

Network Security and ManagementTerm Project

B0829039王語堂



[日期]

[公司名稱]

[公司地址]

操作說明

使用windows系統的電腦下載檔案，並確保該電腦可使用gcc及g++進行編譯，將zip下載完成後，將其解壓縮並放置於任意資料夾，但要注意放置txt文檔的路徑及其內文不能有中文，以cmd打開terminal，並cd至放置sha.cpp的檔案夾，若無執行檔，可使用g++ -o sha sha.cpp，便可看見生成了sha.exe檔案，接著使用sha.exe D:\\Vscode\\CPP\\DS\\short.txt false(路徑可自行指定，此處的路徑僅是說明用，檔案內附帶兩個一長一短的txt，分別叫做long.txt及short.txt，可以使用，或自行使用別的檔案)，會進入完整運作模式，可讀取路徑指定之txt檔案的內文，並將其進行SHA-512加密的結果以及效能輸出至螢幕上，並且會生成output.txt保留其結果。

(若文檔內容太少，在該模式下可能會因為運算速度太快，時間過短導致無法計算其效能)

一張含有 文字, 螢幕擷取畫面 的圖片

自動產生的描述

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

自動產生的描述

若是將最後的參數指定為true則會進入分解運作模式，可將Block及其內的各回合(round)結果輸出至螢幕(a~h)，其格式為16進制，同樣會生成output.txt保留其結果。

一張含有 文字, 螢幕擷取畫面, 軟體 的圖片

自動產生的描述

一張含有 文字, 螢幕擷取畫面, 軟體, 多媒體軟體 的圖片

自動產生的描述

一張含有 文字, 螢幕擷取畫面, 數字, 陳列 的圖片

自動產生的描述

程式說明

程式內容如下所說明:

初始化緩衝區: SHA-512算法首先初始化一個包含8個64bit值的初始hash value以及80個round constant。

填充文本: SHA-512接收到消息後，會對其進行填充以使其長度對1024位求餘的結果等於896(一個128bit(16byte)的block，表示原始消息的長度。

)。填充包括在文本結束後追加一個1，然後是一些0。

完成對輸入訊息的填充之後，會進行一系列的操作：

1. 分割為block：將填充後的訊息分割為多個1024bit的block。每一個塊都將獨立地進行後續的操作。

2. 設定訊息字：每一個1024位的塊都被進一步分割為16個64bit的字，這16個字被標記為W[0]到W[15]。這些就是我們的初始訊息字。

3. 擴展訊息字：接下來，將這16個訊息字擴展為80個64位的字。每一個新的字W[i]（對於i從16到79）都是從前16個字中計算出來的。這個計算包括一些位操作，像是右移和循環右移以及其他邏輯運算如XOR。

4. 設定工作變數：為每個1024位的塊設定8個工作變數，這8個變數是a, b, c, d, e, f, g, h，它們被初始化為內部hash狀態的當前值。

5. 主要循環：這裡將執行80次迴圈。在每一次迴圈中，SHA-512會根據當前的工作變數和當前的訊息字，以及一些預定義的常數和函數，來計算兩個新的值。這兩個新的值然後被用來更新工作變數。

6. 更新內部hash狀態：在每一個1024位的block的80次迴圈之後，工作變數的值被加到內部hash狀態，產生新的hash value。然後進入下一個1024位的block，直到所有的block都被處理過。

在結束所有的block的運算後最後剩下的hash value即是我們所需的SHA-512加密值。

函式及變數之簡短說明

const uint64\_t sha512\_initial\_hash[8]

用來放置初始hash value，讓整個sha512的round運算能夠開始，因為其格式為16位的16進位數字(16\*4=64bit)，因此使用uint64\_t儲存。

const uint64\_t sha512\_round\_constant[80]

hash round constants，每次hash需要運算80個rounds，這是其中所需的常數

uint64\_t rightrotate(uint64\_t value, int positions)

在運行hash時經常會需要做circular tight shift，因此新增一個rightrotate function來完成該任務

uint8\_t \*PadInput(const uint8\_t \*inputBytes, size\_t inputLength, size\_t &paddedLength)

負責進行填充，其運作過程會確保最後的16byte(128bit)能夠儲存原始資訊的長度，並在原始資訊的末尾填上10000…….(以下略)，同時讓填充完的資訊為128byte(1024bit)的倍數，好進行切割及運算

inline uint64\_t Ch(uint64\_t x, uint64\_t y, uint64\_t z) { return (x & y) ^ (~x & z); }

inline uint64\_t Maj(uint64\_t x, uint64\_t y, uint64\_t z) { return (x & y) ^ (x & z) ^ (y & z); }

inline uint64\_t Sigma0(uint64\_t x) { return rightrotate(x, 28) ^ rightrotate(x, 34) ^ rightrotate(x, 39); }

inline uint64\_t Sigma1(uint64\_t x) { return rightrotate(x, 14) ^ rightrotate(x, 18) ^ rightrotate(x, 41); }

每round的運算中，會運用到幾個運算函數（Bitwise Operation Function）：

Ch(x, y, z) = (x AND y) XOR (NOT x AND z)

Maj(x, y, z) = (x AND y) XOR (x AND z) XOR (y AND z)

Σ0(x) = ROTR^28(x) XOR ROTR^34(x) XOR ROTR^39(x)

Σ1(x) = ROTR^14(x) XOR ROTR^18(x) XOR ROTR^41(x)

因此將其寫成function備用。

void shaCompress(uint64\_t \*hash, const uint8\_t block[BlockSize], bool output)

該function負責對填充以及切割完的資料進行設定訊息字W[0:15]，擴展訊息字W[16:79]，設定工作變數，進行80個round的循環運算以持續更新8個工作變數，做addition更新hash value等功能，為了讓外部能取得該hash value，使用了指針，並加上了bool output這一參數。

int main(int argc, char \*argv[])

負責輸入命令列參數以決定文本路徑以及輸出與否，切割PadInput填充完的data並將其送入shaCompress，以及統計運算之效能。

程式

// 請注意，由於編碼問題，檔案名稱及檔案內容請勿包含中文

#include <array>

#include <chrono>

#include <cstddef>

#include <cstdint>

#include <cstring>

#include <fstream>

#include <iomanip>

#include <iostream>

#include <sstream>

#include <string>

using namespace std;

const string outputPath = "output.txt";

const int BlockSize = 128; // Block Size 128Bytes

// const int HashSize = 64;   // HashValue 64Bytes

const uint64\_t sha512\_initial\_hash[8] = {

    0x6A09E667F3BCC908, 0xBB67AE8584CAA73B, 0x3C6EF372FE94F82B, 0xA54FF53A5F1D36F1,

    0x510E527FADE682D1, 0x9B05688C2B3E6C1F, 0x1F83D9ABFB41BD6B, 0x5BE0CD19137E2179};

// 初始hash value

const uint64\_t sha512\_round\_constant[80] = {

    0x428a2f98d728ae22, 0x7137449123ef65cd, 0xb5c0fbcfec4d3b2f, 0xe9b5dba58189dbbc, 0x3956c25bf348b538,

    0x59f111f1b605d019, 0x923f82a4af194f9b, 0xab1c5ed5da6d8118, 0xd807aa98a3030242, 0x12835b0145706fbe,

    0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2, 0x72be5d74f27b896f, 0x80deb1fe3b1696b1, 0x9bdc06a725c71235,

    0xc19bf174cf692694, 0xe49b69c19ef14ad2, 0xefbe4786384f25e3, 0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65,

    0x2de92c6f592b0275, 0x4a7484aa6ea6e483, 0x5cb0a9dcbd41fbd4, 0x76f988da831153b5, 0x983e5152ee66dfab,

    0xa831c66d2db43210, 0xb00327c898fb213f, 0xbf597fc7beef0ee4, 0xc6e00bf33da88fc2, 0xd5a79147930aa725,

    0x06ca6351e003826f, 0x142929670a0e6e70, 0x27b70a8546d22ffc, 0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed,

    0x53380d139d95b3df, 0x650a73548baf63de, 0x766a0abb3c77b2a8, 0x81c2c92e47edaee6, 0x92722c851482353b,

    0xa2bfe8a14cf10364, 0xa81a664bbc423001, 0xc24b8b70d0f89791, 0xc76c51a30654be30, 0xd192e819d6ef5218,

    0xd69906245565a910, 0xf40e35855771202a, 0x106aa07032bbd1b8, 0x19a4c116b8d2d0c8, 0x1e376c085141ab53,

    0x2748774cdf8eeb99, 0x34b0bcb5e19b48a8, 0x391c0cb3c5c95a63, 0x4ed8aa4ae3418acb, 0x5b9cca4f7763e373,

    0x682e6ff3d6b2b8a3, 0x748f82ee5defb2fc, 0x78a5636f43172f60, 0x84c87814a1f0ab72, 0x8cc702081a6439ec,

    0x90befffa23631e28, 0xa4506cebde82bde9, 0xbef9a3f7b2c67915, 0xc67178f2e372532b, 0xca273eceea26619c,

    0xd186b8c721c0c207, 0xeada7dd6cde0eb1e, 0xf57d4f7fee6ed178, 0x06f067aa72176fba, 0x0a637dc5a2c898a6,

    0x113f9804bef90dae, 0x1b710b35131c471b, 0x28db77f523047d84, 0x32caab7b40c72493, 0x3c9ebe0a15c9bebc,

    0x431d67c49c100d4c, 0x4cc5d4becb3e42b6, 0x597f299cfc657e2a, 0x5fcb6fab3ad6faec, 0x6c44198c4a475817};

// hash round constants，每次hash需要運算80個rounds，這是其中所需的常數

uint64\_t rightrotate(uint64\_t value, int positions)

{

    return (value >> positions) | (value << (64 - positions));

}

uint8\_t \*PadInput(const uint8\_t \*inputBytes, size\_t inputLength, size\_t &paddedLength)

{

    size\_t originalLength = inputLength;

    size\_t paddingLength = (BlockSize - ((originalLength + 16) % BlockSize)); // 預留至少16Bytes(128bit)裝原始訊息長度，並計算剩餘可用作填充的空間

    paddedLength = originalLength + paddingLength + 16; // 保留最後用來裝原始訊息長度的16bytes

    uint8\_t \*paddedData = new uint8\_t[paddedLength];

    memcpy(paddedData, inputBytes, originalLength); // 保留原始資訊

    paddedData[originalLength] = 0x80;              // 最後面填充1然後後面放0，1000 0000

    // 填充0x00直到最後的16個位元組之前

    // 順便填充最後的16~9位元組

    for (size\_t i = originalLength + 1; i < paddedLength - 8; i++)

    {

        paddedData[i] = 0x00;

        // 由於使用uint8，因此每一位都是一個byte

    }

    uint64\_t bitLength = originalLength \* 8; // 算出原始長度有幾bit，存在最後16Byte中

    // 因為 inputLength（即原始消息的長度）被定義為 size\_t 類型

    // 而用來讀取的String尺寸也有限(8byte/64bit)

    // 在這種情況下，原始消息的長度不可能超過64位可以表示的最大值

    // 所以長度只需填充最後的64bit，最後128~65可以保持為0

    for (int i = 7; i >= 0; i--)

    {

        paddedData[paddedLength - 8 + i] = (bitLength >> ((7 - i) \* 8)) & 0xFF;

        //  依照big endian，最高位元放在最左側

        //  每個迴圈都向右移一個byte

        //  並使用位元運算符 & 配合遮罩 0xFF，將 bitLength 的位元內容截斷為 8 位元

        //  將結果存儲在 paddedData 的對應位元組中。直到最後

    }

    return paddedData;

    // 回傳填充好的陣列

}

inline uint64\_t Ch(uint64\_t x, uint64\_t y, uint64\_t z) { return (x & y) ^ (~x & z); }

inline uint64\_t Maj(uint64\_t x, uint64\_t y, uint64\_t z) { return (x & y) ^ (x & z) ^ (y & z); }

inline uint64\_t Sigma0(uint64\_t x) { return rightrotate(x, 28) ^ rightrotate(x, 34) ^ rightrotate(x, 39); }

inline uint64\_t Sigma1(uint64\_t x) { return rightrotate(x, 14) ^ rightrotate(x, 18) ^ rightrotate(x, 41); }

// 每round的運算中，會運用到幾個運算函數（Bitwise Operation Function）：

// Ch(x, y, z) = (x AND y) XOR (NOT x AND z)

// Maj(x, y, z) = (x AND y) XOR (x AND z) XOR (y AND z)

// Σ0(x) = ROTR^28(x) XOR ROTR^34(x) XOR ROTR^39(x)

// Σ1(x) = ROTR^14(x) XOR ROTR^18(x) XOR ROTR^41(x)

void shaCompress(uint64\_t \*hash, const uint8\_t block[BlockSize], bool output)

{

    // 設定輸入檔案，以續寫輸入

    ofstream file;

    file.open(outputPath, ios::app);

    uint64\_t W[80];

    // 每個word需要為64bit長，因此用uint64\_t，每位有8byte

    for (int i = 0; i < 16; i++)

    {

        W[i] = 0;

        for (int j = 0; j < 8; j++)

        {

            W[i] |= static\_cast<uint64\_t>(block[i \* 8 + j]) << (56 - j \* 8);

            // 將當前的block的該字節(byte)轉換為64bit(8byte)格式儲存

        }

        // 設定前16(0~15)位訊息字

    }

    for (int i = 16; i < 80; i++)

    {

        uint64\_t s0 = rightrotate(W[i - 15], 1) ^ rightrotate(W[i - 15], 8) ^ (W[i - 15] >> 7);

        uint64\_t s1 = rightrotate(W[i - 2], 19) ^ rightrotate(W[i - 2], 61) ^ (W[i - 2] >> 6);

        W[i] = W[i - 16] + s0 + W[i - 7] + s1;

        // 以運算擴展出16~79位訊息字

    }

    uint64\_t a = hash[0];

    uint64\_t b = hash[1];

    uint64\_t c = hash[2];

    uint64\_t d = hash[3];

    uint64\_t e = hash[4];

    uint64\_t f = hash[5];

    uint64\_t g = hash[6];

    uint64\_t h = hash[7];

    // 主要循環，做80個round的運算

    for (int t = 0; t < 80; t++)

    {

        uint64\_t T1 = h + Sigma1(e) + Ch(e, f, g) + sha512\_round\_constant[t] + W[t];

        uint64\_t T2 = Sigma0(a) + Maj(a, b, c);

        h = g;

        g = f;

        f = e;

        e = d + T1;

        d = c;

        c = b;

        b = a;

        a = T1 + T2;

        if (output == true)

        {

            cout << "\tround:" << dec << t + 1 << endl;

            cout << "\t\ta:" << setfill('0') << setw(16) << hex << a << endl;

            cout << "\t\tb:" << setfill('0') << setw(16) << hex << b << endl;

            cout << "\t\tc:" << setfill('0') << setw(16) << hex << b << endl;

            cout << "\t\td:" << setfill('0') << setw(16) << hex << d << endl;

            cout << "\t\te:" << setfill('0') << setw(16) << hex << e << endl;

            cout << "\t\tf:" << setfill('0') << setw(16) << hex << f << endl;

            cout << "\t\tg:" << setfill('0') << setw(16) << hex << g << endl;

            cout << "\t\th:" << setfill('0') << setw(16) << hex << h << endl;

            file << "\tround:" << dec << t + 1 << endl;

            file << "\t\ta:" << setfill('0') << setw(16) << hex << a << endl;

            file << "\t\tb:" << setfill('0') << setw(16) << hex << b << endl;

            file << "\t\tc:" << setfill('0') << setw(16) << hex << b << endl;

            file << "\t\td:" << setfill('0') << setw(16) << hex << d << endl;

            file << "\t\te:" << setfill('0') << setw(16) << hex << e << endl;

            file << "\t\tf:" << setfill('0') << setw(16) << hex << f << endl;

            file << "\t\tg:" << setfill('0') << setw(16) << hex << g << endl;

            file << "\t\th:" << setfill('0') << setw(16) << hex << h << endl;

        }

    }

    // round完成，做addtion，更新hash

    hash[0] += a;

    hash[1] += b;

    hash[2] += c;

    hash[3] += d;

    hash[4] += e;

    hash[5] += f;

    hash[6] += g;

    hash[7] += h;

    file.close();

}

int main(int argc, char \*argv[])

{

    // string path = "D:\\Vscode\\CPP\\DS\\sha.txt";

    // bool output = false;

    string path = argv[1];

    bool output = false;

    if (strcmp(argv[2], "true") == 0 || strcmp(argv[2], "True") == 0 || strcmp(argv[2], "1") == 0)

    {

        output = true;

        cout << "rounds output mode" << endl;

    }

    else if (strcmp(argv[2], "false") == 0 || strcmp(argv[2], "False") == 0 || strcmp(argv[2], "0") == 0)

    {

        output = false;

        cout << "rounds hide mode" << endl;

    }

    else

    {

        cout << "parameters error!" << endl;

        return 1;

    }

    ifstream file(path);

    if (file.is\_open())

    {

        stringstream buffer;

        buffer << file.rdbuf();

        string content = buffer.str(); // read file

        uint8\_t contentBytes[content.length()];

        memcpy(contentBytes, content.data(), content.length());

        size\_t paddedLength = 0; // inintial paddedLength

        auto start\_time = chrono::high\_resolution\_clock::now();

        uint8\_t \*paddedInput = PadInput(contentBytes, content.length(), paddedLength); // 以128Byte(1024bit)為單位計算填充量

        uint64\_t hash[8];

        for (int i = 0; i < 8; i++)

        {

            hash[i] = sha512\_initial\_hash[i];

        }

        // 初始化hash value

        // 清空輸出檔，或著創建

        ofstream cleanOutput(outputPath, ios::trunc);

        cleanOutput.close();

        for (size\_t i = 0; i < paddedLength; i += BlockSize)

        {

            uint8\_t block[BlockSize];

            memcpy(block, &paddedInput[i], BlockSize);

            // 現在，block 陣列包含了當前的 128 byte 區塊

            // 接著對每個區塊執行 SHA-512 壓縮函數

            ofstream outputfile(outputPath, ios::app);

            if (output == true)

            {

                cout << dec << "Block:";

                outputfile << dec << "Block:";

                if (i == 0)

                {

                    cout << dec << i + 1 << endl;

                    outputfile << dec << i + 1 << endl;

                }

                else

                {

                    cout << i / 128 + 1 << endl;

                    outputfile << i / 128 + 1 << endl;

                }

            }

            outputfile.close();

            shaCompress(hash, block, output);

        }

        auto end\_time = chrono::high\_resolution\_clock::now();

        auto duration = chrono::duration\_cast<std::chrono::microseconds>(end\_time - start\_time).count() / 1000000.0;

        ofstream lasOutputfile(outputPath, ios::app);

        // 紀錄整個計算所花費的時間

        if (duration > 0)

        {

            cout << dec << "Performance: " << fixed << setprecision(5) << (paddedLength / duration) << " bytes/second" << endl;

            lasOutputfile << dec << "Performance: " << fixed << setprecision(5) << (paddedLength / duration) << " bytes/second" << endl;

        }

        else

        {

            cout << dec << "The computation is too fast to measure the performance accurately." << endl;

            lasOutputfile << dec << "The computation is too fast to measure the performance accurately." << endl;

        }

        // 輸出以(bytes/second)為單位的效能

        delete[] paddedInput;

        stringstream final\_hash;

        cout << dec << "The Sha-512 result is:" << endl;

        lasOutputfile << dec << "The Sha-512 result is:" << endl;

        for (int i = 0; i < 8; i++)

        {

            final\_hash << setfill('0') << setw(16) << hex << hash[i];

        }

        // 輸出加密完成的結果

        cout << final\_hash.str() << endl;

        lasOutputfile << final\_hash.str() << endl;

        lasOutputfile.close();

        file.close();

    }

    else

    {

        cout << "Failed to open the file." << std::endl;

    }

    return 0;

}

// reference:

// https://en.wikipedia.org/wiki/SHA-2

// https://medium.com/@zaid960928/cryptography-explaining-sha-512-ad896365a0c1

// https://www.youtube.com/watch?v=orIgy2MjqrA&t=102s

心得

在完成這項SHA-512編碼作業的過程中，我獲得了許多寶貴的經驗與深入的學習機會。我對資訊安全、密碼學以及資訊理論有了更深的理解，並透過實作，親手將抽象的理論轉化為具體的程式碼。

在寫作業過程中，我深入了解了SHA-512的內部運作原理，例如如何將輸入資料拆分成不同的區塊、如何應用不同的變換函數，以及如何將過程中的運算結果組合成最終的雜湊值。

在應對挑戰與解決問題的過程中，我也體認到細節的重要性，以及對於每一行程式碼都必須深入理解的必要性。由於SHA-512是一種相當複雜的雜湊函數，程式中任何一點小錯誤都可能導致最終結果的巨大變化，因此我必須謹慎地對待每一個步驟。此外，透過不斷的除錯與優化，我進一步熟悉了C++的特性與語法，提高了我的程式設計技巧。

例如，我學會了怎麼適當地使用C++中除了int ,double ,float以外的數字類型，如uint64\_t , size\_t, uint32\_t, uint8\_t，來處理大量的數據。我也學習了如何在C++中使用位元運算，以及如何有效地進行數據轉換。

在開始實作前，我首先要清楚理解SHA-512的算法流程和數據結構，這讓我能夠有效地組織我的程式碼，並減少不必要的錯誤，在過程中亦要不斷的考慮如何實現SHA-512的運算邏輯所需的函式，並持續搜索資訊以更加詳細的對SHA-512進行理解，並對自身程式的邏輯進行驗證。

此次的學習過程也讓我深刻體驗到理論與實作的結合之重要。理解密碼學理論與SHA-512的運作原理固然重要，但是能夠將這些理論落實到程式碼中，將抽象的概念具現化，更是讓我獲益良多。此外，我也明白到自我學習的重要性，包括如何有效地搜尋資訊，如何獨立解決問題，以及如何反思並改進自己的作品。

總結來說，這次的作業給了我一個很好的機會去研究並實現一種複雜的雜湊演算法。雖然過程中遇到了一些挑戰，但正是這些挑戰讓我有機會去學習和成長。我相信這次的經驗將對我在資訊安全算法的理解上有著深遠的影響。

參考資料

<https://en.wikipedia.org/wiki/SHA-2>

<https://medium.com/@zaid960928/cryptography-explaining-sha-512-ad896365a0c1>

<https://www.youtube.com/watch?v=orIgy2MjqrA&t=102s>

<https://www.youtube.com/watch?v=JViXozmJnSk&t=36s>

基本要求

⚫ 至少支援 SHA-512 ☑

⚫ 提供使用者以命令列參數指定輸入檔案，並將結果輸出至螢幕。☑

⚫ 提供完整與分解運作模式，分解運作模式係指將各回合(round)結果輸出至螢幕。輸 出格式可自行決定，但可讀性將列入計分。 ☑

⚫ 輸出運算效能 (bytes/second) ☑