операції лівостороннього ковзного кодування. Згідно з рис. 2.8, маємо

 (2.6)

де  є оператор порозрядного додавання за модулем 2, а − 32-розрядний вхідний (базовий) раундовий ключ, який приймає значення  () на -му раунді за шифрування (див. рис. 2.3).

Як випливає з алгоритму прямого лівостороннього ковзного кодування (рис. 2.8), блоковим раундовий ключем для другого блоку на етапі за-шифрування є ключ

 ,

де символом  позначена порозрядна сума за  по всіх восьми 32-розрядних елементах першого блоку (у припущенні, що розмір блоку дорівнює 256 бітам).

Таким чином, для довільного -го блоку () блоковим раундовийм ключем є ключ, що утворений порозрядним підсумовуванням за  базового раундового ключа  і всіх 32-розрядних елементів (слів) тексту, що шифрується та передує -му блоку.

Ковзне кодування є лінійним оборотним перетворенням. Якщо система рівнянь (2.6) відповідає прямому лівосторонньому ковзному кодуванню тексту, що шифрується, то для зворотного перетворення, яке застосовується на етапі розшифрування, відповідає система:

 (2.7)

Структурна схема алгоритму оберненого лівостороннього ковзного кодування, що відповідає системі перетворень (2.7), показана на рис. 2.9.

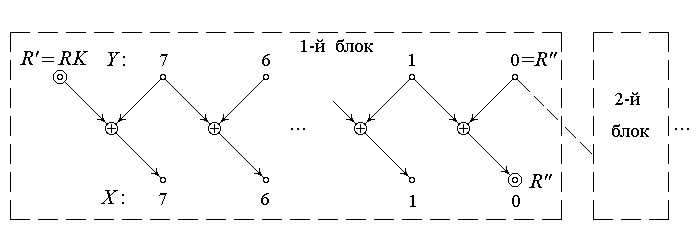


Рис. 2.9. Структурна схема алгоритму оберненого лівостороннього

ковзного кодування

Правостороннє ковзне кодування, що виконується на парних раундах шифрування, розвивається у напрямку справа наліво, починаючи з молодшого (правого) 32-розрядного елемента останнього блоку розширеного файлу до старшого елементу першого блоку. Структурна схема прямого правостороннього ковзного кодування показана на рис. 2.10.

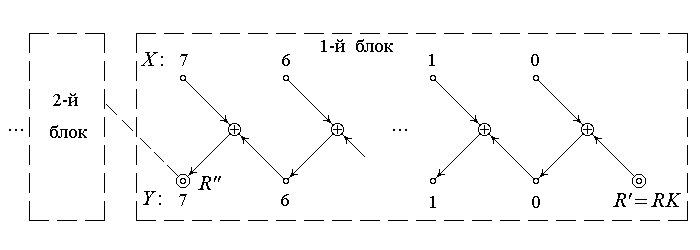


Рис. 2.10. Структурна схема прямого правостороннього ковзного кодування

Правосторонньому ковзному кодуванню (рис. 2.10) відповідає система рівнянь

 (2.8)

Згідно з рис. 2.10 старший 32-розрядний елемент  першого блоку, що перетворений, використовується як блоковий раундовий ключ  для другого блоку. Правостороннє зворотне ковзне кодування виконується за схемою, що наведена на рис. 2.11.

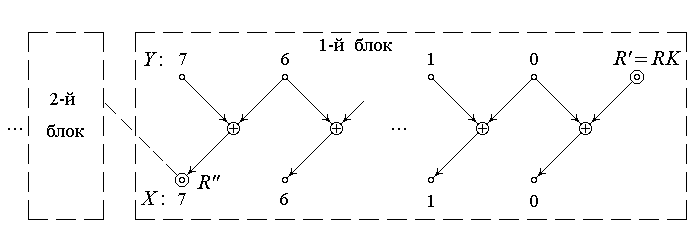


Рис. 2.11. Структурна схема оберненого правостороннього ковзного кодування

Оберненому правосторонньому ковзному кодуванню відповідає система рівнянь:

 (2.9)

Примітив типу лівостороннього ковзного кодування в загальних рисах нагадує схему шифрування в режимі зчеплення блоків шифротексту − CBC (*Ciphertext Block Chaining*) [26]. Відзначимо наступні особливості ковзного кодування в порівнянні з режимом CBC.

По-перше, в CBC здійснюються перетворення над блоками шифротексту, тоді як у шифрі RSB ковзному кодуванню піддаються 32-розрядні елементи блоків на кожному раунді шифрування, причому базові раундови ключі в шифрі RSB виконують роль синхропосилки (вектора ініціалізації) режиму.

По-друге, в RSB алгоритмі ковзним кодуванням досягається не тільки глибоке перемішування шифротексту (як і в режимі CBC), що підвищує крипостійкість шифру, але і відбувається істотна модифікація блокових раундовий ключів, яки управляють перетвореннями наступних блоків тексту.

У результаті таких перетворень блокові раундові ключі, починаючи з другого, стають залежними не тільки від базового раундового ключа (компоненти загального ключа шифрування), але також і від даних усіх попередніх блоків тексту, що шифрується.

І, нарешті, по-третє, крім лівостороннього в алгоритмі застосовується також правостороннє ковзне кодування (для тих же самих цілей, що і лівостороннє). Остання зазначена особливість RSB шифру є унікальною і не зустрічається в жодному з відомих блокових криптографічних алгоритмів.

Звернемо увагу на те, що запропоновані схеми ковзного кодування (рис. 2.4 − 2.11) і системи рівнянь (2.8 − 2.11), які відповідають їм, є нічим іншим, як перетвореннями Грея «навпаки». Тобто, пряме ковзне кодування є ніщо інше, як відповідне (по напрямку) обернене перетворення Грея, тоді як обернене ковзне кодування являє собою пряме перетворення Грея.

##### **2.3.3.3** **Нелінійна заміна байтів**

У сучасних блокових криптосистемах операції нелінійної заміни (підстановки, що реалізуються та званими S-боксами, або S-перетвореннями) виконуються над двійковими кодами невеликої розрядності. Так в популярному у недалекому минулому криптоалгоритмі DES розмір елемента заміни становив чотири бита, а в російському шифрі ГОСТ 28147-89, як і в американському крипто стандарті AES, елементом заміни є байт. У RSB криптосистеми в якості елемента заміни також обрана восьми розрядна бінарна комбінація − байт.

Схема побудови S-боксів, за допомогою яких реалізуються операції нелінійної заміни байтів в RSB алгоритмі, успадковує основні риси S-боксів, що прийняті у AES шифрі. S-перетворення AES алгоритму являє собою нелінійну заміну байтів, яке виконується незалежно з кожним байтом блоку, що обробляться. Таблиці замін S-боксів інвертуємі і побудовані з композиції двох таких перетворень:

1) одержанням зворотного елементу щодо множення в розширеному полі Галуа , нульовий елемент  переходить сам у себе;

2) застосуванням перетворення над GF(2), що визначається співвідношенням

(2.10)

В афінному перетворенні (2.12) використані змінні: *x*-1 − восьми розрядний елемент (байт), що мультиплікативно обернений байту *x* над незвідним двійковим поліномом, в якості якого обраний поліном восьмого порядку ; *M* − обернена циклічна матриця перетворення, перший рядок якої складається з елементів 10001111; *C* − фіксована адитивна компонента перетворення, що дорівнює 11000110.

Мультиплікативний обернений елемент *x*-1 над полем *GF*(28) як раз і доставляє перетворенню (2.10) нелінійні властивості.

У RSB алгоритмі в якості нелінійної підстановки використовується варіант підстановок, що введений в п. 2.3.3.2, а саме

, (2.11)

де *S* и β − байти раундового ключа, що показано на рис. 2.12; *M* − інволютивна матриця перетворення, яка задана співвідношенням

*M* = .

Рис. 2.12. Інволютивна матриця перетворення

Втім, як *M* можна обрати будь-яку іншу двійкову інволютивну матрицю восьмого порядку, тобто матриця *M* може служити в якості довгострокового ключа шифрування.

##### **2.3.3.4** **Стохастична перестановка елементів блоку**

За допомогою цього криптографічного примітиву здійснюється стохастична перестановка (перемішування) двійкових слів в межах блоку, що шифрується. У якості слова в 128-розрядному блоці виступає байт, в 256-розрядному блоці словом є два, а в 512- розрядному блоці − чотири байти; тобто кожен блок містить 16 слів. Ця операція реалізується у такий спосіб. Наприклад, в 128-розрядному блоці міститься 16 байтів, яким надамо десяткові номери від 0 до 15. Нехай *x* означає чотири розрядний двійковий номер байту блоку, що шифрується. Байт, що розташований в осередку блоку, двійковий номер якого дорівнює *x*, переміщується в комірку пам’яті під номером *y*, причому

, (1.14)

де (*a*)2 означає приведення результатів порозрядного матричного добутку до лишку за , а *M*p − матриця перестановці.

У якості *M*p обирається одна з 16-ти інволютивних двійкових матриць четвертого порядку. Адреса *A* матриці *M*p  міститься в секторі *P* блокового раундового ключа *RK* і утворюється за правилом

, (1.15)

тобто, адреса *A* утворюється в результаті порозрядного складання за mod2 полубайтів *p*1 и *p*2 сектора *P* блокового раундового ключа (рис 2.12).

Операція стохастичної перестановки (2.13) здійснюється в RSB шифрі табличним способом, тобто для заданого номера осередку байта *x* з таблиці витягується номер комірки, у який переміщується байт під номером *x*.

### 2.4. Висновки

Як результат можемо підсумувати обрану стратегію проектування вибором алгоритму Ріда-Соломона на основі RAID системи для забезпечення високих показників завадостійкості даних у проектованій системі. А для забезпечення конфіденційності каналу зв’язку розроблено новий алгоритм що отримав назву «RSB алгоритм» в якому використовуються декілька криптографічних примітивів, які в поодинокому варіанті мають досить слабкі характеристики захисту інформації.

# РОЗДІЛ 3

## Синтез та аналіз ефективності завадостійкого кодування та криптографічного захисту

### 3.1 Арифметика полів Галуа

У цифровому світі на ділення цілих та дійсних чисел накладені достатньо жорсткі обмеження. Поділ - досить вимоглива до процесорним ресурсів операція, крім того, вона ще й математично неточна. Тобто, якщо *c = a · b*, то ще не факт, що *a = c / b*! Таким чином, для практичної реалізації кодів Ріда-Соломона звичайна арифметика непридатна і доводиться вдаватися до допомоги особливої математики - математики кінцевих груп *Галуа*.

Під групою тут розуміється сукупність цілих чисел, послідовно пронумерованих від 0 до 2n - 1 наприклад: (0, 1, 2, 3) або (00h 01h, 02h, 03h, 04h, 05h, 06h, 07h, 08h, 09h, 0Ah, 0Bh, 0Ch, 0Dh, 0Eh, 0Fh ). Групи, що містять 2n елементів, називаються полями Галуа *(Galois Field)* і позначаються так: GF (2n). Члени груп в обов'язковому порядку підпорядковуються асоціативному, комутативність і дистрибутивних законами, але обробляються досить протиприродним на перший погляд чином:

1. Сума двох будь-яких членів групи завжди присутня в даній групі;

2. Для кожного члена "а" групи існує тотожний *(identity)* йому член, зазвичай записується як "e", що задовольняє наступні умови:

a + e = e + a = a; (3.1)

3. Для кожного члена "a" групи, існує зворотний *(inverse)* йому член "-a", такий що:

a + (-a) = 0; (3.2)

Почнемо з першого тези. Припустимо, у нас є група (0, 1, 2, 3). Яким чином при обчисленні значення 2 + 3 можна отримати число менше або рівне 3?! Виявляється, складання в полях Галуа здійснюється без урахування перенесення і сума двох членів групи дорівнює: *c = (a + b)* mod *2n*.

Стосовно до нашого випадку: (2 + 3) mod 4 = 1. Ця операція називається "складанням за модулем 4". Додавання за модулем ми машинально виконуємо десятки разів на день, навіть не замислюючись про те, що це і є складання без урахування перенесення. Наприклад, прокинувшись о шостій ранку, ви просиділи за комп’ютером дев’ять годин підряд, а потім несподівано кинули погляд на свої наручний годинник. Який стан займала годинна стрілка в цей час, за умови, що годинник йдуть точно? Шукалося значення з усією очевидністю є сумою 6 і 9 по модулю 12 і одно воно: (6 + 9) mod 12 = 3. Ось вам наочний приклад практичного використання арифметики Галуа. А тепер давайте в порядку експерименту віднімемо з числа 3 число 6 ... (якщо не здогадуєтеся, як це правильно зробити, візьміть у руки годинник).

Тепер найголовніше: раз, результат ділення одного члена групи на інший, природно, нерівний нулю член, в обов'язковому порядку повинен бути присутнім в цій групі, то, незважаючи на те, що розподіл здійснюється в цілих числах, воно буде точним. Точним, а не округлений! Отже, якщо *c = a ·b,* то *a = c / b*. Іншими словами, множення і ділення несуперечливим чином визначено для всіх членів групи, звичайно, за винятком неможливості поділу на нуль, причому, розширення розрядної сітки при збільшенні не відбувається.

Звичайно, це не зовсім звичайне множення (і далеко не в кожному полі Галуа двічі по два буде рано чотирьом), однак, ніхто і не вимагає від арифметики Галуа її відповідності "здоровому глузду" і "життєвому досвіду". Головне, що вона працює, причому працює добре. І існування жорстких дисків, CD-ROM / DVD приводів - найкраще тому підтвердження, бо всі вони, так чи інакше, використовують цю арифметику в своїх цілях.

Як уже говорилося, в обчислювальній техніці найбільшого поширення отримали поля Галуа з основою 2, що пояснюється природністю цих полів з точки зору машинної обробки, двійковій за своєю природою. Для реалізації кодера / декодера Ріда-Соломона нам буде потрібно чотири базових арифметичних операції: додавання, віднімання, множення і ділення. Нижче вони будуть розглянуті у всіх подробицях.

#### 3.1.1 Додавання / віднімання

Додавання за модулем два в полях Галуа тотожне відніманню і реалізується бітової операцією XOR. Це питання ми вже обговорювали при вивченні поліноміальною арифметики, тому не будемо зайвий раз повторюватися, а просто наведемо закінчений приклад програмної реалізації функції складання / віднімання (табл. 3.1):

Таблиця 3.1.

Функція, що реалізує складання / віднімання в полях Галуа

// Функція повертає результат додавання (віднімання)

// двох поліномів a і b за модулем 2

int gf\_sum (int a, int b)

(

return a ^ b;

)

#### 3.1.2 Множення

Відкривши підручник математики за третій клас (якщо мені не зраджує пам'ять), ми знайдемо, що множення є багаторазове складання і, коли незабаром, складання в полях Галуа ми виконувати вже навчилися, ми маємо всі підстави вважати, що реалізація функції множення не створить особливих труднощів. Так? А ось і ні! З'ясувалося, що існують і такі математики, де двічі два не дорівнює чотирьом, а операція множення визначається не через складання, а зовсім по-іншому.

Дійсно, якщо спробувати "обернути" функцію gf\_sum в цикл, ми одержимо те ж саме складання тільки в профіль. a \* b дорівнюватиме а, якщо b парних, і нуля, якщо b - непарній. Власне, функція "справжнього" множення Галуа настільки складна і ресурсомістка, що для спрощення її реалізації припадає до тимчасового перетворення поліномів в індексний форму, подальшого складання індексів, що виконується за модулем GF, і зворотного перетворення суми індексів у поліноміальних форму.

Що таке індекс? Це - показник ступеня при підставі дві, що дає шуканий полином. Наприклад, індекс полінома 8 дорівнює 3 (23 = 8), а індекс полінома 2 дорівнює 1 (21 = 2). Легко показати, що a \* b = 2i + 2j = 2 (i + j). Зокрема, 2 \* 8 = 23 + 21 = 2 (3 +1) = 44 = 16.

Складемо таку таблицю 3.2 і трохи по експериментуємо з нею:

Таблиця 3.2

Таблиця поліномів (ліва колонка) і відповідних їм ступенів двійки (права колонка)

|  |  |
| --- | --- |
| i | alpha\_of[i] |
| 001 | 0 |
| 002 | 1 |
| 004 | 2 |
| 008 | 3 |
| 016 | 4 |

До цих пір ми оперували поняттями звичною нам арифметики і тому два третини полів таблиці залишилися порожніми. Справді, рівняння типу 2x = 3 в цілих числа не розв'язні і ряд індексів не відповідає ніяким полінома! Так-то, воно так, але в силу того, що кількість поліномів всякого поля Галуа дорівнює кількості всіляких індексів, ми можемо певним чином зіставити їх один одному, закривши очі на те, що з точки зору звичайної математики така дія не має ніякого сенсу. Конкретна схема зіставлення може бути будь-який, головне - щоб вона була внутрішньо несуперечливою, тобто задовольняла всіма правилами груп, перерахованим вище (див. "поля Галуа").

Природно, оскільки від обраної схеми зіставлення прямо залежить і кінцевий результат, обидві сторони (кодер і декодер Ріда-Соломона) повинні дотримуватися певні домовленості. Однак, різні кодери / декодери Ріда-Соломона можуть використовувати різні схеми зіставлення, несумісні між собою.

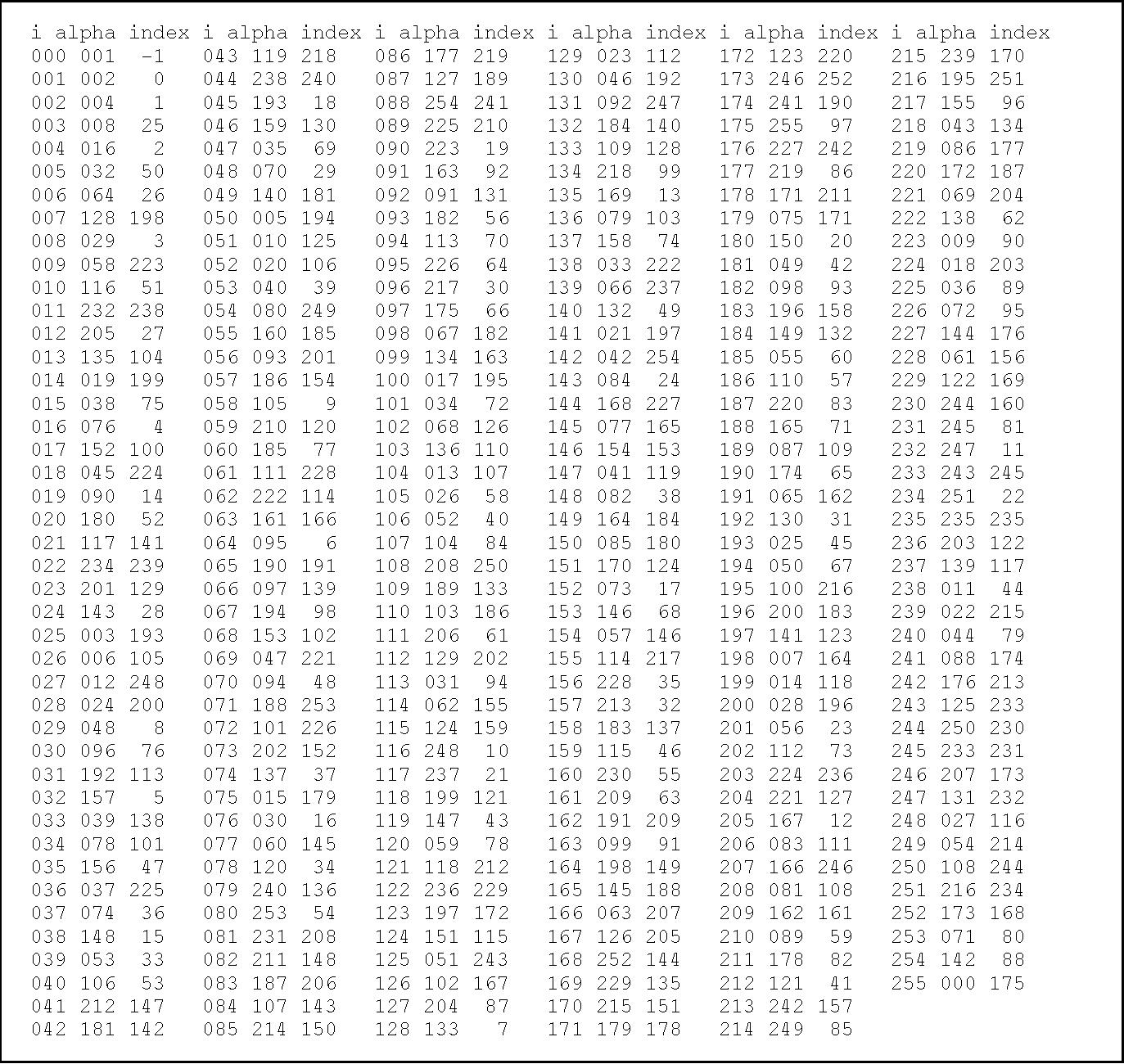
Зокрема, декодер Ріда-Соломона, вбудований в CD-ROM привід, виконує множення по таблиці 3.3. Зустрівши таку таблицю в дізассемблерном лістингу досліджуваної вами програми, ви зможете швидко і надійно ототожнити використовують її функції.

Перша зліва колонка - поліноми / індекси (звичайно позначається як i), другий - таблиця ступенів примітивного поліному 2 (звичайно позначається як alpha\_of), третя - індекси, відповідні даному полиному (звичайно позначається як index\_of).

За допомогою даної таблиці ви легко зможете здійснювати перетворення з поліноміальною форми в індексний, і навпаки (стосовно до звичної арифметики індекс - це логарифм числа, а в стовпці поліномів знаходяться результати зворотного для логарифмування операції - потенціювання). Як користуватися цією таблицею? Припустимо, ми хочемо помножити поліноми 69 та 96. Замінимо твір сумою логарифмів з наступним потенціюванням результату. Знаходимо в першій колонці число 69. Йому відповідає index 221, запам'ятовуємо (записуємо його на папірці) і переходимо до числа 96, index якого дорівнює 30. Складаємо 221 і 30 за модулем 256, отримуючи в результаті: (221 + 30) mod 256 = 251. Тепер переводимо результат твори з індексного форми в поліноміальних: знаходимо в першій колонці число 251 і в другій колонці бачимо відповідний йому поліном: 216. (Якщо ж ми виконаємо зворотну операцію, розділивши 216 на 69 - ми отримаємо 96, що доводить несуперечність операцій ділення та множення, а також всієї арифметики Галуа в цілому). Швидко, чи не правда, хоча місцями і не зовсім зрозуміло, чому таблиця складена саме так, а не інакше? Найгірше, що достовірність результату не можна відчути "в живу", оскільки все це - абстракції чистої води, що серйозно ускладнює налагодження програми (складно налагоджувати те, чий принцип роботи до кінця не розумієш). Функція швидкого табличного множення поліномів в полях Галуа наведена у таблиці 3.4.

Таблиця 3.3.

Look-up-таблиця для GF (256)



Таблиця 3.4.

Функція швидкого табличного множення поліномів в полях Галуа

// Функція повертає результат множення двох поліномів a на b в полях Галуа

int gf\_mul(int a, int b)

{

int sum;

if (a == 0 || b == 0) return 0; // «логарифм нуля не існує!»

sum = index\_of[a] + index\_of[b]; // обчислюємо суму індексів поліномів

if (sum >= GF-1) sum -= GF-1; // наводимо суму до модуля GF

return alpha\_of[sum]; // переводимо результат у поліноміальну...

// ... форму і повертаємо результат

}

#### 3.1.3 Ділення

Розподіл в полях Галуа здійснюється практично точно так, як і множення з тією лише різницею, що індекси не додаються, а віднімаються одне з одного. У самому справі: a / b = 2i/2j = 2 (i-j). Для переведення з поліноміальною в індексний форму і навпаки може використовуватися вже приводиться вище look-up таблиця.

Природно, не забувайте про те, що функція (табл. 3.5) розподілу повинна бути забезпечена перевіркою ділення на нуль.

Таблиця 3.5.

Функція швидкого табличного ділення поліномів в полях Галуа

|  |
| --- |
| // Функція повертає результат ділення двох поліномів a на b в полях Галуа  // При спробі поділу на нуль функція повертає -1  int gf\_div(int a, int b)  {int diff;  if (a == 0) return 0; // «логарифм нуля не існує!» |

Продовження табл. 3.5.

if (b == 0) return -1; // на нуль ділити не можна!

diff = index\_of[a] - index\_of[b]; // обчислюємо різницю індексів

if (diff < 0) diff += GF-1; // наводимо різниця до модуля GF

return alpha\_of[diff]; } //переводимо результат у поліноміальну

// форму і повертаємо результат

### 3.2 Практична реалізація арифметики полів Галуа на С #

Лістинг реалізації арифметики полів Галуа на С # наведено у додатку Б. Проте розглянемо деякі особливості програмної реалізації роботи с полями Галуа (табл. 3.6):

Таблиця 3.6.

Фрагмент коду функції зведення до степеня елемента поля Галуа

// Наводимо результат до розмірів поля (старші байти складаємо з молодшими)

// І повертаємо значення експоненти

return \_GFExp[((pow >> \_GFPower) & \_GFSize) + (pow & \_GFSize)];

Представлений вище лістинг 3.4 потребує додаткових коментарів.

Наприклад, потрібно дізнатися суму цифр числа 167. Це можна зробити за одну операцію в калькуляторі, взявши результат операції mod 9. Перевірте, чи вийде число 5. І це можна зробити з будь-яким цілим числом. Так само і в арифметиці Галуа. Якщо в якості "цифри" поля Галуа виступають 16 біт, то для обчислення суми двох "цифр" розміром 16 біт (що представляють разом змінну типу integer розрядністю 32 біта) потрібно старшу частина зрушити вправо на відповідну кількість розрядів, так, щоб в отриманому тимчасове результаті залишилися тільки старші 16 біт з 32. Їх необхідно скласти з виділеними за допомогою маски битами молодшої "цифри" (16 біт). Результат складання НЕ вимагає приведення до розміру поля, тому що використовується розширена таблиця потенціювання.

### 3.3. Опис архітектури програми «RS Кодер-декодер»

Програмна система виконана повністю на мові C# з використанням Microsoft Visual Studio 2008. Для зручності використання ядро системи виконано у форматі збірок, що дозволяє використовувати його під довільними користувацькими оболонками в рамках. NET.

#### 3.3.1 Діаграма класів

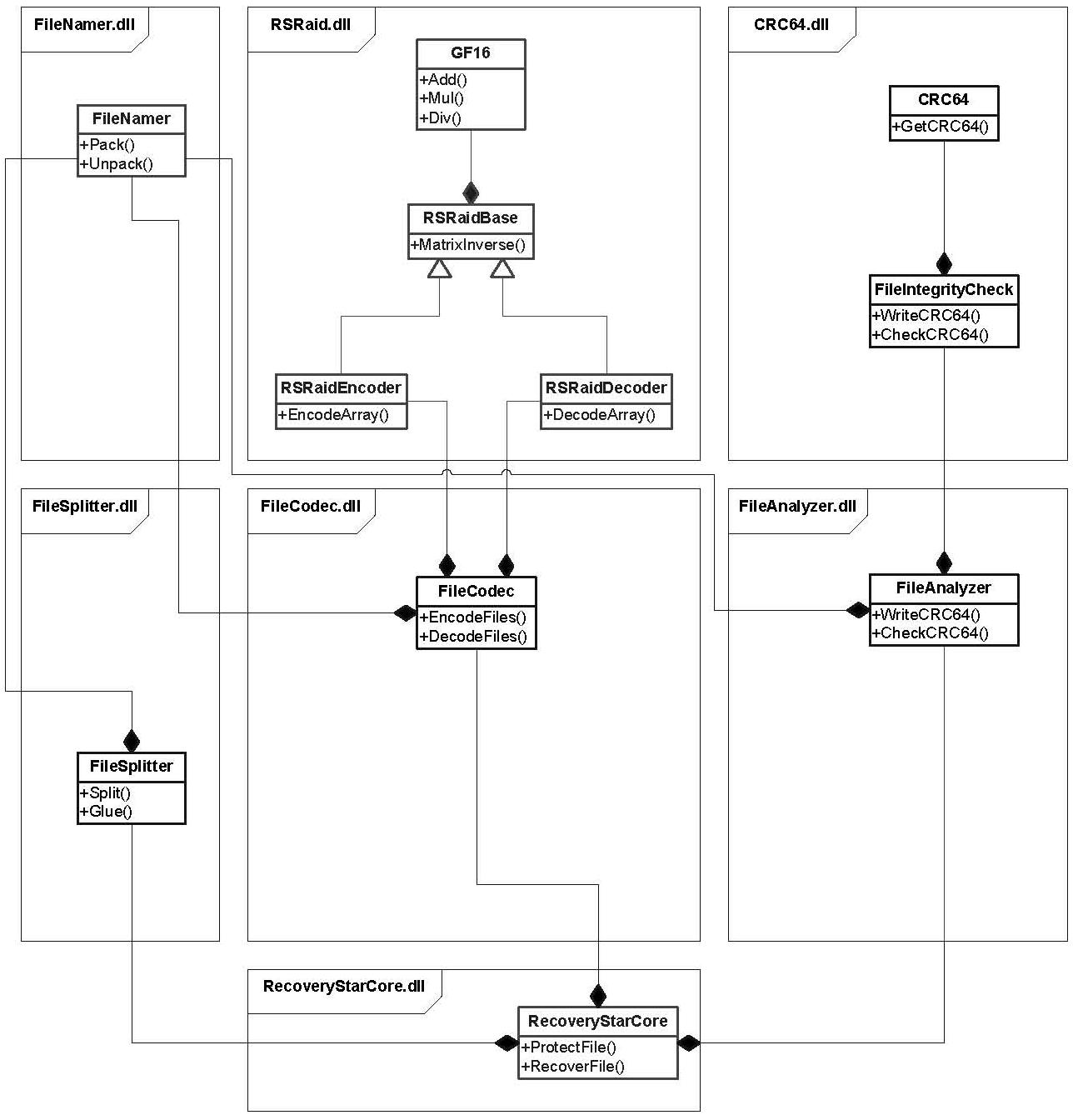
****

Рис. 3.1. Діаграма класів ядра системи з розділенням на інкапсулюючі модулі

Примітка: на даній діаграмі, з метою підвищення компактності візуального подання, не відображує вміст Common.dll, що включає в себе константи і делегати, використовувані іншими збірками, а також клас SystemInfo (надання даних за обсягом фізичної та доступної оперативної пам'яті). Клас SystemInfo реалізує допоміжну функціональність і система могла б функціонувати і без нього).

Таблиця 3.3.

Призначення класів, відображених на рис 3.1

|  |  |
| --- | --- |
| Найменування класу | Призначення класу |
| CRC64 | Розрахунок і перевірка сигнатури цілісності CRC-64, записуваної в кінець файлу. |
| FileIntegrityCheck | Розрахунок і перевірка сигнатур цілісності набору файлів, що породжується розбиванням вихідного на фрагменти-томи. |
| GF16 | Реалізація арифметики поля Галуа GF (2 ^ 16). |
| RSRaidBase | Базовий клас RAID-подібного кодера Ріда-Соломона. |
| RSRaidEncoder | RAID-подібний кодер Ріда-Соломона. |
| RSRaidDecoder | RAID-подібний декодер Ріда-Соломона. |
| FileCodec | Кодер набору файлів (використовує RSRaidEncoder / RSRaidDecoder). |
| FileNamer | Пакувальник / розпакувальник імені файлу в префіксний формат. |
| FileSplitter | Розбиття / склеювання файлу на фрагменти. |
| RecoveryStarCore | Ядро системи, що управляє перерахованими вище модулями. |

#### 3.3.2 Інтерфейс програми «RS Кодер-декодер»

Розроблена программа має досить зручний і зрозумілий інтерфейс який зображено на рис. 3.2.

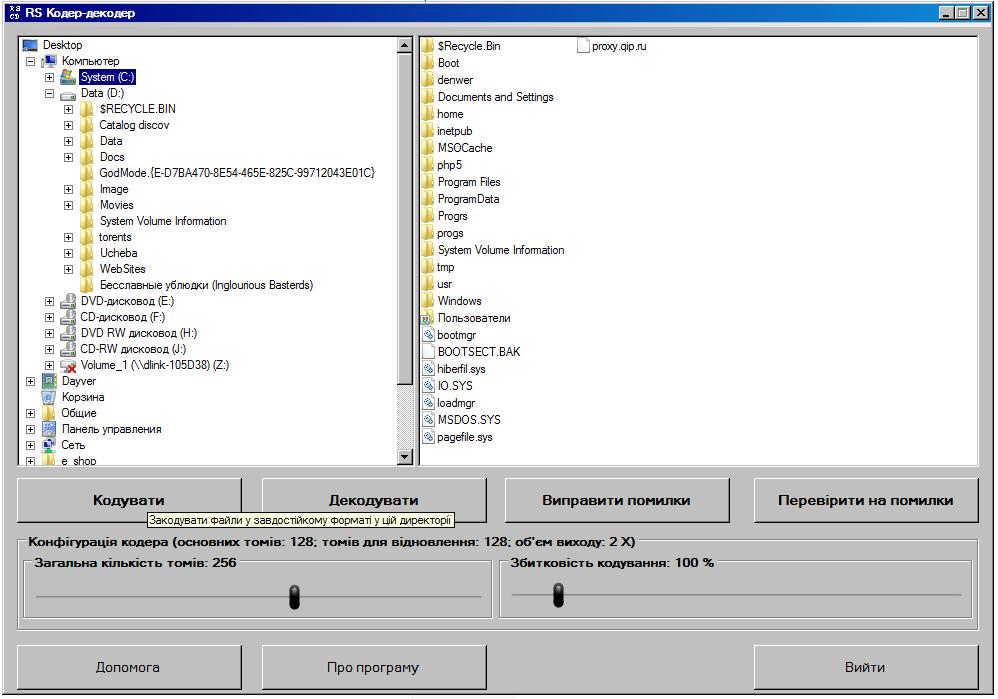


Рис. 3.2. Зовнішній вигляд головного вікна програми «RS Кодер-декодер»

### 3.4 Логіка роботи програми «RS Кодер-декодер»

Спроектована програма «RS Кодер-декодер» складається з чотирьох можливих дій, а саме - процес кодування, декодування, лікування набору томів та тестування наборі томів Далі розглянемо більш детально основні дії що виконує програма «RS Кодер-декодер».

#### 3.4.1 Процес кодування

* Файл розбивається класом FileSplitter на задану кількість фрагментів однакового розміру (імена формуються в оригінальному імені з використанням класу FileNamer шляхом додавання префікса), причому розмір цих фрагментів парний. Потім в кінець кожного файлу пишеться Int64, яке вказує, скільки корисних байт в цьому контейнері.
* Початкове безліч файлів кодується класом FileCodec з отриманням безлічі надлишкових фрагментів. Низькорівневе кодування бере на себе клас RSRaidEncoder.
* Для всіх файлів набору, за допомогою класу FileIntegrityCheck, здійснюється обчислення і дозапісь в кінець сигнатури цілісності CRC-64. Перебір всіх томів здійснює FileIntegrityCheck, кінцеву роботу з кожним файлом - CRC-64.

#### 3.4.2 Процес декодування

* Для всіх файлів набору, за допомогою класу FileIntegrityCheck, здійснюється обчислення і перевірка сигнатури цілісності CRC-64. Перебір всіх томів здійснює FileIntegrityCheck, кінцеву роботу з кожним файлом - CRC-64.
* Початкове безліч файлів декодується класом FileCodec з отриманням безлічі загублених основних фрагментів. Низькорівневе кодування бере на себе клас RSRaidDecoder.
* Оригінальний відновлений файл виходить за допомогою злиття з повного набору основних томів класом FileSplitter з використанням інформації про наявному корисному обсязі даних в межах контейнера.

#### 3.4.3 Процес лікування набору томів

* Для всіх файлів набору, за допомогою класу FileIntegrityCheck, здійснюється обчислення і перевірка сигнатури цілісності CRC-64. Перебір всіх томів здійснює FileIntegrityCheck, кінцеву роботу з кожним файлом - CRC-64.
* Початкове безліч файлів декодується класом FileCodec з отриманням безлічі загублених основних фрагментів. Низькорівневе кодування бере на себе клас RSRaidDecoder.
* Початкове безліч файлів кодується класом FileCodec з отриманням безлічі надлишкових фрагментів. Низькорівневе кодування бере на себе клас RSRaidEncoder.
* Для всіх файлів набору, за допомогою класу FileIntegrityCheck, здійснюється обчислення і дозапісь в кінець сигнатури цілісності CRC-64. Перебір всіх томів здійснює FileIntegrityCheck, кінцеву роботу з кожним файлом - CRC-64.

#### 3.4.4 Процес тестування набору томів

Для всіх файлів набору, за допомогою класу FileIntegrityCheck, здійснюється обчислення і перевірка сигнатури цілісності CRC-64. Перебір всіх томів здійснює FileIntegrityCheck, кінцеву роботу з кожним файлом - CRC-64.

### 3.5. Програмний комплекс з дослідження статистичних властивостей симетричного блочного RSB криптографічного алгоритму

Симетричний RSB (Rоund,Step, Blok) алгоритм належить класу ітераційних блочних шифрів, яки доставляють унікальну можливість по зміні як розмірів секретних ключів, так і числа кроків (раундів) шифрування [13-16]. Відмінна особливість алгоритму полягає у використанні функції шифрування типу ковзного кодування, яка забезпечує не лише глибоке перемішування відкритого тексту, але і бере участь у формуванні блокового раундового ключа (визначення дається нижче) для чергового блоку, що шифрується. Тим самим всі перетворення, що виконуються крипто-алгоритмом, стають залежними не тільки від секретного ключа, а й від даних, що шифруються, тобто відносяться до класу «керованих операцій крипто-перетворень», або «керованих крипто-примітивів».

Детальна характеристика алгоритмічного забезпечення і розробка основних структурно-логічних схем шифратора наведена в матеріалах звіту за перший етап науково-технічної роботи. Тому в даному розділі дається лише короткі відомості, яки необхідні для пояснення технології побудови програмної реалізації алгоритму.

Основні параметри RSB алгоритму такі:

* Розмір блоку шифрування: *N* = 256 біт;
* Довжина раундового ключа - 32 біта;
* Довжина загального (крокового) ключа: ***r*** \* 32, ***r*** = 1,2,…;
* Число кроків шифрування: ***s*** = 1,2,…;
* Загальне число раундів шифрування: ***r*** \* ***s***;
* Розмір елементів ковзного кодування - 32 біта;
* Розмір елементів нелінійної заміни: 8 біт.

Узагальнена структурна схема RSBалгоритму показана на рис. 3.3.

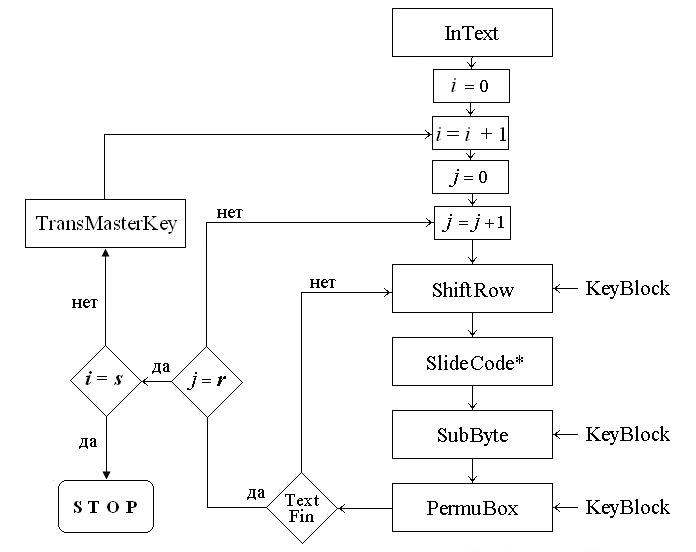


Рис.3.3.Узагальнена структурна схема RSB крипто-алгоритму.

Дана схема відображає процес перетворення текстів як для алгоритму за шифрування, так і розшифрування з урахуванням виконання вимог оберненості перетворень. Тому надалі ми обмежимося в основному поясненням організації процесу за шифрування відкритого тексту.

Структурна схема алгоритму, показана на рис. 3.3, містить два вкладених цикли. Зовнішнім циклом (параметр ***i***) задається крок шифрування (від одного до***s***), а внутрішнім (параметр ***j***) здійснюється ***r***-раундовое шифрування (під яким розуміється як зашифрування, так і розшифрування) тексту. Кожен раунд за шифрування припускає виконання наступних чотирьох криптографічних примітивів над блоками відкритого тексту:

* + - стохастичне прокручування блоку (ShiftRow);
    - ковзне кодування (SlideCode);
    - нелінійна підстановка байтів блоку (SubByte);
    - стохастична перестановка 16-бітних слів блоку (PermuBox).

Структурна схема алгоритму в режимі за шифрування наведена на рис. 3.4.

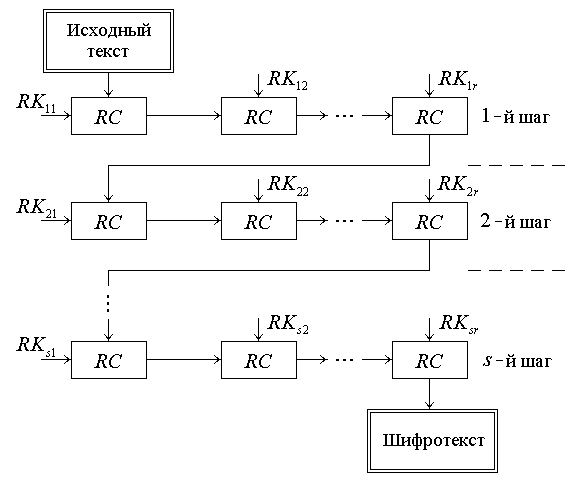


Рис. 3.4. Структурна схема RSB алгоритму в режимі за шифрування

Перед початком процедури за шифрування вхідний відкритий текст (InText) розбивається на блоки, розміром в 256 біт. Якщо останній блок виявився менше обраного розміру, то він доповнюється (прогалинами) до повного блоку. Назвемо такий текст розширеним файлом. Обсяг розширеного файлу в ході за шифрування не міняється, тому обсяг шифротексту завжди буде кратним розміру блоку.

Таким чином, RSB алгоритм (як і більшість сучасних ітераційних симетричних блокових шифрів) складається з великої кількості перетворень − раундів, що повторюються. Як випливає зі структурної схеми крипто-алгоритму (рис. 3.4), спочатку виробляються послідовні перетворення всіх блоків розширеного файлу раундовым ключем RK1, потім ключем RK2 й, нарешті, ключем RKr. На цьому закінчується обробка тексту на першому кроці за шифрування. За умови, що число кроків шифрування ***s*** більше одиниці, відбувається часткове відновлення (модифікація – TransMasterKey на рис. 3.3) загального ключа за шифрування (*Common Key*). Одночасно з модифікацією загального ключа модифікуються й базові раундовые ключі. Це здійснюється за рахунок циклічного (кругового) зрушення на сім розрядів уліво загального (крокового) ключа шифрування СК (рис. 3.5). Далі описана вище процедура перетворення повторюється на черговому кроці за шифрування.

Природно, що на етапі розшифрування послідовність базових раундових ключів повинна бути інверсної стосовно послідовності раундовых ключів за шифрування.

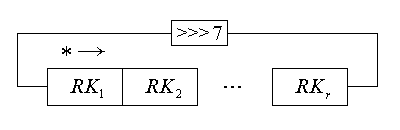


Рис. 3.5*.* Алгоритм модифікації крокового ключа за шифрування

Узагальнена структурна схема RSB алгоритму в режимі розшифрування наведені на рис. 3.6.

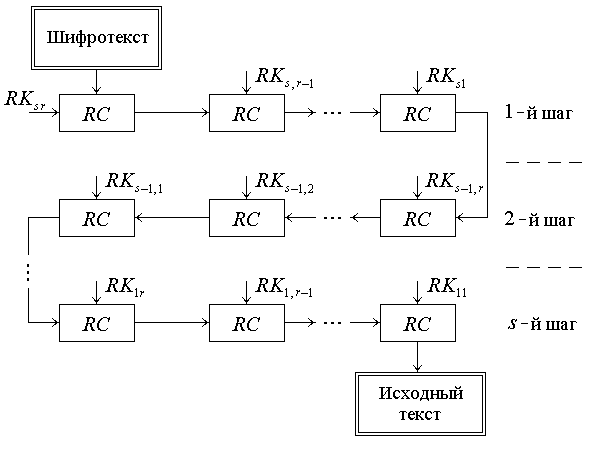


Рис. 3.6. Структурна схема RSB алгоритму в режимі розшифрування

Процес розшифрування починається на першому кроці під керуванням спочатку базового раундового ключа RKs,r, потім RKs,r-1 й, нарешті, RKs,1. Перехід до чергового кроку розшифрування супроводжується модифікацією базових раундовых ключів. Така модифікація досягається за рахунок циклічного зсуву на сім розрядів, але тепер уже вправо, загального ключа СК (рис. 3.6), після чого процедура перетворення повторюється на черговому кроці.

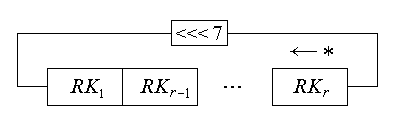


Рис. 3.7. Алгоритм модифікації крокового ключа розшифрування

Керування примітивами здійснюється вмістом раундовых ключів RK (як базових, так і блокових), структура яких наведена на рис. 3.8.

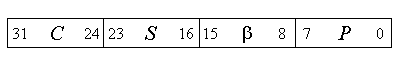


Рис. 3.8. Структурна схема блокового раундового ключа

32-розрядний раундовый ключ RK складається із чотирьох восьми розрядних секторів (байтів), за допомогою яких здійснюються такі функціональні перетворення блоків тексту, що шифрується:

*C* − циклічний зсув;

*S* − нелінійна заміна байтів;

β − параметризація аддитивной компоненти в операторі заміни;

*P* − перестановка 16-бітних слів у межах блоку, що шифрується.

Деталі відомості щодо функціонування основних криптографічних примітивів RSB шифратора, а саме, примітивів підстановки та перестановки наведені у попередніх розділах роботи.

### 3.6. Опис архітектури програми «RSB32 2.0»

Архітектура розробленої программ зображена на рис. 3.9.

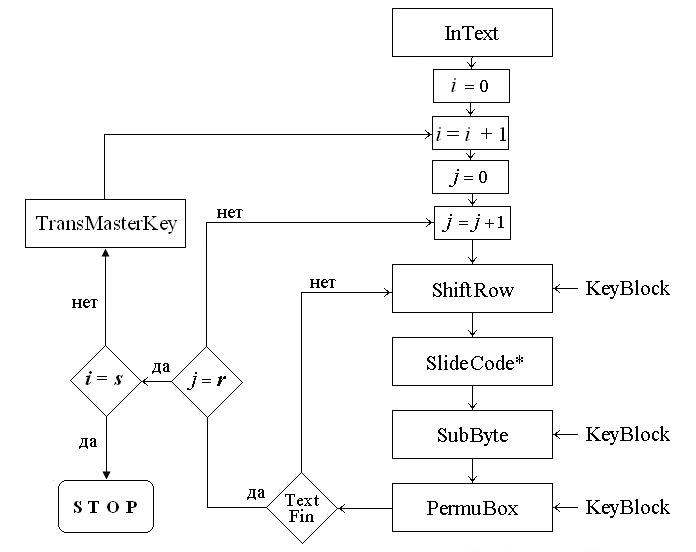


Рис. 3.9. Архітектури програми «RSB32 2.0»

#### 3.6.1 Інтерфейс програми «RSB32 2.0»

Інтерфейс програми «RSB32 2.0» досить простий та має лише найголовніші органи управління. Зовнішній вигляд головного вікна зображено на рис. 3.11.

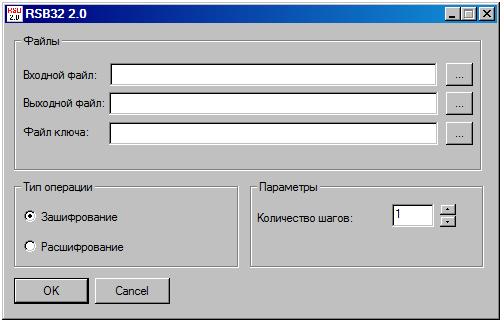
****

Рис. 3.12. Інтерфейс головного вікна програми «RSB32 2.0»

### 3.7. Логіка роботи програми «RSB32 2.0»

Спроектована програма «RSB32 2.0» може виконувати дві дії, а саме - процес кодування та процес декодування. Далі розглянемо більш детально основні дії що виконує програма «RSB32 2.0».

#### 3.7. 1 Процес кодування

Спочатку потрібно вказати «вхідний файл», тобто файл який потрібно закодувати. Надалі вказуємо ім’я та місце на комп’ютері куди потрібно зберегти файл після шифрування. Потім обираємо файл ключа. Вказуємо кількість шагів шифрування. Остання дія – вибір типу операції. Обираємо «Зашифрувати»

#### 3.7.2 Процес декодування

* Спочатку потрібно вказати «вхідний файл», тобто файл який потрібно декодувати.
* Надалі вказуємо ім’я та місце на комп’ютері куди потрібно зберегти файл після розшифрування.
* Потім обираємо файл ключа.
* Вказуємо кількість шагів шифрування (Важливо: кількість шагів повинна співпадати з кількістю шагів при шшифруванні)
* Остання дія – вибір типу операції. Обираємо «Розшифрувати»

### 3.8. Висновки

Підсумком до виконаної роботи можна вважати досягнення поставленої задачі що до розробки програмно-апаратного комплексу завадостійкої системи прийомо-передачі криптографічних даних в ході якого було розроблено дві програми які дозволяють організувати прийомо-передачу без перешкод та можливого втручання несанкціонованою особою.

# ВИСНОВКИ

В роботі на тему «Цифровий апаратно-програмний комплекс завадостійкої системи прийомо-передачі криптографічних даних. Алгоритм завадостійкого кодування-декодування», який розробив Ткаченко П.Л.., керівник, проф. Білецький А.Я., освітлена тема завадостійкого кодування, проаналізовані алгоритми існуючих завадостійких кодів, а також були розроблені програми «RS кодер-декодер» та «RSB32 2.0».

В першій частині був проведений аналіз існуючих методів завадостійкого кодування та алгоритмів що виявляють та виправляють помилки.

В другій частині наведені та розглянуті математичні основи застосованих алгоритмів завадостійкого кодування та елементів криптографічного захисту. А саме наведено процеси матричних обчислень, як елементу RAID-подібного кодування, суть коду Ріда-Соломона та схема застосованого RSB алгоритму.

В третій частині проведено проектування прогамного комплексу завадостійкої та криптографічно захищеної системи для роботи в каналах прийомо-передачі данних. Було зроблено дві програми: «RS кодер-декодер» та «RSB32 2.0». Програма «RS кодер-декодер» (завадостійка компонента розроблюваної системи) може використовуватися як окремо від програми «RSB32 2.0» так і разом та забезпечувати завадостіке кодування інформації для подальшої передачі її в по канал звязку. Програма «RSB32 2.0» (криптографічна складова системи передачі інформації) також в свою чергу може використовуватися яко кремо так а разом з програмою «RS кодер-декодер» та забезпечувати шифрування\розшифруваня конфеденційної інформації для подальшої передачі по незахищеним каналам звязку.

Розглядаючи роботу в цілому, можна зробити такий висновок, що були розглянуті та висвітлені всі поставлені задачі. Розроблені, спроектовані та протестовані програми для завадостійкого та криптографічно захищеного апаратно-програмного комплексу прийомо-передачі даних який функціонально вирішуватиме задачі захисту від завад та несанкціонованого доступу інформації, що передаватиметься цим комплексом.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Алферов А.П., Зубов А.Ю., Кузьмин А.С., Черемушкин А.В. Основы криптографии. Учебное пособие. М., Гелиос АРВ, 2005.

2. Бабаш А.В., Шанкин Г.П. История криптографии. Часть I. М.: Гелиос, 2002.

3. Бабаш А.В., Шанкин Г.П. Криптография. Аспекты защиты. М., Солон-Р, 2002.

4. Бабичев С. Г., Гончаров В. В., Серов Р. Е. Основы современной криптографии. - М.: Горячая линия - Телеком, 2001 - 120 с.

5. Білецький А.Я., Стеценко Д.А., Ткаченко П.Л., Бабич В.В. MARK алгоритм установки ключів шифрування. Защита информации: сборник научных трудов. – Киев: НАУ, 2008. – 323 с.

6. Білецький А.Я., Стеценко Д.А., Ткаченко П.Л., Бабич В.В. Система розповсюдження ключів шифрування. Захист в інформаційно-комунікаційних системах: тези науково-практичної конференції. – Київ: НАУ, 2009.

7. Білецький А.Я., Ткаченко П.Л. Неформальний синтез кодів Хемінга. Електроніка та системи управління – Київ: НАУ, 1(23)-2010, с. 12-19.

8. Венбо Мао. Современная криптографія: теорія и практика.: Пер. с англ. - М.: Издательский дом «Вильямс», 2005 - 768 с.

9. Воробьёв С. Защита информации в персональных ЭВМ. - М.: Издательство «Мир», 1993 - 312 с.

10. Дж. Л. Месси. Введение в современную криптологию. // ТИИЭР, т.76, №5, Май 88 – М, Мир, 1988, с.24-42.

11. Диффи, Хэллман. ”Новые направления в криптографии”.: ТИИЭР, т.67, №3, 1979.

12. У. Диффи. Первые десять лет криптографии с открытым ключом. // ТИИЭР, т.76, №5, Май 88 – М, Мир, 1988, с.54-74.

13. Эрли П. Семья шпионов. М., 1997.

14. В. Жельников. Криптография от папируса до компьютера. – М., ABF, 1996.

15. Закон України “Про авторське право i сумiжнi права” 23.12.1993.

16. Закон України “Про захист інформації в АС” 5.07.1994.

17. Закон України “Про захист інформації в автоматизованих системах” //ВВР № 31 1994.- 286 с.

18. Закон України “Про захист вiд недобросовiсної конкуренцiї”, 7.06.1996.

19. Закон України “Про зв’язок”, 16.05.1995.

20. Закон України “Про державну таємницю” 21.01.1994.

21. Закон України “Про iнформацiю” 2.10.1992.

22. Закон України “Про науково-технiчну iнформацiю” 25.06.1993.

23. Закон України “Про обмеження монополiзму та недопускання недобросовiсної конкуренцiї”, 18.02.1992.

24. Ирвин Дж., Харль Д. Передача даннях в сетях: инженерный поход.: Пер. с англ. - СПб.: БХВ - Петербург, 2003 - 448 с.

25. Кан Д. Взломщики кодов, М.: «Центрполиграф», 2000.

26. Кан Д. Война кодов и шифров. М.: РИПОЛ КЛАССИК, 2004.

27. Концепція технічного захисту інформації в Україні // Урядовий кур'єр, 1997, 12 листопада.

28. Кравченко В. Б. Защита речевой информации в каналах связи/ В. Б. Кравченко //Специальная техника. – 1999. - №4, №5..

29. Леонтьев В. П., Новейшая энциклопедия персонального компьютера 2003. - 5-е изд., перераб. и доп. - М.: ОЛМА-ПРЕСС, 2003. - 957 с.: ил.

30. Математические и компьютерные основы криптологии: Учеб. пособие / Ю. С. Харин, В. И. Берник, Г. В. Матвеев, С. В. Агиевич. - Мн.: Новое знание, 2003 - 382 с.

31. Мафтик С. Механизмы защиты в сетях ЭВМ. - М.: Издательство «Мир», 1993 - 343 с.

32. Молдован А. А. Криптография: скоростные шифры. - СПб.: БХВ - Петербург, 2002 - 496 с.

33. НД ТЗІ 1.1-003-99. Термінологія в галузі захисту інформації в комп'ютерних системах від несанкціонованого доступу. // Департамент спеціальних телекомунікаційних систем та захисту інформації Служби безпеки України. - Київ, 1999.

34. Носов В. В., Орлов П. И., Громыко И. А. организация т обеспечениебезопасности информации. Учебное пособие. - Харьков, 2004 - 141 с.

35. Петров А. А. Компьютерная безопасность. Криптографические методы защиты. - М.: ДМК, 2000 - 448 с.

36. Постанова КМ “Про перелiк вiдомостей, що не становлять комерцiйної таємницi”

37. Программиование алгоритмов защиты информации: Учебное пособие. - М.: «Нолидж», 2000 - 288 с.

38. Романец Ю. В. , Тимофеев П. А., Шаньгин В. Ф. Защита информации в компьютерных системах и сетях / Под ред. В. Ф. Шаньгина. - М.: Радио и связь, 1999 - 328 с.

39. Рябко. Б. Я., Фионов А. Н. Криптографические методы защиты информации: Учебние пособие для вузов. - М.: Горячая линия - Телеком, 2005 - 229 с.

40. Сапегин Л. Н. Специальная техника средств связи, Выпуск 1, Серия «Системы, сети и технические средства конфиденциальной связи», 1996.

41. Скляров Д. В. Искусство защиты и взлома информации. - СПб.: БХВ - Петербург, 2004 - 288 с.

42. Спесивцев А. В. и др. Защита информации в персональных компьютерах. – М., Радио и связь. 1992, с.140-149.

43. Столлингс В. Криптографія и защита сетей: принципы и практика, 2-е издание.: Пер. С англ. - М.: Издательский дом «Вильямс», 2001 - 672 с.

44. Фомичев В. М. Дискретная математика и криптология. Курс лекцій / Под общ. ред. д-ра физ.-мат. н. Н. Д. Подуфалова. - М.: Диалог - МИФИ, 2003 - 400 с.

45. Харченко В.І., Документалістика. - Черкаси: ЧДТУ, 2006. - 147 с.

46. Халяпин Д. Б. Защита информации. Вас подслушивают? Защищайтесь!/ Д.Б. Халяпин. – М: НОУ ШО «Баярд», 2004 – 432 с.– ISBN 5-94896-017-X

47. Черчхаус Р. Коды и шифры. Юлий Цезарь, “Энигма” и Интернет. ¾ М.: “ВЕСЬ МИР”, 2005.

48. Шеннон К. Работы по теории информации и кибернетике.: Пер. с англ. - М.: Издательство иностранной литературы, 1963 - 830 с.

49. Шнайер Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си. - М: Издательство Триумф, 2003 - 816 с.

50. Шнаер Б. Секреты и ложь. Безопасность даннях в цифровом мире. - СПб.: Питер, 2003 - 368 с.

51.A.Menezes, P.van Oorshot, S.Vanstone. Handbook of Applied Cryptography – CRC Press Inc., 1997.

52. Hal Tipton and Micki Krause. Handbook of Information Security Management – CRC Press LLC, 1998.

# Додаток A

Лістинг функцій множення прологарифмованої матриці на вектор (кодер і декодер)

/// <summary>

/// Метод умножения матрицы кодирования на входной прологарифмированный вектор

/// </summary>

/// <param name="dataLog">Прологарифмированный входной вектор (исходные данные)</param>

/// <param name="ecc">Выходной вектор (избыточные данные)</param>

/// <returns>Булевский флаг результата операции</returns>

public bool Process(int[] dataLog, ref int[] ecc)

{

// Если кодер сконфигурирован некорректно, обработка невозможна!

if (!\_configIsOK)

{

return false;

}

// Копируем указатель на массив экспонент для сокращения времени обращения

int[] GF16Exp = \_eGF16.GFExp;

// Вычисление результата умножения матрицы на вектор

for (int i = 0; i < \_m; ++i)

{

int mulSum = 0; // Сумма произведения строки матрицы на столбецint i\_n = i \* \_n; // Смещение в массиве до элементов i-ой строки

for (int j = 0; j < \_n; ++j)

{

mulSum ^= GF16Exp[\_FLog[i\_n + j] + dataLog[j]];

}

ecc[i] = mulSum;

}

return true;

}

/// <summary>

/// Метод множення матрицф кодування на вхфдний прологарифований вектор

/// </summary>

/// <param name="dataEccLog">Прологарифмированный входной вектор (данные + ecc)</param>

/// <param name="data">Выходной вектор (восстановленные исходные данные)</param>

/// <returns>Булевский флаг результата операции</returns>

public bool Process(int[] dataEccLog, ref int[] data)

{

// Если кодер сконфигурирован некорректно, обработка невозможна!

if (!\_configIsOK)

{

return false;

}

// Копируем указатель на массив экспонент для сокращения времени обращения

int[] GF16Exp = \_eGF16.GFExp;

// Вычисление результата умножения матрицы на вектор

for (int i = 0; i < \_n; ++i)

{

// Если текущая строка матрицы не является тривиальной, производим обработку

if (!\_FLogRowIsTrivial[i])

{

int mulSum = 0; // Сумма произведения строки матрицы на столбец

int i\_n = i \* \_n; // Смещение в массиве до элементов i-ой строки

for (int j = 0; j < \_n; ++j)

{

mulSum ^= GF16Exp[\_FLog[i\_n + j] + dataEccLog[j]];

}

data[i] = mulSum;

}

else

{

data[i] = GF16Exp[dataEccLog[i]];

}

}

return true;

}

# Додаток Б

Лістинг реалізації арифметики в GF (16)

/// <summary>

/// Класс поля Галуа

/// </summary>

public class GF16

{

#region Construction & Destruction

public GF16()

{

// Инициализируем таблицы "логарифмирования" и "потенцирования"

GFInit();

}

#endregion Construction & Destruction

#region Public Operations

/// <summary>

/// Сложение элементов поля Галуа

/// </summary>

public int Add(int a, int b)

{

return a ^ b;

}

/// <summary>

/// Вычитание элементов поля Галуа

/// </summary>

public int Sub(int a, int b)

{

return a ^ b;

}

/// <summary>

/// Оптимизированное умножение элементов поля Галуа (без проверок аргументов на ноль)

/// </summary>

public int Mul(int a, int b)

{

return \_GFExp[\_GFLog[a] + \_GFLog[b]];

}

/// <summary>

/// Оптимизированное деление элементов поля Галуа (без проверки первого аргумента на ноль)

/// </summary>

public int Div(int a, int b)

{

// На ноль делить нельзя!

if (b == 0)

{

return -1;

}

// Выражение "+\_GFSize" гарантирует неотрицательное значение индекса

return \_GFExp[\_GFLog[a] - \_GFLog[b] + \_GFSize];

}

/// <summary>

/// Возведение в степень элемента поля Галуа

/// </summary>

public int Pow(int a, int p)

{

// Если показатель степени равен "0", то результат - "1"

if (p == 0)

{

return 1;

}

// Если основание степени равно "0", то результат - "0"

if (a == 0)

{

return 0;

}

24

// Степень числа может быть представлена как произведение

// логарифма основания и показателя степени (с последующим потенцированием)

int pow = \_GFLog[a] \* p;

// Приводим результат к размерам поля (старшие байты складываем с младшими)

// и возвращаем значение экспоненты

return \_GFExp[((pow >> \_GFPower) & \_GFSize) + (pow & \_GFSize)];

}

/// <summary>

/// Вычисление обратного элемента поля Галуа

/// </summary>

public int Inv(int a)

{

// На ноль делить нельзя!

if (a == 0)

{

return -1;

}

return \_GFExp[\_GFSize - \_GFLog[a]];

}

/// <summary>

/// Вычисление логарифма элемента поля Галуа

/// </summary>

public int Log(int a)

{

return \_GFLog[a];

}

/// <summary>

/// Вычисление экспоненты элемента поля Галуа

/// </summary>

public int Exp(int a)

{

return \_GFExp[a];

}

#endregion Public Operations

#region Private Operations

/// <summary>

/// Инициализация таблиц "логарифмирования" и "потенцирования"

/// </summary>

private void GFInit()

{

// Таблица "логарифмирования"

\_GFLog = new int[\_GFSize + 1];

// Таблица "потенцирования"

\_GFExp = new int[(4 \* \_GFSize) + 1];

// Логарифм нуля сделан таким, чтобы выносить результат за пределы

// рабочей области таблицы экспонент, туда, где расположены нули ((0 \* 0) = 0)

\_GFLog[0] = (2 \* \_GFSize);

// Заполняем таблицы логарифмирования и потенцирования

for (int log = 0, b = 1; log < \_GFSize; ++log)

{

\_GFLog[b] = log;

\_GFExp[log] = b;

\_GFExp[log + \_GFSize] = b; // Дополнительная часть таблицы позволяет

//избежать приведения к размеру поля после суммирования результатов

//логарифмирования Удваиваем значение элемента поля, для которого строятся таблицы

b <<= 1;

// Если вышли за размеры поля GF(2^16), приводим значение к нему

if (b > \_GFSize)

{

b ^= \_RSPrimPoly;

}

}

// Заполняем вторую часть таблицы нулями (нужно для оптимизации

// умножения, логарифм нуля сделан таким, чтобы выносить результат за

//пределы рабочей области таблицы экспонент, туда, где расположены

//нули ((0 \* 0) = 0)

for (int i = (2 \* \_GFSize); i < ((4 \* \_GFSize) + 1); ++i)

{

\_GFExp[i] = 0;

}

}

#endregion Private Operations

#region Public Properties

/// <summary>

/// Таблица дискретного логарифмирования GF(2^16)

/// </summary>

public int[] GFLog

{

get

{

return \_GFLog;

}

}

/// <summary>

/// Таблица дискретного потенцирования GF(2^16)

/// </summary>

public int[] GFExp

{

get

{

return \_GFExp;

}

}

#endregion Public Properties

#region Constants

// Несократимый порождающий полином GF(2^16)

private const int \_RSPrimPoly = 0x1100B;

// Степень поля Галуа

private const int \_GFPower = 16;

// Размер поля Галуа

private const int \_GFSize = ((1 << \_GFPower) - 1);

#endregion Constants

#region Data

// Таблица "логарифмирования"

private int[] \_GFLog;

// Таблица "потенцирования"

private int[] \_GFExp;

#endregion Data

}; // Class GF16

# Додаток В

Лістинг програми «RS Кодер-декодер»

/\*----------------------------------------------------------------------+

| filename: MainForm.cs |

|----------------------------------------------------------------------|

| purpose: Отказоустойчивое кодирование по типу RAID-систем |

+----------------------------------------------------------------------\*/

using System;

using System.Collections;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

using System.IO;

namespace RecoveryStar

{

public partial class MainForm : Form

{

#region Data

/// <summary>

/// Объект для упаковки (распаковки) имени в префиксный формат

/// </summary>

private FileNamer eFileNamer;

/// <summary>

/// Массив значений индексов для элементов управления "TrackBar" (количество томов)

/// </summary>

private int[] allVolCountTrackBarValuesArr;

/// <summary>

/// Массив значений индексов для элементов управления "TrackBar" (избыточность)

/// </summary>

private int[] redundancyTrackBarValuesArr;

/// <summary>

/// Общее количество томов

/// </summary>

private int allVolCount;

/// <summary>

/// Избыточность кодирования

/// </summary>

private int redundancy;

/// <summary>

/// Количество основных томов

/// </summary>

private int dataCount;

/// <summary>

/// Количество томов для восстановления

/// </summary>

private int eccCount;

/// <summary>

/// Форма ввода пароля

/// </summary>

private PasswordForm ePasswordForm;

/// <summary>

/// Форма обработки файлов

/// </summary>

private ProcessForm eProcessForm;

/// <summary>

/// Форма тестирования

/// </summary>

private BenchmarkForm eBenchmarkForm;

#endregion Data

#region Construction & Destruction

/// <summary>

/// Конструктор формы

/// </summary>

public MainForm()

{

InitializeComponent();

// Создаем форму ввода пароля

this.ePasswordForm = new PasswordForm();

// Инициализируем экземпляр класса для упаковки (распаковки) имени файла

// в префиксный формат

this.eFileNamer = new FileNamer();

// Инициализируем массивы, хранящие константные значения размера тома и избыточности,

// доступных пользователю

this.allVolCountTrackBarValuesArr = new int[(allVolCountMacTrackBar.Maximum + 1)];

// Задаем стартовые значения степени двойки и полусуммы соседних степеней

int p1 = 2, p2 = 3;

for (int i = 0; i < allVolCountMacTrackBar.Maximum; i += 2)

{

// Ряд доступных нечетных значений количества томов рассчитывается как

// степень двойки, ряд четных значений - как полусумма расположенных рядом

// значений

this.allVolCountTrackBarValuesArr[i + 0] = p1;

this.allVolCountTrackBarValuesArr[i + 1] = p2;

// Увеличиаем степень значений

p1 <<= 1;

p2 <<= 1;

}

// Дописываем не обработанный в цикле элемент

this.allVolCountTrackBarValuesArr[allVolCountMacTrackBar.Maximum] = p1;

this.redundancyTrackBarValuesArr =new int[(redundancyMacTrackBar.Maximum + 1)];

for (int i = 0; i <= redundancyMacTrackBar.Maximum; i++)

{

this.redundancyTrackBarValuesArr[i] = (i + 1) \* 5;

}

}

#endregion Construction & Destruction

#region Private Operations

/// <summary>

/// Метод обработки файлов в выбранной директории

/// </summary>

private void ProcessFiles()

{

// Если в браузере в качестве текущего элемента выбрана директория

if (browser.SelectedItem.IsFolder)

{

// Если в данный момент форма открыта - просто выходим

if (

(this.eProcessForm != null)

&&

(this.eProcessForm.Visible)

)

{

return;

}

// Устанавливаем параметры кодера

this.eProcessForm.DataCount = this.dataCount;

this.eProcessForm.EccCount = this.eccCount;

this.eProcessForm.CodecType = (int)RSType.Cauchy;

// Копируем имена файлов для обработки

for (int i = 0; i < browser.SelectedItem.SubFiles.Count; i++)

{

// Получаем полное имя файла

String fullFileName = ((ShellDll.ShellItem) browser.SelectedItem.SubFiles.Items[i]).Path;

// Получаем короткий вариант имени файла

String shortFileName = this.eFileNamer.GetShortFileName(fullFileName);

// Если имя исходного файла превышает 50 символов - он не может быть обработан

// (потому что при добавлении префикса на выходе получится более 64-х символов)

if (shortFileName.Length > 50)

{

string message = "Довжина імені файлу \"" + shortFileName + "\" перебільшує 50 символів! Пропустити цей файл та продовжити процес формування списку для обробки?";

string caption = " RS Кодер-декодер";

MessageBoxButtons buttons = MessageBoxButtons.YesNo;

DialogResult result = MessageBox.Show(null, message, caption, buttons, MessageBoxIcon.Exclamation, MessageBoxDefaultButton.Button1);

// Если пользователь нажал на кнопку "No" - прерываем обработку...

if (result == DialogResult.No)

{

//...предварительно сбросив установленный режим

this.eProcessForm.Mode = RSMode.None;

return;

}

} else

{

// Если файл присутствует - добавляем его в список на обработку

if (File.Exists(fullFileName))

{

this.eProcessForm.FileNamesToProcess.Add(fullFileName);

}

}

}

// Если размер списка для обработки не равен нулю

// (т.е. есть что обрабатывать) - будем осуществлять обработку

if (this.eProcessForm.FileNamesToProcess.ToArray().Length != 0)

{

// Выходим выше уровнем в директориях, чтобы не тормозить процесс обработки

browser.BrowserUp();

// Отображаем окно обработки

this.eProcessForm.Show();

} else

{

string message = "У вказаній директорії не знайдено доступних файлів для обробки!";

string caption = " RS Кодер-декодер";

MessageBoxButtons buttons = MessageBoxButtons.OK;

MessageBox.Show(null, message, caption, buttons, MessageBoxIcon.Information, MessageBoxDefaultButton.Button1);

// Cбрасываем установленный режим

this.eProcessForm.Mode = RSMode.None;

}

}

}

/// <summary>

/// Метод обработки файлов в выбранной директории с учетом уникального имени,

/// породившего последовательность

/// </summary>

private void ProcessUniqueFiles()

{

// Если в браузере в качестве текущего элемента выбрана директория

if (browser.SelectedItem.IsFolder)

{

// Если в данный момент форма открыта - просто выходим

if (

(this.eProcessForm != null)

&&

(this.eProcessForm.Visible)

)

{

return;

}

// Список уникальных имен файлов для обработки

ArrayList uniqueNamesToProcess = new ArrayList();

// Копируем имена файлов для обработки

for (int i = 0; i < browser.SelectedItem.SubFiles.Count; i++)

{

// Извлекаем очередное имя из списка...

String fullFileName = (String)((ShellDll.ShellItem)browser.SelectedItem.SubFiles.Items[i]).Path;

//...получаем его короткий вариант...

String shortFileName = this.eFileNamer.GetShortFileName(fullFileName);

//...и распаковываем его с получением оригинального имени...

String unpackedFileName = shortFileName;

// Если не удалось корректно распаковать короткое имя - переходим

// на следующую итерацию

if (!this.eFileNamer.Unpack(ref unpackedFileName))

{

continue;

}

//...затем проверяем его на уникальность - если такое имя уже есть

// в словаре "uniqueNamesToProcess", то добавлять его не будем

// Сначала предполагаем, что распакованное имя файла уникально

bool unpackedFileNameIsUnique = true;

// Перебираем весь имеющийся список уникальных имен

foreach (String currUnpackedFileName in uniqueNamesToProcess)

{

// Если обнаружили совпадение - имя не уникально,

// сообщаем об этом и выходим из списка

if (currUnpackedFileName == unpackedFileName)

{

unpackedFileNameIsUnique = false;

break;

}

}

// Если распакованный файл уникален...

if (unpackedFileNameIsUnique)

{

// Если файл присутствует...

if (File.Exists(fullFileName))

{

//...добавляем имя в список уникальных имен...

uniqueNamesToProcess.Add(unpackedFileName);

//...добавляем имя в список для обработки...

this.eProcessForm.FileNamesToProcess.Add(fullFileName);

}

}

}

// Если размер списка для обработки не равен нулю

// (т.е. есть что обрабатывать) - будем осуществлять обработку

if (this.eProcessForm.FileNamesToProcess.ToArray().Length != 0)

{

// Выходим выше уровнем в директориях, чтобы не тормозить процесс обработки

browser.BrowserUp();

// Отображаем окно обработки

this.eProcessForm.Show();

} else

{

string message = "У вказаній директорії не знайдено доступних файлів для обробки!";

string caption = " RS Кодер-декодер";

MessageBoxButtons buttons = MessageBoxButtons.OK;

MessageBox.Show(null, message, caption, buttons, MessageBoxIcon.Information, MessageBoxDefaultButton.Button1);

// Cбрасываем установленный режим

this.eProcessForm.Mode = RSMode.None;

}

}

}

/// <summary>

/// Запуск режима кодирования файла

/// </summary>

private void protectButton\_Click(object sender, EventArgs e)

{

// После клика на кнопке переносим фокус на файловый браузер

browser.Focus();

// Если форма обработки имеет ненулевой хендл и у нее установлен режим -

// идёт обработка и прерывать её нельзя!

if (

(this.eProcessForm != null)

&&

(this.eProcessForm.Mode != RSMode.None)

)

{

return;

}

// Создаем форму кодирования файла

this.eProcessForm = new ProcessForm();

// Делаем форму обработки привязанной к главной форме

this.eProcessForm.Owner = this;

// Закрываем форму, если была открыта в конце обработки

if (this.eProcessForm.Visible)

{

this.eProcessForm.Close();

}

// Если пароль не пуст - устанавливаем безопасность

if (this.ePasswordForm.Password.Length != 0)

{

this.eProcessForm.Security = new Security(this.ePasswordForm.Password);

this.eProcessForm.CBCBlockSize = this.ePasswordForm.CBCBlockSize;

}

// Устанавливаем режим работы

this.eProcessForm.Mode = RSMode.Protect;

// Запускаем обработку файла

ProcessFiles();

}

/// <summary>

/// Запуск режима восстановления данных файла

/// </summary>

private void recoverButton\_Click(object sender, EventArgs e)

{

// После клика на кнопке переносим фокус на файловый браузер

browser.Focus();

// Если форма обработки имеет ненулевой хендл и у нее установлен режим -

// идёт обработка и прерывать её нельзя!

if (

(this.eProcessForm != null)

&&

(this.eProcessForm.Mode != RSMode.None)

)

{

return;

}

// Создаем форму кодирования файла

this.eProcessForm = new ProcessForm();

// Делаем форму обработки привязанной к главной форме

this.eProcessForm.Owner = this;

// Закрываем форму, если была открыта в конце обработки

if (this.eProcessForm.Visible)

{

this.eProcessForm.Close();

}

// Если пароль не пуст - устанавливаем безопасность

if (this.ePasswordForm.Password.Length != 0)

{

this.eProcessForm.Security = new Security(this.ePasswordForm.Password);

this.eProcessForm.CBCBlockSize = this.ePasswordForm.CBCBlockSize;

}

// Используется быстрое извлечение из томов (без проверки CRC-64)?

this.eProcessForm.FastExtraction = быстроеИзвлечениеToolStripMenuItem.Checked;

// Устанавливаем режим работы

this.eProcessForm.Mode = RSMode.Recover;

// Запускаем обработку множества уникальных имен файлов (без учета префиксов)

ProcessUniqueFiles();

}

/// <summary>

/// Запуск режима лечения набора данных файла

/// </summary>

private void repairButton\_Click(object sender, EventArgs e)

{

// После клика на кнопке переносим фокус на файловый браузер

browser.Focus();

// Если форма обработки имеет ненулевой хендл и у нее установлен режим -

// идёт обработка и прерывать её нельзя!

if (

(this.eProcessForm != null)

&&

(this.eProcessForm.Mode != RSMode.None)

)

{

return;

}

// Создаем форму кодирования файла

this.eProcessForm = new ProcessForm();

// Делаем форму обработки привязанной к главной форме

this.eProcessForm.Owner = this;

// Закрываем форму, если была открыта в конце обработки

if (this.eProcessForm.Visible)

{

this.eProcessForm.Close();

}

// Используется быстрое извлечение из томов (без проверки CRC-64)?

this.eProcessForm.FastExtraction = быстроеИзвлечениеToolStripMenuItem.Checked;

// Устанавливаем режим работы

this.eProcessForm.Mode = RSMode.Repair;

// Запускаем обработку множества уникальных имен файлов (без учета префиксов)

ProcessUniqueFiles();

}

/// <summary>

/// Запуск режима тестирования набора данных файла

/// </summary>

private void testButton\_Click(object sender, EventArgs e)

{

// После клика на кнопке переносим фокус на файловый браузер

browser.Focus();

// Если форма обработки имеет ненулевой хендл и у нее установлен режим -

// идёт обработка и прерывать её нельзя!

if (

(this.eProcessForm != null)

&&

(this.eProcessForm.Mode != RSMode.None)

)

{

return;

}

// Создаем форму кодирования файла

this.eProcessForm = new ProcessForm();

// Делаем форму обработки привязанной к главной форме

this.eProcessForm.Owner = this;

// Закрываем форму, если была открыта в конце обработки

if (this.eProcessForm.Visible)

{

this.eProcessForm.Close();

}

// Используется быстрое извлечение из томов (без проверки CRC-64)?

this.eProcessForm.FastExtraction = быстроеИзвлечениеToolStripMenuItem.Checked;

// Устанавливаем режим работы

this.eProcessForm.Mode = RSMode.Test;

// Запускаем обработку множества уникальных имен файлов (без учета префиксов)

ProcessUniqueFiles();

}

/// <summary>

/// Запуск бенчмарка кодирования

/// </summary>

private void тестБыстродействияToolStripMenuItem\_Click(object sender, EventArgs e)

{

// Если форма обработки имеет ненулевой хендл и у нее установлен режим -

// идёт обработка и ей мешать нельзя!

if (

(this.eProcessForm != null)

&&

(this.eProcessForm.Mode != RSMode.None)

)

{

return;

}

// Создаем форму кодирования файла

this.eBenchmarkForm = new BenchmarkForm();

// Устанавливаем параметры кодера

this.eBenchmarkForm.DataCount = this.dataCount;

this.eBenchmarkForm.EccCount = this.eccCount;

this.eBenchmarkForm.CodecType = (int)RSType.Cauchy;

// Делаем форму обработки привязанной к главной форме

this.eBenchmarkForm.Owner = this;

// Закрываем форму, если была открыта в конце обработки

if (this.eBenchmarkForm.Visible)

{

this.eBenchmarkForm.Close();

}

// Отображаем диалоговое окно обработки

this.eBenchmarkForm.ShowDialog();

}

/// <summary>

/// Метод установки конфигурации кодера с отображением соответствующих

/// изменений на форме

/// </summary>

private void SetCoderConfig()

{

// Снимаем данные с элементов управления

this.allVolCount = this.allVolCountTrackBarValuesArr[allVolCountMacTrackBar.Value];

this.redundancy = this.redundancyTrackBarValuesArr[redundancyMacTrackBar.Value];

// Устанавливаем текстовые метки, соответствующую значениям элемента управления

allVolCountGroupBox.Text = "Загальна кількість томів: " + System.Convert.ToString(this.allVolCount);

redundancyGroupBox.Text = "Збитковість кодування: " + System.Convert.ToString(this.redundancy) + " %";

// Абсолютное значение количества процентов на том

double percByVol = (double)this.allVolCount / (double)(100 + this.redundancy);

// Вычисляем количество томов для восстановления

this.eccCount = (int)((double)this.redundancy \* percByVol); // Томов для восстановления

// В случае необходимости корректируем количество томов для восстановления

if (this.eccCount < 1)

{

this.eccCount = 1;

}

// Количество основных томов находим по остаточному принципу

this.dataCount = this.allVolCount - this.eccCount;

// Вычисляем коэффициент выхода

double outX = ((double)(this.dataCount + this.eccCount)) / (double)this.dataCount;

// Вычисляем результирующий объем

String outXStr = System.Convert.ToString(outX);

// Длина подстроки, выделяемой по-умолчанию

int subStrLen = 3;

// Для двузначного значения целой части избыточности нужно брать подстроку

// на один символ больше

if (outX >= 10)

{

subStrLen = 4;

}

// Корректируем (в случае надобности) длину извлекаемой подстроки

if (outXStr.Length < subStrLen)

{

subStrLen = outXStr.Length;

}

// Получаем строковое представление выхода

outXStr = outXStr.Substring(0, subStrLen);

// Преобразуем в число строковое значение избыточности, которое увидит пользователь

double visibleX = System.Convert.ToDouble(outXStr);

// Если в результате преобразования были утеряны значащие цифры, добавляем 0.1

// к выводимому значению

if (visibleX != outX)

{

outX += 0.1;

}

// Вычисляем результирующий объем

outXStr = System.Convert.ToString(outX);

// Длина подстроки, выделяемой по-умолчанию

subStrLen = 3;

// Для двузначного значения целой части избыточности нужно брать подстроку

// на один символ больше

if (outX >= 10)

{

subStrLen = 4;

}

// Корректируем (в случае надобности) длину извлекаемой подстроки

if (outXStr.Length < subStrLen)

{

subStrLen = outXStr.Length;

}

// Получаем строковое представление выхода

outXStr = outXStr.Substring(0, subStrLen);

// Выводим конфигурацию кодера

coderConfigGroupBox.Text = "Конфігурація кодера (основних томів: " + System.Convert.ToString(this.dataCount)

+ "; томів для відновлення: " + System.Convert.ToString(this.eccCount)

+ "; об'єм виходу: " + outXStr + " X)";

}

/// <summary>

/// Обработчик события изменения общего количества томов

/// </summary>

private void allVolCountMacTrackBar\_ValueChanged(object sender, decimal value)

{

// Снимаем данные с элемента управления

this.allVolCount = this.allVolCountTrackBarValuesArr[allVolCountMacTrackBar.Value];

// Устанавливаем текстовую метку, соответствующую значениям элемента управления

allVolCountGroupBox.Text = "Загальна кількість томів: " + System.Convert.ToString(this.allVolCount);

}

/// <summary>

/// Обработчик события изменения избыточности кодирования

/// </summary>

private void redundancyMacTrackBar\_ValueChanged(object sender, decimal value)

{

// Снимаем данные с элемента управления

this.redundancy = this.redundancyTrackBarValuesArr[redundancyMacTrackBar.Value];

// Устанавливаем текстовую метку, соответствующую значениям элемента управления

redundancyGroupBox.Text = "Збитковість кодування: " + System.Convert.ToString(this.redundancy) + " %";

}

/// <summary>

/// Обработчик события отпускания кнопки "бегунка" общего количества томов

/// </summary>

private void allVolCountMacTrackBar\_MouseUp(object sender, MouseEventArgs e)

{

// Устанавливаем конфигурацию кодера

SetCoderConfig();

}

/// <summary>

/// Обработчик события отпускания кнопки "бегунка" избыточности

/// </summary>

private void redundancyMacTrackBar\_MouseUp(object sender, MouseEventArgs e)

{

// Устанавливаем конфигурацию кодера

SetCoderConfig();

}

/// <summary>

/// Ввод пароля

/// </summary>

private void шифрующийФильтрToolStripMenuItem\_Click(object sender, EventArgs e)

{

if (шифрующийФильтрToolStripMenuItem.Checked == true)

{

шифрующийФильтрToolStripMenuItem.Checked = false;

// Удаляем пароль

this.ePasswordForm.ClearPassword();

} else

{

// Выводим диалог ввода пароля

this.ePasswordForm.ShowDialog();

// Если пароль был установлен - фиксируем это

if (this.ePasswordForm.Password.Length != 0)

{

шифрующийФильтрToolStripMenuItem.Checked = true;

}

}

}

/// <summary>

/// Активация режима быстрого извлечения данных (без проверки CRC-64)

/// </summary>

private void быстроеИзвлечениеToolStripMenuItem\_Click(object sender, EventArgs e)

{

if (быстроеИзвлечениеToolStripMenuItem.Checked == true)

{

быстроеИзвлечениеToolStripMenuItem.Checked = false;

} else

{

быстроеИзвлечениеToolStripMenuItem.Checked = true;

}

}

/// <summary>

/// Вызов справки

/// </summary>

private void вызовСправкиToolStripMenuItem\_Click(object sender, EventArgs e)

{

try

{

// Открываем файл справки

System.Diagnostics.Process.Start("Help.html");

}

catch

{

string message = "Не можу відкрити файл допомоги \"Help.html\"!";

string caption = " RS Кодер-декодер";

MessageBoxButtons buttons = MessageBoxButtons.OK;

MessageBox.Show(null, message, caption, buttons, MessageBoxIcon.Exclamation, MessageBoxDefaultButton.Button1);

}

}

/// <summary>

/// О программе

/// </summary>

private void оПрограммеToolStripMenuItem\_Click(object sender, EventArgs e)

{

AboutForm eAboutForm = new AboutForm();

eAboutForm.ShowDialog();

}

/// <summary>

/// Выход

/// </summary>

private void выходToolStripMenuItem\_Click(object sender, EventArgs e)

{

// Если форма обработки имеет ненулевой хендл и у нее установлен режим -

// идёт обработка и ей мешать нельзя!

if (

(this.eProcessForm != null)

&&

(this.eProcessForm.Mode != RSMode.None)

)

{

return;

}

Close();

}

/// <summary>

/// Обработчик закрытия формы

/// </summary>

private void MainForm\_FormClosing(object sender, FormClosingEventArgs e)

{

// Если форма обработки имеет ненулевой хендл - её нужно закрыть

if (this.eProcessForm != null)

{

// На обработчике Close() отработает остановка процесса

this.eProcessForm.Close();

}

}

/// <summary>

/// Обработчик загрузки формы

/// </summary>

private void MainForm\_Load(object sender, EventArgs e)

{

// Устанавливаем общее количество томов по-умолчанию - 256

allVolCountMacTrackBar.Value = 14;

// Устанавлиаем избыточность кодирования - 100%

redundancyMacTrackBar.Value = 19;

// Устанавливаем конфигурацию кодера

SetCoderConfig();

}

#endregion Private Operations

private void button1\_Click(object sender, EventArgs e)

{

try

{

// Открываем файл справки

System.Diagnostics.Process.Start("Help.html");

}

catch

{

string message = "Не можу відкрити файл допомоги \"Help.html\"!";

string caption = " RS Кодер-декодер";

MessageBoxButtons buttons = MessageBoxButtons.OK;

MessageBox.Show(null, message, caption, buttons, MessageBoxIcon.Exclamation, MessageBoxDefaultButton.Button1);

}

}

private void button2\_Click(object sender, EventArgs e)

{

AboutForm eAboutForm = new AboutForm();

eAboutForm.ShowDialog();

}

private void button4\_Click(object sender, EventArgs e)

{

// Если форма обработки имеет ненулевой хендл и у нее установлен режим -

// идёт обработка и ей мешать нельзя!

if (

(this.eProcessForm != null)

&&

(this.eProcessForm.Mode != RSMode.None)

)

{

return;

}

Close();

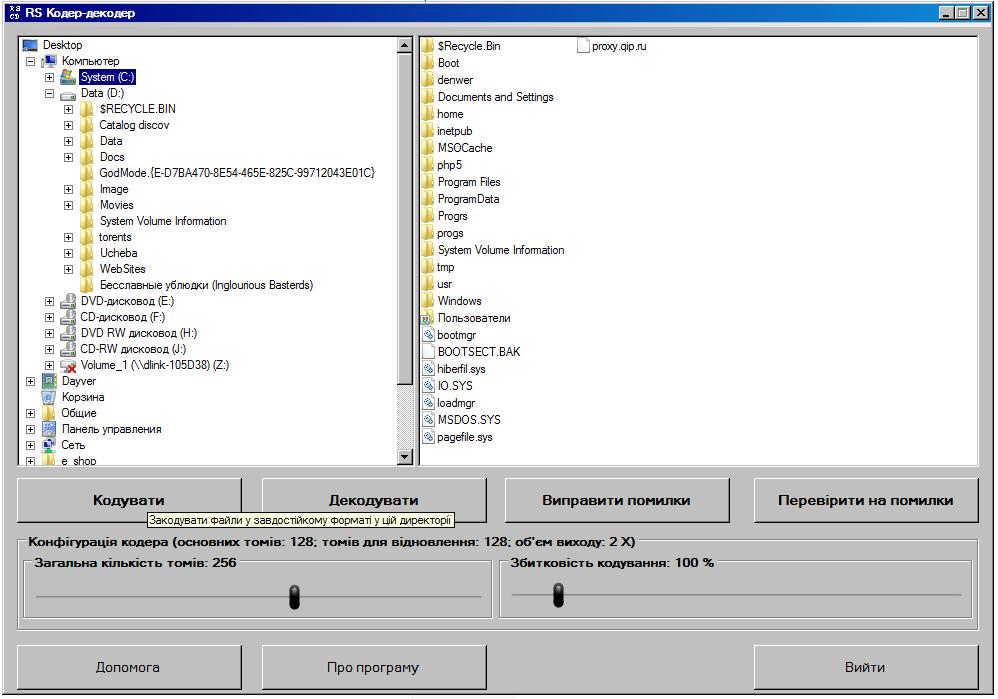
}

}

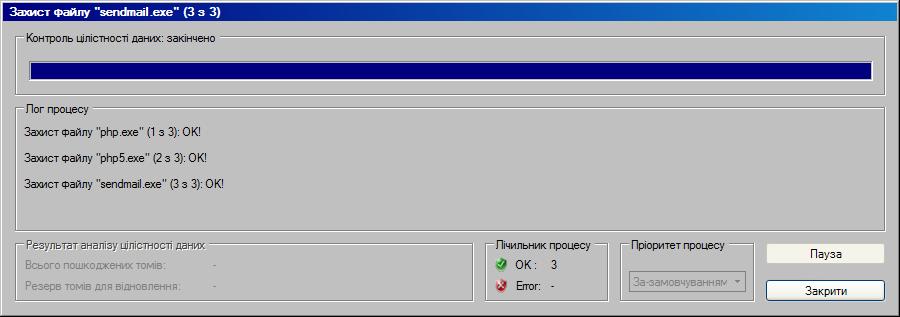
}

# Додаток Г

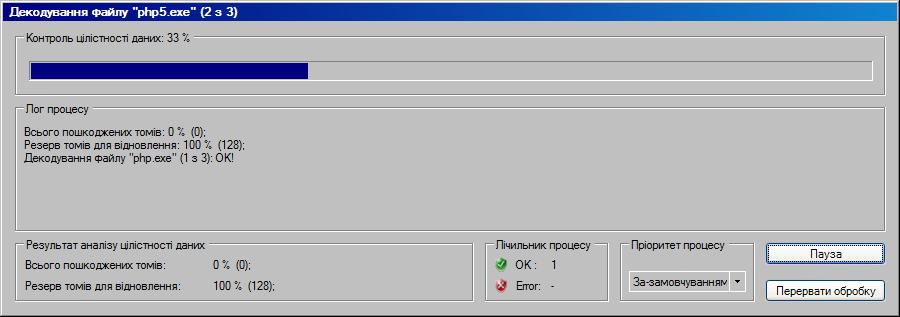
Інтерфейс програми «RS Кодер-декодер»



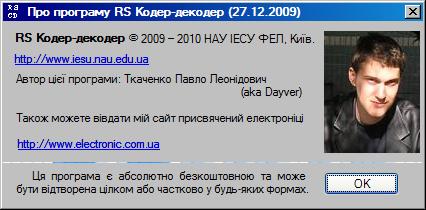
Головне вікно програми «RS Кодер-декодер»



Вікно програми «RS Кодер-декодер» в режимі кодування



Вікно програми «RS Кодер-декодер» в режимі декодування



Вікно програми «RS Кодер-декодер» з інформацією про автора

# Додаток Д

Лістинг програми «RSB 32 2.0»

Подпрограмма RSB-05 rsb

#include "stdafx.h"

#include "rsb32.h"

#include "rsb32\_private.h"

#include "rsb\_stream.h"

//#define \_LOG

namespace RSB {

static unsigned char global\_b\_values[128] = {0};

#ifdef \_LOG

std::string toString (char \*ptr)

{

return std::string (ptr, 4);

}

#endif

void rsb32\_init ()

{

//setlocale (LC\_ALL, ".ACP");

}

void rsb32\_init\_common\_key(unsigned char \*common\_key, const CryptContext &context)

{

long key\_size = context.keySize ();

char \*key\_buffer = context.keyBuffer ();

std::copy(key\_buffer, key\_buffer + key\_size, reinterpret\_cast<char \*>(common\_key));

}

void rsb32\_process\_file (int argc, \_TCHAR\* argv[])

{

CryptContext context (argc, argv);

if (context.operationType() == CryptContext::CRYPT) {

rsb32\_crypt\_file(context);

} else {

rsb32\_decrypt\_file(context);

}

}

void rsb32\_crypt (unsigned char \*buffer, unsigned char \*round\_key, bool direction)

{

unsigned char next\_key[4];

unsigned char cycles = (round\_key[3] | 0x01);

rsb32\_cyclic\_shift\_left (buffer, cycles);

if (!direction) {

rsb32\_slide\_right\_direct(buffer, round\_key, next\_key);

} else {

rsb32\_slide\_left\_direct (buffer, round\_key, next\_key);

}

rsb32\_nonlinear\_direct (buffer, round\_key[2], round\_key[1]);

rsb32\_permutation (buffer, round\_key[0]);

//copy newly generated key to round key buffer

std::copy (next\_key, next\_key + 4, round\_key);

}

void rsb32\_decrypt (unsigned char \*buffer, unsigned char \*round\_key, bool direction)

{

unsigned char next\_key[4];

rsb32\_permutation (buffer, round\_key[0]);

rsb32\_nonlinear\_inverse (buffer, round\_key[2], round\_key[1]);

if (!direction) {

rsb32\_slide\_right\_inverse(buffer, round\_key, next\_key);

} else {

rsb32\_slide\_left\_inverse(buffer, round\_key, next\_key);

}

unsigned char cycles = (round\_key[3] | 0x01);

rsb32\_cyclic\_shift\_right (buffer, cycles);

//copy newly generated key to round key buffer

std::copy (next\_key, next\_key + 4, round\_key);

}

void rsb32\_round\_crypt (rsbstream &stream, unsigned char \*key, bool direction)

{

unsigned char round\_key[4];

unsigned char buffer[32] = {0};

stream.set\_direction (direction);

//copy key to temporary buffer, because it is changed in crypt operation

std::copy(key, key + 4, round\_key);

while (stream.read (reinterpret\_cast <char \*>(buffer))) {

#ifdef \_LOG

std::cout << "\nrsb32\_round\_crypt: direction: " << direction << " round key:";

for (long i = 0; i < 4; i++) std::cout << (int) round\_key[i] << " ";

std::cout << " \n \*\*\* in buffer: ";

for (long i = 0; i < 32; i++) std::cout << (int) buffer[i] << " ";

#endif

rsb32\_crypt (buffer, round\_key, direction);

#ifdef \_LOG

std::cout << " \n \*\*\* out buffer: ";

for (long i = 0; i < 32; i++) std::cout << (int) buffer[i] << " ";

#endif

stream.write (reinterpret\_cast <char \*>(buffer));

}

}

void rsb32\_round\_decrypt (rsbstream &stream, unsigned char \*key, bool direction)

{

unsigned char round\_key[4];

unsigned char buffer[32] = {0};

stream.set\_direction (direction);

//copy key to temporary buffer, because it is changed in crypt operation

std::copy(key, key + 4, round\_key);

while (stream.read (reinterpret\_cast <char \*>(buffer))) {

#ifdef \_LOG

std::cout << "\nrsb32\_round\_decrypt: direction: " << direction << " round key:";

for (long i = 0; i < 4; i++) std::cout << (int) round\_key[i] << " ";

std::cout << " \n \*\*\* in buffer: ";

for (long i = 0; i < 32; i++) std::cout << (int) buffer[i] << " ";

#endif

rsb32\_decrypt (buffer, round\_key, direction);

#ifdef \_LOG

std::cout << " \n \*\*\* out buffer: ";

for (long i = 0; i < 32; i++) std::cout << (int) buffer[i] << " ";

#endif

stream.write (reinterpret\_cast <char \*>(buffer));

}

}

//returns true for lefthanded operation, false overwise

bool direction (long rounds)

{

return (rounds % 2 == 0 ? false : true);

}

void rsb32\_step\_crypt(rsbstream &stream, unsigned char \*common\_key, const CryptContext &context)

{

unsigned char \*key = 0; //common\_key;

for (long rounds = 0; rounds < context.roundsNumber(); rounds++)

{

key = common\_key + rounds \* 4;

#ifdef \_LOG

std::cout << "\nrsb32\_step\_crypt: rounds:" << rounds << " key:" << toString ((char \*)key) << std::endl;

#endif

rsb32\_round\_crypt (stream, key, direction (rounds));

}

}

void rsb32\_step\_decrypt(rsbstream &stream, unsigned char \*common\_key, const CryptContext &context)

{

unsigned char \*key = 0; //common\_key; //+ 4 \* (context.roundsNumber() - 1);

for (long rounds = context.roundsNumber() - 1; rounds >= 0; rounds--)

{

key = common\_key + rounds \* 4;

#ifdef \_LOG

std::cout << "\nrsb32\_step\_decrypt: rounds:" << rounds << " key:" << toString ((char \*)key) << std::endl;

#endif

rsb32\_round\_decrypt (stream, key, direction (rounds));

}

}

unsigned char \_transform\_key\_direct (unsigned char \*key, const CryptContext &context, long step\_number)

{

unsigned char b\_value = calculate\_b\_value (key, context.keySize ());

unsigned char shift\_value = (((b\_value & 0xF0) >> 4) | 0x01);

rsb32\_shift\_key\_left(key, shift\_value, context.keySize ());

rsb32\_nonlinear\_direct (key);

return b\_value;

}

unsigned char \_transform\_key\_inverse (unsigned char \*key, const CryptContext &context, long step\_number)

{

rsb32\_nonlinear\_inverse (key);

unsigned char b\_value = global\_b\_values[context.stepsNumber () - step\_number - 1];

unsigned char shift\_value = (((b\_value & 0xF0) >> 4) | 0x01);

rsb32\_shift\_key\_right(key, shift\_value, context.keySize ());

return b\_value;

}

void rsb32\_crypt\_step\_transform\_key (unsigned char \*key, const CryptContext &context, long step\_number)

{

\_transform\_key\_direct (key, context, step\_number);

}

void rsb32\_decrypt\_step\_transform\_key (unsigned char \*key, const CryptContext &context, long step\_number)

{

\_transform\_key\_inverse (key, context, step\_number);

}

void rsb32\_crypt\_init\_key (unsigned char \*key, const CryptContext &context)

{

//nothing to do..

}

void rsb32\_decrypt\_init\_key (unsigned char \*key, const CryptContext &context)

{

//global\_b\_values is 128 size, it is much more than enought for practial usage

if (context.stepsNumber () > 128) {

throw wruntime\_error (\_T("steps number > 128"));

}

//compute all b\_values

for (long steps = 0; steps < context.stepsNumber(); steps++) {

global\_b\_values[steps] = \_transform\_key\_direct (key, context, steps);

}

}

void rsb32\_crypt\_file(rsbstream &stream, const CryptContext &context)

{

unsigned char \*common\_key = new unsigned char [context.keySize ()];

rsb32\_init\_common\_key(common\_key, context);

rsb32\_crypt\_init\_key (common\_key, context);

for (long steps = 0; steps < context.stepsNumber (); ++steps)

{

rsb32\_crypt\_step\_transform\_key (common\_key, context, steps);

rsb32\_step\_crypt (stream, common\_key, context);

}

delete common\_key;

}

void rsb32\_decrypt\_file(rsbstream &stream, const CryptContext &context)

{

unsigned char \*common\_key = new unsigned char [context.keySize ()];

rsb32\_init\_common\_key (common\_key, context);

rsb32\_decrypt\_init\_key (common\_key, context);

for (long steps = 0; steps < context.stepsNumber (); ++steps)

{

rsb32\_step\_decrypt (stream, common\_key, context);

rsb32\_decrypt\_step\_transform\_key (common\_key, context, steps);

}

delete common\_key;

}

void rsb32\_crypt\_file (const CryptContext &context)

{

if (context.stepsNumber() < 1) {

throw truntime\_error(\_T("steps number < 1\n"));

}

std::ifstream in\_file (context.inputFilename().c\_str(), std::ios::binary | std::ios::ate);

if (!in\_file) {

std::wstring message = \_T("File could not be opened: ");

message += context.inputFilename();

throw truntime\_error (message.c\_str ());

}

std::streamoff file\_size = in\_file.tellg ();

std::ofstream out\_file (context.outputFilename().c\_str(), std::ios::binary);

if (!out\_file) {

std::wstring message = \_T("File could not be opened: ");

message += context.outputFilename();

throw truntime\_error (message.c\_str ());

}

//create extended file

file\_size = create\_extended\_file (in\_file, out\_file, file\_size);

in\_file.close();

out\_file.close ();

std::fstream stream (context.outputFilename().c\_str(), RSB32\_FMODE);

if (!stream) {

throw truntime\_error (\_T("could not create output stream!\n"));

}

rsbstream rstream (stream, file\_size);

rsb32\_crypt\_file (rstream, context);

rstream.close ();

}

void rsb32\_decrypt\_file (const CryptContext &context)

{

if (context.stepsNumber() < 1) {

throw truntime\_error(\_T("steps number < 1\n"));

}

std::ifstream in\_file (context.inputFilename().c\_str(), std::ios::binary | std::ios::ate);

if (!in\_file) {

std::wstring message = \_T("File could not be opened: ");

message += context.inputFilename();

throw truntime\_error (message.c\_str ());

}

std::streamoff file\_size = in\_file.tellg ();

std::ofstream out\_file (context.outputFilename().c\_str(), std::ios::binary);

if (!out\_file) {

std::wstring message = \_T("File could not be opened: ");

message += context.outputFilename();

throw truntime\_error (message.c\_str ());

}

//create extended file

file\_size = create\_extended\_file (in\_file, out\_file, file\_size);

in\_file.close();

out\_file.close ();

std::fstream stream (context.outputFilename().c\_str(), RSB32\_FMODE);

if (!stream) {

throw truntime\_error (\_T("could not create output stream!\n"));

}

rsbstream rstream (stream, file\_size);

rsb32\_decrypt\_file (rstream, context);

rstream.close ();

}

}

Подпрограмма RSB-14 rsb32gui

// rsb32gui.cpp : Defines the class behaviors for the application.

//

#include "stdafx.h"

#include "rsb32gui.h"

#include "rsb32guiDlg.h"

#ifdef \_DEBUG

#define new DEBUG\_NEW

#endif

// Crsb32guiApp

BEGIN\_MESSAGE\_MAP(Crsb32guiApp, CWinApp)

ON\_COMMAND(ID\_HELP, CWinApp::OnHelp)

END\_MESSAGE\_MAP()

// Crsb32guiApp construction

rsb32guiApp::Crsb32guiApp()

{

// TODO: add construction code here,

// Place all significant initialization in InitInstance

}

// The one and only Crsb32guiApp object

Crsb32guiApp theApp;

// Crsb32guiApp initialization

BOOL Crsb32guiApp::InitInstance()

{

// InitCommonControls() is required on Windows XP if an application

// manifest specifies use of ComCtl32.dll version 6 or later to enable

// visual styles. Otherwise, any window creation will fail.

InitCommonControls();

CWinApp::InitInstance();

// Standard initialization

// If you are not using these features and wish to reduce the size

// of your final executable, you should remove from the following

// the specific initialization routines you do not need

// Change the registry key under which our settings are stored

// TODO: You should modify this string to be something appropriate

// such as the name of your company or organization

SetRegistryKey(\_T("Local AppWizard-Generated Applications"));

Crsb32guiDlg dlg;

m\_pMainWnd = &dlg;

INT\_PTR nResponse = dlg.DoModal();

if (nResponse == IDOK)

{

// TODO: Place code here to handle when the dialog is

// dismissed with OK

}

else if (nResponse == IDCANCEL)

{

// TODO: Place code here to handle when the dialog is

// dismissed with Cancel

}

// Since the dialog has been closed, return FALSE so that we exit the

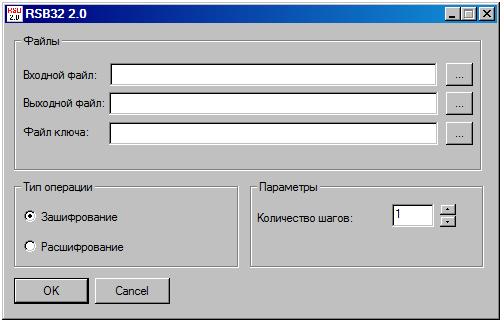
// application, rather than start the application's message pump.

return FALSE;

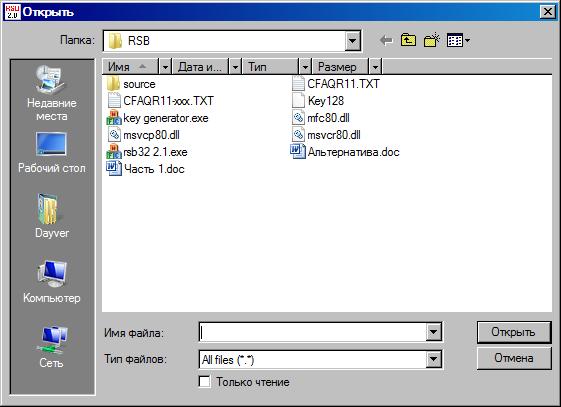
}

# Додаток Е

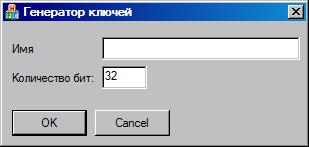
Інтерфейс програми «RSB 32 2.0»



Головне вікно програми «RSB 32 2.0»



Вікно програми «RSB 32 2.0» для вибору файлу



Вікно програми для генерації ключів

1. Том- це файл. Пристрій містить безліч файлів, але на рівні «сирих» даних, тобто елементарних бітів (при розгляді на найнижчому, тобто самому близькому до апаратних засобів рівні) будь-який пристрій може бути представлено як завгодно великим файлом. Оскільки алгоритми корекції помилок працюють на низькому рівні, то межі між сутностями пристрій і том стираються, однак, для користувача зручніше і зрозуміліше оперувати з файлами, ніж з пристроями. [↑](#footnote-ref-1)
2. Операція «складання за модулем два» аналогічна операції додавання в полі Галуа, що застосовується в складних RAID - подібних системах, тому що XOR передбачає відкидання інформації про перенесення даних в старший розряд при складанні двох бітів. Таким чином, значення елемента завжди обмежується одиницею. Значення елемента в полі Галуа також обмежена. [↑](#footnote-ref-2)
3. Під вектор-стовцем зрізу даних множини томів розуміється послідовність байтів (по одному з файлу кожного тому, по одному й тому ж зміщенню у всіх файлах). [↑](#footnote-ref-3)