is 512. □

This demonstrates the contribution to the security of RC4 made by the simple swapping of S table entries in the memory update function.

2.2.3 A Stream Cipher Allegedly Used in WordPerfect 6

In 1999, John Kuslich of CRAK Software, a password recovery company in the USA, posted an article to the Internet newsgroup sci.crypt describing a stream cipher that is allegedly used in "a very well known word processor" [27]. The author of the posting was asking for help in the cryptanalysis of the cipher.

We asked for more specific details about the cipher and Kuslich wrote that the cipher is used in the latest versions of the WordPerfect word processing software (versions 6, 7 and 8) for PCs for encrypting document files with a password [29]. The encryption algorithm was reverse engineered from the WordPerfect software. Kuslich was kind enough to provide us with a reasonably clear description of the stream cipher, but did not provide certain implementation details that would be necessary to: (1) verify that the stream cipher described is the same as the one used in WordPerfect; and (2) write a password cracker for WordPerfect.

The encryption algorithm used to protect documents in WordPerfect versions 4.2 and 5.0 was cryptanalysed in 1987 and 1991 [3, 2], and there are currently a number of web pages that describe how to break the encryption in WordPerfect 5, which was based on a repeating-key cipher and could be broken by adapting classical methods of breaking Vigenere ciphers.

On the other hand, whilst commercial password recovery software for WordPerfect 6 exists, to date there has been no published description of the cipher it uses or its cryptanalysis. WordPerfect offers two levels of protection to the user: regular or enhanced. Kuslich says:

> "I believe, but would have to verify, that the only difference between enhanced and regular WP encryption is the case sensitivity of the password. In other words, it adds nothing to the strength of the cipher." [29]

What follows is a description of the cipher, based on the description in [27], and our cryptanalysis of the cipher.

The memory state of the cipher is a 32-byte shift register. In each cycle, the register is shifted by one byte (the keystream byte) and a new byte is loaded into the empty end of the shift register, which is computed as follows:

All bytes in odd positions of the shift register are XORed and used as an index into a fixed 256-byte table T_O, and all bytes in even positions of the shift register are XORed and used as an index into a different fixed 256-byte table T_E. Both T_O and T_E are non-surjective functions on $\mathbb{Z}_{256}$. The two values obtained, from T_O and T_E, are XORed and used as the new byte in the empty end of the shift register.

Letting a_i denote the $(i+1)$-th byte (from the left) in the shift register, we have

$$(a'_0, \ldots, a'_{30}, a'_{31}) \quad = \quad (a_1, \ldots, a_{31}, T_O(\oplus_{i=0}^{15} a_{2i+1}) \oplus T_E(\oplus_{i=0}^{15} a_{2i})) \tag{2.7}$$

The byte a_0 is the keystream byte, which is XORed with the plaintext byte to give the ciphertext byte.

Kuslich gave some more information about the problem:

> "[T]he password is entered from an standard initial state (always the same for all files) as unicode data one byte at a time as the SR is cycled." [28]
>
> "The password is XORed into the empty end of the shift register at a certain point in the shift cycle in a particular way (the password characters are not just XORed in directly, they have to be in a particular format first), secondly, there is a "magic number" Xored in in a similar fashion." [28]

This suggests that the cipher has a special clocking rule when it is initialised, in which a password or "salt" byte is XORed to the feedback byte in each clock cycle. The *salt* is a sequence of pseudo-random bytes prepended to the password to ensure that two identical files encrypted with the same password result in different ciphertexts. Let $R = (r_0, r_1, \ldots, r_{N-1}, r_N, \ldots, r_{N+P-1})$ be a sequence of bytes, in which the first N bytes are the salt and the subsequent P bytes represent the Unicode-encoded password string. Then, in clock cycle t of the initialisation phase, the shift register is updated by

$$(a'_0, \ldots, a'_{30}, a'_{31}) \quad = \quad (a_1, \ldots, a_{31}, T_O(\oplus_{i=0}^{15} a_{2i+1}) \oplus T_E(\oplus_{j=0}^{15} a_{2j}) \oplus r_t) \tag{2.8}$$

The shift register is initialised to a constant value (which we call the *setup constant*) at clock cycle $t = 0$, then clocked N times with the salt added into the update rule, clocked a further P times with the Unicode-encoded password added into the update rule, and clocked $128 - N - P$ times with the standard update rule. In total, the initialisation phase clocks the shift register 128 times [27].

We assume that the attacker has access to the keystream produced by this cipher (e.g., determined from known plaintext), and that the tables T_E and T_O, the setup constant and the salt are known (in WordPerfect, the salt is stored within the password-protected document file). The problem is to recover the password (i.e., the bytes

$R_N, \ldots, R_{N+P-1}$) used within the initialisation phase. The length of the Unicode-encoded password P will be even, since every character encoded in Unicode uses 16 bits (ASCII uses only 7 bits). When encoding text, eight bits are set to zero, so we can expect the values r_{N+2i+1}, where $0 \leq i < P/2$, to have a value of zero.

We present a fast method to recover the password, provided that the length of the password P is less than or equal to 32 bytes (16 text characters). Our method is a divide-and-conquer attack. Let the shift register at clock cycle t be denoted by A_t.

The first part of the attack determines the values of the shift register after the salt is loaded (at clock cycle $t = N$) and also 32 steps after the password is loaded (at $t = N + 32$).

1. Initialise a 32-byte register A_0 with the setup constant.

2. Clock the register N times with the salt $(r_0, \ldots, r_{N-1})$ to obtain the value of A_N. We store this value in a 32-byte register X.

3. Determine the value of A_{128} from known plaintext. Any 32 consecutive keystream bytes reveal the state of the shift register. Even if the known keystream bytes are not the first 32 bytes generated directly after the cipher initialisation phase (when $t = 128$), the shift register can be clocked backwards (more on this later) to obtain a list of possible values for A_{128}.

4. Clock the register backwards $96 - N$ times to get A_{N+32}. There are no salt or password bytes loaded into the register during this part of the cipher initialisation phase, so it is relatively straightforward to do this. We end up with a list of possible values of A_{N+32}. We store this list of values in an array of 32-byte registers Y.

Step 4 will only give us the correct value of A_{N+32} if the password is less than 32 bytes, because we are assuming that there were no unknown bytes r_t loaded into the register in the last $96 - N$ clock cycles of the cipher initialisation phase.

The second part of the attack is an iterative algorithm that recovers the sequence of 32 bytes $(r_t, \ldots, r_{t+32})$ loaded between two states A_t and A_{t+32}. In this instance, $t = N$, so that the sequence recovered will be the password (assuming that the password is less than 32 bytes in length). The bytes $r_{N+1}, r_{N+3}, \ldots, r_{N+31}$ will be equal to zero, since we expect the password to be Unicode-encoded text. The idea behind the following algorithm is that we compare portions of the registers in the list Y with the register X, exploiting the fact that in each clock cycle precisely one byte changes in the state.

We use this to discard register states in Y that could not have followed from the state X, and thus aim to converge the list Y on the state X. In this process, we will try possible values of the password bytes r_t in order to successfully find the correct predecessor state (which would follow from the state X) and thus determine the actual or equivalent password bytes.

In each iteration, the following algorithm examines one pair of possible states (i.e., the states in two clock cycles): one representing the zero byte of the encoded password, which is used to discard possible states, and the other is the state caused by a non-zero password character (e.g., alphabetic or numeric), which we use to deduce the password character.

1. Let $t = 32$, the number of clock cycles separating X and Y.

3. Decrement t by one.

2. Clock backwards each register in the list Y to obtain the list of states corresponding to A_{N+t}. Let this list of states replace the current list Y.

4. Compare the first $32 - t$ bytes of each register in Y with the last $32 - t$ bytes of the register X. Remove any registers from Y that do not match, since they would not follow from X.

5. Decrement t by one.

6. For each possible non-zero value of the byte r_{N+t} (alternatively: for all possible alphabetic, numeric and other byte values that would be normally be used in a password):

6.1. Let Z be a copy the list of states Y. We use Z as a temporary working copy of Y.

6.2. Clock backwards each register (using our guess for r_{N+t}) in the list Z to obtain the list of states corresponding to A_{N+t}.

6.3. Compare the first $32 - t$ bytes of each register in Z with the last $32 - t$ bytes of the register X. Remove any registers from Z that do not match, since they would not follow from X. If no registers in the list Z remain, then our guess r_{N+t} is not valid. If some states in Z do exist, then we note that r_{N+t} is a possible candidate for the actual r_{N+t} in the password sequence.

7. For each register in the list Y:

7.1. Shift each byte to the right by one (losing the rightmost byte) and let the leftmost byte equal the byte in position t of X (considering the leftmost byte as being in position 0). This has the effect of clocking the register backwards by one, giving us a list of of registers corresponding to A_{N+t}.

8. If t is non-zero then go to Step 3.

Earlier we explained that it was necessary to clock the shift register backwards and that doing so will result in a number of possible predecessor states (depending on how skewed the distribution of the T_E and T_O tables are). The memory update function is that of a non-linear feedback shift register and is not bijective. Given a 32-byte value A_{t+1}, we may find that either (1) there will be at least one possible value of A_t which can produce the state A_{t+1} by the update function; or (2) there are no valid possibilities for the predecessor state A_t (i.e., the state A_{t+1} cannot occur by the update function).

From equation 2.8, we have

$$a'_{31} = T_E(a_0 \oplus a_2 \oplus \ldots \oplus a_{30}) \oplus T_O(a_1 \oplus a_3 \oplus \ldots \oplus a_{31}) \oplus r_t$$

Substituting a'_{i-1} for a_i:

$$a'_{31} = T_E(a_0 \oplus a'_1 \oplus \ldots \oplus a'_{29}) \oplus T_O(a'_0 \oplus a'_2 \oplus \ldots \oplus a'_{30}) \oplus r_t$$

Rearranging to give an equation for a_0:

$$a_0 = T_E^{-1}(r_t \oplus a'_{31} \oplus T_O(a'_0 \oplus a'_2 \oplus \ldots \oplus a'_{30})) \oplus a'_1 \oplus \ldots \oplus a'_{29}$$

This equation is the basis for the shift register inversion algorithm. The function T_E^{-1} may be undefined for certain inputs and may have many possible outputs for other inputs. If the value of $T_E^{-1}(x)$ is undefined for a given x, then the shift register value that we are trying to invert could not have occurred (otherwise we would expect at least one value for $T_E^{-1}(x)$), and so we immediately drop this search. Otherwise, we continue computing the value or values for a_0 and invert each possibility in turn.

First we build a special set of T_E inversion tables.

1. Let C denote an array with 256 entries. Each entry in the array is initialised to zero (i.e., $C(x) = 0$ for all $x = 0, \ldots, 255$).

2. We prepare six T_E "inverse" tables $T_{E,i}$, for $i = 0, \ldots, 5$. Each table maps one byte to one byte.

3. For each byte $x = 0, \ldots, 255$:

3.1. Let $T_{E,C(T_E(x))} = x$.

3.2. Increment $C(T_E(x))$.

The counter array C keeps track of the "depth" of each entry in the T_E "inverse" tables. Once these tables have been computed, the algorithm to invert the shift register can be executed. The algorithm takes a list of 32-byte states Y and a one byte value r_t (set to zero for reversing the normal clocking rule of equation 2.7), and generates a list of predecessor states.

1. For each 32-byte state $A = (a_0, a_1, \ldots, a_{31})$ in the list of states Y:

1.1. Let $x = T_O(\oplus_{i=0}^{15} a_{2i}) \oplus a_{31} \oplus r_t$.

1.2. If $C(x)$ equals zero then this state has no predecessors, so return to Step 1.

1.3. For each $i = 0, \ldots, C(x) - 1$:

1.3.1 Let $z_0 = T_{E,i}(x) \oplus \bigoplus_{j=0}^{14} a_{2j+1}$.

1.3.2 Let $z_j = a_{j-1}$ for $j = 1, \ldots, 31$.

1.3.3 Note that the array $Z = (z_0, \ldots, z_{31})$ is a predecessor state of A.

2.3 Non-Surjective Functions in Keystream Generators

As already explained, the keystream generator should be a surjective function from the memory state to a keystream unit. Given the output of this function, i.e., a keystream unit, it should be difficult to determine precisely which input generated the output. A non-surjective function may be trivially created from a surjective one simply by extending the codomain (without changing the function itself), and our theory does not apply to such a general definition. This chapter specifically concentrates on non-surjective functions in which the domain and codomain are finite and of the same size. Equivalently, the function would be non-bijective. To show that a function of this type is non-surjective, it suffices to show that collisions exist, i.e., that the function has two distinct inputs which result in the same output. Because the domain and codomain are the same size, this implies that there is some element in the codomain that has no preimage.

The stream ciphers which are weakened by non-surjective keystream generators also introduce some key element inside or subsequent to the non-surjective function, which