ZIP Attacks with Reduced Known Plaintext

Michael Stay

AccessData Corporation
2500 N. University Ave. Ste. 200
Provo, UT 84606
staym@accessdata.com

Abstract. Biham and Kocher demonstrated that the PKZIP stream cipher was weak and presented an attack requiring thirteen bytes of plaintext. The *deflate* algorithm "zippers" now use to compress the plaintext before encryption makes it difficult to get known plaintext. We consider the problem of reducing the amount of known plaintext by finding other ways to filter key guesses. In most cases we can reduce the amount of known plaintext from the archived file to two or three bytes, depending on the zipper used and the number of files in the archive. For the most popular zippers on the Internet, there is a fast attack that does not require any information about the files in the archive; instead, it gets doubly-encrypted plaintext by exploiting a weakness in the pseudorandom-number generator.

1 Introduction

PKZIP is a compression / archival program created by Phil Katz. Katz had the foresight to document his file format completely in the file APPNOTE.TXT, distributed with every copy of PKZIP; there are now literally hundreds of "zipper" programs available, and the ZIP file format has become a *de facto* standard on the Internet.

In [BK94] Biham and Kocher demonstrated that the PKZIP stream cipher was weak and presented an attack requiring thirteen bytes of plaintext. Eight bytes of the plaintext must be contiguous, and all of the bytes must be the text that was encrypted, which is usually compressed data. [K92] shows that the compression method used at the time, *implode*, produces many predictable bytes suitable for mounting the attack.

Most zippers available today implement only one of the compression methods defined in APPNOTE.TXT, called *deflate*. *Deflate* uses Huffman coding followed by a variant of Lempel-Ziv. Once the dictionary reaches a certain size, the process starts over. Since the Huffman codes for any of the data depend on a great deal of surrounding data, one is forced to guess the plaintext unless one has the original data. The difficulty of getting known plaintext was one reason Phil Zimmerman decided to use *deflate* in PGP [PGP98]. Practically speaking, if one has enough of the original file to get the thirteen bytes of plaintext required for the attack in [BK94], one has enough to break the encryption almost instantly.

Without the original file, all is not lost; we have the file's type as indicated by its extension, and we have its size. The ZIP file format requires at least one byte of known plaintext for filtering incorrect passwords. Most zippers also encrypt output from a pseudorandom number generator that is vulnerable to attack.

It is the author's opinion that the only reason the PKZIP cipher has held up so well in light of [BK94] is the high entropy of the data produced by the *deflate* algorithm and the related difficulty of getting enough plaintext. This paper treats the question of how far we can reduce the plaintext requirement and still break the cipher with a practical amount of work.

1.1 The PKZIP Stream Cipher

The PKZIP stream cipher was designed by Roger Schaffely and is fully described in the file APPNOTE.TXT found in most PKZIP distributions. The internal state of the cipher consists of three 32-bit words: $key0$, $key1$, and $key2$. These values are initialized to 0x12345678, 0x23456789, and 0x34567890, respectively. The internal state is updated by mixing in the next plaintext byte. The first and third words are updated using the linear feedback shift register known as CRC-32; the second word is updated using a truncated linear congruential generator. The output byte is the result of a truncated pseudo-squaring operation. (See Figure 1.)

```
unsigned char PKZIP_stream_byte (unsigned char pt)
{
   unsigned short temp;
   key0 = crc32 ( key0, pt );
   key1 = ( key1 + LSB( key0 ) ) * 0x08088405 + 1;
   key2 = crc32 ( key2, MSB( key1 ) );
   temp = key2 | 3;
   return LSB( ( temp * (temp ^ 1) ) >> 8);
}
```

where 'unsigned char' is an 8-bit integer; 'unsigned short' is a 16-bit integer; | is bitwise OR; ^ is XOR; >> is right shift; 'LSB' and 'MSB' are the least-significant and most-significant bytes, respectively; and '0x' is a prefix indicating hexadecimal.

For the purposes of this paper, we define $crc32()$ to be

```
unsigned long crc32( unsigned long crc, unsigned char b)
{
   return ( (crc >> 8) ^ crctab [ LSB( crc ) ^ b ] );
}
```

where 'unsigned long' is a 32-bit integer.

The old LFSR state is shifted right eight bits and XORed with the 32-bit entry of a byte-indexed table to produce the new state. The index is the low byte of the old state XORed with b. The function is linear; that is,

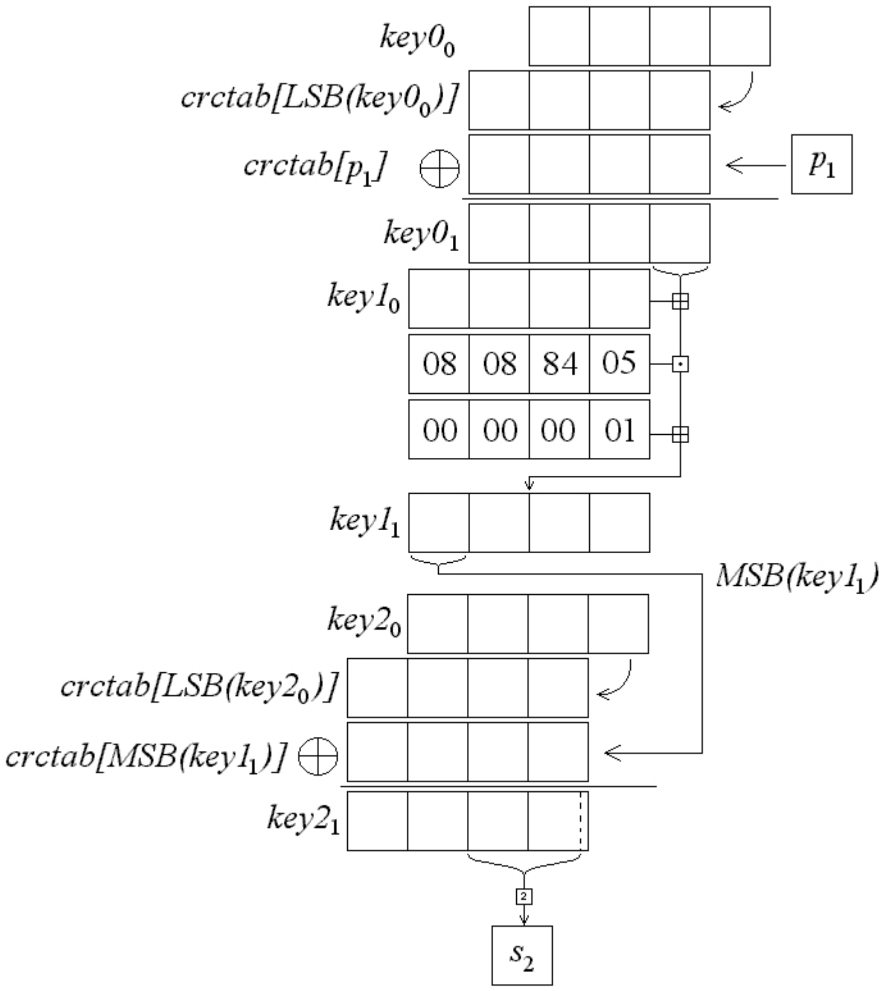

Fig. 1. The PKZIP stream cipher. CRC-32, TLCG, CRC-32, truncated pseudo-square.

```
    crctab[ x ^ y ] = crctab[ x ] ^ crctab[ y ].
```

The cipher is keyed by encrypting the user's password and throwing away the corresponding stream bytes. The stream bytes produced after this point are XORed with the plaintext to produce the ciphertext.

The crux of all of our attacks is the fact that there is almost no diffusion in the internal state. Of the ninety-six bits of internal state, eight bits of *key*0 affect *key*1; eight bits of *key*1 affect *key*2; and fourteen bits of *key*2 affect the output stream byte.

1.2 Encrypted File Format

Zippers must prepend twelve bytes to the beginning of the file to be encrypted. The ZIP file format specifies that the first eleven should be random and that the last should be the low byte of the archived file's CRC. The entire CRC is stored in plaintext, and this byte serves as a password filter. Some zippers, like InfoZIP [IZ] and WinZip [WZ], store ten random bytes and the low two bytes of the CRC.

We assume that *deflate* was the algorithm used to compress the underlying data. The author did a crude statistical test on a few hundred files of varying types and sizes and found that given the file type, as indicated by its extension, and its size, one can guess about the first two and a half bytes of the compressed file. Since the checksum bytes are at the very end of the prepended header, we can use them to augment the plaintext from the file in mounting our attacks.

2 Biham and Kocher's Attack

For completeness, we review [BK94]'s results. We begin with some terminology. Bits are numbered from right to left: bit 0 is the ones' place, bit 1 is the twos' place, bit 2 is the fours' place, etc. Let p_i be the ith known-plaintext byte, $i = 1$, 2, 3, ... Let s_i be the ith stream byte. Let $key0_i$, $key1_i$, and $key2_i$ be the value of $key0$, $key1$, and $key2$ after processing p_i. Note that s_1 is a function of the random header and the password; it is independent of the plaintext. In general, bits 2 through 15 of $key2_i$ determine s_{i+1}.

Their attack proceeds as follows:

- XOR ciphertext and known plaintext to get known stream bytes s_1 through s_{13}.
- Guess 22 bits of $key2_{13}$.
- This guess combined with s_{13} is enough to fill in eight more bits of $key2_{13}$, for a total of thirty. s_{12} provides enough information to derive 30 bits of $key2_{12}$ and the most significant byte of $key1_{13}$. In general, each stream byte s_i allows us to calculate thirty bits of $key2_{i-1}$ and the most significant byte of $key1_i$.
- We continue using stream bytes to make a list of the most significant bytes of $key1_{13}$ through $key1_8$.
- For each list, we find 2^{16} possibilities for the low 24 bits of $key1_{13}$ through $key1_9$ by calculating the low byte of $(key1_i + LSB(key0_{i+1}))$ such that we get the right high byte of $key1_{i+1}$.
- From each of the 2^{16} lists of complete $key1$'s, derive the low bytes of $key0_{13}$ through $key0_{10}$.
- Once we have the low bytes of $key0_{10}$, $key0_{11}$, $key0_{12}$, and $key0_{13}$, we can use our knowledge of the plaintext bytes to invert the CRC function, since it's linear, and find the complete internal state at one point along the encryption.
- Once we have the complete internal state, we can decrypt backwards as far as we want; we decrypt the ciphertext corresponding to p_1 through p_5 and filter out wrong keys.

We can break a file with work equivalent to encrypting around 2^{38} bytes and negligible memory. We need a total of thirteen bytes of known plaintext: eight for the attack, and five to filter the 2^{38} keys that remain. This is an upper bound; each additional byte of plaintext eliminates approximately one list (see [BK94], fig. 1).

2.1 Minor Improvement in the Amount of Plaintext Required

[BK94] throws away six bits in $key1_7$. By using them, we can reduce the plaintext requirement to twelve bytes at the cost of increasing the work factor by four.

2.2 More Files in the Archive

If we have more than one file in the archive, we can make the reasonable assumption that they were encrypted with the same password. Zippers encrypt at least one check byte into every encrypted file to verify that the user entered the correct password. Once we have the complete internal state of the cipher, we can run it backwards to the beginning of the file and read out $key0$, $key1$, and $key2$. Since this state is the same at the beginning of each file (it only depends on the password), we can decrypt the check byte in each file and use it to filter with instead of known plaintext from a single file. This also works if the files are in different archives, but have the same password.

If the file was created in a zipper with two checksum bytes, we can break the file with work equivalent to encrypting $11 * 2^{40} \approx 2^{43}$ bytes. We need two checksum bytes followed by only four more known plaintext bytes in one file, and three other files in the archive (six check bytes) to filter the 2^{40} possible keys. The factor of eleven in the estimate above is due to the fact that to decrypt the checksum byte, we must decrypt the first eleven bytes of the random header.

If there is only one checksum byte per archived file, we can break the cipher with the nearly the same amount of work, but we need seven files in the archive and five bytes of known plaintext in addition to the checksum byte in the first file.

3 Divide and Conquer

The limited diffusion of the internal state prompts us to ask how much of the state we need to guess to process one byte. If it is small enough, we can guess it and filter out keys that won't work with our known stream bytes, then proceed to the next part.

It turns out that we can get by with as few as 23 bits (See Figure 2.) Note that we don't need to guess 16 bits of $key0_0$ to calculate the low byte of $key0_1$: if we distribute the XOR in the definition of $crc32()$, we see that we only need to guess 8 bits of $crc32(key0_0, 0)$:

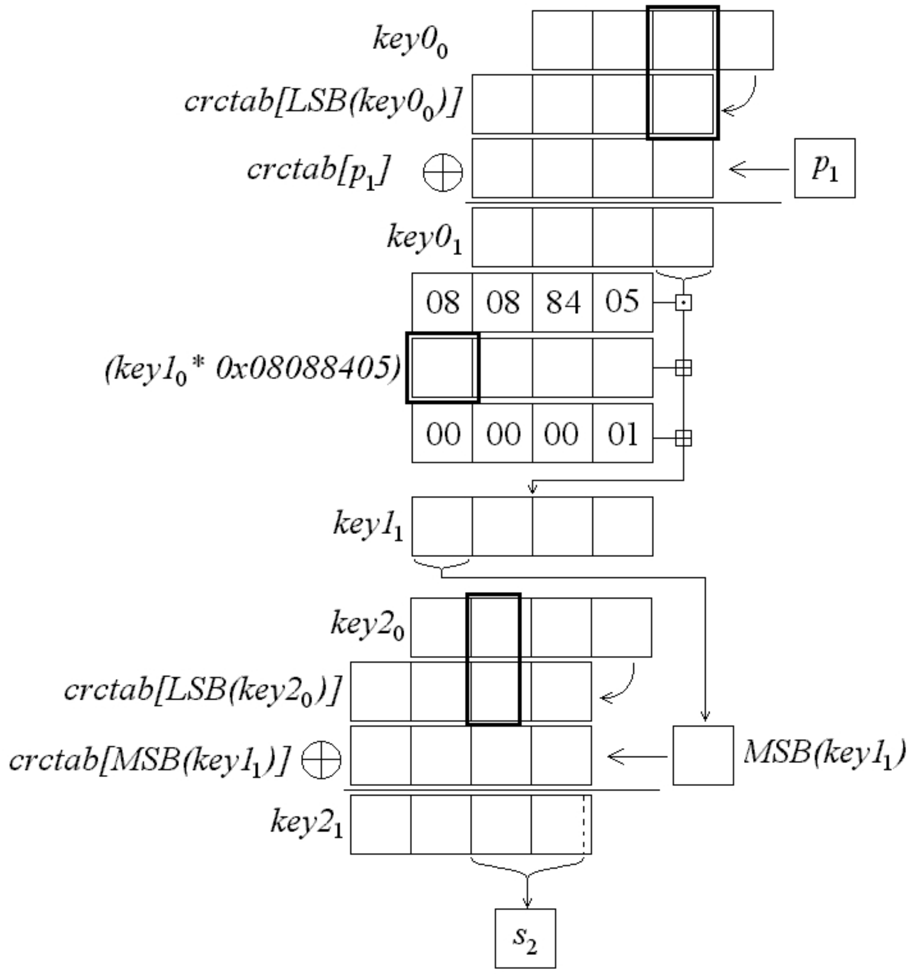

Fig. 2. The twenty-three bits involved in generating a stream byte.

```
LSB(key0$_1$) = LSB( crc32( key0$_0$, p$_1$ ) )
            = LSB( crc32( key0$_0$, 0 ) ) XOR LSB( crctab[ p$_1$ ] ).
```

Now we distribute the multiplication across the addition in the next step:

```
MSB(key1$_1$) =  MSB( ( key1$_0$ + LSB(key0$_1$) ) * 0x08088405 + 1 )
(A)         =  MSB( LSB( key0$_1$ ) * 0x08088405 ) +
(B)            MSB( key1$_0$ * 0x08088405 ) + possible carry bit.
```

We separate the equation into parts we know (A) and parts we need to guess (B), and find we need to guess nine bits, including a possible carry bit. Note that since we know the low bits of $(LSB(key0_1) * 0x08088405)$, the carry bit will usually give us more than one bit of information in the form of an upper or lower bound on the rest of $key1_0 * 0x08088405)$ that we haven't guessed yet.

Given a stream byte s_{i+1}, we can find sixty four values for bits 2..15 of $key2_i$. It's easy to see why: fourteen bits of $key2_i$ produce eight bits of s_{i+1}, so there are six left over. We can create a table of 256 x 64 bytes such that given s_{i+1} and bits 10..15 of $key2_i$, we can look up bits 2..9 of $key2_i$. We call this the preimage table.

We guess bits 10..15 of $crc32(key2_0, 0)$ and use s_2, the preimage table, and $crctab[MSB(key1_1)]$ to find bits 2..9 of $crc32(key2_0, 0)$. We end up with 2^{23} key guesses.

To find the next part of the internal state, we have to guess about the same amount. This guess is not illustrated, but we basically guess about eight more bits of information in each of the three keys. The only complicated part is separating what we know about $key1$ from what we don't.

We guess bits 8..15 of $crc32(key0_0, 0)$ directly; the next guess involving $key1$ is a little more complicated:

```
MSB(key1₂) =  MSB( (key1₁ + LSB(key0₂)) * 0x08088405 + 1 )

           =  MSB( LSB(key0₂) * 0x08088405 ) +
              MSB( key1₁ * 0x08088405 ) + possible carry bit

           =  MSB( LSB(key0₂) * 0x08088405 ) +
              MSB( (key1₀ + LSB(key0₁)) * 0xD4652819 ) +
              possible carry bit

(A)        =  MSB( LSB(key0₂) * 0x08088405 ) +
(A)           MSB( LSB(key0₁) * 0xD4652819 ) +
(B)           MSB( key1₀ * 0xD4652819 ) + possible carry bit.
```

Again, (A) is known and we have to guess (B) nine bits, including a possible carry bit. The carry bit establishes an upper or lower bound on $(key1_0 * 0xD4652819)$. We end this filter by guessing bits 16..23 and bits 0..1 of $crc32(key2_0, 0)$ and calculating a stream byte. We have guessed 27 more bits, but the output byte has to match s_3, so we expect $2^{23+27-8} = 2^{42}$ key guesses to pass this filter.

At this point, we have guessed 24 bits of $crc32(key2_0, 0)$ and we know s_1. From this we can calculate, on average, one full value of $key2_0$. There are also only around 2^{13} possibilities for $key1_0$ due to the restrictions from the carry bits. So the third stage consists of guessing bits 16..23 of $crc32(key0_0, 0)$ and running through the 2^{13} possible values for $key1_0$. We expect $2^{42+13+8-8} = 2^{55}$ key pieces to pass this filter.

Finally, we guess the last eight bits of $key0_0$ and we have a complete internal state. We will have 2^{63} complete keys to filter with other bytes, whether they are in the archived file or in checksum bytes in other files. The cost is approximately the same as encrypting 2^{63} bytes under the stream cipher. The plaintext requirement is four bytes total; at least one of these may come from the file's own check byte(s).

This is 128 times faster than guessing three stream bytes and using [BK94].

4 Parallel Divide and Conquer Attack

InfoZIP is a cross-platform freeware zipper distribution. Because the C source code is readily available and is free, it forms the basis of most non-PKZIP zippers, including the very popular WinZip and NetZip. According to CNET's Download.com [DL], WinZip and NetZip constitute over 96% of the total archiver downloads.

APPNOTE.TXT does not specify how to generate the prepended random bytes; it only says that they are used to scramble the internal state of the cipher and are discarded after decryption. InfoZIP implements it as follows:

1. `srand( time(NULL) ^ getpid() )`
2. For each file in the order they are stored,
3. Generate ten random bytes by calling *rand()* ten times and discarding all but the high eight bits of each return value.
4. Initialize the cipher with the password.
5. Encrypt the ten random bytes.
6. Append the low two bytes of the checksum.
7. Reinitialize the cipher with the password.
8. Encrypt the twelve-byte header and the compressed file.

Note that the random bytes were encrypted *twice*: once in step 5, and again in step 8.

rand() is usually implemented as a truncated linear congruential generator. WinZip and NetZip use Microsoft Visual C++'s implementation, which has a 31-bit seed:

```
unsigned long seed;
void srand( unsigned long s ) { seed = s; }
unsigned short rand()
{
   seed = 0x343FD * seed + 0x269EC3;
   return ( ( seed >> 16) & 0x7FFF );
}
```

Let $r_{i,j}$ be the jth random byte in the ith archived file; $i, j = 1, 2, 3, ...$ Note that the internal state of the cipher is the same both times $r_{i,1}$ is encrypted. Since XOR is its own inverse, $r_{i,1}$ is *decrypted* for all i. Also, every $r_{i,1}$ reveals the high eight bits of the internal state of the random number generator.

Since *rand*() is linear, we can compute two new constants for a generator such that it outputs every tenth output of the original. We know the upper eight bits of the generator, so we guess the low 23 bits and start generating every tenth output and comparing them to the revealed bytes. Five archived files suffice to determine uniquely the seed that was used in the random number generator, and therefore every $r_{i,j}$.

Let us emphasize that we do *not* have known plaintext at this point, in the sense that [BK94] requires. The random bytes were encrypted twice, so we do not know the actual output of the stream cipher during the first and second encryption. What we *can* derive is the XOR of these stream bytes.

4.1 The Attack

We can adapt the divide-and-conquer algorithm from section 3 to use this information. Once we know the "random" headers, we can exploit the fact that the internal state was the same at the beginning of each embedded file and filter guesses with multiple known plaintext bytes in parallel, instead of being restricted to one byte as in section 3.

Let $s_{i,j,k}$ be the jth stream byte of the kth encryption of the bytes in the ith archived file; i, j = 1, 2, 3, ...; k = 1, 2. We guess the same 23 bits as in section 3, but since we don't know the actual value of $s_{1,2,1}$, we have to guess it, too. It is equivalent, and more convenient, to guess bits 2..9 of $crc32(key2_0, 0)$. Now we have a prediction for $s_{1,2,1}$, and can derive $s_{1,2,2}$. We don't have any information at all about $s_{i,1,1}$, since it's the same as $s_{i,1,2}$ and cancels out. We guess it, and check to see that the second encryption spits out $s_{1,2,2}$. We have to guess a carry bit for the second encryption, too, so of the $2^{23+8+8+1} = 2^{40}$ key guesses, we expect $2^{40-8} = 2^{32}$ key pieces to pass this filter on this file.

We want to filter out all but the correct guess at this stage; fortunately, we know that the state we are trying to guess was the same at the start of each encryption. We have an eight-bit value to filter with in each file,
$s_{i,2,1}$ XOR $s_{i,2,2}$, but we also guess two carry bits, so with four more files in the archive, we can reduce the number of false positives to around $2^{32-6*4} = 256$. Note that we now have *ten* carry bits putting restrictions on $key1_0$ instead of just one.

We continue to the next byte of each file. This time we guess the same 26 bits as in section 3 plus two carry bits, one for each encryption. With five files, we have 30 bits to filter with. We expect that $2^{8+26+2-30} = 2^6 = 64$ key guesses survive the second stage. Total work for this stage is $2^{8+26+2} = 2^{36}$ byte encryptions.

At this point we can derive $key2_0$ as before. Due to all the carry bit restrictions, we only have on the order of 2^8 possible $key1_0$'s. We guess eight more bits of $crc32(key1_0, 0)$ and run through all the remaining $key1_0$'s. Since we aren't guessing carry bits any more, we have 40 bits to filter with. $2^{6+16-40} < 1$, and we expect that only the correct guess survives. Finally, we guess the last eight bits of $crc32(key1_0, 0)$ and only the correct guess survives.

Experimentally, we have found that a key guess passes the second stage only if our guess for $s_{i,1,1}$ is correct. This usually occurs about one quarter of the way through the first 40-bit keyspace. After that, we only try one value for $s_{i,1,1}$ instead of 256 and the rest of the attack takes at most a few minutes.

The work done in the first stage dwarfs the rest of the work needed. The total work is therefore about the same as encrypting 2^{39} bytes. We assume that there are five files in the archive that were encrypted consecutively as described above. Decrypting a file created with this kind of weak PRNG usually takes under two hours on a 500 MHz Pentium II. One can then take the three keys and use [BK94]'s second algorithm to derive a password, if one desires, although the three keys suffice to decrypt the files.

Table 1. PKZip Attack Complexity. Files are assumed to have been archived with two checksum bytes.

Attack	Archived files	Plaintext bytes	Complexity
BK94	1	13	2^{38}
BK94 (tradeoff)	1	12	2^{40}
BK94 (tradeoff)	4	6	$11 * 2^{40}$
Divide and conquer	1	2	2^{63}
Parallel divide and conquer (WinZip)	5	0	2^{39}

5 Conclusion

The PKZIP stream cipher is very weak. The *deflate* algorithm makes it harder to get plaintext, but in most cases we can reduce the plaintext requirement to the point where one can guess enough plaintext based on file type and size alone. The most popular zippers on the internet are also susceptible to an attack that runs in two hours on a single PC based on known plaintext provided by the application and independent of the archived files themselves.

References

[BK94] 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.

[DL] `http://download.cnet.com/downloads/0,10151,0-10097-106-0-1-5,00.html?tag=st.dl.10097_106_1.lst.lst&`

[IZ] `ftp://ftp.freesoftware.com/pub/infozip/`

[K92] Kocher, Paul. *ZIPCRACK 2.00 Documentation.* 1992. `http://www.bokler.com/bokler/zipcrack.txt`

[PKZ] `http://www.pkware.com`

[PGP98] *User's Guide, Version 6.0.* Network Associates, Inc., 1998. p.145. `http://www.nai.com`

[WZ] `http://www.winzip.com`