

Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»
(СибГУТИ)

Кафедра ПМиК

Лабораторная работа №1

«Реализация сетевого протокола выработки секретного ключа
Diffie Hellman & MQV»

Выполнил:

студент гр. МГ-211 _____ / Бурдуковский И.А./
подпись

Проверил:

Профессор
кафедры ПМиК _____ / Фионов А.Н./

Новосибирск

2023 г.

ОГЛАВЛЕНИЕ

Задание.....	3
Выполнение	4
Листинг	10

Задание

1. Самостоятельно изучить библиотеку GMP для реализации арифметики с длинными числами. Руководство и рекомендации по установке находятся в ЭИОС. По желанию студента допускается использовать другие известные ему средства реализации арифметики с длинными числами.
2. Реализовать программу генерации чисел p , q , g для операций в мультипликативной группе \mathbb{Z}_p^* и в циклической подгруппе G порядка q . Сгенерированные числа сохранить для последующего использования в файле.
3. Реализовать исходный алгоритм Диффи–Хеллмана, алгоритм Диффи–Хеллмана в подгруппе, алгоритм MQV.
4. Осуществить замеры времени (в виде числа процессорных циклов) при выполнении основных этапов во всех алгоритмах и провести их сопоставление.

Выполнение

1. Библиотека GMP была изучена, но было принято решение использовать стандартную библиотеку C# для реализации BigInteger (System.Numerics.BigInteger).
2. Реализованы генерация чисел p , q , g для операций в мультипликативной группе \mathbb{Z}_p^* :

```
public static CryptoInitializers MultiplicitySubGroupZp(Random random, int
keySize)
{
    if (keySize % 2 != 0)
        throw new ArgumentException($"{nameof(keySize)} must be even");

    var q = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[keySize];
        while (true)
        {
            random.NextBytes(buffer);
            buffer[^1] |= 0x80;

            var value = new BigInteger(buffer, true);

            if (value.IsProbablyPrime(random) && (2 * value +
1).IsProbablyPrime(random))
                return value;
        }
    })) ();
    Console.WriteLine($"Q: {q} [{q.GetBitLength()}b]");

    var p = 2 * q + 1;
    Console.WriteLine($"P: {p} [{p.GetBitLength()}b]");

    var g = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(p - 1).GetByteCount()];
        while (true)
        {
            random.NextBytes(buffer);

            var value = new BigInteger(buffer, true);

            if (value > 1 && value < p - 1 && BigInteger.ModPow(value, q,
p) != 1)
                return value;
        }
    })) ();
    Console.WriteLine($"G: {g} [{g.GetBitLength()}b]");

    return new CryptoInitializers
    {
        P = p,
        Q = q,
        G = g
    };
}
```

В циклической подгруппе G порядка q:

```
public static CryptoInitializers CyclingSubgroupPowerGp(Random random, int
keySizeQ, int keySizeP)
{
    BigInteger q;
    q = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[keySizeQ];
        while (true)
        {
            random.NextBytes(buffer);
            buffer[^1] |= 0x80;

            var value = new BigInteger(buffer, true);

            if (value.IsProbablyPrime(random))
                return value;
        }
    })) ();
    Console.WriteLine($"Q: {q}[{q.GetBitLength()}b]");

    BigInteger p;
    p = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[keySizeP - keySizeQ];
        while (true)
        {
            random.NextBytes(buffer);
            buffer[^1] |= 0x80;

            var value = (new BigInteger(buffer, true) * q + 1);

            if (value.IsProbablyPrime(random))
                return value;
        }
    })) ();
    Console.WriteLine($"P: {p}[{p.GetBitLength()}b]");

    var g = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[keySizeP - keySizeQ];
        while (true)
        {
            random.NextBytes(buffer);

            var r = new BigInteger(buffer, true);
            var g = BigInteger.ModPow(r, (p - 1) / q, p);

            if (g > 1 && BigInteger.ModPow(g, q, p) == 1)
                return g;
        }
    })) ();
    Console.WriteLine($"G: {g}[{g.GetBitLength()}b]");

    return new CryptoInitializers
    {
        P = p,
        Q = q,
        G = g
    };
}
```

Структура для хранения чисел:

```
public record CryptoInitializers
{
    public BigInteger P { get; set; }
    public BigInteger Q { get; set; }
    public BigInteger G { get; set; }
}
```

3. Реализован исходный алгоритм Диффи–Хеллмана

```
public void DiffieHellman()
{
    var rand = new Random();

    Console.WriteLine("\nDiffie Hellman\n");

    var init = MultiplicitySubGroupZp(rand, 2);

    //Person 1
    var a = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(init.P - 1).GetByteCount(true)];
        while (true)
        {
            rand.NextBytes(buffer);

            var value = new BigInteger(buffer, true);
            if (value < init.P - 1 &&
                BigInteger.GreatestCommonDivisor(value, init.P - 1) == 1)
                return value;
        }
    }));
    var A = BigInteger.ModPow(init.G, a, init.P);

    //Person 2
    var b = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(init.P - 1).GetByteCount(true)];
        while (true)
        {
            rand.NextBytes(buffer);

            var value = new BigInteger(buffer, true);
            if (value < init.P - 1 &&
                BigInteger.GreatestCommonDivisor(value, init.P - 1) == 1)
                return value;
        }
    }));
    var B = BigInteger.ModPow(init.G, b, init.P);

    var Zab = BigInteger.ModPow(B, a, init.P);
    var Zba = BigInteger.ModPow(A, b, init.P);

    Console.WriteLine($"Result Key Zab: {Zab}\n");
    Console.WriteLine($"Result Key Zba: {Zba}\n");
}
```

Алгоритм Диффи–Хеллмана в подгруппе

```
public void DiffieHellmanGroup()
{
    var rand = new Random();

    Console.WriteLine("\n Diffie Hellman New \n");
}
```

```

var init = CyclingSubgroupPowerGp(rand, 32, 128);

//Person 1
var a = ((Func<BigInteger>) (() =>
{
    var buffer = new byte[(init.P - 1).GetByteCount(true)];
    while (true)
    {
        rand.NextBytes(buffer);

        var value = new BigInteger(buffer, true);
        if (value < init.P - 1 &&
BigInteger.GreatestCommonDivisor(value, init.P - 1) == 1)
            return value;
    }
})) ();
var A = BigInteger.ModPow(init.G, a, init.P);

//Person 2
var b = ((Func<BigInteger>) (() =>
{
    var buffer = new byte[(init.P - 1).GetByteCount(true)];
    while (true)
    {
        rand.NextBytes(buffer);

        var value = new BigInteger(buffer, true);
        if (value < init.P - 1 &&
BigInteger.GreatestCommonDivisor(value, init.P - 1) == 1)
            return value;
    }
})) ();
var B = BigInteger.ModPow(init.G, b, init.P);

var Zab = BigInteger.ModPow(B, a, init.P);
var Zba = BigInteger.ModPow(A, b, init.P);

Console.WriteLine($"Result Key Zab: {Zab}\n");
Console.WriteLine($"Result Key Zba: {Zba}\n");
}

```

Алгоритм MQV

```

public void MQV()
{
    var rand = new Random();

    var keySizeP = 128;
    var keySizeQ = 32;

    var l = (keySizeQ * 8) / 2;

    Console.WriteLine("\n MQV \n");

    var init = CyclingSubgroupPowerGp(rand, keySizeQ, keySizeP);

    //Person 1
    var a = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(init.Q - 1).GetByteCount()];
        while (true)
        {
            rand.NextBytes(buffer);

            var value = new BigInteger(buffer, true);
            if (value < init.Q - 1 &&
BigInteger.GreatestCommonDivisor(value, init.Q - 1) == 1)

```

```

        return value;
    }
})))();
var A = BigInteger.ModPow(init.G, a, init.Q);

//Person 2
var b = ((Func<BigInteger>) (() =>
{
    var buffer = new byte[(init.Q - 1).GetByteCount()];
    while (true)
    {
        rand.NextBytes(buffer);

        var value = new BigInteger(buffer, true);
        if (value < init.Q - 1 &&
BigInteger.GreatestCommonDivisor(value, init.Q - 1) == 1)
            return value;
    }
})))();
var B = BigInteger.ModPow(init.G, b, init.Q);

{
    Console.WriteLine("Session");

    //Person 1
    var x = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(init.Q - 1).GetByteCount()];
        while (true)
        {
            rand.NextBytes(buffer);

            var value = new BigInteger(buffer, true);
            if (value < init.Q - 1 &&
BigInteger.GreatestCommonDivisor(value, init.Q - 1) == 1)
                return value;
        }
    })))();
    var X = BigInteger.ModPow(init.G, x, init.Q);

    //Person 2
    var y = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(init.Q - 1).GetByteCount()];
        while (true)
        {
            rand.NextBytes(buffer);

            var value = new BigInteger(buffer, true);
            if (value < init.Q - 1 &&
BigInteger.GreatestCommonDivisor(value, init.Q - 1) == 1)
                return value;
        }
    })))();
    var Y = BigInteger.ModPow(init.G, y, init.Q);

    var d = BigInteger.Pow(2, 1) + (X % BigInteger.Pow(2, 1));
    var e = BigInteger.Pow(2, 1) + (Y % BigInteger.Pow(2, 1));

    var Sa = BigInteger.ModPow(Y * BigInteger.ModPow(B, e, init.Q) %
init.Q, (x + d * a) % init.P, init.Q);
    var Sb = BigInteger.ModPow(X * BigInteger.ModPow(A, d, init.Q) %
init.Q, (y + e * b) % init.P, init.Q);

    Console.WriteLine($"Result Key Sa: {Sa}\n");
    Console.WriteLine($"Result Key Sb: {Sb}\n");
}

```



```
}
```

4. Для осуществления замеров времени (в виде числа процессорных циклов) при выполнении основных этапов во всех алгоритма был использован пакет BenchMarkDotNet. Запуск:

```
var summary = BenchmarkRunner.Run(typeof(Program).Assembly);
```

Результат работы тестов (два выполнения):

```
BenchmarkDotNet=v0.13.1, OS=Windows 10.0.19044.1706 (21H2)
11th Gen Intel Core i7-11800H 2.30GHz, 1 CPU, 16 logical and 8 physical cores
.NET SDK=6.0.300
[Host] : .NET 6.0.5 (6.0.522.21309), X64 RyuJIT
QuickJob : .NET 6.0.5 (6.0.522.21309), X64 RyuJIT
ShortRun : .NET 6.0.5 (6.0.522.21309), X64 RyuJIT

IterationCount=3 LaunchCount=1
```

Method	Job	InvocationCount	UnrollFactor	WarmupCount	Mean	Error	StdDev	Median
DiffieHellman	QuickJob	4	1	2	1.902 ms	3.061 ms	0.1678 ms	1.816 ms
DiffieHellmanGroup	QuickJob	4	1	2	3,412.543 ms	26,062.552 ms	1,428.5764 ms	3,787.701 ms
MQV	QuickJob	4	1	2	1,795.819 ms	22,955.060 ms	1,258.2443 ms	1,249.425 ms
DiffieHellman	ShortRun	1	16	3	1.887 ms	3.099 ms	0.1699 ms	1.880 ms
DiffieHellmanGroup	ShortRun	1	16	3	1,961.208 ms	28,916.536 ms	1,585.0129 ms	1,485.336 ms
MQV	ShortRun	1	16	3	2,836.394 ms	66,607.551 ms	3,650.9847 ms	831.977 ms

Листинг

```
using System.Numerics;
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

namespace Lab1;

public record CryptoInitializers
{
    public BigInteger P { get; set; }
    public BigInteger Q { get; set; }
    public BigInteger G { get; set; }
}

public static class Helpers
{
    public static bool IsProbablyPrime(this BigInteger value, Random rand, int witnesses = 10)
    {
        if (value <= 1)
            return false;

        if (witnesses <= 0)
            witnesses = 10;

        var d = value - 1;
        var s = 0;

        while (d % 2 == 0)
        {
            d /= 2;
            s += 1;
        }

        var bytes = new byte[value.ToByteArray().LongLength];

        for (var i = 0; i < witnesses; i++)
        {
            BigInteger a;
            do
            {
                rand.NextBytes(bytes);

                a = new BigInteger(bytes, true);
            } while (a < 2 || a >= value - 2);

            var x = BigInteger.ModPow(a, d, value);
            if (x == 1 || x == value - 1)
                continue;

            for (var r = 1; r < s; r++)
            {
                x = BigInteger.ModPow(x, 2, value);

                if (x == 1)
                    return false;
                if (x == value - 1)
                    break;
            }
        }
    }
}
```

```

        if (x != value - 1)
            return false;
    }

    return true;
}

}

//[SimpleJob(launchCount: 1, warmupCount: 3, targetCount: 5, invocationCount:10, id: "QuickJob")]
[SimpleJob(launchCount: 1, warmupCount: 2, targetCount: 3, invocationCount:4, id: "QuickJob")]
[ShortRunJob]
public class Program
{
    #region Helpers

    public static BigInteger Euclid(BigInteger value1, BigInteger value2)
    {
        while (value2 != 0)
        {
            var r = value1 % value2;
            value1 = value2;
            value2 = r;
        }

        return value1;
    }

    public static void ExtendedEuclid(
        BigInteger a,
        BigInteger b,
        out BigInteger x,
        out BigInteger y,
        out BigInteger z
    )
    {
        if (a < b)
        {
            x = 1;
            y = 0;
            z = a;
        }
        var U = (_1: a, _2: new BigInteger(1), _3: new BigInteger(0));
        var V = (_1: b, _2: new BigInteger(0), _3: new BigInteger(1));
        while (V._1 != 0)
        {
            var q = U._1 / V._1;
            var T = (_1: U._1 % V._1, _2: U._2 - q * V._2, _3: U._3 - q * V._3);
            U = V;
            V = T;
        }

        z = Euclid(a, b);
        x = U._2;
        y = U._3;
    }

    public static BigInteger InversionMod(BigInteger a, BigInteger b)
    {
        if (Euclid(a, b) != 1)

```

```

        return 0;
    if (a < b)
        return 0;

    var U = (_1: a, _2: new BigInteger(1), _3: new BigInteger(0));
    var V = (_1: b, _2: new BigInteger(0), _3: new BigInteger(1));
    while (V._1 != 0)
    {
        var q = U._1 / V._1;
        var T = (_1: U._1 % V._1, _2: U._2 - q * V._2, _3: U._3 - q * V._3);
        U = V;
        V = T;
    }
    if (V._3 < 0)
        return U._3;
    return V._3 + U._3;
}

#endregion

public static CryptoInitializers MultiplicitySubGroupZp(Random random, int keySize)
{
    if (keySize % 2 != 0)
        throw new ArgumentException($"{nameof(keySize)} must be even");

    var q = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[keySize];
        while (true)
        {
            random.NextBytes(buffer);
            buffer[^1] |= 0x80;

            var value = new BigInteger(buffer, true);

            if (value.IsProbablyPrime(random) && (2 * value + 1).IsProbablyPrime(random))
                return value;
        }
    }))( );
    Console.WriteLine($"Q: {q}[{q.GetBitLength()}b]");

    var p = 2 * q + 1;
    Console.WriteLine($"P: {p}[{p.GetBitLength()}b]");

    var g = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(p - 1).GetByteCount()];
        while (true)
        {
            random.NextBytes(buffer);

            var value = new BigInteger(buffer, true);

            if (value > 1 && value < p - 1 && BigInteger.ModPow(value, q, p) != 1)
                return value;
        }
    }))( );
    Console.WriteLine($"G: {g}[{g.GetBitLength()}b]");

    return new CryptoInitializers

```

```

    {
        P = p,
        Q = q,
        G = g
    };
}

public static CryptoInitializers CyclingSubgroupPowerGp(Random random, int keySizeQ, int
keySizeP)
{
    BigInteger q;
    q = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[keySizeQ];
        while (true)
        {
            random.NextBytes(buffer);
            buffer[^1] |= 0x80;

            var value = new BigInteger(buffer, true);

            if (value.IsProbablyPrime(random))
                return value;
        }
    }))(());
    Console.WriteLine($"Q: {q}[{q.GetBitLength()}b]");

    BigInteger p;
    p = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[keySizeP - keySizeQ];
        while (true)
        {
            random.NextBytes(buffer);
            buffer[^1] |= 0x80;

            var value = (new BigInteger(buffer, true) * q + 1);

            if (value.IsProbablyPrime(random))
                return value;
        }
    }))(());
    Console.WriteLine($"P: {p}[{p.GetBitLength()}b]");

    var g = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[keySizeP - keySizeQ];
        while (true)
        {
            random.NextBytes(buffer);

            var r = new BigInteger(buffer, true);
            var g = BigInteger.ModPow(r, (p - 1) / q, p);

            if (g > 1 && BigInteger.ModPow(g, q, p) == 1)
                return g;
        }
    }))(());
    Console.WriteLine($"G: {g}[{g.GetBitLength()}b]");
}

```

```

        return new CryptoInitializers
        {
            P = p,
            Q = q,
            G = g
        };
    }

[Benchmark]
public void DiffieHellman()
{
    var rand = new Random();

    Console.WriteLine("\nDiffie Hellman\n");

    var init = MultiplicitySubGroupZp(rand, 2);

    //Person 1
    var a = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(init.P - 1).GetByteCount(true)];
        while (true)
        {
            rand.NextBytes(buffer);

            var value = new BigInteger(buffer, true);
            if (value < init.P - 1 && BigInteger.GreatestCommonDivisor(value, init.P - 1) == 1)
                return value;
        }
    })))();
    var A = BigInteger.ModPow(init.G, a, init.P);

    //Person 2
    var b = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(init.P - 1).GetByteCount(true)];
        while (true)
        {
            rand.NextBytes(buffer);

            var value = new BigInteger(buffer, true);
            if (value < init.P - 1 && BigInteger.GreatestCommonDivisor(value, init.P - 1) == 1)
                return value;
        }
    })))();
    var B = BigInteger.ModPow(init.G, b, init.P);

    var Zab = BigInteger.ModPow(B, a, init.P);
    var Zba = BigInteger.ModPow(A, b, init.P);

    Console.WriteLine($"Result Key Zab: {Zab}\n");
    Console.WriteLine($"Result Key Zba: {Zba}\n");
}

[Benchmark]
public void DiffieHellmanGroup()
{
    var rand = new Random();

    Console.WriteLine("\n Diffie Hellman New \n");

```

```

var init = CyclingSubgroupPowerGp(rand, 32, 128);

//Person 1
var a = ((Func<BigInteger>) (() =>
{
    var buffer = new byte[(init.P - 1).GetByteCount(true)];
    while (true)
    {
        rand.NextBytes(buffer);

        var value = new BigInteger(buffer, true);
        if (value < init.P - 1 && BigInteger.GreatestCommonDivisor(value, init.P - 1) == 1)
            return value;
    }
}))();
var A = BigInteger.ModPow(init.G, a, init.P);

//Person 2
var b = ((Func<BigInteger>) (() =>
{
    var buffer = new byte[(init.P - 1).GetByteCount(true)];
    while (true)
    {
        rand.NextBytes(buffer);

        var value = new BigInteger(buffer, true);
        if (value < init.P - 1 && BigInteger.GreatestCommonDivisor(value, init.P - 1) == 1)
            return value;
    }
}))();
var B = BigInteger.ModPow(init.G, b, init.P);

var Zab = BigInteger.ModPow(B, a, init.P);
var Zba = BigInteger.ModPow(A, b, init.P);

Console.WriteLine($"Result Key Zab: {Zab}\n");
Console.WriteLine($"Result Key Zba: {Zba}\n");
}
[Benchmark]
public void MQV()
{
    var rand = new Random();

    var keySizeP = 128;
    var keySizeQ = 32;

    var l = (keySizeQ * 8) / 2;

    Console.WriteLine("\n MQV \n");

    var init = CyclingSubgroupPowerGp(rand, keySizeQ, keySizeP);

    //Person 1
    var a = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(init.Q - 1).GetByteCount()];
        while (true)
        {
            rand.NextBytes(buffer);

```

```

        var value = new BigInteger(buffer, true);
        if (value < init.Q - 1 && BigInteger.GreatestCommonDivisor(value, init.Q - 1) == 1)
            return value;
    }
}));
var A = BigInteger.ModPow(init.G, a, init.Q);

//Person 2
var b = ((Func<BigInteger>) (() =>
{
    var buffer = new byte[(init.Q - 1).GetByteCount()];
    while (true)
    {
        rand.NextBytes(buffer);

        var value = new BigInteger(buffer, true);
        if (value < init.Q - 1 && BigInteger.GreatestCommonDivisor(value, init.Q - 1) == 1)
            return value;
    }
}));
var B = BigInteger.ModPow(init.G, b, init.Q);

{
    Console.WriteLine("Session");

    //Person 1
    var x = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(init.Q - 1).GetByteCount()];
        while (true)
        {
            rand.NextBytes(buffer);

            var value = new BigInteger(buffer, true);
            if (value < init.Q - 1 && BigInteger.GreatestCommonDivisor(value, init.Q - 1) ==
1)

                return value;
        }
    }));
    var X = BigInteger.ModPow(init.G, x, init.Q);

    //Person 2
    var y = ((Func<BigInteger>) (() =>
    {
        var buffer = new byte[(init.Q - 1).GetByteCount()];
        while (true)
        {
            rand.NextBytes(buffer);

            var value = new BigInteger(buffer, true);
            if (value < init.Q - 1 && BigInteger.GreatestCommonDivisor(value, init.Q - 1) ==
1)

                return value;
        }
    }));
    var Y = BigInteger.ModPow(init.G, y, init.Q);
    var d = BigInteger.Pow(2, 1) + (X % BigInteger.Pow(2, 1));
    var e = BigInteger.Pow(2, 1) + (Y % BigInteger.Pow(2, 1));
    var Sa = BigInteger.ModPow(Y * BigInteger.ModPow(B, e, init.Q) % init.Q, (x + d * a) %
init.P, init.Q);

```



```

        var Sb = BigInteger.ModPow(X * BigInteger.ModPow(A, d, init.Q) % init.Q, (y + e * b) %
init.P, init.Q);
        Console.WriteLine($"Result Key Sa: {Sa}\n");
        Console.WriteLine($"Result Key Sb: {Sb}\n");
    }
}

public static void Main(string[] args)
{
    var summary = BenchmarkRunner.Run(typeof(Program).Assembly);
    //DiffieHellman();
    //DiffieHellmanGroup();
    //MQV();
}
}

```