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 контроля 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 значнім зміні результату хешування від незначної зміни даних на вході.

Хеш-функція у якості вхідних параметрів отримує наступні зміні:

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

Лістінг .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.

Коєфіціент схожості розраховувався за наступной формулою:

Результати тестування наведені на рисунках 1-3.

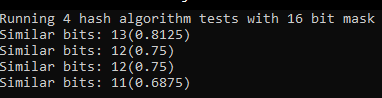


Рис. 1 – Результати тестування для 4 бітного хешу

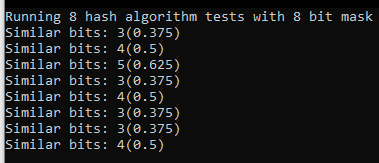


Рис. 2 – Результати тестування для 8 бітного хешу

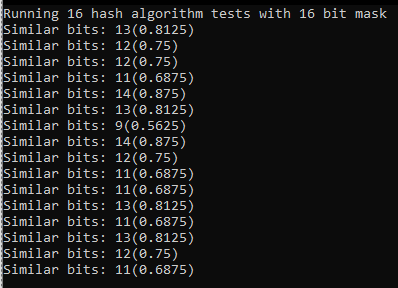


Рис. 3 - Результати тестування для 16 бітного хешу

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

1. 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 i-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.

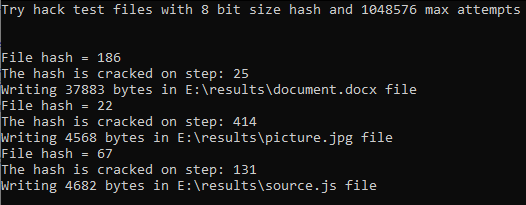


Рис. 3 – Тестовий запуск brute-force атаки на дайджест

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

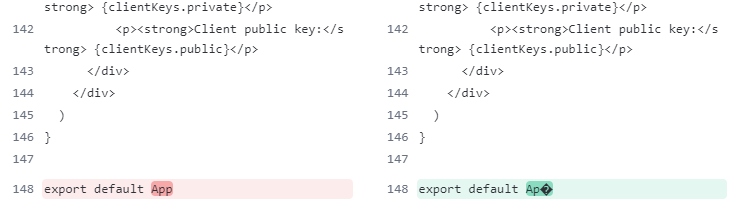


Рис. 4 – Зміни у файлі source.js

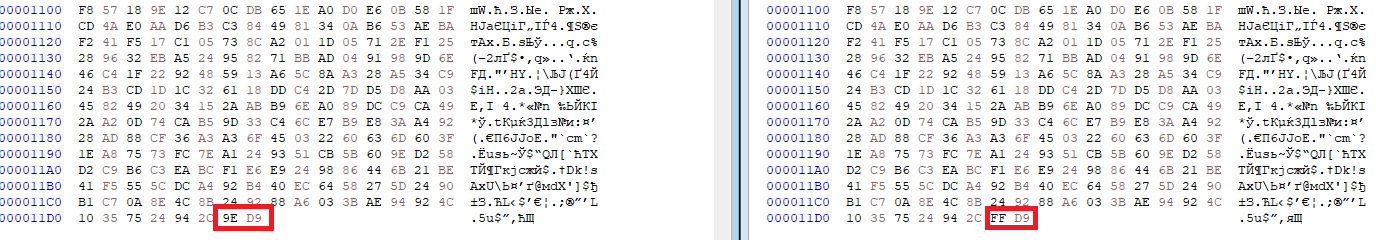


Рис. 4 – Зміни у файлі picture.png

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 використовуваного алгоритму, яка працюватиме лише з байтами вмісту документа. Вміст документа може бути отриманий у результати парсингу розмітки документа.

**Висновки:** під час виконання лабораторної роботи ознайомився з можливостями криптографічних хеш-функцій при організації контроля 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;

}

}

}