Fast Correlation Attacks Based on Turbo Code Techniques

Thomas Johansson* and Fredrik Jönsson*

Dept. of Information Technology
Lund University, P.O. Box 118, 221 00 Lund, Sweden
{thomas,fredrikj}@it.lth.se

Abstract. This paper describes new methods for fast correlation attacks on stream ciphers, based on techniques used for constructing and decoding the by now famous turbo codes. The proposed algorithm consists of two parts, a preprocessing part and a decoding part. The preprocessing part identifies several parallel convolutional codes, embedded in the code generated by the LFSR, all sharing the same information bits. The decoding part then finds the correct information bits through an iterative decoding procedure. This provides the initial state of the LFSR.

Keywords. Stream ciphers, correlation attacks, convolutional codes, iterative decoding, turbo codes.

1 Introduction

Stream ciphers are generally faster than block ciphers in hardware, and have less complex hardware circuitry, implying a low power consumption. Furthermore, buffering is limited and in situations where transmission errors can occur the error propagation is limited. These are all properties that are especially important in, e.g., telecommunications applications.

Consider a binary additive stream cipher, i.e., a synchronous stream cipher in which the keystream, the plaintext, and the ciphertext are sequences of binary digits. The output sequence of the keystream generator, $z_1, z_2, \ldots$ is added bitwise to the plaintext sequence $m_1, m_2, \ldots$, producing the ciphertext $c_1, c_2, \ldots$. The keystream generator is initialized through a secret key k, and hence, each key k will correspond to an output sequence. Since the key is shared between the transmitter and the receiver, the receiver can decrypt by adding the output of the keystream generator to the ciphertext, obtaining the message sequence, see Figure 1.

The design goal is to efficiently produce random-looking sequences that are as "indistinguishable" as possible from truly random sequences. Also, from a cryptanalysis point of view, a good stream cipher should be resistant against different kind of attacks, e.g., a *known-plaintext attack*. For a synchronous stream cipher, a known-plaintext attack is equivalent to the problem of finding the key

* The authors are supported by the Foundation for Strategic Research - PCC under Grant 9706-09.

Michael Wiener (Ed.): CRYPTO'99, LNCS 1666, pp. 181–197, 1999.

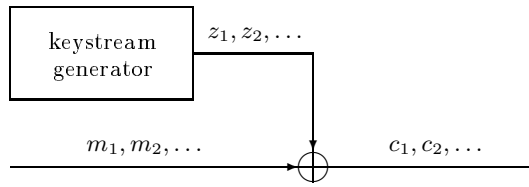

Fig. 1. Principle of binary additive stream ciphers

k that produced a given keystream $z_1, z_2, \ldots, z_N$. We hence assume that a given output sequence from the keystream generator $z_1, z_2, \ldots, z_N$ is known to the cryptanalyst and that his task is to restore the secret key k.

In stream cipher design, it is common to use linear feedback shift registers, LFSRs, as building blocks in different ways. Furthermore, the secret key k is often chosen to be the initial state of the LFSRs.

Several classes of general cryptanalytic attacks against stream ciphers exist [13]. A very important, if not the most important, class of attacks on LFSR-based stream ciphers is *correlation attacks*. If one can, in some way, detect a correlation between the known output sequence and the output of one individual LFSR, it is possible to mount a "divide-and-conquer" attack on the individual LFSR [16,17,11,12]. Observe that there is no requirement of structure of any kind for the key generator. The only requirement is the fact that, if $u_1, u_2, \ldots$ denotes the output of the particular LFSR, we have a correlation of the form $P(u_n = z_n) \neq 0.5$, $n \geq 1$. Other types of correlations may also apply.

A common methodology for producing random-like sequences from LFSRs is to combine the output of several LFSRs by a nonlinear Boolean function f with desired properties [13]. The purpose of f is to destroy the linearity of the LFSR sequences and hence provide the resulting sequence with a large linear complexity [13]. Note that for such a stream cipher, there is always a correlation between the output z_n and either one or a set output symbols from different LFSRs.

Finding a low complexity algorithm that successfully can use the existing correlation in order to determine a part of the secret key can be a very efficient way of attacking stream ciphers for which a correlation is identified. After the initializing ideas of Siegenthaler [16,17], Meier and Staffelbach [11,12] found a very interesting way to explore the correlation in what was called a *fast* correlation attack. A necessary condition is that the feedback polynomial of the LFSR has a very low weight. This work was followed by several papers, providing minor improvements to the initial results of Meier and Staffelbach, see [14,3,4,15]. However, the algorithms demonstrate true efficiency (good performance and low complexity) only if the feedback polynomial is of low weight. Based on these attacks, it is a general advise that the generator polynomial should not be of low weight when constructing stream ciphers.

More recently, a step in another direction was taken, and it was suggested to use convolutional codes in order to improve performance [8]. More precisely, it was shown that one can identify an embedded low-rate convolutional code in the code generated by the LFSR sequences. This convolutional code can then be decoded using, e.g., the Viterbi algorithm, and a correctly decoded information sequence will provide the correct initial state of the LFSR.

The purpose of this paper is to describe new algorithms for fast correlation attacks. They are based on combining the iterative decoding techniques introduced by Meier and Staffelbach [11,12] with the framework of embedded convolutional codes as suggested by the authors [8]. The proposed algorithm consists of two parts, a preprocessing part and a decoding part.

By considering permuted versions of the code generated by the LFSR sequences, several "parallel" embedded convolutional codes can be identified. They all share the same information sequence but have individual parity checks. This is the preprocessing part.

In the decoding part, the keystream $z_1, z_2, \ldots, z_N$ is first used to construct sequences acting as received sequences for the above convolutional codes. These sequences are then used to find the correct information sequence by an iterative decoding procedure.

The code construction in the preprocessing part and the iterative decoding technique in the decoding part resemble very much the by now famous *turbo codes* and its decoding techniques [2]. Iterative decoding requires APP (a posteriori probability) decoding (also called MAP decoding), and for decoding convolutional codes one can use the famous BCJR algorithm [1], or preferably some modification of it [19,5].

For a fixed memory size, the proposed algorithm provides a better performance than previous methods. As a particular example taken from [14], consider a LFSR of length 40 and an observed sequence of length 40000 bits. Let $1-p$ be the correlation probability. Then for a certain memory size ($B = 13$), the best known algorithm [8] is successful up to $p = 0.19$, whereas the maximum p for the proposed algorithm lie in the range $0.20 - 0.27$ when the number of parallel codes varies from one to 32. The price payed for increased performance is an increased computational complexity, but as argued in the paper, it is usually the available memory that limits the performance and not the computational complexity.

The paper is organized as follows. In Section 2 we give some preliminaries on the decoding model that is used for cryptanalysis, and in Section 3 and 4 we shortly review previous algorithms for fast correlation attacks. In Section 5 we present some of the new ideas in a basic algorithm using only one code. In Section 6 the use of several "parallel" codes is introduced and the complete algorithm is described. Simulation results are presented in Section 7. In Section 8 a parallelizable algorithm is proposed, and in Section 9 we conclude with some possible extensions.

2 Preliminaries

As most other authors [17,11,12,14,3], we use the approach of viewing our cryptanalysis problem as a decoding problem. An overview is given in Figure 2. Let the LFSR have length l and let the set of possible LFSR sequences be denoted by $\mathcal{L}$. Clearly, $|\mathcal{L}| = 2^l$ and for a fixed length N the truncated sequences from $\mathcal{L}$ is also a linear $[N, l]$ block code [10], referred to as $\mathcal{C}$. Furthermore, the keystream sequence $\mathbf{z} = z_1, z_2, \ldots, z_N$ is regarded as the received channel output and the LFSR sequence $\mathbf{u} = u_1, u_2, \ldots, u_N$ is regarded as a codeword from the linear block code $\mathcal{C}$.

Due to the correlation between u_n and z_n, we can describe each z_n as the output of the binary symmetric channel, BSC, when u_n was transmitted. The correlation probability $1 - p$, defined by $1 - p = P(u_n = z_n)$, gives p as the crossover probability (error probability) in the BSC. W.l.o.g we assume $p < 0.5$. This is all shown in Figure 2.

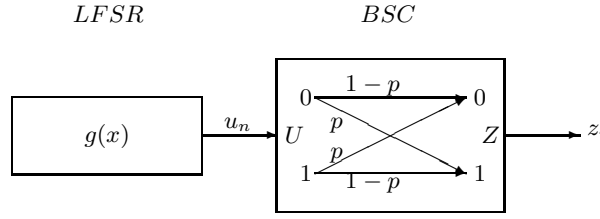

Fig. 2. Model for a correlation attack

The problem of cryptanalysis is now the following. Given the received word $(z_1, z_2, \ldots, z_N)$ as output of the BSC(p), find the length N codeword from $\mathcal{C}$ that was transmitted.

It is known that the length N should be at least around $N_0 = l/(1 - h(p))$ for unique decoding [3], where $h(p)$ is the binary entropy function. If the length N of the observed keystream sequence is small but allows unique decoding, the fastest methods for decoding are probabilistic decoding algorithms like Leon or Stern algorithms [9,18].

We assume instead received sequences of large length, $N >> N_0$. For this case, the *fast correlation attacks* [11,12] are applicable. These attacks resemble very much the iterative decoding process proposed by Gallager [6] for low-weight parity-check codes. The drawback is the fact that the above attacks require the feedback polynomial $g(x)$ (or any multiple of $g(x)$ of modest degree) to have a low weight. Hence one usually refrain from using such feedback polynomials in stream cipher design.

3 Fast Correlation Attacks – An Overview

In [11,12] Meier and Staffelbach presented two algorithms, referred to as A and B, for fast correlation attacks. Instead of the exhaustive search as originally suggested in [17], the algorithms are based on using certain parity check equations created from the feedback polynomial of the LFSR.

In the first pass, a set of suitable parity check equations in the code $\mathcal{C}$ is found. The second pass uses these parity check equations in a fast decoding algorithm to recover the transmitted codeword and hence the initial state of the LFSR.

Parity check equations in [11,12] were created in two separate steps. Let $g(x) = 1+g_1x^1+g_2x^2+\ldots+g_lx^l$ be the feedback polynomial, and t the number of taps of the LFSR, i.e., the weight of $g(x)$ (the number of nonzero coefficients) is $t+1$. Symbol number n of the LFSR sequence, u_n, can then be written as $u_n = g_1u_{n-1} + g_2u_{n-2} + \ldots + g_lu_{n-l}$. Since the weight of $g(x)$ is $t+1$, there are the same number of relations involving a fixed position u_n. Hence, we get in this way $t+1$ different parity check equations for u_n. Secondly, using the fact that $g(x)^j = g(x^j)$ for $j = 2^i$, parity check equations are also generated by repeatedly squaring the polynomial $g(x)$. The obtained parity check equations are then (essentially) valid in each index position of $\mathbf{u}$.

The number of parity check equations, denoted m, that can be found in this way is $m \approx \log(\frac{N}{2l})(t+1)$, where log uses base 2 [11,12].

In the second pass, we have m equations for position u_n as,

$$\begin{aligned} u_n + b_1 &= 0, \\ u_n + b_2 &= 0, \\ &\vdots \\ u_n + b_m &= 0, \end{aligned} \tag{1}$$

where each b_i is the sum of t different positions of $\mathbf{u}$. Applying the above relations to the keystream we can introduce L_i, $\leq i \leq m$, defined as the following sums,

$$\begin{aligned} z_n + y_1 &= L_1 \\ z_n + y_2 &= L_2 \\ &\vdots \\ z_n + y_m &= L_m. \end{aligned} \tag{2}$$

where y_i is the sum of the positions in the keystream $\mathbf{z}$ corresponding to the positions in b_i. Assume that h out of the m equations in (1) hold, i.e.,

$$h = |\{i : L_i = 0, 1 \leq i \leq m\}|,$$

The probability $p^* = P(u_n = z_n | h \text{ equations hold})$ is calculated as

$$p^* = \frac{ps^h(1-s)^{m-h}}{ps^h(1-s)^{m-h} + (1-p)(1-s)^h s^{m-h}},$$

where $p = P(u_n = z_n)$, and $s = P(b_i = y_i)$. This is used as an estimate of whether z_n was correct or not.

Two different decoding methods was suggested in [11,12]. The first algorithm, called Algorithm A, can shortly be described as follows: Calculate the probabilities p^* for each bit in the keystream, select the l positions with highest value of p^*, and calculate a candidate initial state. Finally, find the correct value by checking the correlation between the sequence and the keystream for different small modifications of the candidate initial state.

The second algorithm, called Algorithm B, uses instead an iterative approach. The algorithm uses two parameters p_{thr} and N_{thr}.

1. For all symbols in the keystream, calculate p^* and determine the number of positions N_w with $p^* < p_{thr}$.
2. If $N_w < N_{thr}$ repeat step 1 with p replaced by p^* in each position.
3. Complement the bits with $p^* < p_{thr}$ and reset the probabilities to p.
4. If not all equations are satisfied go to step 1.

This iterative approach is fundamental for our considerations, and we refer to [11,12] for more details.

The algorithms above work well when the LFSR contains few taps, but for LFSRs with many taps the algorithms fail. The reason for this failure is that for LFSRs with many taps each parity check equation gives a very small average correction and hence many more equations are needed in order to succeed. Or in other words, the maximum correlation probability p that the algorithms can handle is much lower if the LFSR has many taps ($\approx l/2$). Minor improvements were suggested in, e.g., [14] and [3].

4 Fast Correlation Attacks Based on Convolutional Codes

The general idea behind this algorithm, proposed recently in [8], is to consider slightly more advanced decoding algorithms *including memory*, but which still have a low decoding complexity. This allows weaker restrictions on the parity check equations that can be used, leading to many more and more powerful equations. The work in [8] then uses the Viterbi algorithm as its decoding algorithm in the decoding part.

We now review the basic results of [8]. The algorithm transforms a part of the code $\mathcal{C}$ stemming from the LFSR sequences into a convolutional code. The encoder of this convolutional code is created by finding suitable parity check equations from $\mathcal{C}$. It is assumed that the reader is familiar with basic concepts regarding convolutional codes (see also [8]).

Let B be a fixed memory size and let R denote the rate. In a convolutional encoder with memory B and rate $R = 1/(m+1)$ the vector $\mathbf{v}_n$ of codeword symbols at time n,

$$\mathbf{v}_n = (v_n^{(0)}, v_n^{(1)}, \ldots, v_n^{(m)}),$$

is of the form

$$\mathbf{v}_n = u_n\mathbf{g}_0 + u_{n-1}\mathbf{g}_1 + \ldots u_{n-B}\mathbf{g}_B, \tag{3}$$

where each $\mathbf{g}_i$ is a vector of length $(m+1)$. The task in the first pass of the algorithm is to find suitable parity check equations that will determine the vectors $\mathbf{g}_i, 0 \leq i \leq m$, defining the convolutional code.

Let us start with the linear code $\mathcal{C}$ stemming from the LFSR sequences. There is a corresponding $l \times N$ generator matrix G_{LFSR}, such that $\mathbf{u} = \mathbf{u}_0 G_{LFSR}$, where $\mathbf{u}_0$ is the initial state of the LFSR. The generator matrix is furthermore written in systematic form, i.e., $G_{LFSR} = \begin{pmatrix} I_l & Z \end{pmatrix}$, where I_l is the $l \times l$ identity matrix.

We are now interested in finding parity check equations that involve a current symbol u_n, and an arbitrary linear combination of the B previous symbols $u_{n-1}, \ldots, u_{n-B}$, together with at most t other symbols. Clearly, t should be small and in this description $t = 2$ is mainly considered.

To find these equations, start by considering the index position $n = B + 1$. Introduce the following notation for the generator matrix,

$$G_{LFSR} = \begin{pmatrix} I_{B+1} & Z_{B+1} \\ 0_{l-B-1} & Z_{l-B-1} \end{pmatrix}. \tag{4}$$

Parity check equations for u_{B+1} with weight t outside the first $B+1$ positions can then be found by finding linear combinations of t columns of Z_{l-B-1} that adds to the all zero vector.

For the case $t = 2$ the parity check equations can be found in a very simple way as follows. A parity check equation with $t = 2$ is found if two columns from G_{LFSR} have the same value when restricted to the last $l - B - 1$ entries (the Z_{l-B-1} part). Hence, we simply put each column of Z_{l-B-1} into one of 2^{l-B-1} different "buckets", sorted according to the value of the last $l-B-1$ entries. Each pair of columns in each bucket will provide us with one parity check equation, provided u_{B+1} is included.

Assume that the above procedure gives us a set of m parity check equations for u_{B+1}, written as

$$\begin{array}{c} u_{B+1} + \sum_{i=1}^{B} c_{i1} u_{B+1-i} + \sum_{i=1}^{\leq t} u_{j_{i1}} = 0, \\ u_{B+1} + \sum_{i=1}^{B} c_{i2} u_{B+1-i} + \sum_{i=1}^{\leq t} u_{j_{i2}} = 0, \\ \vdots \\ u_{B+1} + \sum_{i=1}^{B} c_{im} u_{B+1-i} + \sum_{i=1}^{\leq t} u_{j_{im}} = 0. \end{array} \tag{5}$$

It directly follows from the cyclic structure of the LFSR sequences that the same set of parity checks is valid for any index position n simply by shifting all the symbols in time, resulting in

$$\begin{array}{c} u_n + \sum_{i=1}^{B} c_{i1} u_{n-i} + b_1 = 0, \\ u_n + \sum_{i=1}^{B} c_{i2} u_{n-i} + b_2 = 0, \\ \vdots \\ u_n + \sum_{i=1}^{B} c_{im} u_{n-i} + b_m = 0, \end{array} \tag{6}$$

where $b_k = \sum_{i=1}^{\leq t} u_{j_{ik}}$, $1 \leq k \leq m$ is the sum of (at most) t positions in $\mathbf{u}$.

Using the equations above one can create an $R = \frac{1}{m+1}$ bi-infinite systematic convolutional encoder. Recall that the generator matrix for such a code is of the form

$$G = \begin{pmatrix} \ddots & \ddots & & \ddots & & \\ & \mathbf{g}_0 & \mathbf{g}_1 & \cdots & \mathbf{g}_B & \\ & & \mathbf{g}_0 & \mathbf{g}_1 & \cdots & \mathbf{g}_B \\ & & & \ddots & \ddots & & \ddots \end{pmatrix}, \tag{7}$$

where the blank parts are regarded as zeros. Identifying the parity check equations from (6) with the description form of the convolutional code as in (7) gives

$$\begin{pmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_B \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 0 & c_{11} & c_{12} & \ldots & c_{1m} \\ 0 & c_{21} & c_{22} & \ldots & c_{2m} \\ \vdots & \ddots & & \ddots & \vdots \\ 0 & c_{B1} & c_{B2} & \ldots & c_{Bm} \end{pmatrix}. \tag{8}$$

For each defined codeword symbol $v_n^{(i)}$ in the convolutional code one has an estimate of that symbol from the transmitted sequence $\mathbf{z}$. If $v_n^{(i)} = u_n$ (an information bit) then $P(v_n^{(i)} = z_n) = 1 - p$. Otherwise, if $v_n^{(i)} = u_{j_{1i}} + u_{j_{2i}}$ from (6) then $P(v_n^{(i)} = z_{j_{1i}} + z_{j_{2i}}) = (1-p)^2 + p^2$. Using these estimates one can construct a sequence

$$\mathbf{r} = \ldots r_n^{(0)} r_n^{(1)} \ldots r_n^{(m)} r_{n+1}^{(0)} r_{n+1}^{(1)} \ldots r_{n+1}^{(m)} \ldots,$$

where $r_n^{(0)} = z_n$ and $r_n^{(i)} = z_{j_{1i}} + z_{j_{2i}}$, $1 \leq i \leq m$, that plays the role of a received sequence for the convolutional code, where $P(v_n^{(0)} = r_n^{(0)}) = 1 - p$ and $P(v_n^{(i)} = r_n^{(i)}) = (1-p)^2 + p^2$ for $1 \leq i \leq m$.

To recover the initial state of the LFSR it is enough to decode l consecutive information bits correctly. Optimal decoding (ML decoding) of the convolutional code using the Viterbi algorithm can thus be performed.

The original Viterbi algorithm assumes that the convolutional encoder starts in state $\mathbf{0}$. However, in this application there is neither a starting state, nor an ending state for the trellis corresponding to the convolutional code. Hence, one runs the Viterbi algorithm over a number of "dummy" information symbols, placed before and after the l information symbols that one tries to decode correctly, see [8]. A suitable choice is to decode over $J = l + 10B$ information symbols, where the l symbols in the middle are regarded as the l information bits that we want to estimate. The particular choice of J is based on heuristics for the decision distance of the decoding algorithm.

5 Some New Ideas for Fast Correlation Attacks

Two methods for decoding a noisy LFSR sequence have been described in Section 3 and 4. The Meier and Staffelbach Algorithm B calculates an a posteriori probability for each symbol of the complete received sequence and then iteratively tries to improve these probabilities by recalculating them. The procedure is based on very simple (memoryless) parity checks. The method of Section 4 uses instead convolutional codes but uses a simple Viterbi decoding procedure on a small part of the received sequence.

The ideas to be proposed try to combine the best parts of both methods into a single algorithm. The first and basic construction uses one convolutional code (Section 4 method) and then applies an APP (a posteriori probability) decoding algorithm in order to provide an a posteriori probability for each symbol in a certain part of the received sequence. Optimal APP decoding (also referred to as MAP decoding) on a convolutional code can be performed by the famous BCJR algorithm [1], or variations of it. The a posteriori probabilities are then fed back as a priori probabilities and in this fashion the procedure is iterated until convergence. This is now described in more detail.

The first step involves computing parity check equations for a convolutional code with fixed memory B. We follow the procedure of Section 4 and compute all parity check equations with $t = 2$ involving the particular index position $B + 1$, as given by (5). Parity checks for index position $B + 1 + i$ are then immediately obtained through a cyclic shift of the original parity checks with i steps, as in (6). We refer to Section 4 for a review of the details. Write the obtained parity check equations in the form

$$\begin{array}{c} u_n + \sum_{i=1}^{B} c_{i1} u_{n-i} + u_{i_{n1}} + u_{j_{n1}} = 0, \\ u_n + \sum_{i=1}^{B} c_{i2} u_{n-i} + u_{i_{n2}} + u_{j_{n2}} = 0, \\ \vdots \\ u_n + \sum_{i=1}^{B} c_{im} u_{n-i} + u_{i_{nm}} + u_{j_{nm}} = 0. \end{array} \tag{9}$$

The convolutional code is defined by all codeword sequences $\mathbf{v}$,

$$\mathbf{v} = \ldots v_n^{(0)} v_n^{(1)} \ldots v_n^{(m)} v_{n+1}^{(0)} v_{n+1}^{(1)} \ldots v_{n+1}^{(m)} \ldots, \qquad B + 1 \le n \le J,$$

where

$$v_n^{(0)} = u_n, \quad v_n^{(k)} = u_n + \sum_{i=1}^{B} c_{ik} u_{n-i}, \quad 1 \le k \le m.$$

Observe that the code is defined *only over the interval* $B + 1 \le n \le J$. Since there are neither a starting state nor an ending state for the code trellis, we again decode over $J - B > l$ information symbols. Following Section 4, we choose $J = 10B + l$ as a starting point. Through simulations one can later adjust J to the most suitable value. Furthermore, the starting state, denoted $\mathbf{s_s}$, of the trellis (start value for the memory contents of the convolutional code) is given a probability distribution which is $P(\mathbf{s_s}) = P(u_1, u_2, \ldots, u_B)$.

The second step is the APP decoding phase. After receiving a sequence $\mathbf{z}$, construct a sequence $\mathbf{r}$ acting as a received sequence for the convolutional code by

$$\mathbf{r} = \ldots r_n^{(0)} r_n^{(1)} \ldots r_n^{(m)} r_{n+1}^{(0)} r_{n+1}^{(1)} \ldots r_{n+1}^{(m)} \ldots, \qquad B+1 \leq n \leq J,$$

where

$$r_n^{(0)} = z_n, \quad r_n^{(k)} = z_{i_{nk}} + z_{j_{nk}}, \quad 1 \leq k \leq m.$$

Next, we transfer the a priori probabilities of $\mathbf{u}$ to the sequence $\mathbf{v}$ by

$$P(v_n^{(0)} = r_n^{(0)}) = P(u_n = z_n), \tag{10}$$

where $P(u_n = z_n) = 1 - p$ only in the first iteration, and

$$P(v_n^{(k)} = r_n^{(k)}) = (1-p)^2 + p^2, 1 \leq k \leq m. \tag{11}$$

Then decode the constructed sequence $\mathbf{r}$ stemming from a codeword $\mathbf{v}$ of the convolutional code using an APP decoding algorithm. The original BCJR algorithm requires storage of the whole trellis. However, suboptimal versions of the BCJR algorithm, see [19,5], remove this problem with a negligible decrease in performance. This procedure provides us with the a posteriori probabilities for the information sequence, i.e.,

$$P(v_{B+1}^{(0)}|\mathbf{r}), P(v_{B+2}^{(0)}|\mathbf{r}), \ldots, P(v_J^{(0)}|\mathbf{r}).$$

Finally, since $v_{B+1}^{(0)} = u_{B+1}, v_{B+2}^{(0)} = u_{B+2}, \ldots$ this information is fed back as new a priori probabilities for $(u_{B+1}, u_{B+2}, \ldots, u_J)$ and the a priori probabilities of the codeword sequence $\mathbf{v}$ of the convolutional code is recalculated. The decoding procedure is performed a second time, and this procedure is iterated $2-5$ times (until convergence).

Basic algorithm description:

Input: The $l \times N$ generator matrix G_{LFSR} for the code generated by a LFSR; the received sequence $\mathbf{z}$; the error probability p; the number of iterations I.

1. (Precomputation) For each position n, $B+1 \leq n \leq J$, in G_{LFSR}, find the set of parity check equations of the form (9) and construct the convolutional code.
2. (Decoding phase) After receiving $\mathbf{z}$, construct the a priori probability vector $(P(u_{B+1}), P(u_{B+2}), \ldots, P(u_J))$ by $P(u_n = z_n) = 1 - p$. Construct the received sequence $\mathbf{r}$ by

$$r_n^{(0)} = z_n, \quad r_n^{(k)} = z_{i_{nk}} + z_{j_{nk}}, \quad 1 \leq k \leq m.$$

and the corresponding a priori probabilities for $\mathbf{v}_n$, $B+1 \leq n \leq J$ by

$$P(v_n^{(k)} = r_n^{(k)}) = (1-p)^2 + p^2, 1 \leq k \leq m.$$

3. (Decoding phase) Update

$$P(v_n^{(0)}) = P(u_n), \quad B+1 \leq n \leq J.$$

Run the APP decoding algorithm with starting state distribution $P(\mathbf{s_s}) = P(u_1, u_2, \ldots, u_B)$. Receive the a posteriori probabilities

$$P(v_{B+1}^{(0)}|\mathbf{r}), P(v_{B+2}^{(0)}|\mathbf{r}), \ldots, P(v_J^{(0)}|\mathbf{r}).$$

Since $v_n^{(0)} = u_n$, set

$$P(u_{B+1}) \leftarrow P(v_{B+1}^{(0)}|\mathbf{r}), P(u_{B+2}) \leftarrow P(v_{B+2}^{(0)}|\mathbf{r}), \ldots, P(u_J) \leftarrow P(v_J^{(0)}|\mathbf{r}).$$

4. If the number of iterations $< I$ go to 3., otherwise select the most probable value for each of the symbols $u_{5B+1}, u_{5B+2}, \ldots, u_{5B+l}$, calculate the initial state $\mathbf{u}_0$ and check if it is correct.

We end by presenting some simulation results for the basic algorithm. The obtained results are compared with the results in [11,12,14,8]. We choose to use the same case as tabulated in [14,8], which is based on a LFSR with length $l = 40$, and a weight 17 feedback polynomial.

[11,12], Alg B.	[14]	[8]			Basic algorithm		
		$B = 13$	$B = 14$	$B = 15$	$B = 13$	$B = 14$	$B = 15$
0.092	0.096	0.19	0.22	0.26	0.20	0.23	0.26

Table 1. Maximum p for some different algorithms when $N = 40000$ and $B = 13, 14, 15$.

6 Algorithms Based on Turbo Code Techniques

One of the most revolutionary ideas in coding theory the last decade has been the introduction of turbo codes. The original turbo code [2] consists of two convolutional codes, where the information bits are directly fed into one of them and an interleaved version of the same information bits are fed into the other convolutional code. The fundamentally new idea was the proposed decoding scheme, which uses an iterative procedure. Decode the first code using an APP decoding algorithm which provides a posteriori probabilities for all information symbols. Use these as a priori information when decoding the second code using again APP decoding. The obtained a posteriori probabilities are now used as a priori information when decoding the first code a second time, and the procedure continues in this iterative fashion.

Much the same ideas as described above can be applied to our decoding problem. Instead of using just one fixed convolutional code, as in the basic algorithm described in Section 5, we will show how to find and use two or more different convolutional codes. These are obtained by randomly permuting the index positions of the original code in the interval $B+1 \ldots J$.

The a posteriori probability vector, which is the result of the decoding phase, from one code is fed as a priori information to the other code. This is viewed in Figure 3. A problem arises, however, since we need parity check equations for

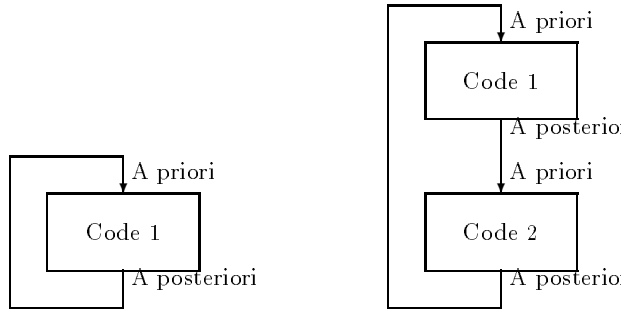

Fig. 3. The basic algorithm and the turbo code algorithm with two constituent codes.

permuted versions of the code $\mathcal{C}$. The shifting technique will no longer provide this for all indices, since after the column permutation the new code has no longer the cyclic properties of $\mathcal{C}$. To overcome this problem we simply search for all valid parity check equations in each index position. It will increase the precomputation time by a factor $J-B$, but for the case $t=2$ this is not at all a problem. Hence, this procedure will create different parity check equations for different index positions, thus leading to a *timevarying* convolutional code (in opposite to the code in Section 5). Also the *number* of parity checks will vary with n.

In order to find the parity check equations for an index position n, where $B+1 \leq n \leq J$, write the permuted generator matrix in the form

$$G_{LFSR} = \begin{pmatrix} Z_{11} & I_{B+1} & Z_{12} \\ Z_{21} & 0_{l-B-1} & Z_{22} \end{pmatrix}, \tag{12}$$

where Z_{21} has length $n-B-1$ and Z_{22} has length $N-n$. Then put each column of Z_{22} together with its index position into one of 2^{l-B-1} different "buckets", sorted according to the column value. Each pair of columns in each bucket will provide us with one valid parity check equation (of the form (9)) for index position n, provided u_n is included. Finally, since the number of parity checks will vary with

n, we introduce $m(n)$ as the number of found parity checks for index position n. The parity check equations for index position n is written as

$$\begin{array}{c} u_n + \sum_{i=1}^{B} c_{i1} u_{n-i} + u_{i_{n1}} + u_{j_{n1}} = 0, \\ u_n + \sum_{i=1}^{B} c_{i2} u_{n-i} + u_{i_{n2}} + u_{j_{n2}} = 0, \\ \vdots \\ u_n + \sum_{i=1}^{B} c_{im(n)} u_{n-i} + u_{i_{nm(n)}} + u_{j_{nm(n)}} = 0. \end{array} \tag{13}$$

For each n, the constants defining the parity check equations,

$$\begin{pmatrix} \mathbf{g}_0(n) \\ \mathbf{g}_1(n) \\ \vdots \\ \mathbf{g}_B(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 0 & c_{11} & c_{12} & \dots & c_{1m(n)} \\ 0 & c_{21} & c_{22} & \dots & c_{2m(n)} \\ \vdots & \ddots & & \ddots & \vdots \\ 0 & c_{B1} & c_{B2} & \dots & c_{Bm(n)} \end{pmatrix}. \tag{14}$$

must be stored in order to build the trellis in the decoding phase. After this precomputation phase, the decoding of the first code follows the procedure of Section 5, but the resulting a posteriori probabilities are now fed as a priori information to the second decoder. The same procedure is repeated until all decoders have completed their task, and then the resulting a posteriori information is fed back to the first decoder, starting the second iteration. After 2-5 iterations the decoding phase is completed. A comprehensive description of the procedure for M constituent codes/decoders follows.

Turbo algorithm description:

Input: The $l \times N$ generator matrix G_{LFSR} for the code generated by the LFSR; the received sequence $\mathbf{z}$; the error probability p; the number of iterations I; the number of constituent codes M.

1. (Precomputation) Let $\pi_2, \dots, \pi_M$ be $M-1$ random permutations permuting indices $B+1, \dots, J$ and leaving the other indices fixed. Let $G_1 = G_{LFSR}, G_2 = \pi_2(G_{LFSR}), \dots, G_M = \pi_M(G_{LFSR})$ be generator matrices for M different codes which are all permuted versions of G_{LFSR}. For G_i, let $\pi_i(\mathbf{z})$ be the received sequence, $2 \le i \le M$. Then find all parity checks of the form (13) for each G_i, $1 \le i \le M$. Initiate $i \leftarrow 1$.
2. (Decoding phase) After receiving $\mathbf{z}$, construct the a priori probability vector $(P(u_{B+1}), P(u_{B+2}), \dots, P(u_J))$ by $P(u_n = z_n) = 1 - p$.
   For each G_i, construct the received sequence $\mathbf{r}$ by

$$r_n^{(0)} = z_n, \quad r_n^{(k)} = z_{i_{nk}} + z_{j_{nk}}, \quad 1 \le k \le m.$$

   and the corresponding a priori probabilities for $\mathbf{v}_n$, $B+1 \le n \le J$ by

$$P(v_n^{(k)} = r_n^{(k)}) = (1-p)^2 + p^2, 1 \le k \le m.$$

3. (Decoding phase) For G_i, update

$$P(v_n^{(0)}) = P(u_n), \quad B+1 \leq n \leq J.$$

Run the MAP algorithm on G_i with starting state distribution $P(\mathbf{s_s}) = P(u_1, u_2, \ldots u_B)$. Receive the a posteriori probabilities

$$P(v_{B+1}^{(0)}|\mathbf{r}), P(v_{B+2}^{(0)}|\mathbf{r}), \ldots, P(v_J^{(0)}|\mathbf{r}).$$

Set

$$P(u_{B+1}) \leftarrow P(v_{B+1}^{(0)}|\mathbf{r}), P(u_{B+2}) \leftarrow P(v_{B+2}^{(0)}|\mathbf{r}), \ldots, P(u_J) \leftarrow P(v_J^{(0)}|\mathbf{r}).$$

Let $i \leftarrow i+1$ and if $i = M+1$ then $i \leftarrow 1$.

4. If the number of iterations $< I \cdot M$ go to 3., otherwise select the most probable value for each of the symbols $u_{5B+1}, u_{5B+2}, \ldots, u_{5B+l}$, calculate the initial state $\mathbf{u}_0$ and check it for correctness.

7 Performance of the Turbo Algorithm

In this section we present some simulation results for the turbo algorithm. The parameter values are $J = 10B + l$ and $I = 3$. In Table 2 we show the maximum error probability for a received sequence of length $N = 40000$ when the memory B is varying in the range $10-13$ and the number of constituent codes is $1, 2, 4, 8$ and 16. Table 3 then shows the same for length $N = 400000$.

B	[8]	$M=1$	$M=2$	$M=4$	$M=8$	$M=16$
12	0.12	0.18	0.21	0.22	0.23	0.25
13	0.19	0.20	0.22	0.24	0.25	0.26
14	0.22	0.23	0.24	0.26	0.27	0.28
15	0.26	0.26	0.27	0.29	0.30	0.30

Table 2. Maximum p for turbo algorithm with $B = 12, \ldots, 15$ and varying M when $N = 40000$.

We can see the performance improvement with growing M for fixed B. A few comments regarding computational complexity and memory requirements are in place.

If one uses the suboptimal APP decoding algorithm in [19] the memory requirements will be roughly the same as in Viterbi decoding. The computational complexity for the algorithm in [19] is roughly a factor 3 higher compared to the Viterbi algorithm, since it runs through the trellis three times. There are also slightly different operations performed in the algorithms. The computational complexity is then further increased a factor M when the turbo algorithm with

B	[8]	$M=1$	$M=2$	$M=4$	$M=8$	$M=16$
10	0.31	0.31	0.33	0.34	0.35	0.36
11	0.34	0.34	0.36	0.37	0.38	0.38
12	0.36	0.37	0.38	0.38	0.39	0.39
13	0.37	0.39	0.40	0.40	0.41	0.41

Table 3. Maximum p for turbo algorithm with $B = 10, \ldots, 13$ and varying M when $N = 400000$.

M constituent codes are considered. Finally, we iterate at least twice. To conclude, for fixed parameters B and N, the turbo algorithm have roughly the same memory requirements, but an increase of computational complexity of at least a factor $6M$.

It is important to note that in many cases, the possible performance is not limited by the computational complexity, but rather, limited by the required memory. For example, if $N = 40000$, the maximal memory size that our current implementation could handle for the basic algorithm on a regular PC (for the example given in Table 2) was $B = 17$, but this required only roughly half an hour CPU time. Hence, in this case we do not consider the penalty of increased computational complexity to be severe.

8 A Parallel Version of the Turbo Algorithm

As can be seen from the description of the turbo algorithm, it is not directly parallelizable (the APP decoding can be partly parallelized). Since it is very reasonable to assume that the opponent can compute in parallel, we shortly describe a modified turbo algorithm. Assume that the opponent has access to M different processors. He then constructs M different constituent convolutional codes exactly as described in Section 6 using some suitable memory B. After having received the keystream $\mathbf{z}$, the received sequences $\mathbf{r}$ are constructed and the a priori probabilities are calculated. Next, processor number i works as the APP decoder for code number i, with the a priori probabilities as input. Observe that all the decoders work on the same input. Each decoder then outputs an a posteriori probability vector.

For each position n, an overall a posteriori probability for that position, $P(u_n = z_n)$ is calculated. Let $P_i(u_n = z_n|\mathbf{r})$ be the a posteriori probability stemming from code i. Then the overall a posteriori probability for this algorithm is given by

$$P(u_n = z_n) = \frac{\Pi_{i=1}^{M} P_i(u_n = z_n|\mathbf{r})}{\Pi_{i=1}^{M} P_i(u_n = z_n|\mathbf{r}) + \Pi_{i=1}^{M}(1 - P_i(u_n = z_n|\mathbf{r}))}. \qquad (15)$$

The probability vector is then fed back as a priori information, and the same process is repeated again. The structure is depicted in Figure 4. The performance is slightly worse than the turbo algorithm. Some simulation result for $N = 40000$ are given in Table 4.

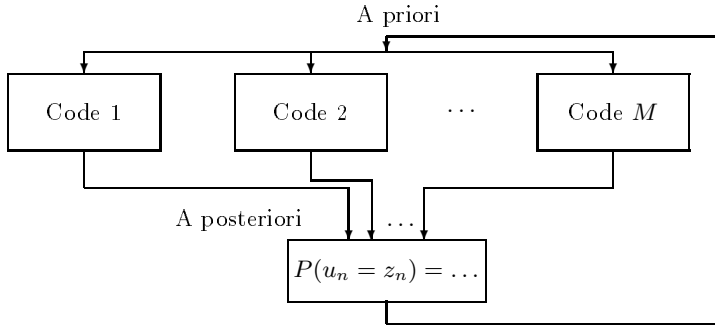

Fig. 4. The parallel turbo code algorithm combining the probabilities as in (15).

	$M = 1$	$M = 2$	$M = 4$	$M = 8$	$M = 16$
Turbo	0.20	0.22	0.24	0.25	0.26
Parallel turbo	0.20	0.22	0.23	0.23	0.24

Table 4. Maximum p for the parallel turbo algorithm when $N = 40000$ and $B = 13$.

9 Conclusions

In this work we have shown how iterative decoding techniques based on the ideas from the construction and decoding of turbo codes can be used as a basis for correlation attacks on stream ciphers. The performance has been demonstrated through simulations. Still, many possible variations of the proposed type of fast correlation attacks exist, that need to be examined. The proposed iterative decoding techniques have opened for other possibilities that can be considered in the future. We mention two possible extensions.

- *Reduced complexity decoding.* As noted in this work, the main performance limit for the proposed decoding algorithms as well as for [8] is the memory requirement. A possible way to overcome this problem is to consider suboptimal decoding algorithms with reduced memory. Examples of such are list decoding and different sequential decoding algorithms [7].
- *Other iterative decoding structures.* An alternative to the decoding structure in this work, as given in Figure 3, could be the following. Consider all index positions that are used to build parity check equations for the convolutional code. Now consider these positions as information symbols for another convolutional code, and find parity checks for this code. Decoding this code will provide APP probabilities for its information symbols and hence more reliable parity checks for the first code. This idea is easily generalized to more complex decoding structures.

References

1. L. R. Bahl, J. Cooke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. IT-20, 1974, pp. 284–287.
2. C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding," *Proc., IEEE Int. Conf. on Communications, ICC'93*, 1993, pp. 1064–1070.
3. V. Chepyzhov, and B. Smeets, "On a fast correlation attack on certain stream ciphers", In *Advances in Cryptology–EUROCRYPT'91*, Lecture Notes in Computer Science, vol. 547, Springer-Verlag, 1991, pp. 176–185.
4. A. Clark, J. Golic, E. Dawson, "A comparison of fast correlation attacks", *Fast Software Encryption, FSE'96*, Lecture Notes in Computer Science, Springer-Verlag, vol. 1039, 1996, pp. 145–158.
5. J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. Inform. Theory*, vol. IT-42, 1996, pp. 429–445.
6. R. G. Gallager, *Low-Density Parity-Check Codes*, MIT Press, Cambridge, MA, 1963.
7. R. Johanesson, K. Sh. Zigangirov, *Fundamentals of convolutional coding,* IEEE Press, New York, 1999.
8. T. Johansson, F. Jönsson, "Improved fast correlation attacks on stream ciphers via convolutional codes", *Advances in Cryptology–EUROCRYPT'99*, Lecture Notes in Computer Science, vol. 1592, Springer-Verlag, 1999, pp. 347–362.
9. J. Leon, "A probabilistic algorithm for computing minimum weights of large error-correcting codes", *IEEE Trans. Information Theory*, vol. 34, 1988, pp. 1354–1359.
10. F. MacWilliams, N. Sloane, *The theory of error correcting codes*, North Holland, 1977.
11. W. Meier, and O. Staffelbach, "Fast correlation attacks on stream ciphers", *Advances in Cryptology–EUROCRYPT'88*, Lecture Notes in Computer Science, vol. 330, Springer-Verlag, 1988, pp. 301–314.
12. W. Meier, and O. Staffelbach, "Fast correlation attacks on certain stream ciphers", *Journal of Cryptology*, vol. 1, 1989, pp. 159–176.
13. A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
14. M. Mihaljevic, and J. Golic, "A fast iterative algorithm for a shift register initial state reconstruction given the noisy output sequence", *Advances in Cryptology–AUSCRYPT'90*, Lecture Notes in Computer Science, vol. 453, Springer-Verlag, 1990, pp. 165-175.
15. W. Penzhorn, "Correlation attacks on stream ciphers: Computing low weight parity checks based on error correcting codes", *Fast Software Encryption, FSE'96*, Lecture Notes in Computer Science, vol. 1039, Springer-Verlag, 1996, pp. 159–172.
16. T. Siegenthaler, "Correlation-immunity of nonlinear combining functions for cryptographic applications", *IEEE Trans. on Information Theory*, vol. IT–30, 1984, pp. 776–780.
17. T. Siegenthaler, "Decrypting a class of stream ciphers using ciphertext only", *IEEE Trans. on Computers,* vol. C–34, 1985, pp. 81–85.
18. J. Stern, "A method for finding codewords of small weight," *Coding Theory and Applications*, Springer-Verlag, 1989, pp. 106–113.
19. A. Trofimov, K. Zigangirov, "A posteriori probability decoding of convolutional codes", to appear in *Problems of Information Transmission*, 1999.