Cryptanalysis of the Bluetooth Stream Cipher

Christophe De Cannière[1⋆], Thomas Johansson[2], and Bart Preneel[1]

[1] Katholieke Universiteit Leuven, Dept. ESAT,
Kasteelpark Arenberg 10,
B–3001 Leuven-Heverlee, Belgium
{christophe.decanniere,bart.preneel}@esat.kuleuven.ac.be
[2] Dept. of Information Technology,
Lund University, P.O. Box 118,
S–221 00 Lund, Sweden
thomas@it.lth.se

Abstract. E_0 is a 128-bit key stream cipher that provides the link encryption in the Bluetooth wireless standard. In this paper we present a new attack on the E_0 key stream generator based on a previous known plaintext attack by Scott Fluhrer. A theoretical method to analyse and improve this new attack is developed, yielding a final algorithm that derives the internal state of the generator in 2^{76} steps and requires 1 Mbit of plaintext.

1 Introduction

When the Bluetooth specification [**?**] was published in 1999, all details regarding the security system were made public as well. In the documents describing the new short-range radio link solution, the developers presented a stream cipher called E_0, which was designed to provide the wireless connections with a strong protection against eavesdropping while at the same time limiting the extra area and power consumption of its implementation on silicon.

The E_0 encryption system is built around a relatively simple key stream generator which is initialised with a key of at most 128 bit. In this paper we develop a new cryptanalytic attack against this key stream generator based on previous work by Scott R. Fluhrer [**?**]. The attack is intended to recover the internal state of the generator from a sequence of known key stream bits. For efficient ways to use this internal state to derive the session key we refer to a very recent paper by Scott R. Fluhrer and Stefan Lucks [**?**].

The paper is organized as follows. We start with a brief description of the E_0 algorithm in Section 2. In Section 3 we give an overview of previous attacks on E_0. Section 4 presents and analyses the new attack and in Section 5 we discuss two possible improvements. The accuracy of our theoretical predictions is verified through simulations in Section 6 and we conclude in Section 7.

⋆ F.W.O. Research Assistant, sponsored by the Fund for Scientific Research – Flanders (Belgium)

COSIC internal report, 15 pages, 2001.

2 Description of the E_0 Key Stream Generator

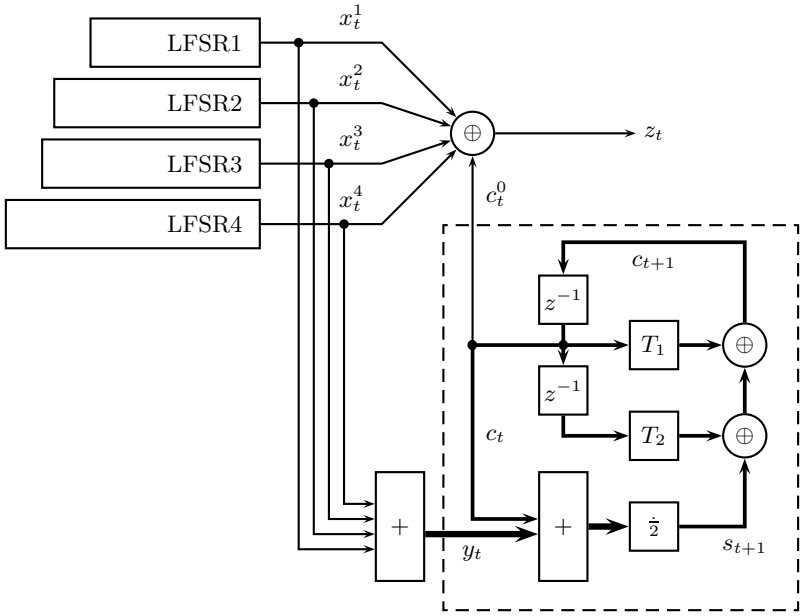

Fig. 1. E_0 key stream generator.

The design of the key stream generator used in E_0 is strongly inspired by Rueppel's summation combiner [**?**]. As shown in Figure 1, the generator consists of four linear feedback shift registers (LFSR) whose outputs x_t^i are combined by a simple finite state machine.

The specifications of the LFSRs are listed in Table 1. The registers are all of different length and contain a total of 128 delay elements. Since all four feedback polynomials can be shown to be primitive, the LFSRs will generate maximum-length sequences.

Table 1. Feedback polynomials of the four LFSRs.

LFSR	Length	Feedback polynomial
1	25	$1 + D^8 + D^{12} + D^{20} + D^{25}$
2	31	$1 + D^{12} + D^{16} + D^{24} + D^{31}$
3	33	$1 + D^4 + D^{24} + D^{28} + D^{33}$
4	39	$1 + D^4 + D^{28} + D^{36} + D^{39}$

The state of the FSM is determined by 4 bits, which are stored in a pair of 2-bit delay elements. At each time t, the lower delay element stores the previous value of the upper element and we can therefore refer to these two 2-bit values as c_{t-1} and c_t respectively. The new contents c_{t+1} of the upper delay element is computed as follows:

$$c_{t+1} = (c_{t+1}^1, c_{t+1}^0) = s_{t+1} \oplus T_1(c_t) \oplus T_2(c_{t-1}) \tag{1}$$

with

$$y_t = (y_t^2, y_t^1, y_t^0) = x_t^1 + x_t^2 + x_t^3 + x_t^4 \qquad \in \{0, 1, 2, 3, 4\} \tag{2}$$

$$s_{t+1} = (s_{t+1}^1, s_{t+1}^0) = \left\lfloor \frac{y_t + c_t}{2} \right\rfloor \qquad \in \{0, 1, 2, 3\} \tag{3}$$

T_1 and T_2 in (1) are two linear bijections over $\mathbb{Z}_2^2$ and are defined below:

$$T_1 : \mathbb{Z}_2^2 \to \mathbb{Z}_2^2, (x_1, x_0) \mapsto (x_1, x_0) \tag{4}$$

$$T_2 : \mathbb{Z}_2^2 \to \mathbb{Z}_2^2, (x_1, x_0) \mapsto (x_0, x_1 \oplus x_0) \tag{5}$$

Finally, the key stream bits z_t are calculated by taking the exclusive-or of the four LFSR sequences x_t^i together with the output bit of the FSM c_t^0:

$$z_t = x_t^1 \oplus x_t^2 \oplus x_t^3 \oplus x_t^4 \oplus c_t^0 \tag{6}$$

The encryption is performed by taking the exclusive-or of this sequence z_t and the bit sequence of a data packet. The maximum size of such a packet is limited to 2745 bits. When the transmission of this packet is completed, the cipher is reinitialised. We won't discuss the details of this initialisation as they are not important for our attack.

3 Previous Attacks on E_0

As is usual in cryptanalysis, we focus on known-plaintext attacks, i.e. we assume a situation in which the attacker is able to obtain a certain amount of decrypted text in one way or another. The goal of a known-plaintext attack is to use this information to recover other (unknown) parts of the plaintext. In the case of additive stream ciphers, this problem reduces to finding a way to predict the entire key stream z_t given a limited number of key stream bits.

To derive the output bits of the key stream generator described in the previous section, at least two fundamentally different approaches are possible.

3.1 Correlation Attacks

The first method consists in separately recovering parts of the initial state of the generator by exploiting specific correlation properties of the finite state machine.

This type of statistical attack was first proposed by Miia Hermelin and Kaisa Nyberg [**?**] and then improved by Patrik Ekdahl and Thomas Johansson [**?**]. An important drawback of this approach, however, is that it cannot be applied to the actual E_0 algorithm, as it assumes sequences of consecutive key stream bits which are considerably longer than the maximum packet size.

3.2 Guess and Determine Attacks

In the second approach, some parts of the initial state are guessed first and the observed key sequence is used to derive the remaining parts in a deterministic way afterwards. This type of guess and determine attacks on E_0 was first introduced by Markku-Juhani O. Saarinen [**?**] in a shortcut attack based on the observation that the contents of a single LFSR can directly be derived from the contents of the three other LFSRs, the 4 bits of the FSM, and some known key stream. By guessing the initial state of the FSM and the contents of the three shortest LFSRs ($4 + 25 + 31 + 33 = 93$ bits), the attack succeeds in recovering the entire internal state in 2^{92} steps.

An improved algorithm is presented by Scott R. Fluhrer [**?**]. Refining Saarinen's idea, he suggests an attack in which only the contents of the two shortest LFSRs are guessed together with the initial state of the FSM. This time, the known key stream bits do not allow the attacker to determine the bits of both remaining LFSRs directly. Instead, a set of linear equations is collected and checked for inconsistencies. As soon as one is found, the corresponding guess is rejected. Fluhrer's algorithm reduces the complexity of the attack to 2^{85} and requires the same amount of data as the previous attack (132 key stream bits). Additionally, he proposes an optimisation saving processor time at the expense of requiring considerably more key stream bits. Unlike correlation attacks, however, his algorithm allows the known key stream bits do be spread over multiple data packets. Complexities of the order of 2^{83}, 2^{81} and 2^{79} are achievable given 2 kbit, 1 Mbit and 60 Mbit of data respectively.

4 Our Attack

The attack we present in this paper is, on its turn, an extension of Fluhrer's attack. Our starting point is to investigate whether we can obtain further improvements with a guess and determine attack guessing only the shortest LFSR. The approach will be similar to the one in Fluhrer's attack, but some additional complications will arise.

In the following subsections, we describe the basic algorithm and develop a method to predict its theoretical complexity.

4.1 Basic Attack

As mentioned above, the attack requires us to guess the contents of the shortest LFSR (called LFSR1) and assume a certain initial state for the FSM. For each

guess, we progressively run through the known key stream bits z_t and maintain a set $\mathcal{S}$ of equations in the 103 unknown state bits of the other three LFSRs: $x^2_{0...30}$, $x^3_{0...32}$ and $x^4_{0...38}$. The main problem is that we should keep track of the current state of the FSM (c_t and c_{t-1}), which depends on the integer sum of the unknown LFSR bits. This leads us to the procedure described below:

1. Compute the exclusive-or of the current output of the FSM, the known output of LFSR1 and the current key stream bit z_t. For a correct guess, this will be equal to the exclusive-or of the three unknown LFSRs: $c^0_t \oplus x^1_t \oplus z_t = x^2_t \oplus x^3_t \oplus x^4_t$.
2. We now know the parity of the integer sum $x^2_t + x^3_t + x^4_t$, but this does not determine the sum uniquely. Both in the even and the odd situation, two possible values will have to be considered separately, yielding two branches of a search tree. The corresponding subset of values of x^2_t, x^3_t and x^4_t can be described by a set of equations in these variables (see Table 2). To proceed, we select one of the two branches and add the appropriate equations to $\mathcal{S}$.
3. When $t \geq 31$, $t \geq 33$ and $t \geq 39$, we respectively use the tap equations of LFSR2, LFSR3 and LFSR4 to express all variables as a linear function of the 103 initial state bits of the three unknown shift registers.
4. The consistency of $\mathcal{S}$ is checked and if we get contradictions, we backtrack in our exploration and consider the next branch. If all branches have been examined, we reject the current guess. On the other hand, if we still have a consistent $\mathcal{S}$ for $t > 132$, we can assume that we found the correct initial state. The argument for this is that the 132 internal state bits of the cipher are likely to be uniquely determined by any sequence of 132 key stream bits.
5. Using x^1_t and the integer sum chosen in step 2, we calculate the next FSM state and proceed with $t + 1$.

Apart from the fact that the algorithm branches into two separate cases after each step, the procedure above looks very similar to Fluhrer's base algorithm. An important difference, however, is that we will have to deal with nonlinear equations in $\mathcal{S}$ (see Table 2). As it is not immediately clear how to efficiently check the consistency of such a system, we will split up the system $\mathcal{S}$ in a linear set of equations $\mathcal{L}$ and a nonlinear set $\mathcal{N}$, and start with the analysis of an attack which simply ignores the nonlinear equations. Afterwards, we will try to use these equations in one way or another to improve the attack.

4.2 Analysis of the Search Tree

It is clear from the description of the algorithm in the previous subsection that the attacker needs to traverse an entire search tree in order to eliminate a single guess. Consequently, the size of this tree will be a determining factor for the complexity of the attack. To get a clear picture of the algorithm, the typical topology of the traversed tree is drawn in Figure 2 (the meaning of n is explained below).

In order to estimate the number of nodes in such a tree, we make a couple of assumptions about the linear equations collected in $\mathcal{L}$:

Table 2. Equations in $\mathcal{S}$.

Sum	$x_t^2 x_t^3 x_t^4$	Equations
Even		
		$x_t^2 = 0$
0	000	$x_t^3 = 0$
		$x_t^4 = 0$
	011	$x_t^2 \oplus x_t^3 \oplus x_t^4 = 0$
2	101	$(x_t^2 \oplus 1) \cdot (x_t^3 \oplus 1) = 0$
	110	
Odd		
	100	$x_t^2 \oplus x_t^3 \oplus x_t^4 = 1$
1	010	$x_t^2 \cdot x_t^3 = 0$
	001	
		$x_t^2 = 1$
3	111	$x_t^3 = 1$
		$x_t^4 = 1$

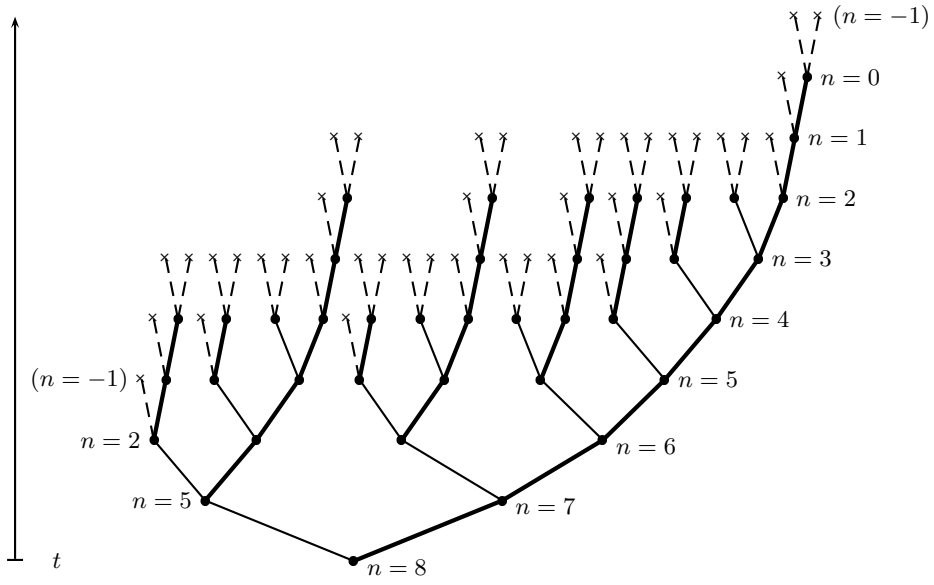

Fig. 2. Typical search tree for $n = 8$.

1. As long as $\mathcal{L}$ contains less equations than unknowns, these equations are independent.
2. $\mathcal{L}$ is inconsistent as soon as it contains more equations than unknowns.

Most of the time, we can safely make both assumptions, but when the number of unknowns and equations are almost equal, we might make errors. As both types of errors have a tendency to compensate each other, we do not expect these assumptions to influence the accuracy of our estimation too much. On the other hand, they will considerably simplify our analysis.

In our simplified model, each additional equation is independent of the previous ones and can thus be used to eliminate an unknown. Let n denote the number of independent unknowns in $\mathcal{L}$ at a certain point in the tree. The expected number of nodes N arising from that point can be expressed recursively by looking at its two immediate successor nodes, which correspond to the two cases that are checked individually at each level of the algorithm in order to get a unique value for $x_t^2 + x_t^3 + x_t^4$. These two cases are highly asymmetric, though. As shown in Table 2, the first one inserts three new linear equations in $\mathcal{L}$, whereas the second one adds only one linear equation (and one nonlinear equation, which we disregard in this first attack). Using each equation to eliminate one of the n unknowns, we obtain the following recursive relation:

$$N(n) = 1 + N(n-1) + N(n-3) \qquad n \geq 3 \tag{7}$$

This relation is also valid for $n = 0, 1$ and 2 if we set $N(-1) = N(-2) = N(-3) = 0$. Negative values of n can be interpreted as nodes where the number of equations exceeds the number of unknowns, and hence $\mathcal{L}$ is inconsistent by assumption 2. The branches to such nodes are indicated with dashed lines in Figure 2. Solving the recursion of (7) for $n \geq 0$, we get:

$$N(n) \approx 1.92 \cdot 1.47^n \tag{8}$$

Since the contents of the three longest LFSRs have to be determined, we start the algorithm with a total of $31 + 33 + 39 = 103$ independent unknowns. Accordingly, the expected number of nodes in the tree is given by:

$$N(103) = 2^{57.7} \tag{9}$$

We now have an estimation of the complexity of rejecting a single guess. Taking into account that we should repeat this procedure for about half of all the possible states of LFSR1 and the FSM ($25 + 4$ bits), we can easily calculate the total estimated amount of work required to reconstruct the initial state of the cipher:

$$\frac{1}{2} \cdot 2^{4+25} \cdot N(103) = 2^{85.7} \tag{10}$$

When we compare this result with the complexity of Fluhrer's attack (2^{85}), we must conclude that the basic version of our new attack is not likely to provide any gain at all. However, there is quite some room for improvement.

5 Improving the Attack

In this section, we will try to improve our attack by simplifying the search tree in two different ways. The first improvement consists in cutting out less 'cost-effective' branches. The second subsection describes a way to gain efficiency by trimming the top of the tree.

5.1 Skipping Branches

When we discussed the different situations of Table 2, we already mentioned the strong asymmetry between the two possible values of $x_t^2 + x_t^3 + x_t^4$, examined separately at each step in the algorithm. Using (7) and (8), we find that, at any given node, the branch corresponding with the single linear equation $x_t^2 \oplus x_t^3 \oplus x_t^4 = a$ contains $1.47^2 = 2.15$ as many nodes as the other branch. As a result, the algorithm will spend a factor 2.15 more time on this possibility than on the other one. On the other hand, it is considerably more likely that this choice will correspond with the correct values of x_t^2, x_t^3 and x_t^4. Indeed, when we take a look at the second column of Table 2, we see that the case with the single equation covers $\frac{3}{4}$ of the possible values.

Let us now define the efficiency of a branch as the ratio between the probability of being correct and the amount of work required to check this correctness. According to this definition, the single equation case is $3/2.15 = 1.4$ times more efficient than the case with the three linear equations.

The basic idea behind our optimization is to retain the most efficient branches only. The consequence of this, however, is that we introduce a certain probabilty of rejecting the correct guess and if this happens, the attack will fail. To compensate for this, we take the same approach as Fluhrer and restart the attack for other sequences of 132 consecutive known key stream bits until the attack succeeds. The most economical way of doing this, is by using bit sequences obtained by repeatedly shifting the starting position in the known key stream by one bit. When we reach the end of a data packet, we simply proceed with the first 132 bits of the next one.

When making our choice of which branches to examine, we must take the following two criteria into account:

1. Make the examined branches as efficient as possible.
2. Minimise the average number of times the attack needs to be restarted in order for the correct internal state to be located somewhere on the examined part of the tree.

To analyse the effect of focusing on a particular set of branches, we introduce a new sequence q_t determined by the three unknown LFSR streams as defined below:

$$q_t = x_t^2 \oplus x_t^3 \oplus x_t^4 \oplus x_t^2 x_t^3 \oplus x_t^3 x_t^4 \oplus x_t^4 x_t^2 \tag{11}$$

The right-hand side of this equation is nothing but a function which returns a 0 when $x_t^2 = x_t^3 = x_t^4$, i.e. in $\frac{1}{4}$ of the cases. Assuming that the initial contents of LFSR1 and the FSM were correctly guessed, the sequence q_t indicates for each t on which side of the tree the correct state is located ($q_t = 1$ corresponds to the single equation case). Consequently, if we choose to examine a particular branch consisting of a given sequence of efficient and less efficient choices, we will only succeed in detecting a correct guess when the corresponding pattern of 1's and 0's is found in q_t.

In terms of this new sequence q_t, the two criteria above come down to finding a set of branches such that:

1. The patterns corresponding with the branches contain as many 1's as possible.
2. The expected distance to the first occurrence of one of those patterns in q_t is as short as possible.

Guided by the considerations above, we choose to examine the set of branches determined by patterns given by:

$$\underbrace{(00)\overbrace{111(01)11\ldots1(01)11}^{(m-l)\text{ 1's and }l\text{ (01)'s}}}_{m+l+2\text{ bits}}$$

By allowing the patterns to contain a limited number of 0's, we dramatically increase the number of possible patterns, which augments the probability of finding a match in q_t. The two leading 0's, on the other hand, assure that no pair of patterns ever overlap. As a result, their occurrences will be equally dispersed in q_t and the expected distance between two consecutive occurrences is kept small[1].

Figure 3 illustrates the reduced search tree for the four patterns corresponding to $m = 4$ and $l = 1$. Bold lines indicate efficient transitions (corresponding to right branches in Figure 2).

To calculate the complexity of the tree, we must make a distinction between the lower ($t \leq m + l + 2$) and the upper part ($t > m + l + 2$). The lower part can once again be described recursively. Guided by Figure 3, we derive[2] the following relation for the number of nodes M:

$$M(m,l) = \begin{cases} 2 \cdot l + 3 & m = l \geq 0 \\ m + 3 & m \geq l = 0 \\ M(m-1,l-1) + M(m-1,l) & m > l > 0 \end{cases} \tag{12}$$

The solution of this recursive equation is found to be:

[1] Indeed, we don't want to optimize the *probability* of finding a pattern in q_t, but the expected *distance* to its first occurrence.

[2] To make clear how the recursion was derived, we should write the equation like this: $M(m,l) = 2 + 1 + (1 + M(m-1,l-1) - 2) + (M(m-1,l) - 2)$.

$$M(m,l) = \binom{m+1}{l} + \binom{m+2}{l+1} \tag{13}$$

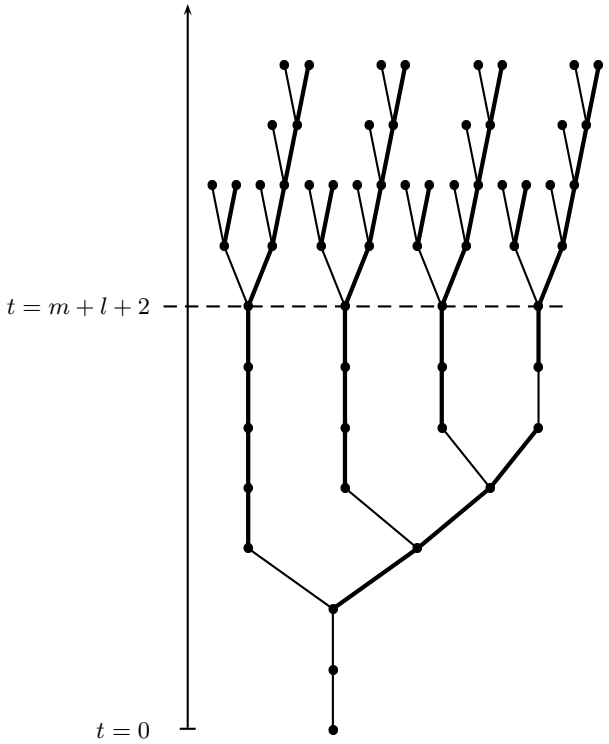

Fig. 3. Tree for $m = 4$ and $l = 1$.

When the algorithm reaches $t = m+l+2$ (the number of bits in the selected patterns), it starts exploring all possible branches for each of the $\binom{m}{l}$ patterns. Since $m+3\cdot(l+2)$ variables will have been eliminated at that point, we conclude that the number of nodes in the upper tree must equal:

$$\binom{m}{l} \cdot \Big(N(103 - m - 3\cdot(l+2)) - 1\Big) \tag{14}$$

By adding both parts, we obtain the complexity of the entire tree:

$$\begin{aligned} C &= M(m,l) + \binom{m}{l} \cdot \Big(N(103 - m - 3\cdot(l+2)) - 1\Big) \\ &= \binom{m}{l} \cdot \left[\frac{m+1}{l+1} \cdot \frac{m+l+3}{m-l+1} + \Big(N(103 - m - 3\cdot(l+2)) - 1\Big)\right] \end{aligned} \tag{15}$$

The next thing to examine is the minimum amount of known key stream bits required for this attack. The algorithm should be repeated at different starting positions until the sequence q_t matches one of the $\binom{m}{l}$ presupposed patterns. Bearing in mind that the occurrences in q_t are equally dispersed and that the probabilities to find a 1 or a 0 in this sequence are $\frac{3}{4}$ versus $\frac{1}{4}$, we expect a match after about:

$$\frac{1}{\left(\frac{3}{4}\right)^{\mathrm{m}}\left(\frac{1}{4}\right)^{\mathrm{l}+2}\cdot\binom{\mathrm{m}}{\mathrm{l}}} = \frac{1.33^{\mathrm{m}}\cdot 4^{\mathrm{l}+2}}{\binom{\mathrm{m}}{\mathrm{l}}} \text{ bits} \tag{16}$$

We are now able to make an estimation of the total complexity. Note that the entire tree should be traversed for each new starting position and for all possible values for the initial state of the FSM and LFSR1 ($4 + 25$ bits). This leads to the final expression:

$$\begin{aligned} C &= 2^{29}\cdot 1.33^{m}\cdot 4^{l+2}\cdot\left[\frac{m+1}{l+1}\cdot\frac{m+l+3}{m-l+1}+\Big(N(103-m-3\cdot(l+2))-1\Big)\right] \\ &\approx 2^{29}\cdot 1.33^{m}\cdot 4^{l+2}\cdot\frac{m+1}{l+1}\cdot\frac{m+l+3}{m-l+1}+2^{29}\cdot 1.10^{-m}\cdot 1.27^{l+2}\cdot N(103) \end{aligned} \tag{17}$$

The last remaining step is to choose adequate values for the parameters m and l. After evaluating (16) and (17) for different combinations of m and l, we retain the case $m = 66$ and $l = 4$, which seems to achieve a reasonable trade-off between efficiency and data requirements. For this choice, the resulting complexity is found to be 2^{80} for 1 Mbit of known key stream. By cutting out the less efficient branches, we clearly improved our basic attack (complexity 2^{86}), but again, we don't obtain a significantly better result than Fluhrer's own optimised attack (2^{81} steps for 1 Mbit).

5.2 Trimming at the Top

After having simplified the base of the search tree, we now try to reduce the complexity of the upper part by taking advantage of the nonlinear equations, collected at each state in a set $\mathcal{N}$ but disregarded thus far.

Let us return to the base algorithm of Subsection 4.1 and assume that, at a certain point, $103 - n$ variables have been eliminated. In order to find out what equations $\mathcal{N}$ typically consists of at this stage, we need to describe the average path leading to such points. Keeping in mind that an efficient branch gives rise to 2.15 as many nodes as the less efficient one and given the fact that they eliminate 1 and 3 variables respectively, we may expect that an average path reaching this level will consist of k transitions of the efficient type and $k/2.15$ of the other type with:

$$k + 3\cdot\frac{k}{2.15} = 103 - n \tag{18}$$

or

$$k = 2.15 \cdot \frac{103 - n}{3 + 2.15} \tag{19}$$

At the same time, we will have collected k nonlinear equations relating the remaining n variables. Considering the fact that each individual nonlinear equation has a probability of $\frac{3}{4}$ to be fulfilled (see Table 2), we can say that a single equation contains $-\log_2 \frac{3}{4} = 0.42$ bits of information about the n variables. Under the assumption that these equations are independent, we expect that this nonlinear system $\mathcal{N}$ will contain inconsistencies as soon as:

$$-\log_2 \frac{3}{4} \cdot k > n \tag{20}$$

If we solve this equation after substitution of (19), we obtain that inconsistencies start appearing when $n < 15$. Suppose now that we have a way to detect these inconsistencies. This would mean that we only need to explore the tree until we reach $n \approx 15$. Consequently, the number of nodes to check would be reduced with a factor $1.47^{15} = 2^{8.3}$, as derived from (8).

Of course, the essential problem still to be resolved, is the detection of inconsistencies in $\mathcal{N}$. Obviously, we don't want to solve this by simply examining all possible values for the n variables and making sure that no solution exists. This approach would require going over 2^{15} possibilities, which would actually increase the total number of nodes to check. In the subsequent paragraphs, we will describe a method which only exploits a fraction of the information delivered by the nonlinear equations, but has the advantage that its implementation is very simple.

Our approach will take advantage of the particular form of the nonlinear equations in $\mathcal{N}$. Moreover, instead of trying to solve these nonlinear equations directly, we will wait until we can use $\mathcal{L}$ to turn them into linear equations. As shown in Table 2, the general form for the nonlinear equations is

$$(x_t^2 \oplus a) \cdot (x_t^3 \oplus a) = 0 \tag{21}$$

In step 3 of our basic algorithm, the variables x_t^2 and x_t^3 are replaced by linear combinations of $x_0^2 \ldots x_{30}^2$ and $x_0^3 \ldots x_{32}^3$. As we climb up in the tree and process more linear equations, an increasing part of these 31+33 variables are eliminated on their turn. Let $\{x_1, x_2, \ldots, x_n\}$ denote the set of n remaining independent variables in $\mathcal{L}$ at some point in the tree. Rewriting the k equations in $\mathcal{N}$ as a function of these n unknowns, we obtain expressions of the form:

$$(a_i^0 \oplus a_i^1 x_1 \oplus \ldots \oplus a_i^n x_n) \cdot (b_i^0 \oplus b_i^1 x_1 \oplus \ldots \oplus b_i^n x_n) = 0 \quad \text{for } 0 \leq i < k \tag{22}$$

Suppose now that at a certain point we find that $a_i^0 = 1$ and $a_i^1 = \ldots = a_i^n = 0$ for some i. In this case, the nonlinear equation would be reduced to the linear equation $1 \cdot (b_i^0 \oplus b_i^1 x_1 \oplus \ldots \oplus b_i^n x_n) = 0$, which we can simply add to $\mathcal{L}$. The same effect would occur when $b_i^0 = 1$ and $b_i^j = 0$ for $1 \leq j \leq n$. It is quite unreasonable to hope to find such a reduced equation at the bottom of the tree,

but as n decreases and the number of nonlinear equations k augments, so does the probability of being able to exploit one of them. Assuming that the binary values a_i^j and b_i^j are balanced and independent, we expect that at least one of the k equations in $\mathcal{N}$ will be reduced when:

$$2 \cdot k \geq 2^{n+1} \tag{23}$$

Once we have found such an equation, removed it from $\mathcal{N}$ and used it to eliminate a variable, it is likely that a chain reaction will be triggered. This can be explained by the fact that the inequality (23) will still be fulfilled when both the number of nonlinear equations k and the number of independent variables n are decremented. We can therefore expect that the elimination of the additional variable will in turn cause the reduction of another nonlinear equation in $\mathcal{N}$. This goes on until we run into an inconsistency.

As demonstrated above, we now have a method to detect inconsistencies as soon as (23) is satisfied. Combining this inequality with (19), we find that the method can be applied when $n \leq 5$. This result shows that inconsistencies will be detected much later than theoretically possible, but the total number of nodes is still reduced with a modest factor $1.47^5 = 2^{2.8}$.

This last result, together with the complexity calculated in the previous subsection, leads to the conclusion that our final attack should be capable of recovering the internal state of the cipher after about 2^{77} steps, given 1 Mbit of known key stream bits.

6 Simulations

When we estimated the number of nodes $N(n)$ in the basic search tree in Subsection 4.2, we made a number of assumptions. In this section we will perform some simulations to verify the accuracy of our theoretical results.

As computed in (9), traversing a full tree requires the inspection of about $2^{58} \approx 10^{17}$ nodes. This far exceeds the capacity of our computer. To get a good estimation of the real complexity anyway, we note that, due to the assumptions we made, the difference between the theoretical model and the simulation will most likely be determined by the shape of the tree for small values of n. We will therefore only explore the upper part of the tree. More precisely, our simulation will randomly select one of the two possible branches for the first 34 steps of the algorithm, and start scanning the full tree from that point on. This is repeated 32 times to obtain average values for $N(n)$. The number of nodes checked during one run appears to fluctuate between $3 \cdot 10^1$ and $8 \cdot 10^7$, depending on where we end up after the first random steps. This is not surprising, considering the tree's asymmetric structure, shown in Figure 2.

The simulated values of $N(n)$ for our basic algorithm are plotted in Figure 4. When we compare these results to the theoretical calculations indicated by the dashed curve, we see that the error is very small and that the difference between the two curves is, as predicted, mainly due to an offset for small values of n.

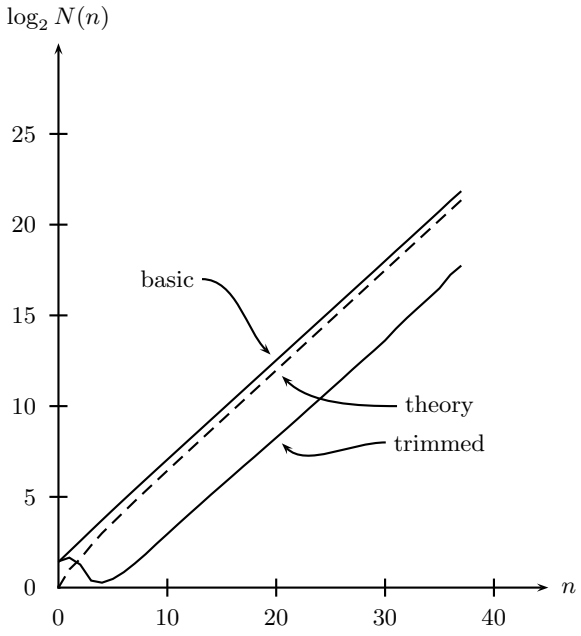

Fig. 4. Theoretical and simulated number of nodes N as a function of n.

In order to verify the theoretical predictions made in Subsection 5.2, we restart the simulation, but this time we try to check the consistency of the non-linear equations by looking for reduced equations as described earlier. The results plotted in Figure 4 clearly show that the number of nodes drop to zero around $n = 5$. This confirms the statements made in Subsection 5.2. Moreover, the gain induced by performing this trimming appears to be slightly larger than expected, which, by extrapolation, reduces the total complexity of our final attack to 2^{76}.

7 Conclusions

We have derived a new attack based on a previous guess and determine attack by Scott Fluhrer, capable of recovering the internal state of the E_0 key stream generator in fewer steps (2^{76} vs. 2^{81}), given the same amount of known key stream bits (1 Mbit). The operations in our algorithm are more complicated, however, and their exact implementation should be further examined in order to conclude whether or not the attack is faster in practice.

Either way, it is clear that both attacks remain theoretical, in the sense that the obtained complexities are still enormous compared to the computational power available in a practical attack. Nevertheless, the algorithms are much more efficient than an exhaustive key search and may indicate that the Bluetooth encryption engine does not make use of its 128 secret key bits in the most optimal way.