

Міністерство освіти та науки України
Харківський національний університет радіоелектроніки
Кафедра програмної інженерії

Лабораторна робота №3
з дисципліни: «Безпека програм та даних»
на тему: «Програмна реалізація хеш-функцій»

Виконав

ст. гр. ПЗПІ-20-1

Бабанін А.К.

Перевірив

доцент кафедри ПІ

Турута О.О.

2023

Мета роботи: Ознайомитись з можливостями криптографічних хеш-функцій при організації контролю цілісності цифрових об'єктів та отримати навички їх використання.

Хід роботи

1. Розробити свою хеш-функцію, яка в якості результату отримує значення 2, 4, 8 біт, яка при зміні будь-якого байту в тексті, призводить до зміни не менше 30% результату.

Для програмної реалізації хеш-функції була взята за основу проста і швидка хеш-функція MurmurHash. Алгоритм цієї функції був обран через його перевагу у значній зміні результату хешування від незначної зміни даних на вході.

Хеш-функція у якості вхідних параметрів отримує наступні зміни:

1. Массив байтів;
2. seed параметр для побудови сімейства хеш-значень у межах тестового застосування;
3. параметр *bitShift* для обмеження розміру хеш-значення.

Лістинг .net коду наведений далі:

```
private const ulong C1 = 0x87c37b91114253d5;
private const ulong C2 = 0x4cf5ad432745937f;

public static ulong Hash(byte[] data, ulong seed = 0, int bitShift = 8)
{
    ulong bitmask = (1UL << bitShift) - 1;
    int length = data.Length;
    int nblocks = length / 8;
    ulong h1 = seed;

    for (int i = 0; i < nblocks; i++)
    {
        ulong k1 = BitConverter.ToUInt64(data, i * 8);
        k1 *= C1;
        k1 = (k1 << 31) | (k1 >> 33);
        k1 *= C2;
        h1 ^= k1;
        h1 = (h1 << 27) | (h1 >> 37);
        h1 = h1 * 5 + 0x52dce729;
    }

    byte[] tail = new byte[8];
    Array.Copy(data, nblocks * 8, tail, 0, length % 8);

    ulong k2 = 0;

    if (length % 8 >= 7)
        k2 ^= (ulong)tail[6] << 48;
    if (length % 8 >= 6)
        k2 ^= (ulong)tail[5] << 40;
    if (length % 8 >= 5)
        k2 ^= (ulong)tail[4] << 32;
    if (length % 8 >= 4)
        k2 ^= (ulong)tail[3] << 24;
    if (length % 8 >= 3)
        k2 ^= (ulong)tail[2] << 16;
    if (length % 8 >= 2)
        k2 ^= (ulong)tail[1] << 8;
    if (length % 8 >= 1)
        k2 ^= tail[0];

    k2 *= C1;
    k2 = (k2 << 31) | (k2 >> 33);
    k2 *= C2;

    h1 ^= k2;

    h1 ^= (ulong)length;
    h1 ^= h1 >> 33;
    h1 *= 0xff51afd7ed558ccd;
    h1 ^= h1 >> 33;
    h1 *= 0xc4ceb9fe1a85ec53;
    h1 ^= h1 >> 33;

    return h1 & bitmask;
}
```

Для тестування функції використовувались рандомізовані тести у кількості залежній від *bitShift* параметру. У кожному тесту вхідна у вхідній строці змінювався випадковий байт.

Коефіцієнт схожості розраховувався за наступною формулою:

$$K = \frac{\text{кількість змінених бітів у хешу}}{\text{розмір хешу}}$$

Результати тестування наведені на рисунках 1-3.

```
Running 4 hash algorithm tests with 16 bit mask
Similar bits: 13(0.8125)
Similar bits: 12(0.75)
Similar bits: 12(0.75)
Similar bits: 11(0.6875)
```

Рис. 1 – Результати тестування для 4 бітного хешу

```
Running 8 hash algorithm tests with 8 bit mask
Similar bits: 3(0.375)
Similar bits: 4(0.5)
Similar bits: 5(0.625)
Similar bits: 3(0.375)
Similar bits: 4(0.5)
Similar bits: 3(0.375)
Similar bits: 3(0.375)
Similar bits: 4(0.5)
```

Рис. 2 – Результати тестування для 8 бітного хешу

```
Running 16 hash algorithm tests with 16 bit mask
Similar bits: 13(0.8125)
Similar bits: 12(0.75)
Similar bits: 12(0.75)
Similar bits: 11(0.6875)
Similar bits: 14(0.875)
Similar bits: 13(0.8125)
Similar bits: 9(0.5625)
Similar bits: 14(0.875)
Similar bits: 12(0.75)
Similar bits: 11(0.6875)
Similar bits: 11(0.6875)
Similar bits: 13(0.8125)
Similar bits: 11(0.6875)
Similar bits: 13(0.8125)
Similar bits: 12(0.75)
Similar bits: 11(0.6875)
```

Рис. 3 - Результати тестування для 16 бітного хешу

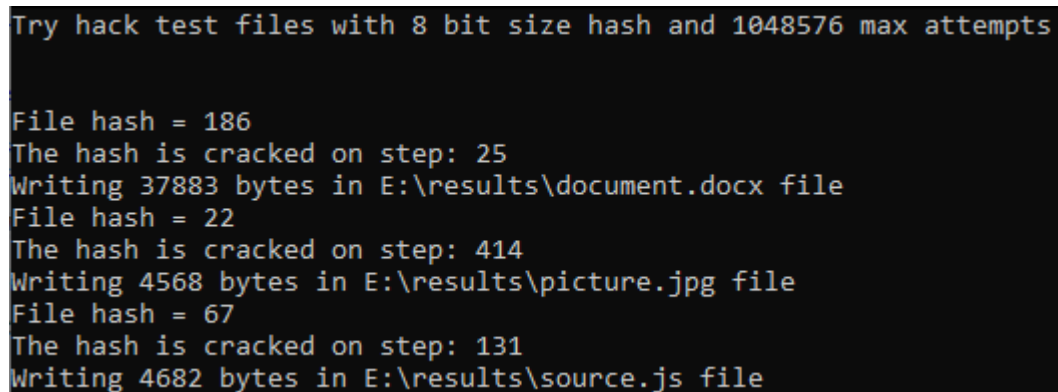
За результатами тестування, зміна будь-якого байту в тексті, призводить до зміни приблизно 25% результату.

2. Створити три документа (документ word, вихідний текст на будь-якій мові програмування source, зображення). Обчислити дайджест повідомлення для кожного з файлів. Внести необхідні зміни в документи, так, щоб дайджест повідомлення нового документа співпало зі старим. Реалізувати програму що дозволяє автоматично робити колізію хеш-функції.

Для атаки на дайджест повідомлення використовувався наступний brute-force алгоритм: починаючи з кінця файлу і-й байт послідовно змінювався на значення з проміжку [0, 255] до тих пір, поки:

1. Хеш-значення зміненого файлу не буде дорівнювати оригінальному значенню;
2. не буде вичерпаний ліміт brute-force ходів.

Результати виконання тестового запуску наведені на рис. 3.



```
Try hack test files with 8 bit size hash and 1048576 max attempts  
  
File hash = 186  
The hash is cracked on step: 25  
Writing 37883 bytes in E:\results\document.docx file  
File hash = 22  
The hash is cracked on step: 414  
Writing 4568 bytes in E:\results\picture.jpg file  
File hash = 67  
The hash is cracked on step: 131  
Writing 4682 bytes in E:\results\source.js file
```

Рис. 3 – Тестовий запуск brute-force атаки на дайджест

Зміни у source.js файлі (тільки текст) та у picture.png майже не помітні. Файли відкриваються без проблем. У випадку зображення, зміни візуально помітити неможливо. Зміни у файлах наведені на рис. 4-5.

<pre> strong> {clientKeys.private}</p> 142 <p>Client public key:</s trong> {clientKeys.public}</p> 143 </div> 144 </div> 145) 146 } 147 148 export default App </pre>	<pre> strong> {clientKeys.private}</p> 142 <p>Client public key:</s trong> {clientKeys.public}</p> 143 </div> 144 </div> 145) 146 } 147 148 export default Ap </pre>
---	--

Рис. 4 – Зміни у файлі source.js

<pre> 1100 F8 57 18 9E 12 C7 0C DB 65 1E A0 D0 E6 0B 58 1F mW.h.S.Ne. Pж.X. 1110 CD 4A E0 AA D6 B3 C3 84 49 81 34 0A B6 53 AE BA HJaEЦrГ,IF4.тS0e 1120 F2 41 F5 17 C1 05 73 8C A2 01 1D 05 71 2E F1 25 тАж.Б.сеў...q.c% 1130 28 96 32 EB A5 24 95 82 71 BB AD 04 91 98 9D 6E (-2пГ\$*,q»...'.кп 1140 46 C4 1F 22 92 48 59 13 A6 5C 8A A3 28 A5 34 C9 ФД."''НУ.:\ьУ(Г4Й 1150 24 B3 CD 1D 1C 32 61 18 DD C4 2D 7D D5 D8 AA 03 \$iH...2a.эд-)ХМЕ. 1160 45 82 49 20 34 15 2A AB B9 6E A0 89 DC C9 CA 49 E,I 4.*«Вп кьйКІ 1170 2A A2 0D 74 CA B5 9D 33 C4 6C E7 B9 E8 3A A4 92 *ў.тКккЗДлзМн:н' 1180 28 AD 88 CF 36 A3 A3 6F 45 03 22 60 63 6D 60 3F (.«П6УJoE." "cm'? 1190 1E A8 75 73 FC 7E A1 24 93 51 CB 5B 60 9E D2 58 .Еузь-ўф"ОЛ('HTX 11A0 D2 C9 B6 C3 EA BC F1 E6 E9 24 98 86 44 6B 21 BE тЙГГкјсжй\$.*Dk!s 11B0 41 F5 55 5C DC A4 92 B4 40 EC 64 58 27 5D 24 90 АхУ\ьн'г@ndX']\$b 11C0 B1 C7 0A 8E 4C 8B 24 92 88 A6 03 3B AE 94 92 4C ±S.тL<9'e!.;@''L 11D0 10 35 75 24 94 2C 9E D9 .Su\$",нм </pre>	<pre> 00001100 F8 57 18 9E 12 C7 0C DB 65 1E A0 D0 E6 0B 58 1F 00001110 CD 4A E0 AA D6 B3 C3 84 49 81 34 0A B6 53 AE BA 00001120 F2 41 F5 17 C1 05 73 8C A2 01 1D 05 71 2E F1 25 00001130 28 96 32 EB A5 24 95 82 71 BB AD 04 91 98 9D 6E 00001140 46 C4 1F 22 92 48 59 13 A6 5C 8A A3 28 A5 34 C9 00001150 24 B3 CD 1D 1C 32 61 18 DD C4 2D 7D D5 D8 AA 03 00001160 45 82 49 20 34 15 2A AB B9 6E A0 89 DC C9 CA 49 00001170 2A A2 0D 74 CA B5 9D 33 C4 6C E7 B9 E8 3A A4 92 00001180 28 AD 88 CF 36 A3 A3 6F 45 03 22 60 63 6D 60 3F 00001190 1E A8 75 73 FC 7E A1 24 93 51 CB 5B 60 9E D2 58 000011A0 D2 C9 B6 C3 EA BC F1 E6 E9 24 98 86 44 6B 21 BE 000011B0 41 F5 55 5C DC A4 92 B4 40 EC 64 58 27 5D 24 90 000011C0 B1 C7 0A 8E 4C 8B 24 92 88 A6 03 3B AE 94 92 4C 000011D0 10 35 75 24 94 2C FF D9 </pre>
--	--

Рис. 4 – Зміни у файлі picture.png

DOCX файл має певну структуру, тому заміна байта вмісту на випадкове значення призвело до того, що файл тепер неможливо відкрити.

Для вирішення цієї проблеми можна використовувати спеціальну варіацію використовованого алгоритму, яка працюватиме лише з байтами вмісту документа. Вміст документа може бути отриманий у результаті парсингу розмітки документа.

Висновки: під час виконання лабораторної роботи ознайомився з можливостями криптографічних хеш-функцій при організації контролю цілісності цифрових об'єктів та отримав навички їх використання.

ДОДАТОК А

Лістінг коду

```
FileCracker.cs

public class FileCracker
{
    public static void CrackFile(string sourcePath, string resultPath, int
hashSizeBit, int maxSteps)
    {
        var contentBuffer = new byte[1024 * 1024];

        using var fs = new FileStream(sourcePath, FileMode.Open,
FileAccess.Read);

        var contentLength = fs.Read(contentBuffer);
        var content = contentBuffer.AsSpan(0, contentLength).ToArray();
        var originalHash = MyHasher.Hash(content, Demo.Seed, hashSizeBit);
        var fileName = Path.GetFileName(sourcePath);
        System.Console.WriteLine("File hash = {0}", originalHash);
        var step = 0;
        // Try to change file char from the tail
        for (var i = content.Length - 1; i >= 0 && step < maxSteps; i-- )
        {
            var initialByte = content[i];
            // Iterate through all possible values
            for (var j = 0; j < 256; j++, step++)
            {
                if (step >= maxSteps)
                {
                    System.Console.WriteLine("The program has reached max step
count. Exiting.");
                }
                if (initialByte == (byte) j)
                {
                    continue;
                }

                content[i] = (byte)j;
            }
        }
    }
}
```

```

        var updatedHash = MyHasher.Hash(content, Demo.Seed,
hashSizeBit);

        if (updatedHash == originalHash)
        {
            var resultFilePath = Path.Combine(resultPath, fileName);
            System.Console.WriteLine("The hash is cracked on step: {0}
", step);

            System.Console.WriteLine("Writing {0} bytes in {1} file",
content.Length, resultFilePath);

            using var updateFs = File.Open(resultFilePath,
FileMode.Create, FileAccess.Write);
            updateFs.Write(content);

            return;
        }
    }

    // Restore initial content
    content[i] = initialByte;
}
}

```


Demo.cs

```
public class Demo
{
    public const int Seed = 1024;

    private static string[] TestFilePaths { get; } = new string[]
    {".\\Properties\\document.docx", ".\\Properties\\picture.jpg",
    ".\\Properties\\source.js"};

    public static void TryHackTestFiles(string resultFolderPath)
    {
        const int bitMask = 8;
        const int maxAttempts = 1024 * 1024;

        Console.WriteLine("Try hack test files with {0} bit size hash and
        {1} max attempts\\n\\n", bitMask, maxAttempts);

        foreach (var path in TestFilePaths)
        {
            FileCracker.CrackFile(path, resultFolderPath, bitMask,
maxAttempts);
        }
    }

    public static void TestHashAlgorithm()
    {
        var rnd = new Random(Seed);

        const int mask = 32;
        var maxIterations = mask;

        const string message = "It is reasonable to make p a prime number
        roughly equal to the number of characters in the input alphabet.";
        var messageBytes = Encoding.ASCII.GetBytes(message);
        var initialHash = MyHasher.Hash(messageBytes, mask);
```

```

var usedCombinations = new HashSet<KeyValuePair<int, byte>>();

var iterations = 0;

Console.WriteLine("Running {0} hash algorithm tests with {1} bit
mask", maxIterations, mask);

while (iterations < maxIterations)
{
    var changedMessageBytes = messageBytes.ToArray();

    var index = rnd.Next(0, messageBytes.Length);
    var _byte = (byte)rnd.Next(64, 128);

    var combination = new KeyValuePair<int, byte>(index, _byte);

    if (usedCombinations.Add(combination) == false)
    {
        continue;
    }

    changedMessageBytes[index] = _byte;

    var newHash = MyHasher.Hash(changedMessageBytes, mask);

    var similarBits = GetBitsSimilarityCoefficient(initialHash,
newHash, mask);
    var coef = (double)similarBits / mask;

    Console.WriteLine("Similar bits: {0}({1})", similarBits, coef);

    iterations++;
}
}

```

```

        private static int GetBitsSimilarityCoefficient(ulong a, ulong b, int
bitSize)
        {
            var a1 = (int)(a & ulong.MaxValue);
            var a2 = (int)(a >> 32);

            var b1 = (int)(b & ulong.MaxValue);
            var b2 = (int)(b >> 32);

            var bits1 = new BitArray(new int[] {a1, a2});
            var bits2 = new BitArray(new int[] { b1, b2 });

            var similarBits = 0;

            for (var i = 0; i < bitSize; i++)
            {
                if (bits1[i] == bits2[i])
                {
                    similarBits++;
                }
            }

            return similarBits;
        }
    }
}

```