A Differential Meet-in-the-Middle Attack on the Zip cipher

Michael Stay

Pyrofex Corp., USA, stay@pyrofex.net

Abstract. We report the successful recovery of the key to a Zip archive containing only two encrypted files. The attack improves on our 2001 ciphertext-only attack, which required five encrypted files. The main innovations are a new differential meet-in-the-middle attack for the initial stages and the use of lattice reduction to recover the internal state of the truncated linear congruential generator.

Keywords: Cryptanalysis · Stream cipher

Contents

1 Introduction

PKZIP is data compression software written by Phil Katz. Roger Schafly designed the original stream cipher used by PKZIP to encrypt files in the archive, herein called the Zip cipher. PKZIP has since adopted strong encryption by default. Katz described the Zip archive file format and the Zip cipher in a file named `appnote.txt` that has been bundled with every version of the software [5].

Info-Zip is an open-source implementation of the Zip software. Most Zip software other than PKZIP has been based on the Info-Zip implementation. In 2001, we described a ciphertext-only attack on implementations derived from Info-Zip's codebase [7]. After that attack was published, most commercial Zip software upgraded to using strong cryptography, but Info-Zip still defaults to the Zip cipher.

The Zip cipher, which we will describe in detail in section 2, is a plaintext feedback stream cipher: it begins with a known initial state, then updates the state by consuming the password, ten salt bytes, two CRC bytes, and the plaintext. After each input byte, the cipher updates its internal state and outputs a stream byte. The stream bytes corresponding to the password are discarded; the remainder get xored with the salt, CRC, and the plaintext. The internal state of the cipher consists of three 32-bit words: `keys[0]`, `keys[1]`, and `keys[2]`. The words `keys[0]` and `keys[2]` are the states of two copies of CRC32, a linear feedback shift register (LFSR). The word `keys[1]` is the internal state of a truncated linear congruential generator (TLCG). The LFSR is linear with respect to xor, while the TLCG is linear with respect to addition modulo 2^{32}. The cipher gets its strength by alternating between the incompatible linear structures, taking the output of one register or generator as input to the next. The output byte of the cipher is a nonlinear function of some bits in the middle of `keys[2]`.

PKZIP gets entropy for the ten salt bytes of each file by allocating uninitialized memory. Info-Zip and other software based on it use the timestamp xor the process ID to initialize C's `rand` pseudorandom number generator. It appears that the authors of Info-Zip knew about Biham and Kocher's known-plaintext attack [1] and knew that encrypting the output of `rand` would provide known plaintext. Instead, Info-Zip generates salt by generating pseudorandom bytes with `rand` and then encrypting them with the password. The salt is then encrypted a second time, followed by the CRC and the compressed plaintext. Unfortunately, because the internal state of cipher is the same during both encryptions of the last password byte, the first stream byte is the same both times. Since any stream byte xor itself is zero, the first byte of each encrypted file in the archive leaks about eight bits of the 31-bit internal state of `rand`.

In the 2001 attack, given an archive with five encrypted files, we recover the internal state of `rand`, giving doubly-encrypted plaintext. We then do several stages of guessing key material and decrypting one byte of each file twice. If the result is consistent with the output of `rand`, we move onto the next stage; if not, we backtrack. The Zip cipher has 96 bits of key material. The five bytes at each stage allow us to eliminate all but one in 2^{40} key guesses; after four stages, we expect to have the unique key.

In August of 2019, we were approached by a man who had lost the password to a zip archive. The archive contained only two files. He had read our 2001 paper and asked whether the attack described there would allow him to open the file. He had the laptop on which the file was created, the timezone, and the timestamp. The timestamp provided sixteen bits of entropy, while the first byte of the two encrypted files provided another sixteen, leaving approximately one candidate internal state for `rand`. However, with only sixteen bits to filter each stage's guessed key material instead of forty, many more candidates would pass each stage, exponentially increasing the amount of work required to find the key. Our initial analysis said it would be feasible, but would cost on the order of ∼\$100,000 of GPU time. Since the zip archive contained information necessary for recovering several hundred thousand dollars worth of Bitcoin, he was willing to pay that much for a successful key recovery. We drew up a contract and began a more detailed analysis.

The analysis revealed two major problems to solve. The first was how to get sets of key candidates over the network to the GPUs fast enough that the GPUs would not be sitting idle. The second was how to find the internal state of a truncated linear congruential generator given a few outputs. In the 2001 attack, there was only one key candidate after stage 4, so we could simply try all 2^{32} possibilities. However, because we had only sixteen bits to filter each key guess with instead of forty, we expected around 2^{52} keys instead of only one; the naive approach would mean trying 2^{84} keys, which was completely infeasible.

To solve the first problem, we came up with a differential meet-in-the-middle (DMITM) attack that not only drastically reduced the amount of work for the first two stages, it let us generate key material in place rather than needing to send it over the network. To solve the second, we used the lattice reduction technique of Freize, Hastad, Kannan, Lagarias, and Shamir [2] to reduce the number of candidates from 2^{32} to 35.

Due to mixing up the thread index and block index in our GPU code, we spent ten days running through the entire keyspace once and failed to find the key. Once we found the bug, we ran the attack again and succeeded after a day and a half. The total cost for GPU time was around $7K. The client was very pleased and gave us a nice bonus.

The full source code is available at https://gitlab.com/pyrofex/breakzip/.

2 The Zip cipher

Here we quote from the Info-Zip codebase [3], slightly edited for clarity. The variable `crctab` refers to an array of 256 thirty-two-bit words. It implements a linear function, *i.e.* `crctab[i ^ j] = crctab[i] ^ crctab[j]`.

The CRC32 function takes the current state `c` of the LFSR, a byte `b`, and `crctab` as inputs. It outputs the new state of the LFSR. The index into `crctab` is given by the eight bits shifted out of the register xored with the input byte; the resulting word is fed back into the state.

```
#  define CRC32(c, b, crctab) (crctab[((int)(c) ^ (b)) & 0xff] ^ ((c) >> 8))
```

To prime the cipher, the `keys` array gets initialized and then gets updated with each byte of the password.

```
/***********************************************************************
 * Initialize the encryption keys and the random header according to
 * the given password.
 */
void init_keys(__G__ passwd)
    __GDEF
    ZCONST char *passwd; /* password string with which to modify keys */
{
    GLOBAL(keys[0]) = 0x12345678;
    GLOBAL(keys[1]) = 0x23456789;
    GLOBAL(keys[2]) = 0x34567890;
    while (*passwd != '\0') {
        update_keys(__G__ (int)*passwd);
        passwd++;
    }
}
```

Each of the three words of the `keys` array gets updated in turn. First, `keys[0]` gets updated using the input byte. Next, `keys[1]` gets updated using the low byte of the new state of `keys[0]`. Finally, `keys[2]` gets updated using the high byte of the new state of `keys[1]`.

```
/***********************************************************************
 * Update the encryption keys with the next byte of plain text
 */
int update_keys(__G__ c)
    __GDEF
    int c;                          /* byte of plain text */
{
    GLOBAL(keys[0]) = CRC32(GLOBAL(keys[0]), c, CRY_CRC_TAB);
    GLOBAL(keys[1]) = (GLOBAL(keys[1])
                      + (GLOBAL(keys[0]) & 0xff))
                      * 0x08088405 + 1;
    {
      register int keyshift = (int)(GLOBAL(keys[1]) >> 24);
      GLOBAL(keys[2]) = CRC32(GLOBAL(keys[2]), keyshift, CRY_CRC_TAB);
    }
    return c;
}
```

Once the cipher has been primed, it will start generating stream bytes. The output byte depends nonlinearly on bits 2 through 15 of `keys[2]`.

```
/***********************************************************************
 * Return the next byte in the pseudo-random sequence
 */
int decrypt_byte(__G)
    __GDEF
{
    unsigned temp;

    temp = ((unsigned)GLOBAL(keys[2]) & 0xffff) | 2;
    return (int)(((temp * (temp ^ 1)) >> 8) & 0xff);
}
```

Info-Zip generates salt bytes by using the timestamp xor the process ID (`ZCR_SEED2` below) as the seed to `rand`, and then encrypts the output of `rand` using the password. The cipher is then re-initialized; after consuming the password, the cipher encrypts the ten salt bytes again and two CRC bytes to produce the encryption header. The `zencode` function calls `update_keys`, then returns the output of `decrypt_byte`.

```
/***********************************************************************
 * Write encryption header to file zfile using the password passwd
 * and the cyclic redundancy check crc.
 */
void crypthead(passwd, crc)
    ZCONST char *passwd;         /* password string */
    ulg crc;                     /* crc of file being encrypted */
{
    int n;                       /* index in random header */
    int t;                       /* temporary */
    int c;                       /* random byte */
    uch header[RAND_HEAD_LEN];   /* random header */
    static unsigned calls = 0;   /* ensure different random header each time */

    /* First generate RAND_HEAD_LEN-2 random bytes. We encrypt the
     * output of rand() to get less predictability, since rand() is
     * often poorly implemented.
     */
    if (++calls == 1) {
        srand((unsigned)time(NULL) ^ ZCR_SEED2);
    }
    init_keys(passwd);
    for (n = 0; n < RAND_HEAD_LEN-2; n++) {
        c = (rand() >> 7) & 0xff;
        header[n] = (uch)zencode(c, t);
    }
    /* Encrypt random header (last two bytes is high word of crc) */
    init_keys(passwd);
    for (n = 0; n < RAND_HEAD_LEN-2; n++) {
        header[n] = (uch)zencode(header[n], t);
    }
    header[RAND_HEAD_LEN-2] = (uch)zencode((int)(crc >> 16) & 0xff, t);
    header[RAND_HEAD_LEN-1] = (uch)zencode((int)(crc >> 24) & 0xff, t);
    bfwrite(header, 1, RAND_HEAD_LEN, BFWRITE_DATA);
}
```

Finally, the compressed plaintext gets encrypted.

3 Analysis of work required to mount the 2001 attack

Because there is limited diffusion between the three `keys`, the stream byte does not depend on all the bits of the state. Given the low eight bits of `CRC32(keys[0], 0)`, the high eight bits of `keys[1] * 0x08088405`, bits 2 through 15 of `keys[2]`, and whether a carry into the most significant byte of `keys[1]` occurs, there is a single output stream byte for any input byte.

In the 2001 attack, given an archive with five encrypted files, we recover the internal state of `rand`, giving doubly-encrypted plaintext. We then do several stages of guessing key material and decrypting one byte of each file twice. We guess somewhat more than the thirty-one bits above, because we have no information about the zeroth stream byte and because we guess one carry bit per encryption per file. Rather than guess more bits of `keys[1] * 0x08088405`, at stage n we guess the high eight bits of `keys[1] * 0x08088405**n`, where `**` denotes exponentiation. If the result is consistent with the output of `rand`, we move onto the next stage; if not, we backtrack. The Zip cipher has 96 bits of key material. The five bytes at each stage allow us to eliminate all but one in 2^{40} key guesses; after four stages, we expect to have the unique key.

In this case, we had only two files instead of five. The timestamp let us derive a single seed for `rand`, but we had far fewer bits to filter with.

In stage 1, we would need to guess bits 0 to 23 of `keys[2]`, bits 0 to 7 of `CRC32(keys[0], 0)`, bits 24 to 31 of `(keys[1] * 0x08088405)`, and two carry bits for each of the two files, a total of 44 bits. We get 16 bits to filter with from the encryption header, resulting in 2^{28} key candidates and 2^{46} encryptions.

In stage 2, for each of the key candidates from the previous stage we would need to guess bits 24 to 31 of `keys[2]`, bits 8 to 15 of `CRC32(keys[0], 0)`, bits 24 to 31 of `(keys[1] * 0xD4652819)`, and two carry bits for each of the two files, a total of 28 bits. We get 16 bits to filter with from the encryption header, resulting in 2^{40} key candidates and 2^{58} encryptions.

In stage 3, for each of the key candidates from the previous stage we would need to guess bits 16 to 23 of `CRC32(keys[0], 0)`, bits 24 to 31 of `(keys[1] * 0x576EAD7C)`, and two carry bits for each of the two files, a total of 20 bits. We get 16 bits to filter with from the encryption header, resulting in 2^{44} key candidates and 2^{62} encryptions.

In stage 4, for each of the key candidates from the previous stage we would need to guess bits 24 to 31 of `(keys[1] * 0x1201d271)`, a total of 8 bits. We would try the 2^{32} possible initial values of `keys[1]` and filter them based on the previous guesses. We get 24 filter bits from previous guesses for the high bytes and 12 carry bits setting upper or lower bounds on the low 24 bits at stage n. At most one in 2^{16} key candidates pass this stage, resulting in 2^{28} key candidates with 2^{84} encryptions.

In stage 5, for each of the key candidates from the previous stage we would need to guess bits 24 to 32 of `CRC32(keys[0], 0)`, a total of 8 bits. We get 16 bits to filter with from the encryption header, resulting in 2^{20} key candidates and 2^{36} encryptions.

In stage 6, we do not need to guess any more key material. We get 16 bits to filter with from the encryption header, resulting in 2^{4} key candidates and 2^{20} encryptions.

In stage 7, we do not need to guess any more key material. We get 16 bits to filter with from the encryption header, resulting in a single key candidates and 2^{4} encryptions.

4 Lattice reduction

The 2^{84} encryptions required in stage 4 make the 2001 attack cost far more than necessary. Freize, Hastad, Kannan, Lagarias, and Shamir [2] described a method of recovering the internal state of a TLCG given several bytes of output. The method relies on lattice reduction: by taking multiple consecutive outputs of a TLCG as coordinates into a high-dimensional space, one can see that they land on hyperplanes in the space. Lattice reduction is the problem of finding short vectors that span those hyperplanes.

Sage [6] is a computer algebra package that includes the LLL lattice reduction algorithm [4]. In the Sage code below, we first define a basis `L` for our lattice. One of the basis vectors is the modulus 2^{32} in the first column, zero elsewhere. The other basis vectors are nth powers of `0x08088405` in the first column and `-1` in the $n+1$st column. We then use LLL to compute a new basis `B` with much smaller entries than `L`.

```
M = 2^32
c = 0x08088405
L = matrix([
    [  M,  0,  0,  0],
    [c^1, -1,  0,  0],
    [c^2,  0, -1,  0],
    [c^3,  0,  0, -1]
])
B = L.LLL()
```

Instead of trying all 2^{32} possible values for `keys[1]`, we can take our existing guesses relating to `keys[1]`, guess the high eight bits of `(keys[1] * 0x1201d271)`, multiply `B` by the vector of most significant bytes, and subtract off the nearest lattice point to get a vector `w` in the new basis.

```
int64_t w[4] = {
    + 109 * msbs[0] -  18 * msbs[1] - 125 * msbs[2] +  74 * msbs[3],
    +  72 * msbs[0] - 145 * msbs[1] -  60 * msbs[2] - 163 * msbs[3],
    - 108 * msbs[0] - 123 * msbs[1] -  19 * msbs[2] + 198 * msbs[3],
    - 319 * msbs[0] + 137 * msbs[1] - 245 * msbs[2] -  85 * msbs[3]};

for (int i = 0; i < 4; ++i) {
    w[i] = -(w[i] & 0xffffffff);
}
```

Next, we switch back to the original basis to get a vector `v`.

```
int64_t v[4] = {
    (  13604679 * w[0] -   563513 * w[1] -  8196160 * w[2] - 6167539 * w[3]) >> 32,
    (   4624483 * w[0] - 16015901 * w[1] - 13783360 * w[2] + 2631745 * w[3]) >> 32,
    (- 18096657 * w[0] -  4513425 * w[1] -    38464 * w[2] - 7189179 * w[3]) >> 32,
    (   8556971 * w[0] - 10689749 * w[1] +  8655040 * w[2] - 2419111 * w[3]) >> 32};
```

With five or more known outputs of the TLCG, the first element of `v` would contain the low 24 bits of `(keys[1] * 0x08088405)`. Given the full value of `(keys[1] * 0x08088405)`, we can multiply by the modular inverse of `0x08088405` to recover the original value of `keys[1]`. However, in this attack, we have guessed only four outputs, so the error vector is too large to get the correct answer every time. The vector `v`, however, always differs from the true value by one of only 35 other vectors, listed below, which we found by iterating over all 2^{32} possible values for `keys[1]`.

```
int64_t ntable[35][4] = {
   {-14363699, -11151615, -7227643, 6235929},
   {-13800186, 4864286, -2714218, 16925678},
   {-8196160, -13783360, -38464, 8655040},
   {-7632647, 2232541, 4474961, 19344789},
   {-6167539, 2631745, -7189179, -2419111},
   {-5604026, 18647646, -2675754, 8270638},
   {-2028621, -16415105, 7150715, 11074151},
   {-1465108, -399204, 11664140, 21763900},
   {-759020, -6527132, -25324300, 14792900},
   {-563513, -16015901, -4513425, -10689749},
   {-195507, 9488769, -20810875, 25482649},
   {0, 0, 0, 0},
   {563513, 16015901, 4513425, 10689749},
   {4138918, -19046850, 14339894, 13493262},
   {5408519, -9158877, -18135121, 17212011},
   {5604026, -18647646, 2675754, -8270638},
   {5972032, 6857024, -13621696, 27901760},
   {6167539, -2631745, 7189179, 2419111},
   {6731052, 13384156, 11702604, 13108860},
   {7437140, 7256228, -25285836, 6137860},
   {8000653, 23272129, -20772411, 16827609},
   {11576058, -11790622, -10945942, 19631122},
   {11771565, -21279391, 9864933, -5851527},
   {12139571, 4225279, -6432517, 30320871},
   {12335078, -5263490, 14378358, 4838222},
   {13041166, -11391418, -22610082, -2132778},
   {13604679, 4624483, -18096657, 8556971},
   {14168192, 20640384, -13583232, 19246720},
   {17743597, -14422367, -3756763, 22050233},
   {19208705, -14023163, -15420903, 286333},
   {19772218, 1992738, -10907478, 10976082},
   {20335731, 18008639, -6394053, 21665831},
   {25376244, -16654908, -8231724, 2705444},
   {25939757, -639007, -3718299, 13395193},
   {26503270, 15376894, 795126, 24084942}};
```

We iterate over these 35 vectors, adding them to `v`, and discard any results whose components are negative, have entries greater than or equal to 2^{24}, or are inconsistent with the carry bits we have guessed. This reduces the total work of stage 4 to roughly 2^{57} encryptions.

5 Differential meet-in-the-middle attack

In the 2001 attack, there are 1.1 trillion key candidates at the end of stage 1 and 17.6 trillion at the end of stage 2. After some experimentation with the GPU farm, it became clear that we could not get the candidates to the GPUs fast enough to keep them occupied. We also wanted to make better use of the information we had to *derive* bits rather than merely guessing them and filtering them.

The basic idea of the DMITM attack is to notice that a difference `delta_m` in the high byte of `keys[1]` gives a difference `crctab[delta_m]` in `keys[2]` that is independent of `keys[2]`. If we guess the stream byte `sx` on the first encryption of a salt byte, we can derive what the stream byte `sy` must have been on the second encryption: `sy` must be `sx` xored with the known `rand` output and the corresponding encrypted byte in the file. Given a stream byte, there are exactly 64 possible values of bits 2 to 15 of `keys[2]` that could have produced it; we build a table of these values and call the index of a value the "prefix". If we guess the prefix for `sx`, the fact that the difference in `keys[2]` must be an entry in `crctab` means that we get, on average, a unique prefix for `sy`.

If we guess the stream byte `sxf0` for the first encryption of salt from file 0, the prefix `pxf0`, and the stream byte `sxf1` for the first encryption of salt from file 1, we can derive three 8-bit differences between pairs of most significant bytes for `keys[1]`.

We guess other bits for the first half of the cipher, compute the three 8-bit differences, and store the guesses for the first half of the cipher in a table. Then we go through the process above for the second half and look up the candidates in the table. Given the guesses for both halves, we can derive bits of `keys[2]` for each encryption pass. We end up producing around 2^{22} candidates and use a few hundred megabytes of memory for the table. The process took about two minutes on a 2017 Macbook Pro.

In the new stage 2, for each of the 2^{22} candidates, we do another DMITM attack. The relevant difference between `keys[2]` values does not depend on the initial value of `keys[2]`, but does depend on the guesses from the previous stage. Running the new stage 2 took about a day on a fast desktop machine with lots of RAM. We wrote batches of key candidates to disk, uploaded them to the machines with the GPUs, and ran the rest of the stages on the GPUs.

The code below refers to "chunks", which are groups of bits all related to a single element of the `keys` array in the order they are needed in the 2001 attack:

chunk	bits
1	0 to 15 of `keys[2]`
2	0 to 7 of `CRC32(keys[0], 0)`
3	23 to 31 of `keys[1] * 0x08088405`
4	16 to 23 of `keys[2]`
5	24 to 32 of `keys[2]`
6	8 to 15 of `CRC32(keys[0], 0)`
7	23 to 31 of `keys[1] * 0xD4652819`

5.1 Code common to both stages

We define `CRYPTCONST` to be `0x08088405` and `CRYPTCONST_POW2`, `CRYPTCONST_POW3`, and `CRYPTCONST_POW4` to be its successive powers.

We split up the `update_keys` function into two parts, `first_half_step` and `second_half_step`. In the `first_half_step` function, we pass in the input byte, the bits we are guessing, and a parameter `extra`, which keeps track of the difference between the state of `keys[1]` at stage $n+1$ and the initial state of `keys[1]` times `0x08088405` to the nth power. We also pass in upper and lower bounds on the unknown low 24 bits of `keys[1]` implied by the carry bits we've guessed. The function updates `keys[0]`, `extra`, and the upper and lower bounds.

```
// Computes one step of the first half of the zip encryption
// Supposing we're on stage n+1,
// x is plaintext[n]
// Set the crc_flag to false for n = 0 and true for n > 0
// k1msb is msb(k10 * CRYPTCONST_POW<n+1>)
// carry is the carry bit for this file and x/y pass
// k0 is crc32(k00, 0) when n=0; gets updated to k0n
// extra is 0 when n=0; gets updated
// upper is the current upper bound on low24(k10*CRYPTCONST_POW<n+1>); may get
// updated lower is the current lower bound on low24(k10*CRYPTCONST_POW<n+1>);
// may get updated
uint8_t first_half_step(uint8_t x, bool crc_flag, uint8_t k1msb, uint8_t carry,
                        uint32_t &k0, uint32_t &extra, uint32_t &upper,
                        uint32_t &lower) {
    if (crc_flag) {
        k0 = crc32(k0, x);
    } else {
        k0 ^= crctab[x];
    }
    extra = (extra + (k0 & 0xff)) * CRYPTCONST + 1;
    uint32_t bound = 0x01000000 - (extra & 0x00ffffff);

    if (carry) {
        lower = bound > lower ? bound : lower;
    } else {
        upper = bound < upper ? bound : upper;
    }

    return k1msb + (extra >> 24) + carry;
}
```

In the `second_half_step` function, we pass in bits 2 to 15 of `keys[2]` from one encryption and the stream byte from another encryption. The function outputs the set of indices `idx` such that `keys[2] ^ crctab[idx]` produces the given stream byte. To find the index, we use a table `crcinvtab` whose entries satisfy

`crcinvtab[(crctab[i] >> 3) & 0xff] == i.`

```
// Finds idxs such that crctab[idx] is the xor of offset and some prefix of
// stream_byte. We expect one on average.
void second_half_step(const uint16_t offset, const uint8_t stream_byte,
                        vector<uint8_t> &idxs) {
    for (uint8_t prefix = 0; prefix < 0x40; ++prefix) {
        uint16_t preimage = preimages[stream_byte][prefix];
        uint16_t xored = offset ^ preimage;
        // For these 8 bits there's one crctab entry that matches them
        uint8_t inv = (xored >> 1) & 0xff;
        uint8_t idx = crcinvtab[inv];
        // Check that the other 6 bits match
        // We expect one prefix on average to work.
        uint16_t match = (crctab[idx] >> 2) & 0x3fff;
        if (match == xored) {
            idxs.push_back(idx);
        }
    }
}
```

5.2 Stage 1

5.2.1 Stage 1a

The `mitm_stage1a` function simply guesses the 0th stream byte, chunk 2, chunk 3, and the carry bits, computes the most significant bytes of `keys[1]` for each encryption, and stores the guess at the 24-bit value formed by xoring the most significant byte from the first encryption with those of the next three.

```
void mitm_stage1a(archive_info &info, vector<vector<stage1a>> &table) {
    // STAGE 1
    //
    // Guess s0, chunk2, chunk3 and carry bits.
    uint8_t xf0 = info.file[0].x[0];
    uint8_t xf1 = info.file[1].x[0];
    uint32_t extra(0);

    for (uint16_t s0 = 0; s0 < 0x100; ++s0) {
        for (uint16_t chunk2 = 0; chunk2 < 0x100; ++chunk2) {
            for (uint16_t chunk3 = 0; chunk3 < 0x100; ++chunk3) {
                for (uint8_t carries = 0; carries < 0x10; ++carries) {
                    uint8_t carryxf0 = carries & 1;
                    uint8_t carryyf0 = (carries >> 1) & 1;
                    uint8_t carryxf1 = (carries >> 2) & 1;
                    uint8_t carryyf1 = (carries >> 3) & 1;
                    uint32_t upper = 0x01000000;  // exclusive
                    uint32_t lower = 0x00000000;  // inclusive

                    uint32_t k0crc = chunk2;
                    uint32_t extra = 0;
                    uint8_t msbxf0 =
                        first_half_step(xf0, false, chunk3, carryxf0, k0crc,
                                        extra, upper, lower);
                    uint8_t yf0 = xf0 ^ s0;
                    k0crc = chunk2;
                    extra = 0;
                    uint8_t msbyf0 =
                        first_half_step(yf0, false, chunk3, carryyf0, k0crc,
                                        extra, upper, lower);
                    if (upper < lower) {
                        continue;
                    }
                    k0crc = chunk2;
                    extra = 0;
                    uint8_t msbxf1 =
                        first_half_step(xf1, false, chunk3, carryxf1, k0crc,
                                        extra, upper, lower);
                    if (upper < lower) {
                        continue;
                    }
                    uint8_t yf1 = xf1 ^ s0;
                    k0crc = chunk2;
                    extra = 0;
                    uint8_t msbyf1 =
                        first_half_step(yf1, false, chunk3, carryyf1, k0crc,
                                        extra, upper, lower);
                    if (upper < lower) {
                        continue;
                    }
                    uint32_t mk = toMapKey(msbxf0, msbyf0, msbxf1, msbyf1);
                    stage1a candidate = {uint8_t(s0), uint8_t(chunk2),
                                         uint8_t(chunk3), carries, msbxf0};
                    table[mk].push_back(candidate);
                }
            }
        }
    }
}
```

5.2.2 Stage 1b

The `mitm_stage1b` function is more complicated. It guesses a stream byte and a prefix for the first encryption of a salt byte from file 0, then computes the set of possible differences in the most significant byte of `keys[1]` that would give the stream byte for the second encryption. If that set is empty, it tries the next guess. Otherwise, it guesses a stream byte for the first encryption of a salt byte from file 1, then computes the set of possible differences for that stream byte. If that set is empty, it tries the next guess. Otherwise, it computes the set of possible differences for the stream byte for the second encryption of the salt byte from file 1. If that set is empty, it tries the next guess.

Given three nonempty sets of differences, it iterates over all possible triples and looks up the set of candidates from stage 1a. It then combines the information from stage 1a with the information from stage 1b to derive, on average, four possible values for the low 24 bits of `keys[2]`. If there are no values of `keys[2]` that work, it goes onto the next guess; otherwise, it outputs the information as a candidate.

```
// info: the info about the archive to attack
// table: the output of mitm_stage1a
// candidates: an empty vector
void mitm_stage1b(const archive_info &info,
                  const vector<vector<stage1a>> &table,
                  vector<stage1_candidate> &candidates) {
    // Second half of MITM for stage 1
    for (uint16_t s1xf0 = 0; s1xf0 < 0x100; ++s1xf0) {
        for (uint8_t prefix = 0; prefix < 0x40; ++prefix) {
            uint16_t pxf0(preimages[s1xf0][prefix]);
            vector<uint8_t> firsts(0);
            uint8_t s1yf0 = s1xf0 ^ info.file[0].x[1] ^ info.file[0].h[1];
            second_half_step(pxf0, s1yf0, firsts);
            if (!firsts.size()) {
                continue;
            }
            for (uint16_t s1xf1 = 0; s1xf1 < 0x100; ++s1xf1) {
                vector<uint8_t> seconds(0);
                second_half_step(pxf0, s1xf1, seconds);
                if (!seconds.size()) {
                    continue;
                }
                vector<uint8_t> thirds(0);
                uint8_t s1yf1 = s1xf1 ^ info.file[1].x[1] ^ info.file[1].h[1];
                second_half_step(pxf0, s1yf1, thirds);
                if (!thirds.size()) {
                    continue;
                }
                for (auto f : firsts) {
```

```
for (auto s : seconds) {
    for (auto t : thirds) {
        uint32_t mapkey(f | (s << 8) | (t << 16));
        for (stage1a candidate : table[mapkey]) {
            stage1_candidate g;
            g.chunk2 = candidate.chunk2;
            g.chunk3 = candidate.chunk3;
            g.cb1 = candidate.cb;
            g.m1 =
                (candidate.msbk11xf0 * 0x01010101) ^ mapkey;

            // Get ~4 possible solutions for lo24(k20) =
            // chunks 1 and 4
            //       A  B  C  D   k20
            // ^  E  F  G  H      crctab[D]
            //    ----------
            //    I  J  K  L      crck20
            // ^  M  N  O  P      crctab[msbk11xf0]
            //    ----------
            //    Q  R  S  T      (pxf0 << 2) matches k21xf0

            // Starting at the bottom, derive 15..2 of KL
            // from 15..2 of ST and OP
            uint16_t crck20 =
                ((pxf0 << 2) ^
                 crctab[candidate.msbk11xf0]) &
                0xfffc;

            // Now starting at the top, iterate over 64
            // possibilities for 15..2 of CD
            for (uint8_t i = 0; i < 64; ++i) {
                uint32_t maybek20 =
                    (preimages[candidate.s0][i] << 2);
                // and 4 possibilities for low two bits of D
                for (uint8_t lo = 0; lo < 4; ++lo) {
                    // CD
                    maybek20 = (maybek20 & 0xfffc) | lo;
                    // L' = C ^ H
                    uint8_t match =
                        (maybek20 >> 8) ^
                        crctab[maybek20 & 0xff];
                    // If upper six bits of L == upper six
                    // of L' then we have a candidate
                    if ((match & 0xfc) == (crck20 & 0xfc)) {
                        // KL ^ GH = BC.  (B = BC >> 8) &
                        // 0xff.
                        uint8_t b =
                            ((crck20 ^
                              crctab[maybek20 & 0xff]) >>
                             8) &
                            0xff;

                        // BCD = (B << 16) | CD
                        g.maybek20[g.k20_count] =
                            (b << 16) | maybek20;
                        g.k20_count += 1;
                    }
                }
            }

            if (0 == g.k20_count) {
                continue;
            }

            candidates.push_back(g);
        }
```

```
                        }
                    }
                }
            }
        }
    }
}
```

5.3 Stage 2

5.3.1 Stage 2a

The function `mitm_stage2a` is similar to `mitm_stage1a`; given a stage 1 candidate, it guesses chunk 6, chunk 7, and some carry bits, then computes the three 8-bit differences and stores the guess in the table.

```
void mitm_stage2a(archive_info& info, stage1_candidate& c1,
                  vector<vector<stage2a>>& table) {
    uint8_t cb1 = c1.cb1;
    uint8_t carryx0f0 = cb1 & 1;
    uint8_t carryy0f0 = (cb1 >> 1) & 1;
    uint8_t carryx0f1 = (cb1 >> 2) & 1;
    uint8_t carryy0f1 = (cb1 >> 3) & 1;
    // We always have at least one k20 candidate.
    // All candidates for k20 give the same s0 byte,
    // and all single-step updates of a candidate
    // give the same four s1 bytes
    uint8_t s0 = get_s0(c1.maybek20[0] & 0xffff);
    uint32_t s1xf0 = get_s0(crc32(c1.maybek20[0], c1.m1 >> 24));
    uint32_t s1xf1 = get_s0(crc32(c1.maybek20[0], (c1.m1 >> 8) & 0xff));

    for (uint16_t chunk6 = 0; chunk6 < 0x100; ++chunk6) {
        for (uint16_t chunk7 = 0; chunk7 < 0x100; ++chunk7) {
            for (uint8_t cb2 = 0; cb2 < 0x10; ++cb2) {
                auto flag = false;
                uint8_t carryx1f0 = cb2 & 1;
                uint8_t carryy1f0 = (cb2 >> 1) & 1;
                uint8_t carryx1f1 = (cb2 >> 2) & 1;
                uint8_t carryy1f1 = (cb2 >> 3) & 1;
                // bounds on low 24 bits of k10 * CRYPTCONST
                uint32_t upper1 = 0x01000000;
                uint32_t lower1 = 0x00000000;
                // bounds on low 24 bits of k10 * CRYPTCONST_POW2
                uint32_t upper2 = 0x01000000;
                uint32_t lower2 = 0x00000000;

                // Compute msbk12s
                uint32_t k0crc = c1.chunk2 | (chunk6 << 8);
                uint32_t extra = 0;
                first_half_step(info.file[0].x[0], false, c1.chunk3, carryx0f0,
                                k0crc, extra, upper1, lower1);
                uint32_t msbxf0 =
                    first_half_step(info.file[0].x[1], true, chunk7, carryx1f0,
                                    k0crc, extra, upper2, lower2);
                k0crc = c1.chunk2 | (chunk6 << 8);
                extra = 0;
                first_half_step(info.file[0].x[0] ^ s0, false, c1.chunk3,
                                carryy0f0, k0crc, extra, upper1, lower1);
                uint32_t msbyf0 =
                    first_half_step(info.file[0].x[1] ^ s1xf0, true, chunk7,
                                    carryy1f0, k0crc, extra, upper2, lower2);
                if (upper2 < lower2) {
                    continue;
                }
                k0crc = c1.chunk2 | (chunk6 << 8);
                extra = 0;
                first_half_step(info.file[1].x[0], false, c1.chunk3, carryx0f1,
                                k0crc, extra, upper1, lower1);
                uint32_t msbxf1 =
                    first_half_step(info.file[1].x[1], true, chunk7, carryx1f1,
                                    k0crc, extra, upper2, lower2);
                if (upper2 < lower2) {
                    continue;
                }
                k0crc = c1.chunk2 | (chunk6 << 8);
                extra = 0;
                first_half_step(info.file[1].x[0] ^ s0, false, c1.chunk3,
                                carryy0f1, k0crc, extra, upper1, lower1);
                uint32_t msbyf1 =
                    first_half_step(info.file[1].x[1] ^ s1xf1, true, chunk7,
                                    carryy1f1, k0crc, extra, upper2, lower2);
                if (upper2 < lower2) {
                    continue;
                }
                uint32_t mk = toMapKey(msbxf0, msbyf0, msbxf1, msbyf1);
                stage2a s2a;
                s2a.chunk6 = chunk6;
                s2a.chunk7 = chunk7;
                s2a.cb2 = cb2;
                s2a.msbk12xf0 = msbxf0;
                table[mk].push_back(s2a);
            }
        }
    }
}
```

5.3.2 Stage 2b

Let k1n and k2n denote the states of keys[1] and keys[2], respectively, after processing n input bytes, msb indicate the most significant byte, and x and y indicate the first and second encryption of the salt byte. If we xor two k22 values together, the result is independent of k20:

```
k22x ^ k22y =   CRC32(k21x, msbk12x) ^ CRC32(k21y, msbk12y)
            =   CRC32(CRC32(k20, msbk11x), msbk12x) ^ CRC32(CRC32(k20, msbk11x), msbk12y)
            =   CRC32(CRC32(k20, msbk11x) ^ CRC32(k20, msbk11y), msbk12x ^ msbk12y)
            =   CRC32(CRC32(0, msbk11x ^ msbk11y), msbk12x ^ msbk12y)
            =   CRC32(crctab[msbk11x ^ msbk11y], msbk12x ^ msbk12y)
            =   (cy >> 8) ^ crctab[(cy & 0xff) ^ msbk12x ^ msbk12y]
```

where cy = crctab[msbk11x ^ msbk11y]. This value depends on the candidate from stage 1, but not on anything else.

The function mitm_stage2b begins by computing the cy value for three keys[2] differences, extracts some relevant bits, then follows the same logic as mitm_stage1b with the exception that it also iterates over the list of keys[2] values in each candidate. Once it has the bits from both halves of the cipher, it derives more bits of keys[2], but the computation is far simpler.

```
void mitm_stage2b(const mitm::archive_info& info,
                  const mitm_stage1::stage1_candidate& c1,
                  const std::vector<std::vector<stage2a>>& table,
                  stage2_candidate* candidates, /* output */
                  const size_t array_size,
                  size_t& stage2_candidate_count, /* output */) {
    uint32_t mk1 = c1.m1 ^ ((c1.m1 >> 24) * 0x01010101);
    // Compute the constants from stage1.
    uint32_t cyf0 = crc32tab[mk1 & 0xff];
    uint32_t cxf1 = crc32tab[(mk1 >> 8) & 0xff];
    uint32_t cyf1 = crc32tab[(mk1 >> 16) & 0xff];
```

```
uint32_t cyf0p = (cyf0 >> 10) & 0x3fff;
uint32_t cxf1p = (cxf1 >> 10) & 0x3fff;
uint32_t cyf1p = (cyf1 >> 10) & 0x3fff;
uint32_t cyf0l = cyf0 & 0xff;
uint32_t cxf1l = cxf1 & 0xff;
uint32_t cyf1l = cyf1 & 0xff;
for (uint16_t s2xf0 = 0; s2xf0 < 0x100; ++s2xf0) {
    uint8_t s2yf0 = s2xf0 ^ info.file[0].x[2] ^ info.file[0].h[2];
    for (uint8_t prefix = 0; prefix < 0x40; ++prefix) {
        uint16_t pxf0(preimages[s2xf0][prefix]);

        vector<uint8_t> firsts(0);
        second_half_step(pxf0 ^ cyf0p, s2yf0, firsts);
        if (!firsts.size()) {
            continue;
        }
        for (uint16_t s2xf1 = 0; s2xf1 < 0x100; ++s2xf1) {
            vector<uint8_t> seconds(0);
            second_half_step(pxf0 ^ cxf1p, s2xf1, seconds);
            if (!seconds.size()) {
                continue;
            }
            vector<uint8_t> thirds(0);
            uint8_t s2yf1 = s2xf1 ^ info.file[1].x[2] ^ info.file[1].h[2];
            second_half_step(pxf0 ^ cyf1p, s2yf1, thirds);
            if (!thirds.size()) {
                continue;
            }
            for (auto f : firsts) {
                for (auto s : seconds) {
                    for (auto t : thirds) {
                        uint32_t mapkey((f ^ cyf0l) | ((s ^ cxf1l) << 8) |
                                        ((t ^ cyf1l) << 16));
                        for (auto c2 : table[mapkey]) {
                            stage2_candidate g;
                            for (int i = 0; i < c1.k20_count; ++i) {
                                uint32_t k20 = c1.maybek20[i];
                                uint32_t k21xf0 = crc32(k20, c1.m1 >> 24);
                                uint32_t k22xf0 =
                                    crc32(k21xf0, c2.msbk12xf0);
                                if ((pxf0 & 0x3f) ==
                                    ((k22xf0 >> 2) & 0x3f)) {
                                    // Find last byte of k20, then store it
                                    uint8_t hi_byte =
                                        (pxf0 >> 6) ^
                                        ((k22xf0 >> 8) & 0xff);
                                    g.maybek20[g.k20_count] =
                                        k20 | (hi_byte << 24);
                                    g.k20_count += 1;
                                }
                            }

                            if (0 == g.k20_count) {
                                continue;
                            }

                            g.chunk2 = c1.chunk2;
                            g.chunk3 = c1.chunk3;
                            g.chunk6 = c2.chunk6;
                            g.chunk7 = c2.chunk7;
                            g.cb = (c1.cb1 << 4) | c2.cb2;
                            g.m1 = c1.m1;
                            g.m2 = mapkey ^ (c2.msbk12xf0 * 0x01010101);

                            candidates[stage2_candidate_count] = g;
                            ++stage2_candidate_count;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

6 Conclusion

The Zip cipher is weak, but not so weak as to be trivial to crack. The 2001 attack works because of flaws that Info-Zip introduced, and then only when there are enough files in the archive; reducing the number of files increases the complexity exponentially. We were able to use some clever tricks to avoid some of the problems associated with that exponential growth, but we still had to use a GPU farm to break it. The amount of computation required is a feat that would have been inconceivable twenty years ago, except perhaps for nation states. An encrypted Info-Zip archive with only one file in it is still out of reach. It will be interesting to see if the cipher is still in use in another twenty years.

References

[1] Biham, Eli and Paul Kocher. “A Known Plaintext Attack on the PKZIP Stream Cipher”. *Fast Software Encryption 2*, Proceedings of the Leuven Workshop, LNCS 1008, December 1994. Also available at http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1994/CS/CS0842.pdf.

[2] Frieze, Alan M., John Hastad, Ravi Kannan, Jeffrey C. Lagarias, and Adi Shamir. “Reconstructing Truncated Integer Variables Satisfying Linear Congruences”. *Siam J. Comput.* 17 (2): 262–280. 1988. Also available at https://www.math.cmu.edu/~af1p/Texfiles/RECONTRUNC.pdf.

[3] Info-Zip. “Info-Zip Home Page”. http://infozip.sourceforge.net/.

[4] Lenstra, A. K., H. W. Lenstra, Jr., and L. Lovász. “Factoring polynomials with rational coefficients". Mathematische Annalen. 261 (4): 515–534. 1982. Also available at https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.310.318.

[5] PKWARE. “Application Note Archives”. https://support.pkware.com/home/pkzip/developer-tools/appnote/application-note-archives.

[6] SageMath Project. “SageMath - Open-Source Mathematical Software System”. https://www.sagemath.org/.

[7] Stay, Michael. “ZIP Attacks with Reduced Known Plaintext”. *Fast Software Encryption*, LNCS 2355, 125–134. Springer-Verlag, 2002. Also available at http://www.math.ucr.edu/~mike/zipattacks.pdf.