WESP: An encryption method that, as the key size increases, require an exponentially growing time to break

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

Abstract. WESP is a new encryption algorithm that is based on equation systems, in which the equations are generated using the values of tables that act as the encryption key, and the equations having features making them suitable for cryptographic use. The algorithm is defined, and its properties are discussed. Besides just describing the algorithm, also reasons are presented why the algorithm works the way it works. The key size in WESP can be altered and has no upper limit, and typically the key size is bigger than currently commonly used keys.

A calculation is presented, calculating how many bytes can be securely encrypted before the algorithm might start to repeat it's sequence of encrypting bytes, and that this period can be adjusted to be arbitrarily large.

It is also shown that withing the period the resulting stream of encrypting bytes is statistically uniformly distributed.

It is also 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, verification 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.

WESP's resilience against some common stream cipher attacks is also discussed.

Keywords: Symmetric • Synchronized • Stream Cipher • Equation system • Exponential • Wesp

1 INTRODUCTION

The WESP algorithm is new, and is not based on any previous cipher.

WESP is based on equation systems. Many problems can be expressed as equation systems, but the equations used in WESP are generated in such a way, that without knowledge of the keys the equations cannot be solved, and this has important consequences.

In WESP the key size can be arbitrarily large, typically from some kilobytes to some hundred kilobytes. Having a big key size has some advantages in WESP. Having a big key is in most cases no problem.

WESP is guaranteed to generate a unique, non-repeating sequence of encrypting bytes within a period. The period length can be adjusted and is typically long enough to not be any problem.

The sequence of encrypting bytes is also guaranteed to have values that are statistically uniformly distributed.

In addition to traditional cryptanalysis, where we discuss various attack methods and how they apply to WESP, we present a proof of how much work is needed to break the cipher.

The target of the WESP algorithm is not that it can be used anywhere, or that it would become a standard. The target of WESP is that many applications can use it in many environments, and that application security will increase. WESP provides proven security.

This paper is intended to be understood by anyone interested in cryptology and having basic knowledge of mathematics.

2 DEFINITION OF THE WESP ALGORITHM

WESP is a stream cipher that encrypts a message one byte at a time. WESP is a synchronized stream cipher i.e. the encrypting 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, both the sender and the receiver must have the same secret key available.

As with synchronized stream ciphers in general, WESP will not note if the encrypted message is altered by a third party, and thus some additional precautions against this should be made, e.g. by using message authenticating codes.

In WESP the same key should be used only once for one stream of encrypted bytes. For another stream a new, unrelated key must be used.

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. Here we describe 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 also 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.

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 introduce the algorithm as pseudo-code, it is the simplest way to describe how WESP works, and gives an exact definition for the algorithm. After the pseudo-code, we go through the code and explain it verbally. Why things are done the way they are done is then described.

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 $\oplus$= Y means that variable X gets the value X $\oplus$ Y.

mod is the modulo operation.

// means that the rest of the line is a comment.

The integer constant Lmultiplier is defined later.

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 pseudo-code, 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, one was added so that no equation would never be used again). 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.

A key of length $2 * Ltot$ have 256^{2*Ltot} possible key combinations, and each combination is equally probable. E.g. a 'weak' key where all values are zero is equally probable than any other key. A key where all values are zero (even if the probability that such a key would ever be used is very low) would result in all Z values being 0. Or a weak key where all value's most significant bit would be zero would result in Z values with all the most significant bits zero (and such a key could easily exist if e.g. a signed/unsigned bug would exist in an implementation of WESP or random number generator usage). Therefore the L and K values are also XORed with n mod 256 (a 'weak key correction'). XORing with n mod 256 ensures that all L and K values contain all values between 0 and 255, regardless of special case weak keys. XORing the L and K values with n mod 256 ensures also that the Z_n values are statistically uniformly distributed for all possible key combinations, also for weak keys (even if the probability of a random key being such a weak key is very low). 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 but does alter how m progresses and how key values are updated.

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). We will see that the number of bits used is important because 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 uniformly 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 tables then often the Lmultiplier is 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 uniformly distributed among all table elements, any number that is uniformly distributed plus any number that can be longer than the length of the tables (mod the table length) can statistically point to any element.

3 PROPERTIES OF WESP

Period

By period we mean the maximum number of bytes that can be encrypted with the algorithm. After the period the algorithm might start repeating the same Z values as earlier. We must not encrypt a message longer than the period, otherwise the encryption can potentially be broken. The same Zn value must never be used more than one time, therefore the same key must never be used twice with the same n value (we can of course use the same key for multiple messages as long as we continue from the previous n value and key state).

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 period is equal to the length of this table. In this case, the algorithm is unbreakable but often impractical, as it is a 'one-time pad' (OTP) [17].

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 period is then the smallest common multiple of the table lengths, so the period 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 period by the length of the added 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, the period, 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, the period, 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 period 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 period, 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 with the smallest possible lengths, the period is slightly over 10^{18}, Ltot is 2198, and 2^{2198} is approximately 10^{662}, which is much larger than the period. Therefore, the probability that any combination of the original key values will occur multiple times during the period is very low. Even if it would happen, utilizing this information would require knowing where these same combinations occurs, 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 (he/she) 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 define prod as the product of the lengths of the key tables. We can repeat the sequence a maximum of 256^{Ltot} times (when we have encrypted at least $\frac{prod}{260} * 256^{Ltot}$ bytes), then the combination of keys will have definitely occurred earlier. Therefore, we can encrypt at least the $\frac{prod}{260}$ bytes securely, after which the probability of sequence repetition slowly increases. For example, with eight tables with the smallest possible lengths, we can securely encrypt more than $3*10^{21}$ bytes, 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 period arbitrarily big.

Possible implementations of WESP would naturally have to use big number mathematics to avoid integer overflow errors, or use other methods like replacing modulus calculations with increments and comparisons, to

avoid these kind of errors.

Random key data

We assume that the values of the elements in the WESP key 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 paper, but we assume that they can be generated.

Using multiple random generators and XORing their result helps in generating better random values for an implementation of WESP, if one of the used random generators would have some bias in the generated value distribution the other random generators compensate for this. At least one of the random generators should be cryptographically strong (generating values that cannot easily be guessed).

WESP itself generates a sequence of uniformly distributed Z values that do not repeat within the period, and can be used as a random generator. Another WESP key, where the key values have been generated with a cryptographically strong random generator, can be used in conjunction with other random generators to generate a random key for WESP.

The WESP algorithm is not very sensitive to biased random data in it's key. Even if the key values would have some bias, or the values would be dependent on each other in some way, since the key values are frequently updated when new Z values are calculated, and each Z value depends on many key values, the result is much better uniformly distributed than the key values itself. The more Z values are calculated the more uniformly distributed the Z values become. If we suspect that the key is not very random, we could generate e.g. Ltot fm values at start, before evaluating the first Z value, and the resulted Z values would be more uniformly distributed right from the start.

Key size comparisons

Most time while encrypting a byte is spent in the g(x) function and in updating the K value to key table elements. Both of these becomes slower the bigger N is, i.e. when the amount of tables increases. The length of the tables do not greatly affect the encryption speed, so if speed is important, reducing the amount of tables helps, and if a long period is needed, then the length of the tables can be increased.

Speed comparisons

Speed was measured in the same computer with similar data, without startup initialization, with a pure-Java implementation. Java implementations of WESP suffers from Java's slow table index references (every reference is checked for array index out of bounds problems, and no pointers can be used).

Algorithm	mb/s
WESP, 8 tables	69
WESP, 4 tables	135
AES	387
ChaCha	510
No encryption, plaintext	20000

WESP can be a bit slower than many current ciphers. In most cases this does not matter, almost always cipher security is much more important than efficiency. WESP is anyway fast enough for most purposes.

Key size properties

Next a table showing the length of the key Ltot (in bytes) and the period for different example key sizes, using a key where the table key lengths are the shortest possible tables starting from the 'Smallest table length'. The Period is in bytes. The 'Trials needed' column shows the average amount of necessary trials needed to break the algorithm (as shown in chapter 'Proof'), rounded to the closest order of magnitude.

Tables	Smallest table length	Ltot	Period	Trials needed
4	261	1064	$2 * 10^7$	10^{2557}
4	2000	8031	$6 * 10^{10}$	10^{19318}
4	20000	80055	$6 * 10^{14}$	10^{192596}
8	261	2198	$1 * 10^{17}$	10^{5285}
8	2000	16179	$1 * 10^{24}$	10^{38921}
8	20000	160245	$1 * 10^{32}$	10^{385520}

The key size in WESP is relatively large, bigger than in most current ciphers. Today this has in most cases little or no importance, security is more important.

Randomness test results

An implementation of the WESP algorithm has been tested with the TestU01 [13] statistical test library. A Z_n sequence was run against the 'big crush', the biggest test suite provided by the TestU01. The result was 'All tests passed'. This does not imply that an implementation of WESP would be secure, but gives some confidence that no obvious errors exist in neither the algorithm nor its implementation.

4 WESP SECURITY

An attacker wants to break the encryption. The attacker can e.g. eavesdrop on encrypted communication as much as they (he/she) want. 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. The attacker can freely select the plaintext to be encrypted ('chosen text attack'), and as long plaintext as they want (as long as it is shorter than the period).

Attack methods for symmetric stream ciphers

Algebraic attack [2] and Cube attack [7]

These attacks can be used in stream ciphers that uses LFSRs (Linear Feedback Shift Registers). WESP do no use LFSRs so these attacks cannot be used.

Boomerang attack [4]

In these attacks the attacker studies how differences in the plaintext affects the encrypted stream. As we have noted, in WESP the plaintext does not affect at all the generated Z stream, and since the Z stream does not repeat within the period, studying different plaintext input will not help in breaking WESP.

Brute force attack

In a brute force attack the attacker tries all possible key combinations until a working key is found. WESP can easily be broken by a brute force attack, but since the key is big, this will take a too long time. The brute force attack is discussed in detail later in the 'Proof' chapter.

Chosen-IV attack [1]

Wesp do not use Initialization Vectors (IV), so this attack cannot be used.

Differential attacks [11]

As with the Boomerang attack, in WESP the plaintext do not affect how the Z stream is generated, and therefore none of the several differential attacks [9] [10] [16] will work.

Integral attack [12]

Integral attacks works like differential attacks, although in a different way that differential attacks do. As with differential attacks, as the plaintext do not affect the Z stream, integral attacks cannot be used with WESP.

Linear attack [14]

In a linear attack the attacker exploits algorithm vulnerabilities where XOR sums of plaintext, cipher text and key values are not statistically uniformly distributed. In WESP the g(x) function will generate a result (for all x values over all key combinations) where the probability of each bit having the value 1 is exactly 0.5. For each evaluated Z value the g(x) function is called 4 times with different x values, and XOR summed with VB (which, like the g(x) function, will have an equal amount of ones and zeros in each VB bit, when all VB key values are considered). In the linear attack a 'bias' is calculated, and in order for the attack to succeed this bias must differ as much as possible from 0.5. In WESP this bias will be in average exactly 0.5, for all intermediate calculations it can be calculated from, and all kinds of linear attacks will fail.

Mod-n attack [8]

As we will see WESP produces a uniformly distributed series of Z values. A mod-n attack could only be used if some bias would exist in the Z values. Therefore any mod-n attack cannot be used against WESP.

Retrospective attack [20]

In retrospective attacks (also known as 'Harvest Now, Decrypt Later') the attacker only record the encrypted stream in the hope that in the future, when computer technology has evolved, the encryption can be broken. Here the big key size of WESP helps, and the fact that there is no upper limit in the WESP key size. It is good to have a so big key that in no foreseeable future can it be broken. If in doubt, do not double the key size, increase it by several decades.

Side channel attacks [15]

In side channel attacks the attacker does not try to break the algorithm itself, but the implementation of the algorithm. While a computer encrypts or decrypts data it will use various amounts of current, emit electromagnetic radiation etc. By studying these the attacker can get some insight on what is going on in the computer, what kind of instructions it is performing, how caches are used etc, and possibly even figure out the key.

Usually side channel attacks usually require physical presence close to the computer encrypting or decrypting, so side channel attacks can rarely be used by just listening on network traffic. The practical dangers of side channel attacks are mostly low.

The WESP algorithm cannot be attacked by side channel attacks, but implementations of WESP can be. We do not know how the algorithm will be implemented in actual implementations, so we cannot discuss here how sensitive the implementation will be, but it is clear that the implementations will give some insight to side channel attackers. Big keys will make WESP vulnerable to e.g. side channel cache attacks.

In the computer implementation of the algorithm there may also be errors that can be used to break the encryption, e.g. an implementation might have integer overflow errors. In this paper we focus on evaluating the algorithm itself and not on the characteristics of possible implementations.

Time Memory trade off attacks [3]

In Time memory trade off attacks the attacker usually tries to reduce the time required to break a cipher by calculating in advance some tables or results, and using these precomputed values to reduce the time needed to break an individual key. In Time Memory trade off attacks for stream ciphers the product of time and memory is typically constant. As we will see the time to break WESP is huge, so by decreasing the time to a practical value would mean increasing the memory requirement so much that big enough memories do not exist, and will not exist in a very long time.

Attacks against RC4

The RC4 cipher is another synchronized stream cipher that uses tables, although in a completely different way than WESP. Multiple attacks against RC4 exist [19]. The attacks against RC4 exploit the fact that the generated keystream does reveal something about the key, particularly in the beginning of the keystream. In WESP the keystream is uniformly distributed right from the start, each Z value is generated using a unique combination of key table values, even the first Z values has used key table combinations that will never occur again. The known attacks against RC4 will not work against WESP.

Summary of attack methods

None of the common attack methods will work against the WESP algorithm.

Theorem

Common attack methods did not work with WESP. Next we present two theorems, and a proof for them.

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 $\frac{256^{Ltot*0.999}}{2}$

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 (both x and y are zero) 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 XOR summed together form a certain value." If a variable changes, some or several other variables change in the accordingly. 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 keys

A key populated with random data will generate a sequence of Z values that do not repeat within the period. Can several different keys generate the same sequence of Z values?

Let us create a key and fill the key with random data that we know. Let us start evaluating manually fm values, taking the VB table into account later. As we calculate the fm values, we write down the equations that we use. Sooner or later we will have enough equations to solve the created equation system fully, resulting in the key we have been using. This is because [6]

- Each equation is unique. Since we update key values as we progress evaluating fm values, even if each equation is unique, some equations can possibly be reduced to some other equation used, even if this is extremely rare. We can simply ignore an equation that existed earlier and evaluate more fm values
- Sooner or later each key value has been used in the equations. Each time a fm value is calculated a key value is used with some probability. Even when the full period length fm values have been evaluated it is not certain that all key values have been used, but the probability that after evaluating the period length fm values not all keys are used is negligible. If a value would not have been used then that value would not take part in the fm sequence and we would have multiple keys (256 different keys where the one byte that is not used could have any value). In average all key values have been used well before Ltot fm evaluations. As shown in Appendix 3, the smallest key with 8 tables is using all key values already after Ltot evaluations with the probability $\approx 1 - 10^{-14}$(=99.999999999999%).
- If some key value would always occur together with some other key value then no distinction between these two key values could be done, and those key values could not be solved, only the xor of them would be known. But when adding equations, sooner or later such pairs would cease to exist because new combinations of key values are constantly created
- If we have Ltot key values then at least Ltot equations are needed, but we can evaluate more equations, as many equations as we need
- There will exist at least one solution since the key exist (and we know the key)

When we have enough equations we can solve the equation system for example using Gauss's [6] method. As the result we will get the key that we already knew. Since we could uniquely solve the equation system, the equation system had only one solution. Since there is only one solution then only one key can generate such a stream of fm values.

We can repeat this for all possible key combinations. There exists 256^{Ltot} different keys (excluding the VB table), and we can solve each key uniquely in the manner presented, each key having a unique stream of fm values, i.e. each key will create a different stream of fm values.

We can then add the VB table. We populate the VB table with any values, and we can still solve the Z equations similar to the fm equations. We would need at least $2 * Ltot$ equations to solve the equation system, but the result is the same, each key generates a unique stream of Z values, and each stream of Z values has only one key that can generate the stream.

So the number of keys that can generate a long enough stream of Z values is one.

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. 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 uniformly distributed, independent random data. Each Z_n is a unique combination of xor results of uniformly distributed random data. Therefore, Z_n values are uniformly distributed. However, statistically uniformly distributed Z_n numbers may still be dependent on each other.

Since Z_n values are uniformly 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 $1/256$, therefore no statistical methods can be used to break the algorithm.

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 (since 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 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? Appendix 1 presents a calculation where the result is that at least 99.9994% of the keys needs to be guessed, which is more than our theorem claims (99.9%). In our example with Nt = 8 the probability would already be $\approx$ 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.

Appendix 2 presents a calculation of the minimum amount of how many key values an attacker must guess in order for the verification to be meaningful. The result is that at least 99.96645% of the key values must be guessed. 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 to other symbols than the key tables. 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 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 those or some derivative value of those.

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 period). 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.

Summary of proof

We have now gone through all the sub-cases of Case 2, and in all of them, the theorem has held true. 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.

5 APPENDIX 1: Amount of guesses in consecutive equations

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 $\lim_{Ltot \to \infty} 1 - (1 - \frac{4 * Nt}{Ltot})^{Ltot}$ is $1 - \frac{1}{e^{4 * Nt}}$ [21].

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 $100 * (1 - \frac{1}{e^{4 * 3}}) \approx 99.9994\%$.

6 APPENDIX 2: Minimum amount of how many key table element values an attacker must guess 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 use 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 $a^{g(x)} * ln(a) * g'(x)$ [18]

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

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:

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

First we calculate 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 [21]. Then $(1 - \frac{1}{Ltot})^{Ltot*4*Nt}$ is $(\frac{1}{e})^{4*Nt}$ when Ltot approaches infinity. 1/e ≈ 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 ≈ 0.000006144, much smaller than 1/256 (≈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 2198 would give an upper bound of probability ≈ 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 ≈ 0.9999999999993096.

Then the probability that each byte has been used is ≈ 0.9999999999993096, so on average we have used ≈ 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. $\lim_{x \to \infty} (1 - \frac{1}{x})^x = \frac{1}{e}$ [21] 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.

7 APPENDIX 3: P vs NP

P vs NP [5] [22] is a well-known mathematical problem that has not yet been solved. We will demonstrate that if no errors are found in the WESP proof, then the WESP algorithm proves that P is not equal to NP.

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

All problems that are NP can be solved in the 'best case' scenario in polynomial time. This is because if a problem is verifiable in polynomial time, it has a certain number of unknown variables for which values can be assigned. The problem solver can then guess the values for all these variables, and with good luck from the solver's perspective, the first guess could be correct. Thus, in the 'best case' scenario, all NP problems are also P problems, and the P vs NP problem would not exist as an issue since P and NP would always be the same set. Therefore in the P vs NP problem, we refer to 'non-best' scenarios for guesses, such as the average or worst-case number of guesses.

Verifiability in P vs NP

In the proof of our theorem, we stated for the required number of trials, that the equation system can be solved sooner or later. An encryption algorithm doesn't need to be easily verifiable, and in the presented proof it suffices that the equations can be eventually solved uniquely. However, in the P vs NP problem, we must demonstrate that this 'sooner or later' is within polynomial time.

When verifying guesses, Z_n values has to be verified until all key values has been used at least once, only after that is it possible to verify the guesses. This happens earliest after Ltot Z values, when all VB tables values have been used once. In a key consisting of Ltot bytes (excluding the VB table), when calculating one fm value, $4 * Nt$ key values are used. The probability that a key value is not used during calculation of one fm value is $\frac{Ltot - 4 * Nt}{Ltot}$, and the probability that a certain key value has not used after Ltot fm values is $(\frac{Ltot - 4 * Nt}{Ltot})^{Ltot}$, and $1 - (\frac{Ltot - 4 * Nt}{Ltot})^{Ltot}$ is the probability that all keys values has been used when evaluating Ltot consecutive Z values. Lets calculate how this behaves when Ltot grows. We call $4 * Nt$ for k.

$$
\begin{aligned}
f(x) &= (1 - kx)^x \\
ln(f(x)) &= ln((1 - kx)^x) \\
ln(f(x)) &= x * ln(1 - kx)
\end{aligned}
$$

When x approaches infinity $(1 - kx)$ approaches negative infinity (when $k > 0$) and $ln(1 - kx)$ approaches negative infinity, and $x * ln(1 - kx)$ approaches negative infinity as well. So $ln(f(x))$ approaches negative infinity.

$e^{ln(fx)} = e^{-\infty}$ when x approaches infinity, so $f(x) = 0$ when x approaches infinity. So the bigger Ltot is, the closer to $1 - 0$ the probability (that all key values have been used) grows.

When Nt grows closer to Ltot (Nt cannot be bigger than Ltot) then the probability $1 - (\frac{Ltot - 4 * Nt}{Ltot})^{Ltot}$ grows also. The smallest probability is when Nt is the smallest possible and Ltot is the smallest possible.

With the smallest possible key Nt=3 and Ltot=793 the probability $\approx$ 0,999994. With the key in our example, where Nt=8 and Ltot= 2198, the probability $\approx 1 - 10^{-14}$. In average all fm key values have been used long before Ltot fm calculations. Ltot calculations is a polynomial amount of calculations, WESP can in average be verified in polynomial time, and therefore WESP belongs to the NP set.

Summary of P vs NP

We have shown that the WESP algorithm is in NP but not in P, meaning that WESP cannot be solved in polynomial time, but can be verified in polynomial time. Since there is at least one algorithm that is in NP but not in P, then P and NP cannot be the same set.

References

[1] Joux Antoine and Muller Frederic. "Chosen IV Attack Against Turing". In: *Lecture Notes in Compter Science* (2004), 194–207.

[2] Frederik Armknecht. "Improving fast algebraic attacks". In: *Lecture notes in Computer Science* (2004), 65–82.

[3] A. Biryukov and A. Shamir. "Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers". In: *Advances in Cryptology - ASIACRYPT 2000* (2000), 1–13.

[4] Biham Eli and Dunkelman Orr. "Related-Key Boomerang and Rectangle Attacks". In: *EUROCRYPT 2005* (2005), 507–525.

[5] Lance Fortnow. *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton, NJ: Princeton University Press, 2013. ISBN: 9780691156491.

[6] Timothy Gowers, June Barrow-Green, and Imre Leader. *The Princeton Companion to Mathematics*. 2008, p. 607. ISBN: 978-0-691-11880-2.

[7] Dinur I and Shamir A. "Cube attacks on tweakable black box polynomials". In: *Proceedings of EUROCRYPT 2009* (2009), 278–299.

[8] John Kelsey, Bruce Schneier, and David Wagner. "Mod n Cryptnanalysis, with Applications Against RC5P and M6". In: *Fast Software Encryption* (1999), 139–155.

[9] Lars Knudsen. "Truncated and Higher Order Differentials". In: *Fast Software Encryption 1994* (1994), 196–211.

[10] Lars Knudsen and Thomas Berson. "Truncated Differentials of SAFER". In: *Fast Software Encryption 1996* (1996), 15–26.

[11] Xuejia Lai. "Higher Order Derivatives and Differential Cryptanalysis". In: *Communications and Cryptography* (1994), 227–233.

[12] Knudsen Lars and Wagner David. "Integral cryptanalysis". In: (2001), 112–127.

[13] P. LEcuyer and R. Simard. *TestU01: A C library for empirical testing of random number generators*. 2007.

[14] M Matsui. "The first experimental cryptanalysis of the data encryption standard". In: *Advances in Cryptology - CRYPTO1994* (1994).

[15] Emmanuel Prouff and Matthieu Rivain. "Masking against Side-Channel Attacs: A Formal Security Proof". In: *Advances in Cryptology - EUROCRYPT 2013* (2013).

[16] A. Shamir. "Impossible differential attacks". In: *CRYPTO 98* (1998).

[17] Claude E. Shannon. "Communication Theory of Secrecy Systems". In: *Bell System Technical Journal* 28.4 (1949), 656–715.

[18] George F. Simmons. *Calculus with Analytic Geometry*. McGraw-Hill Education, 1985, p. 93. ISBN: 0070576424.

[19] Mitra Subhamoy and Paul Goutam. "Analysis of RC4 and Proposal of Additional Layers for Better Security Margin". In: *INDOCRYPT 2008* (2008), 27–39.

[20] Kevin Townsend. "Solving the Quantum Decryption Harvest Now, Decrypt Later Problem". In: (2022).

[21] Robin Wilson. *Eulers Pioneering Equation: The most beautiful theorem in mathematics*. Oxford University Press, 2018, (preface). ISBN: 9780192514059.

[22] Gerhard J. Woeginger. "The P-versus-NP page www.win.tue.nl/wscor/woeginger/P-versus-NP.htm". In: (Retrieved 21 Apr 2024).