Computational aspects of fast correlation attacks

Yves Edel and Andreas Klein

Ghent University

Abstract. In the last years several correlation attacks have been developed. We investigate the implementation of these attacks and find diverse detail improvements which altogether results in a significant speed up.

1 Introduction

Stream ciphers are an important class of crypto-systems. Basically a stream cipher is just a pseudo random number generator. The sequence \( z_0, \ldots, z_{N-1} \) of pseudo random bits will be used like a true random sequence in a one-time pad, i.e. the message \( m_0, \ldots, m_{N-1} \) will be encrypted as \( c_i = m_i + z_i \).

We consider only known plain text attacks. In this case we can compute the output sequence \( z_0, \ldots, z_{N-1} \). The goal of the attacker is to reconstruct the seed of the pseudo random generator, which is identical to the key of the cipher.

Most stream ciphers use linear feedback shift registers (LFSR) as basic elements. It may be possible that the output sequence \( x_0, \ldots, x_{N-1} \) of one of the internal LFSRs is correlated to the output sequence \( z_0, \ldots, z_{N-1} \) of the stream cipher.

For example look at the simple cipher shown in Figure 1 that is build on three LFSRs.

![Fig. 1. A simple LFSR based cryptosystem.](image)

For each of the three LFSRs the output of the whole system is equal to the output of the LFSR with probability \( \frac{3}{7} \). This is a very strong correlation.
An attacker may use this correlation to reconstruct the seed of the internal linear shift registers. Attacks of this kind are known as correlation attacks.

We will not discuss the theory of stream ciphers in detail, for an overview see e.g. [5].

Correlation attacks can be seen as a special decoding problem. The LFSR describes a linear code, which is even cyclic if a full period of the LFSR is considered. The remaining part of the stream cipher can be modeled as a binary symmetric channel (BSC), with some error probability \( p < \frac{1}{2} \). The capacity of the channel is

\[
C(p) = 1 + p \log(p) + (1 - p) \log(p) .
\]

Here and in there remaining article we take the logarithms always to the base 2.

The attacker observes some output bits of the cryptosystem (in a known plaintext attack). This output can be interpreted as a codeword with some error. The goal is correct the error (i.e. to find the seed of the LFSR).

However the problem differs in several aspects from the problem that usually occurs in coding theory.

– In contrast to coding theory, the code is not chosen to have a fast decoding algorithm. In fact the designer of the crypto system has the opposite goal.
– While in coding theory we mostly deal with moderate error probabilities, in cryptography we will deal with error probabilities \( p \) close to \( \frac{1}{2} \).
– In cryptography the attacker can spend much more computational effort on the decoding algorithm than in coding theory. Everything less than \( 2^{30} \) operations is very good.
– If the attacker has a guess for the seed of the cipher, he can compute the corresponding keystream and try to decrypt the message. If the decryption succeeds he knows that the guess was correct. Thus methods which generate only a list of some thousand candidates are acceptable.

In the past years several algorithms for fast correlation attacks where proposed which share a common structure [8, 9, 1, 14, 11].

We sketch this basic attack:

Assume that \( N \) bits \( z_0, \ldots, z_{N-1} \) of the crypto-system can be observed. The sequence \( z_0, \ldots, z_{N-1} \) is correlated to the output \( x_0, \ldots, x_{N-1} \) of the LFSR of length \( l \) by

\[
z_i = x_i + \delta_i
\]

where the \( \delta_i \) are iid. random variables with \( p(\delta_i = 1) = p < \frac{1}{2} \).

The LFSR can be described as a \([N, l]\) linear code \( C \) with generator matrix

\[
G = \begin{pmatrix}
1 & 0 & \cdots & c_{l,0} & \cdots & c_{N-1,0} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & 1 & \cdots & c_{l,l-1} & \cdots & c_{N-1,l-1}
\end{pmatrix},
\]

i.e. \((x_0, \ldots, x_{N-1}) = (x_0, \ldots, x_{l-1})G\).
Fix an integer $B > 0$. The integer $B$ is a parameter controlling the attack. The success probability of the attack will increase with $B$, but also the runtime will increase (exponential in $B$). Thus the minimal $B$ which gives a reasonable success probability should be chosen. But for the moment the value of $B$ is not relevant.

A relation of weight $w$ is a $w$-tuple $(c_{i_1}, \ldots, c_{i_w})$ of columns of the generator Matrix such that the sum of these columns vanishes on all but the first $B$ coordinates. The projection on the first $B$ coordinates of the sum of these columns is called the characteristic of the relation.

Form the relations $r_0, \ldots, r_{R-1}$ with the characteristics $\chi_0, \ldots, \chi_{R-1}$ we construct the $[B, R + B]$ code $C'$ with generator matrix

$$G' = \begin{pmatrix} 1 & 0 \\ \vdots & \vdots \\ 0 & 1 \end{pmatrix} \chi_0 \ldots \chi_{R-1}.$$  

A code word $(x_0, \ldots, x_{N-1})$ of $C$ will be mapped to the code word $w_x := (x_0, \ldots, x_{B-1}, \sum x_{i_k}^{(0)}, \ldots, \sum x_{i_k}^{(R-1)})$ of $C'$. The observed word $(z_0, \ldots, z_{N-1})$ gives rise to the word $w_z := (z_0, \ldots, z_{B-1}, \sum z_{i_k}^{(0)}, \ldots, \sum z_{i_k}^{(R-1)})$. The goal is to correct the error of $w_z$ with respect to the code $C'$, hence to find $w_x$. The error probability changes depending on the size of the relations.

The code $C'$ can simply be decoded by looping over all possible $2^B$ codewords and selecting the code word that comes next to the received word.

The algorithm described above is essentially the simple fast correlation attack of [1]. The attacks of [8, 9] use the LSFR structure to interpret $C$ and $C'$ as convolutional codes and use a modified Viterbi algorithm to decode $C'$. Other variations [14, 11] introduce generalized relations.

We introduce several implementation techniques that have significant impact on the performance of this type of attacks. Some of the modifications are classical (but nevertheless missed in earlier publications), some are tricky and non obvious variations of older algorithms. The speed up depends on the parameters of the problem, but it is never below 2. A factor of 10 is quite common and sometimes it gets even higher. In detail our contributions are the following:

- In Section 2 the preprocessing algorithm is discussed. Our algorithm is similar to the algorithm in [2], but we manage to lower the runtime by a constant factor. For example for the relations of weight 5 our algorithm is more than 40 times faster. Even if we take in account that the data structure chosen in [2] was not optimal, our new specialized algorithms gives an improvement of up to an factor 8 compared to the older algorithms. Since the preprocessing can take days such factors have a large impact.

- In Section 3 we show how to use weighted sums instead of simply counting the satisfied relations. By this modification we remove the paradox situation noted in [14] that more relations can result in a worse success probability.
We show how to use the floating point unit efficiently for that purpose and how to deal with the rounding errors.

- In Section 4 a variation of the classical population count algorithms is presented which speeds up the initialization of relations by a factor 64. For medium sized instances this result leads to a notable speed up of the attack.

- We suggest to modify the Viterbi algorithm by taking two time steps per Viterbi step. This modification allows us to evaluate 1.5 times more relations in approximately the same time. Hence we get a better success probability. Due to restrictions for size of proceedings this part is only in the full version of the article.

2 The preprocessing

In the preprocessing the following problem has to be solved: Delete the first $B$ rows of the generator matrix $G$ of $C$. Denote by $c_0, \ldots, c_{N-1}$ the columns of the reduced matrix. Then the task is to find all $k$-tuples $(i_0, \ldots, i_{k-1})$ with $c_{i_0} + \ldots + c_{i_{k-1}} = 0$ and $k \leq w$.

The best algorithm for the preprocessing we have found in the existing literature is Algorithm 1 in [2]. In this paper, in the theoretical analysis of Algorithm the memory needed for the array $C$ (in the notation of [2]) is neglected. For $w = 6, 7, 9, \ldots$ the memory consumption therefore is underestimated.

We reformulate this Algorithm with correct memory analysis as:

**Theorem 1**

Given that the number of relations is bounded by $N^{(w/2)}$, the preprocessing can be done in $O(N^{(w/2)})$ time and $O(N^{(w-1)/4})$ memory.

**Proof**

First we deal with the case $w$ even, i.e. $w = 2w'$.

For $k \leq w'$ a $k$-tuple $(i_1, \ldots, i_k)$ will be called a partial relation of weight $k$.

The characteristic of this partial relation is $c_{i_1} + \ldots + c_{i_k}$.

For all $k$, with $k \leq \lceil w'/2 \rceil$, we generate tables of the $k$-tuples and sort these partial relations by their characteristic (hash based sorting). Building these tables needs $O(N^{(w'/2)})$ time and memory.

Next we choose $r$ such that $2^r = \Theta(N^{(w'/2)})$. Let $\pi$ denote the projection to the first $r$ coordinates.

For each $a \in \mathbb{F}_2^r$ enumerate the partial relations $(i_1, \ldots, i_s)$ of size $\lceil w'/2 \rceil < s \leq w'$ with the property $\pi(c_{i_1} + \ldots + c_{i_s}) = a$.

To do this enumeration we run through all partial relations $(i_1, \ldots, i_{\lceil s/2 \rceil})$ of size $\lceil s/2 \rceil$ and look in the pre-computed table for relations $(i_{\lceil s/2 \rceil+1}, \ldots, i_s)$ such that $\pi(c_{i_1} + \ldots + c_{i_{\lfloor s/2 \rfloor+1}}) = a + \pi(c_{i_{\lfloor s/2 \rfloor+2}} + \ldots + c_{i_s})$. Thus for fixed $a$ the time needed to enumerate the partial relations $(i_1, \ldots, i_s)$, of size $\lceil w'/2 \rceil < s \leq w'$ with $\pi(c_{i_1} + \ldots + c_{i_s}) = a$, is the number $N_a$ of such relations plus $O(N^{(w'/2)})$ time steps for the algorithm describe above. The memory needed is $N_a = \Theta(N^r/2^r) = \Theta(N^{w'/2})$. 


Once we have all partial relations with $\pi(c_1 + \ldots + c_i) = a$. We can get all relations by searching for pairs of partial relations with the same characteristic.

Altogether the algorithm needs $2^r O(N^{\lceil w/2 \rceil}) + \sum_a N_a = O(N^{\lceil w'/2 \rceil})O(N^{\lceil w'/2 \rceil}) + O(N^{w/2}) = O(N^{w/2})$ time steps.

Next we explain how to deal with the case of $w$ odd. Let $w = 2w' + 1$. In principle we would like to run the same algorithm as for $w - 1 = 2w'$. If we change nothing we need $\Theta(N^{w'+1/2'}) = \Theta(N^{\lceil w'/2 \rceil + 1/2'})$ space to store all partial relations of size $w' + 1$ with $\pi(c_1 + \ldots + c_{i(w/2)}) = a$.

But since $w$ is odd, we will use a partial relations $(j_1, \ldots, j_{w'+1})$ of size $w' + 1$ only, when we look for all partial relations $(i_1, \ldots, i_{w'})$ with $c_1 + \ldots + c_{i_{w'}} = c_{j_1} + \ldots + c_{j_{w'+1}}$. Each solution of this equation give us a relation of size $w = 2w' + 1$.

Since we use the partial relations of size $w' + 1$ only once, we do not need to store them. Thus we store only the partial relations of size up to $w'$ and we need only $\Theta(N^{w'/2'}) = \Theta(N^{\lceil w'/2 \rceil})$ memory to store them.

In the proof of Theorem 1 it is not stated which data structure is used. To obtain the bounds of the theorem, we need a data structure which allows us to store and search a partial relation with given characteristic in $O(1)$ time. In addition we should not use more than $O(N^{\lceil (w-1)/4 \rceil})$ memory to store the $O(N^{\lceil (w-1)/4 \rceil})$ partial relations with $\pi(c_1 + \ldots + c_i) = a$. This calls for a hash table.

In practice other considerations are also important. On modern computers we have at least three types of memory.

- The hard-disk (very slow). We will try to avoid reading and writing to the disk when ever possible.
- The main memory (quite fast, but still slow compared to the CPU).
- The cache, which is almost capable to store and read data at CPU speed.

The most efficient use of a slow memory is to read or write it sequentially into faster memory. This can be done in parallel to other operations, so it is often possible to ignore sequential access completely.

For our problem this has the following consequences. Accessing an element in a hash table or in a sorted list is both a random access in the memory. Therefore a hash table with an $O(1)$ access wins against the sorted list with the $O(\log(n))$ access.

Building a hash table needs also random access. But building an sorted list can be done with sequential access. Furthermore a well written sorting algorithm can make much use of different speed in cache and the memory. Finding pairs with the same key in a sorted list (which corresponds to finding collisions in a hash) also needs only sequential access.

Thus we will use hash based data structures for all partial relations of weight less than $w/2$, because we must probe quite often in these data structures to find matching pairs. (We have found that a closed hash with quadratic probing, is very good for that job.)

If $w$ is even the partial relations of weight $w/2$ are special. We must generate them only once and find matching pairs to get the relation of weight $w$. So
only sequential access is needed and a sorted list is of advantage. The sorting algorithm gives us an asymptotic running time of \(O(n^{w/2} \log n)\) in comparison to the \(O(n^{w/2})\) we can obtain with a hash. But the advantage of the sorting algorithm arising from the different speed of the cache and the main memory is bigger.

For the hash table implementation we want to point out two implementations hints.

We use the last \(h\) coordinates of the characteristic as hash function. Firstly, as small implementation trick, we note that if we use a table of size \(2^h + 100\), the 100 extra places at the end guarantees that the quadratic probing will almost surely never go behind the end of the table. This saves us a modulo operation.

Secondly, a suggestion how to clear the hash table when the loop advance to the next value \(a\). One possibility is to store the value \(a\) of the first \(r\) coordinates of the characteristic also inside the hash. This has the advantage that the algorithm can detect an empty place by the old value of \(a\). The idea to save space by not storing the known value \(a\), results in a serious slow down, since then one must clear the hash table in every iteration of the loop, i.e. we would have \(2^r\) extra operations per loop.

Theorem 1 shows that the preprocessing can be done with only \(O(N)\) memory if \(w \leq 5\). For larger \(w\) the time increases quickly. But especially for \(w = 6\) the time bound is still small enough such that the preprocessing is still feasible, but the \(O(N^2)\) memory can be too large. In such cases one can use the following time-memory tradeoff.

**Theorem 2**

For \(1 \leq x \leq N^\lceil(w-1)/4\rceil-\lceil(w-1)/8\rceil\) one can do the preprocessing in \(O(N^{\lceil w/2 \rceil}x)\) time and \(O(N^\lceil(w-1)/4\rceil/x)\) memory.

**Proof**

The idea is to increase the value of \(r\) in the proof of Theorem 1. That decreases size of the table needed to store all partial relations of weight \([w/2]\). So we choose \(r\) such that \(2^r = \Theta(N^{\lceil w/2 \rceil} - \lceil w/4 \rceil)x\). Following the proof of Theorem 1 we find that the memory requirement decreases to \(O(N^{\lceil(w-1)/4\rceil}/x)\).

The reason why the time requirement increases is the following:

- If \(w \equiv 0 \mod 4\) then we must loop over all partial relation of weight \(w/4 = \lceil(w-1)/4\rceil\) to generate the partial relations with size \(w/2\). Thus we need \(O(N^{w/4})\) time steps per loop and hence \(O(N^{w/2}x)\) altogether.

- In any case we can no longer store the partial relations of weight \(\lceil(w-1)/8\rceil + 1\) to weight \((w-1)/4\) completely as we did it in the algorithm given in the proof of Theorem 1. Instead of storing these partial relations we use a variation of the algorithm given in the proof of Theorem 1: We generate these partial relations in every loop spontaneously. Thus every loop step needs \(O(N^{\lceil(w-1)/4\rceil})\) time steps to enumerate the partial relation of weight \((w-1)/4\). This forces the running time to \(O(N^{\lceil w/2 \rceil}x)\).
The reason why $x$ must be smaller than $N^{((w-1)/4) - [(w-1)/8]}$ is that we must still be able to store the complete table of partial relations of weight $[(w-1)/8]$. 

In principle, by iterating the idea of Theorem 2 the memory can even be can reduced below $O(N^{((w-1)/8)})$; this however for the price of an even bigger slow down. But then we get in a range where the algorithm is impracticable anyway.

For the implementation we want to point out in particular that we use a hash table to find partial relations with the same characteristic. In most of the other publications (for example [14]) on fast correlation attacks sorted arrays are used and consequently it is stated that the running time of the preprocessing is $O(N^{(w/2) \log(N)})$ instead of $O(N^{(w/2)})$. Using a sorted vector (or a balanced tree, etc.) to store the partial relation is unfavorable. For typical instances it costs more than a factor 10 compared to hash based algorithms. We believe that the closed hash described above is the right solution for the problem.

Theorem 1 give the right asymptotic behavior for the preprocessing. But in practice a constant factors matters. We use the following variations of Theorem 1 to speed up the preprocessing by a constant factor.

- For relations $c_{i_1} + c_{i_2} + c_{i_3} + c_{i_4} = 0$ of weight 4 we use the following special algorithm:
  
  Without loss of generality assume that $c_{i_1}$ and $c_{i_2}$ have the same first component. Hence $c_{i_3}$ and $c_{i_4}$ also have the same first component. Therefore it is enough to enumerate all partial relations $(i_{1}, i_{2})$ of weight 2 for which the first component of the characteristic $c_{i_1} + c_{i_2}$ is 0. Thus only roughly $\binom{N}{2}/2$ partial relations, instead of $\binom{N}{2}$, need to be generated.

- For $w = 3$ this technique can be slightly improved. There are two cases. In the first case the first component of $c_{i_1}$ and $c_{i_2}$ is 1 while the first component of $c_{i_3}$ is 0. To deal with this case, all pairs $(c_{i_1}, c_{i_2})$ with the first component of 1 must be enumerated. This are approximately $\binom{N}{2}$ pairs.

  The second case is that all three vectors $c_{i_1}, c_{i_2}$ and $c_{i_3}$ have the first component 0. In this case we can repeat the same argument with the second component.

  In total we enumerate only $\binom{N}{2}/2^2 + \binom{N}{2}/2^3 + \ldots \approx N^2/6$ pairs.

- For $w = 5$ we look at the projection to the first three coordinates. The equation $c_{i_1} + c_{i_2} + c_{i_3} + c_{i_4} + c_{i_5} = 0$, $c_{i_5} \in \mathbb{F}_2^3$, has only the following solutions:

  Either two vectors are equal (without loss of generality we can assume $c_{i_4} = c_{i_3}$). This case can be covered by enumerating all $\binom{N}{3}/2^3$ partial relations for which $c_{i_1} + c_{i_2} + c_{i_3} = 0$.

  The second case is that no two vectors are equal. But this implies that also $c_i + c_j \neq c_k$ for $i \neq j, j \neq k, i \neq k$.

  The only solution (up to permutation of indexes) is $c_{i_5} = 0$, $c_{i_4} = c_{i_2} + c_{i_3}$, $c_{i_1} + c_{i_2} + c_{i_3}$. There are two possibilities (up to permutation) either $c_{i_1} = (100)$, $c_{i_2} = (010)$, $c_{i_3} = (001)$ or the weight of $c_{i_1}$ and $c_{i_2}$ is even. For each of the
four subcases there are only \( \frac{N^3}{8r} \) partial-relations of weight 3 that fall into this subcase. Hence we only need to enumerate \( N^3 \left( \frac{1}{2^3} + 4\frac{1}{3^3} \right) \) of the \( \binom{N}{3} \) possible relations. This is an improvement of a factor of almost 8 compared to the basic algorithm.

Table 1 shows some timing results from our implementation. As can be seen, choosing the wrong data structure results in a serious slow down. The effect will amplify with the number of relations.

Table 2 shows that the specialized algorithms brings a noteworthy improvement and confirms the theoretically predicted speedup. All timings are done on one core of a Intel Core2 CPU, T5500 at 1.66GHz, but the relative speed up is machine independent.

<table>
<thead>
<tr>
<th>data structure</th>
<th>time needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash table</td>
<td>6 min 5 sec</td>
</tr>
<tr>
<td>sorted list</td>
<td>20 min 55 sec</td>
</tr>
</tbody>
</table>

Table 1. Impact of the data structure used to store the partial relations, \( N = 100000 \), \( w = 3 \), \( (l = 40, B = 8) \)

<table>
<thead>
<tr>
<th>parameters</th>
<th>generic version</th>
<th>specialized version</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (l = 40, B = 8) ), ( N = 100000 ), ( w = 3 )</td>
<td>6 min 5 sec</td>
<td>1 min 50 sec</td>
</tr>
</tbody>
</table>

Table 2. Impact of the specialized algorithms

3 Counting the satisfied relations

We now deal with the problem of how to count the number satisfied relations. In principle we think that the algorithm in given [2], that is a mixture of bit-counting and Fourier transformation, is the right choice. The idea to split \( B \) into \( B = B_1 + B_2 \) and apply \( 2^{B_1} \) convolutions of size \( 2^{B_2} \) to count the satisfied relations. To initialize the data structure for the convolutions we must do \( 2^{B_1+B_2} \) bit counts on array of size about \( R/2^{B_2} \). We only have two points to criticize:

The first is that in [2] it is assumed that the constant hidden in the \( O(2^B R) \) steps for the bit counting and in the \( O(2^B B) \) steps of the Fourier transform is the same. Under this assumption in [2] it is (correctly) shown that right point to switch from a bit-counting algorithm to the Fourier transform is obtained by choosing \( B_2 = \log_2 R \). But taking in account that the constants are different we have to choose \( B_2 = \log_2 cR \) for some constant \( c \).

The constant for the bit counting algorithm is very low (as e.g. 64 bits can be processed by a word operation, ...) in comparison to the constant in the Fourier
transform. This implies that the bit counting algorithm should be applied longer than estimated in [2].

In the real world the constants are also effected by the caching of our machine. For example if the data structure for the Fourier transform fits almost in the cache it is always favorable to reduce the size of the Fourier transform to fit in the cache by doing a bit more work in the bit counting part.

The second point is that in [2], like in many other publications in this field, it is ignored that relations of different size have different quality. If a wrong guess is tested every relation is satisfied with probability \( \approx \frac{1}{2} \). For a correct guess a relation of weight \( w \) is satisfied with probability

\[
p_w = \sum_{j=0}^{\lfloor w/2 \rfloor} (1-p)^{w-2j} p^{2j}
\]

where \( p < 0.5 \) is the error probability.

Just for illustration lets take the very small error probability \( p = 0.25 \). Then a relation of weight 2 is satisfied with \( p_2 = 0.625 \) and a relation of weight 5 is satisfied only with \( p_5 = 0.517 \). Thus we should trust relations of smaller weight more than relations of higher weight. Nevertheless most papers simply count the number of satisfied relations ignoring this fact.

In [14] we find the following theorem:

**Result 1 (Theorem 4.10 in [14])**

Let \( p = 1/2 + \delta \) be the strength of the correlation. Then the attack with relations of weight up to \( w + 1 \) has a higher success probability than the one using only weight \( w \) relations if and only if

\[
2\delta \sqrt{\frac{N}{w + 1}} > 1
\]

In [14] it is claimed that maximum likelihood decoding is used, but in maximum likelihood extra information can never decrease the success probability. So Result 1 simply demonstrates that only counting the satisfied relations is not maximum likelihood decoding.

The maximum likelihood decoder works as follows:

Let \( n_i \) be the number of relations of weight \( i \) and \( k_i \) be the number of satisfied relation of weight \( i \). Under the assumption that our guess is correct the probability of observing this pattern of satisfied relations is

\[
p_{\text{guess}} = \prod_{i=1}^{w} p_i^{k_i} (1-p_i)^{n_i-k_i}, \tag{1}
\]

where

\[
p_i = \sum_{j=0}^{\lfloor i/2 \rfloor} \binom{i}{2j} p^{2j} (1-p)^{i-2j}
\]
is the probability that a relation of weight $i$ is satisfied.

By the maximum likelihood principle the guess for which $p_{\text{guess}}$ becomes maximal has to be found. As multiplying small floating point numbers is unfavorable, it is better to work with the logarithmic version of this equation.

$$\log p_{\text{guess}} = \sum_{i=1}^{w} k_i \log(p_i) + (n_i - k_i) \log(1 - p_i)$$

Subtracting the constant $\sum_{i=1}^{w} n_i \log(1 - p_i)$ we see that we have to maximize

$$\sum_{i=1}^{w} k_i (\log(p_i) - \log(1 - p_i))$$

Finally by dividing by $\log(p_w) - \log(1 - p_w)$ we get the equivalent problem to maximize

$$\sum_{i=1}^{w} k_i w_i,$$

with

$$w_i = \frac{\log(p_i) - \log(1 - p_i)}{\log(p_w) - \log(1 - p_w)}.$$

Some textbooks (like [3]) advice to replace the real weights by integer approximations. One reason for such an advise is that in the past the floating point units were very slow. On modern hardware the floating point unit is highly optimized and often faster than the integer unit. For example the integer unit and the floating point unit of the PS3 are of the same speed $25.6 \cdot 10^9$ operations per second (according to the vendor [16]). Especially systems that use graphic processing units for computations are often better in floating point than in integer arithmetic (see [15] for example of such an system).

On the other hand using floating point arithmetic introduces a new source of trouble. Adding many floating point numbers can lead to serious rounding errors (especially a simple loop will almost surly produce a poor result). A lot of sophisticated algorithms exists to deal with the problem. For the implementation of the Fourier transform we choose a variation of the recursive Kahan-Babuška Algorithm presented in [10]. To distinguish in following arguments the floating point operation from the exact operation, we write $\oplus$, $\ominus$, ... for the floating point operations, and $+$, $-$, ... for the exact operations.

Our algorithm relies on the following results:

**Result 2 (see [12] Section 4.2.2 Theorem C)**

Let $u$ and $v$ be binary floating point values with $|u| \geq |v|$. Then

$$u + v = (u \oplus v) + ((u \ominus (u \oplus v)) \oplus v) \quad (2)$$

**Result 3 (see [7] Theorem 2.5)**

Let $u$ and $v$ be binary floating point values with $u \geq v \geq \frac{1}{4}u \geq 0$. Then $u \ominus v = u - v$. 
In combination the two results allow us to save some operations compared to the simple recursive Kahan-Babuška Algorithm of [10]. In the FFT algorithm we have to compute repeatedly \( u + v \) and \( u - v \). Algorithm 1 shows in pseudo-code how to apply the results to improve the computation. In addition to the values \( u \) and \( v \) we also store correction values \( u' \) and \( v' \). At the beginning of the Fourier transform the correction values are initialized with 0 and at the end of the computation we add the correction value to the corresponding value. The error bounds of Algorithm 1 are the same as for the recursive Kahan-Babuška Algorithm of [10].

**Algorithm 1** Computing \( u + v \) and \( u - v \) with correction values

**Require:** Input values \( u, v \) with corrections values stored in \( u' \) and \( v' \)

**Ensure:** Store the sum in \( u \) and the difference in \( v \) and updates the corrections values

1: if \(|v| > |u|\) then
2: swap \( u \) and \( v \)
3: end if
4: if \( uv \geq 0 \) then
5: \( s \leftarrow u \oplus v \)
6: \( c \leftarrow (u \ominus s) \ominus v \)
7: \( v \leftarrow 2u \ominus s \) \{ = 2u - s, since \( s \in [u, 2u] \) (Result 3) \}
8: \( u \leftarrow s \)
9: \( [u', v'] \leftarrow [(u' \oplus v') \ominus c, v' \leftarrow (u' \ominus v') \ominus c] \)
10: \{ \( u + v = s + c \) (Result 2) hence \( u - v = (2u - s) - c \) \}
11: else
12: \( d \leftarrow u \ominus v \)
13: \( c \leftarrow (u \ominus d) \ominus v \)
14: \( v \leftarrow 2u \ominus d \) \{ = 2u - d, since \( d \in [u, 2u] \) (Result 3) \}
15: \( v \leftarrow d \)
16: \( [u', v'] \leftarrow [(u' \oplus v') \ominus c, v' \leftarrow (u' \ominus v') \ominus c] \)
17: \{ \( u + v = s + c \) (Result 2) hence \( u - v = (2u - s) - c \) \}
18: end if

Algorithm 1 may seem like an expensive method to dealing with rounding errors, but effectively it doubles the number of correct digits and is still less expensive that using double precision arithmetic. (At least on machines with highly optimized single precision floating points such as e.g. graphic processing units. Also on a PS3 single precision floats are 10 times faster than double precision.)

It should be noted that the repetitive application of Algorithm 1 in a Fourier transform works best if \( u \) and \( v \) a mostly of the same size. In our application we ensure this by subtraction of the known expect log-likelihood value. In other applications it could be a good idea first to calculate the mean and subtract it.

Of course these corrections need a lot of time. So if almost all relations are of the same weight (which is the normal case for a correlation attack) it is favorable
to use integer arithmetic and accept the small error made by using minimum distance decoding instead of maximum likelihood decoding.

There are at least two cases were one should switch to the more complicated floating point arithmetic. Firstly when we are in the situation in which more relations will decrease the success probability of minimum distance decoding (see Result 1). Then true maximum likelihood decoding gives us the chance to use more relations. Secondly there are situations in which all relations have different probabilities (see for example [6] which attacks the shrinking generator). In this case ignoring the different probabilities throws away too much information.

Finally a remark on the implementation of the convolution. Another possibility to accelerate the computation is to use SIMD instructions now available on many machines (e.g. the SSE instruction set on Intel architectures). The drawback of SIMD instructions, however, is that we cannot use the balancing algorithm to reduce the rounding errors.

4 Initialization of the relations

By “initialization of the relations” we denote the evaluation of $s^{(i)} = r_0^{(i)} + \ldots + r_{k_i - 1}^{(i)}$ for each relation $\{r_0^{(i)}, \ldots, r_{k_i - 1}^{(i)}\}$. In an attack based on convolutional codes the sum $s_t^{(i)} = x_{r_0^{(i)} + 1} + \ldots + x_{r_{k_i - 1} + t}$ has to be evaluated for each time step $t$. So the goal is to count the number of indexes $i$ for which $s_t^{(i)} = 1$.

A naive implementation would run Algorithm 2 for every time step.

<table>
<thead>
<tr>
<th>Algorithm 2 Naive initialization of the relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: for $i$ from 0 to $R - 1$ do</td>
</tr>
<tr>
<td>2: $s \leftarrow 0$</td>
</tr>
<tr>
<td>3: for $j$ from 0 to $k_i - 1$ do</td>
</tr>
<tr>
<td>4: $s \leftarrow s + x_{r_j^{(i)} + t}$</td>
</tr>
<tr>
<td>5: end for</td>
</tr>
<tr>
<td>6: if $s = 1$ then</td>
</tr>
<tr>
<td>7: increase counter</td>
</tr>
<tr>
<td>8: end if</td>
</tr>
<tr>
<td>9: end for</td>
</tr>
</tbody>
</table>

For very big instances the time needed for the initialization of the relation is negligible. But for medium sized instances the initialization takes a big fraction of the total runtime (for example in an attack against an LFSR of length 70 with $B = 15$ and $w = 5$ the naive initialization would need more than 50% of the whole run time). As we want to use the attack against medium sized instances as second step in two step decoding algorithms this part should be optimized.
The observation is that we can evaluate the $s_t(i)$ for $t = 0, \ldots, 63$ simultaneously. Simply store the sequence $x_i$ in a bitfield. Then the word $x = (x_1, \ldots, x_{t+63})$ can be accessed by Algorithm 3. The following algorithms assume that the word size of our computer is 64 bit, but all algorithms can easily adopted to other word sizes. The fixed word size 64 is just to simplify notation.

Algorithm 3 Access a word in a bit field
1: if $t \text{ mod } 64 = 0$ then
2: $x \leftarrow X[t/64]$
3: else
4: $x \leftarrow (X[t/64] \ll (t \text{ mod } 64)) | (X[t/64 + 1] \gg (t \text{ mod } 64))$
5: end if

Word operations allows us to compute the matrix $S = (s_t(i))_{i=0, \ldots, R-1; t=0, \ldots, 63}$ very fast. Our goal is to count the 1’s in the columns of the matrix $S$. The straightforward approach to solve that problem is to transpose the matrix $S$ and apply the classical population count (sideway addition) algorithms (see [19] Section 5.1 or [13]) to the 64 bitfields in $S^t$. This does not work well, since there is no really fast way to transpose a bit matrix (see [19] Section 7.3 and the comments on pmovmskb in [20]). A better solution is to use the following variation of the population count algorithm which counts the 1’s in the columns of the matrix $S$ directly (see also [4]).

Algorithm 4 Counting 1’s in the columns of a bit matrix (Listing 6 in [4])
1: {Input: $A = (a^{(i)})_{i=0, \ldots, 255}$ is a 25564 bit matrix}
2: {Output: 64 bytes which contains $\sum_{j=0}^{254} a^{(j)}$ for $i = 0, \ldots, 63$}
3: for $i$ from 0 to 84 do
4: $b^{(2i)} \leftarrow \sum_{j=0}^{2} (a^{(3i+j)} \& \mu_0)$
5: $b^{(2i+1)} \leftarrow \sum_{j=0}^{2} ((a^{(3i+j)} \gg 1) \& \mu_0)$
6: end for
7: for $i$ from 0 to 16 do
8: for $x$ from 0 to 1 do
9: $c^{(4i+x)} \leftarrow \sum_{j=0}^{4} (b^{(16i+2j+x)} \& \mu_1)$
10: $c^{(4i+2+x)} \leftarrow \sum_{j=0}^{4} ((b^{(16i+2j+x)} \gg 2) \& \mu_1)$
11: end for
12: end for
13: for $x$ from 0 to 3 do
14: $d^{(x)} \leftarrow \sum_{j=0}^{16} (c^{(4j+x)} \& \mu_2)$
15: $d^{(4+x)} \leftarrow \sum_{j=0}^{16} ((c^{(4j+x)} \gg 4) \& \mu_2)$
16: end for
Theorem 3
The words \( d^{(i)} = (d_{i,0}, \ldots, d_{i,7})_{256} \) computed by Algorithm 4 satisfies

\[
d_{i,j} = \sum_{k=0}^{255} a^{(k)}_{i+8j}.
\]

Algorithm 4 needs 2219 word operations (not counting memory access).

\((\text{abcdef})_{256}\) denotes a number written in base 256.)

Proof
We need 5 operations for line 4 and 8 operations for line 5. This makes altogether
\(85 \cdot (5 + 8) = 1105\) operations to compute the \(b\)'s from the \(a\)'s. To get the \(c\)'s from the \(b\)'s we need two AND's and one SHIFT per \(b\) and four additions per \(c\), i.e. \(170 \cdot 3 + 68 \cdot 4 = 782\) operations.
To compute the eight \(d\)'s we need \(68 \cdot 3 + 8 \cdot 16 = 332\) operations.
Altogether these are 2219 operations.

For comparison, the classical population count algorithm (see [19] Section 5.1) needs 14 operations per word. Algorithm 4 is with \(\frac{2219}{255} \approx 8.7\) operation per word faster, moreover we do not to transpose the matrix first.
In addition to Algorithm 4, we can use a technique developed by Harley and Seal [17, 18].

The basic tool for these Methods is a carry-save adder (CSA) or 3 : 2 compressor. A CSA adder is just a sequence of independent full adders. A full adder is circuit that takes the bits \(a\), \(b\), \(c\) and computes the two bit sum \(a + b + c\). For example a 4-bit CSA that gets the words \((1100), (1010)\) and \((1010)\) as input produces as output the words \((1000)\) (the carry) and \((1100)\) (the sum).

With a CSA we can reduce three population counts to 2 by using the identity:

\[
popc(a) + popc(b) + popc(c) = 2popc(h) + popc(l).
\]

Here \((h, l)\) is the output of the CSA called with \(a\), \(b\) and \(c\). There are different ways to use equation (3) for a population count algorithm. For details see the example program \texttt{pop\_arrayHS.c} of [18].

For large bit fields Algorithm 4 for a vertical population count reaches a speed that makes it even attractive to replace a normal population count by a vertical population and add the population count of the columns afterwards. In [4] we show how to optimise the algorithm for normal population count.

With the method described in this section we speed up the initialization of the relations by a factor of almost 64 which eliminates the initialization as a time consuming step.

5 Conclusions
Implementing fast correlation attacks is a difficult task. The right data structures have to be chosen carefully and highly optimized bit manipulation algorithms have to be used to obtain good results. Each of the implementation techniques described in this article give either a significant improvement in speed or in the archived success probability.
References