ZIP、完全に理解した


PK0304=Local
Ver.2.0
Stored
Time
date
CRC
filename
key0
key1
key2
0101110
1101101
1000000
1011010

はじめに

前回の夏コミ、コミックマーケット C92 では CTF を開催し、問題の解説を本にして頒布しました。CTF と一口に言っても問題のジャンルは多岐にわたり、その分、それぞれジャンルの説明が薄くなってしまったことが心残りでした。今度は何か一つのことを掘り下げてみたいと考え、ZIP をテーマに選びました。

ZIP は現在最も広く使われているアーカイブフォーマットでしょう。.zip という拡張子の付いたファイルはもちろんのこと、Microsoft Office のファイルや、Android のアプリケーション、Java の.jar ファイルなども実は ZIP です。

単にファイルを圧縮してまとめる、あるいはそのような ZIP を解凍するだけならば、ZIP はとても簡単に扱うことができます。サークルカットに書いたように、紙と鉛筆で作れるほどです。一方、ZIP の規格には多くの技術が取り込まれています。各種の圧縮アルゴリズムや暗号化、4GB 以上の（32 bit では表現できない）サイズのファイルの取り扱い、ファイル名の Unicode 対応、ファイルパーミッションやシンボリックリンクのサポート、ZIP の分割、さらには公開鍵暗号による暗号化や電子署名にも対応しています（このあたりも解説したかったのですが、時間が足りなくて削りました ……）。

ZIP で特に面白いのは Traditional PKWARE Encryption による暗号化です。3 章で詳述するようにこの暗号は脆弱です。より強力な暗号化に対応した仕様は特許で保護されているためサポートしているソフトウェアが少なく、暗号を目的として ZIP を使用する場合には今でもこの暗号化が広く用いられています。「添付ファイルは暗号化しております。パスワードは後ほど別のメールでお送りいたします」という文面で送付される ZIP ファイルはこの暗号でしょう。AES などのしっかりとした暗号と異なり解読できないわけではなく、かといって単に xor した暗号のように容易に解読できるわけでもなく、ちょうど良い難易度です。

なお、タイトルの「完全に理解した」の元ネタは「ポプテピピック」（1 巻 55 ページ）です。どういう文脈のセリフなのかは、元ネタを参照してください。

目次

第 1 章

準備

プログラミング言語

本文中のコードは C++ で記述しています。 Visual C++ 2017 と、GCC 4.8.4 でコンパイルできることを確認しています。C++11 の機能を使用しているので、GCC や Clang でコンパイルする際には`-std=c++11` オプションを追加してください。

本文中のコード断片は、別のコード断片で定義した関数や変数を参照しています。コンパイルして動かす際には適宜コピーしてください。奥付のサイトで本文中のソースコードを公開しています。

本書の構成

2 章では、ZIP で使われている CRC や Deflate アルゴリズムの説明を交えながら、ZIP の構成を解説しています。

3 章では ZIP で用いられている暗号の攻撃方法を紹介しています。

第 2 章

ZIP の構成

概要

ZIP は全体は一般的に次のように構成されています。暗号化や 64 ビットのサイズを扱う場合には他にも項目が追加されます。

項目
1 個目のファイルの local file header
1 個目のファイル
2 個目のファイルの local file header
2 個目のファイル
:
n 個目のファイルの local file header
n 個目のファイル
1 個目のファイルの central directory header
2 個目のファイルの central directory header
:
n 個目のファイルの central directory header
end of central directory record

ZIP では、local file header の直後に対応するファイルの中身が続くことと、各 central directory header が連続していること以外の各項目の位置の制約がありません。これは他のファイルフォーマットではあまり見ない特徴です。

自己解凍形式の ZIP では、1 個目のファイルの local file header の前にプログラムを置いて、OS が ZIP 自体を実行可能ファイルとして実行できるようにしています。先頭にプログラムがあっても仕様に則った ZIP なので、ファイルに置かれたプログラム以外の解凍ソフトでも解凍することができます。

さらに、end of central directory record の後に何らかのデータを置くこともできます。 次節で説明するように end of central directory はコメントを含んでいて可変長なので、そもそも解凍する際にファイル末尾から一定の距離を遡って end of central directory record とみなすことはできません。

End of central directory record には 1 個目のファイルの central directory header の位置が、各 central directory header には対応するファイルの local file header の位置が記録されています。解凍ソフトが ZIP を解凍するときには、この順で ZIP の中身を辿ります。ファイルの末尾に存在するとは限らないとなると、どのようにして central directory header の位置を知れば良いのでしょうか？ 仕様書にはその方法は書かれていません。ZIP の実装の一つである Info-Zip[*1]ではファイル全体から end of central directory record のシグネチャを探していました（zip 3.0 の zipfile.c を `ENDSIG` で検索すると処理が見つかります）。検索は末尾から行われていたので、やはり end of central directory record は末尾に置くと解凍するプログラムに優しいようです。

End of central directory record

Local file header と central directory record は ZIP 中の各ファイルに対応しているので、ファイルを 1 個も含まない空の ZIP は、end of central directory record を書き出すだけで作ることができます。

End of central directory record の構成は下表の通りです。

項目名	サイズ	説明
signature	4 bytes	`0x06054b50`
disk	2 bytes	このディスクが何番目か
centralDirDisk	2 bytes	central directory が開始するディスクの番号
entryNumber	2 bytes	このディスクの central directory 中のエントリー数
totalEntryNumber	2 bytes	central directory 中のエントリーの総数
centralDirSize	4 bytes	central directory のサイズ
centralDirOffset	4 bytes	central directory のファイル先頭からのオフセット

[*1] http://www.info-zip.org/

項目名	サイズ	説明
commentLength	2 bytes	コメントの長さ
comment	可変長	コメント

項目名は筆者が便宜上付けたもので、仕様書には出てきません。例えば「number of the disk with the start of the central directory」のように書かれています。

signature は、常に 0x06054b50 です。全ての数値はリトルエンディアンで ZIP 中に格納されます。例えば、0x06054b50 は、50 4b 05 06 となります。50 と 4b は ASCII 文字コードの P と K です。全てのシグネチャの下位 2 bytes は 0x4b50 なので、ZIP をバイナリディタで見ると、あちこちに PK の文字列が現れます。

disk と centralDirDisk は、ZIP を分割する際に使われます。単一のファイルの場合はどちらも 0 になります。

Central directory のエントリーは ZIP 中のファイルと（ディレクトリを個別のエントリーで表す場合には）ディレクトリに対応しているので、その合計数が totalEntryNumber です。単一のファイルの場合には、entryNumber は totalEntryNumber と等しくなります。

次のプログラムで空の ZIP ファイルを出力することができます。

```
#include <cstdio>
#include <cstdint>
#include <vector>
#include <map>
#include <algorithm>
#include <ctime>
#include <random>
using namespace std;

#pragma pack(push, 1)

struct EndOfCentralDirectoryRecord
{
    uint32_t signature = 0x06054b50u;
    uint16_t disk = 0;
    uint16_t centralDirDisk = 0;
    uint16_t entryNumber = 0;
    uint16_t totalEntryNumber = 0;
```

```
    uint32_t centralDirSize = 0;
    uint32_t centralDirOffset = 0;
    uint16_t commentLength = 0;
};

#pragma pack(pop)

int main()
{
    FILE *f = fopen("empty.zip", "wb");
    EndOfCentralDirectoryRecord end;
    fwrite(&end, sizeof end, 1, f);
    fclose(f);
}
```

```
00000000  50 4b 05 06 00 00 00 00  00 00 00 00 00 00 00 00  |PK..............|
00000010  00 00 00 00 00 00                                 |......|
00000016
```

後の実装で必要になるヘッダもここでまとめて`#include`しています。

`#pragma pack(push, 1)`はコンパイラが構造体のメンバーの間に隙間を空けることを抑止するためのものです。`#pragma`での指定に対応しているかどうかはコンパイラによって異なります。Visual C++ と GCC はこの指定を解釈します。また、ビッグエンディアンの環境ではファイルに書き出す前にエンディアンを変換する必要があります。

commentLength にコメントの長さを指定してコメントを書き出すことにより、ZIP にコメントを付けることができます。アーカイバによっては何らかの方法でこのコメントを表示します。

```
int main()
{
    FILE *f = fopen("comment.zip", "wb");
    EndOfCentralDirectoryRecord end;
    end.commentLength = 7;
    fwrite(&end, sizeof end, 1, f);
    fwrite("comment", 1, 7, f);
    fclose(f);
}
```

```
00000000  50 4b 05 06 00 00 00 00  00 00 00 00 00 00 00 00  |PK..............|
00000010  00 00 00 00 07 00 63 6f  6d 6d 65 6e 74           |......comment|
0000001d
```

図 2.1　SecureZIP のコメントの表示

CRC

空の ZIP が作れたので次はファイルを追加したいところですが、ZIP にファイルを追加するためにはファイルの CRC（Cyclic Redundancy Check、巡回冗長検査）を計算する必要があります。CRC は、計算対象のデータを数値とみなして、仕様で定められた値で除算したときの余りです。ただし、減算の代わりに xor を用います。例えば、計算対象のデータが `be ef`（`10111110 11101111`）で、除数が `0x1b7`（`1 10110111`）のときの CRC の算出は次のようになり、CRC は `c4`（`00100011`）です。計算対象のデータと CRC は下位ビットを上位ビットとして取り扱います。

```
          01011100
         ------------------
1 10110111)01111101 11110111
            1101101 11
           -----------------
              10000 00110111
              11011 0111
           -----------------
                1011 01000111
                1101 10111
           -----------------
                 110 11111111
                 110 110111
           -----------------
                       00100011
```

ZIP の除数は 0x104c11db7 です。さらに、計算対象のデータの末尾に 00 00 00 00 を追加し、先頭から 32 ビットを反転し、計算結果の CRC も反転させるという処理が加わります。この CRC の計算方法は ZIP に限らず、画像フォーマットの PNG などでも用いられています。

アルゴリズムを素直に実装してみます。

```
uint32_t crc_tmp1(vector<uint8_t> data)
{
    //  被除数
    vector<int> a;
    for (size_t i=0; i<data.size(); i++)
        for (int j=0; j<8; j++)
            a.push_back(data[i]>>j&1);
    for (int i=0; i<32; i++)
        a.push_back(0);
    for (int i=0; i<32; i++)
        a[i] ^= 1;
    //  除数
    vector<int> b(1, 1);
    for (int i=31; i>=0; i--)
        b.push_back(0x04C11DB7>>i&1);
    //  除算
    for (size_t i=0; i<data.size()*8; i++)
```

```
        if (a[i]!=0)
            for (int j=0; j<33; j++)
                a[i+j] ^= b[j];
    //  CRC
    uint32_t c = 0;
    for (int i=0; i<32; i++)
        c |= a[data.size()*8+i]<<i;
    return ~c;
}

vector<uint8_t> stringToUint8(string str)
{
    vector<uint8_t> uint8;
    for (char c: str)
        uint8.push_back((uint8_t)c);
    return uint8;
}

int main()
{
    printf("%08x\n", crc_tmp1(stringToUint8("understand zip")));
    //  3cde314f
}
```

PNG の仕様書に記載されている CRC のサンプルコード[*2]はより簡潔で `vector<int> a` のような追加のメモリも使用していません。上記のソースコードを最適化していくと、このような形になります。

計算の過程で記憶しておかなければいけないのは 32 ビット分だけです（下のソースコードの `c`）。最後のビットの上位と下位を逆転させる処理を省くため、この変数は最初から逆転しておきます。1 ビット分の処理は、最下位ビットを保存しておき、右シフトして入力を最上位ビットに追加し、最下位ビットが 1 ならば除数（を逆転させたもの）との xor を取ることです。

```
uint32_t crc_tmp2(vector<uint8_t> data)
{
    uint32_t c = 0;
    for (size_t i=0; i<data.size()+4; i++)
        for (int j=0; j<8; j++) {
```

[*2] https://www.w3.org/TR/2003/REC-PNG-20031110/#D-CRCAppendix

```
            int lsb = c&1;
            c = ((i<data.size() ? data[i]>>j&1 : 0) ^ (i<4 ? 1 : 0))<<31 | c>>1;
            if (lsb != 0)
                c ^= 0xedb88320;
        }
    return ~c;
}
```

入力データが処理に影響を与えるのは、最下位ビットに達したときなので読むのを遅延させることができます。入力データは除数と xor されます。c の初期値を 0xffffffff とすることで、入力データの先頭 32 bits を反転させる処理に変えることができます。

```
uint32_t crc_tmp3(vector<uint8_t> data)
{
    uint32_t c = 0xffffffffu;
    for (size_t i=0; i<data.size(); i++)
        for (int j=0; j<8; j++) {
            int lsb = c&1 ^ data[i]>>j&1;
            c >>= 1;
            if (lsb != 0)
                c ^= 0xedb88320;
        }
    return ~c;
}
```

このとき、内側のループの処理は c の下位 8 bits と data[i] の xor を取った値にしか依存していません。あらかじめ値を計算しておくことができます。

```
uint32_t crcTable[256];
void makeCrcTable()
{
    for (int i=0; i<256; i++) {
        uint32_t c = i;
        for (int j=0; j<8; j++)
            c = c>>1 ^ ((c&1)!=0 ? 0xedb88320 : 0);
        crcTable[i] = c;
    }
}
```

```
uint32_t crc32(uint32_t c, uint8_t d)
{
    return c>>8 ^ crcTable[c&0xff^d];
}

uint32_t crc(vector<uint8_t> data)
{
    uint32_t c = 0xffffffffu;
    for (size_t i=0; i<data.size(); i++)
        c = crc32(c, data[i]);
    return ~c;
}
```

暗号の処理でも使用するため、1 バイト分の処理を関数 crc32 に切り出しています。

Local file header/Central directory header

Local file header の構成は次の通りです。

項目名	サイズ	説明
signature	4 bytes	0x04034b50
versionNeeded	2 bytes	解凍に必要なバージョン
flag	2 bytes	フラグ
compression	2 bytes	圧縮方法
modifiedTime	2 bytes	更新時刻
modifiedDate	2 bytes	更新日付
crc	4 bytes	ファイルの CRC
compressedSize	4 bytes	圧縮後のファイルのサイズ
uncompressedSize	4 bytes	圧縮前のファイルのサイズ
fileNameLength	2 bytes	ファイル名の長さ
extraFieldLength	2 bytes	拡張フィールドの長さ
fileName	可変長	ファイル名
extraField	可変長	拡張フィールド

versionNeeded で、このファイルを解凍するために準拠している必要がある仕様書のバージョンを指定します。仕様書のバージョンの小数点を取り除いた（10 倍した）数値を指定します。例え

ば、圧縮方法として BZIP2 が使用可能になったのは 4.6 からなので、BZIP2 を使用した場合には 46 になります。この章で取り扱う圧縮や暗号化は 20 を指定すれば足ります。その他のバージョンは ZIP の仕様書を参照してください。

ファイルを暗号化したり、ファイルサイズの指定を後回しにしたりする場合には、flag の特定のビットを立てます。

compression は圧縮方法を示します。主に使用されるのは、0 の無圧縮と、8 の Deflate 圧縮です。

modifiedTime と modifiedDate には、MS-DOS のフォーマットで更新日時を指定します。特定の日時からの経過秒数ではなく、下表の通り各ビットに年や分などを設定します。CRC の算出とは異なり、最下位ビットを 0 としています。[*3]

	ビット	内容
modifiedDate	0-4	日（1-31）
	5-8	月（1-12）
	9-16	年-1980
modifiedTime	0-4	秒/2（0-29）
	5-10	分（0-59）
	11-15	時（0-23）

16 ビットに詰め込むために秒を 2 で割るので、更新時刻に奇数秒を指定することはできません。また、日時はローカル日時です。異なるタイムゾーンの環境に ZIP を持っていっても更新日時は変化しません（実際とは異なる日時にファイルが更新されたように見えます）。

Central directory header の構成は次の通りです。

項目名	サイズ	説明
signature	4 bytes	`0x02014b50`
versionNeeded	2 bytes	解凍に必要なバージョン
versionMade	2 bytes	圧縮に使用されたソフトウェアのバージョン
flag	2 bytes	フラグ
compression	2 bytes	圧縮方法
modifiedTime	2 bytes	更新時刻
modifiedDate	2 bytes	更新日付

[*3] https://msdn.microsoft.com/ja-jp/library/cc429703.aspx

項目名	サイズ	説明
crc	4 bytes	ファイルの CRC
compressedSize	4 bytes	圧縮後のファイルのサイズ
uncompressedSize	4 bytes	圧縮前のファイルのサイズ
fileNameLength	2 bytes	ファイル名の長さ
extraFieldLength	2 bytes	拡張フィールドの長さ
commentLength	2 bytes	コメントの長さ
disk	2 bytes	対応する Local file header が存在するディスクの番号
internalAttr	2 bytes	内部属性
externalAttr	4 bytes	外部属性
offset	4 bytes	対応する Local file header のファイル先頭からのオフセット
fileName	可変長	ファイル名
extraField	可変長	拡張フィールド
comment	可変長	コメント

Local file header と重複している（signature 以外の）項目は同じ値を設定します。

versionMade にはこの ZIP を作成したソフトウェアの情報を記載します。下位バイトは versionNeeded と同様に作成したソフトウェアが準拠している仕様書のバージョンを記載します。versionNeeded 以上の値になるでしょう。上位バイトはファイルの属性の互換性を示します。MS-DOS（FAT32）は 0、UNIX は 3、Macintosh は 7、Windows（NTFS）は 10 です。実際の圧縮元のファイルシステムではなく、externalAttr の内容を示すものなので、作成した ZIP の互換性を考えると、0 を指定して externalAttr は MS-DOS の形式で指定するのが良いようです。

internalAttr にはファイルシステムのファイルの属性には含まれないファイルの情報が格納されます。最下位ビットが 0 ならばバイナリファイル、1 ならばテキストファイルです。下位から 2 ビット目は仕様書を読んでも意味が分かりませんでした。このビットを立てたビットを SecureZIP に読み込ませると、「ZDW」と表示されます。SecureZIP の更新情報[*4]には

> ZDW 解凍のサポート - translate オプションは、新しい EBCDIC line-ending translation のサブオプションを備え、可変長レコードを保存するために SecureZIP for z/OS の Zip Descriptor Word (ZDW) オプションを使用して、圧縮したメインフレームデータの解凍をサポートします。

との記載があり、これに対応しているものと思われます。0 にしておけば良いでしょう。

[*4] https://www.xlsoft.com/jp/products/pkware/support/whatsnew_sz_desktop.html

externalAttr はファイルの属性を示します。versionMade の上位バイトが 0（MS-DOS）ならば、FAT32 の属性フラグと同じ値を使用します。ビットと属性の対応は下表のようになります。

ビット	属性
0	読み取り専用
1	システム
2	隠しファイル
3	ボリューム
4	ディレクトリ
5	アーカイブ

拡張フィールドが存在する場合、ヘッダと対応するデータが、header 1, data 1, header 2, data 2, ..., heder n, data n と並びます。拡張フィールドのヘッダは次の通りです。

項目名	サイズ	説明
id	2 bytes	データの種類を示す ID
size	2 bytes	データのサイズ

無圧縮でファイルを ZIP に格納するプログラムは次のようになります。

```
#pragma pack(push, 1)

struct LocalFileHeader
{
    uint32_t signature = 0x04034b50u;
    uint16_t versionNeeded = 0;
    uint16_t flag = 0;
    uint16_t compression = 0;
    uint16_t modifiedTime = 0;
    uint16_t modifiedDate = 0;
    uint32_t crc = 0;
    uint32_t compressedSize = 0;
    uint32_t uncompressedSize = 0;
    uint16_t fileNameLength = 0;
    uint16_t extraFieldLength = 0;
};
```

```
struct CentralDirectoryHeader
{
    uint32_t signature = 0x02014b50u;
    uint16_t versionMade = 0;
    uint16_t versionNeeded = 0;
    uint16_t flag = 0;
    uint16_t compression = 0;
    uint16_t modifiedTime = 0;
    uint16_t modifiedDate = 0;
    uint32_t crc = 0;
    uint32_t compressedSize = 0;
    uint32_t uncompressedSize = 0;
    uint16_t fileNameLength = 0;
    uint16_t extraFieldLength = 0;
    uint16_t commentLength = 0;
    uint16_t disk = 0;
    uint16_t internalAttr = 0;
    uint32_t externalAttr = 0;
    uint32_t offset = 0;
};

#pragma pack(pop)

void copyHeader(const CentralDirectoryHeader &central, LocalFileHeader *local)
{
    local->versionNeeded = central.versionNeeded;
    local->flag = central.flag;
    local->compression = central.compression;
    local->modifiedTime = central.modifiedTime;
    local->modifiedDate = central.modifiedDate;
    local->crc = central.crc;
    local->compressedSize = central.compressedSize;
    local->uncompressedSize = central.uncompressedSize;
    local->fileNameLength = central.fileNameLength;
    local->extraFieldLength = central.extraFieldLength;
}

void timeToDos(time_t time, uint16_t *dosTime, uint16_t *dosDate)
{
    tm *local = localtime(&time);
    *dosTime =
        local->tm_hour<<11 |
```

```
        local->tm_min<<5 |
        local->tm_sec/2;
    *dosDate =
        (local->tm_year + 1900 - 1980)<<9 |
        (local->tm_mon + 1)<<5 |
        local->tm_mday;
}

int main()
{
    makeCrcTable();

    string name[3] = {"file1.txt", "file2.txt", "file3.txt"};
    string content[3] = {"understand zip", "abc", ""};

    FILE *f = fopen("uncompressed.zip", "wb");

    uint16_t modTime, modDate;
    timeToDos(time(nullptr), &modTime, &modDate);

    CentralDirectoryHeader central[3];
    for (int i=0; i<3; i++) {
        central[i].versionMade = 0<<8 | 20;
        central[i].versionNeeded = 20;
        central[i].modifiedTime = modTime;
        central[i].modifiedDate = modDate;
        central[i].crc = crc(stringToUint8(content[i]));
        central[i].compressedSize = (uint32_t)content[i].size();
        central[i].uncompressedSize = (uint32_t)content[i].size();
        central[i].fileNameLength = (uint32_t)name[i].size();
        central[i].offset = (uint32_t)ftell(f);

        LocalFileHeader local;
        copyHeader(central[i], &local);

        fwrite(&local, sizeof local, 1, f);
        fwrite(name[i].c_str(), name[i].size(), 1, f);
        fwrite(content[i].c_str(), content[i].size(), 1, f);
    }

    EndOfCentralDirectoryRecord end;
    end.centralDirOffset = (uint32_t)ftell(f);
```

```
    for (int i=0; i<3; i++) {
        fwrite(&central[i], sizeof central[i], 1, f);
        fwrite(name[i].c_str(), name[i].size(), 1, f);
    }

    end.entryNumber = 3;
    end.totalEntryNumber = 3;
    end.centralDirSize = (uint32_t)ftell(f) - end.centralDirOffset;
    fwrite(&end, sizeof end, 1, f);

    fclose(f);
}
```

```
00000000  50 4b 03 04 14 00 00 00  00 00 04 28 7c 4b 4f 31  |PK.........(|KO1|
00000010  de 3c 0e 00 00 00 0e 00  00 00 09 00 00 00 66 69  |.<............fi|
00000020  6c 65 31 2e 74 78 74 75  6e 64 65 72 73 74 61 6e  |le1.txtunderstan|
00000030  64 20 7a 69 70 50 4b 03  04 14 00 00 00 00 00 04  |d zipPK.........|
00000040  28 7c 4b c2 41 24 35 03  00 00 00 03 00 00 00 09  |(|K.A$5.........|
00000050  00 00 00 66 69 6c 65 32  2e 74 78 74 61 62 63 50  |...file2.txtabcP|
00000060  4b 03 04 14 00 00 00 00  00 04 28 7c 4b 00 00 00  |K.........(|K...|
00000070  00 00 00 00 00 00 00 00  00 09 00 00 00 66 69 6c  |.............fil|
00000080  65 33 2e 74 78 74 50 4b  01 02 14 00 14 00 00 00  |e3.txtPK........|
00000090  00 00 04 28 7c 4b 4f 31  de 3c 0e 00 00 00 0e 00  |...(|KO1.<......|
000000a0  00 00 09 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000b0  00 00 00 00 66 69 6c 65  31 2e 74 78 74 50 4b 01  |....file1.txtPK.|
000000c0  02 14 00 14 00 00 00 00  00 04 28 7c 4b c2 41 24  |..........(|K.A$|
000000d0  35 03 00 00 00 03 00 00  00 09 00 00 00 00 00 00  |5...............|
000000e0  00 00 00 00 00 00 00 35  00 00 00 66 69 6c 65 32  |.......5...file2|
000000f0  2e 74 78 74 50 4b 01 02  14 00 14 00 00 00 00 00  |.txtPK..........|
00000100  04 28 7c 4b 00 00 00 00  00 00 00 00 00 00 00 00  |.(|K............|
00000110  09 00 00 00 00 00 00 00  00 00 00 00 00 00 5f 00  |..............._.|
00000120  00 00 66 69 6c 65 33 2e  74 78 74 50 4b 05 06 00  |..file3.txtPK...|
00000130  00 00 00 03 00 03 00 a5  00 00 00 86 00 00 00 00  |................|
00000140  00                                                |.|
00000141
```

Deflate 圧縮

ZIP はいくつかの圧縮方法をサポートしていますが、主に Deflate 圧縮が用いられます。ZIP の仕様書には簡単にしか書かれていないので、詳細は Deflate の仕様書（RFC 1951）[*5]を読むと良いでしょう。

Deflate 圧縮では、(a) 同じ文字列が何度も出現することが多いことと、(b) 文字の出現頻度には偏りがあることに着目して圧縮します。

繰り返し出現する文字列の圧縮

`zip`（`122 105 112`）が 8 回繰り返されたデータを圧縮することを考えます。最初の 3 文字は過去に出現していない文字列なので、そのまま `122 105 112` を出力します。入力のまま出力される文字を Deflate では「リテラル」と呼びます。4 文字目以降の 21 文字は、「この先 21 文字は、3 文字前と同じである」と出力します。`122 105 112 (21, 3)`。この先何文字が一致しているか（この例では 21 文字）を「一致長」、何文字前と等しいか（この例では 3 文字前）を「一致距離」と言います。一致長と一致距離で出力する文字列と参照する文字列は重なることができます。この例でも、21 を出力した時点では参照する文字列は 3 文字しかありません。一致長と一致文字列で表された文字列を解凍する際には、（特別な処理をしない限り）先頭から 1 文字ずつ処理をする必要があります。

Deflate ではリテラルと一致長、さらにブロック（後述）が終了するかどうかのフラグををひとまとめにして、リテラル・一致長として取り扱います。リテラル・一致長が、256 未満ならばリテラル、256 ならばブロックの終了、257 以上ならば一致長となります。これによって、リテラルか一致長かを表す 1 ビットを削っています。

一致長は 3 から 258、一致距離は 1 から 32768 までの範囲を取ります。そのまま次項で説明するハフマン符号で取り扱うと、符号の数が多くなりすぎて効率が悪いので、拡張ビットを用いていくつかの値をまとめます。例えば、19 から 22 の一致長は 269 という同一のリテラル・一致長になり、4 通りの一致長を区別するため 2 ビットの拡張符号を直後に出力します。

リテラル・一致長（コード）と拡張ビットの長さ（拡張）や一致長の対応は下表の通りです。

[*5] https://www.ietf.org/rfc/rfc1951.txt

コード	拡張	一致長	コード	拡張	一致長	コード	拡張	一致長
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

一致距離についても同様で、対応は次のようになります。

コード	拡張	一致距離	コード	拡張	一致距離	コード	拡張	一致距離
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

リテラルと一致長を 1 個の数字で表すことと、拡張ビットを考慮して、冒頭の `zip` を 8 回繰り返した文字列の圧縮結果を表すと、`122 105 112 269 2 2 0` となります。最後から 3 番目の `2` と最後の `0` が拡張ビットです。

ハフマン符号

文字の出現数に偏りがあるとき、よく出現する文字には短いビット列を、あまり出現しない文字には長いビット列を割り当てることで、データ全体のサイズを小さくすることができます。ハフマン符号を用いると最適なビット列の割り当て方が求められます[*6]。

ハフマン符号は出現回数が少ない文字をまとめていくことでビット列を計算します。各文字の出現回数が次のようになっている例を考えます。

文字	出現回数
0	1
1	17
2	2
3	3
4	5
5	5

出現回数が最も少ない 2 個の文字 0 と 2 に、ビット列 0 と 1 を割り当てます。これをまとめて出現回数 3 回の文字 a として扱います。すると a と 3 の出現回数がそれぞれ 3 回ずつで最も出現回数が少ない文字になるので、ビット列 0 と 1 を割り当て、出現回数が 6 回の文字 b とします。このとき、0 に割り当てられた文字列は 00、2 は 01、3 は 1 となります。4 と 5 をまとめて c とし、b と c をまとめて d として、最後に d と 1 をまとめます。文字と割り当てられたビット列は次のようになります。

文字	出現回数	ビット列
0	1	0000
1	17	1
2	2	0001
3	3	001
4	5	010
5	5	011

[*6] https://ja.wikipedia.org/wiki/ハフマン符号

プログラム例です。vector<Node>を何度もコピーしているので、ポインタにするなどして最適化をする余地はあるでしょう。

```
#include <iostream>
#include <vector>
#include <queue>
#include <string>
#include <functional>
using namespace std;

struct Node
{
    int symbol = 0;
    int count = 0;
    vector<Node> child;
    Node(int symbol, int count, vector<Node> child)
        :symbol(symbol), count(count), child(child) {}
    bool operator<(const Node &n) const
        {return count>n.count || count==n.count && symbol>n.symbol;}
};

vector<string> huffman(vector<int> count)
{
    priority_queue<Node> node;
    for (size_t i=0; i<count.size(); i++)
        node.push(Node((int)i, count[i], {}));
    while (node.size()>1) {
        Node n1 = node.top(); node.pop();
        Node n2 = node.top(); node.pop();
        node.push(Node(-1, n1.count+n2.count, {n1, n2}));
    }
    vector<string> code(count.size());
    function<void(const Node &, string)> traverse = [&](const Node &n, string c)
    {
        if (n.symbol != -1)
            code[n.symbol] = c;
        else {
            traverse(n.child[0], c+"0");
            traverse(n.child[1], c+"1");
        }
    };
    traverse(node.top(), "");
```

```
    return code;
}

int main()
{
    vector<string> code = huffman({1, 17, 2, 3, 5, 5});
    for (size_t i=0; i<code.size(); i++)
        cout<<i<<" "<<code[i]<<endl;
}
```

```
0 0000
1 1
2 0001
3 001
4 010
5 011
```

なお、このプログラム中の関数は Deflate 圧縮に使用することはできません。Deflate 圧縮ではビット列の長さに制限があるからです。例えば、`huffman` 関数に`{1, 2, 4, 8, 16, 32}`を渡すと 0 に 00000 という長いビット列が割り当てられることが確認できます。

ビット列の長さに上限 L があるときに最適なビット列の割り当てを求めるアルゴリズムとして、Package-merge が知られています[*7]。

n 個の文字の出現回数がそれぞれ $c_0, c_1, \ldots, c_{n-1}$ のとき、n 種類の出現回数に対応した「価値」と、$2^{-1}, 2^{-2}, \ldots, 2^{-L}$ の L 種類の「額面」を組み合わせた nL 枚のコインを用意します。なるべく「価値」が小さくなるように、額面の合計が $n-1$ になるようなコインの組み合わせを選びます。このとき、この組み合わせの中に i 番目の文字に対応するコインが h_i 枚含まれているならば、i 番目の文字に割り当てるべきビット列の長さは h_i となります。各文字に対応するコインは L 枚しかないため、ビット列の最大の長さは L です。

最適なコインを組み合わせは、額面 2^{-k} のコインを小さい順に 2 枚ずつセットにして 2^{-k+1} のコインとして扱うということを繰り返すことで求められます。

```
vector<int> packMerge(vector<int> count, int L)
```

[*7] https://en.wikipedia.org/wiki/Package-merge_algorithm

```
{
    struct Coin
    {
        int value;
        vector<int> symbol;
        Coin(int value, vector<int> symbol): value(value), symbol(symbol) {}
        bool operator<(const Coin &c) const {return value<c.value;}
    };
    vector<vector<Coin>> denom(L+1);
    size_t n = 0;
    for (size_t i=0; i<count.size(); i++)
        if (count[i]>0) {
            for (int d=0; d<L; d++)
                denom[d].push_back(Coin(count[i], vector<int>(1, (int)n)));
            n++;
        }
    vector<int> lengthTmp(n);
    switch (n) {
    case 0:
        break;
    case 1:
        lengthTmp[0] = 1;
        break;
    default:
        for (int d=0; d<L; d++) {
            sort(denom[d].begin(), denom[d].end());
            for (size_t i=0; i+1<denom[d].size(); i+=2) {
                Coin c = denom[d][i];
                c.value += denom[d][i+1].value;
                c.symbol.insert(c.symbol.end(), denom[d][i+1].symbol.begin(),
                    denom[d][i+1].symbol.end());
                denom[d+1].push_back(c);
            }
        }
        sort(denom[L].begin(), denom[L].end());
        for (size_t i=0; i<n-1; i++)
            for (int s: denom[L][i].symbol)
                lengthTmp[s]++;
    }
    vector<int> length;
    size_t c = 0;
    for (size_t i=0; i<count.size(); i++)
```

```
        length.push_back(count[i]>0 ? lengthTmp[c++] : 0);
    return length;
}

int main()
{
    vector<int> length = packMerge({1, 17, 2, 3, 5, 5}, 4);
    for (size_t i=0; i<length.size(); i++)
        printf("%d %d\n", (int)i, length[i]);
}
```

Deflate の仕様上、1 個しか値が存在しない場合でも 1 ビットは使用する必要があります。出力は、

```
0 4
1 1
2 4
3 3
4 3
5 3
```

となり、充分な長さ L を与えれば、長さに制限の無いハフマン符号のビット列長が得られることが確認できます。`packMerge({1, 17, 2, 3, 5, 5}, 3)` とすると出力が変化し、次のようになります。

```
0 3
1 2
2 3
3 3
4 3
5 2
```

出現しない（出現回数が 0 の）文字にビット列を割り当てる必要は無いので、除いて計算しています。

Package-merge で得られるのは各文字に割り当てるべきビット列の長さだけです（2 個のコインを組み合わせるときに、どのコインを組み合わせたかを覚えておけばビット列も得られます）が、これで充分です。例えば、0 に 00 を 1 に 010 を割り当てたときと、0 に 10 を 1 に 001 を割り当

てたときで、このビット列を使ってデータを圧縮するときのサイズは等しくなります。Deflate では次の規則でビット列の長さから一意にビット列が定まるようにしています。

1. 長さが異なるビット列について、短いビット列は長いビット列よりも辞書順で小さい（..., 0100, 0101, 01100, 01101, ...）
2. 同じ長さのビット列ならば、小さい文字は辞書順で小さいビット列が割り当てられる

この規則に従って、最初の例のビット列を割り当て直すと次のようになります。当然、各文字に割り当てられたビット列の長さは変わりません。

文字	出現回数	ビット列
1	17	0
3	3	100
4	5	101
5	5	110
0	1	1110
2	2	1111

ハフマン符号化の実装は次のようになります。

```
class BitStream
{
public:
    vector<uint8_t> data;
    size_t c;
    BitStream(): c(0) {}
    void write(int bit) {
        if (c%8 == 0)
            data.push_back(0);
        data[data.size()-1] |= bit<<c%8;
        c++;
    }
    void writeBits(int bits, int len) {
        for (int i=0; i<len; i++)
            write(bits>>i&1);
    }
};

class Huffman
```

```
{
    vector<int> code;
    vector<int> length;
public:
    Huffman(vector<int> length): length(length) {
        code = vector<int>(length.size());
        size_t n = 0;
        for (int l: length)
            if (l==0)
                n++;
        int c = 0;
        for (int l=1; n<code.size(); l++, c<<=1)
            for (size_t i=0; i<length.size(); i++)
                if (length[i]==l) {
                    code[i] = c++;
                    n++;
                }
    }
    void write(int symbol, BitStream *stream) {
        for (int i=length[symbol]-1; i>=0; i--)
            stream->write(code[symbol]>>i&1);
    }
};

int main()
{
    Huffman huffman({4, 1, 4, 3, 3, 3});
    BitStream stream;
    for (int i=0; i<6; i++)
        huffman.write(i, &stream);
    for (uint8_t c: stream.data)
        printf(" %02x", c);
    printf("\n");
    //  e7 d3 01
}
```

1 ビットずつ値を追加し vector<uint8_t>に格納するクラス BitStream も作成しました。このクラスには後の処理で使用する複数のビット列をまとめて書き込むメソッド writeBits も実装しています。

出力の e7 d3 01 をビット列に直すと 11100111 11001011 1000000 となり、たしかに上記のビット列が書き込まれていることが分かります。Deflate では各バイトのビットは下位から書き

込んでいきます。

ハフマン符号表の圧縮

文字と対応するビット列をまとめた表をハフマン符号表と言います。規則に従ってハフマン符号表を構築することで、各文字のビット長を送れば、圧縮側と解凍側で同じハフマン符号表を共有することができます。素直に各文字のビット長を送るとサイズが大きくなってしまうので、このビット長自体も圧縮します。

まず、下表のコードを使って同じビット長を圧縮します。

コード	意味
0-15	そのままビット長を表す
16	2 ビットの拡張コード x を続け、直前のビット長を x+3 回繰り返すことを表す
17	3 ビットの拡張コード x を読け、0 を x+3 回繰り返すことを表す
18	7 ビットの拡張コード x を読け、0 を x+11 回繰り返すことを表す

例えば、０８８８８８０００３００を圧縮すると、０８ 16 １ 17 ０３となります。ビット長の個数は別に与えられるので、末尾の 0 を明示的に送る必要はありません。

この仕様からビット長の最大値は 15 となります。また、Deflate ではリテラル・一致長と一致距離に異なるハフマン符号を用いますが、ここではリテラル・一致長と一致距離を繋げて圧縮します。例えば、リテラル・一致長の末尾に８があり、一致距離の先頭に８が続く場合、コード 16 を使って参照することができます。

さらに、このコードをハフマン符号化して、コードのビット長を送ります。コードのビット長は 3 ビットで固定なので、コードのビット長は最大 7 です。コードのビット長は次の順番で送られます。

16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

コードのビット長の個数も別に送られるので、例えば 14, 1, 15 を使用しない場合にはコードのビット長の個数は 16 個として、末尾 3 個は送らずに 9 ビットを節約することができます。

ブロック

Deflate ではデータはいくつかのブロックに分かれて格納されます。ブロックごとにハフマン符号表はリセットされますが、一致長と一致距離での参照はブロックを跨ぐことができます。例えば、1 個目のブロックが１ ２ ３で、2 個目のブロックの最初が (4, 2) のとき、2 個目のブロックは２ ３ ２ ３というデータを表しています。

各ブロックの先頭には、最後のブロックかどうかを示す 1 ビットのフラグ BFINAL と、ブロックの圧縮方法を示す 2 ビットの BTYPE があります。

BFINAL が 0 ならばこのブロック以降もブロックが続き、1 ならば終了です。

BTYPE が 00 のとき、ブロックは圧縮されていません。この場合、次のバイト境界までのビットは無視されます。その後、バイト単位でのデータサイズを示す 2 バイトの LEN と、LEN のビットを反転させた NLEN が続き、以降に LEN バイトのデータがそのまま格納されます。

BTYPE が 01 のとき、ブロックは各文字のビット長が仕様で規定された固定ハフマン符号表で圧縮されています。各文字のビット長は次の表の通りです。ハフマン符号表を送る必要が無いので、BTYPE の直後にハフマン符号で圧縮されたデータ本体が続きます。

リテラル・一致長コード	ビット長
0-143	8
144-255	9
256-279	7
280-287	8

一致距離コード	ビット長
0-31	5

リテラル・一致長コードの 286 と 287 と、一致距離コードの 30 と 31 は、ハフマン符号表の作成には使用しますが、圧縮の際に使用することはありません。

BTYPE が 11 のとき、データはカスタムハフマン符号表（圧縮する側が決めたハフマン符号表）で圧縮されます。ハフマン符号表は次のようにして送られます。

項目	ビット数	個数	説明
HLIT	5	1	リテラル・一致長のコードの個数 - 257
HDIST	5	1	一致距離のコードの個数 - 1
HCLEN	4	1	コード長コードの個数 - 4
codeLen	3	HCLEN+4	コード長コードのビット長（紛らわしい .....）
litLen	-	HLIT+257	
distLen	-	HDIST+1	

Deflate の実装

Deflate 圧縮を行うプログラムです。

```
struct Code
{
    int code = 0;
    int ext = 0;
    int extLen = 0;
    Code() {}
    Code(int code, int ext, int extLen): code(code), ext(ext), extLen(extLen) {}
};

//  value からコードと拡張ビットを求める
Code calcCodeExt(int value, const int *table, int tableLen, int offset)
{
    for (int i=0; i<tableLen; i++) {
        if (value <= 1<<table[i])
            return Code(i+offset, value, table[i]);
        value -= 1<<table[i];
    }
    //  erorr
    return Code();
}

vector<uint8_t> deflate(vector<uint8_t> data)
{
    //  リテラル・一致長と一致距離の拡張ビット長
    static int literalExtTable[29] = {
        0, 0, 0, 0, 0, 0, 0, 0,  1, 1, 1, 1, 2, 2, 2, 2,
```

```
    3, 3, 3, 3, 4, 4, 4, 4,  5, 5, 5, 5, 0,
};
static int distExtTable[30] = {
    0, 0, 0, 0, 1, 1, 2, 2,  3, 3, 4, 4, 5, 5, 6, 6,
    7, 7, 8, 8, 9, 9,10,10, 11,11,12,12,13,13,
};

// 入力をリテラル・一致長と一致距離に圧縮する
vector<Code> literal;
vector<Code> dist;
// 添え字の3文字が過去に出現した位置を記録する
map<uint32_t, size_t> hash;
for (size_t i=0; i<data.size();)
{
    int len = 0;
    if (i+3 <= data.size()) {
        uint32_t h = data[i]<<16 | data[i+1]<<8 | data[i+2];
        if (hash.count(h) > 0 && i-hash[h] <= 32768) {
            // 過去に出現していれば、一致長と一致距離に圧縮
            size_t p = hash[h];
            len = 3;
            while (len<258 && i+len<data.size() && data[p+len]==data[i+len])
                len++;
            literal.push_back(calcCodeExt(len-3, literalExtTable, 29, 257));
            dist.push_back(
                calcCodeExt((int)(i-hash[h]), distExtTable, 30, 0));
        }
    }
    if (len == 0) {
        // 過去に出現していなければ、そのままリテラルとして書き出す
        literal.push_back(Code(data[i], 0, 0));
        dist.push_back(Code(-1, 0, 0));
        len = 1;
    }
    for (int j=0; j<len; j++)
        if (2 <= i+j)
            hash[data[i+j-2]<<16 | data[i+j-1]<<8 | data[i+j]] = i+j-2;
    i += len;
}
literal.push_back(Code(256, 0, 0));
dist.push_back(Code(-1, 0, 0));
```

```
//  リテラル・一致長と一致距離の最適なビット長を求める
vector<int> literalCount(286);
for (Code l: literal)
    literalCount[l.code]++;
vector<int> literalLen = packMerge(literalCount, 15);
vector<int> distCount(30);
for (Code d: dist)
    if (d.code>=0)
        distCount[d.code]++;
vector<int> distLen = packMerge(distCount, 15);

//  ビット長の配列を作り、圧縮する
int HLIT = 286 - 257;
while (0<HLIT && literalLen[HLIT+257-1]==0)
    HLIT--;
int HDIST = 30 - 1;
while (0<HDIST && distLen[HDIST+1-1]==0)
    HDIST--;

vector<int> codeLen;
for (int i=0; i<HLIT+257; i++)
    codeLen.push_back(literalLen[i]);
for (int i=0; i<HDIST+1; i++)
    codeLen.push_back(distLen[i]);
vector<Code> codeLenCode;
for (size_t i=0; i<codeLen.size();) {
    int prevLen = 0;
    if (i>0)
        while (prevLen<6 && i+prevLen<codeLen.size() &&
            codeLen[i+prevLen]==codeLen[i-1])
            prevLen++;
    int zeroLen = 0;
    while (zeroLen<138 && i+zeroLen<codeLen.size() &&
        codeLen[i+zeroLen]==0)
        zeroLen++;
    if (zeroLen>=11) {
        codeLenCode.push_back(Code(18, zeroLen-11, 7));
        i += zeroLen;
    } else if (zeroLen>=3) {
        codeLenCode.push_back(Code(17, zeroLen-3, 3));
        i += zeroLen;
    } else if (prevLen>=3) {
```

```
            codeLenCode.push_back(Code(16, prevLen-3, 3));
            i += prevLen;
        } else {
            codeLenCode.push_back(Code(codeLen[i], 0, 0));
            i++;
        }
    }
    vector<int> codeLenCount(19);
    for (Code c: codeLenCode)
        codeLenCount[c.code]++;
    vector<int> codeLenCodeLen = packMerge(codeLenCount, 7);

    static int codeLenTrans[] = {
        16,17,18, 0, 8, 7, 9, 6, 10, 5,11, 4,12, 3,13, 2,
        14, 1,15
    };
    int HCLEN = 19 - 4;
    while (0<HCLEN && codeLenCodeLen[codeLenTrans[HCLEN+4-1]]==0)
        HCLEN--;

    //  圧縮結果を書き出す
    BitStream stream;
    stream.write(1);        //  BFINAL = 1
    stream.writeBits(2, 2); //  BTYPE = 10

    stream.writeBits(HLIT, 5);
    stream.writeBits(HDIST, 5);
    stream.writeBits(HCLEN, 4);

    for (int i=0; i<HCLEN+4; i++)
        stream.writeBits(codeLenCodeLen[codeLenTrans[i]], 3);

    Huffman huffmanCodeLen(codeLenCodeLen);
    for (size_t i=0; i<codeLenCode.size(); i++) {
        huffmanCodeLen.write(codeLenCode[i].code, &stream);
        stream.writeBits(codeLenCode[i].ext, codeLenCode[i].extLen);
    }

    Huffman huffmanLiteral(literalLen);
    Huffman huffmanDist(distLen);
    for (size_t i=0; i<literal.size(); i++) {
        huffmanLiteral.write(literal[i].code, &stream);
```

```
        stream.writeBits(literal[i].ext, literal[i].extLen);
        if (dist[i].code>=0) {
            huffmanDist.write(dist[i].code, &stream);
            stream.writeBits(dist[i].ext, dist[i].extLen);
        }
    }

    return stream.data;
}

int main()
{
    vector<uint8_t> data;
    for (int i=0; i<8; i++) {
        data.push_back(122);
        data.push_back(105);
        data.push_back(112);
    }
    vector<uint8_t> comp = deflate(data);
    for (uint8_t c: comp)
        printf(" %02x", c);
    printf("\n");
    //   6d c2 31 0d 00 00 00 83 30 bd fb 76 a3 1e 03 24 65 4f 02
}
```

このプログラムでは、入力データの連続する 3 バイトの位置を覚えておくことで、過去に出現したパターンの検索を高速化しています。ここでは、各 3 バイトごとに 1 個の場所だけを記録しています。Deflate の仕様書ではいくつかの場所を覚えておき、最も一致長が長いものを選択するという方法が紹介されています。このアルゴリズムであれば特許に抵触する心配が無いとのことです。

Deflate 関数を用いることで圧縮されたファイルを格納した ZIP を作ることができます。

```
int main()
{
    makeCrcTable();

    FILE *f = fopen("compressed.zip", "wb");

    string name = "zip.txt";
```

```
    string content_ = "";
    for (int i=0; i<8; i++)
        content_ += "zip";
    vector<uint8_t> content = stringToUint8(content_);
    vector<uint8_t> compressed = deflate(content);

    CentralDirectoryHeader central;
    central.versionMade = 0<<8 | 20;
    central.versionNeeded = 20;
    central.compression = 8;    // Deflate
    central.crc = crc(content);
    central.compressedSize = (uint32_t)compressed.size();
    central.uncompressedSize = (uint32_t)content.size();
    central.fileNameLength = (uint32_t)name.size();
    central.offset = 0;

    LocalFileHeader local;
    copyHeader(central, &local);

    fwrite(&local, sizeof local, 1, f);
    fwrite(name.c_str(), name.size(), 1, f);
    fwrite(&compressed[0], compressed.size(), 1, f);

    EndOfCentralDirectoryRecord end;
    end.centralDirOffset = (uint32_t)ftell(f);

    fwrite(&central, sizeof central, 1, f);
    fwrite(name.c_str(), name.size(), 1, f);

    end.entryNumber = 1;
    end.totalEntryNumber = 1;
    end.centralDirSize = (uint32_t)ftell(f) - end.centralDirOffset;
    fwrite(&end, sizeof end, 1, f);

    fclose(f);
}
```

```
00000000  50 4b 03 04 14 00 00 00  08 00 00 00 00 00 24 e3  |PK..........$.|
00000010  38 6c 13 00 00 00 18 00  00 00 07 00 00 00 7a 69  |8l..........zi|
00000020  70 2e 74 78 74 6d c2 31  0d 00 00 00 83 30 bd fb  |p.txtm.1.....0..|
00000030  76 a3 1e 03 24 65 4f 02  50 4b 01 02 14 00 14 00  |v...$eO.PK......|
```

```
00000040  00 00 08 00 00 00 00 00  24 e3 38 6c 13 00 00 00  |........$.8l....|
00000050  18 00 00 00 07 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 00 7a 69  70 2e 74 78 74 50 4b 05  |......zip.txtPK.|
00000070  06 00 00 00 00 01 00 01  00 35 00 00 00 38 00 00  |.........5...8..|
00000080  00 00 00                                          |...|
00000083
```

暗号化

ZIP の暗号化で最も互換性があるのは、仕様書で「Traditional PKWARE Encryption」と呼ばれているアルゴリズムです。以降の説明では単に「ZIP の暗号」と呼んでいます。3 章で解説するように、このアルゴリズムは脆弱です。仕様書にも「この形の暗号化は今日の標準では脆弱だと考えられており、低いセキュリティしか要求されない場合か、古い ZIP アプリケーションとの互換性のためにのみ用いられるべきである」と明記されています。

ZIP の暗号は、3 個の 32 ビット整数 key0 と key1、key2（鍵）を内部状態として持っています。パスワードや平文の入力してこれらの整数を攪拌し（update_keys）、key2 から 1 バイトの値を取り出して（decrypt_byte）、平文と xor を取ることで暗号化を行います。仕様書では key0 と key1、key2 の初期値は、それぞれ 305419896 と 591751049、878082192 と 10 進数で書かれていますが、16 進数のほうが分かりやすいでしょう。

```
struct Key
{
    uint32_t key0 = 0x12345678U;
    uint32_t key1 = 0x23456789U;
    uint32_t key2 = 0x34567890U;
    void update_keys(uint8_t c) {
        key0 = crc32(key0, c);
        key1 = (key1 + (key0&0xff)) * 0x08088405U + 1;
        key2 = crc32(key2, key1>>24);
    }
    uint8_t decrypt_byte() {
        uint16_t temp = (uint16_t)(key2 | 2);
        return (temp*(temp^1))>>8;
    }
};
```

```
int main()
{
    makeCrcTable();
    Key key;
    for (uint8_t c=0; c<8; c++) {
        key.update_keys(c);
        printf(" %02x", key.decrypt_byte());
    }
    printf("\n");
    //  1a 63 54 9f e7 41 0e 83
}
```

1、2、...、7 を入力して、予測ができない（ように見える）値が出力されていることがわかります。

ZIP ではファイルごとに暗号化が行われます。あまりそのような使い方はされませんが、ファイルごとに別のパスワードで暗号化したり、ZIP 中の特定のファイルだけを暗号化したりすることができます。

ZIP 中の暗号化されたファイルは、local file header とファイルの中身の間に、12 バイトの encryption header が置かれます。暗号化される前の encryption header の最後の 1 バイトもしくは 2 バイトは、ファイルの CRC 値の最上位バイトもしくは上位 2 バイトです。解凍プログラムは、encryption header を復号して最後のバイトが CRC 値に一致するか確認することで、パスワードが正しいかどうかを判定します。2 バイトのチェックは古い PKZIP で行われていたものであり、特に互換性を重視する場合以外は、攻撃者に余計な情報を与えないために 1 バイトのみを CRC にするべきでしょう。encryption header の前半 11 バイト（もしくは 10 バイト）はランダムなデータで埋めます。

仕様書には記述がありませんが、Info-zip の Zip 3.00 のソースコードを見ると、local file header と central directory header の `flag` の 3 ビット目が立っているときは CRC の代わりに `modifiedTime` を使うという実装になっています。PKWARE 社の SecureZIP もこの方法でパスワードのチェックをしていました。`flag` の 3 ビット目を立てると、ファイルサイズや CRC を local file header ではなくファイル本体の直後に出力して、標準入出力などのシークができないファイルで ZIP を取り扱うことができます。圧縮プログラムが ZIP ファイルを書き込むときも、解凍プログラムが ZIP を読み込むときもシークができない場合、encryption header を処理する時点でファイルの CRC が分からないのでこのような仕様になっているのでしょう。

```
    /* If last two bytes of header don't match crc (or file time in the
```

```
     * case of an extended local header), back up and just copy. For
     * pkzip 2.0, the check has been reduced to one byte only.
     */
#ifdef ZIP10
    if ((ush)(c0 | (c1<<8)) !=
        (z->flg & 8 ? (ush) z->tim & 0xffff : (ush)(z->crc >> 16))) {
#else
    if ((ush)c1 != (z->flg & 8 ? (ush) z->tim >> 8 : (ush)(z->crc >> 24))) {
#endif
```

Encryption header の役割はファイル本体の暗号化を行う前に、鍵を異なる値にすることです。Encryption header が存在しない場合、例えば、各ファイルの最初のバイトを xor するために使う値が同じになってしまい、暗号化前後で 2 個のファイルの最初のバイトの xor を取った値が変化しません。

まず、パスワードの各バイトを `update_keys` に入力して鍵を初期し、encryption header とファイル本体を続けて暗号化します。

Local file header と central file header の `flag` の 0 ビット目を立てることでファイルが暗号化されていることを示します。`compressedSize` には encryption header の 12 バイトを加えます。

```
int main()
{
    makeCrcTable();

    FILE *f = fopen("encrypted.zip", "wb");

    string name = "secret.txt";
    vector<uint8_t> content = stringToUint8("secret message");
    vector<uint8_t> password = stringToUint8("p@ssw0rd");

    vector<uint8_t> encryption(12);
    random_device rand;
    for (size_t i=0; i<11; i++)
        encryption[i] = (uint8_t)rand();
    uint32_t crc32 = crc(content);
    encryption[11] = (uint8_t)(crc32>>24);

    // 暗号化
    Key key;
```

```
for (uint8_t c: password)
    key.update_keys(c);
for (uint8_t &c: encryption) {
    uint8_t t = key.decrypt_byte();
    key.update_keys(c);
    c ^= t;
}
for (uint8_t &c: content) {
    uint8_t t = key.decrypt_byte();
    key.update_keys(c);
    c ^= t;
}

CentralDirectoryHeader central;
central.versionMade = 0<<8 | 20;
central.versionNeeded = 20;
central.flag = 1;   //  encrypted
central.crc = crc32;
central.compressedSize = (uint32_t)content.size() + 12;
central.uncompressedSize = (uint32_t)content.size();
central.fileNameLength = (uint32_t)name.size();
central.offset = 0;

LocalFileHeader local;
copyHeader(central, &local);

fwrite(&local, sizeof local, 1, f);
fwrite(name.c_str(), name.size(), 1, f);
fwrite(&encryption[0], 12, 1, f);
fwrite(&content[0], content.size(), 1, f);

EndOfCentralDirectoryRecord end;
end.centralDirOffset = (uint32_t)ftell(f);

fwrite(&central, sizeof central, 1, f);
fwrite(name.c_str(), name.size(), 1, f);

end.entryNumber = 1;
end.totalEntryNumber = 1;
end.centralDirSize = (uint32_t)ftell(f) - end.centralDirOffset;
fwrite(&end, sizeof end, 1, f);
```

```
    fclose(f);
}
```

```
00000000  50 4b 03 04 14 00 01 00  00 00 00 00 00 00 64 03  |PK............d.|
00000010  cd e5 1a 00 00 00 0e 00  00 00 0a 00 00 00 73 65  |..............se|
00000020  63 72 65 74 2e 74 78 74  23 04 b2 1f 88 1e e9 f4  |cret.txt#.......|
00000030  5e c8 f0 e1 67 26 4a 94  26 fb 78 e5 44 1e 64 50  |^...g&J.&.x.D.dP|
00000040  0f 0c 50 4b 01 02 14 00  14 00 01 00 00 00 00 00  |..PK............|
00000050  00 00 64 03 cd e5 1a 00  00 00 0e 00 00 00 0a 00  |..d.............|
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000070  73 65 63 72 65 74 2e 74  78 74 50 4b 05 06 00 00  |secret.txtPK....|
00000080  00 00 01 00 01 00 38 00  00 00 42 00 00 00 00 00  |......8...B.....|
00000090
```

ここでは、encryption header の各バイトを `random_device` を用いて設定しています。3 章で説明するように、ここで用いる乱数が予測可能だと攻撃が可能になってしまうので、暗号論的乱数を使用するべきです。`random_device` は多くの環境では予測不可能な乱数を生成しますが、MinGW では予測可能な乱数となってしまう[*8]ので注意が必要です。

[*8] https://cpprefjp.github.io/reference/random/random_device.html

第 3 章

ZIP の暗号に対する攻撃

ZIP の暗号（Traditional PKWARE Encryption）は既知平文攻撃に対して脆弱であることが知られています。ある暗号文とその暗号文に対する平文を攻撃者が入手した場合、攻撃者が他の暗号文を解読することが可能になります。同じ ZIP 内のファイルは同じパスワードで暗号化されることが多いため、例えばソースコードをまとめた ZIP ファイル中に OSS として公開されているソースコードが含まれていると、解読が可能になります。また、多くのファイルはシグネチャなどで先頭部分が予測可能なため、この部分に対して既知平文攻撃を行い、残りの部分を解読することができる場合もあります。

攻撃を試す前に、Linux の zip コマンドで攻撃の対象となる暗号化された ZIP ファイルを作成します。`-e` で暗号化を、`-0` で圧縮しないことを指示しています。

```
$ zip -e -0 secret.zip kusano_k.png secret.txt
Enter password:
Verify password:
  adding: kusano_k.png (stored 0%)
  adding: secret.txt (stored 0%)
```

これらのファイルは奥付に記載のウェブサイトで公開しています。ここでは、kusano_k.png が既知のファイル、secret.txt は攻撃者が内容を知ることができないファイルであると想定しています。

PkCrack

PkCrack[*1]は次節で解説する攻撃法を実装したプログラムです。なお、このソフトウェアはソースコードが公開されていますが、フリーソフトウェアではありません。カードウェアであり、制作者に絵ハガキを送る必要があります。国際郵便は意外に安く航空便でも 70 円なので、送りましょう。

サイト上に Windows 用バイナリが置かれていますが、16 bit のプログラムであり、現在の 64 bit Windows では実行できません。自分でコンパイルする必要があります。

使用方法は、引数`-c` に暗号化されたファイルを、`-p` に暗号化されていないファイルを指定するだけです。ファイルが ZIP 中に含まれている場合は、暗号文と平文それぞれに`-C` と`-P` で ZIP ファイル名を指定します。

```
> pkcrack.exe -c kusano_k.png -C secret.zip -p kusano_k.png
```

ファイルサイズが大きいほど解読が速くなります。 1 MB ほどのファイルならば数分で解読できます。

パスワードが長い場合はファイルの復号に必要な情報が得られた後のパスワードの探索（`Stage 2 completed. Starting password search` という出力の後）に時間がかかります。そのときは起動引数`-d` でファイル名を指定すると、そのファイル名で暗号化されていない ZIP が出力されます。

Biham と Kocher の攻撃法

本節は Biham と Kocher による論文[*2]の解説です。

この攻撃法の流れは次の通りです。

1. 暗号文と平文から各位置の `decrypt_byte()` の値の配列を得る

[*1] https://www.unix-ag.uni-kl.de/~conrad/krypto/pkcrack.html

[*2] BIHAM, Eli; KOCHER, Paul C. A known plaintext attack on the PKZIP stream cipher. In: International Workshop on Fast Software Encryption. Springer, Berlin, Heidelberg, 1994. p. 144-153.

2. key2 の配列の候補を求める（数百〜2^{22} 個程度）
3. 各 key2 の配列から、key1 の配列の候補を求める（key2 の配列 1 個あたり約 2^{16} 個）
4. key1 の配列の候補から key0 の候補を求める（key1 の配列 1 個に対して 1 個）
5. 暗号文と平文に矛盾しない key0 をフィルタリングする
6. ある位置での key0 と key1、key2 が手に入ったので、暗号化の処理を逆に辿り、パスワードを処理した直後の各 key の値を求める
7. パスワードを計算する

6 の時点での各 key の値があれば、同じパスワードで暗号化されたファイルを復号することができるので、7 の処理は必須ではありません。候補の数は key1 の配列の候補が 2^{22} 個の場合で、約 2^{38} 個となります。

key2

key2 から平文に xor する値を求める処理 decrypt_byte を見ると、（uint16_t にキャストしているので）上位 16 ビットと（2 と or を取っているので）2 番目のビットは返り値に影響が無いことが分かります。また、temp*(temp^1) という処理から最下位ビットも返り値に影響しません。影響があるのは key2 の 14 ビットであり、事前にテーブルを作っておくことで、平文と暗号文からこの 14 ビットを求めることができます。なお、decrypt_byte の各返り値には、この 14 ビットがそれぞれちょうど $2^6 = 64$ 個ずつ対応します。

```
uint8_t decrypt_byte(uint32_t key2) {
    uint16_t temp = (uint16_t)(key2 | 2);
    return (temp*(temp^1))>>8;
}
```

key2 の配列の候補は、ファイルを暗号化するときは逆に、ファイル後方から前方に向かって求めます。位置 p での key2 の値を key2[p] と書くと、key2[p] と key2[p+1] の間には次の関係が成り立ちます。crcTable の添え字の 1 バイトの値 key2[p]&0xff ^ key1>>24 は、この段階では key1 の値が分からないのでまとめて i としています。

```
                key2[p+1] = crc32(key2[p], key1>>24)
                          = key2[p]>>8 ^ crcTable[X],
              crcTable[X] = key2[p+1] ^ key2[p]>>8,
crcTableInv[key2[p+1]>>24] = key2[p+1]<<8 ^ key2[p] ^ X
                  key2[p] = key2[p+1]<<8 ^ crcTableInv[key2[p+1]>>24] ^ X
```

crcTableInv には、crcTable[i]=c について crcTableInv[c>>24]=c<<8^i をあらかじめ格納しておきます。右辺の^i は key1 の値を推測するときに役立ちます。

下位 2 ビット以外の key2[p+1] が与えられたとき、暗号文と平文から key2[p] の値を列挙し、両辺の 0x0000fc00 の位置のビットが一致するものをフィルタリングすることで、key2[p+1] から key2[p] の下位 2 ビット以外の値を求めることができます。逆に、key2[p] から key2[p+1] の下位 2 ビットの値が定まります。key2[p] の候補が 64 個であり、0x0000fc00 のビットが一致する確率は 1/64 なので、平均的に 1 個の key2[p+1] から 1 個の key2[p] が求まります。

```
#include <cstdio>
#include <cstdint>
#include <vector>
#include <set>
using namespace std;

//  暗号文
vector<uint8_t> C;
//  平文
vector<uint8_t> P;

uint32_t key2Table[256][64];
void makeKey2Table()
{
    Key key;
    int count[256] = {};
    for (uint32_t key2=0; key2<0x10000; key2+=4) {
        key.key2 = key2;
        uint8_t d = key.decrypt_byte();
        key2Table[d][count[d]++] = key2;
    }
}

uint32_t key2[12];

bool guessKey2(int p)
{
    if (p == -1) {
        for (int i=0; i<12; i++)
            printf(" %08x", key2[i]);
        printf("\n");
        return false;
```

```
    }

    uint32_t inv = crcTableInv[key2[p+1]>>24];
    for (int i=0; i<64; i++) {
        uint32_t left = key2Table[C[p+12]^P[p]][i];
        uint32_t right = key2[p+1]<<8 ^ inv;
        if ((left&0xfc00) == (right&0xfc00)) {
            key2[p] = right&0xffff0000 | left;
            key2[p+1] = key2[p+1]&~0x3 | (left^inv)>>8&0x3;
            if (guessKey2(p-1))
                return true;
        }
    }
    return false;
}

void attack() {
    for (uint32_t i=0; i<0x10000; i++)
        for (int j=0; j<64; j++) {
            key2[11] = i<<16 | key2Table[C[11+12]^P[11]][j];
            guessKey2(10);
        }
}

int main()
{
    makeCrcTable();
    makeKey2Table();

    size_t n = 0x5517b;
    //  先頭 12 バイト個は encryption header
    C = vector<uint8_t>(n+12);
    P = vector<uint8_t>(n);

    FILE *f = fopen("secret.zip", "rb");
    fseek(f, 0x46, SEEK_SET);
    fread(&C[0], 1, n+12, f);
    fclose(f);

    f = fopen("kusano_k.png", "rb");
    fread(&P[0], 1, n, f);
    fclose(f);
```

```
    attack();
}
```

```
345c39bc 1283b569 8077e47e f939a88b 51273992 3889e5fd bc5e0aff a3005e7e 8670d…
345c3ac8 1283b56a 8077e47e f939a88b 51273992 3889e5fd bc5e0aff a3005e7e 8670d…
da681d64 13b6f04b 8076d13b f939a9be 51273993 3889e5fd bc5e0aff a3005e7e 8670d…
da695c74 13b6f10a 8076d13a f939a9be 51273993 3889e5fd bc5e0aff a3005e7e 8670d…
 :
```

key2[11] は上位 16 ビットの全ての組み合わせを試してます。下位 2 ビットの値がわからないため、key2[0] は以降の処理では使用しません。guessKey2 が bool 値を返すのは、この後で正しい鍵が見つかった時点で処理を打ち切るように書き換えるためです。

key2 の配列の候補を減らすことを考えます。異なる key2[p+1] から導出した key2[p] がたまたま同じ値になることがあります。ファイルのより後方から、同様の key2 の探索を行い、この際に同じ key2 の値をまとめることで、key2[11] の候補を減らすことができます。

```
void attack() {
    size_t n = 10000;   //  <= P.size();

    set<uint32_t> key2s;
    for (uint32_t i=0; i<0x10000; i++)
        for (int j=0; j<64; j++)
            key2s.insert(i<<16 | key2Table[C[n-1+12]^P[n-1]][j]);
    printf("%d\n", (int)key2s.size());

    for (size_t p=n-2; p>=11; p--) {
        set<uint32_t> key2sTmp;
        for (uint32_t k2: key2s)
            for (int i=0; i<64; i++) {
                uint32_t left = key2Table[C[p+12]^P[p]][i];
                uint32_t right = k2<<8 ^ crcTableInv[k2>>24];
                if ((left&0xfc00) == (right&0xfc00))
                    key2sTmp.insert(right&0xffff0000 | left);
            }
        key2s = key2sTmp;
        printf("%d %d\n", (int)p, (int)key2s.size());
```

```
    }

    for (uint32_t k2: key2s) {
        printf("Try key2=%08x\n", k2);
        key2[11] = k2;
        if (guessKey2(10))
            break;
    }
}
```

```
4194304
9998 2686976
9997 2013184
9996 1560559
  :
14 4114
13 4076
12 4017
11 4174
Try key2=00152d80
Try key2=00207458
 f3075c74 dc250a93 e4d2ea01 8b5a6a00 fde9a310 f34498eb 7a991e30 d1c1feef dddcb…
 f3075ce8 dc250a93 e4d2ea01 8b5a6a00 fde9a310 f34498eb 7a991e30 d1c1feef dddcb…
  :
```

10,000 バイト後方から処理をすることで、key2[11] の候補を 1/1,000 に減らすことができました。

key1

key2 の配列の候補から、key1 の配列の候補を求めます。

key2[p-1] から key2[p] を計算する際には key1[p] の最上位バイト（key1[p]>>24）が使われるので、key2 の値から key1 の最上位バイトを逆算することができます。

```
    key2[p] = crc32(key2[p-1], key1[p]>>24)
            = key2[p-1]>>8 ^ crcTable[key2[p-1]&0xff ^ key1[p]>>24],
```

```
key1[p]>>24 = key2[p]<<8 ^ crcTableInv[key2[p]>>24] ^ key2[p-1]
```

最上位バイトがここで求めた値になるような key1 の配列を、key2 と同様に後ろから計算していきます。key1[p] と key1[p+1] では次の関係が成り立ちます。

```
key1[p+1] = (key1[p] + (key0[p+1]&0xff)) * 0x08088405U + 1,
  key1[p] = (key1[p+1] - 1) * 0xd94fa8cdU - key0[p+1]&0xff
```

0xd94fa8cd は、32 ビット整数の演算 (2^{32} の剰余環）で、1/0x08088405 に相当する値です。0x08088405*0xd94fa8cd = 1 が成り立ちます。

key0[p]&0xff（key0[p] の最下位バイト）が変化しても、key1[p] の最上位バイトはほとんど変わりません。そこで、key1[p-1] の最上位バイトが key2 から求めた値になるという条件でフィルタリングします。key0[p] の最下位バイトの候補は 256 個であり、フィルタリングが通過する確率は 1/256 なので、ここでも 1 個の key1[p+1] から平均 1 個の key1[p] が求められます。key[11] は最上位バイトが固定なので、約 2^{16} 個がフィルタリングを通ります。

```
uint32_t key1[12];

uint32_t key1inv = 0xd94fa8cd;  //  key1inv * 0x08088405 = 1

bool guessKey1(int p)
{
    if (p==2)
        return guessKey0();

    uint32_t base;
    int count;
    if (p==11) {
        uint8_t msb = key2[p]<<8 ^ crcTableInv[key2[p]>>24] ^ key2[p-1];
        //  base-i で最上位バイトが msb になる
        base = ((msb+1)<<24)-1;
        count = 0x1000000;
    } else {
        base = (key1[p+1]-1)*key1inv;
        count = 0x100;
    }
    uint8_t prevMsb = key2[p-1]<<8 ^ crcTableInv[key2[p-1]>>24] ^ key2[p-2];
```

```
    for (int i=0; i<count; i++) {
        uint32_t k1 = base - i;
        if (((k1-1)*key1inv     )>>24 == prevMsb &&
            ((k1-1)*key1inv-0xff)>>24 == prevMsb) {
            key1[p] = k1;
            if (guessKey1(p-1))
                return true;
        }
    }
    return false;
}

bool guessKey2(int p)
{
    if (p == -1)
        return guessKey1(11);
  :
```

key2[p] の全てのビットが定まっているのは p>=1 以降であり、key1 の最上位バイトを求めるためには key2[p] と key2[p-1] を用いることと、key1[p] のフィルタリングに key1[p-1] の最上位バイトが必要であることから、key1[p] が有効なのは p が 3 以上の場合です。

key0

key1[p] と key1[p-1] から次のように key0[p] の最下位バイトが求まります。

```
     key1[p] = (key1[p-1] + (key0[p]&0xff)) * 0x08088405 + 1,
key0[p]&0xff = (key1[p] - 1) * 0xd94fa8cd - key1[p-1]
```

key0 の更新式は key1 や key2 と異なり、追加の入力 P[p] が得られています。

```
          key0[p+1] = crc32(key0[p], P[p])
                    = key0[p]>>8 ^ crcTable[key0[p]&0xff^P[p]],
  key0[p]&0xffffff00 = (key0[p+1] ^ crcTable[key0[p]&0xff^P[p]])<<8
```

key0[4] から key0[7] の最下位バイトを用いて、key0[4] の全てのビットを求めることができ

ます。p=4 について、key0 と key1、key2 が全て求められたので、暗号化と同じ処理を行い key1 から求められる key0 の最下位バイトと矛盾が無いことを確認すれば、この鍵が正しものかどうかが分かります。

```
bool guessKey0()
{
    uint32_t key0 = 0;
    for (int p=7; p>=4; p--) {
        uint8_t lsb = (key1[p]-1)*key1inv - key1[p-1];
        key0 = (key0 ^ crcTable[lsb^P[p]])<<8 | lsb;
    }
    bool ok = true;
    uint32_t k0 = key0;
    for (int i=4; i<12; i++) {
        uint8_t lsb = (key1[i]-1)*key1inv - key1[i-1];
        if ((k0&0xff) != lsb) {
            ok = false;
            break;
        }
        k0 = k0>>8 ^ crcTable[lsb^P[i]];
    }
    if (ok)
        printf("Key found! pos=4, key0=%08x, key1=%08x, key2=%08x\n",
            key0, key1[4], key2[4]);
    return ok;
}
```

```
  :
Try key2=f5a799e8
Try key2=f5a79b68
Try key2=f5b90700
Key found! pos=4, key0=75a4dda6, key1=0eaf62f4, key2=4e243990
```

ファイルの復号

ある位置 p で鍵の値が分かれば、位置 p-1 での鍵の値を求めることができます。同じパスワードであればどのファイルもパスワードを処理した直後の鍵の値は等しいので、そこまで戻り、暗号

化された他のファイルに適用することで、ファイルを復号することができます。

```
int main()
{
    makeCrcTable();

    vector<uint8_t> known(0x55187);
    vector<uint8_t> secret(0x39);

    FILE *f = fopen("secret.zip", "rb");
    fseek(f, 0x46, SEEK_SET);
    fread(&known[0], 1, known.size(), f);
    fseek(f, 0x55221, SEEK_SET);
    fread(&secret[0], 1, secret.size(), f);
    fclose(f);

    uint32_t key0 = 0x75a4dda6;
    uint32_t key1 = 0x0eaf62f4;
    uint32_t key2 = 0x4e243990;
    for (int p=12+3; p>=0; p--) {
        key2 = key2<<8 ^ crcTableInv[key2>>24] ^ key1>>24;
        key1 = (key1-1)*key1inv - (key0&0xff);
        uint8_t plain = ((key2|2)*((key2|2)^1))>>8&0xff ^ known[p];
        key0 = key0<<8 ^ crcTableInv[key0>>24] ^ plain;
    }
    printf("%08x %08x %08x\n", key0, key1, key2);

    Key key;
    key.key0 = key0;
    key.key1 = key1;
    key.key2 = key2;
    for (uint8_t &c: secret) {
        c ^= key.decrypt_byte();
        key.update_keys(c);
    }

    printf("%.*s\n", (int)(secret.size()-12), (const char *)&secret[12]);
}
```

```
e4b115e8 01b5f84c 98b20d06
Leete Latobarita Uruth Ariaroth Bal Netoreel.
```

既知のファイル kusano_k.png から、内部で使用されている鍵 key0 と key1、key2 を探索し、これを用いてファイル secret.txt を復号することができました。

パスワードの復元

ファイルの復号にはパスワードそのものは必須ではなく、パスワードを処理した直後の key0 と key1、key2 で足りますが、これらの値からパスワードを総当たりするよりも速くパスワードを探索することが可能です。

パスワード長が 4 以下の場合

パスワード長 l が 4 以下の場合には、key0 の値から key0 の更新に使われたパスワードを容易に求めることができます。

key0 の更新は CRC の導出そのものです。 CRC には線形性があります。初期値 C に、x0，x1，x2，x3 を与えて crc32 で更新した値が X であり、y0，y1，y2，y3 を与えて更新した値が Y のとき、x0^y0，x1^y1，x2^y2，x3^y3 を与えた結果は X^Y となります。C を 1 ビットだけ変化させるような入力をあらかじめ求めておけば、それらの xor を取ることで、key0 の初期値とパスワードを処理した直後の値から、key0 への入力、すなわちパスワードを求めることができます。

次のプログラムで C を 1 ビットだけ変化させる入力を計算します。

```
int main()
{
    makeCrcTable();
    uint32_t basis[32] = {};
    for (uint32_t x=0xffffffff; x>0; x--) {
        uint32_t crc = 0;
        for (int i=0; i<32; i+=8)
            crc = crc32(crc, (uint8_t)(x>>i));
        for (int i=0; i<32; i++)
            if (crc == 1<<i)
```

```
                basis[i] = x;
    }
    for (int i=0; i<32; i++)
        printf("%08x %08x\n", 1<<i, basis[i]);
}
```

パスワードを求めるプログラムは次のようになります。求めたパスワード用いて鍵を初期化してみることで、パスワードがその長さであったかどうかが分かります。

```
bool checkPassword(uint32_t key0, uint32_t key1, uint32_t key2,
    vector<uint8_t> password)
{
    Key key;
    for (uint8_t c: password)
        key.update_keys(c);
    return key.key0==key0 && key.key1==key1 && key.key2==key2;
}

//  CRCが start から endに変換する入力を求める
vector<uint8_t> recoverCrc(uint32_t start, uint32_t end, int n)
{
    uint32_t basis[32] = {
        0xdb710641U, 0x6d930ac3U, 0xdb261586U, 0x6d3d2d4dU,
        0xda7a5a9aU, 0x6f85b375U, 0xdf0b66eaU, 0x6567cb95U,
        0xcacf972aU, 0x4eee2815U, 0x9ddc502aU, 0xe0c9a615U,
        0x1ae24a6bU, 0x35c494d6U, 0x6b8929acU, 0xd7125358U,
        0x7555a0f1U, 0xeaab41e2U, 0x0e278585U, 0x1c4f0b0aU,
        0x389e1614U, 0x713c2c28U, 0xe2785850U, 0x1f81b6e1U,
        0x3f036dc2U, 0x7e06db84U, 0xfc0db708U, 0x236a6851U,
        0x46d4d0a2U, 0x8da9a144U, 0xc02244c9U, 0x5b358fd3U,
    };

    for (int i=0; i<n; i++)
        start = start>>8 ^ crcTable[start&0xff];

    uint32_t x = 0;
    for (int i=0; i<32; i++)
        if (((start^end)>>i&1) != 0)
            x ^= basis[i];

    vector<uint8_t> input;
```

```
    for (int i=4-n; i<4; i++)
        input.push_back(x>>(8*i)&0xff);
    return input;
}

int main()
{
    makeCrcTable();

    uint32_t key0 = 0x58927f21U;
    uint32_t key1 = 0xc0b74fb5U;
    uint32_t key2 = 0x8297062bU;

    vector<uint8_t> password;
    for (int n=0; n<=4; n++) {
        password = recoverCrc(0x12345678U, key0, n);
        if (checkPassword(key0, key1, key2, password))
            break;
    }
    printf("%.*s\n", (int)password.size(), &password[0]);
    //  p@ss
}
```

パスワード長が 6 の場合

間隔が 4 ならば crc32 を用いた処理の入力が計算できることを利用して、次の順にパスワードを求めます。ここでは、パスワードを x バイト処理した直後の鍵の値を key0_x、key1_x、key2_x としています。鍵の初期値が key0_0、key1_0、key2_0 であり、パスワードの長さが 6 なので key0_6、key1_6、key2_6 が得られています。

1. key2_6 と key1_6 から key2_5 を求める
2. key1_6 と key0_6 から key1_5 を求める
3. key2_5 と key1_5 から key2_4 を求める
4. key2_0 と key2_4 から key1_1、key1_2、key1_3、key1_4 の最上位バイト（Most significant byte、MSB）を求める
5. key1_0 と key1_1 の最上位バイトから、key1_1 と key0_1 の最下位バイト（Least significant byte、LSB）を求める
6. key1_1 と key1_2 の最上位バイトから、key1_2 と key0_2 の最下位バイトを求める
7. key0_0 と key0_1 の最下位バイトから、key0_1 と password_0 を求める

8. key0_1 と key0_2 の最下位バイトから、key0_2 と password_1 を求める
9. key0_2 と key0_6 から、password_2、password_3、password_4、password_5 を求める

図示すると下表になります。 0 は既知の値です。

位置	0	1	2	3	4	5	6
password	7	8	9	9	9	9	0
key0	0	7	8				0
LSB(key0)	0	5	6				0
key1	0	5	6			2	0
MSB(key1)	0	4	4	4	4	2	0
key2	0				3	1	0

パスワード長が 5 か 7 以上の場合

パスワード長が 5 の場合、論文ではパスワード長が 6 の場合の方法を用いていますが、最初の 1 バイトを全探索してパスワード長が 4 の場合の方法を用いることもできます。パスワード長 1 が 7 以上の場合は最初の 1-6 バイトを全列挙して、パスワード長が 6 の場合の方法を適用します。鍵は 2^{96} 通りなので、パスワードを 13 文字まで探索すれば、実際に使用されたパスワードでなくともファイルを復号することが可能なパスワードが見つかる可能性が高いです。

```
bool searchPassword6(vector<uint8_t> prefix, uint32_t key0_6, uint32_t key1_6,
    uint32_t key2_6)
{
    Key key;
    for (uint8_t c: prefix)
        key.update_keys(c);
    uint32_t key0_0 = key.key0;
    uint32_t key1_0 = key.key1;
    uint32_t key2_0 = key.key2;

    uint32_t key2_5 = key2_6<<8 ^ crcTableInv[key2_6>>24] ^ key1_6>>24;
    uint32_t key1_5 = (key1_6-1)*key1inv - (key0_6&0xff);
    uint32_t key2_4 = key2_5<<8 ^ crcTableInv[key2_5>>24] ^ key1_5>>24;

    vector<uint8_t> key1msb = recoverCrc(key2_0, key2_4, 4);
    key1msb.insert(key1msb.begin(), 0);
```

```
    //  key0_1 と key0_2 の LSB を探索
    for (int key0_1lsb=0; key0_1lsb<0x100; key0_1lsb++) {
    uint32_t key1_1 = (key1_0 + key0_1lsb)*0x08088405U + 1;
    if (key1_1>>24 == key1msb[1]) {
        for (int key0_2lsb=0; key0_2lsb<0x100; key0_2lsb++) {
        uint32_t key1_2 = (key1_1 + key0_2lsb)*0x08088405U + 1;
        if (key1_2>>24 == key1msb[2]) {
            //  パスワードの先頭 2 文字を探索
            for (int password_0=0; password_0<0x100; password_0++) {
            uint32_t key0_1 = crc32(key0_0, password_0);
            if ((key0_1&0xff) == key0_1lsb) {
                for (int password_1=0; password_1<0x100; password_1++) {
                uint32_t key0_2 = crc32(key0_1, password_1);
                if ((key0_2&0xff) == key0_2lsb) {
                    //  パスワードの残り 4 文字を探索
                    vector<uint8_t> pass4 = recoverCrc(key0_2, key0_6, 4);
                    vector<uint8_t> password = prefix;
                    password.push_back(password_0);
                    password.push_back(password_1);
                    password.insert(password.end(), pass4.begin(), pass4.end());
                    if (checkPassword(key0_6, key1_6, key2_6, password)) {
                        printf("Password: %.*s\n", (int)password.size(),
                            &password[0]);
                        return true;
                    }
                }}
            }}
        }}
    }}
    return false;
}

bool searchPassword(vector<uint8_t> prefix, uint32_t key0, uint32_t key1,
    uint32_t key2, int l)
{
    if (l==6)
        return searchPassword6(prefix, key0, key1, key2);
    for (int i=0; i<0x100; i++) {
        prefix.push_back(i);
        if (searchPassword(prefix, key0, key1, key2,l-1))
            return true;
        prefix.pop_back();
```

```
    }
    return false;
}

int main()
{
    makeCrcTable();

    uint32_t key0 = 0xe4b115e8U;
    uint32_t key1 = 0x01b5f84cU;
    uint32_t key2 = 0x98b20d06U;

    bool found = false;
    for (int l=0; !found; l++) {
        printf("l = %d\n", l);
        if (l<=4) {
            vector<uint8_t> password = recoverCrc(0x12345678U, key0, l);
            if (checkPassword(key0, key1, key2, password)) {
                printf("Password: %.*s\n", (int)password.size(), &password[0]);
                found = true;
            }
        } else if (l==5) {
            for (int p=0; p<0x100; p++) {
                Key key;
                key.update_keys((uint8_t)p);
                vector<uint8_t> pass4 = recoverCrc(key.key0, key0, 4);
                vector<uint8_t> password(1, (uint8_t)p);
                password.insert(password.end(), pass4.begin(), pass4.end());
                if (checkPassword(key0, key1, key2, password)) {
                    printf("Password: %.*s\n", (int)password.size(),
                        &password[0]);
                    found = true;
                }
            }
        } else
            if (searchPassword({}, key0, key1, key2, l))
                found = true;
    }
}
```

```
l = 0
l = 1
l = 2
l = 3
l = 4
l = 5
l = 6
l = 7
l = 8
Password: p@ssw0rd
```

searchPassword6 は多重ループが深いですが、for の直後の if を通過する候補はそれぞれ平均 1 個です。

鍵やパスワードの探索ではループを回して、直後にフィルタリングをしています。論文にはルックアップテーブルを用いて高速化するという記載がありますが、PkCrack の実装を見ると、かなり複雑な処理が必要なようです。

圧縮されたファイルに対する既知平文攻撃

ここまでは圧縮されていないファイルを扱いましたが、実際に暗号化されている ZIP は中のファイルが圧縮されていることがほとんどでしょう。PNG や Jpeg などはすでに圧縮済みなので、再度圧縮してもサイズはほとんど変わらず、圧縮の意味はあまりありません。しかし、ファイルのヘッダは冗長なデータが多いので圧縮が効き、わざわざ圧縮しないことを指定しなければ、わずかに縮むことにより圧縮ソフトは圧縮することを選択します。圧縮されたファイルに対する既知平文攻撃を 4 個の場合に分けて考えます。

ファイル全体が既知であるとき

筆者の知る限り、圧縮ソフトは圧縮時にランダムな処理を行うことは無いので、同じソフトで同じファイルを同じ設定で圧縮すれば、同じ圧縮データが出力されます。ファイルを作成した人の環境を推測しながら、いくつかのソフトで圧縮率を変えながら試してみれば良いでしょう。時間を掛けてパスワードの探索を行わなくても、ファイルサイズが攻撃対象の ZIP と同じになることが目安となります。

ファイルの先頭数百 KB が既知であるとき

ファイルの先頭部分が一致していても、圧縮後の先頭部分は一致しません。Deflate 圧縮されたデータの先頭にはハフマン符号表が付き、ハフマン符号表の中身は deflate ブロック全体の符号の出現率に依存するからです。

圧縮ソフトの実装に依存しますが、Info-ZIP の deflate では、リテラル・一致長 32,768 個ごとにブロックに区切られていました。先頭のブロックの圧縮前のデータが同じならば、圧縮後のブロックも等しくなるでしょう。

ただ、ファイルの先頭数百 KB が既知であるという状況はあまり無さそうです。

先頭 16 バイト程度が既知で、固定ハフマン符号が用いられている場合

Deflate 圧縮でどれだけ前のデータを保持しているかや、探索にどの程度力を入れるかは圧縮ソフトや設定に依存します。しかし、各ファイルフォーマットのヘッダ程度のサイズであれば、圧縮結果に差はほとんどなく、固定ハフマン符号が用いられていると仮定すれば圧縮結果も推測できるでしょう。

圧縮ソフトのアルゴリズムによっては、同じバイトの連続が 2 個目から一致長と一致距離に符号化されるとは限らないことに注意する必要があります。例えば、`０００００００`というデータを圧縮するとき、`0（5，1）`と圧縮するのが最適ですが、`００（4，1）`と圧縮されることがあります。

先頭 16 バイト程度が既知で、カスタムハフマン符号が用いられている場合

このケースが一番多いと思いますが、攻撃が最も困難です。

先述したようにブロック全体の符号の出現率でハフマン符号表が変化するので、ハフマン符号表の部分を推測することが現実的ではありません。既知の先頭部分がどのように符号化されるかを推測するしかないでしょう。

ハフマン符号表の内容によって圧縮されたハフマン符号表の長さも異なります。いくつかのデータで試してみたところ、50 バイトから 100 バイト程度になりました。ハフマン符号表のサイズをビット単位で総当たりする必要があります。

手元にあった PDF を圧縮してみたところ、最初のブロックのリテラル・一致長の各ビット長はこのようになっていました。

```
  8  8  8  8  8  8  8  9  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  9  8  8  8  9  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  9  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
  8  8  9  8  8  8  8  8  8  9  8  8  8  8  8  8
 15  9  9 11 11 12 12 14 13 11 10 11  0  9  0 13
 15  9  0 14  0  0  0  0 14
```

0 から 255 は、6 個が 9 である以外は、全て 8 となっていました。これは PDF のヘッダ以降は 0 から 256 がほぼ均等に出現するためです。他のファイルフォーマットでも、ヘッダ以外は圧縮されていることが多く、圧縮されていれば各数値の出現率はほぼ一定でしょう。

このとき、0x00 から 0x07 のビット列には、0 から 7 のリテラルがそのまま割り当てられます（ハフマン圧縮されたビット列は上位ビットから格納されるため、ビット順は逆になります）。0x08 には 9、0x09 には 10 と以降は 1 個ずつずれた値が割り当てられます。

0 から 255 までには多数のビット長 8 と、小数のビット長 9 が割り当てられると仮定して、9 の位置を総当たりすれば、ハフマン符号表の長さの総当たりと合わせて攻撃ができる可能性があります。とはいえ、既知の部分のサイズが小さいと既知平文攻撃には時間が掛かることもあり、実際に攻撃しようとすると多量の計算機資源が必要になるでしょう。

Stay の攻撃法

ここまで見てきたように ZIP の暗号は既知平文攻撃に対して脆弱です。さらに、平文が全く分からない場合でも攻撃可能な方法が Stay によって提案されています[*3]。

論文が発表されたのは 2001 年なのでとっくに修正されているかと思いましたが、現在でもこの攻撃法は有効でした。ZIP の暗号化はそもそも脆弱なので実装のみを強化しても意味が無いという判断なのかもしれません。Info-ZIP の実装は Linux の `zip` コマンドとして広く使われています。

この攻撃法では、Info-ZIP の実装に不備があり、encryption header の乱数が予測可能であることを利用します。 Encryption header の乱数を予測することで既知平文とします。後述するように、生成された乱数をそのまま encryption header とするのではなく、乱数を一度 ZIP の暗号化と同じ方法で暗号化する（生成された乱数は 2 回暗号化される）ため、Biham と Kocher の攻撃法を用いることはできません。

Stay はある 1 バイトの暗号化には鍵の一部の情報しか寄与しないことに着目しました。寄与する部分のみを総当たりする、実際に暗号化の処理を行って結果が一致するもののみをフィルタリングして次の総当たりに回す、ということを繰り返します。Encryption header の 2 回の暗号化には同じ鍵が使われるので、1 回目の暗号化の結果（2 回目の暗号化の平文）も総当たりすることで、Info-ZIP の暗号化に対しても適用することができます。1 個のファイルに対する攻撃では Biham と Kocher の攻撃法よりも計算方法が大きくて実用にはなりませんが、パスワードを処理した直後の鍵は同じパスワードで暗号化された全てのファイルで共通であり、複数のファイルでフィルタリングすることで高速化しています。

論文には「5 個のファイルがあれば解ける」と書かれていますが、計算資源を注ぎ込めばより少ないファイルでも解読できそうです。以降の説明では 5 個のファイルを用いています。

処理の流れは次のようになります。`key0`、`key1`、`key2` はパスワードを処理した直後の値、`R`、`P`、`C` はそれぞれ暗号化前、1 回目の暗号化後、2 回目の暗号化後（対象の ZIP に保存される）の encryption header です。

1. 各ファイルの encryption header の先頭 1 バイトを集めて、乱数のシードを推測し、`R` を求める

[*3] STAY, Michael. ZIP attacks with reduced known plaintext. In: International Workshop on Fast Software Encryption. Springer, Berlin, Heidelberg, 2001. p. 125-134.

2. LSB(crc32(key0,0))、MSB(key1*0x08088405)、crc32(key2,0)&0x0000fffc、R[0]^P[0] を総当たりする（2^{38} ビット）
3. 各ファイルについて、Encryption header の 2 バイト目の値でフィルタリングする（5 ファイルの場合、$2^{-6.5}$）
4. crc32(key0,0)&0x0000ff00、MSB(key1*0x08088405*0x08088405)、crc32(key2,0)&0x00ff0003 を総当たりする（2^{26} ビット）
5. 各ファイルについて、Encryption header の 3 バイト目の値でフィルタリングする（5 ファイルの場合、$2^{-6.5}$）
6. ここまでに定まっている crc32(key2,0) の 24 ビットと、R[0]^P[0] から key2 を復元する（平均 1 個）
7. ここまでに定まっている情報から key1 と key0 を総当たりする（最大 2^{32} ビット）

乱数シードの推測

まずは攻撃に使用する ZIP を作成します。このファイルも奥付のサイトで公開しています。

```
echo -n cocoa > cocoa.txt
echo -n chino > chino.txt
echo -n rize > rize.txt
echo -n chiya > chiya.txt
echo -n syaro > syaro.txt

$ zip -e -0 rabbit.zip cocoa.txt chino.txt rize.txt chiya.txt syaro.txt
Enter password:
Verify password:
  adding: cocoa.txt (stored 0%)
  adding: chino.txt (stored 0%)
  adding: rize.txt (stored 0%)
  adding: chiya.txt (stored 0%)
  adding: syaro.txt (stored 0%)
```

Info-ZIP の encryption header の初期化とデータの暗号化は次の擬似コードです。

```
if (最初のファイルならば)
    srand(time(nullptr)^pid());
for (int i=0; i<10; i++)
```

```
    header[i] = rand()>>7&0xff;
encrypt(header, 10, password);
header[10,11] = 更新時刻（CRC の 2 バイトの代わりに使われる）;
encrypt(header+data, 12+dataSize, password);
```

ZIP の暗号では、平文は decrypt_byte(key2) と xor されます。その後、入力の平文を用いて update_key(plain[i]) が実行されるので、平文が異なれば鍵の状態が異なっていきます。各ファイルごとに異なる乱数で初期化された encryption header をまず暗号化するのはこれが目的です。header[0] と xor する値の導出には常に、パスワードを処理した直後の key2 が用いられます。異なる平文を入力する前なので、この値は header[0] の 2 回の暗号化で共通です。同じ値で 2 回 xor を掛けているので、ZIP ファイル中の header[0] は rand()>>7&0xff の値そのままとなります。

5 個のファイルから rand()>>7&0xff の値が 5 個手に入るので、総当たりによってシード値を求めることができます。ただ、論文が執筆された頃と異なり、現在の glibc の rand の実装は線形合同法ではありません[*4]。次のプログラムは ZIP を暗号化した環境と rand の実装が同じ環境で実行する必要があります。

```
#include <cstdio>
#include <cstdint>
#include <cstdlib>
#include <cstring>
#include <ctime>
using namespace std;

uint8_t R[5][10];
uint8_t C[5][12];

void guessRand()
{
    for (int i=0; ; i++) {
        int seed = time(nullptr)^i;
        srand(seed);
        bool ok = true;
        for (int j=0; j<5; j++) {
            if ((rand()>>7&0xff) != C[j][0])
                ok = false;
```

[*4] http://inaz2.hatenablog.com/entry/2016/03/07/194000

```
            for (int k=0; k<9; k++)
                rand();
        }
        if (ok) {
            printf("Seed: %08x\n", seed);
            srand(seed);
            for (int j=0; j<5; j++)
                for (int k=0; k<10; k++)
                    R[j][k] = rand()>>7&0xff;
            break;
        }
    }
}

int main()
{
    makeCrcTable();

    int pos[5] = {0x43, 0xa7, 0x10a, 0x16d, 0x1d1};
    FILE *f = fopen("rabbit.zip", "rb");
    for (int i=0; i<5; i++) {
        fseek(f, pos[i], SEEK_SET);
        fread(C[i], 1, 12, f);
    }
    fclose(f);

    guessRand();
    //uint8_t R_[5][10] = {
    //    {0xdd, 0xb7, 0x42, 0x72, 0x6c, 0x74, 0x9d, 0x9c, 0xe6, 0x40},
    //    {0xf7, 0x39, 0xf5, 0x01, 0x4f, 0x1b, 0x45, 0xcf, 0x60, 0x3f},
    //    {0xc8, 0xfa, 0x4b, 0xec, 0x32, 0x77, 0xab, 0x9d, 0x58, 0xfd},
    //    {0x24, 0x35, 0xb5, 0x67, 0xa7, 0x21, 0xdc, 0x45, 0xbd, 0xc2},
    //    {0x85, 0xb5, 0xfb, 0x7a, 0xb6, 0x4b, 0x95, 0xfb, 0x1b, 0xf6},
    //};
    //memcpy(R, R_, sizeof R);


    if (attack0()) {
        printf("key0: %08x\n", key0);
        printf("key1: %08x\n", key1);
        printf("key2: %08x\n", key2);
    }
```

```
    printf("%.2f\n", (double)clock()/CLOCKS_PER_SEC);
}
```

実装が線形合同法ではなくなり、シード値の 2^{32} の総当たりは重かったので、暗号化した環境の PID の推測と暗号化した日時の推測を兼ねて、`time(nullptr)^i` としています。暗号化された日時が分かっているならば、`time(nullptr)` の部分を置き換えるとより高速になります。

1 段階目の総当たり

特筆することはありません。素直に総当たりをしています。ループの横に `rabbit.zip` を攻撃する場合の正解を書いています。攻撃には時間が掛かるので、プログラムの作成中はこの値で置き換えて、以降の処理で正しく鍵が探索できるかを確認しました。

```
bool attack0()
{
    //  LSB(crc32(key0,0))
    for (uint32_t g0=0; g0<0x100; g0++) {   //  0x5f
        key0_crc = g0;
        //  MSB(key1*0x08088405)
        for (uint32_t g1=0; g1<0x100; g1++) {   //   0xfe
            key1_mul1 = g1<<24;
            //  crc32(key2,0)&0x0000fffc
            for (uint32_t g2=0; g2<0x4000; g2++) {  //  0x5ce
                key2_crc = g2<<2;
                //  R[x][0]^P[x][0]
                for (uint32_t g3=0; g3<0x100; g3++) {   //  0xa9
                    s0[0][0] = g3;
                    if (attack1(0))
                        return true;
                }
            }
        }
    }
    return false;
}
```

Encryption header の 2 バイト目によるフィルタリング

総当たりした値で、各ファイルの 2 バイト目の 2 回の暗号化を実行し、値が一致する場合のみ次に進みます。以降の処理で使用するため、1 バイト目を処理した後の鍵の値の最下位バイトや最上位バイトを保存しています。

g0 と g1 はキャリーです。例えば、x=y=0x01800000 のとき、msb(x+y)=msb(x)+msb(y)+1=3 となります。この 1 です。各ファイルについて、追加で 2 ビットのキャリーを総当たりして、1 バイト（8 ビット）の一致を確認しているので、平均して 2^6 個に 1 個の鍵が通過することになります。

```
bool attack1(int f)
{
    if (f==5)
        return attack2();

    for (uint32_t g0=0; g0<2; g0++)
    for (uint32_t g1=0; g1<2; g1++) {
        key0_10lsb[f] = key0_crc^crcTable[R[f][0]];
        key1_10msb[f] = msb(key1_mul1) + msb(key0_10lsb[f]*0x08088405) + g0;
        s0[f][1] = decrypt_byte(crcTable[key1_10msb[f]] ^ key2_crc);
        key0_11lsb[f] = key0_crc^crcTable[R[f][0]^s0[0][0]];
        key1_11msb[f] = msb(key1_mul1) + msb(key0_11lsb[f]*0x08088405) + g1;
        s1[f][1] = decrypt_byte(crcTable[key1_11msb[f]] ^ key2_crc);
        if ((R[f][1]^s0[f][1]^s1[f][1]) == C[f][1])
            if (attack1(f+1))
                return true;
    }
    return false;
}
```

2 段階目の総当たり

処理の様子を確認するために、1 段階目で総当たりした値を表示しています。1 回目のフィルタリングを通過してここに来るのは、1 回の攻撃で平均 256 回です。

```
bool attack2()
{
    printf("attack2 %08x %08x %08x\n", key0_crc, key1_mul1, key2_crc);

    //  crc32(key0,0) & 0x0000ff00
    for (uint32_t g0=0; g0<0x100; g0++) {   //  0xdb
        key0_crc = key0_crc&~0xff00 | g0<<8;
        //  MSB(key1*0x08088405*0x08088405)
        for (uint32_t g1=0; g1<0x100; g1++) {   //  0x7f
            key1_mul2 = g1<<24;
            //  crc32(key2,0)&0x00ff0003
            for (uint32_t g2=0; g2<0x100; g2++) //  0xfb
            for (uint32_t g3=0; g3<4; g3++) {   //  0
                key2_crc = g2<<16 | key2_crc&~0xff0003 | g3;
                if (attack3(0))
                    return true;
            }
        }
    }
    return false;
}
```

Encryption header の 3 バイト目によるフィルタリング

2 バイト目よりも処理が複雑になっていますが、2 バイト目と同様に、encryption header の 3 バイト目を計算して正しく暗号文を生成できるもののみを通しています。

key1 の計算について、Stay の論文には 0x08 の記載が抜けているようです。1 回目の key1 の計算の +1 と、2 回目の key1 の計算での 0x08088405 との乗算で出てくるので加えています。

```
bool attack3(int f)
{
    if (f==5)
        return attack4();

    for (uint32_t g0=0; g0<2; g0++)
    for (uint32_t g1=0; g1<2; g1++) {
        key0_20lsb[f] = crc32(key0_crc^crcTable[R[f][0]], R[f][1]);
```

```
        key1_20msb[f] = msb(key1_mul2) + 0x08 + g0 +
            msb(key0_10lsb[f]*0xd4652819+key0_20lsb[f]*0x08088405);
        uint32_t k20 = crc32(key2_crc^crcTable[key1_10msb[f]], key1_20msb[f]);
        uint8_t s20 = decrypt_byte(k20);
        key0_21lsb[f] = crc32(key0_crc^crcTable[R[f][0]^s0[0][0]],
            R[f][1]^s0[f][1]);
        key1_21msb[f] = msb(key1_mul2) + 0x08 + g1 +
            msb(key0_11lsb[f]*0xd4652819+key0_21lsb[f]*0x08088405);
        uint32_t k21 = crc32(key2_crc^crcTable[key1_11msb[f]], key1_21msb[f]);
        uint8_t s21 = decrypt_byte(k21);
        if ((R[f][2]^s20^s21) == C[f][2])
            if (attack3(f+1))
                return true;
    }
    return false;
}
```

key2 の復元

crc(key2,0) の下位 24 ビットと、decrypt_byte(key2) をすでに決定しているので、ここから key2 を復元します。

```
bool attack4()
{
    printf("  attack4 %08x %08x %08x\n", key0_crc, key1_mul2, key2_crc);
    //  key2
    for (uint32_t g11=0; g11<0x100; g11++) {
        key2 = (key2_crc ^ crcTable[g11])<<8 | g11;
        if (decrypt_byte(key2) == s0[0][0])
            if (attack5())
                return true;
    }
    return false;
}
```

key1 の復元

key1 は 16 ビット分の情報が定まっているので、残りの 16 ビットを総当たりします。ここまで決めてきたキャリーの値は矛盾している可能性があります。例えば、ある値によって繰り上がりが生じるのに、より大きな値で繰り上がりが無いということはありえません。総当たりした key1 の値でもう一度計算を行い、正しいもののみをフィルタリングしています。

```
bool attack5()
{
    //  key1
    for (uint32_t g12=0; g12<0x1000000; g12++) {
        key1 = (key1_mul1 | g12) * 0xd94fa8cdU;
        if (msb(key1*0xd4652819U) == msb(key1_mul2)) {
            bool ok = true;
            for (int f=0; f<5 && ok; f++) {
                uint32_t k1 = (key1 + key0_10lsb[f])*0x08088405U + 1;
                if (msb(k1)!=key1_10msb[f]) {ok=false; break;}
                k1 = (k1 + key0_20lsb[f])*0x08088405U + 1;
                if (msb(k1)!=key1_20msb[f]) {ok=false; break;}
                k1 = (key1 + key0_11lsb[f])*0x08088405U + 1;
                if (msb(k1)!=key1_11msb[f]) {ok=false; break;}
                k1 = (k1 + key0_21lsb[f])*0x08088405U + 1;
                if (msb(k1)!=key1_21msb[f]) {ok=false; break;}
            }
            if (ok)
                if (attack6())
                    return true;
        }
    }
    return false;
}
```

key0 の復元

key0 の値も 16 ビットが決定済みです。8 ビットずつ探索することも可能ですが、ここに来るまでに候補の個数は充分に減っているため、単純に 16 ビットを全て総当たりしています。

これで key0、key1、key2 の全てのビットが求められたので、encryption header を暗号化してみて暗号化された ZIP 中の encryption header が生成されれば、正しい鍵です。ここまでに使用した R と C は 3 バイト目までであり、残りの 7 バイトで $1/2^{56}$ に絞れるので確率的に充分です。

```
bool attack6()
{
    //  key0
    for (uint32_t g0=0; g0<0x100; g0++)
    for (uint32_t g1=0; g1<0x100; g1++) {
        uint32_t key0_h = g1<<16 | key0_crc;
        key0 = (key0_h^ crcTable[g0])<<8 | g0;

        Key keys0;
        keys0.key0 = key0;
        keys0.key1 = key1;
        keys0.key2 = key2;
        Key keys1 = keys0;
        bool ok = true;
        for (int i=0; i<10 && ok; i++) {
            uint8_t t = C[0][i] ^ keys1.decrypt_byte();
            keys1.update_keys(t);
            t ^= keys0.decrypt_byte();
            keys0.update_keys(t);
            if (t != R[0][i])
                ok = false;
        }
        if (ok)
            return true;
    }
    return false;
}
```

実際の実行の様子です。

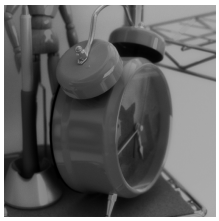
```
Seed: 5a2acd2a
attack2 00000001 02000000 000051f0
attack2 0000ff01 02000000 0000d1f0
attack2 0000ff01 82000000 000052d0
  :
  attack4 0000db5f 7f000000 00439418
  attack4 0000db5f 7f000000 00c39418
```

```
attack2 0000ff5f fe000000 00001738
  attack4 0000cb5f ff000000 007b1738
  attack4 0000cb5f ff000000 00fb1738
  attack4 0000db5f 7f000000 007b1738
  attack4 0000db5f 7f000000 00fb1738
key0: e4b115e8
key1: 01b5f84c
key2: 98b20d06
2870.06
```

全探索空間の 37% くらいのところ（最も外側のループは `LSB(crc32(key0,0))`）で解が見つかり、実行時間は Core i7-4790 3.6GHz で 1 時間弱でした。 論文には、Pentium II 500 MHz で 2 時間未満と書かれているので、もう少し高速化できそうです。このプログラムでは、キャリーの情報は `key1` の全てのビットが求められてからのフィルタリングにしか使用していませんが、`key1*0x08088405` の上限や下限を定めるのに活用できるそうです。

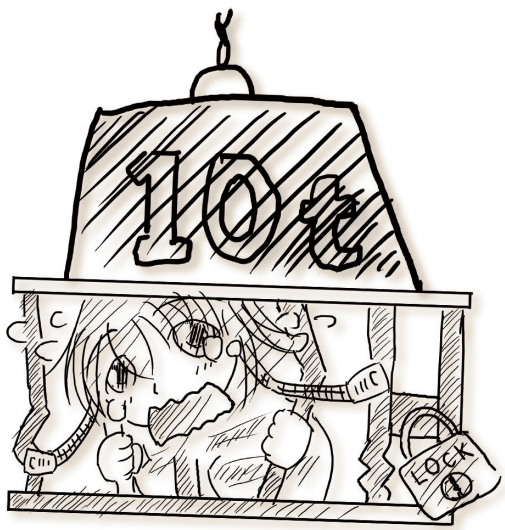
著者近影

@kusano_k



書名	ZIP、完全に理解した
発行日	2017 年 12 月 29 日（初版第 1 刷）
	2018 年 1 月 10 日（初版第 2 刷）
発行者	@kusano_k
ウェブサイト	http://sanya.sweetduet.info/understandzip/
	https://github.com/kusano/understandzip_web
連絡先	https://twitter.com/kusano_k
	kusano4a+understandzip@gmail.com
印刷所	株式会社ポプルス

ZIP、完全に理解した



@kusano_k