МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ „ЛЬВІВСЬКА ПОЛІТЕХНІКА”

ПРОГРАМНА РЕАЛІЗАЦІЯ ОСНОВНИХ СЕРВІСІВ БЕЗПЕКИ

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт
з дисциплін "Програмне забезпечення захисту інформації",
"Захист програм та даних"
для студентів базових напрямів 6.050101 "Комп'ютерні науки",
6.050103 "Програмна інженерія"

Затверджено
на засіданні кафедри
програмного забезпечення
Протокол № 8 від 24.04.2008 р.

Львів – 2008

Програмна реалізація основних сервісів безпеки: Методичні вказівки до виконання лабораторних робіт з дисциплін "Програмне забезпечення захисту інформації", "Захист програм та даних" для студентів базових напрямів "Комп'ютерні науки" та "Програмна інженерія" / Укл.: В.С. Яковина – Львів: Видавництво Національного університету "Львівська політехніка", 2008. – 47 с.

Укладач Яковина В.С., канд. фіз.-мат. наук, доц.

Відповідальний за випуск Федасюк Д.В., д-р тех. наук, проф.

Рецензенти Білас О.Є., канд. тех. наук, доц.
Пелешко Д.Д., канд. тех. наук, доц.

Лабораторна робота № 1
СТВОРЕННЯ ГЕНЕРАТОРА ПСЕВДОВИПАДКОВИХ ЧИСЕЛ

Мета роботи: ознайомитись з джерелами та застосуванням випадкових чисел, алгоритмами генерування псевдовипадкових чисел та навчитись створювати програмні генератори псевдовипадкових чисел для використання в системах захисту інформації.

Теоретичні відомості

Сучасна інформатика широко використовує випадкові числа в різних програмах – від методу Монте-Карло до криптографії. Ряд алгоритмів захисту мережі, заснованих на засобах криптографії, передбачає використання випадкових чисел. Ці застосування висувають дві вимоги до послідовності випадкових чисел: випадковість і непередбачуваність.

Джерелами дійсно випадкових чисел потенційно можуть бути фізичні генератори шумів, такі як імпульсні детектори іонізуючого випромінювання, газорозрядні лампи, конденсатори з втратами струму тощо. Однак такі пристрої можуть знайти доволі обмежене застосування в додатках для захисту інформації. Туту існують проблеми як з випадковістю, так і з точністю отриманих таким методом чисел, не кажучи вже про проблеми підключення такого роду пристроїв до кожної системи в мережі.

Тому криптографічні додатки зазвичай використовують алгоритмічні методи генерування випадкових чисел. Відповідні алгоритми є детермінованими і тому породжують послідовності чисел, які не є статистично випадковими. Однак, якщо алгоритм є достатньо хорошим, породжувані ним послідовності чисел витримують багато тестів на випадковість. Такі числа часто називають псевдовипадковими.

Генератор псевдовипадкових чисел – алгоритм, що генерує послідовність чисел, елементи якої незалежні один від одного і підлягають заданому розподілу.

Найбільш популярним алгоритмом для генерування псевдовипадкових чисел є алгоритм, запропонований Лемером, який називається методом лінійного порівняння. Цей алгоритм має чотири наступних параметри.

m	модуль порівняння	$m > 0$
a	множник	$0 \le a < m$
c	приріст	$0 \le c < m$
X_0	початкове число	$0 \le X_0 < m$

Послідовність псевдовипадкових чисел $\{X_0\}$ отримують за допомогою ітерацій наступного співвідношення:

$$X_{n+1} = (aX_n + c) \bmod m.$$

При цьому якщо m, a, c та X_0 є цілими, то буде отримано послідовність цілих чисел з діапазону $0 \le X_n < m$.

Вибір значень для a, c та m є дуже важливим з точки зору створення хорошого генератора псевдовипадкових чисел. Розглянемо, наприклад, випадок $a = c = 1$. Породжена при цьому послідовність очевидно не буде задовільною. Тепер розглянемо значення $a = 7$, $c = 0$, $m = 32$ $X_0 = 1$. В цьому випадку генерується послідовність {7, 17, 23, 1, 7, …}, яка також очевидно не буде задовільною. З 32 можливих значень тут виявляються задіяними тільки 4 (в такому випадку кажуть, що послідовність має період 4). Якщо ж, залишивши інші значення тими самими, змінити значення a і прийняти a=5, то результуючою послідовністю буде {1, 5, 25, 29, 17, 21, 9, 13, 1, …} і її період вже буде рівним 8.

Бажано, щоб m було дуже великим, щоб потенційно могли генеруватись дуже довгі серії різних псевдовипадкових чисел. Загальним правилом тут є вибір значення m, близького до максимально допустимого для даного комп'ютера додатного цілого числа. Тому доволі часто значення m вибирається рівним або майже рівним значенню 2^{31}.

Пропонується три критерії, за якими можна оцінити якість будь-якого генератора псевдовипадкових чисел.

1) Функція генерації повинна бути функцією повного періоду, тобто функція повинна породити усі числа від 0 до m перед тим, як числа почнуть повторюватись.
2) Створена послідовність повинна вести себе як випадкова. Насправді ця послідовність не буде випадковою, оскільки генерується детермінованим алгоритмом, але існує багато статистичних тестів, які можна використовувати для того, щоб оцінити ступінь випадковості поведінки послідовності.
3) Функція генерації повинна ефективно реалізовуватись в рамках 32-бітної арифметики.

Усі ці три критерії можуть задовольнятись при адекватному виборі значень a, c та m. Відносно першого критерію можна довести, що якщо m є простим і $c = 0$, то для певних значень a період генерованої функцією послідовності виявляється рівним m-1 і в цій послідовності буде відсутнім тільки значення 0. В 32-бітній арифметиці зручним простим значенням для m є значення 2^{31}-1. В цьому випадку функція генерації приймає вигляд

$$X_{n+1} = (aX_n) \bmod (2^{31} - 1).$$

З більш ніж двох мільйонів можливих значень a тільки декілька множників відповідають функції, що витримує усі три тести. Одним з таких значень є $a = 7^5 = 16807$, яке було знайдено і використано для родини комп'ютерів IBM 360. Відповідний генератор знайшов широке застосування, і тому він був підданий більш ретельному аналізу, ніж будь-який інший генератор псевдовипадкових чисел. Він нерідко рекомендується для статистичного та імітаційного моделювання різноманітних процесів.

Перевагою алгоритму лінійного порівняння є те, що якщо вибрати адекватні множник та модуль порівняння, то створювана послідовність чисел виявляється статистично невідрізнимою від послідовності чисел, що вибираються випадково (але незворотно) з множини чисел 1, 2, …, m-1. Однак в самому алгоритмі немає нічого випадкового взагалі, крім вибору початкового значення X_0. Якщо це значення вибране, інші числа послідовності визначаються ним однозначно. Це виявляється дуже важливим з точки зору криптоаналізу.

Якщо противник знає, що використовується алгоритм лінійного порівняння і якщо до того ж йому відомі параметри алгоритму (наприклад, $a = 7^5$, $c = 0$, $m = 2^{31}$-1), то, відкривши усього одно число, противник може отримати всі наступні. Але якщо навіть опонент знає тільки те, що вибрано алгоритм лінійного порівняння, знання невеликої частини послідовності вже достатньо для того, щоб визначити усі параметри алгоритму. Припустимо, наприклад, що противник зможе визначити значення для X_0, X_1, X_2 та X_3. Тоді

$$X_1 = (aX_0 + c) \bmod m,$$
$$X_2 = (aX_1 + c) \bmod m,$$
$$X_3 = (aX_2 + c) \bmod m.$$

Ці рівняння можуть бути розв'язані відносно a, c та m.

Отже, хоча і зручно використовувати хороший генератор псевдовипадкових чисел, бажано подбати про те, що генерована послідовність була дійсно невідтворюваною, щоб знання частини послідовності не давало опоненту можливості визначити наступні елементи послідовності. Ця мета може бути досягнута цілим рядом способів. Наприклад можна змінювати потік псевдовипадкових чисел, використовуючи для цього системний час. Один зі способів на основі системного годинника полягає в ініціалізації нової послідовності після отримання кожних N чисел, використовуючи для початкового числа поточне значення часу (mod m). А можна просто додавати до кожного псевдовипадкового числа поточне значення часу (mod m).

Завдання до виконання роботи

Згідно до варіанту, наведеного в таблиці, створити програмну реалізацію генератора псевдовипадкових чисел за алгоритмом лінійного порівняння. Програма повинна генерувати послідовність із заданої при вводі кількості псевдовипадкових чисел, результати повинні як виводитись на екран, так і зберігатись у файл. Перевірити період функції генерації, зробити висновок про адекватність вибору параметрів алгоритму. У звіті навести протокол роботи програми, значення періоду функції генерації та зробити висновок про придатність цього генератора для задач криптографії.

Варіанти для виконання лабораторної роботи:

№ варіанту	Модуль порівняння, m	Множник, a	Приріст, c	Початкове значення, X_0
1.	2^{10}-1	2^5	0	2
2.	2^{11}-1	3^5	1	4
3.	2^{12}-1	4^5	2	8
4.	2^{13}-1	5^5	3	16
5.	2^{14}-1	6^5	5	32
6.	2^{15}-1	2^3	8	64
7.	2^{16}-1	3^3	13	128
8.	2^{17}-1	4^3	21	256
9.	2^{18}-1	5^3	34	512
10.	2^{19}-1	6^3	55	1024
11.	2^{20}-1	7^3	89	1
12.	2^{21}-1	8^3	144	3
13.	2^{22}-1	9^3	233	5
14.	2^{23}-1	10^3	377	7
15.	2^{24}-1	11^3	610	9
16.	2^{25}-1	12^3	987	11
17.	2^{26}-1	13^3	1597	13
18.	2^{27}-1	14^3	2584	17
19.	2^{28}-1	15^3	4181	19
20.	2^{29}-1	16^3	6765	23
21.	2^{30}-1	17^3	10946	29
22.	2^{31}-1	7^5	17711	31
23.	2^{31}	2^{16}	28657	33
24.	2^{31}-3	2^{15}	46368	37
25.	2^{31}-7	2^{14}	75025	41

Контрольні запитання

1. Назвіть вимоги до послідовності випадкових чисел.
2. Що таке генератор псевдовипадкових чисел?
3. Що таке період послідовності псевдовипадкових чисел?
4. Назвіть критерії оцінки якості генератора псевдовипадкових чисел.
5. Якими повинні бути параметри алгоритму лінійного порівняння для того, щоб генерувався повний період псевдовипадкових чисел?
6. Назвіть переваги алгоритму лінійного порівняння.
7. Назвіть недоліки алгоритму лінійного порівняння.

Лабораторна робота № 2
СТВОРЕННЯ ПРОГРАМНОГО ЗАСОБУ ДЛЯ ЗАБЕЗПЕЧЕННЯ ЦІЛІСНОСТІ ІНФОРМАЦІЇ

Мета роботи: ознайомитись з методами криптографічного забезпечення цілісності інформації, навчитись створювати програмні засоби для забезпечення цілісності інформації з використанням алгоритмів хешування.

Теоретичні відомості

Хеш функція – це функція, що відображає вхідне слово скінченної довжини у скінченному алфавіті в слово заданої, зазвичай фіксованої довжини. Таким чином, функція хешування отримує на вхід повідомлення M довільної довжини, а на вихід видає хеш-код $H(M)$ фіксованого розміру, який іноді називають профілем повідомлення. Хеш-код є функцією усіх бітів повідомлення і забезпечує можливість контролю помилок: зміна будь-якої кількості бітів повідомлення призводить до зміни хеш-коду.

Основні області використання хеш функцій – аутентифікація інформації та цифровий підпис. Практичне використання функцій хешування накладає на них ряд вимог, наведених нижче:

1. Хеш-функція H повинна застосовуватися до блоку даних будь-якої довжини.
2. Хеш-функція H створює вихід фіксованої довжини.
3. $H(M)$ відносно легко (за поліноміальний час) обчислюється для будь-якого значення M, а алгоритм обчислення повинен бути практичним з погляду як апаратної, так і програмної реалізації.
4. Для будь-якого даного значення хеш-коду h обчислювально неможливо знайти M таке, що $H(M)=h$. Таку властивість іноді називають **односторонністю**.
5. Для будь-якого даного блоку x обчислювально неможливо знайти $y \neq x$, для якого $H(x)=H(y)$. Таку властивість іноді називають **слабкою опірністю колізіям**.
6. Обчислювально неможливо знайти довільну пару різних значень x та y, для яких $H(x)=H(y)$. Таку властивість іноді називають **сильною опірністю колізіям**.

Перші три властивості описують вимоги, що забезпечують можливість практичного застосування функції хешування для аутентифікації повідомлень.

Четверту властивість визначає вимога односторонності хеш-функції: легко створити хеш-код за даним повідомленням, але неможливо відновити повідомлення за даним хеш-кодом.

П'ята властивість гарантує, що неможливо знайти інше повідомлення, значення хеш-функції якого збігалося б зі значенням хеш-функції даного повідомлення. Це запобігає підробці аутентифікатора при використанні зашифрованого хеш-коду.

Шоста властивість захищає проти класу атак, побудованих на парадоксі задачі про дні народження. Вона унеможливлює знаходження двох довільних різних повідомлень з однаковим хешем.

Алгоритм хешування MD5

Алгоритм MD5 обчислення профілю повідомлення (описаний в RFC 1321) був розроблений Роном Райвестом (Ron Rivest) з Масачусетського технологічного інституту (MIT) на початку 90-х років. До недавнього часу MD5 залишався найбільш поширеним із захищених алгоритмів хешування і використовувався як в найпопулярніших операційних системах Windows, Linux, BSD, так і в системах захисту інформації, таких як PGP. На даний час основне використання цього алгоритму – криптографічні контрольні суми, наприклад для перевірки цілісності отриманих з мережі Інтернет файлів.

Алгоритм одержує на вході повідомлення довільної довжини (довжина в бітах може дорівнювати нулю, вона не обов'язково повинна бути кратною восьми) й створює в якості виходу хеш повідомлення довжиною 128 біт. Алгоритм складається з наступних кроків:

Крок 1: додавання доповнення

Повідомлення доповнюється таким чином, щоб його довжина стала рівною 448 по модулю 512 (довжина 448 mod 512). Це означає, що довжина доданого повідомлення на 64 біта менша, ніж число, кратне 512. Додавання здійснюється завжди, навіть якщо повідомлення має потрібну довжину. Наприклад, якщо довжина повідомлення 448 бітів, воно доповнюється 512 бітами до 960 бітів. Таким чином додається від 1 до 512 бітів.

Структура бітів доповнення наступна: перший біт дорівнює 1, а усі решта дорівнюють 0.

Крок 2: додавання значення довжини

64-бітне значення довжини вихідного (до доповнення) повідомлення в бітах приєднується до результату першого кроку. Якщо початкова довжина більша, ніж 2^{64}, то використовуються тільки молодші 64 біта. Таким чином, поле містить довжину вихідного повідомлення по модулю 2^{64}. Структура розширеного повідомлення схематично зображена на рис. 1.

Повідомлення	Доповнення (від 1 до 512 бітів)	Довжина вихідного повідомлення

Рис. 1. Структура розширеного повідомлення

У результаті перших двох кроків створюється повідомлення, довжина якого кратна 512 бітам. Це розширене повідомлення представляється як послідовність 512-бітних блоків $Y_0, Y_1, \ldots, Y_{L-1}$ (рис. 2), при цьому загальна довжина розширеного повідомлення дорівнює $(L \cdot 512)$ бітів. Таким чином, довжина отриманого розширеного повідомлення кратна довжині блоку з шістнадцяти 32-бітних слів.

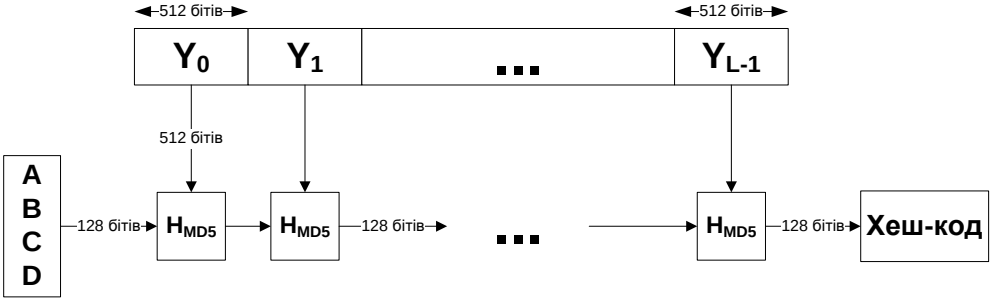

Рис. 2. Загальна схема виконання алгоритму MD5

Крок 3: ініціалізація MD-буфера

Для зберігання проміжних і кінцевих результатів функції хешування використовується 128-бітний буфер. Буфер може бути представлений як чотири 32-бітних регістри (A, B, C, D). Ці регістри ініціалізуються наступними 32-бітовими цілими значеннями (в шістнадцятковому представленні):

A = 0x67452301

B = 0xEFCDAB89

C = 0x98BADCFE

D = 0x10325476

Ці значення зберігаються у форматі прямого порядку запису байтів, коли найменш значущий байт слова знаходиться в позиції з молодшою адресою. Як 32-бітові рядки, значення ініціалізації (в шістнадцятковому запису) матимуть вигляд:

слово A: 01 23 45 67,

слово B: 89 AB CD EF,

слово C: FE DC BA 98,

слово D: 76 54 32 10.

Крок 4: обробка повідомлення блоками по 512 бітів (16-слів)

Основою алгоритму є функція стиснення – модуль, що складається із чотирьох циклічних обробок, позначений на рис. 2 як H_{MD5}, а логіка роботи модуля показана на рис. 3. Чотири цикли мають схожу структуру, але кожний цикл використовує свою елементарну логічну функцію, позначену f_F, f_G, f_H, f_I відповідно.

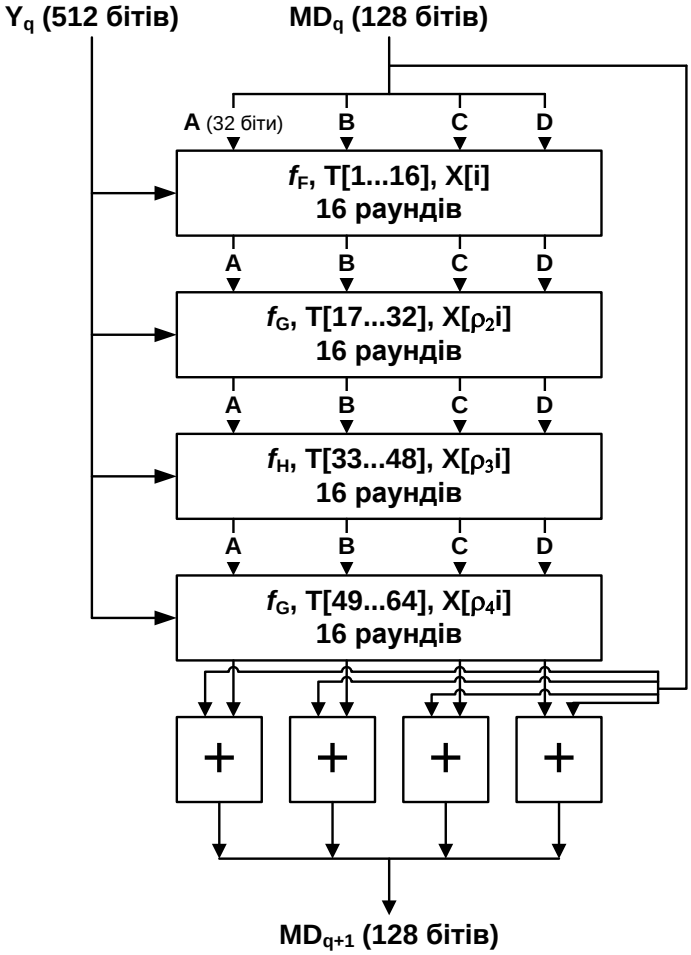

Рис. 3. Обробка чергового 512-бітового блоку

Кожен цикл приймає в якості входу поточний 512-бітовий блок Y_q і 128-бітове значення буфера ABCD, в якому міститься проміжне значення профілю, і змінює вміст цього буфера. Кожний цикл також використовує четверту частину 64-елементної таблиці $T[1…64]$, складеної зі значень функції синуса. У таблиці T i-ий елемент, позначений $T[i]$, має значення, що дорівнює цілій частині числа $2^{32}\cdot|\sin(i)|$, де i відповідає значенню кута в радіанах. Оскільки $|\sin(i)|$ лежить у діапазоні між 0 і 1, кожен елемент T є цілим числом, яке може бути представлено 32 бітами. Таблиця забезпечує "випадкову" множину 32-

бітних значень, які повинні ліквідувати будь-яку регулярність у вхідних даних. Список значень *T* наведено в табл. 1.

Вихід четвертого циклу додається по модулю 2^{32} до входу першого циклу (MD_q), в результаті чого отримують MD_{q+1}. Додавання виконується незалежно для кожного з чотирьох слів у буфері.

Таблиця 1. Значення *T*, створені на основі функції синуса

i	***T*[*i*]**	***i***	***T*[*i*]**	***i***	***T*[*i*]**	***i***	***T*[*i*]**
1.	0xd76aa478	17.	0xf61e2562	33.	0xfffa3942	49.	0xf4292244
2.	0xe8c7b756	18.	0xc040b340	34.	0x8771f681	50.	0x432aff97
3.	0x242070db	19.	0x265e5a51	35.	0x6d9d6122	51.	0xab9423a7
4.	0xc1bdceee	20.	0xe9b6c7aa	36.	0xfde5380c	52.	0xfc93a039
5.	0xf57c0faf	21.	0xd62f105d	37.	0xa4beea44	53.	0x655b59c3
6.	0x4787c62a	22.	0x2441453	38.	0x4bdecfa9	54.	0x8f0ccc92
7.	0xa8304613	23.	0xd8a1e681	39.	0xf6bb4b60	55.	0xffeff47d
8.	0xfd469501	24.	0xe7d3fbc8	40.	0xbebfbc70	56.	0x85845dd1
9.	0x698098d8	25.	0x21e1cde6	41.	0x289b7ec6	57.	0x6fa87e4f
10.	0x8b44f7af	26.	0xc33707d6	42.	0xeaa127fa	58.	0xfe2ce6e0
11.	0xffff5bb1	27.	0xf4d50d87	43.	0xd4ef3085	59.	0xa3014314
12.	0x895cd7be	28.	0x455a14ed	44.	0x4881d05	60.	0x4e0811a1
13.	0x6b901122	29.	0xa9e3e905	45.	0xd9d4d039	61.	0xf7537e82
14.	0xfd987193	30.	0xfcefa3f8	46.	0xe6db99e5	62.	0xbd3af235
15.	0xa679438e	31.	0x676f02d9	47.	0x1fa27cf8	63.	0x2ad7d2bb
16.	0x49b40821	32.	0x8d2a4c8a	48.	0xc4ac5665	64.	0xeb86d391

Крок 5: вихід

Після обробки всіх *L* 512-бітових блоків виходом *L*-ої стадії є 128-бітовий профіль повідомлення.

В цілому роботу алгоритму MD5 можна описати наступними формулами:

$MD_0 = IV$,

$MD_{q+1} = MD_q + f_I\left(Y_q, f_H\left(Y_q, f_G\left(Y_q, MD_q\right)\right)\right)$,

$MD = MD_L$,

де *IV* – початкове значення буфера ABCD, визначене на кроці 3,

Y_q – *q*-ий 512-бітовий блок повідомлення,

L – кількість блоків у повідомленні (включаючи біти доповнення і довжини),

MD – остаточне значення профілю повідомлення.

Функція стиснення MD5

Розглянемо більш детально логіку кожного із чотирьох циклів обробки одного 512-бітного блоку. Кожний цикл складається з 16 раундів, що оперують із буфером ABCD. Кожний раунд можна представити у вигляді:

$$A = B + CLS_S(A + f(B,C,D) + X[k] + T[i]),$$

де A, B, C, D – чотири слова буфера; після виконання кожного окремого кроку відбувається циклічний зсув вліво на одне слово,

f – одна з елементарних функцій f_F, f_G, f_H, f_I,

CLS_S – циклічний зсув вліво на S біт 32-бітового аргументу,

$X[k]$ – k-те 32-бітове слово в q-му 512-бітовому блоці повідомлення, тобто $M[q \cdot 16+k]$,

$T[i]$ – i-те 32-бітове слово в матриці T,

+ – додавання по модулю 2^{32}.

Структура операції раунду показана на рис. 4. Порядок, в якому використовуються чотири слова (A, B, C, D), забезпечує на рівні слів циклічний зсув вліво на одне слово на кожному раунді.

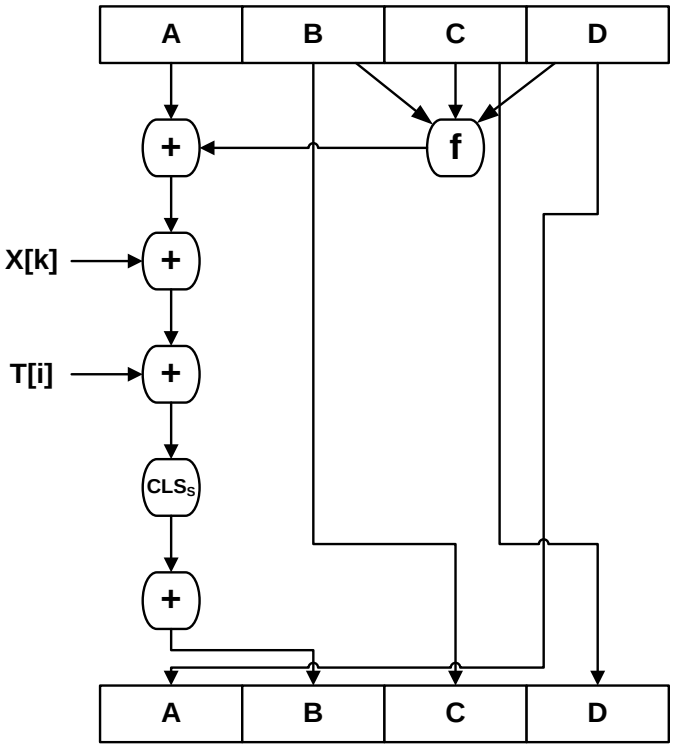

Рис. 4. Логіка виконання окремого раунду

На кожному із чотирьох циклів алгоритму використовується одна із чотирьох елементарних логічних функцій. Кожна елементарна функція одержує на вхід три 32-бітних слова, а на виході створює одне 32-бітне слово. Кожна функція виконує ряд побітових логічних операцій, тобто n-ий біт

виходу є функцією від *n*-их бітів трьох вхідних значень. Елементарні функції мають наступний вигляд:

$$f_F = (B \wedge C) \vee (\overline{B} \wedge D),$$
$$f_G = (B \wedge D) \vee (C \wedge \overline{D}),$$
$$f_H = B \oplus C \oplus D$$
$$f_I = C \oplus (B \vee \overline{D})$$

Тут бітові логічні операції AND, OR, NOT, XOR представлені символами $\wedge, \vee, \bar{}, \oplus$ відповідно.

На рис. 4 показано алгоритм виконання окремого раунду. Обробляється масив 32-бітових слів *X*[0…15], що містить значення поточного 512-бітного вхідного блоку. У рамках раунду кожне з 16 слів *X*[0…15] застосовується рівно один раз, для одного кроку, а порядок, у якому ці слова використаються, міняється від циклу до циклу. В першому циклі слова фігурують в їх вихідному порядку. Для циклів з 2-го по 4-ий визначаються наступні перестановки[1]:

$$\rho_2(i) = (1 + 5i) \bmod 16,$$
$$\rho_3(i) = (5 + 3i) \bmod 16,$$
$$\rho_4(i) = 7i \bmod 16.$$

Кожен з 64 елементів *T* довжиною в 32-бітове слово використовується тільки один раз, для одного раунду одного циклу. Зверніть увагу й на те, що в результаті виконання одного раунду оновлюється тільки 4 байти буфера ABCD. Отже, протягом циклу кожен байт буфера оновлюється чотири рази і ще один раз наприкінці, коли отримується кінцеве вихідне значення блоку. Нарешті зазначимо, що в кожному циклі використовуються різні послідовності чотирьох циклічних зсувів уліво з різними значеннями зсуву (послідовності значень зсуву S для кожного циклу наведено в табл. 2). Під час виконання кожного циклу використовується послідовність з чотирьох значень зсувів S_{ij}, яка повторюється чотири рази, фактично при обробці однакового слова буфера ABCD в одному циклі використовується однакове значення зсуву. Метою усіх цих маніпуляцій є забезпечення практичної неможливості відшукати колізії (пари 512-бітових блоків, що породжують однаковий вихід).

Таблиця 2. Значення циклічного зсуву S для кожного циклу

1-й цикл	**2-й цикл**	**3-й цикл**	**4-й цикл**
S_{11}=7	S_{21}=5	S_{31}=4	S_{41}=6
S_{12}=12	S_{22}=9	S_{32}=11	S_{42}=10
S_{13}=17	S_{23}=14	S_{33}=16	S_{43}=15

[1] Зверніть увагу, що *i*=0…15.

S_{14}=22	S_{24}=20	S_{34}=23	S_{44}=21

Завдання до виконання роботи

Створити програмну реалізацію алгоритму хешування MD5. Тестування створеної програми провести з використання наступних тестових значень хешу (згідно RFC 1321):

H() = D41D8CD98F00B204E9800998ECF8427E
H(a) = 0CC175B9C0F1B6A831C399E269772661
H(abc) = 900150983CD24FB0D6963F7D28E17F72
H(message digest) = F96B697D7CB7938D525A2F31AAF161D0
H(abcdefghijklmnopqrstuvwxyz) = C3FCD3D76192E4007DFB496CCA67E13B
H(ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789) = D174AB98D277D9F5A5611C2C9F419D9F
H(12345678901234567890123456789012345678901234567890123456789012345678901234567890) = 57EDF4A22BE3C955AC49DA2E2107B67A

Програмна реалізація повинна виводити значення хешу як для рядка, заданого в полі вводу, так і для файлу. Результат роботи програми повинен відображатись на екрані з можливістю наступного запису в файл. Крім того програма повинна мати можливість перевірити цілісність будь-якого файлу за наявним файлом з MD5 хешем, записаним у шістнадцятковому форматі. У звіті навести протокол тестування і роботи програми та зробити висновки.

Контрольні запитання

1. Що таке хеш функція?
2. Області використання хеш функцій.
3. Назвіть вимоги до криптографічних хеш функцій.
4. Що таке слабка опірність колізіям?
5. Основні області використання алгоритму MD5.
6. Яка довжина хеш-значення алгоритму MD5?
7. Яка довжина блоку, що обробляється алгоритмом MD5?

Лабораторна робота № 3
СТВОРЕННЯ ПРОГРАМНОГО ЗАСОБУ ДЛЯ ЗАБЕЗПЕЧЕННЯ КОНФІДЕНЦІЙНОСТІ ІНФОРМАЦІЇ

Мета роботи: ознайомитись з методами криптографічного забезпечення конфіденційності інформації, навчитись створювати комплексні програмні продукти для захисту інформації з використанням алгоритмів симетричного шифрування, хешування та генераторів псевдовипадкових чисел.

Теоретичні відомості.

RC5 – це алгоритм симетричного шифрування, розроблений Роном Райвестом в середині 90-х років. При розробці RC5 ставилась задача досягнення наступних характеристик.

- **Придатність для апаратної та програмної реалізації.** В RC5 використовуються тільки елементарні обчислювальні операції, які зазвичай застосовуються в мікропроцесорах.
- **Швидкість виконання.** RC5 є простим алгоритмом, що працює з даними розміром в машинне слово. Усі основні операції передбачають також роботу з даними довжиною в слово.
- **Адаптованість до процесорів з різною довжиною слова.** Довжина слова в бітах є параметром RC5 – при зміні довжини слова змінюється сам алгоритм.
- **Змінна кількість раундів.** Кількість раундів є другим параметром RC5. Цей параметр дозволяє вибрати оптимальне співвідношення необхідної швидкості роботи і вимог до ступеня захисту.
- **Змінна довжина ключа.** Довжина ключа є третім параметром RC5. Як і в попередньому випадку, цей параметр дозволяє знайти прийнятний компроміс між швидкістю роботи та необхідним рівнем безпеки.
- **Простота.** Структура RC5 дуже проста не тільки для реалізації, але й для оцінки її криптоаналітичної стійкості.
- **Низькі вимоги до пам'яті.** Низькі вимоги до пам'яті роблять RC5 придатним для використання в смарт-картах та інших подібних пристроях з обмеженим об'ємом пам'яті.
- **Високий ступінь захисту.** RC5 покликаний забезпечити високий ступінь захисту за умови вибору відповідних значень параметрів.

- **Залежність циклічних зсувів від даних.** В RC5 використовуються циклічні зсуви, величина яких залежить від даних, що повинно підвищувати криптоаналітичну стійкість алгоритму.

Алгоритм RC5 вбудований в багатьох основних продуктах компанії RSA Data Security Inc., включаючи BSAFE, JSAFE та S/MAIL.

RC5 фактично являє собою родину алгоритмів шифрування, що визначається трьома наступними параметрами (табл. 3).

Таблиця 3. Параметри алгоритму RC5

Параметр	**Визначення**	**Допустимі значення**
w	Розмір слова в бітах. RC5 шифрує дані блоками довжиною в 2 слова	16, 32, 64
r	Кількість раундів	0, 1, …, 255
b	Кількість 8-бітових байтів (октетів) в таємному ключі *K*	0, 1, …, 255

Таким чином, RC5 шифрує блоки відкритого тексту довжиною 32, 64 чи 128 бітів в блоки шифрованого тексту тієї самої довжини. Довжина ключа може змінюватись від 0 до 2040 бітів. Конкретна версія RC5 позначається RC5-*w*/*r*/*b*. Наприклад, RC5-32/12/16 використовує 32-бітові слова (64-бітові блоки відкритого і шифрованого тексту), 12 раундів шифрування і ключ довжиною 16 байтів (128 бітів). Райвест пропонує використовувати RC5-32/12/16 в якості "стандартної" версії RC5.

В алгоритмі RC5 виконуються три елементарні операції (а також обернені до них):

- **Додавання.** Додавання слів, позначене символом +, виконується по модулю 2^w. Обернена операція, позначена символом -, є відніманням по модулю 2^w.
- **Побітове виключне АБО.** Ця операція позначається символом ⊕.
- **Циклічний зсув вліво.** Циклічний зсув слова *x* вліво на *y* бітів позначається *x*<<<*y*. Обернена операція є циклічним зсувом слова *x* вправо на *y* бітів і позначається *x*>>>*y*.

Шифрування RC5.

Вхідні дані: відкритий текст M=(A,B) довжиною 2*w* бітів; кількість раундів *r*; ключ *K*=K[0]…K[b-1].

Вихідні дані: шифрований текст C довжиною 2*w* бітів.

1. Обчислити (2r+2) підключів S[0]…S[2r+1] за відповідним алгоритмом з вхідного *K* та *r*.
2. A:=A+S[0], B:=B+S[1] (додавання здійснюється за модулем 2^w).

3. **For** *i*=1 **to** *r* **do**: A:=((A⊕B)<<<B)+S[2i], B:=((B⊕A)<<<A)+S[2i+1].
4. Вихідним значенням є C:=(A,B). (див. рис. 5*a*)

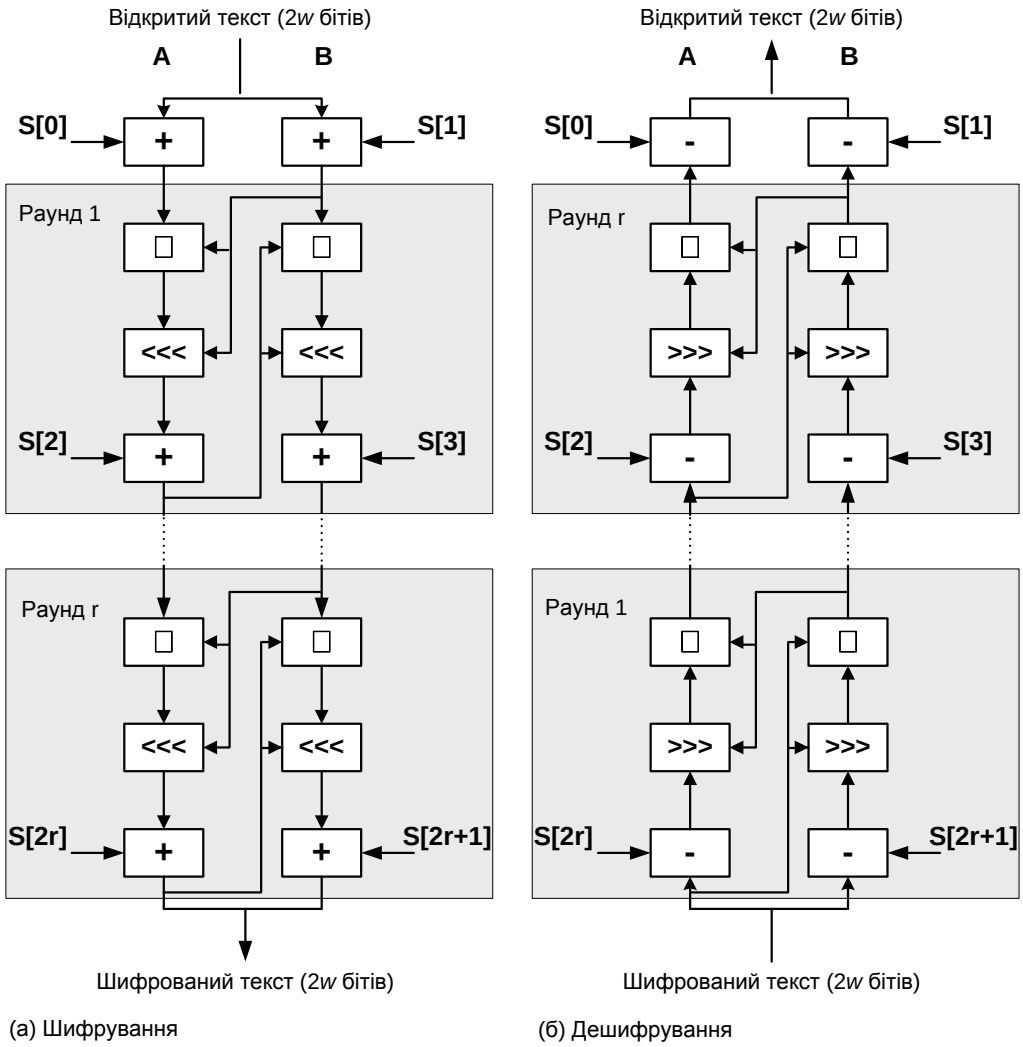

Рис. 5. Шифрування і дешифрування RC5.

Входом процесу блокового шифрування є розширена таблиця ключів S, кількість раундів r, вказівник на вхідний буфер in, та вказівник на вихідний буфер out. Заголовок процедури мовою C, наприклад, може виглядати наступним чином:

```
void RC5_Block_Encrypt (S, r, in, out)
RC5_WORD   *S;
int  r;
char   *in;
char   *out;
```

Для початкового завантаження даних слід перетворити вхідні байти на дві беззнакові цілі змінні A та B. Якщо RC5 використовується як 64-бітовий блочний шифр, A та B є 32-бітовими значеннями. Перший байт входу стає

найменш значущим байтом A, четвертий байт входу стає найбільш значущим байтом A, п'ятий байт входу стає найменш значущим байтом B і останній байт входу стає найбільш значущим байтом B. Це перетворення найбільш ефективно для процесорів з прямим порядком бітів у байті (little-endian), таких як процесори Intel. Код мовою C для такого перетворення може виглядати, наприклад, так:

```
int  i;
RC5_WORD   A, B;
A = in[0] & 0xFF;
A += (in[1] & 0xFF) << 8;
A += (in[2] & 0xFF) << 16;
A += (in[3] & 0xFF) << 24;
B = in[4] & 0xFF;
B += (in[5] & 0xFF) << 8;
B += (in[6] & 0xFF) << 16;
B += (in[7] & 0xFF) << 24;
```

Останнім кроком є перетворення A та B у послідовність байтів. Ця операція є оберненою до операції завантаження. Мовою C це можна, наприклад, представити як:

```
out[0] = (A >> 0) & 0xFF;
out[1] = (A >> 8) & 0xFF;
out[2] = (A >> 16) & 0xFF;
out[3] = (A >> 24) & 0xFF;
out[4] = (B >> 0) & 0xFF;
out[5] = (B >> 8) & 0xFF;
out[6] = (B >> 16) & 0xFF;
out[7] = (B >> 24) & 0xFF;
return;
```

Дешифрування RC5.

Для дешифрування використовуються ті самі підключі, які застосовуються для шифрованого тексту C=(A,B) наступним чином (віднімання здійснюється за модулем 2^w).

1. **For** *i*=*r* **downto** 1 **do**: B:=((B-S[2i+1])>>>A)⊕A, A:=((A-S[2i])>>>B)⊕B.
2. Результат M:=(A-S[0],B-S[1]). (рис. 5*б*)

Створення підключів RC5.

На рис. 6 показана схема обчислення підключів. Підключі зберігаються в масиві з (2*r*+2) слів, елементи якого позначені S[0], S[1],…,S[2r+1]. Використовуючи в якості вхідних даних параметри r та w, цей масив ініціалізується псевдовипадковими фіксованими значеннями. Потім ключ K[0…*b*-1] довжиною *b* байтів перетворюється в масив L[0…*c*-1], що містить *c* слів. Для цього байти ключа копіюються в масив L доповнюючи при

необхідності останнє слово справа нулями. Зрештою виконується деяка операція змішування, що об'єднує вміст L з ініціалізованими значеннями S, в результаті чого отримують остаточне значення масиву S.

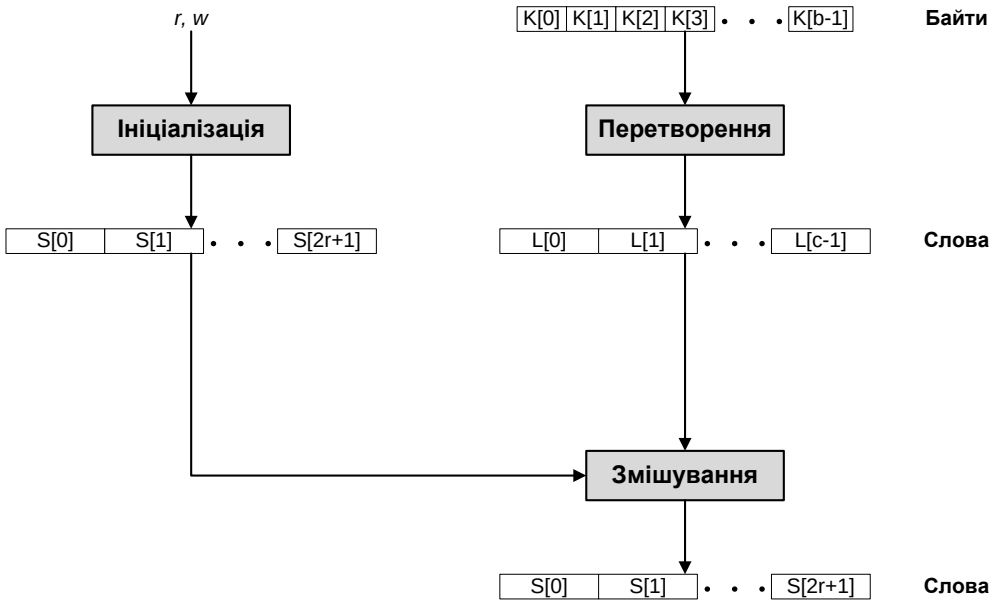

Рис. 6. Розгортання ключа RC5.

Алгоритм створення підключів виглядає наступним чином:

Вхідні дані: розмір слова в бітах *w*; кількість раундів *r*; ключ довжиною *b* байтів K[0]…K[*b*-1].

Вихідні дані: підключі S[0]…S[2r+1] (де кожне S[i] має довжину *w* бітів).

1. Перетворення ключа K в масив L. Нехай $u=w/8$ (кількість байтів у слові) і $c=\text{int}[b/u]$ (довжина K в словах).

Ключ K заповнюється справа нулями при необхідності отримати кількість байтів, що ділиться на *u* (тобто **for** j=*b* **to** (*c*·*u*-1) **do**: K[j]:=0).

For i=0 **to** c-1 **do**: $L[i]:=\sum_{j=0}^{u-1} 2^{8j} K[i\cdot u+j]$ (тобто заповнюємо L[j] від молодшого до старшого байта використовуючи по одному разу кожен байт K[·]).

2. Ініціалізація масиву S:

S[0]:=P_w; **for** i=1 **to** 2r+1 **do**: S[i]:=S[i-1]+Q_w. (тут P_w Q_w є константами, заснованими на двійковому представленні чисел *e* та φ (основи натуральних логарифмів та відношення золотого перетину). Значення цих констант для алгоритму RC5 наведено в табл. 4.)

3. Змішування ініціалізованого масиву S з масивом ключів L:

i:=0, j:=0, A:=0, B:=0, t:=max(*c*, 2*r*+2).

For s=1 **to** 3t **do**: (здійснюється три проходи більшого з масивів L і S)

S[i]:=(S[i]+A+B)<<<3, A:=S[i], i:=(i+1)mod(2*r*+2).

L[j]:=(L[j]+A+B)<<<(A+B), B:=L[j], j:=(j+1)mod(c)

4. Виходом є S[0], S[1], … , S[2r+1].

При створенні програмного забезпечення захисту інформації особливу увагу слід приділяти безпечній реалізації таких програм. Так, наприклад, основними операціями з ключами шифрування є створення, знищення та встановлення значення. З метою запобігання розповсюдження ключової інформації в інші частини програми, операція знищення повинна заповнити нулями область пам'яті виділену для роботи з ключем, перед тим як звільняти її менеджеру пам'яті. В загальному об'єкт ключа може підтримувати й інші операції, такі як створення нового випадкового ключа чи отримання ключа з даних протоколу обміну ключами.

Таблиця 4. Константи алгоритму RC5 (у шістнадцятковому вигляді)

w	16	32	64
P_w	B7E1	B7E15163	B7E15162 8AED2A6B
Q_w	9E37	9E3779B9	9E3779B9 7F4A7C15

Для створення ключа необхідно виділити та ініціалізувати область пам'яті для об'єкту ключа. В наступному прикладі коду мовою С передбачається, що функція з назвою "malloc" повертає блок ініціалізованої пам'яті з кучі, або нуль, що означає помилку.

```
rc5UserKey *RC5_Key_Create ()
{
rc5UserKey *pKey;
pKey = (rc5UserKey *) malloc (sizeof(*pKey));
if (pKey != ((rc5UserKey *) 0))
{
pKey->keyLength = 0;
pKey->keyBytes = (unsigned char *) 0;
}
return (pKey);
}
```

Для знищення ключа пам'ять повинна бути заповнена нулями та звільнена менеджеру пам'яті. В наступному прикладі коду мовою С передбачається, що функція з назвою "free" повертає блок пам'яті в кучу.

```
void RC5_Key_Destroy (pKey)
rc5UserKey     *pKey;
{
unsigned char   *to;
int         count;
```

```
if (pKey == ((rc5UserKey *) 0))
return;
if (pKey->keyBytes == ((unsigned char *) 0))
return;
to = pKey->keyBytes;
for (count = 0 ; count < pKey->keyLength ; count++)
*to++ = (unsigned char) 0;
free (pKey->keyBytes);
pKey->keyBytes = (unsigned char *) 0;
pKey->keyLength = 0;
free (pKey);
}
```

Двома найбільш важливими особливостями RC5 є простота алгоритму та використання керованих даними циклічних зсувів. Циклічні зсуви – єдина нелінійна складова даного алгоритму. Райвест стверджує, що, у зв'язку з тим, що величина зсуву визначається даними, що обробляються алгоритмом, лінійний та диференційний криптоаналіз алгоритму буде серйозно утруднений.

Режими RC5.

З метою забезпечення можливості ефективного використання RC5 в неоднорідному середовищі, специфікація RFC 2040 визначає чотири різних режими роботи цього алгоритму.

- **Блоковий шифр RC5.** Алгоритм прямого шифрування, при якому береться блок даних заданого розміру (2^w бітів) і з нього за допомогою залежного від ключа перетворення генерується блок шифрованого тексту такого самого розміру (див. рис. 7). Цей режим часто називають режимом ECB (режим електронної шифрувальної книги).
- **RC5-CBC.** Режим зв'язаних шифрованих блоків для RC5 (рис. 8). В режимі CBC обробляються повідомлення, довжина яких кратна розміру блока RC5 (тобто кратна 2*w* бітам). Режим CBC забезпечує вищий ступінь захисту, ніж ECB, оскільки генерує різні блоки шифрованого тексту для однакових повторних блоків відкритого тексту.
- **RC5-CBC-Pad.** Модифікація режиму CBC, призначена для роботи з відкритим текстом будь-якої довжини. Довжина шифрованого тексту в цьому режимі перевищує довжину відкритого тексту не більш ніж на довжину одного блоку RC5.
- **RC5-CTS.** Режим запозичення шифрованого тексту (ciphertext stealing), теж є модифікацією CBC. В цьому режимі допускається обробка відкритого тексту будь-якої довжини і генерується шифрований текст тієї самої довжини.

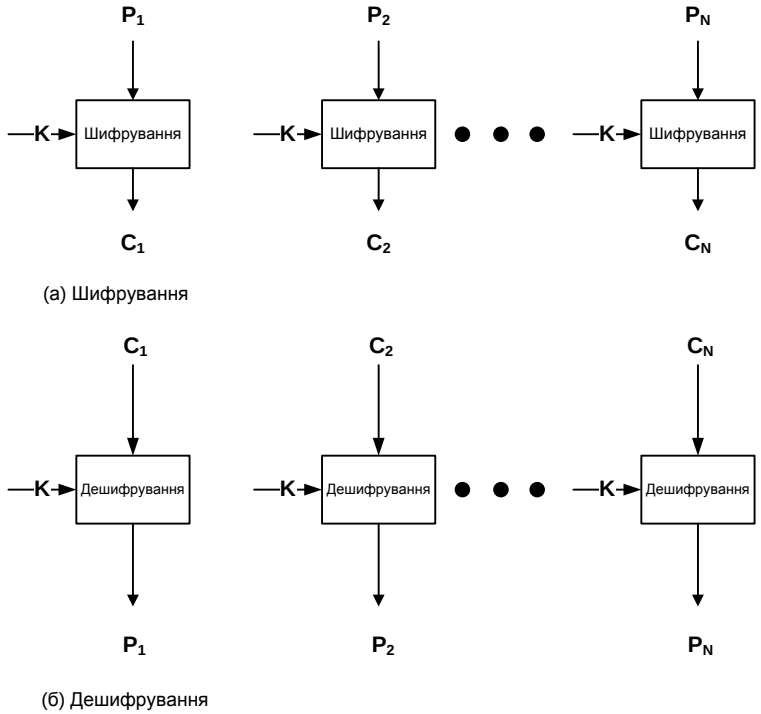

Рис. 7. Режим електронної шифрувальної книги.

Режим зв’язаних шифрованих блоків (Cipher Block Chaining – CBC) забезпечує вищий ступінь захисту ніж режим електронної шифрувальної книги (Electronic Codebook – ECB), в якому кожен блок відкритого тексту шифрується незалежно, оскільки генерує різні блоки шифрованого тексту для однакових блоків відкритого тексту, які повторюються в повідомленні. В режимі CBC (рис. 8) вхідне значення алгоритму шифрування дорівнює результату операції XOR поточного блоку відкритого тексту і отриманого на попередньому кроці блоку шифрованого тексту. Шифрування будь-якого блоку виконується за допомогою одного і того ж ключа. В результаті в процесі шифрування усі блоки відкритого тексту виявляються зв'язаними, а вхідні дані, що поступають на вхід функції шифрування, вже не жорстко зв'язані з блоками відкритого тексту. Тому однакові блоки відкритого тексту перетворюються в різні блоки шифрованого тексту.

При дешифруванні текст також проходить через алгоритм дешифрування поблочно. При цьому відповідний блок відкритого тексту отримується як результат операції XOR вихідного блоку алгоритму дешифрування і попереднього блоку шифрованого тексту. У вигляді формули цей режим можна записати як:

$$C_n = E_K\left[C_{n-1} \oplus P_n\right].$$

Тут E_K і D_K означає операцію шифрування (дешифрування) з ключем K, C_i та P_i – i-й блок шифрованого та відкритого тексту відповідно, а знак $\oplus$ позначає операцію побітового виключного АБО (XOR).

Тоді

$$D_K[C_n] = D_K[E_K(C_{n-1} \oplus P_n)] = (C_{n-1} \oplus P_n), \text{ а}$$

$$C_{n-1} \oplus D_K[C_n] = C_{n-1} \oplus C_{n-1} \oplus P_n = P_n.$$

Щоб отримати перший блок шифрованого тексту, розглядається результат операції XOR деякого вектора ініціалізації (IV) і першого блоку відкритого тексту. При дешифруванні для відновлення першого блоку відкритого тексту необхідно буде також виконати операцію XOR по відношенню до цього вектора IV і першого блоку на виході алгоритму дешифрування.

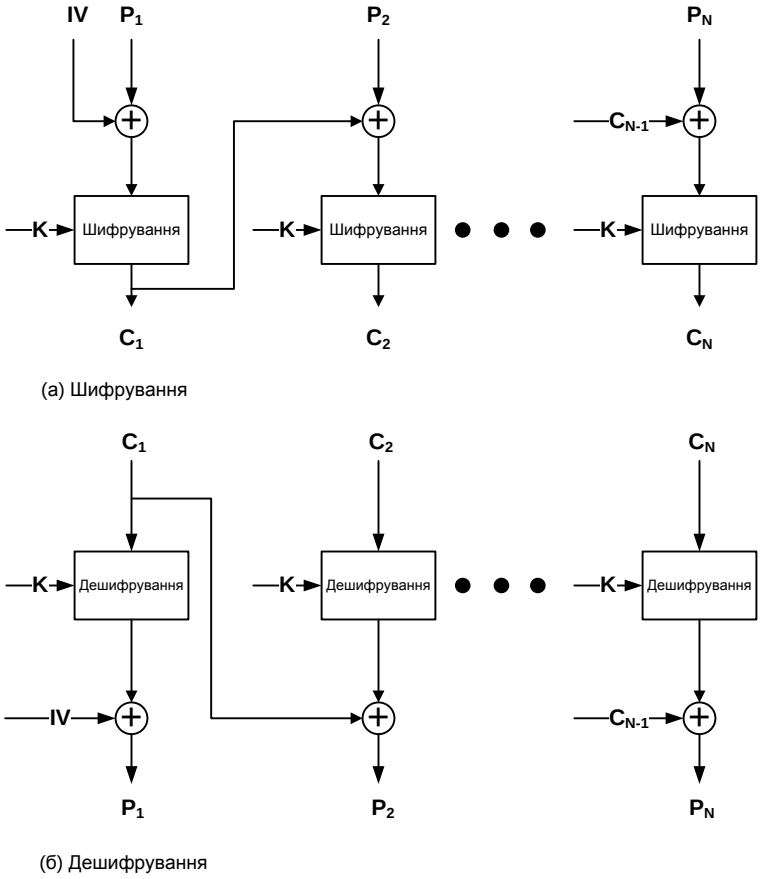

Рис. 8. Режим зв’язаних шифрованих блоків.

Значення IV повинно бути відомим і відправнику і отримувачу повідомлення. Для забезпечення максимальної безпеки значення IV повинно бути захищено так само, як і ключ шифрування. Можна, наприклад, відправити значення IV зашифроване в режимі ECB. Документ RFC 2040 передбачає в якості значення за замовчуванням рядок відповідної довжини, що складається з нульових байтів.

Коли шифрування повідомлення проводиться в режимі CBC, необхідний деякий алгоритм підготовки до шифрування повідомлень, довжина яких не кратна довжині блоку. Одним з таких методів для алгоритму шифрування RC5 є режим RC5-CBC-Pad. В цьому режимі, при підготовці до шифрування повідомлень, довжина яких не кратна довжині блоку, використовується заповнювач для доповнення повідомлення до потрібної довжини. В результаті довжина шифрованого тексту в цьому режимі перевищує довжину відкритого тексту не більше ніж на довжину одного блоку RC5. В алгоритмі RC5 вважається, що будь-яке повідомлення складається з цілої кількості байтів. В кінець повідомлення додається від 1 до *bb* байтів заповнювача[1], де *bb* дорівнює довжині блоку RC5 в байтах ($bb=2w/8$). Усі байти заповнювача вибираються однаковими і рівними значенню кількості доданих байтів. Наприклад, якщо додається 8 байтів, то кожен байт вибирається рівним 00001000.

Завдання до виконання роботи

Згідно до варіанту, наведеного в таблиці, створити прикладну програму для шифрування інформації за алгоритмом RC5.

Програма повинна отримувати від користувача парольну фразу і, на її основі, шифрувати файли довільного розміру, а результат зберігати у вигляді файлу з можливістю подальшого дешифрування (при введенні тієї самої парольної фрази).

Для перетворення парольної фрази у ключ шифрування використати алгоритм MD5, реалізований в лабораторній роботі № 2 – ключем шифрування повинен бути хеш парольної фрази. Якщо згідно варіанту довжина ключа становить 64 біти, беруться молодші 64 біти хешу; якщо довжина ключа повинна бути 256 бітів, то хеш парольної фрази стає старшими 128 бітами, а молодшими є хеш від старших 128 бітів (тобто, позначивши парольну фразу через *P*, отримаємо $K=H(H(P))||H(P)$).

Для забезпечення можливості роботи створеного програмного продукту з відкритим текстом довільної довжини, програмну реалізацію здійснити в режимі RC5-CBC-Pad. В якості вектора ініціалізації (IV) використати генератор псевдовипадкових чисел, реалізований в лабораторній роботі № 1. Для кожного нового шифрованого повідомлення слід генерувати новий вектор ініціалізації. Вектор ініціалізації зашифровується в режимі ECB і зберігається в першому блоці зашифрованого файлу.

[1] Зверніть увагу, що заповнювач додається завжди, навіть коли довжина відкритого тексту кратна довжині блоку!

У звіті навести протокол роботи програми та зробити висновки про поєднання різних криптографічних примітивів для задач захисту інформації.

Варіанти для виконання лабораторної роботи:

№ варіанту	*Довжина слова (w)*, біт	*Кількість раундів (r)*	*Довжина ключа (b)*, байт
1.	16	8	16
2.	32	12	16
3.	64	16	32
4.	16	20	16
5.	32	8	32
6.	64	12	8
7.	16	16	8
8.	32	20	16
9.	64	8	32
10.	16	12	16
11.	32	16	8
12.	64	20	16
13.	16	8	32
14.	32	12	32
15.	64	16	16
16.	16	20	8
17.	32	8	8
18.	64	12	16
19.	16	16	32
20.	32	20	32
21.	64	8	16
22.	16	12	8
23.	32	16	32
24.	64	20	8
25.	16	8	8

Контрольні запитання.

1. Для яких задач використовуються алгоритми симетричного шифрування?
2. Якими параметрами визначається алгоритм RC5?
3. Яка довжина ключа шифрування алгоритму RC5?
4. Які елементарні операції використовуються в алгоритмі RC5?
5. Які операції є нелінійними функціями алгоритму шифрування RC5?
6. Які режими роботи передбачені для алгоритму шифрування RC5?

7. Який режим роботи алгоритму RC5 слід використати якщо довжина шифрованого тексту повинна дорівнювати довжині довільного відкритого тексту?

Лабораторна робота № 4
СТВОРЕННЯ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ АЛГОРИТМУ ШИФРУВАННЯ З ВІДКРИТИМ КЛЮЧЕМ RSA З ВИКОРИСТАННЯМ MICROSOFT CRYPTOAPI

Мета роботи: ознайомитись з методами і засобами криптографії з відкритим ключем, навчитись створювати програмні засоби з використанням криптографічних інтерфейсів.

Теоретичні відомості.

Концепція криптографії з відкритим ключем була запропонована Уітфілдом Діффі (Whitfield Diffie) та Мартіном Хеллманом (Martin Hellman), і, незалежно, Ральфом Мерклом (Ralph Merkle). Основна ідея – використовувати ключі парами, що складаються з ключа шифрування та ключа дешифрування, які неможливо обчислити один з одного. Перша праця, присвячені цій проблемі вийшла у 1976 році, і з того часу було створено багато алгоритмів, що використовують концепцію відкритих ключів. Загальна схема виглядає наступним чином:

1. Кожен користувач генерує пару ключів: один для шифрування і один для дешифрування.
2. Кожен користувач публікує свій ключ шифрування, розміщує його у відкритому для всіх доступі. Другий ключ, парний до відкритого, зберігається в таємниці.
3. Якщо користувач А збирається надіслати повідомлення користувачеві В, він шифрує повідомлення відкритим ключем користувача В.
4. Коли користувач В отримує повідомлення, він дешифрує його за допомогою свого приватного ключа. Інший користувач не зможе дешифрувати повідомлення, оскільки приватний ключ В відомий тільки користувачеві В.

Алгоритми шифрування з відкритим ключем розроблялися для того, щоб вирішити дві найбільш важкі задачі, що виникли при використанні симетричного шифрування.

Першою задачею є розподіл ключа. При симетричному шифруванні потрібно, щоб обидві сторони вже мали спільний ключ, що якимось чином повинний бути їм заздалегідь переданий. Ця вимога заперечує всю суть криптографії, а саме можливість підтримувати загальну таємність при комунікаціях.

Другою задачею є необхідність створення таких механізмів, при використанні яких неможливо було б підмінити кого-небудь з учасників, тобто потрібен цифровий підпис.

Криптографія з відкритим ключем дозволяє вирішити набагато ширше коло задач, ніж криптографія класична. Однак існує ряд причин, з яких асиметричні алгоритми шифрування не можуть повноцінно замінити симетричні алгоритми:

- По-перше, алгоритми з таємним ключем набагато простіше реалізовуються як програмно, так і апаратно. Через це за однакових характеристик продуктивності та стійкості складність, а значить і ціна апаратних засобів, що реалізують шифр з відкритим ключем помітно вища за ціну апаратури, що реалізує класичний шифр, а при програмній реалізації на одному й тому ж типі процесора симетричні шифри працюють швидше асиметричних.
- По-друге, надійність алгоритмів з відкритим ключем на даний час обґрунтована набагато гірше, ніж надійність алгоритмів з таємним ключем і немає гарантії, що через деякий час вони не будуть розкриті, як це сталося з криптосистемою, заснованою на задачі про вкладання ранця.

Тому для організації шифрованого зв’язку на даний час застосовуються винятково класичні шифри, а методи криптографії з відкритим ключем використовуються тільки там, де вони не працюють, тобто для організації різних протоколів типу цифрового підпису, відкритого розподілу ключів тощо.

Алгоритм RSA

RSA (авторами є Rivest, Shamir і Adleman) – це алгоритм з відкритим ключем (асиметричний алгоритм), призначений як для шифрування, так і для аутентифікації. Був розроблений в 1977 році. З тих пір алгоритм RSA широко застосовується практично в усіх додатках, що використовують криптографію з відкритим ключем.

Алгоритм заснований на використанні того факту, що задача факторизації є складною, тобто легко перемножити два числа, у той час як не існує поліноміального алгоритму знаходження простих множників великого числа.

Алгоритм RSA є блоковим алгоритмом шифрування, де зашифровані і незашифровані дані є цілими числами в діапазоні між 0 і n-1 для деякого n.

Алгоритм, розроблений Райвестом, Шаміром і Адлеманом, використовує вирази з експонентами. Дані шифруються блоками, кожен блок розглядається як число, менше деякого числа n. Шифрування і дешифрування мають такий вигляд для деякого незашифрованого блоку M і зашифрованого блоку C.

$$C = M^e \bmod n,$$

$$M = C^d \bmod n = \left(M^e\right)^d \bmod n = M^{ed} \bmod n.$$

Як відправник, так і одержувач повинні знати значення n. Відправник знає значення e, одержувач знає значення d. Таким чином, відкритим ключем є $KU=\{e,n\}$, а закритим ключем є $KR=\{d,n\}$. При цьому повинні виконуватися наступні умови:

1. Можливість знайти значення e, d та n такі, що $M^{ed} = M \bmod n$ для усіх $M<n$.
2. Відносна легкість обчислення M^e і C^d для усіх значень $M<n$.
3. Неможливість визначити d, знаючи e та n.

Тепер розглянемо сам алгоритм RSA. Нехай p та q – прості числа.

$$n = p \cdot q.$$

Таким чином, слід вибрати e і d такі, що $e \cdot d = k \cdot \Phi(n) + 1$ (тут k – ціле число, $\Phi(n)$ – функція Ейлера). Це еквівалентно наступним співвідношенням:

$$e \cdot d \equiv 1 \bmod \Phi(n),$$
$$d \equiv e^{-1} \bmod \Phi(n).$$

Отже e і d є взаємно оберненими по множенню за модулем $\Phi(n)$. Зауважимо, що це вірно тільки в тому випадку, якщо d (і отже, e) є взаємно простими з $\Phi(n)$. Таким чином, $\gcd(\Phi(n), d) = 1$ (gcd – найбільший спільний дільник). Усі елементи алгоритму наведені в табл. 5.

Таблиця 5. Елементи алгоритму RSA

Елемент	**Опис**
p, q – два простих цілих числа	закриті, вибираються
$n = p \cdot q$	відкритий, обчислюється
e, таке що $\gcd(\Phi(n), e) = 1$, $1 < e < \Phi(n)$	відкритий, вибирається
$d \equiv e^{-1} \bmod \Phi(n)$	закритий, обчислюється

Як шифрування, так і дешифрування в RSA передбачають використання операції піднесення цілого числа до цілого степеня по модулю n. Якщо піднесення до степеня виконувати безпосередньо з цілими числами і потім проводити порівняння по модулю n, то проміжні значення виявляються дуже великими. Однак можна скористатись властивостями модульної арифметики: $(a \cdot b) \bmod n = ((a \bmod n) \cdot (b \bmod n) \bmod n)$. Таким чином, можна розглядати проміжні результати по модулю n. Це робить обчислення практично виконуваними.

Як вже зазначалось асиметричні алгоритми шифрування і, зокрема, RSA є набагато повільнішими за симетричні алгоритми. Так, наприклад, апаратно

RSA приблизно в 100–1000 разів повільніший за алгоритми симетричного шифрування. На даний час розроблено мікросхеми, які, використовуючи 512-бітовий модуль, перейшли рубіж 1 Мбіт/с. Виробники також застосовують RSA в смарт-картах, але ці реалізації є повільнішими. Програмно алгоритми симетричного шифрування приблизно у 10–100 разів швидші за RSA. Ці числа можуть незначно змінитись при зміні технології, однак RSA ніколи не досягне швидкості симетричних алгоритмів.

Швидкість програмної реалізації алгоритму RSA можна суттєво збільшити, якщо правильно вибрати значення *e*. Трьома найбільш вживаними варіантами є 3, 17 та 65537 ($2^{16}+1$). (Двійкове представлення 65537 містить тільки дві одиниці, тому для піднесення до степеня треба виконати тільки 17 множень.) Стандарт X.509 рекомендує використовувати значення 65537, PEM рекомендує 3, а PKCS#1 – 3 або 65537. Не існує жодних проблем безпеки, пов’язаних з використанням в якості *e* будь-якого з цих трьох значень (за умови доповнення повідомлення випадковими числами), навіть якщо одне й те саме значення *e* використовується цілою групою користувачів.

Операції із закритим ключем можна пришвидшити за допомогою китайської теореми про остачі, якщо зберегти значення *p* і *q*, а також додаткові значення: $d \bmod (p-1)$, $d \bmod (q-1)$ і $q^{-1} \bmod p$. Ці додаткові числа можна легко обчислити за закритим і відкритим ключами.

Криптографічні інтерфейси та Microsoft CryptoAPI

Підсистема безпеки комп'ютерної системи потребує реалізації ряду спільних функцій, пов'язаних з перетворенням вмісту об'єктів системи (файлів, записів бази даних) або з обчисленням деяких функцій зі спеціальними властивостями, які істотно залежать від вмісту об'єктів. До таких функцій відносяться алгоритми контролю цілісності об'єктів комп'ютерної системи, алгоритми аутентифікації або авторизації суб'єктів (чи користувачів, які управляють суб'єктами), алгоритми підтримки конфіденційності вмісту об'єктів (функції шифрування).

Суб'єкти (програми, процеси) комп'ютерної системи, пов'язані з виконанням захисних функцій, можуть використовувати деяку загальну підмножину функцій логічного перетворення об'єктів (зокрема, алгоритми шифрування і контролю цілісності об'єктів). При проектуванні і реалізації суб'єктів комп'ютерної системи, як правило, реалізується наступний підхід щодо використання спільного ресурсу – застосування розділювальних суб'єктів, що виконують функції, спільні для деякої підмножини зовнішніх по відношенню до даного суб'єкту функцій (динамічно завантажуваних бібліотек для Microsoft Windows). Логічно розповсюдити даний підхід на функції реалізації захисту. Серед множини функцій, що відносяться до захисної

компоненти комп'ютерної системи, можна виділити три класи: функції, пов'язані з роботою засобів ідентифікації і авторизації користувачів, суб'єктів і об'єктів; функції, пов'язані з контролем незмінності об'єктів; функції, пов'язані з логічним перетворенням об'єктів системи і підтримкою функцій конфіденційності (криптографічні функції). Також окремо виділяють функції генерації випадкових послідовностей, необхідних, зокрема, для формування індивідуальних аутентифікаційних ознак або ключів користувачів.

Проблему проектування суб'єктів, що реалізовують функції логічного захисту, можна розглядати в декількох аспектах:

- оптимальна реалізація об'єктів в рамках деякого об'єкта комп'ютерної системи, до якого звертається решта суб'єктів за виконанням відповідних функцій (в даному випадку йдеться про завдання оптимізації параметрів "швидкодія – пам'ять" при реалізації захисних функцій);
- мобільність суб'єктів, що використовують захисні функції при зміні внутрішнього наповнення, яке реалізує захисні функції об'єкта (мається на увазі задача максимальної переносимості суб'єктів, що використовують захисні функції на інші алгоритми, наприклад на інший алгоритм шифрування);
- коректне використання об'єкта, що реалізовує захисні функції, з боку сторонніх модулів, що його викликають.

Технологія застосування розділювальних функцій логічного захисту – це порядок використання засобів логічного захисту інформації в системі, при якому:

- не вимагається змін в програмному забезпеченні (у змісті і складі суб'єктів комп'ютерної системи) при зміні алгоритмів захисту;
- у комп'ютерній системі однозначно виділяється модуль реалізації захисних функцій.

Відкритим інтерфейсом модуля реалізації захисних функцій називають детальну специфікацію функцій, реалізованих в модулі, що дозволяє організувати виконання цих функцій із зовнішніх суб'єктів.

"Відкритість" інтерфейсу розуміється як його повний опис для використання зовнішніми суб'єктами реалізованих в модулі функцій. Для проектування схем інформаційних потоків комп'ютерної системи, зокрема тих, що мають відношення до забезпечення безпеки, такий опис виконує ключову роль.

В даний час існує досить багато криптографічних інтерфейсів. Однак розглянемо тільки базові криптографічні перетворення, тобто криптографічні інтерфейси, що в явному вигляді реалізовують основні криптографічні

операції: симетричне шифрування, хешування, цифровий підпис і несиметричний обмін ключами. Одним з найбільш поширених є Microsoft CryptoAPI 2.0. Розповсюдження CryptoAPI пов'язане не тільки з його зручністю, документованістю та іншими об'єктивними чинниками. Найважливішим чинником є те, що Microsoft найактивнішим чином інтегрувала його в свої операційні системи і прикладні програми. Сучасні операційні системи Microsoft (Windows 2000, Windows XP, Windows Vista) містять багато криптографічних підсистем різного призначення як прикладного рівня, так і рівня ядра, і ключову роль в реалізації цих підсистем відіграє інтерфейс CryptoAPI. Зрозуміло, в підсистемах рівня ядра базові криптографічні перетворення здійснюються безпосередньо в драйверах, що реалізовують ці підсистеми. Функції CryptoAPI в таких випадках використовуються для допоміжних операцій на прикладному рівні.

Далі детальніше розглянемо роботу саме інтерфейсу CryptoAPI, точніше, набору базових криптографічних функцій (base cryptography functions), який називають також інтерфейсом CryptoAPI 1.0.

Інтерфейс Microsoft CryptoAPI 2.0 містить як функції, що здійснюють базові криптографічні перетворення, так і функції, що реалізовують перетворення вищого рівня – роботу з сертифікатами X.509, роботу з криптографічними повідомленнями PKCS#7 та інші функції, що підтримують так звану інфраструктуру відкритих ключів (Public Key Infrastructure).

На основі використання можливостей базових криптографічних функцій можна не тільки більш глибоко розуміти роботу всього інтерфейсу CryptoAPI 2.0, але й створювати власні криптографічні підсистеми будь-якого рівня.

Реалізація всіх алгоритмів (шифрування, цифрового підпису і т.п.) повністю винесена із складу самого CryptoAPI і реалізується в окремих незалежних динамічних бібліотеках – "криптопровайдерах" (Cryptographic Service Provider – CSP).

Криптопровайдером називають незалежний модуль, що забезпечує безпосередню роботу з криптографічними алгоритмами. Кожен криптопровайдер повинен забезпечувати:

- реалізацію стандартного інтерфейсу криптопровайдера;
- роботу з ключами шифрування, призначеними для забезпечення роботи алгоритмів, специфічних для даного криптопровайдера;
- неможливість втручання третіх осіб в схему роботи алгоритмів.

Криптопровайдери реалізуються у вигляді динамічно завантажуваних бібліотек (DLL). Таким чином, достатньо важко втрутитись у хід виконання алгоритму, реалізованого в криптопровайдері, оскільки компоненти криптосистеми Windows повинні мати цифровий підпис. У криптопровайдерів

повинні бути відсутні можливості зміни алгоритму через встановлення його параметрів. Таким чином вирішується задача забезпечення цілісності алгоритмів криптопровайдера.

CryptoAPI надає наступні стандартні криптопровайдери:

- Microsoft Base Cryptographic Provider
- Microsoft Strong Cryptographic Provider
- Microsoft Enhanced Cryptographic Provider
- Microsoft AES Cryptographic Provider
- Microsoft DSS Cryptographic Provider
- Microsoft Base DSS and Diffie-Hellman Cryptographic Provider
- Microsoft DSS and Diffie-Hellman/Schannel Cryptographic Provider
- Microsoft RSA/Schannel Cryptographic Provider

Уся архітектура CryptoAPI може бути розділена три основні частини:

- Базові функції;
- Функції шифрування і роботи з сертифікатами;
- Функції роботи з повідомленнями.

Базові функції

До цієї групи входять функції для вибору і підключення до криптопровайдера, генерації і зберігання ключів, обміну ключами. Сюди так само входять можливості управління параметрами ключа, такими як: режим зчеплення блоків, вектор ініціалізації.

До базових функцій також можна віднести криптографічну функцію генерації випадкових даних. Це найбільш важлива функція у всій криптосистемі, оскільки вона користується для генерації ключів. Вся відповідальність за її реалізацію лягає на розробників криптопровайдера.

Основні функції в цій групі: *CryptAcquireContext, CryptReleaseContext, CryptGenKey, CryptDestroyKey, CryptExportKey, CryptImportKey, CryptDeriveKey, CryptGenRandom.*

Функції шифрування і для роботи з сертифікатами

До цієї групи входять всі функції для хешування даних, шифрування і дешифрування, а також функції для використання сертифікатів, основним завданням яких є надання доступу до відкритого ключа. CryptoAPI підтримує сертифікати специфікації X.509, в які входить інформація про версію сертифіката, його серійний номер, періоді дії, алгоритм шифрування публічного ключа та ін.

Основні функції в цій групі: *CryptEncrypt, CryptDecrypt, CryptCreateHash, CryptDestroyHash, CryptHashData, CryptSignHash, CertOpenStore, CertCloseStore, CertFindCertificateInStore.*

Функції для роботи з повідомленнями

Під повідомленнями в CryptoAPI розуміються дані в стандартизованому форматі PKCS #7, розробленому RSA Laboratories. Функції для роботи з ними діляться на дві частини: високорівневі функції (спрощені) і низькорівневі.

Використання функцій CryptoAPI для реалізації алгоритмів шифрування

Будь-який сеанс роботи з CryptoAPI починається з ініціалізації (отримання контексту)[1]. Ініціалізація здійснюється за допомогою функції CryptAcquireContext. В якості параметрів ця функція приймає ім'я контейнеру ключів, ім'я криптопровайдера, тип провайдера і «прапорці», що визначають тип і дії з контейнером ключів і режим роботи криптопровайдера.

```
HCRYPTPROV hProv;
::CryptAcquireContext(&hProv,/*параметр*/,ms_enhanced_prov,prov_rsa_full,0)
```

Криптопровайдер – це бібліотека, що реалізовує певний набір криптографічних алгоритмів і яка забезпечує роботу з ними. Існує близько семи стандартних провайдерів, встановлених в системі. Так, наприклад, Microsoft Base Cryptographic Provider вказується як MS_DEF_PROV, а Microsoft Enhanced Cryptographic Provider – як MS_ENHANCED_PROV.

Кожен криптопровайдер відноситься до певного типа. Це дозволяє, перебравши всі встановлені на машині провайдери, вибрати ті, які підтримують потрібні алгоритми. Два згадані провайдери мають тип PROV_RSA_FULL.

Криптопровайдер підтримує захищені області, звані контейнерами ключів. Контейнери дозволяють додаткам зберігати і використовувати згенеровані ключі, забезпечуючи захист самого ключа.

Контейнери бувають двох типів – користувацькі (цей тип використовується за умовчанням) і машинні (CRYPT_MACHINE_KEYSET). Користувацький контейнер доступний тільки додаткам, що виконуються від імені власника контейнера. Додаток може використовувати такий контейнер для збереження персональних ключів. Доступ до машинних контейнерів дозволений тільки адміністраторам. У них зазвичай зберігаються ключі, що використовуються службами і системними програмами. Тип контейнера задається прапорцем при отриманні контексту.

Для первинного створення контейнера потрібно викликати CryptAcquireContext з прапорцем CRYPT_NEWKEYSET. Для видалення контейнера потрібно вказати прапор CRYPT_DELETEKEYSET.

[1] Не забудьте інструкцію #include <wincrypt.h>.

Якщо додатку не потрібен доступ до контейнера ключів (наприклад, додаток обчислює хеш MD5), то варто викликати CryptAcquireContext з прапорцем CRYPT_VERIFYCONTEXT, передаючи NULL замість імені контейнера.

Деініціалізація CryptoAPI виконується за допомогою функції CryptReleaseContext, єдиним значущим параметром якої є одержаний раніше дескриптор криптографічного контексту.

Для генерації ключів в CryptoAPI передбачені дві функції – CryptGenKey і CryptDeriveKey. Перша з них генерує ключі випадковим чином, а друга – на основі даних користувача. При цьому гарантується, що для одних і тих же вхідних даних CryptDeriveKey завжди видає один і той же результат. Цей спосіб генерації ключів може бути корисним для створення симетричного ключа шифрування на базі пароля. Зупинимося детальніше на функції CryptGenKey, яка використовується найчастіше. Ця функція має прототип:

BOOL WINAPI CryptGenKey(HCRYPTPROV hProv, ALG_ID Algid, DWORD dwFlags, HCRYPTKEY phKey);*

Перший і четвертий параметри говорять самі за себе. Другим параметром передається ідентифікатор алгоритму шифрування, для якого генерується ключ (наприклад CALG_3DES). При генерації ключових пар RSA для шифрування і підпису використовуються спеціальні значення AT_KEYEXCHANGE і AT_SIGNATURE. Третій параметр задає різні опції ключа, які залежать від алгоритму і провайдера. Наприклад, старші 16 бітів цього параметра можуть задавати розмір ключа для алгоритму RSA. Докладний опис всіх прапорів можна знайти в MSDN. Параметри, які не можна задати при генерації ключа (наприклад, вектори, ініціалізацій), можна встановити вже після його створення за допомогою функції CryptSetKeyParam.

Обмін ключами в CryptoAPI реалізується за допомогою функцій CryptExportKey і CryptImportKey, що мають наступні прототипи:

BOOL WINAPI CryptExportKey(HCRYPTKEY hKey,HCRYPTKEY hExpKey,DWORD dwBlobType,DWORD dwFlags,BYTE pbData,DWORD* pdwDataLen);*

BOOL WINAPI CryptImportKey(HCRYPTPROV hProv,BYTE pbData,DWORD dwDataLen,HCRYPTKEY hImpKey,DWORD dwFlags,HCRYPTKEY* phKey);*

В якості ключів експорту/імпорту можуть використовуватися або ключова пара RSA (з типом AT_KEYEXCHANGE), або симетричний сеансовий ключ. Параметр dwBlobType залежить від того, який ключ експортується (імпортується), і задає тип структури, в яку поміщається ключ, що експортується. Для відкритого ключа це PUBLICKEYBLOB, і ключ експорту/імпорту при цьому позбавлений сенсу і повинен бути нулем. Для закритого ключа це PRIVATEKEYBLOB, і як ключ експорту може використовуватися сеансовий ключ. Для сеансового ключа це зазвичай

SIMPLEBLOB, а експортується він, як правило, на відкритому ключі одержувача.

Параметри pbData і pdwDataLen задають адресу і розмір буфера під структуру ключа, що експортується. Якщо розмір структури не відомий при написанні програми, то можна визначити необхідний розмір буфера, встановивши в pbData нуль. В цьому випадку потрібний розмір повертається в pdwDataLen.

Після закінчення роботи з ключем, його потрібно знищити викликом CryptDestroyKey.

Наступний приклад демонструє створення і експорт пари ключів для шифрування RSA:

```
//створення і експорт пари ключів  RSA
if(CryptGenKey(hProv,AT_KEYEXCHANGE,1024<<16,&hKey))  //генеруємо  1024-бітовий ключ
{
RSAPubKey1024 key;
DWORD dwLen=sizeof(RSAPubKey1024);
if(CryptExportKey(hKey,null,publickeyblob,0,(byte *)&key,pdwLen)) //експорт ключа
{
//... виконуються певні дії
}
CryptDestroyKey(hKey); //знищуємо ключ
}
```

Структура типу RSAPubKey1024 має наступний вигляд:

```
template<int bitlen> struct RSAPubKey
{
  PUBLICKEYSTRUC  publickeystruc ;
  RSAPUBKEY rsapubkey;
  BYTE modulus[bitlen/8];
};
template<int bitlen> struct RSAPrivKey
{
  RSAPubKey<bitlen> pubkey;
  BYTE prime1[bitlen/16];
  BYTE prime2[bitlen/16];
  BYTE exponent1[bitlen/16];
  BYTE exponent2[bitlen/16];
  BYTE coefficient[bitlen/16];
  BYTE privateExponent[bitlen/8];
};
```

Починаючи з Windows 2000 розширений (Enhanced) кріптопровайдер підтримує безпосереднє шифрування даних за алгоритмом RSA. Максимальний розмір даних, які можна зашифрувати за один виклик функції

Cryptencrypt, дорівнює розміру ключа мінус 11 байт. Справа в тому, що при шифруванні додається обов'язковий заповнювач (padding), який згодом перевіряється при дешифруванні. Відповідно, використання шифру RSA може бути доцільним тільки при невеликих об'ємах шифрованих даних (наприклад, при обміні ключами) через істотне збільшення об'єму шифрованого тексту і повільну роботу алгоритму RSA в порівнянні з блоковими шифрами.

Для шифрування даних відкритий текст зчитується блоками із вхідного файлу, а результат шифрування записується у вихідний файл, шифрування здійснюється функцією:

::CryptEncrypt(hKey,hHash,fSize<=BUFFER_SIZE,0,buf,&dwSzLow,sizeof(buf)),

а дешифрування:

::CryptDecrypt(hKey,hHash,fEncSize<=BUFFER_SIZE,0,buf,&dwSzLow))

Параметр hHash дозволяє паралельно з шифруванням/дешифруванням здійснювати хешування даних для подальшого електронного підпису або його перевірки. Прапор Final визначає, чи є шифрований блок даних останнім. Він необхідний, оскільки дані можна шифрувати по шматках, але для останнього блоку завжди виконується певна деініціалізація алгоритму (звільняються внутрішні структури), і багато алгоритмів проводять додавання (і перевірку коректності при дешифруванні) заповнювача (padding) після основних даних. Параметри pbData і pdwDataLen задають адресу буфера і розмір шифрованих даних. Для не останнього блоку даних (Final=FALSE) розмір даних повинен бути завжди кратний розміру шифрованого алгоритмом блоку. Для останнього блоку допускається порушення цієї умови.

Приклад шифрування наведено в наступному коді:

```
BYTE buf[BUFFER_SIZE+8]; //8 – запас на padding
while(fSize)
{
if(!::ReadFile(hInFile,buf,BUFFER_SIZE,pdwLen,NULL)) //читаємо блок даних
break;
dwSzLow=dwLen;
if(!::CryptEncrypt(hKey,hHash,fSize<=BUFFER_SIZE,0,buf,&dwSzLow,sizeof(buf)))
//шифруємо і хешуємо його
break;
if(!::WriteFile(hOutFile,buf,&dwSzLow,pdwSzLow,NULL))
break;
fSize-=dwLen;
}
```

Завдання до виконання роботи

З використання функцій CryptoAPI створити програмну реалізацію алгоритму шифрування RSA. Оцінити швидкість шифрування алгоритму RSA та порівняти її зі швидкістю шифрування алгоритму RC5, реалізованого в роботі № 3, зробити відповідні висновки та відобразити їх у звіті до лабораторної роботи.

Контрольні запитання.

1. Опишіть схематично використання алгоритмів шифрування з відкритим ключем.
2. Для яких задач створені алгоритми з відкритим ключем?
3. Основні недоліки алгоритмів з відкритим ключем.
4. Області використання алгоритмів симетричного і асиметричного шифрування.
5. На якій складній задачі засновано алгоритм RSA?
6. Як можна збільшити швидкодію програмної реалізації алгоритму RSA?
7. Що включають в інтерфейс CryptoAPI 1.0?

Лабораторна робота № 5
СТВОРЕННЯ ПРОГРАМНОГО ЗАСОБУ ДЛЯ ЦИФРОВОГО ПІДПИСУ ІНФОРМАЦІЇ З ВИКОРИСТАННЯМ MICROSOFT CRYPTOAPI

Мета роботи: ознайомитись з методами криптографічного забезпечення цифрового підпису, навчитись створювати програмні засоби для цифрового підпису з використанням криптографічних інтерфейсів.

Теоретичні відомості.

Аутентифікація захищає двох учасників, які обмінюються повідомленнями, від впливу деякої третьої сторони. Однак проста аутентифікація не захищає учасників один від одного, тоді як і між ними теж можуть виникати певні форми суперечностей.

У ситуації, коли обидві сторони не довіряють один одному, необхідно щось більше, ніж аутентифікація на основі спільного секрету. Можливим рішенням подібної проблеми є використання цифрового підпису. Цифровий підпис повинен володіти наступними властивостями:

1. Повинна бути можливість перевірити автора, дату й час створення підпису.
2. Повинна бути можливість встановити достовірність вмісту повідомлення на час створення підпису.
3. Повинна бути можливість перевірки підпису третьою стороною для вирішення суперечок.

Таким чином, функція цифрового підпису включає, зокрема, і функцію аутентифікації.

На підставі цих властивостей можна сформулювати наступні вимоги до цифрового підпису:

1. Підпис повинен бути бітовим зразком, який залежить від повідомлення, що підписується.
2. Підпис повинен використовувати деяку унікальну інформацію відправника для запобігання підробки або відмови.
3. Створювати цифровий підпис повинно бути відносно легко.
4. Повинно бути обчислювально неможливо підробити цифровий підпис як створенням нового повідомлення для існуючого цифрового підпису, так і створенням фальшивого цифрового підпису для деякого повідомлення.
5. Цифровий підпис повинен бути досить компактним і не займати багато пам'яті.

Стандарт цифрового підпису DSS

Національний інститут стандартів і технології США (NIST) розробив федеральний стандарт цифрового підпису DSS. Для створення цифрового підпису використовується алгоритм DSA (Digital Signature Algorithm). В якості хеш-алгоритму стандарт передбачає використання SHA-1 (Secure Hash Algorithm). DSS спочатку був запропонований в 1991 році й переглянутий в 1993 році у відповідь на публікації, що стосуються безпеки його схеми. У 1996 році в нього були внесені незначні зміни.

DSS використовує алгоритм, що розроблявся для використання тільки в якості цифрового підпису. На відміну від алгоритму RSA, його не можна використовувати для шифрування чи обміну ключами.

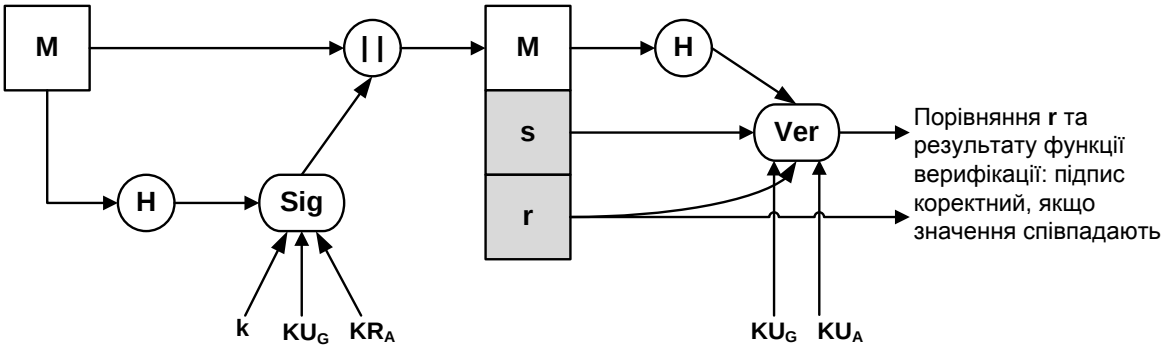

Рис. 9. Створення й перевірка підпису згідно стандарту DSS.

Стандарт DSS використовує при створенні підпису хеш-функцію. Значення хешу повідомлення є входом функції підпису разом з випадковим числом *k*, створеним для цього конкретного підпису (рис. 9). Функція підпису також залежить від приватного ключа відправника KR_A і деякої множини параметрів, відомих усім учасникам. Можна вважати, що ця множина утворює глобальний відкритий ключ KU_G. Результатом є підпис, що складається з двох компонентів, позначених як *s* та *r*.

Для перевірки підпису одержувач також обчислює хеш-код отриманого повідомлення. Цей хеш-код разом з підписом є входом функції верифікації. Функція верифікації залежить від глобального відкритого ключа KU_G і відкритого ключа відправника KU_A. Виходом функції верифікації є значення, що повинне дорівнювати компоненті *r* підпису, якщо підпис коректний. Функція підпису має такий вигляд, що тільки відправник, якому відомий приватний ключ, може створити коректний підпис.

Розглянемо деталі алгоритму, що використовується в стандарті DSS. Цей алгоритм заснований на труднощі обчислення дискретних логарифмів.

Спільні компоненти групи користувачів (глобальний відкритий ключ)

Існує три параметри, які є відкритими й можуть бути спільними для великої групи користувачів.

Вибирається 160-бітне просте число q, тобто $2^{159} < q < 2^{160}$.

Потім вибирається таке просте число p довжиною від 512 до 1024 бітів, так, щоб q було дільником (p-1).

Нарешті, вибирається число g виду $h^{(p-1)/q} \bmod p$, де h є цілим в проміжку від 1 до (p-1) з тим обмеженням, що g повинне бути більшим за 1.

Знаючи ці числа, кожний користувач вибирає приватний ключ і створює відкритий ключ.

Приватний ключ відправника.

Приватний ключ x повинен бути числом між 1 і (p-1) і повинен бути обраним випадково або псевдовипадково.

x – випадкове або псевдовипадкове ціле, $0<x<q$.

Відкритий ключ відправника.

Відкритий ключ обчислюється із приватного ключа за формулою $y = g^x \bmod p$. Обчислити y за відомим x досить просто. Однак, маючи відкритий ключ y, обчислювально неможливо визначити x, що є дискретним логарифмом y за основою g.

$$y = g^x \bmod p$$

Випадкове число, унікальне для кожного підпису.

k – випадкове або псевдовипадкове ціле число, $0<k<q$, унікальне для кожного підпису.

Підписування.

Для створення підпису відправник обчислює дві величини, r і s, які є функцією компонентів спільного відкритого ключа (p,q,g), приватного ключа користувача (x), хеш-коду повідомлення $H(M)$ і цілого числа k, унікального для кожного підпису.

$$r = \left(g^k \bmod p\right) \bmod q$$

$$s = \left[k^{-1} \cdot \left(H(M) + x \cdot r\right)\right] \bmod q$$

$$Підпис = (r, s)$$

Перевірка підпису.

Одержувач виконує перевірку підпису наступним чином. Він обчислює величину v, що є функцією компонентів спільного відкритого ключа, відкритого ключа відправника й хеш-коду отриманого повідомлення. Якщо значення цієї величини дорівнює значенню компонента r в підписі, то підпис вважається дійсним.

$$w = s^{-1} \bmod q$$

$u_1 = [H(M) \cdot w] \bmod q$

$u_2 = r \cdot w \bmod q$

$v = [(g^{u_1} \cdot y^{u_2}) \bmod p] \bmod q$

підпис коректний, якщо *v=r*.

Зверніть увагу на те, що перевірка здійснюється зі значенням *r*, що не залежить від повідомлення взагалі. Значення *r* є функцією *k* й трьох компонентів глобального відкритого ключа.

При труднощі обчислення дискретних логарифмів для противника виявляється нереальним з погляду обчислень знайти *k* за відомим *r* або знайти *x* за відомим *s*.

Використання функцій CryptoAPI для реалізації цифрового підпису

Базова функція отримання підпису хешу даних має наступний опис:

BOOL CryptSignHash(HCRYPTHASH hHash, DWORD dwKeySpec, LPCTSTR sDescription, WORD dwFlags, BYTE pbSignature, DWORD* pdwSigLen);*

В якості першого параметру використовується значення дескриптора хеш-об'єкту, вже ініціалізованого даними (за допомогою, наприклад, функції CryptHashData). Параметр *dwKeySpec* визначає, яка саме пара ключів буде використана для формування підпису (*AT_KEYEXCHANGE* (пара для обміну ключами) чи *AT_SIGNATURE* (пара для формування цифрового підпису)). В багатьох (однак не в усіх) криптопровайдерах пара ключів, призначена для обміну ключами, може також використовуватись і для формування цифрового підпису. Параметр *sDescription* більше не використовується в цій функції, і його значення повинно завжди бути встановленим в *NULL*. Параметр *dwFlags* зазвичай також встановлюють в 0. Параметри *pbSignature* та *pdwSigLen* використовують для коректного визначення посилання на масив вихідних даних та його розмір.

Приклад використання цієї функції наведено нижче:

```
// Цифровий підпис хеш значення
count = 0;
if(!CryptSignHash(hHash, AT_SIGNATURE, NULL, 0, NULL, &count))
{
Error("CryptSignHash");
return;
}
char* sign_hash = static_cast<char*>(malloc(count + 1));
ZeroMemory(sign_hash, count + 1);
if(!CryptSignHash(hHash, AT_SIGNATURE, NULL, 0, (BYTE*)sign_hash, &count))
{
Error("CryptSignHash");
return;
}
```

```
std::cout << "Signature created" << std::endl;
// Вивід на екран значення цифрового підпису
std::cout << "Signature value: " << sign_hash << std::endl;
```

Для перевірки цифрового підпису хеш-значення використовується базова функція, що має наступний опис:

BOOL CryptVerifySignature(HCRYPTHASH hHash, BYTE pbSignature, DWORD dwSigLen, HCRYPTKEY hPubKey, LPCTSTR sDescription, DWORD dwFlags);*

В якості першого параметру в функцію передається дескриптор хеш-об'єкту, попередньо ініціалізованого даними засобами функції CryptHashData. Другий і третій параметри відповідають за передачу значення підпису, що перевіряється. Параметр *hPubKey* використовується для зазначення дескриптора публічного ключа відправника підпису. Параметр *sDescription* на даний час більше не використовується і його значення повинно бути встановленим в *NULL*. Параметр *dwFlags* також зазвичай не несе корисної інформації і встановлюється в 0.

Приклад перевірки підпису наведено нижче:

```
// Отримання публічного ключа (для перевірки цифрового підпису)
HCRYPTKEY hPublicKey;
if(!CryptGetUserKey(hProv, AT_SIGNATURE, &hPublicKey))
{
Error("CryptGetUserKey");
return;
}
std::cout << "Public key is received" << std::endl;
// перевірка цифрового підпису
BOOL result = CryptVerifySignature(hHash, (BYTE*)sign_hash, count, hPublicKey, NULL, 0);
std::cout << "Check is completed" << std::endl;
// Вивід на екран результату перевірки цифрового підпису
std::cout << "Check result: " << ((result)? "Verified!" : "NOT verified!") << std::endl;
```

Розмір підпису дорівнює розміру ключа RSA або DSS, відповідно.

Завдання до виконання роботи

З використання функцій CryptoAPI створити прикладну програму для створення і перевірки цифрового підпису за стандартом DSS. Програмна реалізація повинна виводити значення підпису як для рядка, заданого в полі вводу, так і для файлу. Результат роботи програми повинен відображатись на екрані з можливістю наступного запису в файл. Крім того програма повинна мати можливість перевірити цифровий підпис будь-якого файлу за наявним файлом підпису, записаним у шістнадцятковому форматі. У звіті навести протокол роботи програми та зробити висновки.

Контрольні запитання.

1. Для чого призначений цифровий підпис?
2. Назвіть вимоги до цифрового підпису.
3. Який алгоритм хешування використовується в стандарті DSS?
4. Чи можна використати алгоритм DSA для обміну ключами?
5. На якій математичній проблемі засновано алгоритм DSA?
6. Що таке глобальний відкритий ключ алгоритму DSA?
7. Яка функція CryptoAPI використовується для створення цифрового підпису?

СПИСОК ЛІТЕРАТУРИ

1. Столлингс В. Криптография и защита сетей: принципы и практика, 2-е изд.: Пер. с англ. – М.: Издательский дом «Вильямс», 2001. – 672 с.
2. Б. Шнайер Прикладная криптография: Протоколы, алгоритмы, исходные тексты на языке Си. – М.: Издательство ТРИУМФ, 2003. – 816 с.
3. Ростовцев А.Г., Маховенко Е.Б. Теоретическая криптография. – СПб.: НПО "Профессионал", 2004. – 478 с.
4. Абашев А.А., Жуков И.Ю., Иванов М.А., Метлицкий Ю.В., Тетерин И.И. Ассемблер в задачах защиты информации.– М.: КУДИЦ-ОБРАЗ, 2004.– 544 с.
5. Саломаа А. Криптография с открытым ключом.– М.: Мир, 1995.– 318 с.
6. А. Щербаков, А. Домашев Прикладная криптография. Использование и синтез криптографических интерфейсов. – М: Русская редакция, 2003. – 406 с.
7. R.L. Rivest "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.
8. R. Baldwin, R.L. Rivest "The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms", RFC 2040, October 1996.

НАВЧАЛЬНЕ ВИДАННЯ

ПРОГРАМНА РЕАЛІЗАЦІЯ ОСНОВНИХ СЕРВІСІВ БЕЗПЕКИ

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт
з дисциплін "Програмне забезпечення захисту інформації",
"Захист програм та даних"
для студентів базових напрямів 6.050101 "Комп'ютерні науки",
6.050103 "Програмна інженерія"

Укладач Яковина Віталій Степанович

Редактор

Комп’ютерне верстання