

Додаток А

Слайди презентації

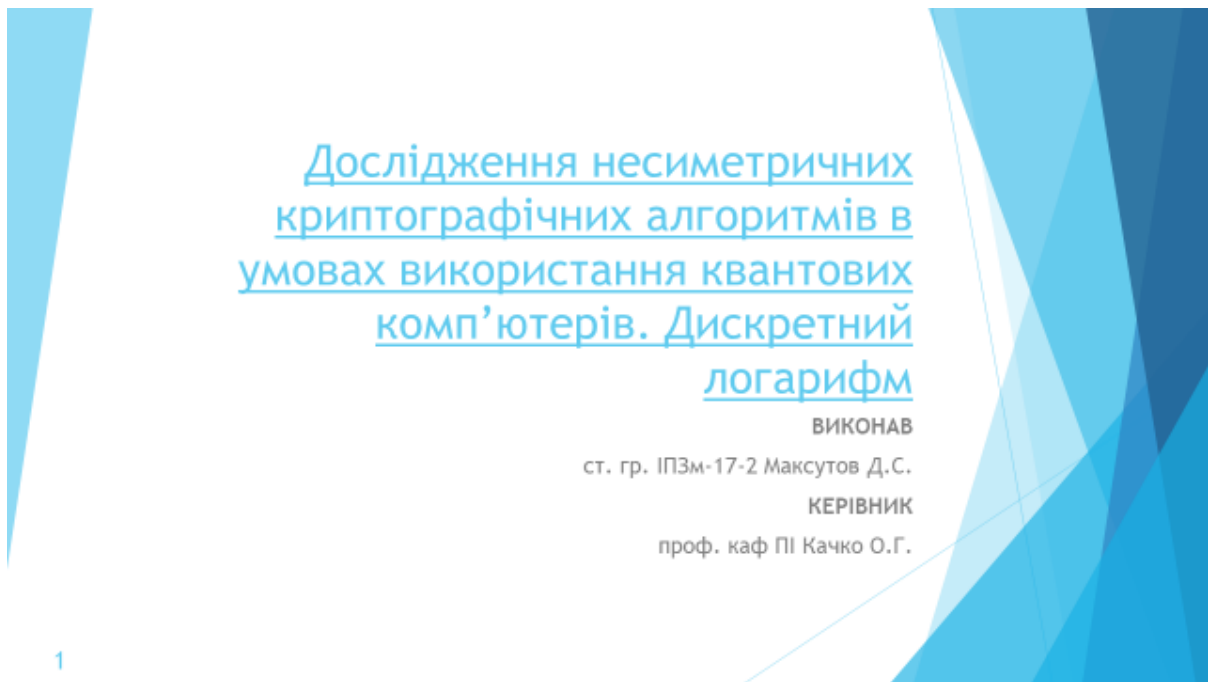


Рисунок А.1 – Слайд 1

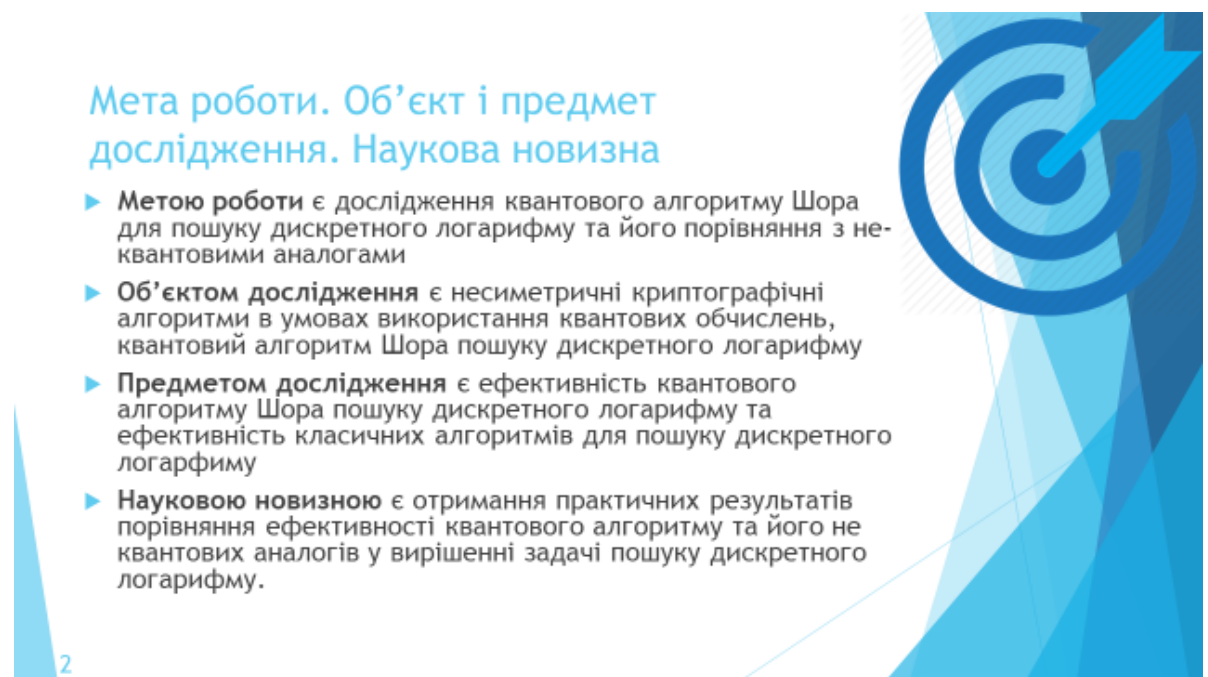


Рисунок А.2 – Слайд 2

Актуальність теми

UCLA Quantum Computing

Quantum computers can solve important problems that cannot be solved on today's computers and allow more secure communication.

Quantum Breakthrough

Classic Computer	Quantum Computer
<ul style="list-style-type: none"> carries data in bits, which are sequences of 0s and 1s. Important problems in drug and material design would take millions of years. Reaching the end of Moore's Law. Present increases in speed are slowing. 	<ul style="list-style-type: none"> carries data in qubits, which are sequences of 0s, 1s, and combinations of 0s and 1s. Could solve important drug and material design problems in seconds. Reaching-shifting increases in computational speed are forecast.

3

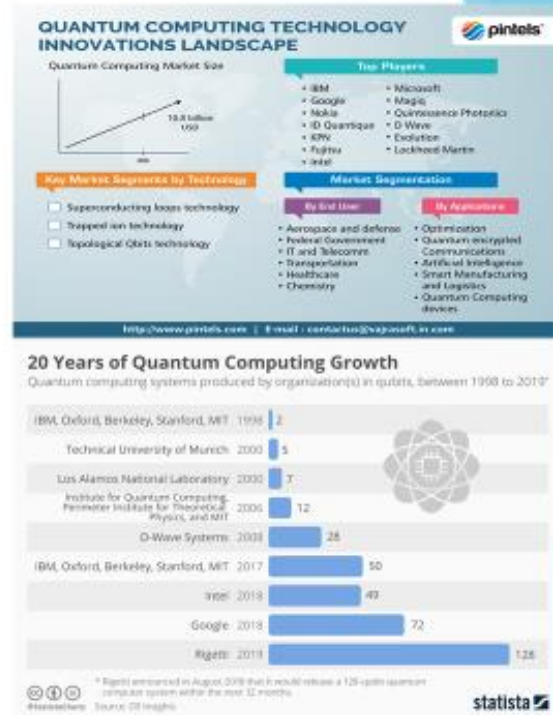


Рисунок А.3 – Слайд 3

Постановка задачі

- ▶ провести аналіз проблеми пошуку дискретного логарифму та її застосування в асиметричній криптографії;
 - ▶ дослідити існуючі класичні алгоритми для пошуку дискретного логарифму;
 - ▶ дослідити сьогоденний стан квантових обчислень та їх застосовність;
 - ▶ проаналізувати квантовий алгоритм Шора пошуку дискретного логарифму і його варіації;
 - ▶ Реалізувати квантовий алгоритм Шора та його класичні аналоги. З'ясувати можливість практичного використання квантового алгоритму Шора на сьогоднішній день. Порівняти його продуктивність з класичними аналогами. Дослідити основні перешкоди в практичному застосуванні квантового алгоритму Шора.
- 4

Рисунок А.4 – Слайд 4

Проблеми класу NP

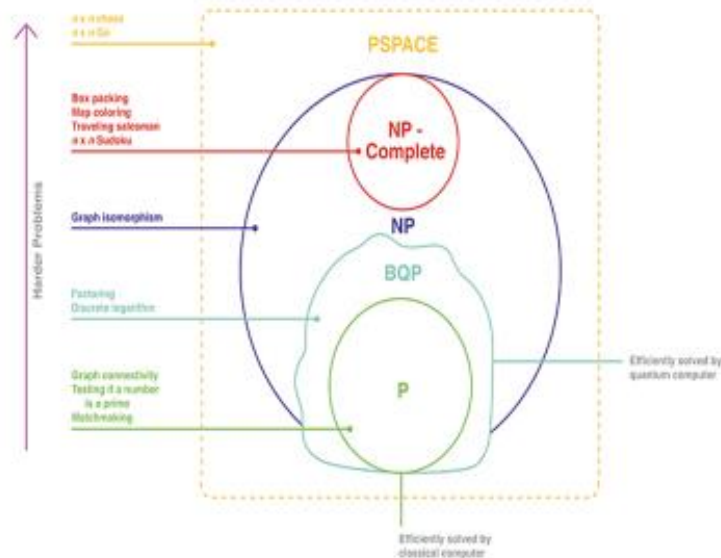


Рисунок А.5 – Слайд 5

Проблема пошуку дискретного логарифма

Discrete Logarithm

- Discrete log problem:
- Given p , g and $g^a \pmod{p}$, determine a
 - This would break Diffie-Hellman and ElGamal
- Discrete log algorithms analogous to factoring, except no sieving
 - This makes discrete log harder to solve
 - Implies smaller numbers can be used for equivalent security, compared to factoring

Рисунок А.6 – Слайд 6

Криптографічні протоколи побудовані на основі проблеми пошуку дискретного логарифму

Diffie-Hellman key exchange protocol

DSA - Digital Signature Algorithm

ElGamal Encryption

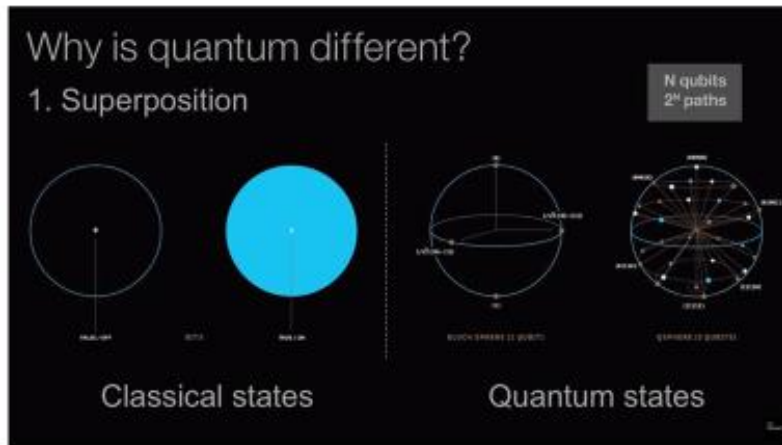
ECDH - Diffie-Hellman key exchange protocol on Elliptic curve

ECDSA - Elliptic Curve Digital Signature Algorithm

7

Рисунок А.7 – Слайд 7

Квантові обчислення



8

Рисунок А.8 – Слайд 8

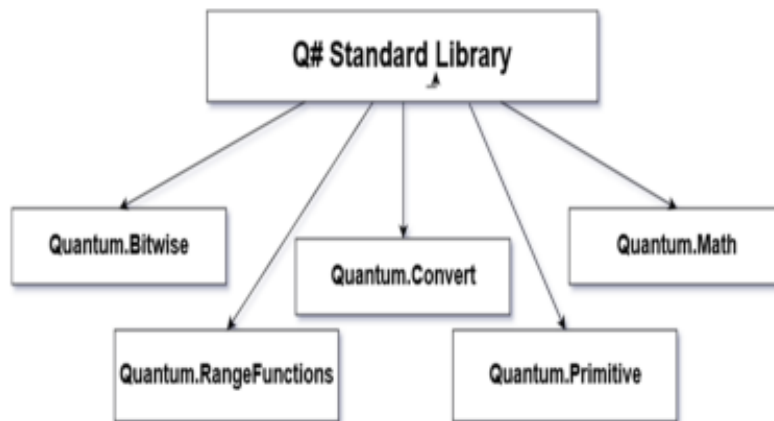
Класичні алгоритми пошуку дискретного логарифму

- ▶ Brute Force
- ▶ Baby Step Giant Step
- ▶ Pollard-Rho algorithm
- ▶ Pohling - Hellman's algorithm

9

Рисунок А.9 – Слайд 9

Q# та QDK



10

Рисунок А.10 – Слайд 10

Алгоритм Шора пошуку дискретного логарифму

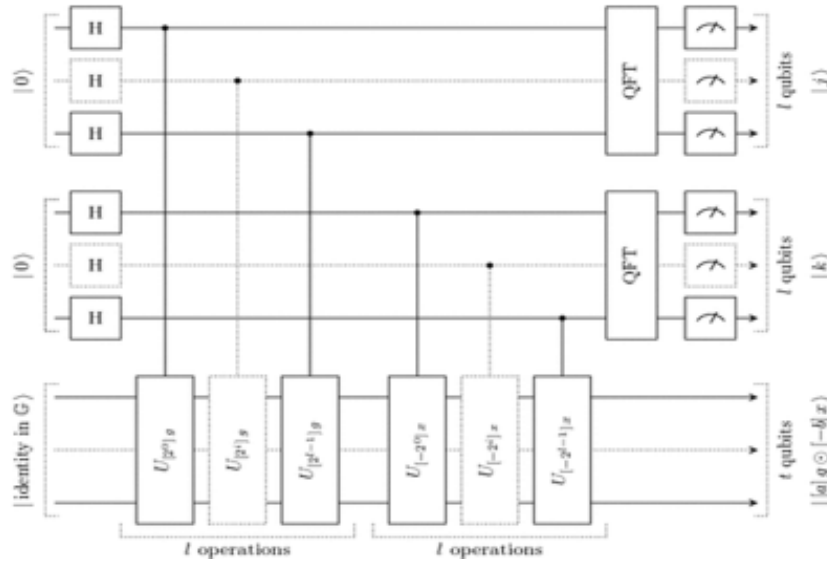


Рисунок А.11 – Слайд 11

Продуктивність класичних алгоритмів

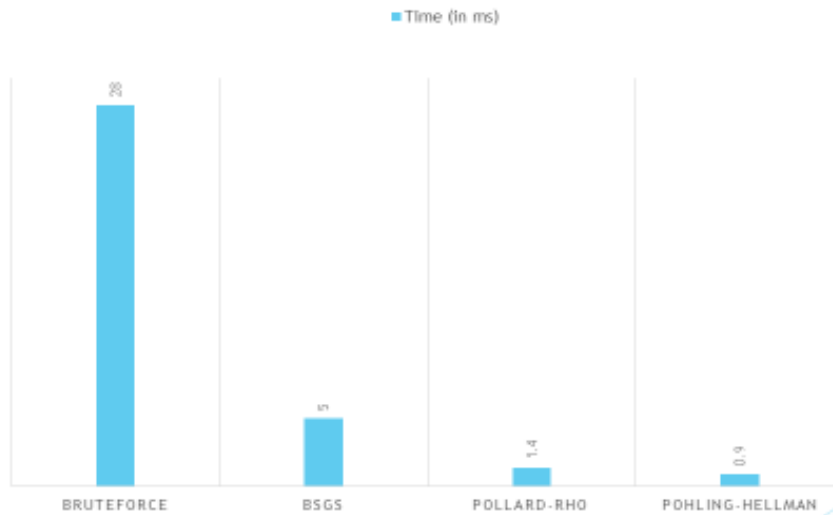



Рисунок А.12 – Слайд 12

Квантовий алгоритм Шора. Результати та перешкоди

	<p>Реалізований алгоритм завершив роботу тільки для найпростішої циклічної групи за модулем 3. При цьому час його виконання наблизився дорівнював 2 годинам 57 хвилинам.</p>
	<p>Це, в більшій мірі, було зумовлено використання емулятора замість реального квантового комп'ютера.</p>
	<p>Навіть у теорії алгоритм Шора пошуку дискретного логарифму потребує $6n$ кубітів. Де n - число байт в модулі циклічної групи.</p>
	<p>Враховуючи що сучасний рекомендований розмір модуля для протоколу Діффі-Хеллмана дорівнює 2048 бітам, для його злому буде потрібен 12288-кубітний квантовий комп'ютер.</p>

13

Рисунок А.13 – Слайд 13

Висновки

- ▶ Було досліджено проблему вирішення NP-задач, а саме проблему пошуку дискретного логарифму за допомогою парадигми квантових обчислень.
- ▶ Було виконано порівняння існуючих, не квантових, методів вирішення проблеми пошуку дискретного логарифму.
- ▶ Був проведений детальний аналіз декількох варіантів квантового алгоритму Шора для пошуку дискретного логарифму, а також досліджено не квантові алгоритми для пошуку дискретного логарифму. В ході дослідження було виявлено, що алгоритм Шора пошуку дискретного логарифму все ще не може конкурувати навіть з найпростішими класичними алгоритмами для пошуку дискретного логарифму.

14

Рисунок А.14 – Слайд 14

Додаток Б

Лістинг коду програми

```

package ua.nure.maksutov.algo.impl.pollardrho;

import com.google.common.hash.HashFunction;
import com.google.common.hash.Hashing;
import ua.nure.maksutov.algo.impl.DiscreteLogProblemSolver;

import java.math.BigInteger;
import java.nio.charset.Charset;
import java.security.InvalidParameterException;
import java.util.List;
import java.util.logging.Logger;

@SuppressWarnings("UnstableApiUsage")
public class PollardRhoSolver implements DiscreteLogProblemSolver {

    private static final Logger LOGGER =
    Logger.getLogger(PollardRhoSolver.class.getName());
    private static final LinearCongruenceSolver LINEAR_SOLVER = new
    LinearCongruenceSolver();
    private static final HashFunction[] HASH_FUNCTIONS = {
    Hashing.murmur3_32(), Hashing.murmur3_128(), Hashing.sha256(),
    Hashing.sipHash24()};
    private static final int FIRST_BUCKET_UPPER_LIMIT = Integer.MAX_VALUE /
    3;
    private static final int SECOND_BUCKET_UPPER_LIMIT = 2 *
    FIRST_BUCKET_UPPER_LIMIT;
    private static final BigInteger TWO = BigInteger.valueOf(2);
    private static final String INVALID_BUCKET_INDEX = "Invalid bucket index:
    ";

    @Override
    public BigInteger logarithm(BigInteger a, BigInteger b, BigInteger p) {
        for (HashFunction hashFunction : HASH_FUNCTIONS) {

            InfoPair pairIndexFirstI = new InfoPair(BigInteger.ONE,
            BigInteger.ZERO, BigInteger.ZERO);

            pairIndexFirstI = computeNextPair(pairIndexFirstI, a, b, p,
            hashFunction);

```

```

        InfoPair pairIndexSecondI = computeNextPair(pairIndexFirstI, a,
b, p, hashFunction);

        while (pairIndexFirstI.getX().compareTo(pairIndexSecondI.getX())
!= 0) {
            pairIndexFirstI = computeNextPair(pairIndexFirstI, a, b, p,
hashFunction);

            pairIndexSecondI = computeNextPair(pairIndexSecondI, a, b, p,
hashFunction);
            pairIndexSecondI = computeNextPair(pairIndexSecondI, a, b, p,
hashFunction);
        }

        // Now the collision is found out, now only have to solve the
congruency linear equation:
        //  $x(\delta_1 - \delta_2) = (\epsilon_2 - \epsilon_1) \pmod{p-1}$ .

        // We denote  $\alpha = \delta_1 - \delta_2$ ,  $\beta = \epsilon_2 - \epsilon_1$  and  $\phi = p - 1$ .
        // Therefore, the congruence to solve is:  $\alpha x = \beta \pmod{\phi}$ .
        // So, we must find  $\gcd(\alpha, \phi)$  and verify if

        BigInteger phi = p.subtract(BigInteger.ONE);
        BigInteger alpha =
pairIndexFirstI.getDelta().subtract(pairIndexSecondI.getDelta()).mod(phi);
        BigInteger beta =
pairIndexSecondI.getEpsilon().subtract(pairIndexFirstI.getEpsilon()).mod(phi)
;

        try {
            List<BigInteger> solutions = LINEAR_SOLVER.solve(alpha, beta,
phi);
            for (BigInteger solution : solutions) {
                if (isCorrectLog(a, b, solution, p)) {
                    return solution.mod(phi);
                }
            }
            throw new InvalidParameterException("Could not find Discrete
Logarithm!");
        } catch (InvalidParameterException e) {
            LOGGER.warning(e.getMessage());
        }
    }
    throw new InvalidParameterException("Could not find Discrete
Logarithm, did not find any hash function to result a solvable linear
congruence!");
}

```

```

private InfoPair computeNextPair(InfoPair current, BigInteger a,
    BigInteger b, BigInteger p, HashFunction hashFunction) {
    int bucket = computeBucketIndex(current.getX(), hashFunction);

    return new InfoPair(
        computeNextX(current.getX(), a, b, p, bucket),
        computeNextEpsilon(current.getEpsilon(), p, bucket),
        computeNextDelta(current.getDelta(), p, bucket));
}

private BigInteger computeNextX(BigInteger currentX, BigInteger a,
    BigInteger b, BigInteger p, int bucket) {
    switch (bucket) {
    case 1:
        return currentX.multiply(b).mod(p);
    case 2:
        return currentX.multiply(currentX).mod(p);
    case 3:
        return currentX.multiply(a).mod(p);
    default:
        throw new InvalidParameterException(INVALID_BUCKET_INDEX +
bucket);
    }
}

private BigInteger computeNextEpsilon(BigInteger currentEpsilon,
    BigInteger p, int bucket) {
    BigInteger phi = p.subtract(BigInteger.ONE);
    switch (bucket) {
    case 1:
        return currentEpsilon;
    case 2:
        return currentEpsilon.multiply(TWO).mod(phi);
    case 3:
        return currentEpsilon.add(BigInteger.ONE).mod(phi);
    default:
        throw new InvalidParameterException(INVALID_BUCKET_INDEX +
bucket);
    }
}

private BigInteger computeNextDelta(BigInteger currentDelta, BigInteger
    p, int bucket) {
    BigInteger phi = p.subtract(BigInteger.ONE);
    switch (bucket) {
    case 1:
        return currentDelta.add(BigInteger.ONE).mod(phi);

```

```

        case 2:
            return currentDelta.multiply(TWO).mod(phi);
        case 3:
            return currentDelta;
        default:
            throw new InvalidParameterException(INVALID_BUCKET_INDEX +
bucket);
    }
}

private boolean isCorrectLog(BigInteger a, BigInteger b, BigInteger x,
BigInteger p) {
    return a.modPow(x, p).compareTo(b) == 0;
}

/**
 * Method that uniformly distributes a number to a bucket, using a hash
function. The possible number of
 * buckets is 3. The same number will be every time assigned to the same
bucket.
 *
 * @param number the number to be randomly assigned to a bucket.
 * @return the corresponding bucket, a value of 1, 2 or 3.
 */
private int computeBucketIndex(BigInteger number, HashFunction
hashFunction) {
    int hash = hashFunction.hashString(number.toString(),
Charset.defaultCharset()).asInt();
    int bucket = 3;
    if (hash < 0) {
        hash *= -1;
    }

    if (hash < FIRST_BUCKET_UPPER_LIMIT) {
        bucket = 1;
    } else if (hash < SECOND_BUCKET_UPPER_LIMIT) {
        bucket = 2;
    }
    return bucket;
}

private static final class InfoPair {
    private BigInteger x;
    private BigInteger epsilon;
    private BigInteger delta;

    InfoPair(BigInteger x, BigInteger epsilon, BigInteger delta) {

```

```

        this.x = x;
        this.epsilon = epsilon;
        this.delta = delta;
    }

    BigInteger getX() {
        return x;
    }
    BigInteger getEpsilon() {
        return epsilon;
    }
    BigInteger getDelta() {
        return delta;
    }
}
}

namespace UA.NURE.Maksutov.DiscreteLog.Shor
{
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Arithmetic;
    open Microsoft.Quantum.Extensions.Diagnostics;

    // Rotation Gate
    // Input: A qubit in state  $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$  and an integer  $k$ .
    // Result: Change the state of the qubit to  $\alpha |0\rangle + \beta * e^{\{2\pi i / 2^k\}}$ 
    |1>.
    operation Rotation (q : Qubit, k : Int) : Unit {
        body(...) {
            // ...
            R1Frac(2, k, q);
        }

        adjoint auto;
        controlled auto;
        adjoint controlled auto;
    }

    // QFT
    // Input: A quantum register in big endian format.
    // Result: QFT is run on the input register.
    operation QuantumFT (qs : Qubit[]) : Unit {
        body(...) {

```

```

    let n = Length(qs);
    for (i in 0 .. n - 1) {
        H(qs[i]);
        for (j in i + 1 .. n - 1) {
            Controlled Rotation([qs[j]], (qs[i], j - i + 1));
        }
    }
    SwapReverseRegister(qs);
}

adjoint auto;
controlled auto;
adjoint controlled auto;
}

// Inverse QFT
// Input: A quantum register in big endian format.
operation InverseQFT (qs : Qubit[]) : Unit {
    body(...) {
        Adjoint QuantumFT(qs);
    }

    adjoint auto;
    controlled auto;
    adjoint controlled auto;
}

// Power  $|x\rangle|y\rangle \rightarrow |x\rangle|a^x \cdot y \bmod N\rangle$ 
// Input: integers a and N, registers x and y.
operation PowerOfa (a : Int, N : Int, x : Qubit[], y : Qubit[]) : Unit {

    body(...) {
        let y_LE = BigEndianAsLittleEndian(BigEndian(y));
        let oracle = MultiplyByModularInteger(a, N, _);
        for (p in 0 .. Length(x) - 1) {
            Controlled (OperationPowCA(oracle, 1 <<< p))([x[Length(x) - 1
- p]], y_LE);
            Message($"p={p} completed");
        }
    }
    adjoint auto;
    controlled auto;
    adjoint controlled auto;
}

// Oracle  $U|x_1\rangle|x_2\rangle|y\rangle \rightarrow |x_1\rangle|x_2\rangle|y \oplus f(x_1,x_2)\rangle$  where  $f(x_1,x_2)=b^{x_1} \cdot a^{x_2} \bmod N$ 

```

```

// Input: integers a, b, and N, registers x1, x2, and qs.
operation OracleFunc (a : Int, b : Int, N : Int, x1 : Qubit[], x2 :
Qubit[], qs : Qubit[]) : Unit {
  body(...) {
    X(qs[Length(qs) - 1]);
    PowerOfa(b, N, x1, qs);
    PowerOfa(a, N, x2, qs);
  }

  adjoint auto;
  controlled auto;
  adjoint controlled auto;
}

operation DLOracle (a : Int, b : Int, N : Int, x1 : Qubit[], x2 :
Qubit[], y : Qubit[]) : Unit {
  body(...) {
    using (qs = Qubit[Length(y)]) {
      OracleFunc(a, b, N, x1, x2, qs);
      for (i in 0 .. Length(qs) - 1) {
        CNOT(qs[i], y[i]);
      }
      Adjoint OracleFunc(a, b, N, x1, x2, qs);
    }
  }

  adjoint auto;
  controlled auto;
  adjoint controlled auto;
}

// Discrete logarithm Algorithm
// Input: integers a, b, N
// Goal: compute the discrete logarithm log_a(b), returns -1 on failure
cases
operation DiscreteLogShor (a: Int, b: Int, N: Int) : Int {

  mutable appr_slmodr = 0;
  mutable appr_l = 0;
  let order = N - 1; // Euler function
  Message($"Order is ={order}");
  let t = BitSizeI(N) * 2 + 1;
  Message($"Register size is ={t}");
  using ((x1, x2, y) = (Qubit[t], Qubit[t], Qubit[t])) {
    ApplyToEach(H, x1);
    ApplyToEach(H, x2);
    DLOracle(a, b, N, x1, x2, y);
    InverseQFT(x1);
  }
}

```

```

        InverseQFT(x2);
        set appr_slmodr =
MeasureInteger(BigEndianAsLittleEndian(BigEndian(x1)));
        set appr_1 =
MeasureInteger(BigEndianAsLittleEndian(BigEndian(x2)));
        ResetAll(x1);
        ResetAll(x2);
        ResetAll(y);
    }
    let (slmodr, _) = (ContinuedFractionConvergentI(Fraction(appr_slmodr,
t), N))!;
    let (l, _) = (ContinuedFractionConvergentI(Fraction(appr_1, t), N))!;
    if (GreatestCommonDivisorI(l, order) != 1) {
        Message($"Algorithm failed due to measure l={l}, no coprime to
r={order}");
        return -1;
    }
    let l_inv = InverseModI(l, order);
    let s = slmodr * l_inv % order;
    return s;
}
}

```

Додаток В

Наукові публікації

В.1 Тези до XXIII Міжнародного молодіжного форуму



Рисунок В.1 – Обкладинка збірника матеріалів XXIII Міжнародного молодіжного форуму

МОДИФІКОВАНИЙ КВАНТОВИЙ АЛГОРИТМ ШОРА ДЛЯ ПОШУКУ ДИСКРЕТНОГО ЛОГАРИФМУ

Максутов Д.С

Науковий керівник – к.т.н., проф. Качко О.Г.

Харківський національний університет радіоелектроніки

(61166, Харків, просп. Науки, 14, каф. Системотехніки,

тел. (057) 702-13-06)

e-mail: dmytro.maksutov@nure.ua, факс (057) 702-11-13

The given work is devoted to the modern developments in the field of quantum computer algorithms related to cryptanalysis. The major topic of the work is a modification of the Shor quantum algorithm that is designed to solve a discrete logarithm problem. The work also contains information about the current state of quantum computing technologies and known non-quantum solutions to the discrete logarithm problem. The aim of the work is to demonstrate that nowadays-cryptographic systems based on Elliptic-curve cryptography (ECC) are vulnerable to quantum algorithms or will be vulnerable in the near future.

З розвитком технології виробництва та емуляції квантових комп'ютерів, а також стрімкого розвитку технологій що покладаються на засоби асиметричної криптографії з використанням еліптичних кривих, таких як шифрування даних у пристроях компанії Blackberry та деякі імплементації Blockchain, нагальним стає питання крипто аналізу криптографії на еліптичних кривих та пошук і вдосконалення алгоритмів пошуку дискретного логарифму, який лежить в її основі.

На сьогодні існує декілька не квантових алгоритмів для пошуку дискретного логарифму, які досягають теоретичного максимуму швидкодії можливого на звичайних ЕВМ. Це такі алгоритми, як модифікація загального методу решета числового поля (GNFS) [1] з асимптотичною оцінкою $O(e^{(\frac{64}{9})^{\frac{1}{3}} (\ln p)^{\frac{1}{3}} (\ln \ln p)^{\frac{2}{3}}})$ та р-алгоритм Поларда [2] з асимптотичною оцінкою $O(\sqrt{q})$, де q – це порядок підгрупи F_p і p – порядок поля Галуа F_p .

З появою теорії квантових комп'ютерів почалася активна розробка квантових алгоритмів для ефективного вирішення проблем факторизації та пошуку дискретного логарифму. В 1994 році Пітер Шор опублікував перший варіант квантового алгоритму для вирішення вищезначених проблем. Через три роки був опублікований докладний опис алгоритму [3]. Шор описав алгоритм поліноміальної складності для пошуку дискретного логарифму лише для певного випадку, а саме, пошук дискретного логарифму у мультиплікативній групі F_p^* поля F_p .

Це залишило простір для вдосконалення алгоритму з метою покращення асимптотичної оцінки складності, кількості задіяних кубітів і

застосування його в інших групах поля F_p . Одним з таких вдосконалень є адаптація алгоритму Шора для пошуку дискретного логарифму в групі G з відомим простим порядком q і груповою операцією \odot [4]. Вдосконалений алгоритм складається з двох кроків:

1. Власне квантовий алгоритм, який приймає на вхід генератор групи g і елемент $x = [d]g$ і повертає, як результат пару (k, j) і $[e]g$, що ігнорується.

2. Класичний алгоритм, що приймає на вхід пару (k, j) і повертає, як результат шуканий дискретний логарифм d , якщо пара «хороша», тобто відповідає певним вимогам.

Описаний алгоритм потребує $2[\log_2 q]$ регістрів для індексу і обчислення двох квантових перетворень Фур'є розміру $2^{[\log_2 q]}$.

Теоретична нижня границя вірогідності отримання шуканого дискретного логарифму з першого запуску дорівнює 2^{-10} . Практична вірогідність має бути вищою, але це ще необхідно перевірити.

Алгоритм теоретично обґрунтовано, тому подальше дослідження буде стосуватися вивчення поведінки алгоритму на практиці, а саме його реалізацію і запуск на емуляторі або одному з доступних квантових комп'ютерів (IBM), з подальшим відстеженням його роботи під впливом можливої декогеренції та квантового шуму.

Список джерел:

1. O. Schirokauer, "Discrete Logarithms and Local Units", in *Philosophical Transactions of the Royal Society of London*, volume A 345, 1993, pp. 409-423.
2. S. C. Pohlig, M. E. Hellman, "An Improved Algorithm for Computing Logarithms over $GF(p)$ and Its Cryptographic Significance", in *IEEE Transactions on Information Theory*, volume IT-24, no 1, 1978, pp. 106-110.
3. P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer", in *SIAM Journal of Computing* volume 26, no 5, 1997, pp. 1484-1509.
4. Martin Ekerå, "Modifying Shor's algorithm to compute short discrete logarithms", in *IACR Cryptology ePrint Archive*, 2016

Додаток Г
Електронні матеріали на CD