WESP: An encryption method that is proven to require an exponentially growing time to break it

Sam Widlund
sam.widlund@wesp.fi
https://www.wesp.fi

1 ABSTRACT

First, the WESP encryption algorithm is defined. Encryptions are created by a system of equations in which the equations are generated using the values of tables that act as the encryption key. Next, it is shown that if the encryption tables are not known, the equations in the system of equations cannot be known, and it is demonstrated that the system of equations cannot be solved if the equations are not known, and thus the encryption cannot be broken in a closed form.

Then, we calculate for all symbols used in the algorithm, the minimum amount of trials needed, in order to be able to verity the trials. Since the algorithm is constantly updating key values the verifying becomes impossible if equations are not evaluated in order. The calculation shows that the minimum number of trials required is such that the number of trials, i.e., the time required to break the encryption, increases exponentially as the key size grows. Since there is no upper limit on the key size there is neither any upper limit on the time it requires to break the encryption.

It is also shown that the well-known mathematical "P vs NP" problem is solved by the presented proof. Since there is at least one algorithm (WESP) that is NP (verifiable in polynomial time) but not P (solvable in polynomial time), P cannot be the same set as NP.

2 KEYWORDS

Cipher; Encryption; Equation system; Exponential time; P = NP;

3 INTRODUCTION

It has been proven that the 'One-Time Pad' OTP [1] is an encryption method that cannot be broken in any way. OTP uses a one-time pad for each character to be encrypted, which is distributed in advance to the parties for encryption. The main problem with OTP is the practical issue of organizing the distribution and management of sufficiently large keys in advance, so that the keys do not become compromised. Therefore, OTP is not widely used in computer applications.

All encryption methods in which the key is shorter than the message being encrypted are susceptible to the attacker / adversary obtaining some information about the plaintext message because not all combinations of plaintext messages are possible. In practice, it is difficult to arrange a key that is at least as long as the message being encrypted, and we can usually only use encryption where the key is shorter than the message.

Except for OTP, no known encryption technique has previously demonstrated how difficult it is to break the encryption in absolute terms. There are many proofs that show that 'this is as difficult to break as some other problem X'. But for this other problem X, there is no other proof than that no one has yet figured out how to break it. For example, if a state intelligence organization had discovered a way to break one of the currently widely used ciphers, would they publish it? Probably not, but instead enjoy the exclusive ability to read other people's messages without their knowledge. Therefore, all encrypted communication today is very uncertain; we do not know whether the encryption algorithm we are using has already been broken or not. And even if it has not been broken yet, foreign states undoubtedly store encrypted messages, such as those of politicians, and if the messages cannot be opened yet, they might be able to do so soon. In encryption, the most important thing is that it is reliable. The speed at which it is done is now secondary on modern machines. And the encryption algorithm is reliable only if it has been proven to be reliable. If it has not been proven, it is not reliable, but only based on belief and hope. This is why it is important to develop encryption algorithms whose breaking time is known and proven.

Next, we present an encryption algorithm and a proof that demonstrates that the time required to break the WESP algorithm can be adjusted to any value, using keys that are practically manageable. We achieve almost OTP level encryption.

The WESP algorithm uses only mathematical techniques that have been used for centuries, such as equation systems. The properties of equation systems are well-known. By using only very well-known mathematical methods, we can be sure that there are no shortcuts to solve the mathematics we use in encryption. Also the proof uses only simple mathematics, e.g. derivatives.

Design principles while developing the proof and the algorithm has been security first. The key size, the complexity of the algorithm, and the execution speed has had no relevance. However, the developed algorithm is fast enough, and do not require too much memory or CPU power, to function well in most environments, e.g. in mobile phones encrypting video streams.

We will also demonstrate (in Appendix 2) that the encryption algorithm solves the well-known mathematical P vs NP problem. Many different solutions [2] have been proposed for the P vs NP problem, none of which have gained general acceptance. Various requirements have also been shown to be required by a proof to the P vs NP problem, which the described encryption algorithm also meets [9] [10] [11].

It is often perceived that the probability of a new proof solving the P vs NP problem is low. However, the perceived low probability do not in any way affect the correctness of the presented proof. If there is no definite logical flaw in the presented proof, it holds true, whether someone thinks it is likely or not.

The starting point of the proof is that an algorithm creates a system of equations from key values. In a system of equations, generally, at most one unknown variable can be solved with one equation, and if all variables are to be solved, at least as many equations are needed as there are variables[3]. If no equations are known, no unknowns can be solved. The proof will show that we can only know equations if we guess values for the key values. We also calculate how many key values need to be guessed in order to be able to verify the guesses. The end result is that the required number of guessed values to keys increases exponentially as the number of key values increases. In a Turing machine[8], what current computers are, if the number of guesses needed increases exponentially then the also the time to break the encryption increases exponentially.

4 DEFINITION OF THE WESP ALGORITHM

WESP is a "stream cipher" that encrypts a message one byte at a time. The encrypted bytes produced by WESP are completely independent of the plaintext being encrypted, and therefore the content of the plaintext has no significance in terms of breaking the encryption. WESP is a symmetric encryption method, which means that both the sender and the receiver have the same secret key available. In this document we have used as the byte length 8 bits, but the byte could be of any other length.

The key consists of N tables that contain secret random data. The size of the key can be adjusted, and can be, for example, a few kilobytes or a few tens of kilobytes. This describes an algorithm in which there are 8 tables named A-H, but the number of tables can be something else entirely. We refer to these tables as key tables. The lengths of the tables are different from each other, and taken in pairs, the greatest common factor of all lengths is one. All table lengths must be longer than 260. The easiest way to ensure that the greatest common factor is one is to make sure that the lengths are all different prime numbers.

We define Ltot as the sum of the lengths of key tables A-H. VB (Verification Blocker) is a separate table that belongs to the key, whose length is Ltot, and which also contains random key data. The total length of the key is thus 2*Ltot.

$$CB_n = Z_n \oplus D_n$$

CB is the encrypted byte whose value the attacker attempting to break the algorithm knows, Z is the encrypting byte produced by the WESP algorithm, and D is the byte to be encrypted. $\oplus$ is the bitwise XOR operation. n is the order number of the byte to be encrypted, starting from 0. In "chosen text" attacks, the value of Z_n is known for those n values for which the attacker also knows the value of D_n.

Decrypting the encrypted message is easy when the sender and the receiver have the same key available. When we know the key that creates the Z_n value, we swap the positions of CB_n and D_n in the equation, and we can decrypt the encrypted byte.

We first introduce the algorithm as pseudo-code. After the pseudo-code, we go through the code. Why things are done the way they are done is described in the proof.

With the notation |X|, we mean the length of table X. A[n] means the nth element of table A.
X := Y means that variable X gets the value Y.
X ⊕= Y means that variable X gets the value X ⊕ Y.
mod is the modulo operation.
// means that the rest of the line is a comment.
The integer constant Lmultiplier is defined later in the explanation.

WESP as pseudo-code

```
// init variables
   int n := -1 // The sequence number of the byte to be encrypted
   int m := -1 // Current position in tables

   function g(x)
     return
       A[x mod |A|] ⊕
       B[x mod |B|] ⊕
       C[x mod |C|] ⊕
       D[x mod |D|] ⊕
       E[x mod |E|] ⊕
       F[x mod |F|] ⊕
       G[x mod |G|] ⊕
       H[x mod |H|]
   endfunction

   function nextZ()
     n := n + 1
     m := m + 1
     int deltaM := g(m) + 1

     m := m + deltaM // progressing based on the key values
     int K := g(m) ⊕ (n mod 256) // the value to update to keys
     m := m + 1
     int L := (g(m) ⊕ (n mod 256)) * Lmultiplier
     int mL := m + L // which key values to update

     // Modify the key values by XORing them with K
     A[mL mod |A|] ⊕= K
     B[mL mod |B|] ⊕= K
     C[mL mod |C|] ⊕= K
     D[mL mod |D|] ⊕= K
     E[mL mod |E|] ⊕= K
     F[mL mod |F|] ⊕= K
     G[mL mod |G|] ⊕= K
     H[mL mod |H|] ⊕= K

     m := m + 1
     int P = g(m)

     int fm := P ⊕ K ⊕ (L mod 256)
     if (deltaM < 64) then
       fm ⊕= (m mod 256)

     return fm ⊕ VB[n mod |VB|]
   endfunction
```

Call the nextZ function for each byte to be encrypted and perform a xor operation between the nextZ result Z_n and the byte to be encrypted D_n, resulting in the encrypted byte CB_n.

Description of pseudo-code

We define Nt as the number of key tables used. In this example, Nt = 8 (tables A-H). Nt must be at least 3. The smallest possible Ltot is 793 (three tables, the length of the first table is the smallest prime number > 260, which is 263, the length of the second table is the second smallest prime number > 260, which is 269, and the length of the third table is the smallest number greater than 260, which is 261).

The m value and the progress of the algorithm are calculated as follows:

At the beginning of the encryption, n and m are set to -1.

When the next byte is encrypted, n and m are incremented by one. The value of g(m) is calculated with the new value of m, one is added to it, and the result is called deltaM. The value of m is increased by deltaM (which advances through the encryption tables by 1-256 values of n). The value of g(m) is calculated with this new value of m, XORed with n mod 256, and the result is called K. The value of m is then incremented by one. The value of g(m) is calculated again with this new value of m, XORed with n mod 256, and the result is called L. The values of tables A-H at the index [(current m + L) mod the length of the corresponding table] are set to the current value of the table XORed with the value of K (in other words, the key values are modified). m is then incremented by one again, g(m) is calculated, and the result is called P.

The L and K values are also XORed with n mod 256. This is done because some key combinations may cause regularities in the g(m) values, and XORing with n mod 256 ensures that all L and K values contain all values between 0 and 255. For example, if we didn't XOR the K and L values with n mod 256, and all key values in table A were 2, and all other key values in the other tables were 0, then K and L would always be 2. This means that we would be XORing even table key values with an even number, and all other bits would always be 0, so all fm values would be 2. Even though the attacker knows the value of n (and therefore n mod 256), the use of L and K ensures that the attacker does not gain any advantage from this knowledge. The final value of nextZ will contain the n mod 256 component twice, so this value will cancel out in the final result.

The elements of the key tables contain independent, random values, and all key combinations are equally likely. Also, key combinations that produce statistically deviant Z_n values (such as all even values or their equivalent) are equally likely as any other key combination. XORing the L and K values with n mod 256 ensures that the Z_n values are statistically evenly distributed for all possible key combinations.

fm is defined as follows: If deltaM < 64, then fm = P $\oplus$ K $\oplus$ L, otherwise fm = P $\oplus$ K $\oplus$ L $\oplus$ (m mod 256).

On average, 25% of the n values are fm = P $\oplus$ K $\oplus$ L, and 75% of the n values are fm = P $\oplus$ K $\oplus$ L $\oplus$ (m mod 256).

The conditionality of the fm value may complicate e.g. the use of quantum computers (or other future devices) when trying to break the algorithm. The purpose of using K and L values is also to increase the number of bits used on average for each Z_n value calculation (on average, we use 8 bytes for K calculations and 8 bytes for L calculations, a total of 16 bytes more key values than without using K and L values). The number of bits used is important because we will show that it affects the number of guesses required if the values of the tables are to be guessed.

The algorithm proceeds in a way that depends on the key values, and constantly changes the key values in a way that depends on the key values.

Since the lengths of the tables are > 255, the K value does not update evenly over the entire area of the tables because L (without the Lmultiplier) is at most 255. Therefore, we calculate the coefficient Lmultiplier = 'the longest length of the A-H tables divided by 256', rounded up to the nearest integer, if using smallish keys then often the value being 2. Once the L value is calculated, we multiply it by this coefficient. Although the L value cannot directly point to any table element after this, but since the m value is evenly distributed among all table elements, any number that is evenly distributed plus any number that can be longer than the length of the tables (mod the table length) can statistically point to any element.

The algorithm will repeat the same Z values at some point. The algorithm can only be used as long as the Z values do not repeat. In Appendix 1, we present a calculation of the maximum length of messages that the algorithm can encrypt and call it the sequence length. In practice, it is easy to adjust the key size so that the Z values do not repeat with the encryption amounts we will use. Similarly, we can adjust the key size so that a currently stored encrypted message can unlikely be opened in the future, even after a long time.

5 BREAKING THE ALGORITHM

An attacker wants to break the encryption. The attacker can e.g. eavesdrop on encrypted communication as much as they (he/she) want, and they know what both the CB_n and D_n values are for all values of the entire sequence except for one, where the attacker knows the CB_n value but not the D_n value, and the attacker wants to solve this one D_n byte. The attacker can also freely choose D_n values for all but this one byte ("chosen text attack"). The byte to be solved can therefore be located anywhere, at the beginning, in the middle, or at the end of the message. By breaking the encryption, we mean that the attacker obtains such information about this one unknown D byte that at least one D value is more likely than 1/256.

We assume that the attacker knows the algorithm, the number of key tables, and their lengths, but not their contents, nor the contents of the VB table. In addition, we assume that the attacker knows the n values.

We assume that the values of the elements in the key used for encryption are completely random, with each element having a byte value between 0 and 255 and each value having a probability of 1/256, and that the values of the key elements are completely independent of each other. Generating such completely random key elements is not within the scope of this proof, but we assume that they can be generated in some way.

In the computer implementation of the algorithm there may be errors that can be used to break the encryption. The implementation may also be sensitive to so-called "side channel attack" attacks. In the proof presented, we focus on breaking the algorithm itself and not on the characteristics of possible implementations.

6 THEOREM

1. It is impossible to determine any value of Z_n without trying values for the key elements
2. If we try values for the key, the number of necessary trials is on average at least $\frac{256^{Ltot*0.999}}{2}$

7 PROOF

When we encrypt a certain byte, we update the K value to multiple key values. The Z_n values are known and are formed by the XOR result of multiple key values, forming equations. We can write the Z_n equations so that with a certain n value the K value is calculated as, for example, A[5] $\oplus$ B[6] $\oplus$ C[12] $\oplus$ D[2], and it is updated to e.g. the key value A[35], then in the future, whenever we use the updated key value A[35], we write A[35] $\oplus$ A[5] $\oplus$ B[6] $\oplus$ C[12] $\oplus$ D[2] instead of A[35] in the equations. Thus, in the equations we use only values that appeared in the key at the beginning of encryption. We follow this procedure whenever we update a key value. As the encryption progresses, the equations for each key value become longer, and all key values starts to appear multiple times in the equations. Some key values appear an even number of times, while others appear an odd number of times. The key values that appear an even number of times can be removed (X $\oplus$ X is zero and Y $\oplus$ 0 is Y), and those that appear an odd number of times can be replaced with one key value. The equations written in this way form a system of equations that use only immutable values that were present in the key values at the beginning of encryption. As the encryption progresses, the equations obtained become longer but contain only immutable initial key values. The algorithm thus forms a system of equations, even if the attacker does not know the equations of the equation system.

To solve the system of equations, we need at least as many equations as the number of variables we want to solve. If there are constraints in some equations, fewer equations may be sufficient. For example, from the equation $x^2 + y^2 = 0$, we can solve two variables with one equation because the equation has a constraint: the second power of a real number is always $>= 0$. All equations in the WESP algorithm have the form "some variables XORed together form a certain value." If a variable changes, some or several other variables change in the same way. No possible value of any key byte is excluded due to the XOR operations. Therefore, there are no such constraints in the WESP algorithm. In our algorithm, we can solve at most as many variables as we have known equations, and one equation can solve at most one variable.

The number of solutions

In the fm values, Ltot variables has been used, and this can result in 256^{Ltot} different combinations of variables. Whenever any of these variables change, the deltaM value changes when the changed variable is used in the

calculation of the deltaM value. Each of these combinations of variables forms an equation system in deltaM calculation, in which all equations are unique, containing all Ltot variables, and due to the n mod 256 factor, all possible bit combinations between 0-255 are certainly present. No equation is redundant just because it is similar to another equation. Then each of these "deltaM equations" can be solved uniquely after at least Ltot equations (if the deltaM equations are known). More than Ltot equations may be needed for the solution, but as the encryption proceeds, we constantly receive more equations, and sooner or later, at some point, the equation system becomes fully solvable. The solution can be done, for example, using Gauss's [6] method. Since each of the 256^{Ltot} possible equation systems has a unique solution, all 256^{Ltot} different equation systems must be different; otherwise, the solution of some equation system would not have been unique.

Therefore, all 256^{Ltot} key combinations form a different deltaM series and thus a different equation system of fm values. In the same way as above, each of these equation systems of fm values is uniquely solvable, and the solution is precisely the key combination that created the fm equation system. Any other key combination would have created a different deltaM series and a different fm equation system with a different solution. So all 256^{Ltot} possible key combinations of fm values have exactly one solution. We will use this information later in the proof.

Breaking the encryption

An attacker wants to break the algorithm. The attacker only knows a large set of n values and their corresponding CB values. We have assumed that the attacker knows the encrypted message to a large extent and can therefore derive the corresponding Z values from the CB values. The attacker can try to determine the values of the keys and use them to create new Z values, which we will call key solving from now on. The attacker can also examine the n and Z values and try to find regularities, irregularities, statistical deviations, or other characteristics in the Z values and exploit them. We will call this utilizing dependencies. The attacker can also use both of these methods.

We can divide all attempts to break the encryption into two different sets.

Case 1: The attacker tries to break the algorithm without solving the key or any part of it.

Case 2: The attacker tries to break the algorithm by solving the key or a part of it.

Case 1 and Case 2 are negations of each other, meaning that together they form all possible ways of breaking the encryption.

Case 1

The attacker tries to break the algorithm without knowing anything about the key or any part of the key.

Since the attacker does not know anything about the key, the attacker cannot solve the key in Case 1, because if the attacker were to solve the key, we would move to Case 2.

The only thing the attacker knows in addition to the algorithm is a large set of n and corresponding Z_n values derived from CB values. The attacker can only examine Z_n values and exploit any regularities, dependencies, or other properties that may be useful to them. If the attacker were to obtain Z_n values by examining some information about the key, we would move to Case 2.

The tables used as keys contain evenly distributed, independent random data. Each Z_n is a unique combination of xor results of evenly distributed random data. Therefore, Z_n values are evenly distributed. However, statistically evenly distributed Z_n numbers may still be dependent on each other.

Since Z_n values are evenly distributed, if we cannot solve Z_n values, then all possible values of each Z_n value, 0-255, are equally likely, with a probability of 1/256.

Z values depend on fm values, key tables, and the VB table, the xor results of some of the elements in these tables. When calculating Z, the VB table keys repeat every Ltot byte of encryption. If we do not find any regularity in the fm values, we will not find any regularity in their xor (Z), even though there is regularity in the VB table.

In fm values, the same key values are used only through the g(m) function, when the table indices are multiples of the table lengths, and only these may exhibit dependencies (the table elements are completely independent of each other). If the attacker intends to exploit any dependencies that may exist in Z, then the

attacker must compare different fm values that are partially composed of the same key table elements, and the differences in the indices of the comparable fm values must be multiples of the table lengths.

If the attacker knew the differences in the m values of the comparable fm values, the attacker could determine the m values (starting from the beginning or guessing one starting value), and by knowing the m values, the attacker could determine the equations on which the fm value is calculated. With enough such equations, they could calculate the values of the key table elements, and we would no longer be in Case 1. Therefore, in Case 1, the attacker cannot know the difference in the m value of comparable fm values, nor compare such Z_n values. Therefore, we cannot compare any fm values to any other fm values and utilize table lengths in Case 1. Since the multiples of table lengths are the only places where dependencies may exist, and we cannot utilize table lengths in comparing fm values, we cannot break the algorithm in Case 1, and the claim of the theorem holds true.

Case 2

Attacker tries to break the algorithm by solving the key or a part of it.

We divide Case 2 into two sub-cases, Case 2a and Case 2b. In Case 2a, the attacker tries to solve the key or a part of it (for example, in closed form using some mathematical method) without guessing anything, obtaining certain values for at least one element of the key table. In Case 2b, the attacker guesses something. Together, these cover the entire Case 2, since 2b is the negation of 2a.

Case 2a Attacker tries to solve the key without guessing anything. The fm formula creates equations, and after several encrypted bytes, we have a system of equations from these equations. We have already shown that in our algorithm, we can solve at most one variable with one equation. Therefore, we need at least as many equations as we want to solve values in the key. Since the attacker does not know (and cannot guess in case 2a) any m value, they do not know which equations are used for each n value. Since the attacker does not know any equations, they cannot solve any variables. Complete or even partial solutions to the system of equations are not possible. Because key values are used only in equation systems, and equation systems cannot be solved in any other way, key values cannot be determined in case 2a.

The same thing can also be thought of as follows: If the key could be found, there would either be a method to solve the system of equations without knowing any equations, or the attacker would know at least one equation. Since the system of equations cannot be solved even partially without knowing any equations, the attacker should therefore know at least one equation. Equations are formed by selecting some key values depending on m values in equations. At the beginning of the encryption, the attacker knows the keys by which the first deltaM is calculated (xor plus 1 of all the first elements of the key tables), but does not know the value of deltaM and thus not the equation. After evaluating the first deltaM the attacker does not know even which key table values are used in the equations. Therefore, the attacker does not know any equations used in the algorithm.

Since the system of equations cannot be solved even partially, the attacker cannot obtain any information about the keys, nor any information about the probabilities of the keys. The attacker does not know anything else about the probabilities of key values other than that each key value is equally likely with a probability of $1/256$.

Therefore, the claim of the theorem holds true in case 2a.

Case 2b The algorithm cannot be broken by solving equation systems or in any closed form, nor can any certain information about the keys be obtained. However, an attacker can try to break the algorithm by guessing. The attacker can break the algorithm, for example, by guessing all the values in the key tables. The attacker can calculate the values in the VB table by XORing n consecutive values of known Z and guessed fm. After this, it is easy for him to solve all the Z values. The required guesses are just too many, more than the claim of the theorem.

By brute-force guessing all the key values, the attacker has to guess at least Ltot bytes, so the number of combinations is 256^{Ltot}, and on average the attacker has to make $\frac{256^{Ltot}}{2}$ guesses before finding the right key. This is greater than the claim presented in the theorem. The attacker must use other methods than brute force to reduce the number of combinations.

If the attacker guesses even one byte, how does they know if they guessed correctly? The attacker must verify the guesses because without verification, the probability of each guessed byte being correct is 1/256, and the guesses have been of no use.

When verifying guesses the attacker needs to find equations that uses only already guessed variables, and verify those equations against known Z values. Because all Z equations contain a VB element, and VB elements repeat only at every Ltot encrypted byte, if a certain VB element occurs in a certain equation, the next time the same VB element appears is only after Ltot encryptions. Then, when verifying a guess, the equation that contains the guessed value contains several other guessed values also, and all these values needs to be verified. The VB element in that equation occurs for the next time after Ltot encryptions. Therefore only after at least Ltot encryptions can all elements in that equation have occurred in some other equations, and therefore verification of guesses need to verify Z values at a minimum of Ltot encryptions apart.

The attacker first tries to guess values for the key table elements. During the first Ltot consecutive encrypted bytes (even if they starts from an arbitrary value of n), the attacker cannot calculate any values for the key tables because they does not know the values of the VB table. Once the attacker has encrypted at least Ltot consecutive bytes, and VB table values are used again, can they verify the guesses.

How many keys must the attacker guess when guessing values to keys in Ltot consecutive equations? For each byte that is encrypted they evaluate P, K, L and deltaM for a total of 4*Nt bytes, in our example with 8 tables 32 bytes. These 4*Nt bytes are selected from Nt tables, each table having a unique length, half of the tables being shorter than the median table length and half being longer. In average the probability that a certain key value is not used in an encryption is $\approx 1 - \frac{32}{Ltot}$. The probability that a certain key value is not used after Ltot encryptions is $(1 - \frac{4 * Nt}{Ltot})^{Ltot}$. The probability that a certain value is used after Ltot encryptions is $1 - (1 - \frac{4 * Nt}{Ltot})^{Ltot}$. In average we have used $Ltot * (1 - (1 - \frac{4 * Nt}{Ltot})^{Ltot})$ bytes after Ltot encryptions.

We define function b(Ltot) = $(1 - \frac{4 * Nt}{Ltot})^{Ltot}$

We take the natural logarithm on both sides:

$$ln(b(Ltot)) = ln((1 - \frac{4 * Nt}{Ltot})^{Ltot})$$
$$ln(b(Ltot)) = Ltot * ln(1 - \frac{4 * Nt}{Ltot})$$

First we calculate the Ltot derivative of $ln(1 - \frac{4 * Nt}{Ltot})$ which we need later

$$ln(1 - \frac{4 * Nt}{Ltot}) = ln(\frac{Ltot - 4 * Nt}{Ltot})$$
$$ln(\frac{Ltot - 4 * Nt}{Ltot}) = ln(Ltot - 4 * Nt) - ln(Ltot)$$
$$ln'(1 - \frac{4 * Nt}{Ltot}) = \frac{1}{Ltot - 4 * Nt} - \frac{1}{Ltot}$$
$$ln'(1 - \frac{4 * Nt}{Ltot}) = \frac{Ltot - Ltot + 4 * Nt}{Ltot * (Ltot - 4 * Nt)}$$
$$ln'(1 - \frac{4 * Nt}{Ltot}) = \frac{4 * Nt}{Ltot * (Ltot - 4 * Nt)}$$

We differentiate both sides, applying the chain rule to the left-hand side and the product rule and the chain rule to the right-hand side

$$ln'(b(Ltot)) = ln(1 - \frac{4 * Nt}{Ltot}) + Ltot * ln'(1 - \frac{4 * Nt}{Ltot})$$

$$b'(Ltot) * \frac{1}{b(Ltot)} = ln(1 - \frac{4 * Nt}{Ltot}) + Ltot * (\frac{4 * Nt}{Ltot * (Ltot - 4 * Nt)})$$
$$b'(Ltot) * \frac{1}{b(Ltot)} = ln(1 - \frac{4 * Nt}{Ltot}) + \frac{4 * Nt}{Ltot - 4 * Nt}$$
$$b'(Ltot) * \frac{1}{b(Ltot)} = ln(1 - \frac{4 * Nt}{Ltot}) + \frac{4 * Nt}{Ltot - 4 * Nt}$$

We multiply both sides with b(Ltot)

$$b'(Ltot) = b(Ltot) * (ln(1 - \frac{4 * Nt}{Ltot}) + \frac{4 * Nt}{Ltot - 4 * Nt})$$

We replace b(Ltot) with its value $(1 - \frac{4 * Nt}{Ltot})^{Ltot}$

$$b'(Ltot) = (1 - \frac{4 * Nt}{Ltot})^{Ltot} * (ln(1 - \frac{4 * Nt}{Ltot}) + \frac{4 * Nt}{Ltot - 4 * Nt})$$

All parts of the derivative are positive except $ln(1 - \frac{4 * Nt}{Ltot})$, which is a negative number when Ltot >= 793

The derivative is positive if the last part $ln(1 - \frac{4 * Nt}{Ltot}) + \frac{4 * Nt}{Ltot - 4 * Nt} >= 0$

The derivative of the last part is

$$= \frac{4 * Nt}{Ltot * (Ltot - 4 * Nt)} - \frac{4 * Nt}{(Ltot - 4 * Nt)^2}$$
$$= 4 * Nt * (\frac{1}{Ltot * (Ltot - 4 * Nt)} - \frac{1}{(Ltot - 4 * Nt)^2}$$
$$= 4 * Nt * (\frac{1}{1/Ltot^2 - 4 * Nt * Ltot} - \frac{1}{Ltot^2 - 4 * Nt * Ltot - 4 * Nt * Ltot + (4 * Nt)^2}$$

Since $Ltot^2 - 4 * Nt * Ltot$ is bigger than $Ltot^2 - 4 * Nt * Ltot - 4 * Nt * Ltot + (4 * Nt)^2$ if 4*Nt is smaller than Ltot and Ltot >= 793 (which it is since if increasing Nt with 1, Ltot increases with at least 261), the derivative of the last part is negative. Since the derivative of the last part is negative, the last part will reduce when Ltot increases. The smallest value of the last part is when Ltot is infinity, and then the value is 0. So the last part is always >=0 when Ltot >= 793.

Since the last part is positive the derivative g'(Ltot) is positive. So the biggest value of b(Ltot) is when Ltot is infinite.

When Ltot approaches infinity $\lim_{Ltot \to \infty} 1 - (1 - \frac{4 * Nt}{Ltot})^{Ltot} = 1 - \frac{1}{e^{4 * Nt}}$ [7].

So the smallest amount of used key values is when Ltot is infinity, the percentage of key values being used $100 * (1 - \frac{1}{e^{4 * Nt}})$. This gets its smallest value when $e^{4 * Nt}$ gets its smallest value. The smallest value Nt can have is 3. So the smallest percentage is $100 * (1 - \frac{1}{e^{4 * 3}}) \approx 99.9994\%$, which is more than our theorem claims (99.9%). In our example with Nt = 8 the probability would already be ≈ 99.999999999998 . So by guessing values to Ltot consecutive equations the theorem holds.

The attacker can of course guess values for keys or variables in an order other than systematically guessing consecutive equation values. If the attacker jumps exactly one equation (not guessing key values for one n), they do not know what the next m value is, so they can try and guess the value for m or deltaM. If the attacker jumps more than one value, then because each of the n values modifies Nt key values in the key tables, after just a few jumps, the attacker has likely used changed key values. Even the verification result of possibly correctly guessed keys shows that at least one guess were wrong. After the jump, the attacker can get only a few correct verification results with correct values. A few correct values may come easily even with wrong keys, so verification with non-consecutive n values becomes difficult.

If the attacker jumps exactly one value, the attacker has thus had to guess the m value (more than one byte), or the deltaM byte, so the attacker has gained nothing. If the attacker jumps over two n values, the attacker has saved guessing one deltaM byte, but the attacker has already unknowingly changed 2*Nt key values, making

verification more difficult. The more the attacker jumps n values, the more likely it is that the first verifiable byte will immediately give the wrong result even for the originally correctly guessed key values.

Next, we will examine how many values of key table elements an attacker must guess at a minimum in order for the verification to be meaningful.

In our example, let us assume that the length of our key, Ltot, is 10000 bytes, and that we have Nt = 8 tables. If the attacker jumps over n=10000/8=1250 values, then they will update the key values 1250*8=10000 times such that the attacker does not know which keys are updated and what values they are updated to. Some of the updates may have gone to table values that have already been updated, which means that the number of changed key values can be less than 10000. After updating one encrypted byte, the attacker has updated 8 key values and any specific key value is unchanged with probability $(1 - \frac{1}{10000})^8 \approx 0.99200$. After updating 1250 encrypted bytes, any specific key value is unchanged with probability $((1 - \frac{1}{10000})^8)^{1250} = (1 - \frac{1}{10000})^{10000} \approx 0.368$. When the attacker verifies the first byte, they uses 32 key values, and the probability that all of them are unchanged is $(1 - \frac{1}{10000})^{10000*32} \approx 0.0000000000000126$. This is much smaller than $\frac{1}{256} \approx 0.0039$, which is the probability of simply guessing the correct value.

The formula for the calculation mentioned above goes as follows:

If the attacker skips $\frac{Ltot}{Nt}$ bytes, then they will update $\frac{Ltot}{Nt} * Nt =$ Ltot bytes such that they do not know which key elements are updated and what values the elements are updated to. Then, none of the 4*Nt key values have changed with probability $(1 - \frac{1}{Ltot})^{Ltot*4*Nt}$.

We calculate the derivative of this with respect to Nt:

Probability that none of the key values used in verification have changed is

$$g(Nt) = (1 - \frac{1}{Ltot})^{Ltot*4*Nt}$$

The formula for the derivative of a composite function $a^{g(x)}$ is $a^{g(x)} * \ln(a) * g'(x)$ [5]

$$g'(Nt) = (1 - \frac{1}{Ltot})^{Ltot*4*Nt} * \ln(1 - \frac{1}{Ltot}) * 4 * Ltot$$

Since Ltot is at least 793, $1 - \frac{1}{Ltot}$ is greater than zero and less than one, i.e., a positive number, and this positive number to the power of a positive number is a positive number. This times 4*Ltot is also a positive number. The natural logarithm between zero and one is a negative number, so the whole derivative is always negative. Since the derivative is negative, the probability that none of the key values have changed is smaller the larger Nt is. The highest probability that none of the key values have changed is when Nt is the smallest possible, which is 3.

We also calculate the derivative with respect to Ltot. The probability that none of the key values have changed = g(Ltot)

$$g(Ltot) = (1 - \frac{1}{Ltot})^{Ltot*4*Nt}$$

We take the natural logarithm on both sides:

$$\ln(g(Ltot)) = \ln((1 - \frac{1}{Ltot})^{Ltot*4*Nt})$$
$$\ln(g(Ltot)) = 4 * Nt * Ltot * \ln(1 - \frac{1}{Ltot})$$

We calculate first ln'(1-1/Ltot) which we need later

$$\ln(1-\frac{1}{Ltot}) = \ln(\frac{Ltot-1}{Ltot})$$
$$\ln(\frac{Ltot-1}{Ltot}) = \ln(Ltot-1) - \ln(Ltot)$$
$$\ln'(1-\frac{1}{Ltot}) = \frac{1}{Ltot-1} - \frac{1}{Ltot}$$
$$\ln'(1-\frac{1}{Ltot}) = \frac{1}{Ltot*(Ltot-1)}$$

We differentiate both sides, applying the chain rule to the left-hand side and the product rule and the chain rule to the right-hand side

$$g'(Ltot)*\frac{1}{g(Lgot)} = 4*Nt*(\ln(1-\frac{1}{Ltot}) + Ltot*\frac{1}{Ltot*(Ltot-1)})$$
$$g'(Ltot)*\frac{1}{g(Lgot)} = 4*Nt*(\ln(1-\frac{1}{Ltot}) + \frac{1}{Ltot-1})$$

we multiply both sides with the g(Ltot) value

$$g'(Ltot) = g(Ltot)*4*Nt*(\ln(1-\frac{1}{Ltot}) + \frac{1}{Ltot-1})$$

We replace g(Ltot) with its value $(1-\frac{1}{Ltot})^{Ltot*4*Nt}$

$$g'(Ltot) = (1-\frac{1}{Ltot})^{Ltot*4*Nt}*4*Nt*(\ln(1-\frac{1}{Ltot}) + \frac{1}{Ltot-1})$$

All parts of the derivative are positive except $ln(1-\frac{1}{Ltot})$, which is a negative number when Ltot >= 793

The derivative is positive if the last part $\ln(1-\frac{1}{Ltot}) + \frac{1}{Ltot-1} >= 0$

The derivative of the last part is $\frac{Ltot}{Ltot-1} - \frac{1}{(Ltot-1)^2}$

which is $\frac{1}{Ltot^2-Ltot} - \frac{1}{Ltot^2-Ltot-Ltot+1}$

Since $Ltot^2-Ltot$ is bigger than $Ltot^2-Ltot-Ltot-1$ when Ltot >= 793, the derivative of the last part is negative when Ltot >= 793

The value of the last part when Ltot = 793 is ≈ 0.0000008

Since the derivative of the last part is negative when Ltot >= 793, the last part will reduce when Ltot increases. The smallest value of the last part is when Ltot is infinity, and then the value is 0. So the last part is always >=0 when Ltot >= 793.

Since the last part is positive the derivative g'(Ltot) is positive. So the biggest value of g(Ltot) is when Ltot is infinite.

$(1-\frac{1}{Ltot})^{Ltot*4*Nt} = (1-\frac{1}{Ltot})^{Ltot}$ to the power of 4 * Nt

$(1-\frac{1}{Ltot})^{Ltot}$ when Ltot approaches infinity is 1/e [7]. Then $(1-\frac{1}{Ltot})^{Ltot*4*Nt}$ is $(\frac{1}{e})^{4*Nt}$ when Ltot approaches infinity. 1/e $\approx$ 0.367. The probability that no key value has changed, regardless the Ltot value, is thus smaller than 0.37^{4*Nt}

The highest probability that none of the values have changed when we jump over $\frac{Ltot}{Nt}$ formula, is when Nt is the smallest possible. Nt = 3 gives the highest upper bound of probability, which is $\approx$ 0.000006144, much smaller than 1/256 ($\approx$0.0039). Therefore, the probability for all Nt and Ltot values is significantly less than 1/256. In our example, Nt = 8 and smallest Ltot with 8 tables 2245 would give an upper bound of probability $\approx$ 0.00000000000000126.

Since there is only one correct solution, if the attacker has guessed correctly and has jumped over at least $\frac{Ltot}{Nt}$ formulas, the verification result is likely to fail, but the attacker still cannot reject this (only correct) guess, otherwise they will not find this only correct solution. Therefore, the attacker must accept all guesses that may also provide the wrong solution. The attacker must also accept as correct those guesses that provide the correct value when verified. The result of verification is that all guesses can be correct. If there were many correct solutions, the attacker might be able to infer something statistically, but since there is only one correct solution, the attacker cannot reject their guess regardless of whether the verification result is correct or incorrect. If the attacker jumps more than $\frac{Ltot}{Nt}$ bytes, verification is therefore impossible.

How many key table values has the attacker used on average if they has jumped over the $\frac{Ltot}{Nt}$ formulas? If we jump over 1250 bytes in our example, the probability that we do not use any particular key table value when encrypting one byte is $(1 - \frac{1}{10000})^{32} \approx 0.968$. The probability that we do not use one selected key value when encrypting 10000-1250 bytes is $(1 - \frac{1}{10000})^{32*8750} \approx 0.0000000000006903$.

The probability that we use the selected key value when encrypting 8750 bytes $\approx 0.9999999999993096$.

Then the probability that each byte has been used is $\approx 0.9999999999993096$, so on average we have used $\approx 0.9999999999993096$*10000 $>$ 9999.99999999 key values. Therefore, if we have jumped over 1250 bytes, according to the theorem, we have had to guess more than 9999 key values on average.

The formula is as follows: If we jump over $\frac{Ltot}{Nt}$ bytes, the probability that we do not use any particular selected key value when encrypting one byte is $(1 - \frac{1}{Ltot})^{4*Nt}$. The probability that we do not use one selected key value when encrypting $Ltot - \frac{Ltot}{Nt}$ bytes is $(1 - \frac{1}{Ltot})^{4*Nt*(Ltot-Ltot/Nt)}$

The probability that each byte has been used is the same, so on average we have used amount of key values

$$= (1 - (1 - \frac{1}{Ltot})^{4*Nt(Ltot-Ltot/Nt)}) * Ltot$$
$$= (1 - (1 - \frac{1}{Ltot})^{4*Ltot*(Nt-1)}) * Ltot$$

The smaller the Nt, the fewer key values are used. $\lim_{x\to\infty}(1 - \frac{1}{x})^x = \frac{1}{e}$ [7] as x approaches infinity. The smallest possible Nt is 3. Using Nt = 3, 4*(3-1) is 8, and as Ltot approaches infinity, the average number of key values used is $(1 - (\frac{1}{e})^8) * Ltot \approx 0.9996645$ * Ltot

If the attacker jumps more than $\frac{Ltot}{Nt}$ bytes and Nt >= 3, then the attacker has to guess more than $256^{Ltot*0.999}$ bytes, which is our theorem's claim.

Case 2b, other guesses The attacker can try to guess values into the VB table. In this case, they get as many equations as they guessed values into the VB table, and they can potentially solve at most as many key table values as the number of guesses made into the VB table. The attacker does not gain any additional information compared to guessing values directly into the key tables and solving VB values. They only get more difficult equations and can only potentially solve the same number of key values as if they had guessed directly into the key tables. The verification requirement is the same as if they had guessed into the key tables. The theorem's claim holds because the attacker cannot solve more key values than the number of guesses made into the VB table.

The attacker can also guess values to other symbols than the key tables, or guess values to both key values and other symbols. Symbols in the definition of the algorithm are m, deltaM, K, L, P, Z and fm, but the attacker can of course define any new symbol, e.g. as some combination of these.

By guessing values to other symbols than the key tables, these guesses has also to be verified. If key table values cannot be inferred from the guesses, then verifying the guesses is impossible since the key values (key table values and VB values) are the only values that are recurring, all other symbols have unique values for each n (within the sequence length). The only symbols from which key table values can be inferred from are such that the attacker can infer the equations of key values, the equations are the only place where the key values

are used and thus can be inferred from. Such symbols in the definition of the algorithm are m and deltaM, K, L and P, and a combination of these fm and Z.

By guessing values to m (or consecutive deltaM values), K, L or P the attacker gets one equation per guess that can solve at most one variable. The attacker gains nothing compared to guessing values directly to the key tables. But if the attacker guesses values to all of these, then the attacker can get with 4 guesses 5 equations since the Z value is known. However this last equation contains only the same key table values that existed in the K, L and P equations, so the 5^{th} equation does not solve any new key table values. The VB table value in the 5^{th} equation exist in a similar way in all verification equations so it does not give any new information to the attacker either.

We have now discussed all possible combinations of keys and symbols that the attacker can guess values for. The theorem's claim holds true in Case 2b.

8 SUMMARY

We have now gone through all the sub-cases of Case 2, and in all of them, the theorem has held true. The sub-cases of Case 2 cover all possible ways of breaking the algorithm. Therefore, in Case 2, the theorem holds true.

We have now gone through Case 1 and Case 2, which together make up all possible ways of breaking the algorithm, and in both cases, the theorem has held true. Hence, the theorem is proven. Q.E.D.

There is no upper limit on the length of the key tables or the number of tables, and therefore there is no upper limit on the value of Ltot. The average number of trials required to break the algorithm is at least $\frac{256^{Ltot*0.999}}{2}$. By increasing the value of Ltot, the average time required to break the algorithm can be made greater than any arbitrary value. Q.E.D.

9 REFERENCES

1. *Shannon, Claude (1949). Communication Theory of Secrecy Systems (PDF). Bell System Technical Journal.* ***28*** *(4): 656–715. Search engine search term 'one-time pad' gives a lot of good references.*

2. *Gerhard J. Woeginger. "The P-versus-NP page". Retrieved 13 May 2023.*

3. *Burden, Richard L.; Faires, J. Douglas (1993), Numerical Analysis (5th ed.), Boston: Prindle, Weber and Schmidt, ISBN 0534932193. Search term 'equation system'*

4. *Fortnow, Lance (2013). The Golden Ticket: P, NP, and the Search for the Impossible. Princeton, NJ: Princeton University Press. ISBN 9780691156491. Search term 'P vs NP' or 'P = NP'*

5. *George F. Simmons, Calculus with Analytic Geometry (1985), p. 93*

6. *Timothy Gowers; June Barrow-Green; Imre Leader (8 September 2008). The Princeton Companion to Mathematics. Princeton University Press. p. 607. ISBN978-0-691-11880-2.*

7. *Wilson, Robinn (2018). Euler's Pioneering Equation: The most beautiful theorem in mathematics (illustrated ed.). Oxford University Press. p. (preface). ISBN 9780192514059. Search term '(1-1/x)^x'*

8. *Turing (1954), "Solvable and Unsolvable Problems", Science News, (February, Penguin), 31: 7–23.*

9. *Razborov, Alexander A.; Steven Rudich (1997). Natural Proofs. Journal of Computer and System Sciences.* ***55*** *(1): 24–35.*

10. *T. P. Baker; J. Gill; R. Solovay (1975). "Relativizations of the P =? NP Question". SIAM Journal on Computing. 4 (4): 431–442.*

11. *S. Aaronson; A. Wigderson (2008). Algerization: A New Barrier in Complexity Theory. Proceedings of ACM STOC 2008. pp. 731–740.*

10 APPENDIX 1: Sequence Length

By sequence length, we mean the maximum number of bytes that can be encrypted with the algorithm, that is, when the algorithm starts repeating the same Z values as earlier. We cannot encrypt a message longer than the sequence length, otherwise the encryption can be broken.

Let us first consider moving through the fm value and the g() function according to the number n (rather than m).

If we have only one key table, then the sequence length is equal to the length of this table. In this case, the algorithm is unbreakable but impractical, as it is a 'one-time pad' (OTP).

If we have two tables, whose greatest common divisor of lengths is 1, then the smallest common multiple of the table lengths is the product of the lengths. The sequence length is then the smallest common multiple of the table lengths, so the sequence length will be the product of the table lengths.

Whenever we add a table whose greatest common divisor of length with all the other tables is 1, this table will increase the smallest common multiple of the table lengths by the length of the table, i.e., increasing the sequence length by the length of the table.

If all the table lengths are prime numbers, then the greatest common divisor of any two tables' lengths is 1, and the smallest common multiple, or the sequence length, is the product of the table lengths.

The smallest common divisor of a prime number and any number (that is not a multiple of the prime number in question) is 1. If the table lengths are prime numbers except for one table, and this table is not a multiple of any other table length, then the smallest common multiple, or the sequence length, will be the product of the table lengths.

Now, m increases at most 256+4=260 times faster than n (which is why the table lengths are greater than 260; if the tables could be shorter, the probabilities of occurrence for all table elements would not be equal in the formulas). The sequence length is then at least the product of the key table lengths / 260. On average, m increases $\frac{256}{2} + 4 = 132$ times faster than n, but for safety, we can use the worst-case scenario as the estimate for the sequence length, where we divide the product of the table lengths by 260.

We are modifying the values of the keys as the encryption progresses. Soon, all positions of the elements will start to contain XOR combinations of all original key values. Some of the original key values appear an even number of times, while others appear an odd number of times. Key values that appear an even number of times can be removed, and key values that appear an odd number of times can be replaced with a single occurrence. As a result, each key value will soon have one of 2^{Ltot} possible combinations of original key values. For example, with eight tables of the smallest possible lengths, the n-sequence is slightly over $3 * 10^{21}$, Ltot is 2245, and 2^{2245} is approximately 10^{676}, which is much larger than the n-sequence. Therefore, the probability that any combination of the original key values will occur multiple times during the n-sequence is very low. Even if it does happen, utilizing this information would require knowing where these same combinations occur, and we have already shown that without knowledge of the key, the attacker cannot know which key values are being used at any given time, so they cannot know where and when the same combinations occur.

Although we change the values of the keys as the encryption progresses, we do not change the lengths of the tables or the positions of the elements. However, when the sequence ends and we start a new sequence, the same g(m) values will not repeat because we have modified the key values. The probability that all key values have changed since the beginning of the encryption is very high. We can repeat the sequence again and again countless times without g(m) repeating itself. Each time we start a new sequence, the probability increases that the same key values have occurred at the beginning of some previous sequence. We can repeat the sequence a maximum of 256^{Ltot} times (when we have encrypted at least the product of the lengths of the key tables

/ 260 * 256^{Ltot} bytes), then the combination of keys will have definitely occurred earlier. Therefore, we can encrypt at least the product of the lengths of the key tables / 260 bytes securely, after which the probability of sequence repetition slowly increases. For example, with eight tables of the smallest possible lengths, we can securely encrypt more than 3 * 10^{21} bytes (which is 300,000,000 terabytes, three hundred million terabytes, not easily achievable with current devices). If necessary, we can of course increase the lengths of the tables, or the amount of tables, to make the sequence arbitrarily long.

11 APPENDIX 2: P vs NP

P vs NP [4] is a well-known mathematical problem that has not yet been solved. We will show that the WESP algorithm proves that P is not equal to NP.

In the P vs NP problem, P is the set of problems that can be solved in polynomial time, and NP is the set of problems that can be verified in polynomial time. The question is whether P is the same set as NP, that is, whether all problems that can be verified in polynomial time can also be solved in polynomial time.

In the case of P vs NP we set the problem to be: we fill the encryption keys with completely random data and generate the first 10*Ltot bytes of Z_n using this secret key. We give these generated Z_n bytes to an attacker and ask the attacker to present a key that generates such a sequence of Z_n bytes.

The key presented by the attacker does not have to be the same key as we generated. There is a theoretical very small probability, that some key elements have not yet been used after 10*Ltot encryptions, and if so, there will exist multiple keys generating the same sequence. This if fine, the proof of the theorem takes this into consideration.

Verification is easy. We use the key presented by the attacker and generate 10*Ltot Z_n values, and check if they are correct. This can be done in polynomial time, so the problem belongs to the NP set.

If no definite errors are found in the proof of the theorem then the breaking of the WESP encryption algorithm requires an exponential amount of time. Therefore, the WESP algorithm does not belong to the P set. Since there is at least one algorithm that is not P but is NP, P and NP are not the same set. Q.E.D