A Known Plaintext Attack on the PKZIP Stream Cipher

Eli Biham* Paul C. Kocher**

Abstract. The PKZIP program is one of the more widely used archive/compression programs on personal computers. It also has many compatible variants on other computers, and is used by most BBS's and ftp sites to compress their archives. PKZIP provides a stream cipher which allows users to scramble files with variable length keys (passwords).
In this paper we describe a known plaintext attack on this cipher, which can find the internal representation of the key within a few hours on a personal computer using a few hundred bytes of known plaintext. In many cases, the actual user keys can also be found from the internal representation. We conclude that the PKZIP cipher is weak, and should not be used to protect valuable data.

1 Introduction

The PKZIP program is one of the more widely used archive/compression programs on personal computers. It also has many compatible variants on other computers (such as Infozip's zip/unzip), and is used by most BBS's and ftp sites to compress their archives. PKZIP provides a stream cipher which allows users to scramble the archived files under variable length keys (passwords). This stream cipher was designed by Roger Schlafly.

In this paper we describe a known plaintext attack on the PKZIP stream cipher which takes a few hours on a personal computer and requires about 13–40 (compressed) known plaintext bytes, or the first 30–200 uncompressed bytes, when the file is compressed. The attack primarily finds the 96-bit internal representation of the key, which suffices to decrypt the whole file and any other file encrypted under the same key. Later, the original key can be constructed. This attack was used to find the key of the PKZIP contest.

The analysis in this paper holds to both versions of PKZIP: version 1.10 and version 2.04g. The ciphers used in the two versions differ in minor details, which does not affect the analysis.

The structure of this paper is as follows: Section 2 describes PKZIP and the PKZIP stream cipher. The attack is described in Section 3, and a summary of the results is given in Section 4.

* Computer Science Department, Technion - Israel Institute of Technology, Haifa 32000, Israel
** Independent cryptographic consultant, 7700 N.W. Ridgewood Dr., Corvallis, OR 97330, USA

2 The PKZIP Stream Cipher

PKZIP manages a ZIP file[1] which is an archive containing many files in a compressed form, along with file headers describing (for each file) the file name, the compression method, whether the file is encrypted, the CRC-32 value, the original and compressed sizes of the file, and other auxiliary information.

The files are kept in the zip-file in the shortest form possible of several compression methods. In case that the compression methods do not shrink the size of the file, the files are stored without compression. If encryption is required, 12 bytes (called the *encryption header*) are prepended to the compressed form, and the encrypted form of the result is kept in the zip-file. The 12 prepended bytes are used for randomization, but also include header dependent data to allow identification of wrong keys when decrypting. In particular, in PKZIP 1.10 the last two bytes of these 12 bytes are derived from the CRC-32 field of the header, and many of the other prepended bytes are constant or can be predicted from other values in the file header. In PKZIP 2.04g, only the last byte of these 12 bytes is derived from the CRC-32 field. The file headers are not encrypted in both versions.

The cipher is byte-oriented, encrypting under variable length keys. It has a 96-bit internal memory, divided into three 32-bit words called key0, key1 and key2. An 8-bit variable key3 (not part of the internal memory) is derived from key2. The key initializes the memory: each key has an equivalent internal representation as three 32-bit words. Two keys are equivalent if their internal representations are the same. The plaintext bytes update the memory during encryption.

The main function of the cipher is called update_keys, and is used to update the internal memory and to derive the variable key3, for each given input (usually plaintext) byte:

$\text{update\_keys}_i(\text{char})$:
  local unsigned short temp
  $\text{key0}_{i+1} \leftarrow \text{crc32}(\text{key0}_i, \text{char})$
  $\text{key1}_{i+1} \leftarrow (\text{key1}_i + \text{LSB}(\text{key0}_{i+1})) * 134775813 + 1 \pmod{2^{32}}$
  $\text{key2}_{i+1} \leftarrow \text{crc32}(\text{key2}_i, \text{MSB}(\text{key1}_{i+1}))$
  $\text{temp}_{i+1} \leftarrow \text{key2}_{i+1} \mid 3 \qquad \text{(16 LS bits)}$
  $\text{key3}_{i+1} \leftarrow \text{LSB}((\text{temp}_{i+1} * (\text{temp}_{i+1} \oplus 1)) \gg 8)$
end update_keys

where $\mid$ is the binary inclusive-or operator, and $\gg$ denotes the right shift operator (as in the C programming language). LSB and MSB denote the least significant byte and the most significant byte of the operands, respectively. Note that the indices are used only for future references and are not part of the algorithm, and that the results of key3 using inclusive-or with 3 in the calculation of temp are the same as with the original inclusive-or with 2 used in the original algorithm. We prefer this notation in order to reduce one bit of uncertainty about temp in the following discussion.

Before encrypting, the key (password) is processed to update the initial value of the internal memory by:

process_keys(key) :
  $\text{key0}_{1-l} \leftarrow$ 0x12345678
  $\text{key1}_{1-l} \leftarrow$ 0x23456789
  $\text{key2}_{1-l} \leftarrow$ 0x34567890
  loop for $i \leftarrow 1$ to l
    $\text{update\_keys}_{i-l}(\text{key}_i)$
  end loop
end process_keys

where l is the length of the key (in bytes) and hexadecimal numbers are prefixed by 0x (as in the C programming language). After executing this procedure, the internal memory contains the internal representation of the key, which is denoted by key0_1, key1_1 and key2_1.

The encryption algorithm itself processes the bytes of the compressed form along with the prepended 12 bytes by:

Encryption	Decryption
prepend $P_1, \ldots, P_{12}$	
loop for $i \leftarrow 1$ to n	loop for $i \leftarrow 1$ to n
$C_i \leftarrow P_i \oplus \text{key3}_i$	$P_i \leftarrow C_i \oplus \text{key3}_i$
$\text{update\_keys}_i(P_i)$	$\text{update\_keys}_i(P_i)$
end loop	end loop
	discard $P_1, \ldots, P_{12}$

The decryption process is similar, except that it discards the 12 prepended bytes.

The crc32 operation takes a previous 32-bit value and a byte, XORs them and calculates the next 32-bit value by the crc polynomial denoted by 0xEDB88320. In practice, a table crctab can be precomputed, and the crc32 calculation becomes:

$$\text{crc32} = \text{crc32}(\text{pval}, \text{char}) = (\text{pval} \gg 8) \oplus \text{crctab}[\text{LSB}(\text{pval}) \oplus \text{char}]$$

The crc32 equation is invertible in the following sense:

$$\text{pval} = \text{crc32}^{-1}(\text{crc32}, \text{char}) = (\text{crc32} \ll 8) \oplus \text{crcinvtab}[\text{MSB}(\text{crc32})] \oplus \text{char}$$

crctab and crcinvtab are precomputed as:

init_crc() :
  local unsigned long temp
  loop for $i \leftarrow 0$ to 255
    temp $\leftarrow \text{crc32}(0, i)$
    $\text{crctab}[i] \leftarrow$ temp
    $\text{crcinvtab}[\text{temp} \gg 24] \leftarrow (\text{temp} \ll 8) \oplus i$
  end loop
end init_crc

in which crc32 refers to the original definition of crc32:

```
crc32(temp, i) :
  temp ← temp ⊕ i
  loop for j ← 0 to 7
    if odd(temp) then
      temp ← temp ≫ 1 ⊕ 0xEDB88320
    else
      temp ← temp ≫ 1
    endif
  end loop
  return temp
end crc32
```

3 The Attack

The attack we describe works even if the known plaintext bytes are not the first bytes (if the file is compressed, it needs the compressed bytes, rather than the uncompressed bytes). In the following discussion the subscripts of the n known plaintext bytes are denoted by $1,\ldots,n$, even if the known bytes are not the first bytes. We ignore the subscripts when the meaning is clear and the discussion holds for all the indices.

Under a known plaintext attack, both the plaintext and the ciphertext are known. In the PKZIP cipher, given a plaintext byte and the corresponding ciphertext byte, the value of the variable key3 can be calculated by

$$\mathrm{key3}_i = P_i \oplus C_i.$$

Given $P_1,\ldots,P_n$ and $C_1,\ldots,C_n$, we receive the values of $\mathrm{key3}_1,\ldots,\mathrm{key3}_n$. The known plaintext bytes are the inputs of the update_keys function, and the derived key3's are the outputs. Therefore, in order to break the cipher, it suffices to solve the set of equations derived from update_keys, and find the initial values of key0, key1 and key2.

In the following subsections we describe how we find many possible values for key2, then how we extend these possible values to possible values of key1, and key0, and how we discard all the wrong values. Then, we remain with only the right values which correspond to the internal representation of the key.

3.1 key2

The value of key3 depends only on the 14 bits of key2 that participate in temp. Any value of key3 is suggested by exactly 64 possible values of temp (and thus 64 possible values of the 14 bits of key2). The two least significant bits of key2 and the 16 most significant bits do not affect key3 (neither temp).

Given the 64 possibilities of temp in one location of the encrypted text, we complete the 16 most significant bits of key2 with all the 2^{16} possible values,

Side	Term	Bits Number	Bits Position	Number of Values
Left	key2_i	14	2–15	64
Right	$\text{key2}_{i+1} \ll 8$	22	10–31	1
	$\text{crcinvtab}[\text{MSB}(\text{key2}_{i+1})]$	32	0–31	1
	$\text{MSB}(\text{key1}_{i+1})$	24	8–31	1
Total Left Hand Side		14	2–15	64
Total Right Hand Side		22	10–31	1
Common bits		6	10–15	
Total bits		30	2–31	

Table 1. The Known Bits in Equation 1

and get 2^{22} possible values for the 30 most significant bits of key2. key2_{i+1} is calculated by $\text{key2}_{i+1} \leftarrow \text{crc32}(\text{key2}_i, \text{MSB}(\text{key1}_{i+1}))$. Thus,

$$\begin{aligned} \text{key2}_i &= \text{crc32}^{-1}(\text{key2}_{i+1}, \text{MSB}(\text{key1}_{i+1})) \\ &= (\text{key2}_{i+1} \ll 8) \oplus \text{crcinvtab}[\text{MSB}(\text{key2}_{i+1})] \oplus \text{MSB}(\text{key1}_{i+1}). \end{aligned} \tag{1}$$

Given any particular value of key2_{i+1}, for each term of this equation we can calculate the value of the 22 most significant bits of the right hand side of the equation, and we know 64 possibilities for the value of 14 bits of the left hand side, as described in Table 1. From the table, we can see that six bits are common to the right hand side and the left hand side. Only about 2^{-6} of the possible values of the 14 bits of key2_i have the same value of the common bits as in the right hand side, and thus, we remain with only one possible value of the 14 bits of key2_i in average, for each possible value of the 30 bits of key2_{i+1}. When this equation holds, we can complete additional bits of the right and the left sides, up to the total of the 30 bits known in at least one of the sides. Thus, we can deduce the 30 most significant bits of key2_i. We get in average one value for these 30 most significant bits of key2_i, for each value of the 30 most significant bits of key2_{i+1}. Therefore, we are now just in the same situation with key2_i as we were before with key2_{i+1}, and we can now find values of 30 bits of key2_{i-1}, key2_{i-2}, ..., key2_1. Given this list of 30-bit values, we can complete the 32-bit values of key2_n, key2_{n-1}, ..., key2_2 (excluding key2_1) using the same equation. We remain with about 2^{22} lists of possible values (key2_n, key2_{n-1}, ..., key2_2), of which one must be the list actually calculated during encryption.

3.2 Reducing the number of possible values of key2

The total complexity of our attack is (as described later) 2^{16} times the number of possible lists of key2's. If we remain with 2^{22} lists, the total complexity becomes 2^{38}. This complexity can be reduced if we can reduce the number of lists of key2's without discarding the right list.

bytes	key2 list entries
1	$2^{22} = 4194304$
2	3473408
3	2152448
4	1789183
5	1521084
10	798169
15	538746
20	409011
25	332606
30	283930
40	213751
50	174471
100	88248
200	43796
300	31088
500	16822
1000	7785
2000	5196
4000	3976
6000	3000
8000	1296
10000	1857
12000	243
12289	801

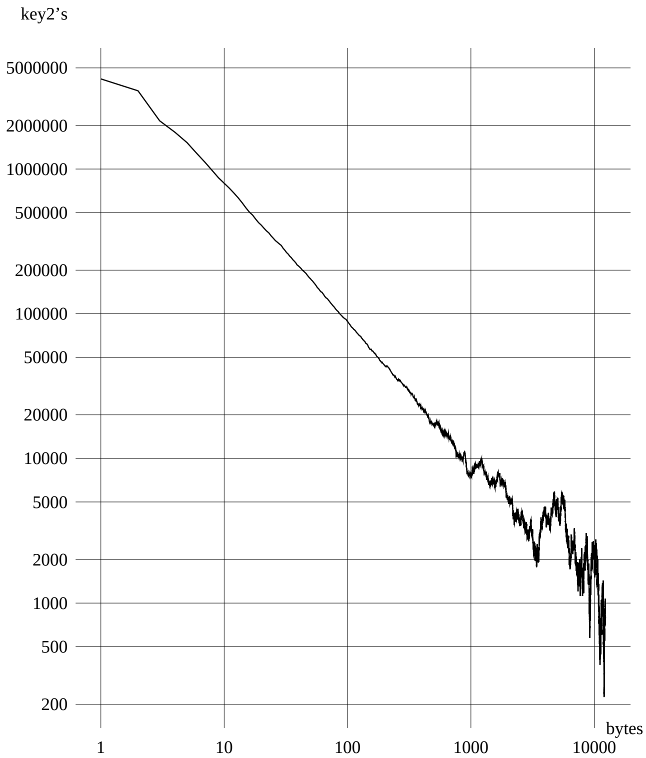

Fig. 1. Decrease in the number of key2 candidates using varying amounts of known plaintext. These results are for the PKZIP contest file and are fairly typical, though the entry 12000 is unusually low. (logarithmic scaling).

We observed that the attack requires only 12–13 known plaintext bytes (as we describe later). Our idea is to use longer known plaintext streams, and to reduce the number of lists based on the additional plaintext. In particular, we are interested only in the values of $\mathrm{key2}_{13}$, and not in all the list of $\mathrm{key2}_i$, $i = 13, \ldots, n$. $\mathrm{key2}_{13}$ is then used in the attack as is described above.

We start with the 2^{22} possible values of $\mathrm{key2}_n$, and calculate the possible values of $\mathrm{key2}_{n-1}$, $\mathrm{key2}_{n-2}$, etc. using Equation 1. The number of possible values of $\mathrm{key2}_i$ ($i = n-1, n-2$, etc.) remains about 2^{22}. However, some of the values are duplicates of other values. When these duplicates are discarded, the number of possible values of $\mathrm{key2}_i$ is substantially decreased. To speed things up, we calculate all the possible values of $\mathrm{key2}_{n-1}$, and remove the duplicates. Then we calculate all the possible values of $\mathrm{key2}_{n-2}$, and remove the duplicates, and so on. When the duplicates fraction becomes smaller, we can remove the duplicates only every several bytes, to save overhead. Figure 1 shows the number of remaining values for any given size of known plaintext participating in the reduction, as was measured on the PKZIP contest file (which is typical). We observed that using about 40 known plaintext bytes (28 of them are used for the reduction and 12 for the attack), the number of possible values of $\mathrm{key2}_{13}$ is reduced to about 2^{18}, and the complexity of the attack is 2^{34}. Using 10000-byte known plaintext, the complexity of our attack is reduced to 2^{24}–2^{27}.

3.3 key1

From the list of ($\mathrm{key2}_n$, $\mathrm{key2}_{n-1}$, ..., $\mathrm{key2}_2$) we can calculate the values of the most significant bytes the key1's by

$$\mathrm{MSB}(\mathrm{key1}_{i+1}) = (\mathrm{key2}_{i+1} \ll 8) \oplus \mathrm{crcinvtab}[\mathrm{MSB}(\mathrm{key2}_{i+1})] \oplus \mathrm{key2}_i.$$

We receive the list ($\mathrm{MSB}(\mathrm{key1}_n)$, $\mathrm{MSB}(\mathrm{key1}_{n-1})$, ..., $\mathrm{MSB}(\mathrm{key1}_3)$) (excluding $\mathrm{MSB}(\mathrm{key1}_2)$).

Given $\mathrm{MSB}(\mathrm{key1}_n)$ and $\mathrm{MSB}(\mathrm{key1}_{n-1})$, we can calculate about 2^{16} values for the full values of $\mathrm{key1}_n$ and $\mathrm{key1}_{n-1} + \mathrm{LSB}(\mathrm{key0}_n)$. This calculation can be done efficiently using lookup tables of size 256–1024. Note that

$$\mathrm{key1}_{n-1} + \mathrm{LSB}(\mathrm{key0}_n) = (\mathrm{key1}_n - 1) \cdot 134775813^{-1} \pmod{2^{32}}$$

and that $\mathrm{LSB}(\mathrm{key0}_n)$ is in the range 0, ..., 255. At this point we have about $2^{11} \cdot 2^{16} = 2^{27}$ (or $2^{22} \cdot 2^{16} = 2^{38}$) possible lists of key2's and $\mathrm{key1}_n$. Note that in the remainder of the attack no additional complexity is added, and all the additional operations contain a fixed number of instructions for each of the already existing list.

The values of $\mathrm{key1}_{n-1} + \mathrm{LSB}(\mathrm{key0}_n)$ are very close to the values of $\mathrm{key1}_{n-1}$ (since we lack only the 8-bit value $\mathrm{LSB}(\mathrm{key0}_n)$). Thus, an average of only $256 \cdot 2^{-8} = 1$ possible value of $\mathrm{key1}_{n-1}$ that leads to the most significant byte of $\mathrm{key1}_{n-2}$ from the list. This value can be found efficiently using the same lookup tables used for finding $\mathrm{key1}_n$, with only a few operations. Then, we remain with a similar situation, in which $\mathrm{key1}_{n-1}$ is known and we lack only eight bits of $\mathrm{key1}_{n-2}$. We find $\mathrm{key1}_{n-2}$ with the same algorithm, and then find the rest of $\mathrm{key1}_{n-3}$, $\mathrm{key1}_{n-4}$, and so on with the same algorithm. We result with about 2^{27} possible lists, each containing the values of ($\mathrm{key2}_n$, $\mathrm{key2}_{n-1}$, ..., $\mathrm{key2}_2$, and $\mathrm{key1}_n$, $\mathrm{key1}_{n-1}$, ..., $\mathrm{key1}_4$) (again, $\mathrm{key1}_3$ cannot be fully recovered since two successive values of MSB(key1) are required to find each value of key1).

3.4 key0

Given a list of ($\mathrm{key1}_n$, $\mathrm{key1}_{n-1}$, ..., $\mathrm{key1}_4$), we can easily calculate the values of the least significant bytes of ($\mathrm{key0}_n$, $\mathrm{key0}_{n-1}$, ..., $\mathrm{key0}_5$) by

$$\mathrm{LSB}(\mathrm{key0}_{i+1}) = ((\mathrm{key1}_{i+1} - 1) \cdot 134775813^{-1}) - \mathrm{key1}_i \pmod{2^{32}}.$$

$\mathrm{key0}_{i+1}$ is calculated by

$$\begin{aligned} \mathrm{key0}_{i+1} &\leftarrow \mathrm{crc32}(\mathrm{key0}_i, P_i) \\ &= (\mathrm{key0}_i \gg 8) \oplus \mathrm{crctab}[\mathrm{LSB}(\mathrm{key0}_i) \oplus P_i]. \end{aligned}$$

Crc32 is a linear function, and from any four consecutive LSB(key0) values, together with the corresponding known plaintext bytes it is possible to recover the full four key0's. Moreover, given one full key0, it is possible to reconstruct all the other key0's by calculating forward and backward, when the plaintext bytes are given. Thus, we can now receive $\mathrm{key0}_n, \ldots, \mathrm{key0}_1$ (this time including $\mathrm{key0}_1$). We can now compare the values of the least significant bytes of $\mathrm{key0}_{n-4}$, $\ldots$, $\mathrm{key0}_{n-7}$ to the corresponding values from the lists. Only a fraction of 2^{-32} of the lists satisfy the equality. Since we have only about 2^{27} possible lists, it is expected that only one list remain. This list must have the right values of the key0's, key1's, and key2's, and in particular the right values of $\mathrm{key0}_n$, $\mathrm{key1}_n$ and $\mathrm{key2}_n$. In total we need 12 known plaintext bytes for this analysis (except for reducing the number of key2 lists) since in the lists the values of $\mathrm{LSB}(\mathrm{key0}_i)$ start with $i = 5$, and $n - 7 = 5 \Rightarrow n = 12$.

If no reduction of the number of key2 lists is performed, 2^{38} lists of (key0, key1, key2) remain at this point, rather than 2^{27}. Thus, we need to compare five bytes $\mathrm{key0}_{n-4}, \ldots, \mathrm{key0}_{n-8}$ in order to remain with only one list. In this case, 13 known plaintext bytes are required for the whole attack, and the complexity of analysis is 2^{38}.

3.5 The Internal Representation of the Key

Given $\mathrm{key0}_n$, $\mathrm{key1}_n$ and $\mathrm{key2}_n$, it is possible to construct $\mathrm{key0}_i$, $\mathrm{key1}_i$ and $\mathrm{key2}_i$ for any $i < n$ using only the ciphertext bytes, without using the known plaintext, and even if the known plaintext starts in the middle of the encrypted file this construction works and provides also the unknown plaintext and the 12 prepended bytes. In particular it can find the internal representation of the key, denoted by $\mathrm{key0}_1$, $\mathrm{key1}_1$ and $\mathrm{key2}_1$ (where the index denotes again the index in the encrypted text, rather than in the known plaintext). The calculation is as follows:

$$
\begin{aligned}
\mathrm{key2}_i &= \mathrm{crc32}^{-1}(\mathrm{key2}_{i+1}, \mathrm{MSB}(\mathrm{key1}_{i+1})) \\
\mathrm{key1}_i &= ((\mathrm{key1}_{i+1} - 1) * 134775813^{-1}) - \mathrm{LSB}(\mathrm{key0}_{i+1}) \pmod{2^{32}} \\
\mathrm{temp}_i &= \mathrm{key2}_i \mid 3 \\
\mathrm{key3}_i &= \mathrm{LSB}((\mathrm{temp}_i * (\mathrm{temp}_i \oplus 1)) \gg 8) \\
P_i &= C_i \oplus \mathrm{key3}_i \\
\mathrm{key0}_i &= \mathrm{crc32}(\mathrm{key0}_{i+1}, P_i)
\end{aligned}
\tag{2}
$$

The resulting value of ($\mathrm{key0}_1$, $\mathrm{key1}_1$, $\mathrm{key2}_1$) is the internal representation of the key. It is independent of the plaintext and the prepended bytes, and depends only on the key. With this internal representation of the key we can decrypt any ciphertext encrypted under the same key. The two bytes of crc32 (one byte in version 2.04g) which are included in the 12 prepended bytes allow further verification that the file is really encrypted under the found internal representation of the key.

Key length	1–6	7	8	9	10	11	12	13
Complexity	1	2^8	2^{16}	2^{24}	2^{32}	2^{40}	2^{48}	2^{56}

Table 2. Complexity of finding the key itself

3.6 The Key (Password)

The internal representation of the key suffices to break the cipher. However, we can go even further and find the key itself from this internal representation with the complexities summarized in Table 2. The algorithm tries all key lengths 0, 1, 2, ..., up to some maximal length; for each key length it does as described in the following paragraphs.

For $l \leq 4$ it knows key0_{1-l} and key0_1. Only $l \leq 4$ key bytes are entered to the crc32 calculations that update key0_{1-l} into key0_1. Crc32 is a linear function, and these $l \leq 4$ key bytes can be recovered, just as key0_n, ..., key0_{n-3} recovered above. Given the l key bytes, we reconstruct the internal representation, and verify that we get key1_1 and key2_1 as expected (key0_1 must be as expected, due to the construction). If the verification succeeds, we found the key (or an equivalent key). Otherwise, we try the next key length.

For $5 \leq l \leq 6$ we can calculate key1_0, key2_0 and key2_{-1}, as in Equation 2. Then, key2_{2-l}, ..., key2_{-2} can be recovered since they are also calculated with crc32, and depend on $l-2 \leq 4$ unknown bytes (of key1's). These unknown bytes $\text{MSB}(\text{key1}_{2-l})$, ..., $\text{MSB}(\text{key1}_{-1})$ are also recovered at the same time. key1_{1-l} is known. Thus, we can receive an average of one possible value for key1_{2-l} and for key1_{3-l}, together with $\text{LSB}(\text{key0}_{2-l})$ and $\text{LSB}(\text{key0}_{3-l})$, using the same lookup tables used in the attack. From $\text{LSB}(\text{key0}_{2-l})$ and $\text{LSB}(\text{key0}_{3-l})$ and key0_{1-l}, we can complete key0_{2-l} and key0_{3-l} and get key_1 and key_2. The remaining $l-2$ key bytes are found by solving the $l-2 \leq 4$ missing bytes of the crc32 as is done for the case of $l \leq 4$. Finally, we verify that the received key has the expected internal representation. If so, we have found the key (or an equivalent key). Otherwise, we try the next key length.

For $l > 6$, we try all the possible values of key_1, ..., key_{l-6}, calculating key0_{-5}, key1_{-5} and key2_{-5}. Then we used the $l = 6$ algorithm to find the remaining six key bytes. In total we try about $2^{8 \cdot (l-6)}$ keys. Only a fraction of 2^{-64} of them pass the verification (2^{-32} due to each of key1 and key2). Thus, we expect to remain with only the right key (or an equivalent) in trials of up to 13-byte keys. Note that keys longer than 12 bytes will almost always have equivalents with up to 12 (or sometimes 13) bytes, since the internal representation is only 12-bytes long.

4 Summary

In this paper we describe a new attack on the PKZIP stream cipher which finds the internal representation of the key, which suffices to decrypt the whole file

Bytes	13	40	110	310	510	1000	4000	10000
Complexity	2^{38}	2^{34}	2^{32}	2^{31}	2^{30}	2^{29}	2^{28}	2^{27}

Table 3. Complexity of the attack by the size of the known plaintext

and any other file which is encrypted by the same key. This known plaintext attack breaks the cipher using 40 (compressed) known plaintext bytes, or about the 200 first uncompressed bytes (if the file is compressed), with complexity 2^{34}. Using about 10000 known plaintext bytes, the complexity is reduced to about 2^{27}. Table 3 describes the complexity of the attack for various sizes of known plaintext. The original key (password) can be constructed from the internal representation. An implementation of this attack in software was applied against the PKZIP cipher contest. It found the key "f7 30 69 89 77 b1 20" (in hexadecimal) within a few hours on a personal computer.

A variant of the attack requires only 13 known plaintext bytes, in price of a higher complexity 2^{38}. Since the last two bytes (one in version 2.04g) of the 12 prepended bytes are always known, if the known plaintext portion of the file is in its beginning, the attack requires only 11 (12) known plaintext bytes of the compressed file. (In version 1.10 several additional prepended bytes might be predictable, thus the attack might actually require even fewer known plaintext bytes.)

We conclude that the PKZIP cipher is weak and that it should not be used to protect valuable information.

References

1. PKWARE, Inc., *General Format of a ZIP File*, technical note, included in PKZIP 1.10 distribution (pkz110.exe: file appnote.txt).