**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ
ІНСТИТУТ**

Кафедра математичних методів захисту інформації

«До захисту допущено»
В.о. завідувача кафедри
__________ Сергій ЯКОВЛЄВ
«__» __________ 2022 р.

Дипломна робота

на здобуття ступеня бакалавра

зі спеціальності: 113 Прикладна математика
на тему: **«Побудова і дослідження атак на систему шифрування RSA»**

Виконав: студент 4 курсу, групи ФІ-83
Гузей Дмитро Русланович

Керівник: доцент, д.ф-м.н., професор Савчук М. М. __________

Консультант: _ __________

Рецензент: доцент, канд. тех. наук Гальчинський Л.Ю. __________

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент __________

Київ — 2022

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра математичних методів захисту інформації

Рівень вищої освіти — перший (бакалаврський)
Спеціальність (освітня програма) — 113 Прикладна математика,
ОПП «Математичні методи криптографічного захисту інформації»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
__________ Сергій ЯКОВЛЄВ
«__» __________ 2022 р.

ЗАВДАННЯ
на дипломну роботу

Студент: Гузей Дмитро Русланович

1. Тема роботи: «*Побудова і дослідження атак на систему шифрування RSA*»,

керівник: доцент, д.ф-м.н., професор Савчук М. М.,

затверджені наказом по університету №__ від «__» __________ 2022 р.

2. Термін подання студентом роботи: «__» __________ 2022 р.

3. Вихідні дані до роботи: *опубліковані джерела за тематикою дослідження*

4. Зміст роботи: *Дослідження криптосистеми RSA та їх особливостей; реалізація атак на RSA та дослідження впливу параметрів атаки на її час роботи; оцінка складності атаки*

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): *Презентація доповіді*

6. Дата видачі завдання: 10 вересня 2021 р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання	Примітка
1	Узгодження напряму дослідження із науковим керівником	01 - 15 вересня 2021 р.	Виконано
2	Огляд опублікованих джерел за тематикою дослідження	15 вересня - 8 січня 2021 р.	Виконано
3	Огляд опублікованих джерел за тематикою дослідження	8 січня - 5 лютого 2021 р.	Виконано
4	Реалізація атаки зустріч посередині	5 лютого - 19 лютого 2021 р.	Виконано
5	Реалізація атаки Вінера	19 лютого - 12 березня 2021 р.	Виконано
6	Обчислення оцінки складності для атаки зустріч посередині	12 березня - 26 березня 2021 р.	Виконано
7	Експериментальне дослідження ймовірності успіху атаки зустріч посередині	26 березня - 9 квітня 2021 р.	Виконано
8	Обчислити експериментально залежність складності від розміру параметрів в атаці Вінера	9 квітня - 30 квітня 2021 р.	Виконано
9	Обчислити теоретично залежність складності від розміру параметрів в атаці Вінера	1 травня - 31 травня 2021 р.	Виконано
10	Оформлення пояснювальної записки до роботи	червень 2021 р.	Виконано

Студент ____________ Гузей Д.Р.

Керівник ____________ Савчук М. М.

РЕФЕРАТ

Кваліфікаційна робота містить: 58 стор., 3 рисунки, 3 таблиці, 13 джерел.

Метою дослідження є аналіз криптографічних атак на систему шифрування RSA, побудова програмних реалізацій деяких атак на систему шифрування RSA, та експериментальне дослідження складності атак та ймовірності успіху, а також уточнення теоретичних оцінок складності.

Об'єктом дослідження є інформаційні процеси в системах криптографічного захисту.

Предметом дослідження є алгоритми атаки на криптосистему RSA та методи оцінки складності атак, асимптотичний аналіз, статистичне моделювання атак, ймовірність успіху атак.

У ході дослідження зроблено огляд криптосистеми RSA. Розглянуто наявні атаки на криптосистему RSA. Результатами роботи є побудовані функції залежності складності атаки Вінера і атаки зустрічі посередині від параметрів у криптосистемі RSA. Уточнено теоретичні оцінки складності складності атаки Вінера. Обчислено ймовірность успіху атаки зустріч посередині залежно від параметрів у криптосистемі RSA.

RSA, АТАКА ВІНЕРА, АТАКА ЗУСТРІЧ ПОСЕРЕДИНІ

ABSTRACT

The thesis contains: 58 pages, 13 sources.

The purpose of work is analyze cryptographic attacks on the RSA encryption system and build software implementations of some attacks on the RSA encryption system, and experimental study of attack complexity and probability of success, as well as refinement of theoretical estimates of complexity.

The object is information processes in cryptographic protection systems.

The subject are algorithms of attack on the cryptosystem RSA and methods of estimating the complexity of attacks, asymptotic analysis, statistical modeling.

The thesis reviews the cryptosystem RSA. Existing attacks on the RSA cryptosystem are considered. The results of the work are constructed functions of the dependence of the complexity of the Wiener's attack and meet in the middle attack on the parameters in the cryptosystem RSA. Theoretical estimates of the complexity of Wiener's attack have been refined. The probability of success of an meet in the middle attack is calculated depending on the parameters in the RSA cryptosystem.

RSA, WIENER'S ATTACK, MEET IN THE MIDDLE ATTACK

ЗМІСТ

Перелік умовних позначень, скорочень і термінів 7
Вступ 8
1 Асиметричні криптосистеми RSA та її криптоаналіз 10
1.1 Опис криптосистем RSA 10
1.2 Атаки на криптосистему RSA 13
Висновки до розділу 1 21
2 Побудова атак на криптосистему RSA та дослідження їх параметрів 22
2.1 Атака на RSA зустріч посередині 22
2.2 Атака Вінера з використанням ланцюгових дробів 24
Висновки до розділу 2 31
3 Програмна реалізація атак та їх експериментальне дослідження 32
3.1 Теоретична оцінка складності атаки Вінера 32
3.2 Експериментальна оцінка складності та ймовірності успіху атаки зустріч посередині 33
3.3 Експериментальна оцінка складності атаки Вінера 36
Висновки до розділу 3 38
Висновки 39
Перелік посилань 40
Додаток А Тексти програм 42
А.1 Атака зустріч посередині 42
А.2 Атака Вінера 47
А.3 Ймовірність успіху атаки зустріч посередині 53

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

НСД(a,b) – найбільший спільний дільник чисел a та b.

НСК(a,b) – найменше спільне кратне чисел a та b.

mod – остача від ділення

φ(n) – функція Ейлера

ln() – натуральний логарифм

Z – поле

Z_n – множина цілих чисел від 0 до n - 1

O($\cdot$) – O-велике, нотація Ландау

o($\cdot$) – o-маленьке, нотація Ландау

$\lambda(N)$ – функція Карлмайкла

ВСТУП

Актуальність дослідження. Поява асиметричної криптографії дозволила вирішити багато проблем захисту інформації, зокрема: передача секретних ключів, автентифікація, безпечне зберігання паролів. Криптосистема RSA на сьогоднішній день є найпопулярнішою схемою шифрування. RSA використовується для захисту програмного забезпечення й у схемах цифрового підпису. Також вона використовується у відкритій системі шифрування PGP, DarkCryptTC і форматі xdc, TLS, SSH в поєднанні з симетричними алгоритмами. Дуже чутливим параметром RSA є швидкість розшифрування і розмір ключів. Саме для цього і будуються чимало атак підбираючи параметри, щоб якомога зменшити стійкість криптосистеми, а також і при використанні даної атаки підбираються параметри таким чином, щоб дана система була стійкою для більшості атак Тому дослідження стійкості таких криптосистем є актуальною задачею, і для забезпечення надійності побудова кожної з них повинні враховувати наявні атаки.

Метою дослідження є аналіз криптографічних атак на систему шифрування RSA та побудова програмних реалізацій деяких атак на систему шифрування RSA, та експериментальне дослідження складності атак та ймовірності успіху, а також уточнення теоретичних оцінок складності.

Для досягнення мети необхідно вирішити такі завдання:

1) дослідити криптосистему RSA;

2) зробити огляд наявних атак на криптосистему шифрування RSA;

3) застосувати атаку "зустріч посередині"для різних значень відкритого ключа на криптосистему RSA, дослідити її експериментально та обчислити оцінку складності;

4) реалізувати атаку Вінера, дослідити її експериментально та обчислити залежність складності від розміру параметрів (теоретично та

експериментально).

Об'єктом дослідження являються інформаційні процеси в системах криптографічного захисту.

Предметом дослідження є алгоритми атаки на криптосистему RSA та методи оцінки складності атак, асимптотичний аналіз, статистичне моделювання.

При розв'язанні поставлених завдань використовувались такі *методи дослідження*: методи теорії імовірностей, комбінаторного аналізу, теорії складності алгоритмів.

Наукова новизна результатів полягає в експериментальній побудові функцій залежності складності атаки Вінера та складності і ймовірності успіху атаки зустрічі посередині від параметрів у криптосистемі RSA, а також уточненні теоретичних оцінок складності.

Практичне значення результатів полягає у підтвердженні рекомендацій для виборів параметрів при побудові атаки Вінера та атаки зустрічі посередині та секретних параметрів криптосистеми RSA.

1 АСИМЕТРИЧНІ КРИПТОСИСТЕМИ RSA ТА ЇЇ КРИПТОАНАЛІЗ

У даному розділі розглядається криптосистема RSA та деякі відомі атаки, які побудовані на цю криптосистему, зосереджено увагу на деяких задачах RSA, оцінках складності обчислень і зосереджено увагу на алгоритмах даних атак, а також параметрах даних атак при яких вони працюють ефективно та мають поліноміальну та субекспоненційну складність.

1.1 Опис криптосистем RSA

Криптосистема RSA

В першому підрозділі були використані такі джерела: [4, 5]. В 1976 році В. Діффі, М. Хеллман в «Нові напрями в криптографії»[9], ввели термін односторонньої та односторонньої функції з секретом на основі якої згодом була побудована криптосистема RSA.

Означення 1.1. Односторонньою функцією с секретом RSA називається функція:

$y = x^e mod n$, (1)

де – n = pq, $p \neq q$ великі прості числа, (e,φ(n)) = 1, p і q є секретом, знання якого дає можливість швидко знайти обернену функцію. При невідомих p і q і обернення функції (1) практично неможливо. Якщо x,y є 0,1,2...n-1, то функція (1) бієкция. Обернена функція для функції RSA називається дискретним обчисленням кореня (степеня e за модулем n).

Розглянемо, власне, побудову криптосистеми RSA

Алгоритм 1.1. Побудова криптосистеми RSA користувачем:

Користувач робитиме наступне:

1) Вибирає великі прості p і q (наприклад, по 512 біт);

2) обчислює n=pq та φ(n) = (p-1)(q-1). При цьому n – не секретне, p, q і φ(n) а – секрет A;

3) вибирає e, 1<e< φ(n), (e, φ(n)) = 1;

4) розв’язує відносно d наступне рівняння для знаходження секретного ключа:

ed=1modφ(n), тобто d = $e^{(-1)}\varphi$(n)

Побудований ключ d — це секретний ключ A для розшифрування повідомлень до A і формування цифрового підпису абонентом A;

5) Публікує відкритий ключ (n,e) для зашифрування повідомлень до A і перевірки цифрового підпису в повідомленнях від A.

На практиці відкритий ключ , зазвичай, вміщується в центрі сертифікації ключів (ЦСК), де його може отримати будь-який абонент з підтвердженням цифровим підписом ЦСК.

Зашифрування повідомлення M, 1 < M < n, для A довільним користувачем B

Вважаємо, що закодоване повідомлення представлено цілим числом 1<M<n. Зашифрування виконується з використанням відкритого ключа (n,e), наприклад, користувачем B:

M^e mod n = C, C = ШТ відправляється в A.

Розшифрування ШТ в RSA. Розшифрування може бути виконано тільки користувачем A з використання секретного ключа:

C^d mod n = M

Стійкість RSA грунтується над знаходженням відкритого тексту m, враховуючи відкритий ключ RSA (e, N) і зашифрований текст c = me mod N. По суті, це проблема обернення функції RSA. Оскільки визначення фактичної складності проблеми RSA є відкритою проблемою, ми фактично покладаємося на припущення, що вона справедлива.

Зокрема, у нас є припущення RSA, яке стверджує, що проблему RSA важко вирішити, коли відкритий текст m Z_N вибирається випадковим чином, а модуль досить великий при випадково згенерованих простих числах.

Факторизація цілих чисел Іншою проблемою, яка зазвичай пов'язана з безпекою RSA, є добре відома знаходження нетривіального множника даного цілого числа.

Варто звернути увагу, що якщо модуль RSA може бути врахований, то секретна змінна d для будь-якого допустимого публічної змінної e може бути ефективно обчислена і, отже, все шифрування з використанням відкритого ключа (e, N) можуть бути розшифроване. Тобто ми можемо ефективно вирішити проблему RSA для відкритого ключа (e, N) і будь-якого допустимого зашифрованого тексту. Тому проблему RSA вирішити не складніше, ніж проблему факторизації цілих чисел.

Однак невідомо, чи вірне зворотне. Тобто невідомо, чи дозволяє рішення задачі RSA ефективно вирішувати задачу цілочисельної факторизації. Це найважливіша відкрита проблема, що стосується RSA.

Незважаючи на те, що проблему RSA насправді вирішити може бути простіше, ніж проблему цілочисельної факторизації, на практиці передбачається, що вони еквівалентні.

Оцінка складності обчислень

1) ∀(x,k) складність обчислення функції $y = x^e modn$ по схемі Горнера оцінюється в числі операцій множення за модулем n нерівністю

$L_1 \leq \ 2log_2 n$

2) При невідомих p і q для складність обернення функції RSA в загальному випадку можна записати такі ж оцінки, як для складності обернення функції Діффі-Хеллмана:

- для простого алгоритму $L_2 = O(\sqrt{n})$,

- для кращого на сьогоднішній день субекспоненйіального алгоритму $L_3 = exp(c_0 + o(1))ln^{(1/3)}n * (lnlnn)^{2/3}, c_0 \approx 1{,}923$.

При відомих p і q складність обчислення оберненої функції, як побачимо далі, оцінюється нерівністю $L_4 \leq\ 2log_2 n$. Тобто для того, хто не знає p і q функція RSA є просто односторонньою функцією, а той, хто знає p і q, може швидко обчислити і функцію RSA, і обернену функцію. Стійкість функція RSA ґрунтується на складності задачі дискретного добуванням кореня, а також на складності задачі факторизації – розкладання великих чисел на множники, оскільки знання дільників p і q в RSA дає можливість швидко обчислювати обернені значення функції. Також ще одним хорошим способом для підвищення ефективності роботи криптосистеми RSA це зменшувати довжину e. Однак це водночас є і вразливістю даної системи, оскільки побудовано дуже багато атак на RSA, які використовують малі значення відкритого ключа e. Наприклад, для e = 3 наведено декілька прикладів атак в роботах [10, 11] або [12].

У 2009 році Н. Хенінгер та Х. Шахам створили алгоритм для відновлення секретних ключів криптосистеми RSA, залежно від кількості відомих бітів, однак дані біти повинні йти послідовно та знаходитись в середині довжини числа, важлива тільки їх кількість та значення. *Алгоритм 1.2 здатен відновити секретні ключі у таких випадках* [13]:

1) Коли відомо 27 відсотків біт кожного з секретних ключів: p, q, d, d (mod p) та d (mod q).

2) Коли відомо 42 відсотків біт кожного з секретних ключів: p, q та d.

3) Коли відомо 57 відсотків біт кожного з простих чисел p та q.

Ця атака є імовірнісною і має поліноміальну складність її стійкість залежить від задачі факторизації та дискретного логарифмування.

1.2 Атаки на криптосистему RSA

Розглянемо деякі атаки на криптосистему RSA з параметрами:

– (n,e) - відкритий ключ, n=pq, p і q великі прості числа, $(e, \varphi(n)) = 1$

– d - секретний ключ, $d = e^{(-1)} mod \varphi(n)$.

– Криптоаналітику E відомі відкриті параметри (n,e) і не відомі секретні: d,p,q,$\varphi(n)$.

– Будемо вважати, що RSA побудував користувач A і тільки він має знає секретні параметри RSA.

Атака з відомою або малою різницею |p-q| [4, 5]

Нехай криптоаналітику E відома різниця |p-q|. Тоді E з квадратного рівняння відносно p+q

$4n = 4pq = (p+q)^2 - (p-q)^2$

знаходить $p + q = \sqrt{4pq + (p-q)^2}$. Далі E обчислює числа

$\frac{1}{2}(p + q + |p - q|)$,

$$\frac{1}{2}(p + q - |p - q|) \tag{1.1}$$

Одне з чисел 1.1 дорівнює p, інше q. Такою атакою криптоаналітик E зламує систему RSA.

Якщо криптоаналітику E не відома різниця |p-q|, але відомо, що вона не дуже велика, то E може спробувати знайти p і q , перебираючи можливі значення для |p-q| і для кожного виконуючи дану атаку. Вважається, для протидії такої атаці достатньо вибирати p і q такими, щоб $|p - q| > \sqrt[4]{n}$. В той же час потрібно, щоб p і q були більші за $\sqrt[4]{n}$. Саме O($\sqrt[4]{n}$) мають складності простих, але не найшвидших методів факторизації великих чисел. Рекомендують вибирати p і q порядку $p = C_1\sqrt{n}, C_2\sqrt{n}, C_1 i C_2$ деякі не рівні константи. Тоді $|p - q| = |C_1 - C_2|\sqrt{n}$.

Атака за зв'язаними повідомленнями з малим відкритим

ключем [4, 5]

Нехай користувач A побудував RSA з відкритим ключем (n,3) . Криптоаналітик E перехопив два різних шифрованих повідомлення до A

$$\begin{cases} C_1 = M_1^3 \quad modn \\ C_2 = M_2^3 \quad modn \end{cases}$$

Також E стало відомо, що повідомлення M_1 і M_2 пов'язані між собою, а саме

$$M_2 = aM_1 + b \tag{1.2}$$

де a і b відомі криптоаналітику константи. Тоді E обчислює M_1 за формулою

$\frac{b(C_2+2a^3C_1-b^3)}{a(C_2-a^3C_1+(2b)^3)} = \frac{3a^3bM_1^3+3a^2b^2M_1^2+3ab^3M_1}{(3a)^3bM_1^2+3a^2b^2M_1+3ab^3} = M_1$

І далі E знаходить M_2 за формулою 1.2.

Зокрема, для a=b=1 отримаємо

$\frac{C_2+2C_1-1}{C_2-C_1+2} = \frac{3M_1^3+3M_1^2+3M_1}{3M_1^2+3M_1+3} = M_1$

Ситуація лінійної залежності 1.2 повідомлень $M_1 i M_2$ можлива, наприклад, в випадку, коли повідомлення M_2 отримано з повідомлення M_1 дописуванням додаткової інформації.

Атака на RSA по часу обчислень [2]

Це атака з використанням інформації з побічного каналу. Нехай криптоаналітик E має можливість по електромагнітному випромінюванню з шифратора або по потужності, що споживається шифратором, визначити час виконання операцій при реалізації криптографічних алгоритмів. Так, наприклад, операція піднесення у квадрат виконується

швидше ніж операція множення. Маючи таку інформацію про піднесення у степінь $H^d \quad mod n$ при формуванні цифрового підпису або при розшифруванні $C^d mod n$ E може знайти таємний ключ d .

Нехай, наприклад, криптоаналітик E вимірює час виконання операцій в процедурі розшифруванні $C^d \quad mod n$, де піднесення у степінь виконується за схемою Горнера. Секретний ключ розшифрування записується у двійковому вигляді

$d = \sum_{i=0}^{r-1} x_i 2^i$, де $x_i \in 0{,}1$.

Розглянемо два варіанта піднесення в степінь за схемою Горнера (за зростанням і за зменшуванням):

1.$C^d \quad mod n = C^{x_0}(C^2)^{x_1}(C^{2^2})^{x_2}(C^{2^3})^{x_3} \dots (C^{2^{r-1}})^{x_{r-1}}$,

2. $C^d mod n = (\dots(((C^{x_{r-1}})^2 C^{x_{r-2}})^2 C^{x_{r-3}})^2 \dots C^{x_1})^2$.

В обох варіантах послідовність значень $x_i \in (0{,}1)$ повністю відновлюється і однозначно визначається двійкове представлення секретного ключа d .

Атака «мікрохвильовкою» [4, 5]

Нехай користувач A на шифраторі з таємним ключем d в системі RSA виконує операцію

$$y = x^d \quad mod n, n = pq \tag{1.3}$$

наприклад, $H^d \quad mod n$ при формуванні цифрового підпису або $C^d \quad mod n$ при розшифруванні.

Нехай піднесення в степінь в 1.3 виконується за китайською теоремою про лишки. Обчислюється:

$y_p = x^d \quad mod p, y_q = x^d \quad mod q$,

$$y = [y_p q(q^{-1} mod p) + y_q p(p^{-1} \quad mod q)] \quad mod n. \tag{1.4}$$

Алгоритм 1.3 атаки «мікрохвильовкою»

1) Припустимо, що після отримання числа $y = x^d \ \ mod n$, обчисленого за формулами 1.4, криптоаналітик E здійснює фізичний вплив на апаратуру шифрування або цифрового підпису (наприклад, електромагнітним випромінюванням) так, що при другому обчисленні $y = x^d \ \ mod n$ виникає помилка: замість $y_q = x^d \ \ mod q$ видано $y_q^{'}$ відповідно замість y буде

$$y^{'} = [y_p q(q^{-1} \quad mod p) + y_q^{'} p(p^{-1} \quad mod q)] \quad mod n. \tag{1.5}$$

Нехай криптоаналітик E має можливість отримати і друге число $y^{'}$, обчисленого за формулами 1.4.

2) Маючи значення y і $y^{'}$, E віднімає рівності 1.4 і 1.5 спочатку за mod q, потім mod p і отримує систему конгруенцій
$y - y^{'} = (y_q - y_q^{'}) \neq 0 \quad mod q,$

$$y - y^{'} = 0 \quad mod p \tag{1.6}$$

З системи рівностей 1.6 криптоаналітик E знаходить
$(y - y^{'}, n) = p.$

3) E обчислює $q = \frac{p}{n}$, $\varphi(n) = (p-1)(q-1)$ і знаходить секретний ключ $d = e^{-1} \quad mod \varphi(n)$.
Таким чином, атака «мікрохвильовкою» повністю зламує криптосистему RSA користувача A. Зауважимо, що отримати y і $y^{'}$, обчисленими за формулами 1.4 і 1.5, криптоаналітик може різними шляхами.

Атаки на RSA по часу обчислень і «мікрохвильовкою» відносяться до атак з використанням інформації з побічного каналу.

Атака пов'язаних повідомлень [2]

Коли два відкритих тексти, які пов'язані афінними співвідношеннями, зашифровані одним і тим же відкритим ключом, то можна відновити відкриті тексти, якщо публічна змінна малої степені і зв'язок між відкритими текстами відома. початкова атака на публічну експоненту e = 3, зроблена Франкліном і Рейтером, міститься в наступній теоремі.

Теорема 1.1. *[2] Нехай (e, N) буде дійсним публічним ключем RSA з e = 3. Нехай m_1 і m_2 будуть два повідомлення з відкритим текстом, що задовольняє $m_2 = am_1 + b$. Тоді a, b, $c_1 = m^3{}_1 \bmod N$ і публічний ключ разом з m_1 і m_2 можуть бути обчислені за поліноміальний час за log (N)).*

У наведеній вище атаці Франклін і Рейтер побудували два поліноми, $f(m_1)$ і $g(m_1) = m_1 f(m_1)$, використовуючи лише відкритий ключ, два шифротексти та явне знання зв'язку між ними. відкритих текстів (a і b). Відкритий текст m_1 легко отримати, оскільки $m_1 = g(m_1)/f(m_1) modN$. Як відомо, цей підхід можна поширити на довільну публічну експоненту e. Однак складність поліномів зростає із збільшенням значень e, і не існує відомого систематичного методу визначення цих поліномів. На додаток до побудови явних формул, як у атаці Франкліна та Райтера, Копперсміт та ін. показують, що відкритий текст зазвичай можна відновити шляхом обчислення НСД певних поліномів. Основний результат для двох пов'язаних відкритих текстів із довільною публічною експонентою наведено в наступному припущенні.

Припущення для атаки пов'язаних повідомлень 1.1 [2] *Нехай (e, N) будуть дійсним відкритим ключем RSA і нехай $m_1 i m_2$ будуть двома повідомленнями відкритого тексту, які задовольнять $m_2 = f(m_1)$, для деякого полінома f. Враховуючи поліном f,*

шифротексти $c_1 = m_1^e \mod N, c_2 = f(m_1)^e \mod N$ *і відкритий ключ, то існує імовірність, що обидва* m_1 *і* m_2 *можуть бути обчислені за поліноміальний час в e і log(N).*

Дана атака може бути узагальнена, щоб відновити будь-яку кількість пов'язаних поліномів, враховуючи їхні шифротексти та знання того, як відкриті тексти пов'язані. Основний результат з Копперсміта та інш. наведено в наступній атаці:

Припущення для атаки пов'язаних повідомлень 1.2 [2] *Нехай (e,N) є дійсним відкритим ключем RSA і* $m_1,...,m_l$ *буде l повідомлент відкритого тексту, що задовольняють поліноміальне відношення* $f(m_1,...,m_l) \equiv 0(modN)$. *Враховуючи поліном f, шифротексти* $c_1 = m_1^e modN,...,c_l = m_l^e modN$, *і відкритий ключ, то в деяких випадках відкриті тексти можуть бути обчислені за поліноміальний час в e, l і log(N).*

Подвійна RSA [2]

Розглянемо два екземпляри RSA з однаковим публічним і приватним показником, але різними модулями. Комбінуючи ключі цих двох екземплярів і видаляючи зайві експоненти, можна отримати один екземпляр подвійної RSA з відкритим ключем (e, N_1, N_2) і закритим ключем (d, p_1, q_1, p_2, q_2), коли використовується стандартне розшифрування. Припускаємо, що публічний і приватний експоненти є оберненими за модулем $\varphi(N_1)$ і за модулем $\varphi(N_2)$. Тобто вони обидва задовольняють

$ed \equiv 1(mod\varphi(N_1))$ $ed \equiv 1(mod\varphi(N_2))$

і також вони обидва задовольняють дві рівності:

$ed = 1 + k_1\varphi(N_1) = 1 + k_1(N_1 - s_1)$

$ed = 1 + k_2\varphi(N_2) = 1 + k_1(N_2 - s_2)$, де k_1 і k_2 - деякі натуральні числа.

Ці рівняння називають подвійними ключовими рівняннями RSA або просто ключовими рівняннями.

Спочатку ми розглянемо подвійний RSA з малою публічною експонентою, яка називається Dual RSA-Small-e. Як буде видно, наявність двох екземплярів RSA з однаковими невеликими публічною і однаковими приватними експонентами призводить до деяких недоліків, яких немає в RSA. Звернемо увагу, що віднімання ключових рівнянь

$ed = 1 + k_1\varphi(N_1) = 1 + k_1(N_1 - s_1)$

$ed = 1 + k_2\varphi(N_2) = 1 + k_1(N_2 - s_2),$

дає рівняння

$k_1'(N_1 - s_1) = k_2'(N_2 - s_2),$

де $k_1' = k_1/(k_1,k_2)$ і $k_2' = k_2/(k_1,k_2)$

Оскільки модулі задовольняють $1/2 \leq N_1/N_2 \leq 2$ з цього рівняння випливає, що константи в ключі рівняння задовольняють $1/2 \leq k_1/k_2 \leq 2$. Використовуючи це рівняння як відправну точку, можна показати, що публічні експоненти, менші ніж приблизно $N^{1/4}$, є небезпечними. Перший результат, який ми наведемо, показує, що константи k_1' і k_2' можна отримати, коли публічна експонента досить мала.

Нехай $(e, N_1,N_2), (d, p_1, q_1, p_2, q_2)$ буде дійсним екземпляром подвійної RSA з n-бітовими модулями та відкритою експонентою, е = $N^{(\alpha)}$. Якщо n > 14 і

$\alpha < 1/4 - log_N(18)/2,$

тоді ми можемо згенерувати список, який містить k_2/k_1, у найнижчих термінах, за поліноміальний час в log(N). Розмір списку також поліноміальний у log(N).

Після того, як константи k_1' і k_2', відомі, модулі подвійної RSA можна розкласти на множники, використовуючи результат з Sun et al.. Ми повторюємо їхній результат (і обґрунтування) у наступній атаці.

Припущення для атаки подвійної RSA 1.3 [2] Для кожного е > 0 існує таке n_0 таке, що для кожного $n > n_0$ виконується наступне: Нехай $(e,N_1,N_2),(d,p_1,q_1,p_2,q_2)$ є дійсним екземпляром подвійного RSA з n-

бітовими модулями, публічною експонентою $e = N^{\alpha} < N_{1/2}$ і приватною експонентою $d = N^{\delta}$. Нехай k_1 і k_2 — константи в ключових рівняннях, k = НСД(k_1,k_2) = N^{γ} , і нехай $k_1^{'} = k_1/k$ і $k_2 = k_2/k$. Дано $k_1^{'}$ і $k_1^{'}$, якщо

$\alpha + \beta > 1 + \gamma - \epsilon$,

тоді модулі можна розкласти на поліном часу в n за умови, що припущення 2.15 і 2.14[2] виконуються (Усі поліноми, отримані з LLL-зведених базисних векторів, є алгебраїчно незалежними та Поліноми з відомим малим розв'язком, над Z або над Z_N , мають лише один малий розв'язок відповідно.

Висновки до розділу 1

В даному розділі було детально розглянуто побудову криптосистеми RSA. Було розглянуто історію виникнення, процедури шифрування та розшифрування тексту в даній системі, деякі ключові моменти безпеки RSA, зокрема вирішення деякої задачі RSA та знаходження нетривіального множника даного цілого числа.

Також тут були розглянуті деякі атаки на RSA та наведено алгоритми для більшості наведених атак, наведено припущення для атаки пов'язаних повідомлень та припущення атаки подвійної RSA, які дозволяють працювати за поліноміальний час за певного підбору параметрів.

2 ПОБУДОВА АТАК НА КРИПТОСИСТЕМУ RSA ТА ДОСЛІДЖЕННЯ ЇХ ПАРАМЕТРІВ

В даному розділі детально розглянуто атаки Вінера та зустріч посередині, для кожної наведено алгоритм побудови, та в майбутньому реалізації. Також наведено випадки, які дозволяють послабити або посилити атаку.

2.1 Атака на RSA зустріч посередині

В даному підрозділі було використано [4, 5]. Нехай криптоаналітик E перехопив криптограму

$$y = x^e modn \tag{2.1}$$

де $x \leq \ 2^l$, l < log n або навіть $l \ll \ logn$, тобто набагато менше log n .

З великою ймовірністю число x складене. Якщо x вибрано випадково з інтервалу [1, 2^l-1] , то ймовірність того, що x складене, асимптотично дорівнює

$\frac{1}{2^l}(2^l - \frac{2^l}{ln2^l}) = 1 - \frac{1}{l*ln2} = 1 - \frac{1}{0,693*l}$

Існує ймовірність, що число $x = x_1 * x_2$, де x_1,$x_2 \leq \ 2^{\frac{l}{2}}$. При цьому $x = x_1 * x_2 \leq \ 2^l$ і виконується

$$y = x^e modn = (x_1 * x_2)^e modn = x_1^e * x_2^e modn \tag{2.2}$$

Алгоритм атаки зустріч посередині на RSA

Криптоаналітик E перехопивши повідомлення 2.1 і припускаючи, що виконується 2.2 для $x = x_1 * x_2$, де x_1,$x_2 \leq \ 2^{\frac{l}{2}}$, робить наступні кроки та обчислення.

1. Обчислює та заносить в пам’ять множину (вважаємо, що парне)

$$X = 1^e, 2^e \quad modn, 3^e \quad modn, \quad \dots \quad , (2^{l/2})^e modn \tag{2.3}$$

як впорядковану за першою координатою множину пар

$$X^{'} = (t, t^e \quad modn), \quad t = 1,2,3,\dots, 2^{l/2} \tag{2.4}$$

2. Оскільки функція RSA є бієкцією, то всі числа в множині X різні. Впорядковує множину пар 2.4 за другою координатою по зростанню чисел. Позначимо $X^{''}$ цю впорядковану за другою координатою множину пар.

3. Обчислює по порядку послідовність чисел (причому s^e mod n можна брати з множини 2.4)

$y_s = y * s^{-e} modn, s = 1,2,3,4,\dots,2^{\frac{l}{2}}$

і для кожного числа y_s (одразу після його обчислення, починаючи з y_1) шукає в множині $X^{''}$ таке пару, що друга координата пари число z задовольняє рівності

$$y_s = z \tag{2.5}$$

Якщо не знайдено таке z, яке задовольняє рівності 2.5, то обчислюється наступне число y_{s+1} і виконується пошук числа z , яке задовольняє рівності 2.5. У впорядкованій за другою координатою множині $X^{''}$ пошук такого числа або з'ясування його відсутності можна зробити в середньому за $log_2 2^{\frac{l}{2}} = \frac{l}{2}$ парних порівнянь.

Алгоритм закінчує роботу в двох випадках:

1) На 3-му кроці алгоритму для деякого s рівність 2.5. Нехай перша координата для числа z пари в $X^{''}$ дорівнює t. Тоді

$y_s = y * s^{-e} modn = t^e \quad modn,$

де t відоме криптоаналітику як перша координата пара для знайденого числа z. Останнє означає, що

$y = s^e t^e \quad modn = (st)^e \quad modn.$

Таким чином, в першому випадку знайдено відкритий текст криптограми 2.1 x = s*t і алгоритм закінчує роботу. Причому, якщо x =

$x_1 * x_2, x_1,\ x_2 \leq\ 2^{l/2}$, то алгоритм завжди знайде відкритий текст x.

2) Якщо для всіх s = 1, 2, 3, 4,. . . , $2^{l/2}$ на кроці 3 не знайдено z, для якого виконується рівність 2.5, то алгоритм закінчує роботу з відповіддю: відкритий текст x не знайдено.

Складність методу $L = O((2^{l/2} * log^3 n$ бітових операцій при обсязі необхідної пам'яті $(2^{l/2} * logn)$.

2.2 Атака Вінера з використанням ланцюгових дробів

В даному підрозділі було використано [2].Першою серйозною атакою на малу секретну експоненту RSA була ланцюгова дробова атака Вінера. Враховуючи лише відкритий ключ (e, N), атака впливає на модуль, використовуючи інформацію, отриману від одного з наближених дробів $\frac{e}{N}$. Переформулюємо атаку Вінера в її загальній формі в наступній теоремі.

Теорема 2.1. *[2] Нехай N= pq ,буде модулем RSA, нехай e буде допустимою відкритою величиною і нехай d буде його відповідно секретною, визначеною за модулем* $\lambda(N)$. *Нехай k - ціле число, для якого ed = 1 + k mod* $\lambda\ (N)$, *g = НСД(p - 1, q - 1),* $g_0 = \frac{g}{(g,k}$ *i* $k_0 = \frac{k}{(k,g}$. *Якщо таємна величина задовольняє*

$$d < \frac{pq}{(2(p+q-1)g_0 k_0} = \frac{N}{2sg_0 k_0} \tag{2.6}$$

тоді N можна розкласти за поліноміальний час в log(N) і $\frac{g}{k}$.

Доведення. [2]

Згадаємо, що модуль RSA N = p*q,

$\lambda\ (N)$ = НСК (p - 1, q - 1) = $\frac{\varphi(N)}{(p-1,q-1)} = \frac{N-s}{g}$,

де g = НСД (p - 1, q - 1). Таким чином, ключове рівняння можна записати так:

$$ed = 1 + kmod\lambda\ (N) = 1 + \frac{k}{g} * \varphi(N) = 1 + \frac{k_0}{g_0}(N - s) \tag{2.7}$$

де k деяке додатнє ціле число, $k_0 = \frac{k}{(k,g}$ і $g_0 = \frac{g}{(k,g)}$. Розділивши обидві частини цього рівняння на dN і припустивши, що секретний параметр задовольняє нерівності 2.6, отримаємо (після деякої перебудови):

$\mid \frac{e}{N} - \frac{k_0}{d{*}g_0} \mid = \mid \frac{1}{dN} - \frac{k_0 s}{dg_0 N} \mid < \frac{k_0 s}{dg_0 N} < \frac{1}{2(dg_0)^2}$,

де виконується перша нерівність, оскільки $g_0 <$ s і всі (індивідуальні) величини додатні; а друга нерівність виконується, оскільки виконується 2.6. Зокрема, оскільки

$$\mid \frac{e}{N} - \frac{k_0}{d * g_0} \mid < \frac{1}{2(dg_0)^2} \tag{2.8}$$

ми знаємо з теореми 2.2 (про ланцюгові дроби) [2], що $\frac{k_0}{dg_0}$ є одним із наближених дробів в ланцюговому дробі розширення $\frac{e}{N}$. Нехай $_i = \frac{a_i}{b_i}$ буде i-й наближений дріб з $\frac{e}{N}$, ми знаємо, що $\frac{k_0}{dg_0} = \frac{a_i}{b_i}$ для деякого j. Тепер запишемо ключове рівняння як ed = 1 + $(\frac{k_0}{g_0})\varphi(N)$, помітимо, що

$\varphi(N) =$ e$(\frac{dg_0}{k_0})$ - $\frac{g_0}{k_0}$ = [e$(\frac{b_j}{a_j})$] - $[\frac{g_0}{k_0}]$.

Таким чином, ми можемо обчислити $\varphi(N)$, якщо знаємо правильний наближений дріб c_j і можемо вгадати значення $[\frac{g_0}{k_0}]$. Ми знаємо, що ми обчислили $\varphi(N)$, коли ми можемо розкласти модуль на множники (оскільки знаючи N = pq і $\varphi(N)$ = (p - 1)(q - 1) дозволяє нам розв'язувати для p і q).

Щоб знайти правильний наближений дріб і обчислити $\varphi(N)$, ми можемо зробити наступне. Починаючи з m = 0, ми обчислюємо кандидатів на $\varphi(N)$, перебираючи наближені дроби $\frac{e}{N}$ і обчислити $\varphi(N)$' = $[\frac{e}{c_i}$ + m]. З кожним кандидатом на $\varphi(N)$ ми намагаємося розкласти модуль на множники. Якщо жоден з кандидатів не дає $\varphi(N)$ (і, отже, розкладання N), ми повторюємо процес із m, збільшеним на 1. Таким чином ми гарантовано в кінцевому підсумку обчислимо кандидата $\varphi(N)$' = $[\frac{e}{c_j}$ + $[\frac{g_0}{k_0}]$ і, отже, розклад модуля. Кожен тест (спроба розкладання

модуля з N і кандидата на $\varphi(N)$) може бути виконана за поліноміальний час в log(N). Оскільки загальна кількість дробів-кандидатів $\frac{e}{N}$ є поліноміальною від log(N), і ми перевіряємо не більше $[\frac{g_0}{k_0}] = [\frac{g}{k}]$ кандидатів для кожного наближеного дробу, ми і маємо поданий результат. □

Метод отримання правильного наближеного дробу $\frac{k}{d}$ в цьому доказі не є оптимальним. Наприклад, якщо $|\frac{e}{N} - c_i|$ занадто великий, ми можемо (виходячи з $|\frac{e}{N} - \frac{k_0}{d*g_0}|$ з доведення теореми 1.2 [2]) проігнорувати його. Насправді, наближені дроби можна звузити до дуже маленької множини. Крім того, коли прості числа RSA вибираються випадковим чином, з високою ймовірністю очікується, що g буде дуже малим. Тому очікується, що $\lfloor\frac{g}{k}\rfloor = 0$, і насправді потрібна лише одна ітерація збіжних кандидатів. Майже в усіх виведеннях атаки Вінера передбачається, що $\lfloor\frac{g}{k}\rfloor = 0$ (або $\frac{g}{k}$ < 1) у певний момент (включаючи вихідне виведення). Достатня умова в теоремі 1.2 [2] не є звичайною умовою, яка в більшості асоціюється з атакою Вінера. Більш поширений стан коли,

$d < \frac{1}{c}N^{\frac{1}{4}}$

для деякої невеликої константи c > 1, виходить, якщо припустити, що публічна експонента має приблизно такий самий розмір, як і модуль, що прості числа збалансовані і g_0 є малим. У типовому прикладі RSA із випадково згенерованими простими числами та невеликою приватною експонентою ці припущення справедливі. Ця межа (грубо кажучи d $<$ $N^{\frac{1}{4}}$) є орієнтиром для атаки Вінера (і іноді її називають межею Вінера).

Коли використовується нетиповий екземпляр RSA, атака Вінера може бути ослаблена або посилена (порівняно з межею $N^{\frac{1}{4}}$. Розглянемо загальну достатню умову з теореми 1.2 [2],

$d < \frac{pq}{2(p+q-1)g_0k_0} = \frac{N}{2sg_0k_0}$

Є три способи, якими можна зменшити межу приватної експоненти (і, отже, послабити атаку):

1) Використовувати незбалансовані прості числа, щоб s = p + q - 1 ставало більшим.

2) Використовувати прості числа з великим g = НСД(p - 1, q - 1), щоб (імовірно) g_0 також ставало більшим.

3) Використовувати публічну експоненту e > N, щоб $k \approx \frac{ed}{N}$ (і, ймовірно, k_0) стає більше. Публічну експоненту, більшу за N, можна отримати, просто додавши кратне λ (N) до існуючої (нормальної) публічної експоненти. Фактично атака Вінера стає абсолютно неефективною, коли $e > N^{\frac{3}{2}}$.

Останній із методів послаблення атаки Вінера працює в обох напрямках. Більші публічні експоненти послаблюють атаку, тоді як менші публічні експоненти посилюють її. Розглянемо RSA із збалансованими простими числами, малим g_0 , малою приватною експонентою $d = N^{\delta} < N^{1/2}$ та публічною експонентою $e = N^{\alpha}$ для деяких $1/2 < \alpha < 1$. Оскільки $ed = 1 + k\lambda(N)$, ми маємо $k \approx N^{\alpha+\delta-1}$. Підставляючи їх у загальну достатню умову для атаки Вінера та ігноруючи малі константи, отримаємо $N^{\delta} < N^{1-1/2-(\alpha+\delta-1)-\epsilon}$, або простіше

$\delta < \frac{3}{4} - \frac{\alpha}{2} - \epsilon,$

де $\epsilon > 0$ — невелика константа, яка враховує будь-які малі фактори, які були проігноровані. У типовому випадку $e \approx N$ ми маємо $\alpha \approx 1$ і $\delta < 1/4$, як і очікувалося. Для більших публічних експонентів межа δ зменшується до тих пір, поки не зникне в нуль при $\alpha = 3/2$, після чого атака не може нічого гарантувати, а для менших публічних експонентів межа збільшується до $\delta < 1/2$, коли $\alpha = 1/2$.

Розглянемо алгоритм атаки Вінера, але дещо в інших позначеннях:

Алгоритм атаки Вінера:

Припустимо, що у нас є відкритий ключ (n,e) , ця атака визначає p та q:

1) Перетворити дріб $\frac{e}{n}$ у ланцюговий дріб [$a_0; a_1; a_2; ...a_{k-1}; a_k$];

2) вирахувати кожен дріб із ланцюгового дробу $\frac{a_0}{1}$, $a_0 + \frac{1}{a_1}$, $a_0 + \frac{1}{a_1 + \frac{1}{a_2}}$,

..., $a_0 + \frac{1}{a_1 + ... \frac{1}{a_{k-2} + \frac{1}{a_k - 1}}}$;

3) перевірити кожен дріб із кроку 2 наступним чином:

а) встановити чисельник дробу = k, знаменник = d;

б) перевірити чи d непарне, якщо ні, то взяти інший дріб;

в) перевірити чи ed $\equiv$ 1 mod k, якщо ні, то взяти інший дріб;

г) обрахувати $\varphi(n) = \frac{ed-1}{k}$ і знайти корені x^2 - (n - $\varphi(n)$ + 1)x + n;

д) Якщо корені поліному цілі, то ми знайшли p і q. Звідси можна визначити d;

4) якщо всі дроби були перевірені, і жоден з них не задовольняє всі умови, то дані параметри RSA не вразливі для атаки Вінера.

Зв'язок алгоритму Евкліда з ланцюговими дробами [7]

Оскільки в нас на першому кроці алгоритму розглядаються ланцюгові дроби, які будуються за допомогою алгоритму Евкліда, то розглянемо зв'язок алгоритму Евкліда з ланцюговими дробами:

1) Нехай α – будь-яке дійсне число. Позначимо буквою q_1 найбільше ціле, яке не перевищує α. При не цілому α маємо:

$\alpha = q_1 + \frac{1}{\alpha_2}$; $\alpha_2 > 1$.

Точно так само при не цілих α_2, ..., α_{s-1} маємо

$\alpha_2 = q_2 + \frac{1}{\alpha_3}$; $\alpha_3 > 1$;

. . . .

$\alpha_{s-1} = q_{s-1} + \frac{1}{\alpha_s}$; $\alpha_s > 1$,

внаслідок чого отримуємо наступний розклад α в ланцюговий дріб:

$$\alpha = q_1 + \frac{1}{q_2 + \frac{1}{q_3 + ... + \frac{1}{q_{s-1} + \frac{1}{\alpha_s}}}} \tag{2.9}$$

Якщо α ірраціональне, то в ряді α, α_2, ..., очевидно, не може зустрітись цілих і вказаний процес може нескінченно продовжуватись.

Якщо α раціональне, то як побачимо у наступному пункті в ряді α, α_2, ... в будь-якому випадку зустрінеться ціле число і вказаний процес буде

завершений.

2) Якщо α – раціональний нескоротний дріб $\alpha = \frac{a}{b}$, то розклад α в неперервний дріб тісно пов'язано з алгоритмом Евкліда. Дійсно, маємо:

a = b$q_1 + r_1$; $\quad \frac{a}{b} = q_1 + \frac{r_2}{b}$,

b = $r_2 * q_2 + r_3$; $\quad \frac{b}{r_2} = q_2 + \frac{r_3}{r_2}$,

$r_2 = r_3 * q_3 + r_4$; $\quad \frac{r_2}{r_3} = q_3 + \frac{r_4}{r_3}$,

. . . .

$r_{n-2} = r_{n-1} * q_{n-1} + r_n$; $\quad \frac{r_{n-2}}{r_{n-1}} = q_{n-1} + \frac{r_n}{r_{n-1}}$, $r_{n-1} = r_n * q_n$; $\quad \frac{r_{n-1}}{r_n} = q_n$,

звідки

$$\frac{a}{b} = q_1 + \frac{1}{q_2+\frac{1}{q_3+\ldots+\frac{1}{q_n}}}.$$

3) Числа $q_1, q_2,$ ... , які використані для розкладу числа α в ланцюговий дріб, називаються *неповними частковими* (у випадку раціонального α це буде згідно пункту 2) неповні часткові послідовних ділень алгоритму Евкліда), дроби ж

$$\delta_1 = q_1, \qquad \delta_2 = q_1 + \frac{1}{q_2}, \qquad \delta_3 = q_1 + \frac{1}{q_2+\frac{1}{q_3}}, \ldots$$

називаються *наближеними дробами.*

4) досить просте правило утворення ланцюгових дробів легко знайдемо, помітивши, що δ_s (s > 1) виходить з δ_{s-1} заміною в буквеному виразі для δ_{s-1} числа q_{s-1} на $q_{s-1} + \frac{1}{q_s}$.

Дійсно, припускаючи $P_0 = 1$, $Q_0 = 0$, ми можемо наближеними дробами послідовно представити в наступному вигляді (тут рівність $\frac{A}{B} = \frac{P_s}{Q_s}$ пишемо, бажаючи позначити A символом P_s, а B символом Q_s):

$$\delta_1 = \frac{q_1}{1} = \frac{P_1}{Q_1}, \quad \delta_2 = \frac{q_1+\frac{1}{q_2}}{1} = \frac{q_2 q_1+1}{q_2 * 1+0} = \frac{q_2 P_1+P_0}{q_2 Q_1+Q_0} = \frac{P_2}{Q_2},$$

$$\delta_3 = \frac{(q_2+\frac{1}{q_3})P_1+P_0}{(q_2+\frac{1}{q_3})Q_1+Q_0} = \frac{q_3 P_2+P_1}{q_3 Q_2+Q_1} = \frac{P_3}{Q_3}$$

і т.д., а в загальному:

$$\delta_s = \frac{q_s P_{s-1}+P_{s-2}}{q_s Q_{s-1}+Q_{s-2}} = \frac{P_s}{Q_s}.$$

Таким чином чисельники і знаменники наближених дробів ми можемо послідовно вираховувати по формулах:

$$\begin{cases} P_s = q_s P_{s-1} + P_{s-2}, \\ Q_s = q_s Q_{s-1} + Q_{s-2}. \end{cases} \quad (2.10)$$

5) Розглянемо різницю $\delta_s - \delta_{s-1}$ наближених дробів. При s $>$ 1 знаходимо
$\delta_s - \delta_{s-1} = \frac{P_s}{Q_s} - \frac{P_{s-1}}{Q_{s-1}} = \frac{h_s}{Q_s Q_{s-1}}$,
де $h_s = P_s Q_{s-1} - Q_s P_{s-1}$; підставляючи ж замість P_s і Q_s їх вирази з 2.10 і виконавши певні спрощення, отримаємо $h_s = -h_{s-1}$. Останнє, поєднане з $h_1 = q_1 * 0$ - 1*1 = -1 дає $h_s = (-1)^s$. Отже,

$$P_s Q_{s-1} - Q_s P_{s-1} = (-1)^s = (-1)^s, \qquad (s > 0) \quad (2.11)$$

$$\delta_s - \delta_{s-1} = \frac{(-1)^s}{Q_s Q_{s-1}}, \qquad (s > 1) \quad (2.12)$$

6) З 2.11 випливає, що (P_s, Q_s) ділить $(-1)^s = \pm 1$. Тому $(P_s, Q_s) =$ 1, тобто *наближені дроби* $\frac{P_s}{Q_s}$ *нескоротні.*

7) При δ_s, яке не дорівнює α (тобто виключається випадок, коли при раціональному α, δ_s є останнім наближеним дробом), досліджуємо різницю δ_s - α. Очевидно, δ_s виходить внаслідок заміни α_s на q_s у виразі 2.9 для α. Але як видно з першого пункту, від такої заміни
α_s зменшиться,
α_{s-1} збільшиться,
α_{s-2} зменшиться,
....
α при непарному s зменшиться,
α при парному s збільшиться.
Тому δ_s - $\alpha < 0$ при непарному s і δ_s - $\alpha > 0$ при парному s, і як наслідок, знак δ_s - α співпадає з знаком $(-1)^s$

8) Маємо
$\mid \alpha - \delta_{s-1} \mid \leq \frac{1}{Q_s Q s-1}$.

Дійсно, при $\delta_s = \alpha$ це твердження випливає (зі знаком дорівнє) з 2.12. При δ_s, не рівному α воно слідує (із знаком нерівності) з 2.12 і з тієї обставини, що ввиду g, δ_s - α і δ_{s-1} - α мають різні знаки.

Висновки до розділу 2

В даному розділі було розглянуто детально атаки на криптосистему RSA як атака зустріч посередині та атака Вінера. В атаці зустріч посередині наведено алгоритм роботи даної атаки та досліджено умови за яких вона буде працювати. В атаці Вінера було детально розглянуто її алгоритм побудови, використання алгоритму Евкліда та зв'язок між даним алгоритмом та ланюговими дробами, які використовуються в атаці та параметри підбору відповідного наближеного дробу. Також наведено три способи для послаблення даної атаки.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ АТАК ТА ЇХ ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

В даному розділі наводяться практичні реалізації та оцінки складності та ймовірності успіху для атаки зустріч посередині та оцінки складності атаки Вінера експериментальні та теоретичні і відповідно вибір параметрів для кожної з атак.

3.1 Теоретична оцінка складності атаки Вінера

Складність в найгіршому випадку

Дана складність складається із кількох складових: обчислення ланцюгового дробу (що вносить найбільшу вагу в даний дану складність), обрахування даних дробів згідно рекурентних співвідношень, які вивів Ейлер, кожна з яких має по одній операції множення та одній операції додавання та перевірки чи задовольняють кандидати-дроби певні умови.

В [6] показано, що кількість ділень з лишком в алгоритмі Евкліда, що використовується для побудови ланцюгових дробів дорівнює $2 + [log_\delta x_2]$, $x_1 > x_2 > 0$, де x_1, x_2 – числа з алгоритму Евкліда, δ – додатній корень квадратного рівняння $x^2 - x - 1 = 0$ Також звідси візьмемо і складність ділення з остачею, що дорівнює m(k - m +1), де k – кількість розрядів у числі x_1, m – кількість розрядів у числі x_2, $x_1 > x_2 > 0$, k $\geq$ m. Отже, складність побудови ланцюгових дробів дорівнює:

$T_1 = (2 + [\log_\delta x_2])$ * m(k - m + 1).

T_2 розраховується із рекурентних формул виведених Ейлером для розрахунку чисельника і знаменник ланцюгових дробів:

$p_{-1} = 1, \quad p_0 = a_0, p_n = a_n * p_{n-1} + p_{n-2}$

$q_{-1} = 0, \quad q_0 = 1, \quad q_n = a_n * q_{n-1} + q_{n-2}$

Як бачимо, в них використовується за один раз по одному додаванні

та одному множенню. Кількість застосувань даних рекурент дорівнюватиме числу ділень в алгоритмі Евкліда помножене на 2.

Кількість множень дорівнюватиме складності множення в алгоритмі Карацуби [8]: $k^{log_2 3}$, де k – кількість розрядів у чисел, які перемножуються.

T_2 = 2*((2 + [$\log_\delta x_2$]))*($k^{log_2 3}$ + (2k - 1))

Згідно з алгоритмом атаки Вінера на третьому кроці: ми маємо 4 операції множення (одна з кроку 3 в), одна з 3 г) і ще дві з 3 г) для обрахування дискримінанту), 3 операції ділення (одна з 3 г) та дві з 3 г) для обрахування коренів кв. рівняння) та 6 операцій додавання або віднімання (одна з обрахування $\varphi(n)$, дві з кв рівняння, одна при обрахуванні дискримінанту і дві для обрахування коренів кв. рівняння).

T_3 = ($4k^{\log_2 3}$ + 3*m(k - m + 1) + 6(2k - 1))

Отже, загальна складність дорівнює:

T = T_1 + T_2 + T_3 = (2 + [$\log_\delta x_2$]) * m(k - m + 1) + 2*(2 + [$\log_\delta x_2$])*($k^{log_2 3}$ + (2k - 1)) + ($4k^{\log_2 3}$ + 3*m(k - m + 1) + 6(2k - 1))

3.2 Експериментальна оцінка складності та ймовірності успіху атаки зустріч посередині

Криптосистема RSA та дана атака була побудована на мові програмування Java, було побудовано для числа n = 512 біт та l = 40 біт.

Для дослідження складності роботи було вибрано параметр e з заданими значеннями від 4 до 1024. Нижня межа вибрана тому що, коли довжина e дорівнює 2 біти, то існує лише 2 можливих варіанти для e. Це e = 2 та e = 3. Значення e = 2 не підходить за замовчуванням. Якщо розглядати значення e = 3, то інколи трапляється, що функція Ейлера ділиться на 3. У такому випадку e та функція Ейлера не є взаємно прості постійно, тому ми генеруємо нове значення знову і знову до нескінченності. 1024 біт не були вибрані тому що для моєї генерації n =

512 біт і відповідно функція Ейлера це 510 біт. Оскільки $1 < e < \varphi(n)$, а в моєму випадку $e > \varphi(n)$, то d не може правильно згенеруватись.

Для отримання усередненої складності роботи атаки залежно від e пройдено 5 разів по однаковій довжині e та по 2 рази кожному індивідуальному значенню e.

Результати наведено у таблиці 3.1 та проілюстровані на рис. 3.1. Як бачимо з графіка, складність зростає прямо пропорційно.

Таблиця 3.1 – Залежність часу роботи атаки від довжини числа e

Довжина числа e, біт	Час роботи, с
4	9,2729
16	14,3643
32	21,7638
32	36,6188
64	59,3784
128	113,8636
256	212,8819
512	462,2989

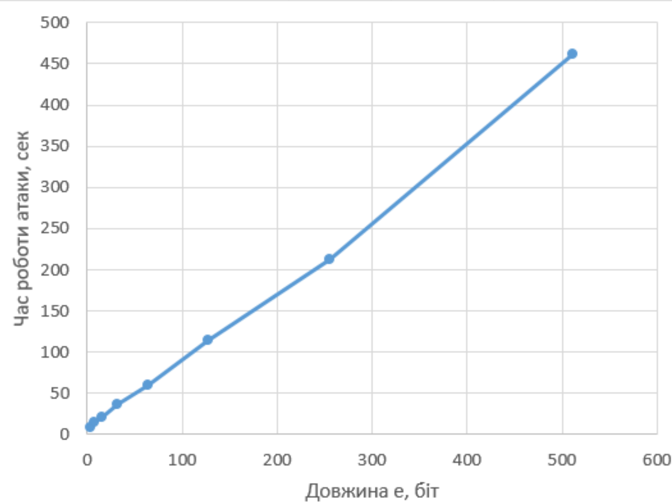

Рисунок 3.1 – Результати залежності часу роботи атаки від довжини e

Програмна реалізація для визначення оцінки ймовірності успіху атаки була написана на мові програмування Java. Вона була проведена для довжини l = 40 біт. Проходження по l відбувалось від l = 25 біт до l = 48 біт з кроком в 2 біти. Якщо ж вибрати l менше за 25 біт, то атака спрацює щоразу, оскільки обидва множники, на яке розкладається відкрите повідомлення будуть будуть не більші за $2^{\frac{l}{2}}$. Якщо ж вибрати довжину випадково згенерованого числа більше за 48, то ймовірність успіху роботи атаки буде дорівнювати нулю, оскільки M > l. Генерація випадково числа M відбувалась за допомогою вбудованого генератора випадкових чисел для мови програмування Java. Було проведено 100 експериментів для кожного параметра l, що відповідає за довжину відкритого повідомлення. Числа p і q використовувались щоразу одні й ті, які мають довжину по 256 біт кожне, отже n матиме довжину 512 біт. За-для пришвидшення роботи атаки, було вибрано одне значення e, яке дорівнє 10001.

Результати наведено у таблиці 3.2 та проілюстровані на рис. 3.2.

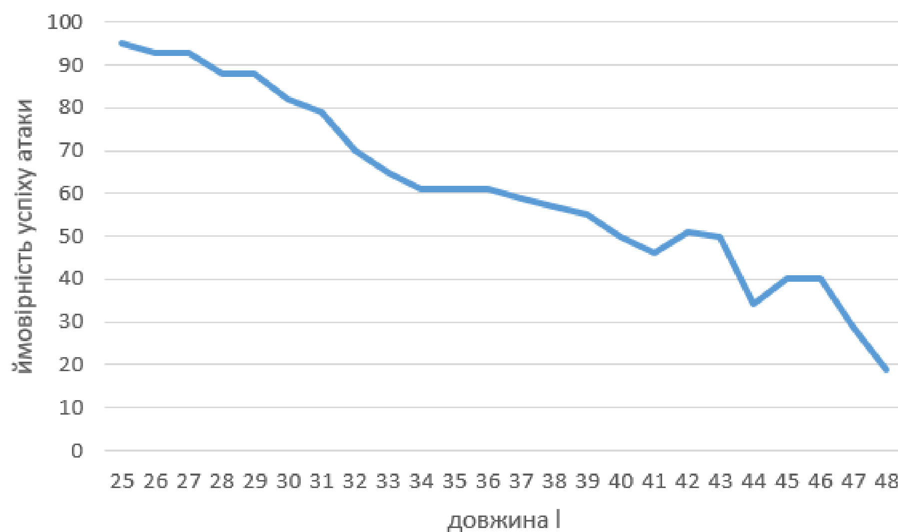

Рисунок 3.2 – Результати залежності ймовірності успіху атаки від довжини l

Таблиця 3.2 – Результати дослідження ймовірності успіху атаки від довжини l

Довжина числа l, біт	Ймовірність успіху, проц.
25	95
27	93
29	88
31	79
33	65
35	61
37	59
39	55
41	46
43	50
45	49
47	29

3.3 Експериментальна оцінка складності атаки Вінера

Криптосистема RSA та дана атака була побудована на мові програмування Java, було побудовано для числа n до 2048 біт.

Для дослідження складності роботи було вибрано довжину числа n від 32 біт до 2048 біт. Нижня межа вибрана тому що, коли довжина n дорівнює 16 біт, то накладене нами обмеження на d: $\frac{\sqrt[4]{n}}{\sqrt{6}} = 4$, тобто $d = 2$ або $d = 3$. Але для цих двох випадків d не взаємно просте з функцією Ейлера, тому ми постійно відкидаємо d і генеруємо нове і так до нескінченності. А для $n = 8$ біт завжди $\frac{\sqrt[4]{n}}{\sqrt{6}} = 1$, тому для 8 біт взагалі не реально знайти d.

Для отримання усередненої складності роботи атаки залежно від n пройдено 500 разів по однаковій довжині n (і відповідно p та q) та по 100 разів кожному індивідуальному значенню e.

Результати наведено у таблиці 3.3 та проілюстровані на рис. 3.3. Як бачимо з графіка, складність зростає прямо пропорційно.

Таблиця 3.3 – Результати залежності часу роботи атаки від довжини числа n

Довжина числа n, біт	Час роботи, мкс
32	16,140992
64	41,305624
128	100,92379
256	200,86394
512	459,278204
1024	1291,800294
2048	4346,155524
3072	7835,627106

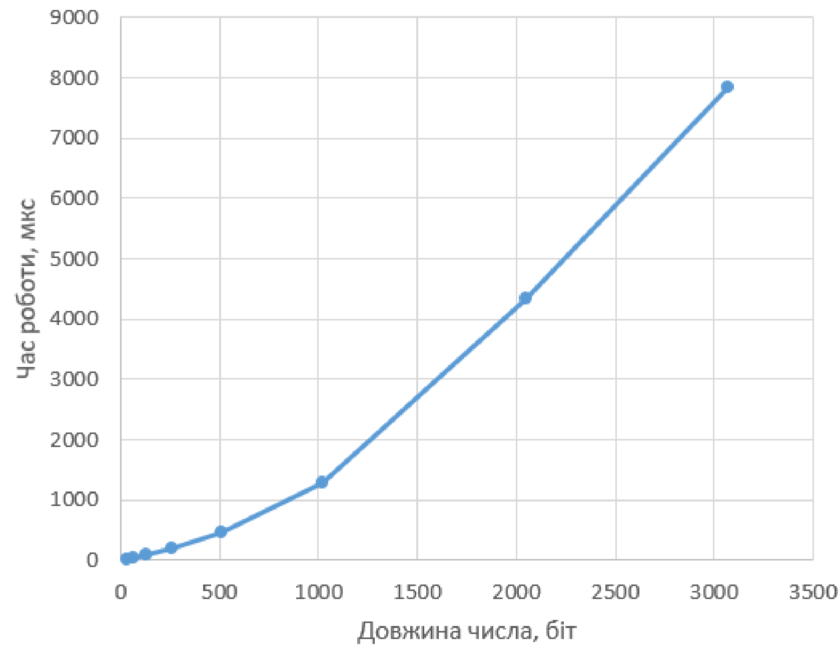

Рисунок 3.3 – Результати залежності часу роботи атаки від довжини n

Висновки до розділу 3

Отже, в даному розділі були побудовані програмні реалізації атак Вінера та зустрічі посередині на криптосистему RSA, експериментально досліджено складність роботи залежно від e для атаки зустріч посередині та від n для атаки Вінера і показано що вони мають пряму залежність від даних параметрів, що формують графіки типу $f(x) = x$. Також було побудовано теоретично оцінки складності в найгіршому випадках для атаки Вінера, а також обчислено як змінюється ймовірність успіху атаки зустріч посередині залежно від довжини та значення l та показано що зі зростанням l ймовірність атаки зменшується.

ВИСНОВКИ

У ході даної роботи був проведений аналіз опублікованих джерел за тематикою, що містить опис про криптосистему RSA та деяких атак на дану криптосистему. Було досліджено грунтовніше та побудовані програмні реалізації для атаки зустріч посередині та атаки Вінера.

Для атаки зустріч посередині було уточнено ймовірність того наскільки успішно проходить атака залежно від довжини параметра l, та відповідної можливості розкладу числа M на множники, x_1 та x_2, де x_1, $x_2 \leq \ 2^{\frac{l}{2}}$, було побудовано відповідний графік залежності. Також було показано, що складність роботи даної атаки залежно від довжини та, власне, значення відкритого параметру e зростає прямо пропорційно та наведено відповідний графік.

Для атаки Вінера було показано, що складність роботи даної атаки залежно від відкритого параметру n зростає прямо пропорційно та наведено відповідний графік. Також було підтверджено високу ефективність роботи атаки Вінера, якщо параметр d вибрано $\leq \ \frac{\sqrt[4]{n}}{\sqrt{6}}$ та відповідно, нестійкість RSA при такому параметрі. Отже, розроблена реалізація дозволяє підвищити якість вибору параметрів для побудови криптосистеми RSA. Більше того, теоретично обчислено залежність складності від розміру параметрів в найгіршому випадку.

ПЕРЕЛІК ПОСИЛАНЬ

1. Song Y.Yan.Cryptanalytic Attacks on RSA.- Springer.- 2008. - 255 p.

2. Hinek, M. Jason. Cryptanalysis of RSA and its variants. -2010. – 268.

3. Грушо А.А., Тимонина Е.Е., Применко Е.А. Анализ и синтез криптоалгоритмов. Курс лекций. Йошкар-Ола, изд. МФ МОСУ, 2000. – 112с

4. Мао, Венбо. Современная криптография: теория и практика.: Пер. с англ. – М.: Издательский дом «Вильямс», 2005. – 768 с.

5. Тилборг Ван Х.К.А. Основы криптологии. Профессиональное руководство и интерактивный учебник. - М.: Мир, 2006. – 471 с.

6. Глухов М.М., Круглов И.А., Пичкур А.Б., Черёмушкин А.В. Введение в теоретико-числовые методы криптографии. Учебное пособие. — СПб.: Лань. 2011. — 400 с. — ISBN: 978-5-8114-1116-0

7. Виноградов І. М. "Основи теорії чисел". М.-Л., Гостехиздат, 1952. 180 с.

8. Белов А.Я., Тихомиров В.М. "Сложность алгоритмов". Квант 1999/№2

9. Whitfield Diffie, Martin E. Hellman New directions in cryptography. IEEE Transactions on Information Theory (Volume: 22, Issue: 6, Nov 1976) - 644 - 654 с.

10. Hastad J. Solving Simultaneous Modular Equations of Low Degree / Johan Hastad // SIAM J. Comput. – 1988. – Vol. 17. – P. 336–341.

11. Franklin M., Reiter M. A linear protocol failure for RSA with exponent three. 1995 //Rump Session of Crypto. – Vol. 95.

12. Coppersmith D. et al. Low-exponent RSA with related messages / Don Coppersmith, Matthew Franklin, Jacques Patarin, Michael Reiter //International Conference on the Theory and Applications of Cryptographic Techniques. – Springer, Berlin, Heidelberg, 1996. – P. 1-9.

13. Heninger N., Shacham H. Reconstructing RSA private keys from random key bits //Annual International Cryptology Conference. – Springer, Berlin, Heidelberg, 2009. – P. 1-17.

ДОДАТОК А ТЕКСТИ ПРОГРАМ

А.1 Атака зустріч посередині

```
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.Random;

public class RSAAttack {

        ////////////////////////////////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////////////////////////////////
    // Prime random numbers generation
        ////////////////////////////////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////////////////////////////////

        static Random gRandomGen = new Random();

        public static BigInteger generateRandomNumber(int numBits) {
// Constructs a randomly generated BigInteger, uniformly distributed over the range 0 to (2^numBits - 1), inclusive.
                return new BigInteger(numBits, 10, gRandomGen);
        }

        // Test Miller-Rabin
    public static boolean testPrimeNumber(BigInteger p) {
        // step 0
        BigInteger pMinus1 = p.subtract(BigInteger.ONE);
         pMinus1.toString());
        // find s
        int s = 0;
        // divide by 2
        while (pMinus1.mod(BigInteger.TWO).equals(BigInteger.ZERO)) {
                s++;
                pMinus1 = pMinus1.divide(BigInteger.TWO);
        }
        BigInteger d = pMinus1;
        pMinus1 = p.subtract(BigInteger.ONE); // refresh value

        for (int k = 1; k < 10; k++) {
                // step 1
                BigInteger x = generateRandomNumber(p.bitLength());

                if (x.equals(BigInteger.ZERO) || x.equals(BigInteger.ONE) || x.equals(pMinus1)) {
```

```
                continue;
            }

            BigInteger resGcd = x.gcd(p);
            if (!resGcd.equals(BigInteger.ONE)) {
                return false;
            }

            // step 2
            BigInteger x_r = x.modPow(d, p);
            // step 2.1
            if (x_r.equals(BigInteger.ONE) || x_r.equals(pMinus1)) {

            } else {
                // step 2.2
                for (int r = 1; r < s; r++) {
                    x_r = x_r.modPow(BigInteger.TWO, p);

                    if (x_r.equals(pMinus1)) {
                        continue;
                    }

                    if (x_r.equals(BigInteger.ONE)) {
                        return false;
                    }
                }
                return false;
            }

        }

        return true;
    }

    public static BigInteger generatePrimeNumber(int numBits) {
        BigInteger newRndNumber = BigInteger.TWO; // just to avoid errors - set NOT prime number
        boolean isPrime = false;

        while (isPrime == false) {
            newRndNumber = generateRandomNumber(numBits);
            isPrime = testPrimeNumber(newRndNumber);
        }
        return newRndNumber;
    }
```

```java
    ////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////
// RSA functions
    ////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////


 public static ArrayList<BigInteger> GenerateKeyPair(BigInteger p, BigInteger q, int e_lenght) {
        BigInteger n = p.multiply(q);
        BigInteger funOylera = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
        BigInteger e = null;
        boolean genE = false;
        while (genE == false) {
                e = generatePrimeNumber(e_lenght);
if (e.compareTo(BigInteger.ONE) == 1 && e.compareTo(funOylera) == -1 && e.gcd(funOylera).equals(BigInteger.ONE) == true)
                        genE = true;
                } else {
                        continue;
                }
        }
        BigInteger d = e.modInverse(funOylera);

        ArrayList<BigInteger> keys = new ArrayList<BigInteger>();
        keys.add(n); // index 0 - public Key
        keys.add(e); // index 1 - public Key
        keys.add(d); // index 2 - private Key
        return keys;
}

public static BigInteger Encrypt(BigInteger M, BigInteger pubKeyE, BigInteger n) {
    return M.modPow(pubKeyE, n);
}

public static BigInteger Decrypt(BigInteger C, BigInteger privKeyD, BigInteger n) {
    return C.modPow(privKeyD, n);
}

public static void meet_in_middle(BigInteger e, BigInteger n, BigInteger C) {
            int l = 256 - 216;
            HashMap<BigInteger,BigInteger> X_st = new HashMap<BigInteger,BigInteger>();
            HashMap<BigInteger,BigInteger> X_st2 = new HashMap<BigInteger,BigInteger>();
            BigInteger end = BigInteger.TWO.pow(l/2).add(BigInteger.ONE);
            BigInteger start = BigInteger.ONE;

            while (!start.equals(end)) {
```

```
                BigInteger t = start;
                X_st.put(t, t.modPow(e, n));
                X_st2.put(t.modPow(e, n), t);
                start = start.add(BigInteger.ONE);
        }
        System.out.println("Attack: X' i X'' created ");

        BigInteger y_s = null;
        start = BigInteger.ONE;
        while (!start.equals(end)) {
                BigInteger s_e = X_st.get(start);
                y_s = s_e.modInverse(n);
                y_s = C.multiply(y_s).mod(n);

                if (X_st2.containsKey(y_s)) {
                        BigInteger C_x_1 = start;
                        BigInteger C_x_2 = X_st2.get(y_s);
                        System.out.println("Attack: s(x1) = " + C_x_1.toString(16));
                        System.out.println("Attack: t(x2) = " + C_x_2.toString(16));
        System.out.println("Attack: y = (st)^e mod n = " + C_x_1.multiply(C_x_2).modPow(e, n).toString(16));
                        break;
                }
                start = start.add(BigInteger.ONE);
        }
        if (start.equals(end)) {
                System.out.println("Attack: s, t not found");
        }

        System.out.println("Attack: done");
}

    public static void main(String[] args) {


        // Generate p,q
        BigInteger p, q;
        while (true) {
        p = generatePrimeNumber(256);
        q = generatePrimeNumber(256);
        int temp = q.compareTo(p);
        int temp2 = p.compareTo(BigInteger.TWO.multiply(q));
                if (temp == -1 | temp2 == -1) {
                        break;
                }
        }
```

```
        for (int e_lenght = 4; e_lenght < 1025; e_lenght = e_lenght*2) {
            double average_one_lenght_count = 0;
             for (int j = 0; j < 5; j++) {
                    ArrayList<BigInteger> keys = GenerateKeyPair(p, q, e_lenght);
                    BigInteger pubKey_n = keys.get(0);
                    BigInteger pubKey_e = keys.get(1);
                    BigInteger privKey_d = keys.get(2);

                    System.out.println("");
                    System.out.println("Test Encrypt / Decrypt");
                    BigInteger x_1 = new BigInteger("7B8B", 16);
                    BigInteger x_2 = new BigInteger("CEB7", 16);
                    BigInteger M = x_1.multiply(x_2);
                    BigInteger C = Encrypt(M, pubKey_e, pubKey_n);
                    BigInteger M_dec = Decrypt(C, privKey_d, pubKey_n);

                    System.out.println("RSA: x_1  = " + x_1.toString(16));
                    System.out.println("RSA: x_2  = " + x_2.toString(16));
                    System.out.println("RSA: open text (x_1*x_2)  = " + M.toString(16));
                    System.out.println("RSA: ciphertext  = " + C.toString(16));
                    System.out.println("RSA: decrypted text  = " + M_dec.toString(16));

                    double count = 0;
                    for (int i = 0; i < 2; i++ ) {
                        long m = System.currentTimeMillis();
                        meet_in_middle(pubKey_e, pubKey_n, C);
                        System.out.println("Time of one work in miliseconds: ");
                        System.out.println((double) (System.currentTimeMillis() - m));
                        count = count + System.currentTimeMillis() - m;
                    }
                    double average_personal = count/2;
    System.out.println("Average time of attack for personal e in miliseconds: " + average_personal);
                    average_one_lenght_count  = average_one_lenght_count + average_personal;
             }
             average_one_lenght_count = average_one_lenght_count/5;
System.out.println("Average time of attack for one lenght e: " + e_lenght + " in miliseconds: "
+ average_one_lenght_count);
        }
    }

}
```

A.2 Атака Вінера

```
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.Random;

public class Second_attack {

        ////////////////////////////////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////////////////////////////////
    // Prime random numbers generation
        ////////////////////////////////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////////////////////////////////

        static Random gRandomGen = new Random();

        public static BigInteger generateRandomNumber(int numBits) {
// Constructs a randomly generated BigInteger, uniformly distributed over the range 0 to (2^numBits - 1), inclusive.
                return new BigInteger(numBits, 10, gRandomGen);
        }

        // Test Miller-Rabin
    public static boolean testPrimeNumber(BigInteger p) {
        // step 0
        BigInteger pMinus1 = p.subtract(BigInteger.ONE);

        // find s
        int s = 0;
        // divide by 2
        while (pMinus1.mod(BigInteger.TWO).equals(BigInteger.ZERO)) {
                s++;
                pMinus1 = pMinus1.divide(BigInteger.TWO);
        }
        BigInteger d = pMinus1;

        pMinus1 = p.subtract(BigInteger.ONE); // refresh value

        for (int k = 1; k < 10; k++) {
                // step 1
                BigInteger x = generateRandomNumber(p.bitLength());

            if (x.equals(BigInteger.ZERO) || x.equals(BigInteger.ONE) || x.equals(pMinus1) || x.compareTo(p) >= 0) {
```

```
                System.out.println("Test Miller-Rabin 1: bad number - generate one more time");
                continue;
            }

            BigInteger resGcd = x.gcd(p);
            if (!resGcd.equals(BigInteger.ONE)) {
                System.out.println("Test Miller-Rabin 1: number failed - not prime");
                return false;
            }

            // step 2
            ///System.out.println("Test Miller-Rabin: step 2");
            BigInteger x_r = x.modPow(d, p);
            // step 2.1
            if (x_r.equals(BigInteger.ONE) || x_r.equals(pMinus1)) {

            } else {
                // step 2.2
                for (int r = 1; r < s; r++) {
                    x_r = x_r.modPow(BigInteger.TWO, p);

                    if (x_r.equals(pMinus1)) {
                        continue;
                    }

                    if (x_r.equals(BigInteger.ONE)) {
                        return false;
                    }
                }
                return false;
            }

        }

        return true;
    }
    public static BigInteger generatePrimeNumber(int numBits) {
        BigInteger newRndNumber = BigInteger.TWO; // just to avoid errors - set NOT prime number
        boolean isPrime = false;

        while (isPrime == false) {
            newRndNumber = generateRandomNumber(numBits);
            isPrime = testPrimeNumber(newRndNumber);
        }
        return newRndNumber;
```

```java
    }

    ////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////
  // RSA functions
    ////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////

  public static ArrayList<BigInteger> WinnerGenerateKeyPair(BigInteger p, BigInteger q) {
    BigInteger n = p.multiply(q);
    BigInteger funOylera = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
    BigInteger e = null, d = null;



    while (true) {
     int numbits = (int) (Math.random()*(n.bitLength()/4 - 2) + 2);

     d = generatePrimeNumber(numbits);

if (d.gcd(funOylera).equals(BigInteger.ONE) == true && (BigInteger.valueOf(36).multiply(d.pow(4)).compareTo(n)) == -1) {
      // nothing here
     } else {
      // need to re-generate d
      continue;
     }
     /*
     System.out.println("d don't fit condition d < (1/sqrt(6))*n^(1/4)");
     System.out.println("d             = " + d);
     System.out.println("temp2 = " + numbits);
     */
     e = d.modInverse(funOylera);
     //check gcd (e, funOylera)
     if (e.gcd(funOylera).equals(BigInteger.ONE) == false) {
      continue;
     }
     break;
    }
    ArrayList<BigInteger> keys = new ArrayList<BigInteger>();
    keys.add(n); // index 0 - public Key
    keys.add(e); // index 1 - public Key
    keys.add(d); // index 2 - private Key
    return keys;

   }
```

```
public static BigInteger Encrypt(BigInteger M, BigInteger pubKeyE, BigInteger n) {
    return M.modPow(pubKeyE, n);
}

public static BigInteger Decrypt(BigInteger C, BigInteger privKeyD, BigInteger n) {
    return C.modPow(privKeyD, n);
}

public static void wienner_attack(BigInteger e, BigInteger n) {
        BigInteger frac = BigInteger.ONE;

            BigInteger r1 = e;
            BigInteger r2 = n;

        ArrayList<BigInteger> array = new ArrayList<BigInteger>();

        for ( ; !frac.equals(BigInteger.ZERO); ) {

                BigInteger cf = r1.divide(r2);
            frac = r1.mod(r2);
            array.add(cf);
            r1 = r2;
            r2 = frac;
           }



        // step 2
        ArrayList<BigInteger> P = new ArrayList<BigInteger>();
        ArrayList<BigInteger> Q = new ArrayList<BigInteger>();
        // add p0, q0
        P.add(array.get(0));
        Q.add(BigInteger.ONE);
        // add p1, q1
        P.add(array.get(1).multiply(P.get(0)).add(BigInteger.ONE));
        Q.add(array.get(1).multiply(Q.get(0)).add(BigInteger.ZERO));
        // add p2..pn, q2..qn
        for (int i = 2 ; i < array.size(); i++) {
                P.add(array.get(i).multiply(P.get(i - 1)).add(P.get(i - 2)));
            Q.add(array.get(i).multiply(Q.get(i - 1)).add(Q.get(i - 2)));
        }
        //step 3
        boolean found = false;
        for ( int i = 1; i < P.size(); i++) {
                // 3.1
                BigInteger k = P.get(i);
```

```
            BigInteger d = Q.get(i);

            // 3.2
            if (d.mod(BigInteger.TWO).equals(BigInteger.ZERO)) {
                continue;
            }
            // 3.3
            if (!k.equals(BigInteger.ONE) && !e.multiply(d).mod(k).equals(BigInteger.ONE)) {
                continue;
            }
            // 3.4
            BigInteger phi = e.multiply(d).subtract(BigInteger.ONE).divide(k);
            // x^2 - (n - phi + 1)x + n == x^2 + bx + c
            // b = (phi - n - 1)
            // c = n
            BigInteger b = phi.subtract(n).subtract(BigInteger.ONE);
            BigInteger c = n;
            BigInteger D = b.multiply(b).subtract(BigInteger.valueOf(4).multiply(c));
            if (D.compareTo(BigInteger.ZERO) == -1) {
                continue;
            }
            BigInteger D_sqrt = D.sqrt();
            if (!D_sqrt.multiply(D_sqrt).equals(D)) {
                continue;
            }
            BigInteger x1 = BigInteger.ZERO.subtract(b).add(D.sqrt());
            if (x1.mod(BigInteger.TWO).equals(BigInteger.ONE)) {
                continue;
            }
            BigInteger x2 = BigInteger.ZERO.subtract(b).subtract(D.sqrt());
            if (x2.mod(BigInteger.TWO).equals(BigInteger.ONE)) {
                continue;
            }
            x1 = x1.divide(BigInteger.TWO);
            x2 = x2.divide(BigInteger.TWO);
            found = true;
            System.out.println("found x1: " + x1);
            System.out.println("found x2: " + x2);
            break;
        }

        if (!found) {
            System.out.println("Given RSA parameters are not vulnerable to Wiener's attack");
        }
    }
```

```
    public static void main(String[] args) {

        int lenght;
        for (lenght = 2048; lenght < 3073; lenght = lenght + 1024) {

            double count2 = 0;
            for (int j = 0; j < 500; j++ ) {
                // Generate p,q
                BigInteger p, q;
                while (true) {
                    p = generatePrimeNumber(lenght/2);
                    q = generatePrimeNumber(lenght/2);
                    int temp = q.compareTo(p);
                    int temp2 = p.compareTo(BigInteger.TWO.multiply(q));
                    if (temp == -1 && temp2 == -1) {
                        break;
                    }
                }
                ArrayList<BigInteger> keys = WinnerGenerateKeyPair(p, q);
                BigInteger pubKey_n = keys.get(0);
                BigInteger pubKey_e = keys.get(1);
                BigInteger privKey_d = keys.get(2);

                double count = 0;
                for (int i = 0; i < 100; i++ ) {
                    double startTime = System.nanoTime();
                    wienner_attack(pubKey_e, pubKey_n);
                    double elapsedNanos = System.nanoTime() - startTime;
                    System.out.println("Time of one work in nanoseconds: " + elapsedNanos);
                    count = count + elapsedNanos;
                }
                double average_one_lenght = count/100;
        System.out.println("Average specific lenght time in nanoseconds of attack: " + average_one_lenght);
                count2 = count2 + average_one_lenght;
                //wienner_attack(new BigInteger("17993", 10), new BigInteger("90581", 10));
                //wienner_attack(new BigInteger("1073780833", 10), new BigInteger("1220275921", 10));
                //wienner_attack(new BigInteger("1779399043", 10), new BigInteger("2796304957", 10));
            }
            double average = count2/500;
    System.out.println("For lenght " + lenght + " Average time in nanoseconds of attack: " + average);
        }
  System.out.println("Average time in nanoseconds of attack: " + average);
    }
}
```

A.3 Ймовірність успіху атаки зустріч посередині

```
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;

public class Meet_in_middle_probability {

        ////////////////////////////////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////////////////////////////////
    // Prime random numbers generation
        ////////////////////////////////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////////////////////////////////

        static Random gRandomGen = new Random();

        public static BigInteger generateRandomNumber(int numBits) {

                return new BigInteger(numBits, 10, gRandomGen);
        }

        public static BigInteger generateRandomNumberNotPrime(int numBits) {

                return new BigInteger(numBits, gRandomGen);
        }

    public static int generateRandomInt(int min, int max) {
        return ThreadLocalRandom.current().nextInt(min, max);
    }

        // Test Miller-Rabin
    public static boolean testPrimeNumber(BigInteger p) {
        // step 0
        BigInteger pMinus1 = p.subtract(BigInteger.ONE);
        // find s
        int s = 0;
        // divide by 2
        while (pMinus1.mod(BigInteger.TWO).equals(BigInteger.ZERO)) {
                s++;
                pMinus1 = pMinus1.divide(BigInteger.TWO);
        }
```

```
BigInteger d = pMinus1;

pMinus1 = p.subtract(BigInteger.ONE); // refresh value

for (int k = 1; k < 10; k++) {
        // step 1
        BigInteger x = generateRandomNumber(p.bitLength());

        if (x.equals(BigInteger.ZERO) || x.equals(BigInteger.ONE) || x.equals(pMinus1)) {

                continue;
        }

        BigInteger resGcd = x.gcd(p);
        if (!resGcd.equals(BigInteger.ONE)) {

                return false;
        }

        // step 2

        BigInteger x_r = x.modPow(d, p);

        // step 2.1
        if (x_r.equals(BigInteger.ONE) || x_r.equals(pMinus1)) {

        } else {
                // step 2.2
                for (int r = 1; r < s; r++) {
                        x_r = x_r.modPow(BigInteger.TWO, p);

                        if (x_r.equals(pMinus1)) {

                                continue;
                        }

                        if (x_r.equals(BigInteger.ONE)) {

                                return false;
                        }
                }

                return false;
        }
```

```
    }

    return true;
}

    public static BigInteger generatePrimeNumber(int numBits) {
            BigInteger newRndNumber = BigInteger.TWO; // just to avoid errors - set NOT prime number
            boolean isPrime = false;

            while (isPrime == false) {
                    newRndNumber = generateRandomNumber(numBits);

                    isPrime = testPrimeNumber(newRndNumber);
            }

            return newRndNumber;
    }

    ////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////
// RSA functions
    ////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////

public static ArrayList<BigInteger> GenerateKeyPair(BigInteger p, BigInteger q) {

        BigInteger n = p.multiply(q);
        BigInteger funOylera = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
        BigInteger e = BigInteger.TWO.pow(16).add(BigInteger.ONE);

        BigInteger d = e.modInverse(funOylera);

        ArrayList<BigInteger> keys = new ArrayList<BigInteger>();
        keys.add(n); // index 0 - public Key
        keys.add(e); // index 1 - public Key
        keys.add(d); // index 2 - private Key
        return keys;
}

public static BigInteger Encrypt(BigInteger M, BigInteger pubKeyE, BigInteger n) {
    return M.modPow(pubKeyE, n);
}

public static BigInteger Decrypt(BigInteger C, BigInteger privKeyD, BigInteger n) {
    return C.modPow(privKeyD, n);
```

```
}

public static ArrayList<HashMap<BigInteger,BigInteger>> get_X_st_X_2st (BigInteger e, BigInteger n, int l) {

        HashMap<BigInteger,BigInteger> X_st = new HashMap<BigInteger,BigInteger>();
        HashMap<BigInteger,BigInteger> X_st2 = new HashMap<BigInteger,BigInteger>();
        HashMap<BigInteger,BigInteger> S_e_modInverse = new HashMap<BigInteger,BigInteger>();
        BigInteger end = BigInteger.TWO.pow(l/2).add(BigInteger.ONE);
        BigInteger start = BigInteger.ONE;

        while (!start.equals(end)) {
                BigInteger t = start;
                BigInteger t_e = t.modPow(e, n);
                //X_st.put(t, t_e);
                X_st2.put(t_e, t);

                S_e_modInverse.put(t, t_e.modInverse(n));

                start = start.add(BigInteger.ONE);
        }
        ArrayList<HashMap<BigInteger,BigInteger>> X_s = new ArrayList<HashMap<BigInteger,BigInteger>>();
        X_s.add(X_st);
        X_s.add(X_st2);
        X_s.add(S_e_modInverse);
        return X_s;
}

public static boolean meet_in_middle(BigInteger e, BigInteger n,
BigInteger C, int l, HashMap<BigInteger,BigInteger> X_st,
HashMap<BigInteger,BigInteger> X_st2, HashMap<BigInteger,BigInteger> S_e_modInverse) {

    BigInteger end = BigInteger.TWO.pow(l/2).add(BigInteger.ONE);
        BigInteger start = BigInteger.ONE;
        boolean found = false;


        BigInteger y_s = null;
        start = BigInteger.ONE;
        while (!start.equals(end)) {
                        y_s = S_e_modInverse.get(start);
                y_s = C.multiply(y_s).mod(n);

                if (X_st2.containsKey(y_s)) {
                        BigInteger C_x_1 = start;
                        BigInteger C_x_2 = X_st2.get(y_s);
```

```
                    found = true;
                    break;
                }
                start = start.add(BigInteger.ONE);
            }
            if (found == false) {
                System.out.println("Attack: s, t not found");
            }

            System.out.println("Attack: done");
            return found;
    }

    public static void main(String[] args) {

            // Generate p,q
            BigInteger p, q;
            while (true) {
                p = generatePrimeNumber(256);
                q = generatePrimeNumber(256);
                int temp = q.compareTo(p);
                int temp2 = p.compareTo(BigInteger.TWO.multiply(q));
                if (temp == -1 | temp2 == -1) {
                    break;
                }
            }



            System.out.println("");
            System.out.println("");
            ArrayList<BigInteger> keys = GenerateKeyPair(p, q);
            BigInteger pubKey_n = keys.get(0);
            BigInteger pubKey_e = keys.get(1);
            BigInteger privKey_d = keys.get(2);
            System.out.println("RSA: private key part: p  = " + p.toString(16));
            System.out.println("RSA: private key part: q  = " + q.toString(16));
            System.out.println("RSA: private key: d  = " + privKey_d.toString(16));
            System.out.println("RSA: public key: e  = " + pubKey_e.toString(16));
            System.out.println("RSA: public key: n  = " + pubKey_n.toString(16));



            System.out.println("");
            int l = 256 - 196;
            ArrayList<HashMap<BigInteger,BigInteger>> X_s = get_X_st_X_2st(pubKey_e, pubKey_n, l);
            HashMap<BigInteger,BigInteger> X_st = X_s.get(0);
```

```
        HashMap<BigInteger,BigInteger> X_2st = X_s.get(1);
        HashMap<BigInteger,BigInteger> S_e_modInverse = X_s.get(2);
        System.out.println("X' and X'' generated ");
        System.out.println("l: " + l);
        BigInteger M = null;
        int total_exp = 100;
        for (int i = l/2 + 1; i < l + 1; i = i + 2) {
            int count = 0;
            for (int j = 0; j < total_exp; j++ ) {

                M = generateRandomNumberNotPrime(i);



                BigInteger C = Encrypt(M, pubKey_e, pubKey_n);
                boolean found = meet_in_middle(pubKey_e, pubKey_n, C, l, X_st, X_2st, S_e_modInverse);

                if (found) { count++; }

            }
            double probability = (double) count/ (double) total_exp;
            System.out.println("len(M) = " + i);
            System.out.println("Probability working attack is: " + probability);
        }

    }

}
```