Linear Consistency Test (LCT) in cryptanalysis of irregularly clocked LFSRs in the presence of noise

Guro Bu

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2011

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Linear Consistency Test (LCT) in cryptanalysis of irregularly clocked LFSRs in the presence of noise

Guro Bu

2011/07/01

Abstract

Keystream generators using irregular clocking are frequently used to generate the keystream in a stream cipher. These generators are typically composed of two Linear Feedback Shift Registers (LFSRs), where the clocking of the second LFSR is controlled by the output sequence from the first LFSR.

Various attacks exist to cryptanalyse such generators. Among the more popular are correlation attacks and algebraic attacks. When only the ciphertext is known, the usual approach is to use correlation methods. These attacks are accomplished in two stages. First, the initial state of the clocked LFSR is found, then the clock control sequence is reconstructed. Algebraic attacks, on the other hand, have only been applied in the known-plaintext scenario to cryptanalyse irregularly clocked generators.

The Linear Consistency Test (LCT) is an algebraic key recovery attack which uses some guessed bits from the internal state of the generator to set up a set of equations used to determine whether the initial guess was correct or not. In this thesis, an extension of the LCT attack is implemented that successfully reconstructs the clock control sequence in the ciphertext only scenario. Experiments show that this attack can handle low to moderate levels of noise, and it is estimated to perform better than correlation attacks on irregularly clocked generators.

Sammendrag

Nøkkelstrømgeneratorer med uregelmessig klokking blir ofte brukt for å generere nøkkelstrømmen i et flytchiffer. Disse generatorene er typisk bygd opp av to lineært tilbakekoblede skiftregistre (LFSRer) hvor klokkingen av det andre LFSRet kontrolleres av utdatasekvensen fra det første LFSRet.

Forskjellige typer angrep eksisterer for å knekke slike generatorer. Korrelasjonsangrep og algebraiske angrep er metoder som er mye brukt. Når bare chifferteksten er kjent, er det vanlig å bruke korrelasjonsangrep. Disse angrepene utføres i to trinn. Først blir starttilstanden til det klokkede LFSRet funnet, deretter rekonstrueres klokkekontrollsekvensen. Algebraiske angrep derimot, er bare blitt brukt i kjent klartekstscenariet i kryptoanalyse av uregelmessig klokkede generatorer.

Lineær Konsistenstest (LKT) er et algebraisk nøkkelgjenfinningsangrep som bruker kjente bit fra den indre tilstanden til en generator til å sette opp et ligningssett som brukes for å bestemme om den gjettede starttilstanden var riktig eller ikke. I denne oppgaven implementeres en utvidelse av LKT-angrepet som lykkes i å rekonstruere klokkekontrollsekvensen i bare-chiffertekst-scenariet. Eksperimenter viser at dette angrepet kan håndtere lave til moderate støynivåer, og det er estimert til å ha bedre ytelse enn korrelasjonsangrep på uregelmessig klokkede generatorer.

Acknowledgements

First of all I would like to thank my supervisor, Slobodan Petrović. He suggested the topic for my thesis and has always been available to answer my questions.

Also, I would like to thank my classmate Andreas Bråthen for valuable feedback as my opponent.

Guro Bu, 1st July 2011

Contents

List of Figures

List of Tables

1 Introduction

1.1 Topic covered

During the last three decades, the interest for stream cipher analysis has increased in the academic community. In a stream cipher, the plaintext bits are added modulo 2 to a keystream sequence to yield the ciphertext. Pseudorandom number generators are commonly used to produce the keystream sequence in a stream cipher.

Linear Feedback Shift Registers (LFSRs) produce a sequence of bits from a shorter seed. Since, if designed properly, the output sequence of an LFSR has statistical properties similar to those of a truly random sequence, the LFSR would be a good candidate for a pseudorandom number generator. But LFSRs are linear devices, and effective algorithms exist to recover the initial state of the LFSR (the secret key in the cipher). Therefore, LFSRs should never be used by themselves as keystream generators.

However, several techniques exist to destroy the linearity of LFSRs and thus make them adequate for use in cryptography. One method is to use two LFSRs, where the output of the first LFSR controls the clocking (stepping) of the second, and the output of the second LFSR is the keystream. Such generators are known as irregularly clocked generators, and are the topic of this thesis. The Binary Rate Multiplier (BRM) [2] is a typical representative of these schemes, and is widely used in practice.

To cryptanalyse an irregularly clocked generator, different methods can be employed. Among the more popular attacks are correlation attacks and algebraic attacks. The Linear Consistency Test (LCT) is an example of an algebraic attack. The LCT is a key recovery attack which uses algebraic methods and some guessed bits from the internal state of an irregularly clocked generator to determine the unknown bits of the key and to accept or recject the guessed initial state. Correlation attacks, on the other hand, are based on the existence of statistical dependencies between the output of one of the component LFSRs and the output of the generator, and consider the distance between a pair of sequences to determine whether a guessed initial state is the correct one.

This work discusses the possibility of using the Linear Consistency Test in the cryptanalysis of irregularly clocked generators when only the ciphertext is known. Additionally, it attempts to determine whether such an attack would perform better than correlation attacks on this type of generator.

1.2 Keywords

Cryptanalysis, Linear Consistency Test, Correlation attack, Stream cipher, Irregularly clocked shift registers

1.3 Problem description

The Linear Consistency Test is a method of cryptanalysis that has been applied previously to irregularly clocked shift registers [3].

The Linear Consistency Test was originally suggested by Zeng, Yang and Rao in [4], where it was applied to the generators of Jennings [5] and Massey-Rueppel [6]. The attack has also been applied to the E0 cipher used in Bluetooth [7].

The authors of [4] first prove a theorem on the consistency probability of a system of linear algebraic equations, and then apply the theorem to disclose the key of the two pseudorandom generators.

The LCT procedure is as follows: First, a candidate subkey is guessed. Then a system of equations parametrized by this subkey is set up. If the candidate subkey coincides with the subkey used in generating the captured sequence, then the set of equations will be consistent. But if the candidate subkey is not the subkey used, then by the theorem of [4] the consistency probability of the system will be very small when the captured sequence is long enough. The consistency of the system of equations is tested for all possible choices of the candidate subkey, and the right subkey is found whenever the system is found to be consistent. The system of equations can be solved for example by means of the Gaussian algorithm.

Molland [3] developed an improved Linear Consistency Test. The method employs a low weight cyclic equation rather than the Gaussian algorithm to test for consistency, and thus the performance of the LCT attack is significantly improved. Molland [3], however, considered only the case without noise, that is, it is assumed that the keystream sequence (the output of the generator) is known to the cryptanalyst. This is a known plaintext attack, which is an unlikely scenario in practice.

A more common situation is that only the ciphertext is known to the cryptanalyst. In this case, the plaintext introduces noise in the system, and the input to the attack algorithm is a noisy version of the output of the generator. The LCT has not previously been applied to irregularly clocked generators in the ciphertext only scenario.

1.4 Justification, motivation and benefits

In the cryptanalysis of stream ciphers, a significant part of the research conducted assumes that the plaintext itself is available and used as input to the attack algorithms. However, some research has been carried out that considers correlation attacks in the ciphertext-only scenario (see for example [8]). These attacks have a complexity of $2^{l_s} + 2^{l_u}$ (where l_s is the length of the clocking register and l_u is the length of the clocked register). In other words, the performance is dependent on the length of both registers.

In [3] it was proved that the performance of the Linear Consistency Test, in the known-plaintext scenario, is dependent only on the length of the clocking register. Unfortunately, this attack has not been implemented for the ciphertext-only scenario.

Since in practical applications it is more likely that only the ciphertext is available to the attacker, if it turns out that a special implemntation of an essentially algebraic attack can perform better than correlation attacks for irregularly clocked shift registers, then this kind of generators must be considered less secure than previously believed.

In this thesis, we examine whether the Linear Consistency Test can successfully recover the secret key in the ciphertext-only scenario, and make an attempt at predicting the performance of such an attack.

1.5 Research questions

Based on the previous discussion, several research questions were identified, that this thesis aims to answer.

1. Can the Linear Consistency Test (LCT) be successful in revealing the key of an irregularly clocking scheme based on the Binary Rate Multiplier (BRM) when only the ciphertext is known?
2. Is the LCT efficient compared to other attacks on the BRM?

1.6 Contributions

This thesis investigates the feasibility of the Linear Consistency Test in the presence of noise. A keystream generator based on the Binary Rate Multiplier scheme is studied. The Linear Consistency Test has been applied to this type of generator previously, but only in the known-plaintext scenario. In real life applications, however, the ciphertext-only attack is more realistic, since it only requires a passive attacker. In such scenarios other methods, such as correlation attacks, are commonly employed.

However, this thesis shows that essentially algebraic attacks, or more specifically the Linear Consistency Test, can be successful in the ciphertext only scenario.

In [3], the performance of the LCT is proven to depend only on the clocking register (in the known-plaintext scenario), whereas the performance of correlation attacks in the ciphertext-only scenario depends on both registers. In this thesis, it will be shown that it is possible that the LCT can perform better than correlation attacks in the ciphertext-only scenario.

A proof of concept is carried out in order to examine the feasibility of such an attack and to estimate its performance.

2 State of the art

2.1 Background

Stream ciphers play an important role in protecting communications in the high frequency domain (e.g. wireless networks and mobile communications), where speed and simplicity of implementation in hardware are important issues.

In a stream cipher, the plaintext is added modulo 2 to a keystream sequence to yeild the ciphertext. Pseudorandom number generators are commonly used to produce the keystream sequence in a stream cipher.

In this section, some background material on cryptography and the theory on which irregularly clocked shift registers are built is summarized.

2.1.1 Cryptography

The field of cryptography can be classified into two main areas: symmetric-key cryptography and public-key cryptography.

In symmetric-key cryptography the sender and the receiver share a common key that is used for both encryption and decryption, and the common key is always kept secret. Examples of symmetric-key cryptosystems include the Data Encryption Standard (DES), the Advanced Encryption Standard (AES) and RC4.

In public-key cryptography, on the other hand, different keys are used for encryption and decryption. The key used to encrypt the message is made public, while the decryption key is kept secret. The Diffie-Hellman and RSA algorithms are well-known and widely used examples of public-key cryptosystems.

Symmetric-key cryptography is further classified into block ciphers and stream ciphers. In block ciphers, the plaintext is divided into to blocks of size n, and each block is encrypted independently.

In stream ciphers, the plaintext is encrypted bit-by-bit. The private key in the system is input to an algorithm which creates a long pseudorandom sequence known as the keystream sequence. These algorithms are commonly known as pseudorandom number generators (PRNGs). Encryption is performed by adding modulo 2 the sequence of plaintext bits to the keystream sequence to produce the ciphertext.

2.1.2 Pseudorandom number generators

The history of stream ciphers started around 1918 with the development of the one-time-pad cipher [9]. The encryption procedure is as follows: Each bit of the plaintext is added modulo 2 with a bit from a secret random key of the same length as the plaintext to form the ciphertext. In 1949 Claude Shannon proved the one-time-pad to be perfectly secret, i.e. unbreakable [10]. A major drawback of the cipher, however, is that the key must be as long as the plaintext, which makes key distribution and key management difficult [11].

Thus, the stream cipher scheme evolved to address this problem. Rather than using a truly random key of the same length as the plaintext, this scheme employs a shorter (secret) random key as a seed to a pseudorandom number generator, which generates an infinite sequence that *appears* to be random.

In [12], three requirements for cryptographically secure keystream generators are summarized:

(1) The period of the keystream must be large enough to accommodate the length of the transmitted message.
(2) The output bits must be easy to generate.
(3) The output bits must be hard to predict. Given the generator and the first n output bits, $a(0), a(1), ..., a(n-1)$, it should be computationally infeasible to predict the $(n+1)^{th}$ bit $a(n)$ in the sequence with better than a 50-50 chance. That is, given a portion of the output sequence, the cryptanalyst should not generate other bits forward or backward.

Several methods exist for generating pseudorandom sequences, using for instance ([12]) Linear congruence generators (LCGs), Nonlinear feedback shift registers (NLFSRs) or Linear feedback shift registers (LFSRs). The LFSR is the most commonly used basic component of keystream generators, and will be described briefly in the following.

2.1.3 Linear Feedback Shift Registers

A feedback shift register (FSR) of length n consists of n flip-flops, or stages, and a feedback function that expresses each new element of the ouput sequence as a function of the n previous elements. Each stage can store one bit, and a clock controls the movement of data. At each clock pulse, the contents of the stages are shifted one position. The key of the FSR is the initial contents (the initial state) of the FSR.

A linear feedback shift register is an FSR that has a feedback function of the form $a(t) = c_1 a(t-1) \oplus c_2 a(t-2) \oplus ... \oplus c_{n-1} a(t-n+1) \oplus a(t-n)$, where $a(t)$ is the output at time t, and the c_i are feedback coefficients with values in $\{0,1\}$. That is, each output bit is expressed as a linear combination of the previous elements. Any such recurrence relation can be represented by a characteristic feedback polynomial $f(x) = 1 + c_1 x + ... + c_{n-1} x^{n-1} + x^n$ over GF(2) [13].

The authors of [13] further describe two important properties of polynomials over GF(2) which determine the characteristics of the output sequence. A polynomial f(x) is *irreducible* if the only polynomials which divide it are 1 and f(x) itself. An irreducible polynomial is called *primitive* if f(x) divides $x^e + 1$ where $e = 2^{n-1}$ but f(x) does not divide $x^r + 1$ for any $r \in \langle 0, e \rangle$.

If a polynomial is irreducible, the length of the output sequence does not depend on the initial state (except for the all-zero state). That is, all possible initial states except the all-zero state produce output sequences of the same length. If it is also primitive, the period of any output sequence of the LFSR is $2^L - 1$ (maximum period), where L is the length of the LFSR. In order to be adequate for use in cryptography, the characteristic polynomial must be both irreducible and primitive.

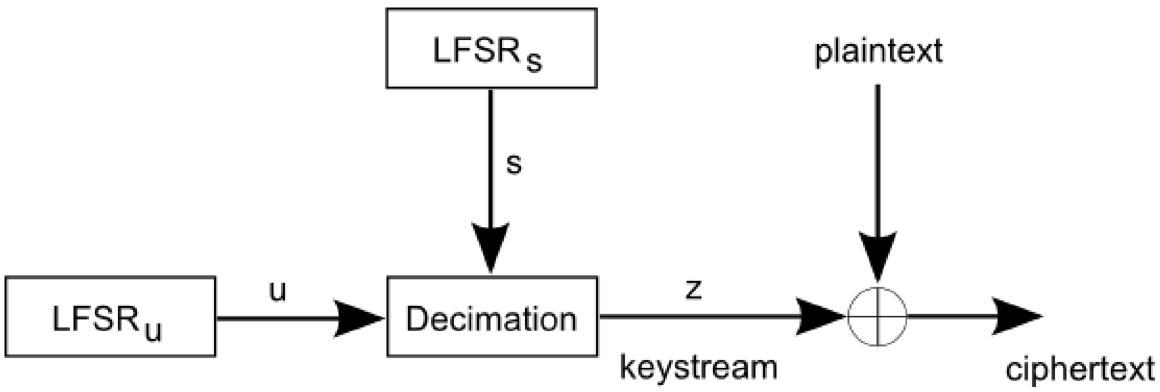

Figure 1: Stream cipher with an irregularly clocked generator

2.1.4 Irregularly clocked generators

To cryptanalyse an LFSR, if the feedback polynomial is not known, the Berlekamp-Massey algorithm can be applied to some consecutive $2n$ bits of the intercepted sequence generated by the LFSR [14], where n is the length of the LFSR. The algorithm takes as input the sequence, and outputs the characteristic polynomial, the length and the initial state of the shortest LFSR capable of generating the input sequence. The *linear complexity* (LC) of the intercepted sequence is defined as the length of such a minimum LFSR.

But if the linear complexity is very high, the algorithm cannot be applied because the computational complexity of the Berlekamp-Massey algorithm is quadratic in the length of the LFSR. Thus, we should attempt to increase the linear complexity, while preserving good statistical properties of the sequence.

Several techniques exist to accomplish this [15]. One possibility is to use several LFSRs in parallel. Then the keystream is generated as a non-linear function of the outputs of the component LFSRs. These generators are called *non-linear combination generators*. Another possibility is to generate the keystream as some non-linear function of the stages of a single LFSR. Such generators are called *non-linear filter generators*.

Irregularly clocked generators on the other hand, introduce non-linearity by having the output of one LFSR control the clocking of a second LFSR, as illustrated in Figure 1. Examples of such generators are the Binary Rate Multiplier [2] and the Shrinking Generator [16]. The Alternating Step Generator [17] is another type of irregularly clocked generators that consists of three LFSRs instead of two. In this project, the Binary Rate Multiplier will be considered.

The Binary Rate Multiplier consists of two shift registers, which in this thesis are referred to as LFSR_s and LFSR_u. LFSR_s is an m-length shift register with a primitive feedback polynomial generating a sequence of period $M = 2^m - 1$ and LFSR_u is an n-length shift register generating a sequence of period $N = 2^n - 1$. At time t, both LFSR_s and LFSR_u are clocked. Then LFSR_u is clocked further a_t times, where a_t is the integer represented by the contents of *k* fixed stages of LFSR_s. The output of the system is the output of LFSR_u after both registers have been clocked once and LFSR_u has been clocked a further a_t times. [2] further shows that the linear complexity

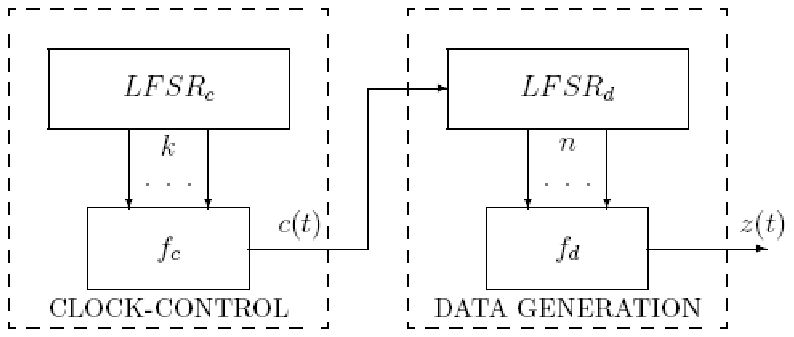

Figure 2: Overview of LILI keystream generators [1].

of the system is nM and the period of the generated sequence is MN if certain conditions are met.

2.2 The LILI-128 Keystream Generator

In [1], the LILI-128 Keystream generator, which is an implementation of the general BRM scheme, is described. The components of the generator can be grouped into two subsystems - the clock-control subsystem and the data-generation subsystem, as illustrated in Figure 2. Here, $LFSR_c$ corresponds to $LFSR_s$ and $LFSR_d$ to $LFSR_u$ in the general model.

$LFSR_c$ is a regularly clocked LFSR of length 39. The feedback polynomial of $LFSR_c$ is the primitive polynomial:

$$x^{39} + x^{35} + x^{33} + x^{31} + x^{17} + x^{15} + x^{14} + x^2 + 1$$

Since the polynomial is primitive, $LFSR_c$ produces a maximum-length sequence of period $P_c = 2^{39} - 1$.

A function f_c determines how many times the data generating register $LFSR_d$ should be clocked before output is taken from the system. At time t, the contents of stages 12 and 20 of $LFSR_c$ are input to the function. The output of the function is an integer c(t) in the range $[1, 4]$, and the function is given by

$$f_c(x_{12}, x_{20}) = 2(x_{12}) + x_{20} + 1$$

The data-generation subsystem consists of the register $LFSR_d$ of length 89 and a function f_d. The integer sequence c described above controls the clocking of $LFSR_d$, hence $LFSR_d$ is irregularly clocked. The contents of a fixed set of 10 stages of $LFSR_d$ are input to a Boolean function f_d, and the output of this function is the keystream bit z(t). When z(t) is produced, both LFSRs are clocked and the process is repeated to yield the keystream bit z(t+1).

The feedback polynomial of $LFSR_d$ is the primitive polynomial

$$x^{89} + x^{83} + x^{80} + x^{55} + x^{53} + x^{42} + x^{39} + x + 1$$

and $LFSR_d$ produces a maximum-length sequence of period $P_d = 2^{89} - 1$ (a Mersenne Prime).

The period of the keystream z(t) is $P_z = (2^{39}-1)(2^{89}-1) \approx 2^{128}$ and the linear complexity is conjectured to be at least 2^{68} [1]. The authors of [1] further examine the security of LILI-128 and conclude that LILI-128 is a secure cipher based on the conjecture that the complexity of divide and conquer attacks on LILI are at least 2^{112} operations. However, in [18] it is demonstrated that an attack with a significantly lower complexity is possible on LILI-128.

2.3 Attacks on stream ciphers

In this section two types of attacks on LFSRs are briefly explained; algebraic attacks and correlation attacks.

2.3.1 Algebraic attacks on LFSRs

In [13], general algebraic attacks on LFSRs are described. Two different scenarios are considered:

1. The cryptanalyst knows $2n$ consecutive bits of the keystream sequence and aims to find the feedback function.
2. The cryptanalyst knows at least $2n$ bits of the sequence, but the bits are not necessarily consecutive. The goal is to find the initial state of the LFSR as well as the feedback function.

When $2n$ consecutive bits are known

This method assumes that we know a subsequence $s_r, s_{r+1}, ..., s_{r+2n-1}$ of the keystream sequence, where n is the linear complexity of the sequence.

A linear shift register is defined as a shift register where the feedback function can be written in the form $f(s_0, s_1, ..., s_{n-1}) = c_0 s_0 + c_1 s_1 + ... + c_{n-1} s_{n-1}$. Here, the s_i's are the stages of the register and n the length of the register. Furthermore, each feedback coefficient c_i is 0 or 1 and all addition is over GF(2) [13].

Thus, each bit is a linear combination of previous bits, and the output bit s_{t+n} of the register is given by the recurrence relation $s_{t+n} = \sum_{i=0}^{n-1} c_i s_{t+i}$. This recurrence relation can be written as a matrix equation $\mathbf{s} = \mathbf{Sc}$ where

$$\mathbf{s} = \mathbf{s_{r+n}} = \begin{pmatrix} s_{r+n} \\ s_{r+n+1} \\ \vdots \\ s_{r+2n-1} \end{pmatrix}, \mathbf{c} = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix}$$

and $\mathbf{S} = (s_{ij})$ is the n by n matrix with $s_{ij} = s_{r+i+j-2}$ i.e.:

$$\mathbf{S} = \begin{pmatrix} s_r & s_{r+1} & \cdots & s_{r+n-1} \\ s_{r+1} & s_{r+2} & \cdots & s_{r+n} \\ \vdots & \vdots & & \vdots \\ s_{r+n-1} & s_{r+n} & \cdots & s_{r+2n-2} \end{pmatrix}$$

[13] shows that $\mathbf{S}$ is non-singular and has an inverse, and thus the feedback constants are determined by calculating $\mathbf{c} = \mathbf{S}^{-1}\mathbf{s}$.

When $2n$ non-consecutive bits of the sequence are known

From the keystream sequence, for instance 1?101??0?1??0?11???0?10, $2n$ consecutive positions are selected for which the maximum number of entries is known. Then, every possibility for filling the remaining positions are tried in turn. For each guess, the method outlined above is applied. Then the resulting feedback function is used to generate the first terms of the sequence. If this sequence disagrees with the original sequence, the guess is eliminated. The number of possibilities to try is obviously 2^m, where m is the number of question marks in the selected subsequence.

An alternative method in this case is to guess the feedback constants and use the known bits of the sequence to find the initial state. Then a sequence is generated from each guess of the feedback function and the corresponding initial state, and if this sequence differs from the original sequence in the known bits, the guessed feedback constants are eliminated.

Since both methods require approximately the same amount of time per trial, the method requiring fewer trials is usually chosen [13].

2.3.2 Correlation attacks

In a keystream generator consisting of multiple LFSRs, the output sequence of one of the component LFSRs (the target LFSR) can be correlated to the output of the generator. If $x_1, x_2, ...$ denotes the bits of the LFSR output sequence and $z_1, z_2, ...$ the keystream bits, then there is a correlation if $P(x_i = z_i) \neq 0.5$, and it may be possible to restore the initial state of the target LFSR independently of the initial states of the other LFSRs by means of a correlation attack.

The correlation between the output sequence of the target LFSR and the keystream sequence can be calculated by the formula [19]

$$\sum_{t=0}^{N-1} (-1)^{z_t + x_t \mathrm{mod} 2}$$

, where the sum is defined over real numbers. This correlation can be compared to the correlation between the keystream and another sequence r that is independent of z.

The first correlation attack on stream ciphers was proposed by [20]. The attack is performed by an exhaustive search on the initial state of each component LFSR, and an initial state is accepted if the magnitude of the correlation exceeds a certain decision threshold. The number of trials for each LFSR is $2^{L_{i-1}}$, thus the complexity of the attack is $n \times 2^{L_{i-1}}$ where n is the number of component LFSRs. The attack as described here is a known plaintext attack, but when there exists redundancy in the plaintext, a similar ciphertext only attack is also possible [19,20]. The attack proposed by [20] applies only to combination generators, but variants of this original attack can be applied to other keystream generators [19].

As stated earlier, in the algorithm proposed by [20], exhaustive search is used. Later, more efficient correlation alrgorithms were developed that required output sequences of larger length, these attacks are known as *Fast correlation attacks* [18, 21–24].

The algorithms for fast correlation attacks typically consist of two phases. In the first phase, suitable parity check equations are generated from the LFSR feedback polynomial. In the second phase, these equations are input to a decoding algorithm to restore the initial state of the LFSR.

The common approach in proposed correlation attacks is to see the problem of restoring the LFSR initial state as a decoding problem. [22] describes the model for such an attack as follows. The set of all possible output sequences of the target LFSR is regarded as a linear $[N, l]$ block code, denoted C, where l is the degree of the feedback polynomial of the LFSR and N is the sequence length. The LFSR sequence x (which is regarded as a codeword from C) is transmitted over a binary symmetric channel with an error probability p, and the observed sequence z is regarded as the received channel output. The correlation probability is defined by $1 - p = P(x_i = z_i)$. Thus, the problem of restoring the LFSR's initial state $x_1, x_2, ..., x_l$ is redefined as the problem of finding the length N codeword from C that was transmitted. This problem is solved by decoding the code C.

[23] presents a fast correlation algorithm with reduced memory requirements compared to previous algorithms, and derives theoretical estimates on the computational complexity. Rather than decoding the $[N, l]$ associated with the target LFSR, where l is the degree of the feedback polynomial, [23] associates with the target LFSR another linear code $[n_2, k]$. Here, $k < l$ and the codeword is considered to have passed through another channel with a 'double' noise level p_2. [23] shows that if the length n_2 of the new code is at least $\lceil k/C(p_2) \rceil$ (where $C(p_2)$ is the capacity of the code) the decoding of this code will recover the first k bits of the initial state of the target LFSR.

2.4 Attacks on irregularly clocked generators

The previous section described general attacks on LFSRs and LFSR based generators. In this section, the focus is on attacks on irregularly clocked generators.

For these generators, when only the ciphertext is known, correlation attacks are typically applied, whereas when the plaintext is known, algebraic or correlation attacks can be applied.

This section will begin with a short explanation of correlation attacks, followed by a more detailed description of the Linear Consistency Test as implemented by [3].

2.4.1 Correlation attacks

The ciphertext-only attack introduced by [20], mentioned above, is based on the Hamming distance measure, and thus the attack is applicable for regularly (but not irregularly) clocked LFSRs. Based on the work of [20], [25] introduced a generalized correlation attack employing the Levenshtein distance (also known as the edit distance) rather than the Hamming distance. Thus, the attack is aimed at generators consisting of irregularly clocked LFSRs. This ciphertext-only attack determines a set of candidate initial states for the clocked LFSR, arranged in order of increasing value of the edit distance between the ciphertext and the LFSR sequence, with the smallest distance corresponding to the most likely solution. However, the attack is only applicable when the combining function is memoryless and zero-order correlation immune (which means that the output of the generator is correlated to at least one input). The attack was extended in [26] to arbitrary *m*th-order correlation immune functions with memory.

While [25] determines the candidate initial states for the clocked LFSR, [8] also considers reconstruction of the initial state of the clocking LFSR. The attack proposed by [8] is a ciphertext only attack, which reconstructs the clock control sequence by means of a directed depth-first like search through the edit distance matrix. The complete attack consists of two phases [8]:

1. Reconstruction of candidate initial states ($LFSR_u$)
2. Clock control sequence reconstruction ($LFSR_s$)

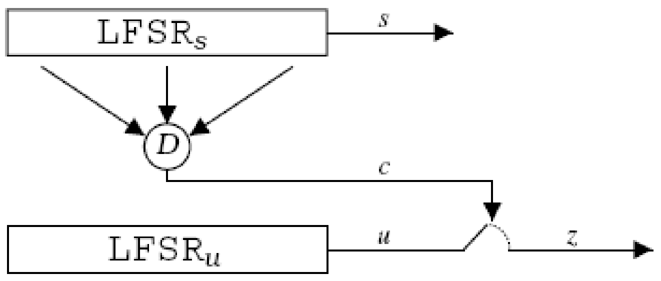

Figure 3: The general model for irregularly clocked keystream generators [3].

In the first phase, a set of candidates for the initial state of LFSR$_u$ is determined. This phase corresponds to the attack of [25] described above.

In the second phase, a candidate initial state for LFSR$_u$ is fixed, and candidates for the initial state of LFSR$_s$ are determined. Together, the candidate for LFSR$_u$ and the set of candidates for LFSR$_s$ could generate the intercepted sequence. First, the edit distance matrix is filled with the values of the edit distance together with four associated vectors. Then the candidate clock control sequences are reconstructed by searching for optimal paths and suboptimal paths through the matrix using a special depth-first like search algorithm. Here, the optimal paths through the matrix are those corresponding to the clock control sequences when the noise level is 0. In addition, suboptimal paths must be reconstructed, since with noise in the system, the clock control sequences corresponding to the optimal paths do not necessarily generate the captured output sequence. The suboptimal paths to be reconstructed are those whose weight-difference from the optimal ones does not overcome a certain discrepancy D depending on the noise level.

The complete attack has a time complexity of $2^{L_s} + 2^{L_u}$.

2.4.2 LCT in cryptanalysis of irregularly clocked LFSRs

Molland [3] and Molland and Helleseth [27] describe an attack on clock-controlled generators applying the linear consistency test. Here, z is the output from the complete system, u is the stream produced by the data generator LFSR$_u$ and s the stream produced by the clock control generator LFSR$_s$. The bit stream s is sent through a function D() that outputs the sequence c, which is used to clock LFSR$_u$. When c is applied to the output u, u is irregularly decimated, yielding the keystream z. This is illustrated in figure 3.

The goal of the attack in [3, 27] is to find the correct initial state for the clocking register LFSR$_s$. When this initial state is determined, other attacks can be used to determine the initial state of the data generator.

First, the initial state s^I of LFSR$_s$ is guessed. From this initial state, the clock control sequence $\hat{c}$ is generated. Using $\hat{c}$ and z, $\hat{u}* = (.., *, z_i, ..., z_j, ...*, ..., z_k, ..., *, ...)$ can be found, where z_i, z_j and z_k are keystream bits and the stars are deleted bits. $\hat{u}*$ is a guess for the position that the bits in z had in u. Using an equation h(x) determined from the feedback polynomial of LFSR$_u$, m entries are found in the stream $\hat{u}*$ where this equation is defined. For example, if the equation is $u_k + u_{k+6} + u_{k+8} = 0$, m=4 and the first guess for u is $(*, z_0, z_1, z_2, *, z_3, *, z_4, *, z_5, z_6, *, z_7, z_8, z_9, *, z_{10}, *, z_{11}, *, z_{12}, z_{13}, *, z_{14}, z_{15}, z_{16}$

$, *, z_{17}, *, z_{18})$ (the example from [3]), a set of equations over z is found by choosing 4 values of k that yield only non-deleted bits from the first guess for u:

k= 1: $z_0 + z_4 + z_5 = 0$

k=10: $z_6 + z_{10} + z_{11} = 0$

k=12: $z_7 + z_{11} + z_{12} = 0$

k=21: $z_{13} + z_{17} + z_{18} = 0$

If all of these equations hold, the guess for the initial state is output as the correct initial state of LFSR_s. If not, a new guess for s^I is made and the steps are repeated.

Thus, [3] derives a low weight cyclic equation h(x) from the feedback polynomial of LFSR_u that will hold for all bit streams generated by LFSR_u. Then the resulting set of equations is evaluated to decide whether to reject or accept the guess for s^I. [4], on the other hand, uses the Gaussian algorithm to reject or accept the initial state.

A modified version of Wagner's General birthday algorithm [28] is used to determine h(x). Also, [3] devises an efficient algorithm that can calculate each guess of u* using only a few operations, by reusing most of u* from one guess to another. This results in a significant improvement of the computational complexity of the complete attack.

3 Attack scenario - the proof of concept

In this thesis we are going to conduct a proof of concept of an attack scenario where the LCT is used in the presence of noise. Provided that such an attack is indeed possible, the next step is to find out whether it is efficient compared with other attacks.

In this chapter, some aspects of the Linear Consistency Test relevant for the forthcoming experimental work are discussed. Then, an extension of the LCT is presented that we believe will be successful in the presence of noise. The experimental work in the next chapter is built on this method. But first, the generator that will be used in the examples throughout the thesis is introduced.

3.1 The generator

The generator we are going to use in the experiments, is based on the BRM scheme. Because we are going to perform a proof of concept, for simplicity small shift registers are used.

In figure 4, $LFSR_s$ is the clocking register, which produces a sequence s of period P_s. $LFSR_u$ is the clocked register, which produces a sequence u of period P_u. The feedback polynomials are $f_s = 1 + x + x^4$ of $LFSR_s$ and $f_u = 1 + x^3 + x^4$ of $LFSR_u$. The polynomials are chosen to be irreducible and primitive.

With initial state $I_s = 0011$ of $LFSR_s$ and $I_u = 0110$ of $LFSR_u$, the sequences generated are $s = 100011110101100$ and $u = 101111000100110$ (see figure 5). Decimating the sequence u with s, yields $z = 011100011..$ of period $P_z = 15 \times 15 = 225$ as output of the generator.

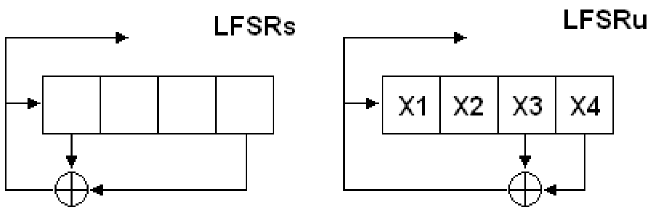

Figure 4: $LFSR_s$ with $f_s(x) = 1 + x + x^4$ and $LFSR_u$ with $f_u(x) = 1 + x^3 + x^4$

$LFSR_s$				$LFSR_u$			
0	0	1	1	0	1	1	0
1	0	0	1	1	0	1	1
0	1	0	0	0	1	0	1
0	0	1	0	1	0	1	0
0	0	0	1	1	1	0	1
1	0	0	0	1	1	1	0
1	1	0	0	1	1	1	1
1	1	1	0	0	1	1	1
1	1	1	1	0	0	1	1
0	1	1	1	0	0	0	1
1	0	1	1	1	0	0	0
0	1	0	1	0	1	0	0
1	0	1	0	0	0	1	0
1	1	0	1	1	0	0	1
0	1	1	0	1	1	0	0
0	0	1	1	0	1	1	0

Figure 5: Generation of the s and u sequences.

Suppose that the cryptanalyst knows the sequence z as well as the feedback polynomials $f_s(x)$ and $f_u(x)$, and guesses $I_{s_{guessed}} = 0011$ for the initial state of $LFSR_s$. Then he can reconstruct parts of the sequence u. By means of the sequence $s_{guessed} = 100011110101100$ he reconstructs $u_{guessed}$ as $x_5 0111 x_6 0 x_7 0 x_8 0 x_9 11 x_{10} 1$ and creates the following system of equations using $f_u(x)$:

$$\begin{aligned} x_3 + x_4 + x_5 &= 0 \\ x_2 + x_3 &= 0 \\ x_1 + x_2 &= 1 \\ x_1 + x_5 &= 1 \\ x_5 &= 1 \end{aligned} \tag{3.1}$$

Since this system has full rank, it can be solved and yield the initial state $x_1x_2x_3x_4$ of $LFSR_u$.

3.2 The Linear Consistency Test without noise

The LCT says that given enough output bits, the probability is high that the resulting system of equations is either contradictory (i.e. inconsistent) or can be solved to yield the correct initial state of $LFSR_u$.

If the number of equations in the system is too low, many false consistency alarms will occur, i.e. the system will be consistent for several initial states of $LFSR_s$ none of which is the correct one. This problem is solved by increasing the number of equations, and thus utilising a greater portion of the intercepted sequence. [4] claims that the number of equations should exceed significantly the number of unknowns plus the length of LFSR_s. Since the number of equations in the system is greater than the number of unknowns, we are dealing with an *overdefined* (or *overdetermined*) system.

[4] further shows (in the proof of Theorem 1) that the system is consistent if and only if the right hand side of the system of equations is contained in the subspace spanned by the column vectors of the **A** matrix (denoted $L(A)$). Hence, rather than solving the system of equations using Gauss elimination, the correct I_s can be determined by setting up a sufficiently large number of equations, generating the set $L(A)$ corresponding to the **A** matrix, and searching for the right hand side of the system (**b**) in the set $L(A)$. If, for a given guessed I_s the **b** vector is found in $L(A)$, and the number of equations is large enough, this guessed I_s is indeed the correct I_s.

The example below illustrates the consistency check using the registers described above (both of length 4), with the number of equations set to 11.

Here, the intercepted sequence is generated by $I_s = 0011$ and $I_u = 0110$, and a guess of 0011 for I_s yields the following system of equations:

$$\mathbf{A}x = \mathbf{b} \tag{3.2}$$

where

$$\mathbf{A} = \begin{pmatrix} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

$L(A)$ is the set of all linear combinations of the column vectors from **A**, that is $L(A) = (v_1, v_2, ..., v_n, v_1 + v_2, ..., v_1 + v_n, v_1 + v_2 + v_3, ..., ..., v_1 + v_2 + ... + v_{n-1} + v_n)$. Since **b** here equals $v_2 + v_3 + v_5 + v_6 + v_8$, **b** is indeed in the set $L(A)$, and the consistency check will output the system as consistent for $I_s = 0011$. When the number of equations (neq) is 11, $I_s = 0011$ is the only initial state for which the system is consistent, but for $neq < 11$ more than one I_s yield consistent systems (i.e. false consistency alarms). For example, when the number of equations is 10, 2 guesses for I_s (0011 and 0110) yield consistent systems. This illustrates that the number of equations must exceed some limit.

Several tests were run in order to investigate how many equations are needed to avoid or reduce the number of false consistency alarms for registers of different lengths.

3.2.1 Test 1

For the selected combinations of registers in Table 1, the intercepted sequences are generated with the given I_s and I_u. For each choice of the neq variable, all possible guesses for I_s are

polynomials and initial states	neq	% yielding consistent system
$f_s = 1 + x + x^4$, $f_u = 1 + x^3 + x^4$, $I_s = 0011$, $I_u = 0110$	8	10 / 15
	9	5 / 15
	10	2 / 15
	11	1 / 15
$f_s = 1 + x + x^4$, $f_u = 1 + x^3 + x^4$, $I_s = 0001$, $I_u = 0001$	8	8 / 15
	9	7 / 15
	10	5 / 15
	11	2 / 15
	12	1 / 15
$f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$, $I_s = 0000001$, $I_u = 0000001$	11	62 / 127
	12	55 / 127
	13	47 / 127
	14	29 / 127
	15	20 / 127
	16	9 / 127
	17	4 / 127
	18	4 / 127
	19	2 / 127
	20	1 / 127
$f_s = 1 + x^4 + x^7$, $f_u = 1 + x + x^4 + x^9 + x^{10}$, $I_s = 0000001$, $I_u = 0000000001$	16	64 / 127
	17	48 / 127
	18	32 / 127
	19	20 / 127
	20	9 / 127
	21	6 / 127
	22	4 / 127
	23	3 / 127
	24	1 / 127

Table 1: Percentage of guesses for I_s yielding a consistent system for different number of equations.

tried and the resulting systems are checked for consistency. The last column shows the number of guessed I_s yielding a consistent system.

3.2.2 Test 2

In this test, for a given guess for I_s a new equation is added to the system if the condition $neq < (nvar + len_s + delta)$ is true, whereas in the previous test all the guesses were tried with a fixed value of neq.

With this test, we want to investigate the condition of Theorem 1 mentioned in [4], namely that the number of equations should exceed the number of variables plus the length of $LFSR_s$ significantly in order to make the number of false consistency alarms as small as possible.

The results for registers of different lengths are summarized in tables 2 and 3.

As Table 2 shows, when $I_s = 1110$ and $delta = 0$, 2 different guessed I_s's yield consistent systems. When we increase $delta$ to 1, however, only the guess 1110 yields a consistent system.

In Table 2, all tests are performed with $I_u = 0001$, and in Table 3 with $I_u = 0101010$.

Subsequently, Test 2 was performed for all $15 \times 15 = 225$ combinations of I_s and I_u (not

I_s	delta	guessed I_s	neq	nvar	consistent system?
0001	0	0001	15	11	yes
		0010	14	10	no
		0011	14	10	no
		…			
		1111	12	8	no
				Total:	1 / 15
0010	0	0001	14	10	no
		0010	14	10	yes
		0011	14	10	no
		…			
		1111	12	8	no
				Total:	1 / 15
1110	0	0001	14	10	no
		0010	14	10	yes
		…			
		1110	17	13	yes
		1111	12	8	no
				Total:	2 / 15
1110	1	0001	16	11	no
		0010	16	11	no
		…			
		1110	19	14	yes
		1111	15	10	no
				Total:	1 / 15

Table 2: The sufficient magnitude of `delta` such that the consistency test returns consistent=true only for the correct guess for I_s. $f_s = 1 + x + x^4$, $f_u = 1 + x^3 + x^4$. The table shows only results for $I_u = 0001$ for selected values of I_s.

I_s	delta	guessed I_s	neq	nvar	consistent system?
0000001	0	0000001	27	20	yes
				Total:	1 / 127
0000011	0	0000011	27	20	yes
		1000001	28	21	yes
				Total:	2 / 127
0000011	1	0000011	30	22	yes
				Total:	1 / 127
0000111	0	0000111	26	19	yes
		0001011	24	17	yes
		1111100	26	19	yes
		1111111	25	18	yes
				Total:	4 / 127
0000111	1	0000111	32	24	yes
		0001011	26	18	yes
				Total:	2 / 127
0000111	2	0000111	35	26	yes
				Total:	1 / 127
0010110	0	0010110	24	17	yes
		1111000	27	20	yes
		1111110	26	19	yes
				Total:	3 / 127
0010110	1	0010110	26	18	yes
		1111000	29	21	yes
		1111110	28	20	yes
				Total:	3 / 127
0010110	2	0010110	28	19	yes
				Total:	1 / 127

Table 3: The sufficient magnitude of `delta` such that the consistency test returns consistent=true only for the correct guess for I_s. $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. The table shows only results for $I_u = 0101010$ for selected values of I_s.

including the all zero states). This was repeated 5 times with $\mathtt{delta}$ between 0 and 4. For all possible combinations of I_s, I_u and $\mathtt{delta}$ (where $\mathtt{delta} \in [0, 4]$), the correct guess for I_s is indeed in the set of the guesses that yield a consistent system, as expected. The results are summarized in Table 4. The table shows only the combinations of I_s and I_u for which the number of consistent systems is greater than 1 for $\mathtt{delta} = 0$, as well as displaying the results for these same combinations with increasing values of $\mathtt{delta}$.

This result supports the condition given in [4].

I_s	I_u	delta	# consistent
0001	0100	0	2/15
		1	2/15
		2	1/15
0001	1000	0	2/15
		1	1/15
0001	1010	0	2/15
		1	1/15
0001	1100	0	2/15
		1	1/15
0010	0010	0	2/15
		1	1/15
0010	0100	0	2/15
		1	1/15
0010	0101	0	2/15
		1	1/15
0010	0110	0	2/15
		1	1/15
0010	1000	0	2/15
		1	2/15
		2	1/15
0011	1101	0	2/15
		1	1/15
0011	1110	0	2/15
		1	1/15
0100	0100	0	2/15
		1	2/15
		2	1/15
0101	0110	0	2/15
		1	2/15
		2	1/15
0110	1010	0	2/15
		1	2/15
		2	1/15
0110	1101	0	2/15
		1	1/15
0111	0011	0	2/15
		1	1/15

1000	1001	0	2/15
		1	1/15
1000	1110	0	2/15
		1	1/15
1001	0011	0	2/15
		1	1/15
1010	0101	0	3/15
		1	2/15
		2	1/15
1010	1011	0	2/15
		1	1/15
1011	1001	0	2/15
		1	1/15
1100	0111	0	2/15
		1	1/15
1101	1010	0	2/15
		1	1/15
1101	1101	0	2/15
		1	1/15
1110	0001	0	2/15
		1	1/15
1111	0110	0	2/15
		1	1/15

Table 4: The sufficient magnitude of $\mathtt{delta}$ such that the consistency test returns consistent=true only for the correct guess for I_s, for all combinations of initial states of $LFSR_s$ and $LFSR_u$. The table only shows combinations of I_s and I_u for which the number of consistent systems are greater than 1 for delta=0. $f_s = 1 + x + x^4$, $f_u = 1 + x^3 + x^4$.

3.2.3 The number of equations and delta

Test 1 and Test 2 described above verify that in order for the consistency test not to give false alarms (i.e. a wrongly guessed initial state is reported as the correct one) sufficiently many equations must be created.

It can be seen from tables 2-4 that the required magnitude of delta depends on the initial states used to genereate the kestream sequence, and that the required number of equations depends both on the register lengths and the initial states.

3.3 The Linear Consistency Test with noise

Continuing the example from the previous section, we are going to add noise to the intercepted sequence and investigate whether the Linear Consistency Test can be successful also in this situation.

Consider the first 7 equations of the system in Equation 3.2:

$$\begin{aligned} x_3 + x_4 + x_5 &= 0 \\ x_2 + x_3 &= 0 \\ x_1 + x_2 &= 1 \\ x_1 + x_5 &= 1 \\ x_5 &= 1 \\ x_6 &= 1 \\ 0 &= 0 \end{aligned} \tag{3.3}$$

Obviously, this system of equations is consistent. However, if we change the value of the 2nd bit of the intercepted sequence z from 1 to 0, yielding $noisyZ = 0011000111$ and $u_{guessed} = x_5 0011 x_6 0 x_7 0 x_8 0 x_9 11 x_{10} 1$, the first 7 equations are:

$$\begin{aligned} x_3 + x_4 + x_5 &= 0 \\ x_2 + x_3 &= 0 \\ x_1 + x_2 &= 0 \\ x_1 + x_5 &= 1 \\ x_5 &= 1 \\ x_6 &= 0 \\ 0 + 1 &= 0 \end{aligned} \tag{3.4}$$

The last equation clearly shows that this system is not consistent, and thus the LCT will reject the guess of 0011 for I_s. But, provided that the intercepted sequence is long enough, we can start building equations from a position farther to the right in the sequence, such that the altered 2nd position will not appear in the system of equations. For example, starting from the 8th position of $u_{guessed}$, we get the following system:

$$\begin{aligned} x_7 &= 0 \\ x_6 &= 1 \\ x_6 + x_8 &= 0 \\ (x_7 &= 0) \\ x_7 + x_9 &= 0 \\ x_8 &= 1 \\ (x_8 &= 1) \\ x_9 + x_{10} &= 0 \\ x_9 &= 0 \end{aligned} \tag{3.5}$$

Positions: 44,22,148,9,196,155,142,132,189,75,78,57	
nIterations	nConsistent
11	0
21	0
31	0
38	0
39	1
40	2
41	3

Table 5: The number of iterations for which the LCT test yields consistent=true for selected numbers of iterations. $f_s = 1 + x + x^4$, $f_u = 1 + x^3 + x^4$. NoiseLevel=0.05. Guessed I_s equals actual I_s (0011).

which is, indeed, consistent. [1]

Bringing this idea further, suppose that n arbitrary bits of z are altered, yielding a system with a noise level of n/l, where l is the length of z. For example, if $l = 40$ and we set the noise level to 0.05, $noisyZ$ differs from z in two bits. The bits to be altered are chosen randomly.

Suppose also that we run the LCT an odd number of times with the same guess for I_s (denoted $nIterations$). For each iteration, the starting position (k) of the equations is increased by one, starting with $k = 0$ in the first iteration. With $k = 0$ the equations are created starting from the first bit of $u_{guessed}$, and the example above where we started on the 8th bit would correspond to $k = 7$. For each iteration, the result will be either "consistent=false" or "consistent=true", and the number of consistent iterations is denoted by $nConsistent$.

Our conjecture is that when $nIterations$ is sufficiently high, a majority function can determine whether the guess for I_s is indeed correct. For example, when $nIterations = 5$, if three or more of the iterations yield consistent systems ($nConsistent >= 3$), we conclude that the guess for I_s is correct, and if less than three of the iterations yield consistent systems ($nConsistent < 3$), we conclude that the guess for I_s is wrong.

Preliminary experimentation shows that (with $nIterations = 11$, $delta = 2$ and $noiseLevel = 0.05$) when the guessed I_s is different from the actual I_s, the LCT seems to always yield $nConsistent = 0$, irrespective of the combination of positions selected to be noisy. This result looks promising.

However, when the guessed I_s is the correct one, $nConsistent$ varies between 0 and $nIterations$, inclusive (examples are given in Table 5 and Table 6). Thus, for some combinations of noise positions, $nConsistent = 0$ even if the guessed I_s is the correct one, even for larger values of $nIterations$. Then, how can we distinguish between a wrong guess for I_s and a correct guess?

On the other hand, when we increase $nIterations$ sufficiently (e.g. $nIterations = 79$, $nIterations = 99$, $nIterations = 199$ etc for registers of length 4), it seems that the number of combinations yielding $nConsistent = 0$ decreases when the guessed I_s is the correct one.

[1] Since two of the equations (in parentheses) are duplicates of earlier equations, these are discarded and 2 additional equations are produced.

Positions: 88,43,74,148,195,176,19,41,95,61,39,124	
nIterations	nConsistent
11	9
21	9
31	9
38	11
39	11
40	11
41	11

Table 6: The number of iterations for which the LCT test yields consistent=true for selected numbers of iterations. $f_s = 1 + x + x^4$, $f_u = 1 + x^3 + x^4$. NoiseLevel=0.05. Guessed I_s equals actual I_s (0011).

These observations lead to the following hypotheses:

H1 When $nIterations$ exceeds a certain level, the probability for a correct guess for I_s to yield $nConsistent = 0$ is very low.

H2 Even when $nIterations$ is high, the probability that a wrong guess for I_s yields $nConsistent = 0$ is close to 1.

To be able to investigate these hypotheses further, we select a random sample of noise position combinations and run consistency tests for each of these combinations for a given noise level for two cases (corresponding to H1 and H2):

1. When the guessed I_s equals the actual I_s
2. When the guessed I_s is different from the actual I_s

Then, given that the selected sample of combinations is representative of all possible noise position combinations, we can, for each noise level, find the lower bound for the value of $nIterations$ for which the number of combinations yielding $nConsistent = 0$ is 0 (or a value close to 0) when the guessed I_s equals the actual I_s. Denote this value by $lowestNIterations$. If, for this $lowestNIterations$ value, the probability that $nConsistent = 0$ is close to 1 when the guessed I_s is different from the actual I_s, we may conclude that the LCT is indeed successful for the given noise level. The experimental setup is explained more thoroughly in the next chapter.

3.3.1 Computational complexity

The discussion in the previous section addressed the first research question. In this section, the second research question is investigated.

In [3] it was proved that the LCT attack (without noise) has a maximum complexity of $O(2^{l_1})$, where l_1 is the length of $LFSR_s$, for the BRM model. Thus, the runtime is independent of the length of $LFSR_u$. Assuming the most efficient LCT implementation, the complexity of the LCT with noise is given as $c \times 2^{l_1}$ for a given noise level and register length, where c is a constant. This is obvious, since in the scenario with noise the LCT attack is run c times.

The magnitude of c can be found for small register lengths (see the results in Section 4.4) since it corresponds to the $lowestNIterations$ variable. By investigating how this variable changes as the register lengths are increased, it might be possible to estimate c for larger registers and make a generalization on the runtime complexity of the attack.

However, in order for this argumentation to hold, it must be possible to modify the algorithm of [3] (see 2.4.2) such that for each iteration of the algorithm, the starting point of the equations is shifted one position to the right. Just like the attack scenario described in this thesis, the equations of [3] are built from $u_{guessed}$. But rather than using the feedback polynomial of $LFSR_u$, a low weight equation $h(x)$ determined from this polynomial is used to create the set of equations.

Consider what will happen when the input to the algorithm of [3] is a noisy keystream sequence, and our guess for I_s is correct. If some of the equations from the generated set of equations contain some noisy bits, then, by the argumentation in 3.3, the set of equations might not be consistent. Further, we anticipate that by shifting the starting position of the equations, the next set of equations might contain no noisy bits, and the set of equations will be consistent. This corresponds to the situation in 3.3, so intuitively, it would be possible to replace the LCT implementation of the attack in this thesis with the faster LCT attack of [3].

Unlike the LCT attack, the runtime of correlation attacks is dependent on the length of both registers. The ciphertext only clock reconstruction correlation attack of [8] (see 2.4.1) is performed in two phases, and has a complexity of $O(2^{L_s} + 2^{L_u})$.

Thus, if the c constant discussed above grows linearly with the register lengths (or sub-linearly), we can conclude that the LCT attack with noise performs better than the ciphertext only correlation attacks when the length of $LFSR_s$ is sufficiently shorter than the length of $LFSR_u$[2]. The hypothesis to be investigated in order to answer the second research question (see Section 1.5) is stated as follows:

H3 The growth of the c constant is linear (or sub-linear) with increasing register lengths

The growth of the c constant can be estimated by running the experiment outlined in the previous section (case 1) with generators of different register lengths, and finding the best fitting function to describe the relationship between the x's (register length) and the y's (the number of iterations of the LCT test).

[2]This is often the case in practical implementations, cf. the LILI-128 keystream generator [1]

4 Experimental work

The aim of the experimental work is to give an answer to the hypotheses (H1, H2 and H3) stated in the previous section, which in turn will answer the research questions (see Section 1.5).

The generator used in the experiments was described in Figure 4. In addition to this length 4 generator with feedback polynomials $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$, generators with registers of length 7 and 12 are also used. Table 7 gives an overview of these generators.

In order to describe the experimental setup, the pseudocode together with an example is provided. Subsequently, the choice of parameter values for the experiment is discussed and the results are presented. An analysis of the results follows in the next chapter.

4.1 Case 1 - Pseudocode and example

For the experiments described in this section a random sample consisting of a number of noise position combinations is selected, and for each of the combinations consistency tests are run with guessed I_s equal to the actual I_s.

For example, for noise level 0.05, let the length of the intercepted sequence be 225 bits, then $\lceil 225 \times 0.05 \rceil = 12$ of the bits from the output of the generator (z) are altered to yield the noisy sequence $noisyZ$. The indices of the positions to be altered are chosen pseudorandomly, and one such combination of noise positions (e.g. positions 1, 23, 56, 89, 2, 19, 17, 221, 90, 40, 51 and 111) is called a *noise position combination*, or simply a *combination*. Then, for each combination, the bits of the z sequence are altered (a 0 is changed to 1, and a 1 is changed to 0) according to the given noise positions to yield the sequence $noisyZ$. The number of combinations is determined by the $nCombinations$ variable.

The $maxNIterations$ variable determines the total number of times the consistency check is going to be run for each combination. The $nIterations$ variable is incremented from 0 to $maxNIterations$, and for each increment the starting position of the equations (as explained in Section 3.3) is shifted one position to the right in the decimated sequence. For each value of $nIterations$, we count the number of consistency checks yielding consistent=true. For example, when $nIterations = 11$ and the number of consistency checks yielding consistent=true equals 7, then $nConsistent = 7$ (out of 11).

In the case 1 experiments, we are interested in the combinations that yields $nConsistent = 0$, and we want to know how large $nIterations$ must be in order for the number (or percentage)

$f_s(x)$	$f_u(x)$	register lengths
$1 + x + x^4$	$1 + x^3 + x^4$	4
$1 + x^4 + x^7$	$1 + x^6 + x^7$	7
$1 + x + x^4 + x^6 + x^{12}$	$1 + x^2 + x^3 + x^9 + x^{12}$	12

Table 7: Feedback polynomials of the generators used in the experimental work.

of combinations yielding $\mathit{nConsistent} = 0$ to be close to 0, when the initial state of LFSR_s is guessed correctly. The graphs presenting the results later in this chapter show this percentage as $\mathit{nIterations}$ increases, and we are especially interested in the lowest value of $\mathit{nIterations}$ for which this percentage is below a certain threshold (denoted by $\mathit{lowestNIterations}$).

4.1.1 An example experiment

When the experiment (see Algorithm 1) is run with the parameters in Table 8 and with the combinations (chosen by the algorithm) from Table 9, the output reported in Table 10 is obtained.

The table is split in 11 smaller tables, each showing the results for each combination in the given iteration.

For example, when iteration 1 is finished ($\mathit{nIterations} = 1$), 1 consistency check has been run for each of the combinations. For comb 0 this check yielded consistent=true, for comb 1 the check yielded consistent=false etc. Here, $\mathit{nConsistent} = 0$ for two of the combinations (comb 0 and comb 1), thus the result for $\mathit{nIterations} = 1$ is $\mathit{nNConsistent0} = 2$.

Further, when iteration 2 is finished ($\mathit{nIterations} = 2$), a total of 2 consistency checks has been run for each of the combinations, and since the consistency check for comb 0 in iteration 2 yielded consistent=true, $\mathit{nConsistent}$ is increased to 2 for comb 0.

Observe that in iteration 8 the consistency check for comb 2 yields consistent=true, such that for $\mathit{nIterations} = 8$, the value of the $\mathit{nNConsistent0}$ variable is decreased to 1. Thus, 20% of the combinations yield $\mathit{nConsistent} = 0$. In the graphs in Section 4.4, this percentage is shown in the y-axis, while the corresponding value of $\mathit{nIterations}$ is shown in the x-axis.

4.1.2 Pseudocode

The code that produced the results of Table 10 is shown as pseudocode in Algorithm 1. Since the inner loop (lines 13-23) is run $\mathit{maxIterations}$ times, some data needed here is generated during preprocessing (lines 4-8) and stored in the $\mathit{combinations}$ array:

- the decimation vector for the given combination (in the LctNoise object initialized in line 6)
- the $\mathit{nConsistent}$ variable (in the Combination object initialized in line 8)

The results for each iteration is stored in Result objects in the $\mathit{results}$ array. The Result object has an $\mathit{nNConsistent0}$ member variable to keep track of the number of combinations yielding $\mathit{nConsistent} = 0$ when the given iteration is finished (see Table 10). This variable is updated in line 22.

In the inner for loop (lines 13-23), a system of equations is created for a given noisyZ, with k as starting point. When the equations are created, the system is checked for consistency. If the system is consistent, the $\mathit{nConsistent}$ of the given combination is updated (line 18). To prepare for the next iteration, the amat and cvec variables (in which the system of equations is stored) in the LctNoise object are reset. Then, if $\mathit{nConsistent} = 0$ for this combination, the number of combinations yielding $\mathit{nConsistent} = 0$ is increased by one for the given iteration (line 22).

4.2 Case 2

Rather than running consistency test with guessed I_s equal to the actual I_s, as was done in the previous section, for the case 2 experiments the guessed I_s is *different* from the actual I_s. Thus,

noiseLevel	0.05
nCombinations	5
maxNIterations	11
lengthZ	225
delta	0
sPoly	$1 + x + x^4$
uPoly	$1 + x^3 + x^4$
initS	0011
initU	0110
guessedInitS	0011

Table 8: Parameters of the example experiment

combination	noise positions
0	19 39 41 43 61 74 88 95 124 148 176 195
1	6 57 72 74 109 120 123 144 147 154 196 205
2	2 19 25 42 98 119 145 151 163 173 175 221
3	16 50 67 85 86 91 96 141 155 179 201 215
4	27 28 42 62 82 108 115 151 152 168 197 207

Table 9: Noise positions of the example experiment

Algorithm 1 ExperimentLctNoise ( )

```
 1: s ← GenerateSequence(sPoly, initS)
 2: u ← GenerateSequence(uPoly, initU)
 3: for j ← 0, nCombinations do
 4:     z ← DecimateSequence(s, u)
 5:     noisyZ ← AddNoise(z, noiseLevel)
 6:     lct ← CreateLctNoise(sPoly, uPoly, guessedInitS, noisyZ, delta)
 7:     lct.CreateDecimationVector( )
 8:     combinations[j] ← CreateCombination(lct)
 9: end for
10: for k ← 0, maxIterations do
11:     results[k] ← CreateResult(k+1, nCombinations)
12:     for i ← 0, nCombinations do
13:         comb ← combinations[i]
14:         lct ← comb.lct
15:         lct.CreateEquations(k)
16:         consistent ← lct.IsConsistent( )
17:         if consistent then
18:             comb.nConsistent ← comb.nConsistent + 1
19:         end if
20:         lct.ClearAmatCvec( )
21:         if comb.nConsistent = 0 then
22:             results[k].nNConsistent0 ← results[k].nNConsistent0 + 1
23:         end if
24:     end for
25: end for
```

Iteration 1, result.nNConsistent0 = 2					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	1	0	0	1	1

Iteration 2, result.nNConsistent0 = 2					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	2	0	0	2	2

Iteration 3, result.nNConsistent0 = 2					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	3	0	0	3	3

Iteration 4, result.nNConsistent0 = 2					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	4	0	0	4	4

Iteration 5, result.nNConsistent0 = 2					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	5	0	0	5	5

Iteration 6, result.nNConsistent0 = 2					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	6	0	0	6	6

Iteration 7, result.nNConsistent0 = 2					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	7	0	0	7	7

Iteration 8, result.nNConsistent0 = 1					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	8	0	1	8	8

Iteration 9, result.nNConsistent0 = 1					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	9	0	2	9	9

Iteration 10, result.nNConsistent0 =					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	10	0	3	10	10

Iteration 11, result.nNConsistent0 = 1					
	comb 0	comb 1	comb 2	comb 3	comb 4
comb.nConsistent	11	0	4	11	11

Table 10: Result of ExperimentLctNoise with maxNIterations=11, noiseLevel=0.05, delta=0 and nComb=5. The table shows the accumulated number of iterations for which the LCT yields consistent=true (denoted by nConsistent) for each of the five combinations.

the pseudocode of Algorithm 1 is still employed, but it is executed 14 times rather than 1 time for the length 4 registers, each time with a new guess for I_s.

After the execution of the code of Algorithm 1 $2^n - 2$ times, where n is the length of $LFSR_s$, the $nNConsistent0$ variable is totalized over all initial states. As before, the case 2 graphs of section 4.4 show the percentage out of all combinations for which $nConsistent = 0$. Thus, the case 2 experiments aim to answer H2 in section 3.3.

4.3 Discussion of parameter choices

4.3.1 Choice of initial states, delta and the number of combinations

The $delta$ value, discussed previously, determines how many equations are created. Thus, it has a large impact on the runtime of the experiment (and the attack itself), and consequently, $delta$ should be kept small.

Table 4 suggests the value of $delta$ for different combinations of I_s and I_u. For the experiments with the $length = 4$ registers, a I_s/I_u combination that yields 1/15 consistent for $delta = 0$ is chosen (these are the I_s/I_u combinations that are *not* shown in the table). One such combination is $I_s = 0011$ / $I_u = 0110$.

Similarly, for the $length = 7$ registers, table 3 shows that $I_s = 0000001$ / $I_u = 0101010$ is an initial state combination that yields 1/127 consistent for $delta = 0$. So for the experiments with the $length = 7$ registers, this combination of initial states is chosen.

In order to get more accurate results, the number of noise position combinations in the sample should be high. For the $length = 4$ registers, $nCombinations = 1000$ is feasible. However, $nCombinations = 1000$ for the $length = 7$ experiments is not feasible due to limited computational resources. Thus, for the $length = 7$ experiments, $nCombinations$ is set to 100.

It was also considered to run the experiments for all combinations of I_s/I_u (or a sample consisting of some number of randomly selected combinations). However, experimentation revealed that varying the initial state combination had little impact on the results. Thus, for simplicity, only one combination of initial states was used for each of the generators.

4.3.2 Choosing a threshold

From a cryptanalyst's point of view, it is better to believe that our guess is correct when in reality it is not, than to believe that our guess is wrong when in reality it is not. The reason is that if we believe our guess is correct, we would normally be able employ the guessed key in deciphering and see from the resulting plaintext whether the key believed to be correct was indeed the correct one. On the other hand, if we believe that our guess is wrong, we may miss the correct key.

According to this reasoning, we should make sure that the probability that the LCT yields ”wrong guess” when the guess was indeed correct is very low. That is, in finding an adequate trade-off for the value of $nIterations$, we should make sure that the probability for $nConsistent = 0$ for case 1 is very low, while we can accept a greater deviation from 1 for the probability for $nConsistent = 0$ for case 2. So in order to decide the $lowestNIterations$ value for different noise levels, we set the threshold for the case 1 percentages to 1%.

For instance, it can be seen from the results in the next section that for noise level 0.20 and the length 7 registers, if we demand that $nConsistent = 0$ be less than or equal to 1%

for case 1, then $nIterations$ must be (at least) 4139. That means that we must accept that $nConsistent = 0$ is 91.41% for case 2.

4.4 Results

The figures (6-15) show the percentage of combinations for which $nConsistent = 0$ (y-axis) for different values of $nIterations$ (x-axis). In the experiments, noise levels in the range 0.05-0.25 are employed, and one sample consists of 1000 randomly selected combinations of noise positions for the length=4 registers, and 100 for the length=7 registers. For each noise level (0.05, 0.10, 0.15, 0.20 and 0.25) a new sample of 1000 (100) combinations is selected. Note that the scale of the x-axis is not the same for all graphs.

4.4.1 $nConsistent = 0$ when guessed I_s equals actual I_s (case 1)

In the experiments reported in this section, the consistency tests are run with guessed I_s equal to the actual I_s. This suggests that low percentages should be observed for the $nConsistent = 0$ variable, at least when the level of noise is low.

The charts are displayed in figures 6-10.

4.4.2 $nConsistent = 0$ when guessed I_s is different from actual I_s (case 2)

Here, the consistency tests are run with guessed I_s different from the actual I_s. This suggests that high percentages should be observed for the $nConsistent = 0$ variable.

In the experiments, the numbers are totalized over all possible values for the guess of I_s (except for the all zero state and the actual I_s). Hence, for registers of length 4, 14 different guesses are employed, and for registers of length 7, 126 different guesses are employed.

The charts are displayed in figures 11-15.

It should be observed that, in order to be able to obtain the results for case 2, which was quite time consuming for higher noise levels and register lengths, it should be observed that the length of the intercepted sequence for case 2 in many cases was chosen to be significantly shorter than for the corresponding case 1 experiment. However, this should have no impact on the results since the number of bits that were altered up to the magnitude of lowestNIterations would be approximately the same in both cases. For completeness, the length of the intercepted sequence ($lenZ$) and the number of bits altered for each noise level, register combination and delta used in the experiments are given in tables 16 and 17.

4.4.3 Trade-off between a high case 2 percentage and low case 1 percentage

Tables 11 and 12 show the $lowestNIterations$ value for different noise levels for the length 4 registers with delta=0. According to Section 4.3.2, the tables show the lowest value of $nIterations$ for which the case 1 percentage is equal to or less than 1%, and the corresponding case 1 and case 2 percentages. It should be noted that when this value is found, only odd values of $nIterations$ are considered.

4.4.4 Increasing $delta$

As Table 11 shows, the results for the $length = 4$ registers with $delta = 0$ are quite bad, in terms of the *case 2* percentages.

A low value of $delta$ means fewer equations (compared with a higher value of $delta$),

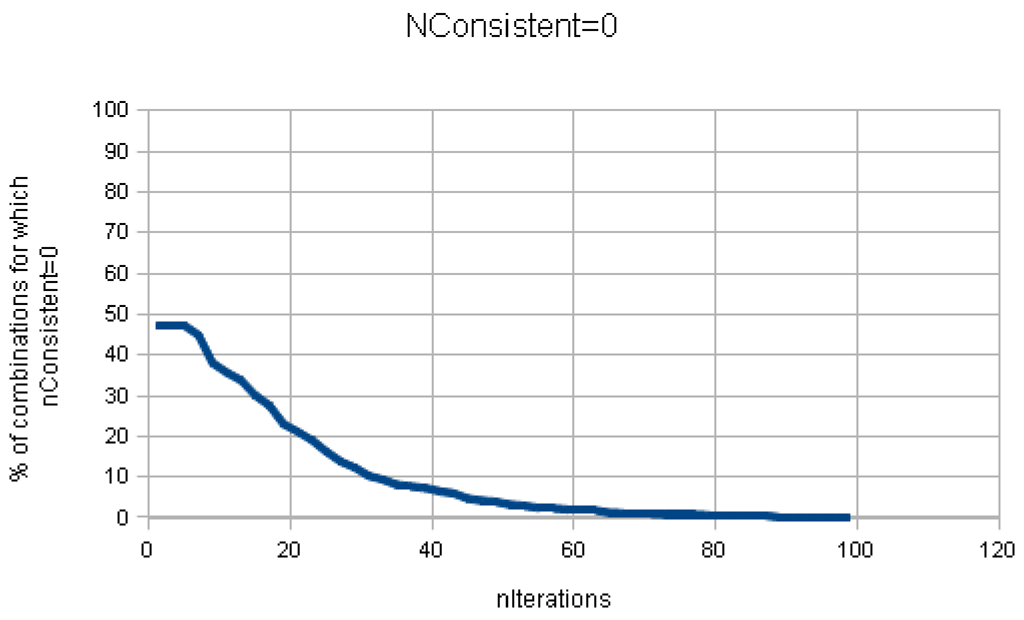

Figure 6: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.05, nComb=1000, delta=0.

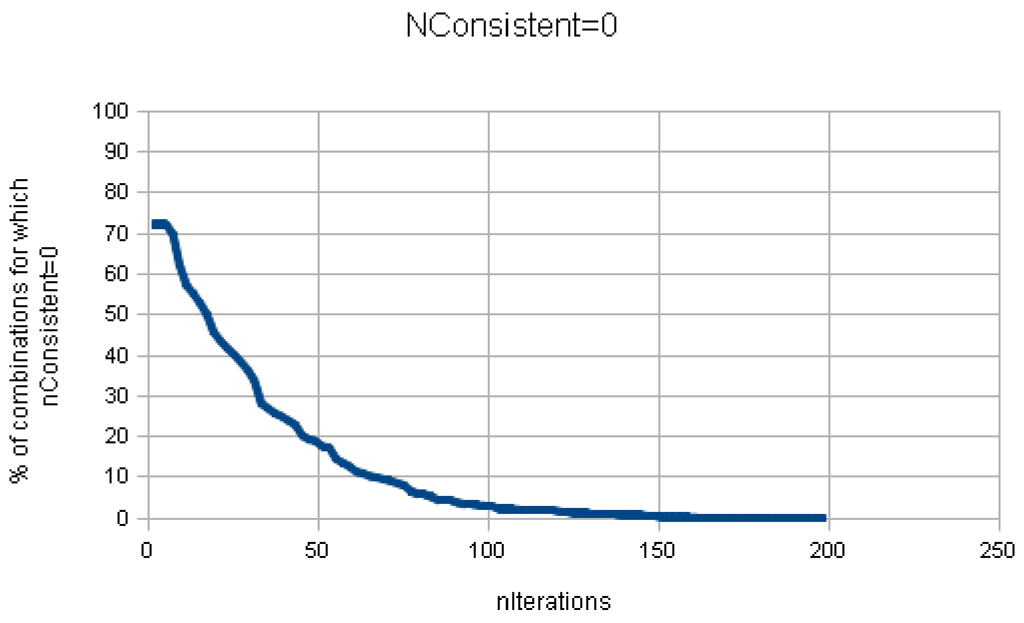

Figure 7: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.10, nComb=1000, delta=0.

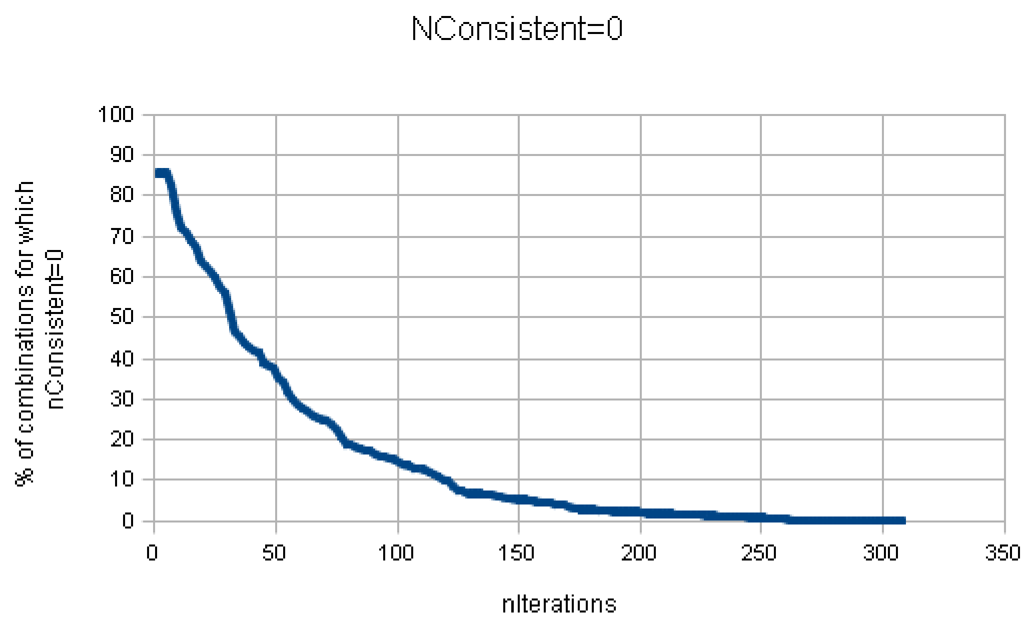

Figure 8: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.15, nComb=1000, delta=0.

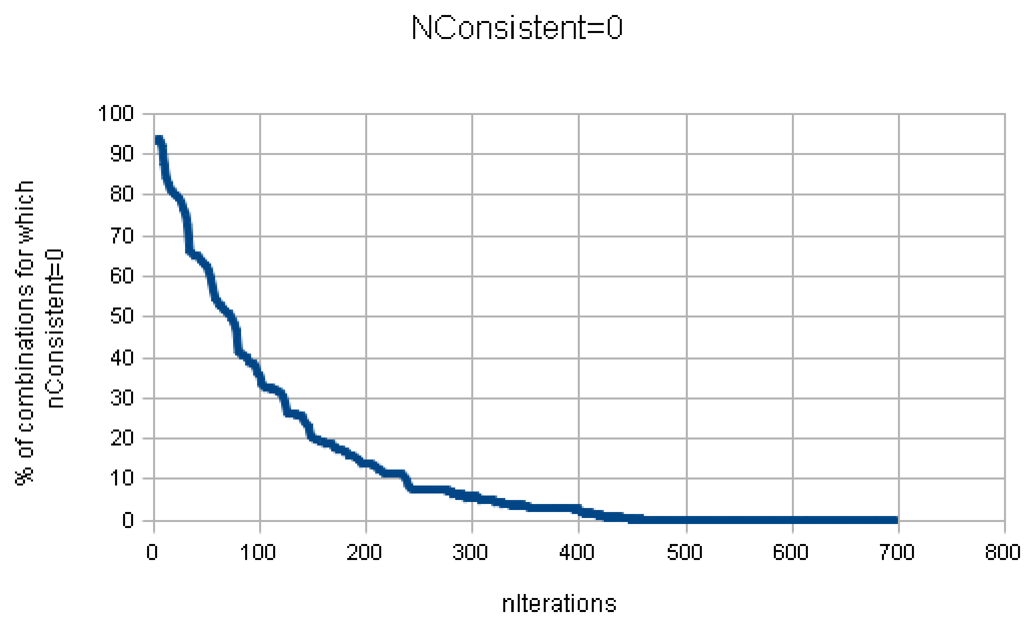

Figure 9: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.20, nComb=1000, delta=0.

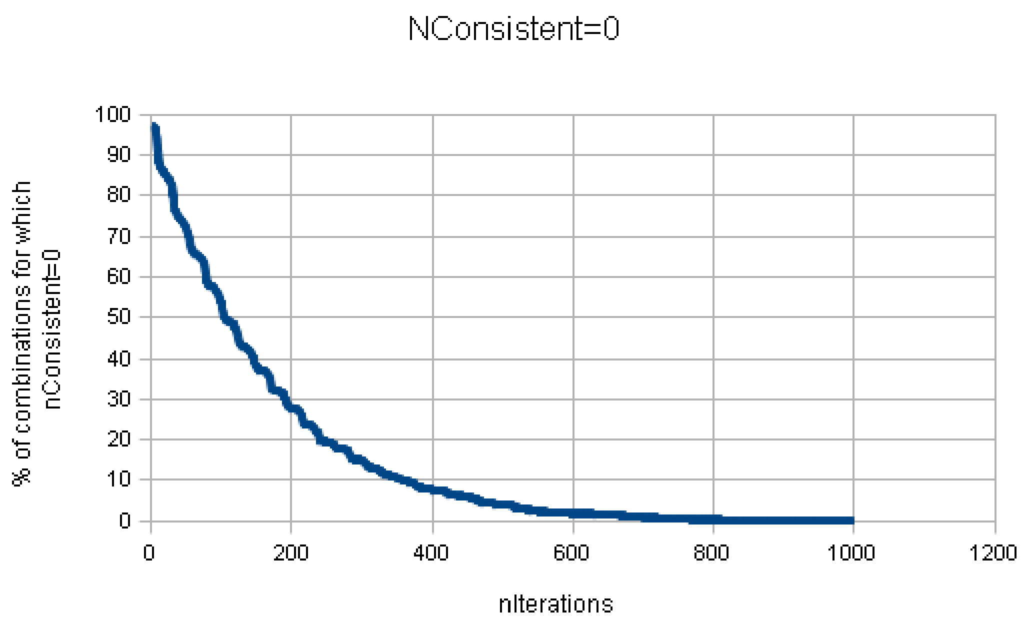

Figure 10: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.25, nComb=1000, delta=0.

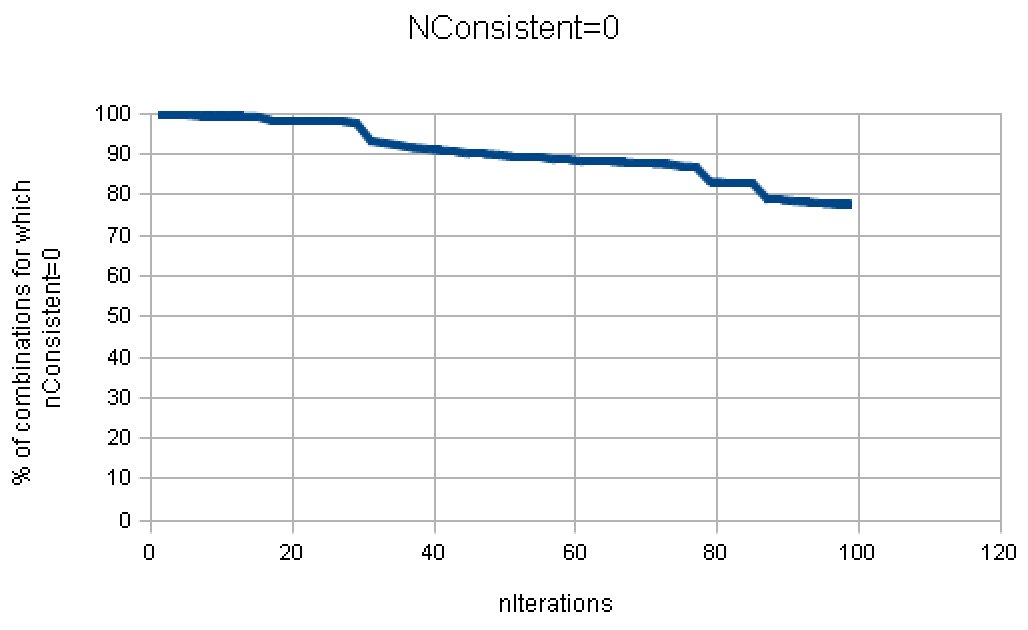

Figure 11: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.05, nComb=1000, delta=0.

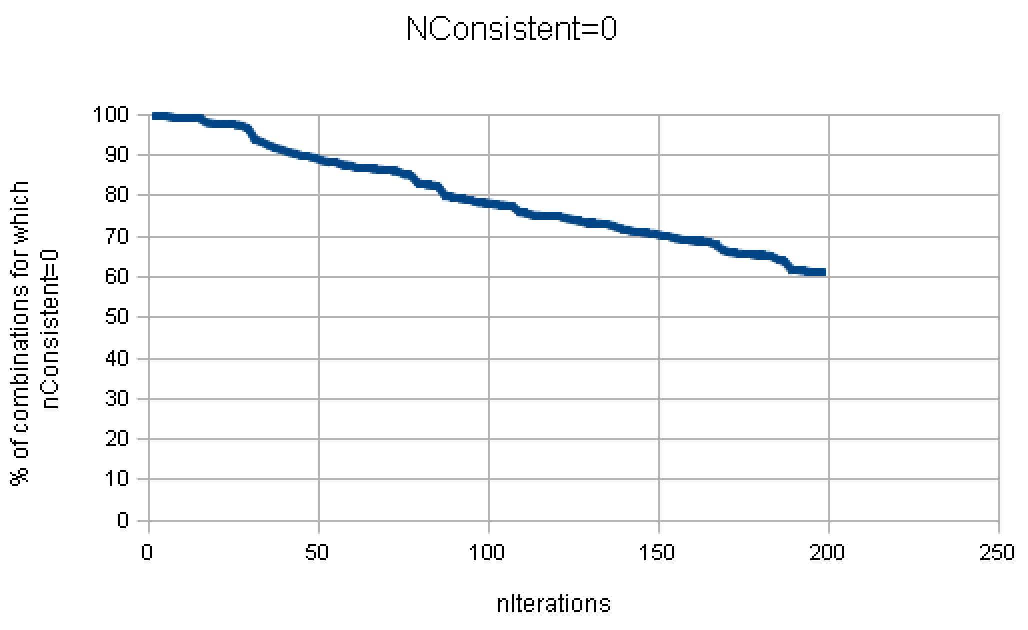

Figure 12: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.10, nComb=1000, delta=0.

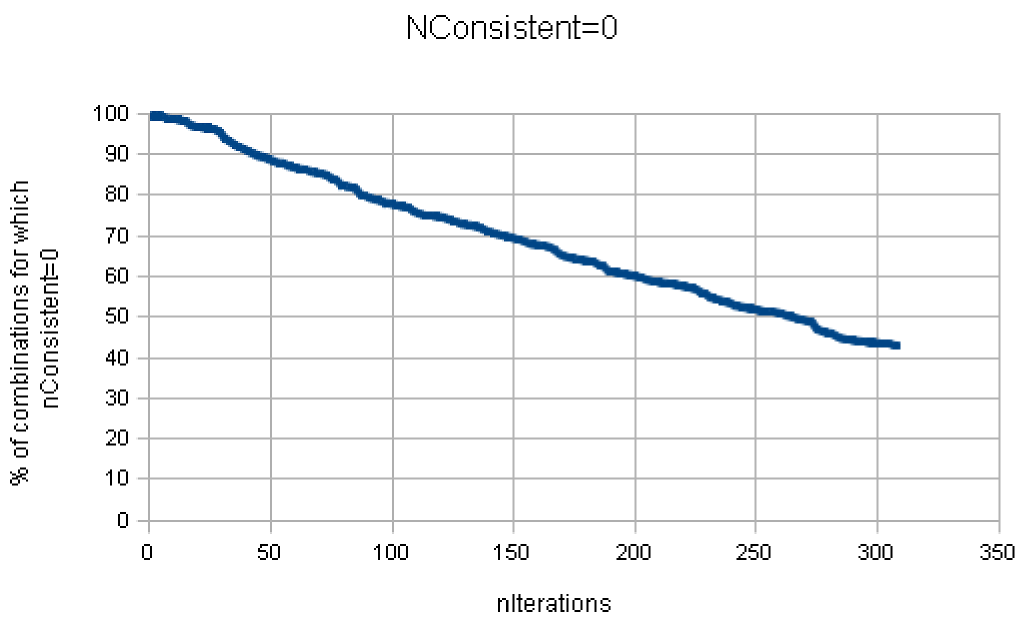

Figure 13: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.15, nComb=1000, delta=0.

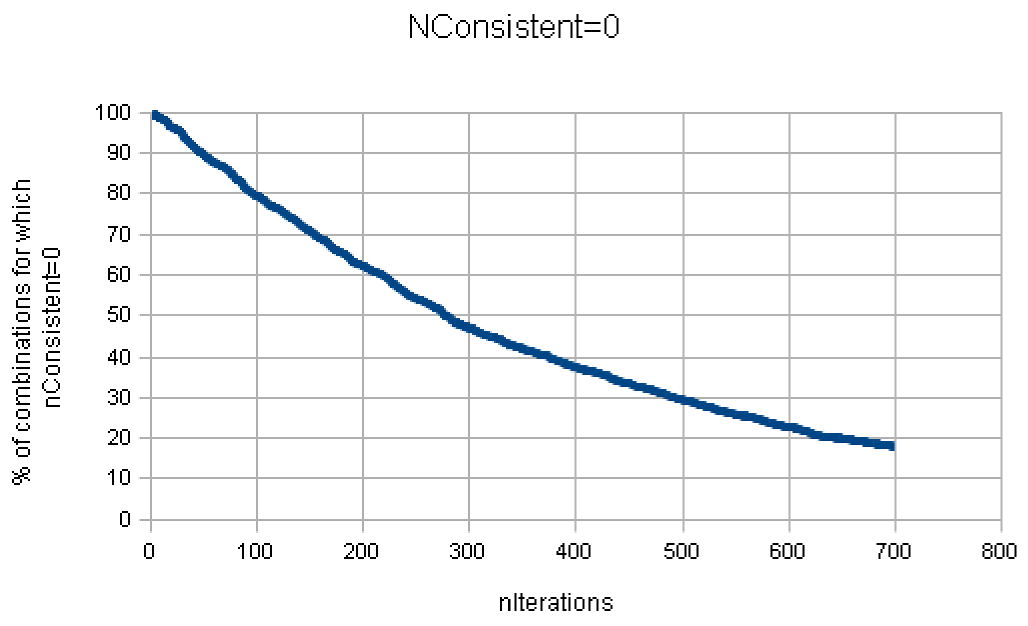

Figure 14: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.20, nComb=1000, delta=0.

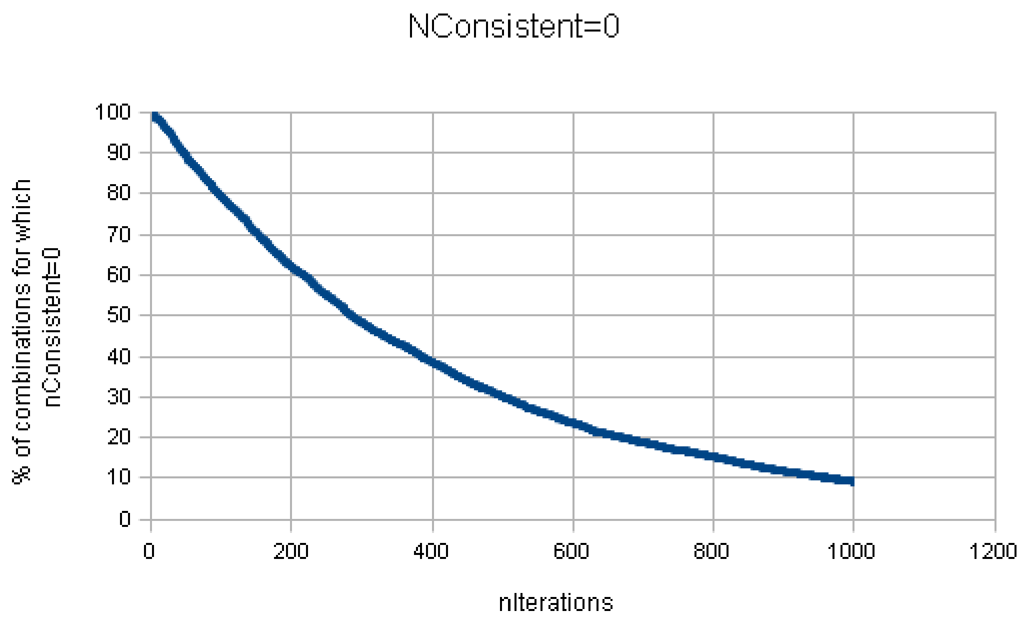

Figure 15: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.25, nComb=1000, delta=0.

$noiseLevel$	$lowestNIterations$	$nConsistent = 0$, case 1	$nConsistent = 0$, case 2
0.05	69	1.0%	87.74%
0.10	131	1.0%	73.18%
0.15	235	1.0%	54.11%
0.20	423	0.9%	35.94%
0.25	697	1.0%	18.9%

Table 11: Trade-off between $nConsistent = 0$ for case 1 and $nConsistent = 0$ for case 2. $f_s = 1 + x + x^4$, $f_u = 1 + x^3 + x^4$. Delta=0.

$noiseLevel$	$lowestNIterations$	$nConsistent = 0$, case 1	$nConsistent = 0$, case 2
0.05	135	1%	98.84%
0.10	483	1%	98.33%
0.15	1625	1%	95.56%
0.20	4139	0%	91.41%
0.25	-	-	-

Table 12: Trade-off between $nConsistent = 0$ for case 1 and $nConsistent = 0$ for case 2. $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Delta=0.

hence fewer restrictions on the system of equations, which would result in a greater number of consistent systems. This applies when the guess is correct (with noise) as well as when the guess is wrong. Thus, by increasing $delta$ slightly, higher percentages for *case 2* are expected.

The results obtained for $delta = 0$ and the $length = 4$ registers show that not only the choice of initial states from which the keystream sequence is generated has an impact on the magnitude of delta; the presence of noise enforces even more equations to be created and thus the use of a higher value of delta in order to obtain good results.

The subsequent section shows the results with $delta = 1$ and $delta = 2$.

4.4.5 Results for $delta = 1$ and $delta = 2$

Figures 16-25 and Table 13 show the results for the $length = 4$ registers width $delta = 2$. As the table shows, the results for case 2 are much better with $delta = 2$ than with $delta = 0$. Tables 14 and 15 show the results for $delta = 1$ with register lengths 4 and 7, respectively. The corresponding graphs can be found in the appendix.

$noiseLevel$	$lowestNIterations$	$nConsistent = 0$, case 1	$nConsistent = 0$, case 2
0.05	117	1.0%	99.76%
0.10	293	0.9%	98.46%
0.15	749	0.6%	95.89%
0.20	1607	1.0%	93.04%
0.25	4507	1.0%	82.48%

Table 13: Trade-off between $nConsistent = 0$ for case 1 and $nConsistent = 0$ for case 2. $f_s = 1 + x + x^4$, $f_u = 1 + x^3 + x^4$. Delta=2.

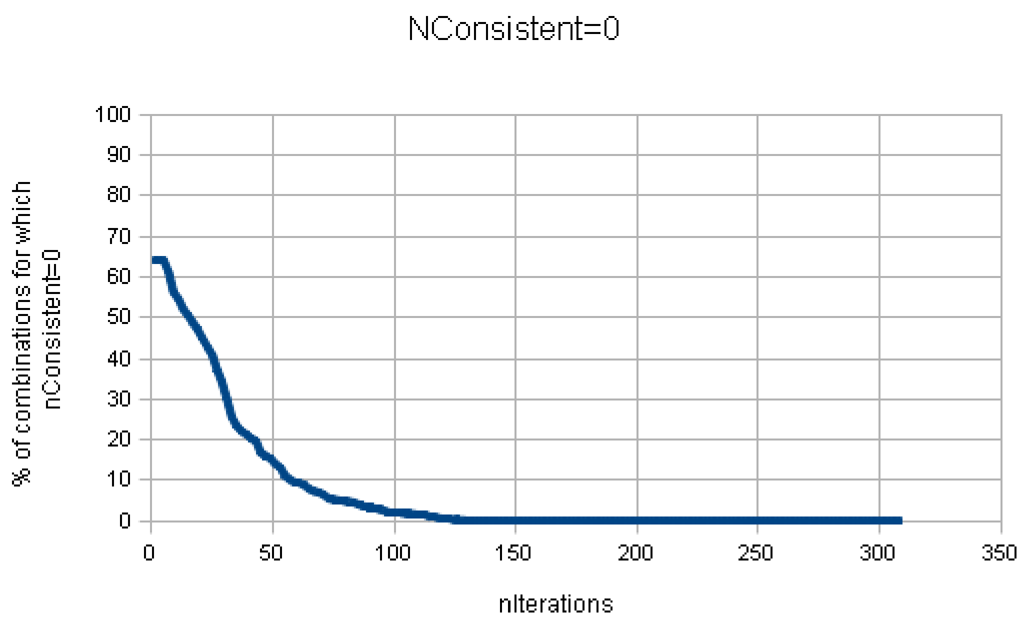

Figure 16: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.05, nComb=1000, delta=2.

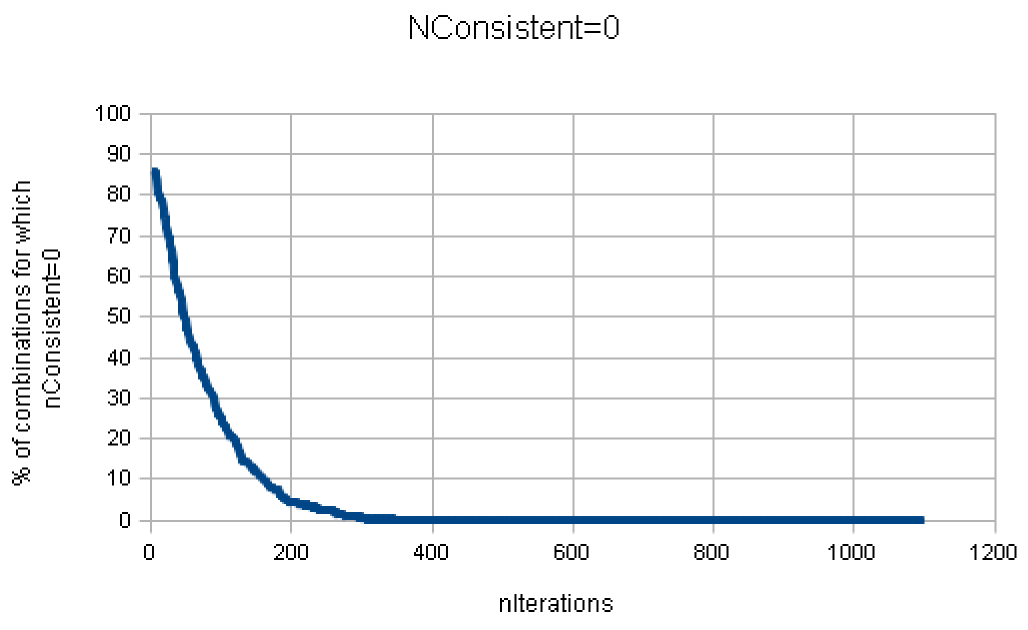

Figure 17: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.10, nComb=1000, delta=2.

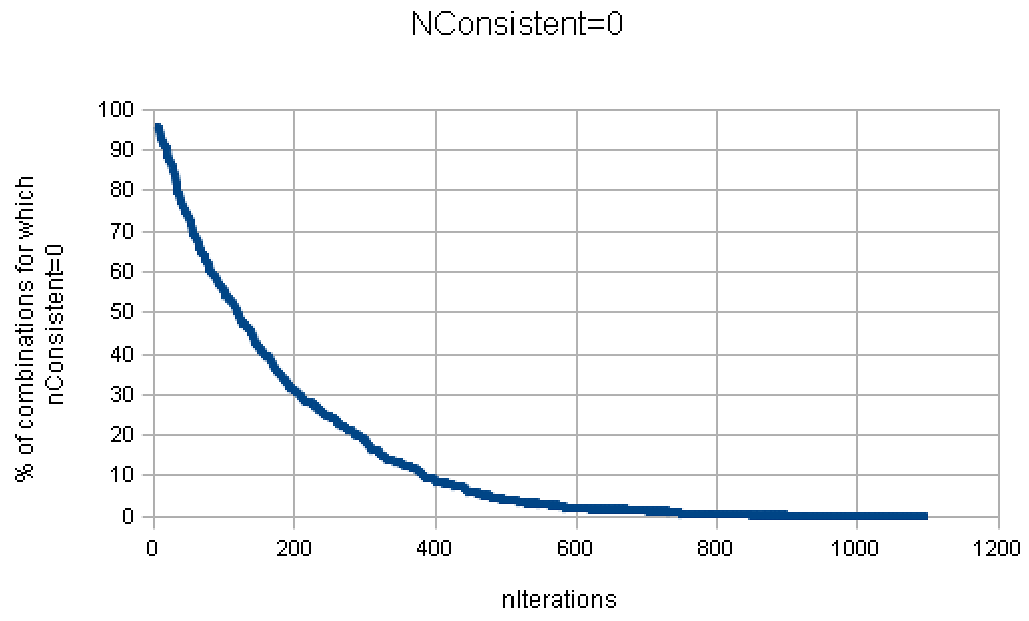

Figure 18: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.15, nComb=1000, delta=2.

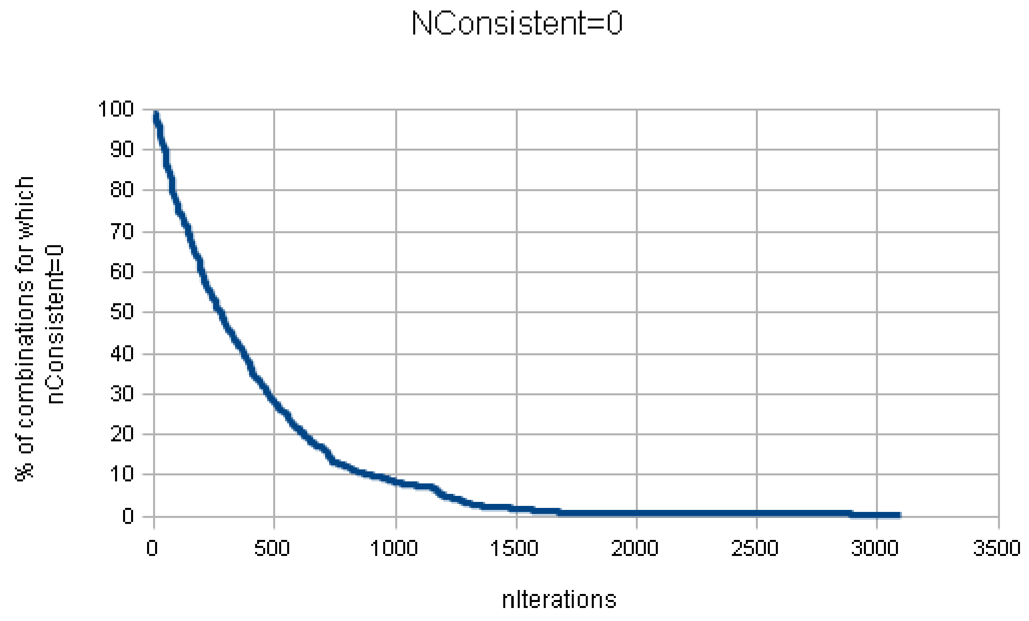

Figure 19: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.20, nComb=1000, delta=2.

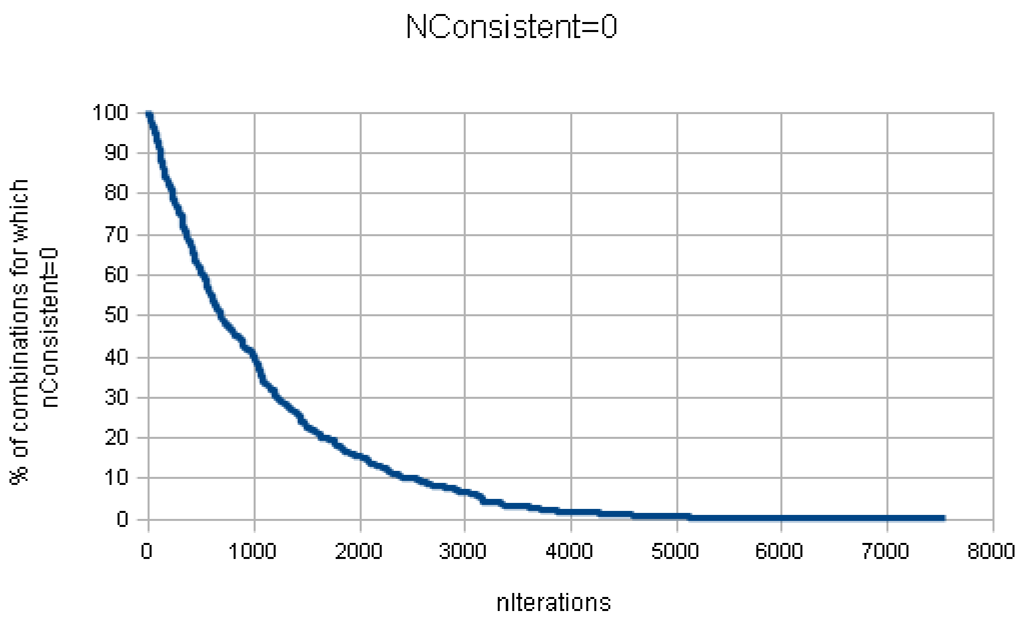

Figure 20: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.25, nComb=1000, delta=2.

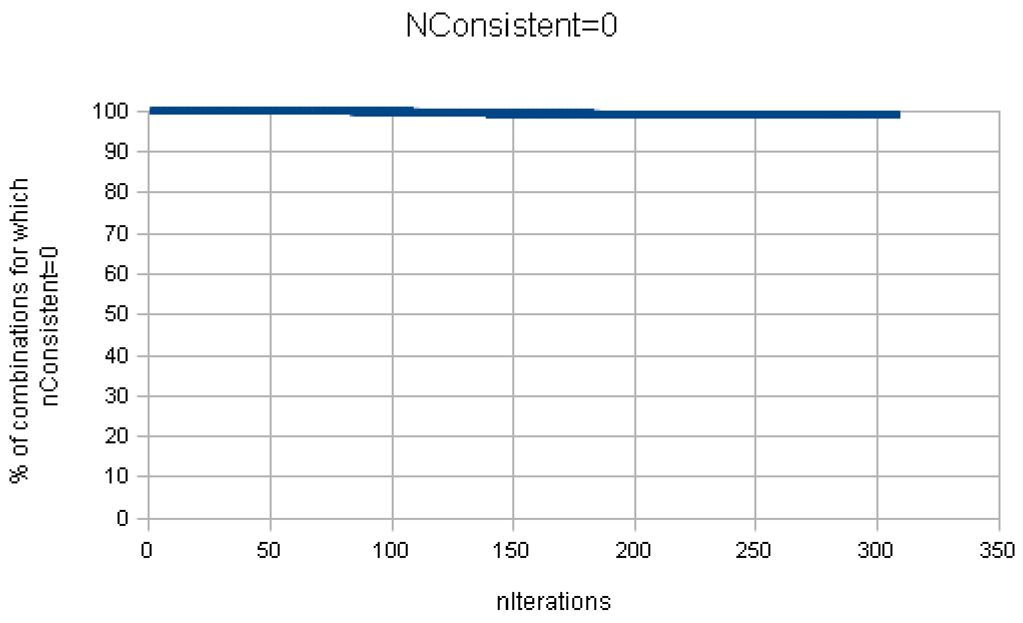

Figure 21: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.05, nComb=1000, delta=2.

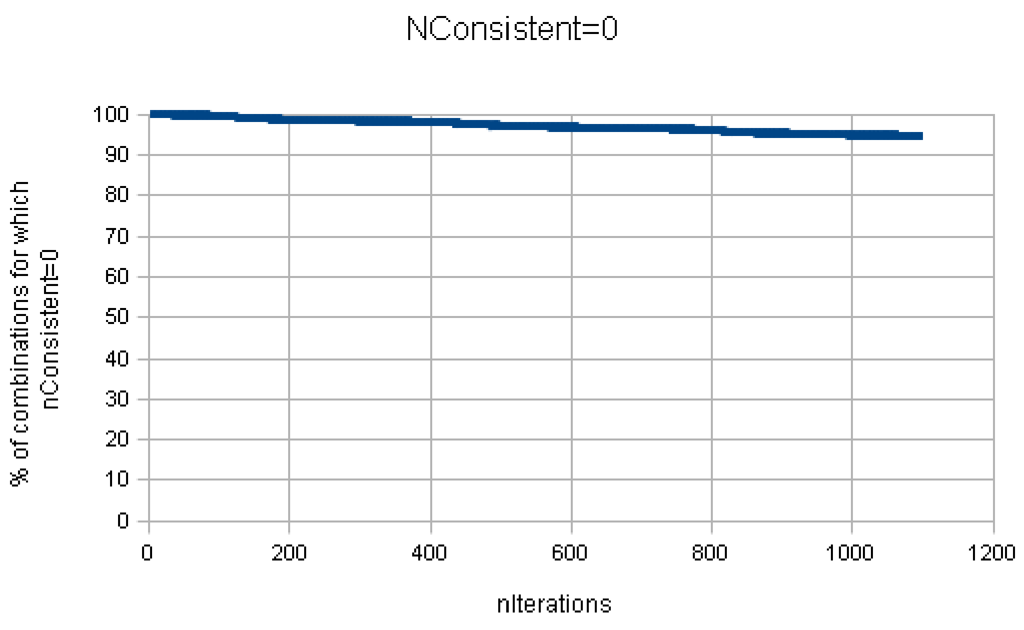

Figure 22: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.10, nComb=1000, delta=2.

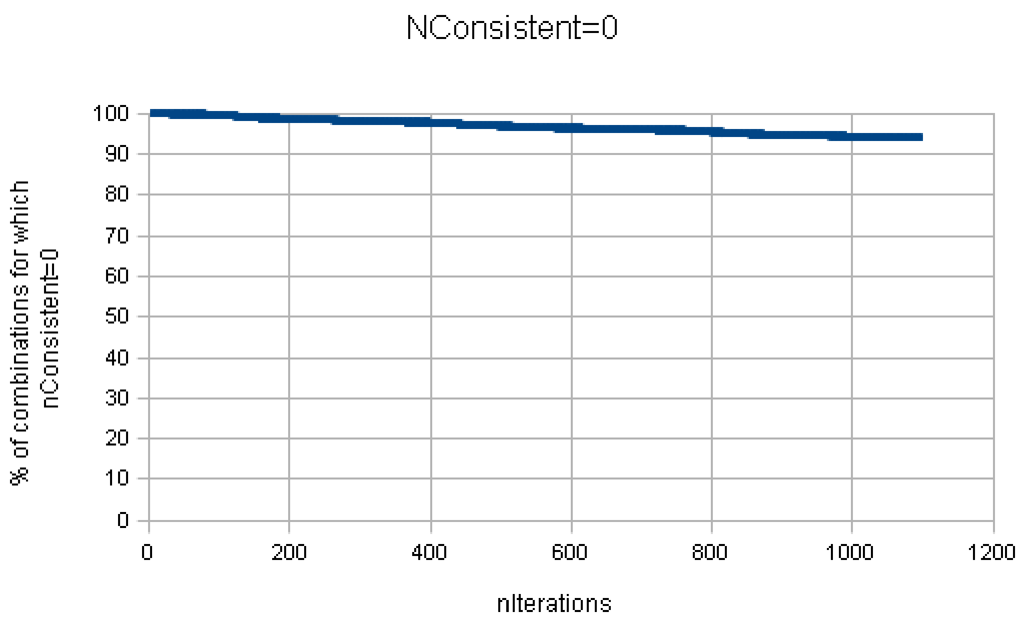

Figure 23: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.15, nComb=1000, delta=2.

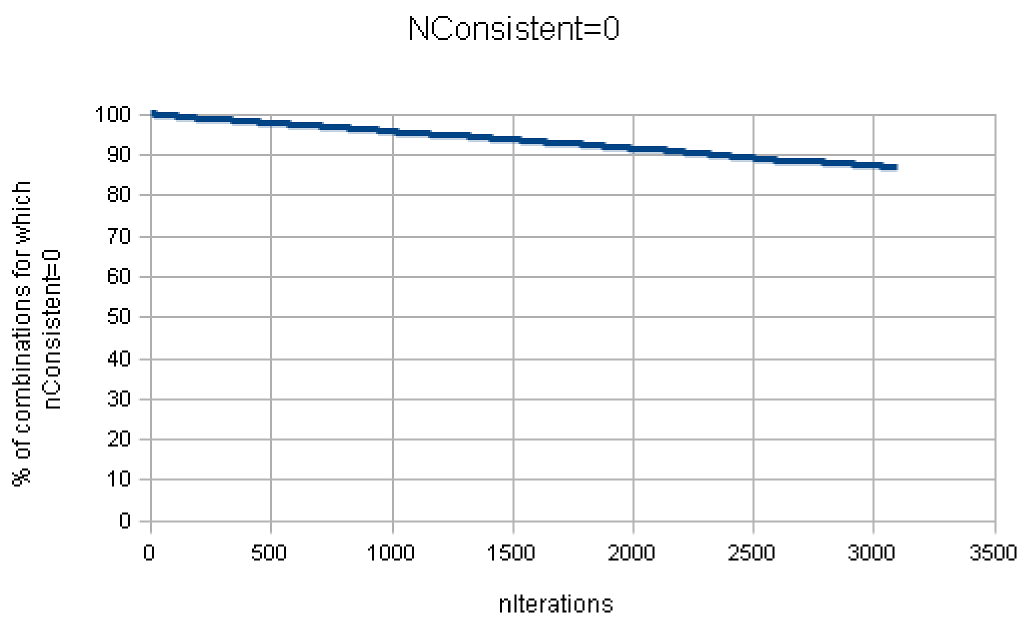

Figure 24: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.20, nComb=1000, delta=2.

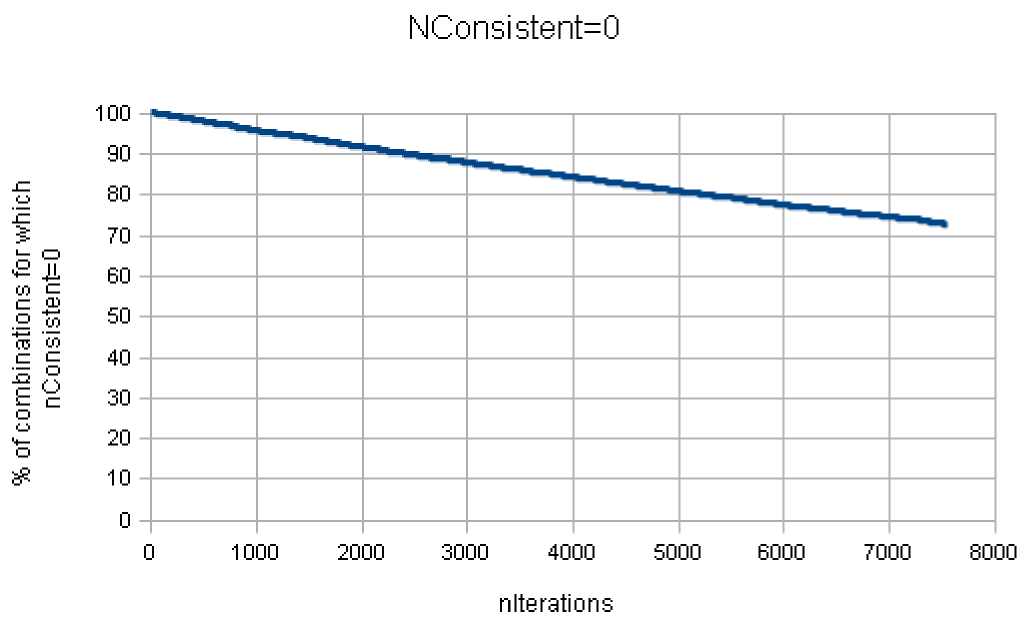

Figure 25: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.25. nComb=1000, delta=2.

$noiseLevel$	$lowestNIterations$	$nConsistent = 0$, case 1	$nConsistent = 0$, case 2
0.05	85	1.0%	92.04%
0.10	193	1.0 %	82.67%
0.15	355	1.0%	71.89%
0.20	799	1.0%	50.11%
0.25	1549	0.9%	28.69%

Table 14: Trade-off between $nConsistent = 0$ for case 1 and $nConsistent = 0$ for case 2. $f_s = 1 + x + x^4$, $f_u = 1 + x^3 + x^4$. Delta=1.

$noiseLevel$	$lowestNIterations$	$nConsistent = 0$, case 1	$nConsistent = 0$, case 2
0.05	183	1%	99.84%
0.10	659	1%	99.52%
0.15	2617	1%	98.29%
0.20	-	-	-
0.25	-	-	-

Table 15: Trade-off between $nConsistent = 0$ for case 1 and $nConsistent = 0$ for case 2. $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Delta=1.

	length=4 register			length=7 register	
delta/noise	0	1	2	0	1
0.05	225/12	225/12	225/12	500/25	250/13
0.10	1000/100	500/50	1000/100	1500/150	1000/100
0.15	1500/225	1000/150	1500/225	2040/306	2041/307
0.20	3000/600	1500/300	3000/600	2859/572	-
0.25	9000/2250	2000/500	15000/3750	-	-

Table 16: Experiment parameters: lenZ/number of altered bits. Case 1.

	length=4 register			length=7 register	
delta/noise	0	1	2	0	1
0.05	225/12	199/10	225/12	299/15	309/16
0.10	225/23	225/23	1000/100	599/60	739/74
0.15	225/34	500/75	1000/150	2019/303	2041/307
0.20	600/120	1200/240	3000/600	2866/574	-
0.25	900/225	2000/500	8000/2000	-	-

Table 17: Experiment parameters: lenZ/number of altered bits. Case 2.

	lowestNIterations		
length/noise	0.05	0.10	0.15
4	69	131	235
7	135	483	1625
12	259	619	2463

Table 18: Number of iterations needed such that the probability that a correct guess for I_s yields 0 consistent iterations is $\leq$ 1%. Delta=0.

	lowestNIterations		
length/noise	0.05	0.10	0.15
4	85	193	355
7	183	659	2617
12	221	799	3815

Table 19: Number of iterations needed such that the probability that a correct guess for I_s yields 0 consistent iterations is $\leq$ 1%. Delta=1.

4.4.6 Computational complexity

As explained in Section 3.3.1, in order to determine the growth of the c constant the results from the case 1 experiments could be used. However, this experiment was only run for register lengths 4 and 7. To be able to judge the growth of the c constant at least 3 (and preferably as many as possible) generators of increasing register lengths will be needed. Fortunately, for case 1 it was also possible to get results for length=12. Tables 18 and 19 show the results from the previous sections for register lengths 4 and 7, and additional results for length 12. It should be noted, however, that for length 12 $nCombinations$ was set to 10, so the results may be less accurate than for length 4 ($nCombinations = 1000$) and length 7 ($nCombinations = 100$). The results are further analysed in the next chapter.

5 Analysis

In Section 3.3, hypotheses H1, H2 and H3 were stated as follows:

H1 When $nIterations$ exceeds a certain level, the probability for a correct guess for I_s to yield $nConsistent = 0$ is very low.

H2 Even when $nIterations$ is high, the probability that a wrong guess for I_s yields $nConsistent = 0$ is close to 1.

H3 The growth of the c constant is linear (or sub-linear) with increasing register lengths

In this chapter, analysis is performed to decide whether these hypotheses can be accepted or not.

5.1 Hypothesis 1

From the data of Section 4.4, it is observed that H1 is fulfilled, since, for all levels of noise, deltas and registers considered, sooner or later the percentage of combinations yielding $nConsistent = 0$ for a correct guess reaches 0, and it seems that the percentage stays 0 for subsequent values of $nIterations$.

5.2 Hypothesis 2

For H2, it is not just as obvious whether the hypothesis can be accepted or not.

Tables 20 and 21 provide an overview of the results for $nConsistent = 0$. Table 20 shows the lowest value of $nIterations$ (denoted $lowestNIterations$) for which 1% or less of the combinations yield $nConsistent = 0$, for different noise levels, delta and register lengths. Table 21 shows the case 2 percentages corresponding to the $lowestNIterations$ values in table 20. This part of the analysis will mainly be concerned with the Case 2 table and aims at answering the H2 hypothesis.

From Table 21, visualized in Figure 26 several observations are made:

	length=4 register			length=7 register	
delta/noise	0	1	2	0	1
0.05	69	85	117	135	183
0.10	131	193	293	483	659
0.15	235	355	749	1625	2617
0.20	423	799	1607	4139	-
0.25	697	1549	4507	-	-

Table 20: $lowestNIterations$ for different noise levels, delta and register lengths

	length=4 register			length=7 register	
delta/noise	0	1	2	0	1
0.05	87.74%	92.04%	99.76%	98.84%	99.84%
0.10	73.18%	82.67%	98.46%	98.33%	99.52%
0.15	54.11%	71.89%	95.89%	95.56%	98.29%
0.20	35.94%	50.11%	93.04%	91.41%	-
0.25	18.90%	28.69%	82.48%	-	-

Table 21: Case 2 when $\mathtt{nIterations} = \mathtt{lowestNIterations}$ for different levels of noise and delta/register lengths

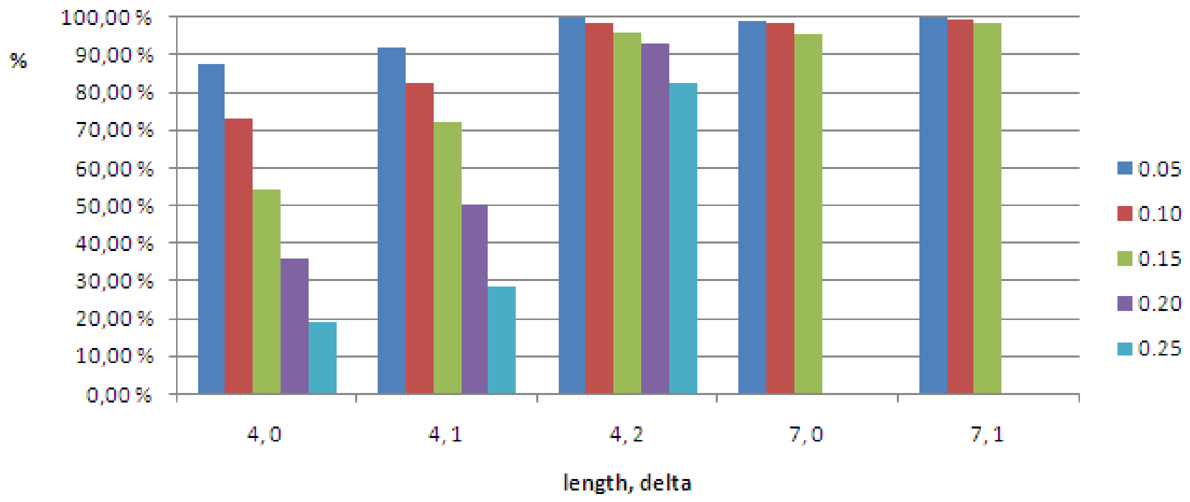

Figure 26: Case 2 when $\mathtt{nIterations} = \mathtt{lowestNIterations}$ for different levels of noise and delta/register lengths

- The percentage depends on $delta$ - the tendency is that an increase in $delta$ yields an increase in the percentage.
- The percentages are high (above 95%) for noise levels 0.05-0.15 for length=7. For length=4, only when delta=2 and the noise level is <= 0.15 the case 2 percentages exceed 95%.
- The percentages are (very) low for delta=0 and delta=1 for length=4 and noise levels > 0.15.
- For the length 4 registers, the percentages decrease rapidly for $delta = 0$ and $delta = 1$ as the level of noise increases, whereas for $delta = 2$ the percentages decrease slowly. For the length 7 registers we see the same pattern, but it is not equally obvious since, for technical reasons, no results were obtained for noise levels 0.20 and 0.25 for $delta = 1$ and 0.25 for $delta = 0$.

In order to obtain a good case 2 percentage, thus, $delta$ should not be too low. But obviously, since a higher $delta$ leads to a higher $lowestNIterations$ (see Table 20), an increase in $delta$ would lead to an increase in the runtime. But since the complexity of the LCT is only dependent on the length of the clocking register $LFSR_s$, which in a stream cipher is often significantly less than the length of the clocked register $LFSR_u$, an increase in the runtime will be accepted provided that the gain in terms of improved case 2 percentages is significant. The runtime is further analysed in the next section, and will not be considered further here.

An interesting question is why the percentages for the length 4 registers with $delta = 0$ and $delta = 1$ are low, while this is not the case for the length 7 registers. Would similar results be obtained if we used another combination of polynomials of length 4? One possible explanation is that since fewer equations are created for the length 4 generator (remembering that a new equation is added as long as $neq < (nvar + len_s + delta)$), an increase by 1 in the $delta$ variable have a greater impact for small register lengths than for larger register lengths. However, this question must be investigated further in order to justify any conclusions.

Based on the previous discussion, let $delta = 2$ for the length=4 registers, and $delta = 0$ for the length=7 registers. With these assumptions, we accept H2 for noise levels <= 0.20 for the length=4 and the length=7 registers.

5.3 Hypothesis 3

In this section, an attempt is made on estimating the computational complexity of our method based on the results from Section 4.4.6. The results are reproduced in tables 22 and 23. These tables additionally have three lines showing the correlation coefficient r for exponential, linear and logarithmic[1] regression. Each coefficient is calculated from the register lengths in the first column (x values) and $lowestNIterations$ from the column of the coefficient(y values). Values close to 1 indicate that there is a strong correlation between the length of the registers and $lowestNIterations$.

The correlation coefficients corresponding to the best approximations are emphasised in the tables, and show that only for noise level 0.05 and delta 0 a linear approximation seems to be the

[1]Natural logarithm (base e)

	lowestNIterations		
length/noise	0.05	0.10	0.15
4	69	131	235
7	135	483	1625
12	259	619	2463
r (exp)	0.98848399	0.86895996	0.87742192
r (linear)	**0.99951629**	0.92355683	0.95954188
r (log)	0.98288327	**0.97149029**	**0.99140005**

Table 22: lowestNIterations for different noise levels and register lengths. The correlation coefficients (r) in each column are obtained from lowestNIterations of the same column (y values) and the register lengths (x values). Delta=0.

	lowestNIterations		
length/noise	0.05	0.10	0.15
4	85	193	355
7	183	659	2617
12	221	799	3815
r (exp)	0.88725536	0.8568663	0.8684501
r (linear)	0.92385722	0.90283035	0.94953065
r (log)	**0.97167589**	**0.95815625**	**0.98643855**

Table 23: lowestNIterations for different noise levels and register lengths. The correlation coefficients (r) in each column are obtained from lowestNIterations of the same column (y values) and the register lengths (x values). Delta=1.

best fit, whereas for all other combinations of delta and noise level a logarithmic approximation fits best. However, since the number of samples is so small (only 3), the exact approximation function for each noise level and delta cannot be determined with sufficient confidence. But the fact that 5 out of 6 situations yields a logarithmic curve as the best approximation strongly indicates that the growth of $lowestNIterations$, and thus the c constant, is less than linear.

As an illustration of the complexity of the LCT with noise attack versus the complexity of a correlation attack, consider a generator of the BRM scheme with $length_s = X \in [4, 89]$ and $length_u = 89$. Let $noiselevel = 0.15$ and $delta = 1$. The complexity of the LCT attack is given by $c \times 2^X$ where $c = a + b \times \ln(89)$ assuming a logarithmic function and $c = a \times 89 + b$ assuming a linear function for the estimate of $nIterations$. Using the length of $LFSR_u$ (and not the length of $LFSR_s$) in the estimation of $nIterations$ ensures that the c constant is not underestimated. Table 24 shows the values of a and b obtained in the regression analysis. The complexity of the correlation attack is given by $2^X + 2^{89}$. In table 25 the number of operations for a correlation attack and the LCT attack is calculated for different values of the length of $LFSR_s$ with the length of $LFSR_u$ fixed to 89.

The table shows that, assuming a logarithmic function, the LCT performs better than the correlation attack for all lengths of $LFSR_s$ less than 76. Correspondingly, assuming a linear function, the LCT performs better than the correlation attack for $len_s < 74$. Also, it should be noted that for smaller register lengths, such as $len_s = 39$, the LCT performs several orders of magnitude

	a	b
log	-3855,5541	3155,11631
linear	412,816326	-902,59183

Table 24: Parameters a and b for logarithmic and linear regression.

len u	len s	log	linear	correlation
89	4	222739,1631	573408,9789	6,1897E+26
89	7	1808902,183	4587271,832	6,1897E+26
89	12	58004392,04	146792698,6	6,1897E+26
89	20	14850107528	37578930844	6,1897E+26
89	30	1,52065E+13	3,84808E+13	6,1897E+26
89	39	7,78574E+15	1,97022E+16	6,1897E+26
89	40	1,55715E+16	3,94044E+16	6,1897E+26
89	50	1,59452E+19	4,03501E+19	6,1897E+26
89	60	1,63279E+22	4,13185E+22	6,1897E+26
89	70	1,67197E+25	4,23101E+25	6,18971E+26
89	71	3,34395E+25	8,46202E+25	6,18972E+26
89	72	6,6879E+25	1,6924E+26	6,18975E+26
89	73	1,33758E+26	3,38481E+26	6,18979E+26
89	74	2,67516E+26	6,76962E+26	6,18989E+26
89	75	5,35032E+26	1,35392E+27	6,19008E+26
89	76	1,07006E+27	2,70785E+27	6,19046E+26
89	80	1,7121E+28	4,33256E+28	6,20179E+26
89	90	1,75319E+31	4,43654E+31	1,85691E+27

Table 25: Computational complexity. LCT with noise versus correlation attack.

better than the correlation attack.

6 Conclusion and Future work

The literature study demonstrated that a significant part of the research concerned with cryptanalysis of stream ciphers assumes that the plaintext is available and used as input to the attack algorithms. The exceptions include some correlation attacks that can handle moderate levels of noise. However, these attacks have a time complexity of $2^{l_s} + 2^{l_u}$, thus their complexity is dependent on the length of both registers.

In this thesis, an essentially algebraic attack - the Linear Consistency Test - is shown to be successful with low to moderate levels of noise in the intercepted sequence. The advantage of such an attack, compared to the existing correlation attacks, is that it could be implemented with a significantly lower runtime complexity, as demonstrated in Section 5.3.

A proof of concept of the attack scenario was conducted. The method proposed is an extension of the LCT attack. In the original LCT attack, the consistency test is run once for each guess for the initial state of $\mathtt{LFSR}_s$ with the plaintext as input. The proposed method takes as input a noisy version of the plaintext (i.e. the ciphertext). Then, the consistency test is run multiple times for each guess for the initial state of $\mathtt{LFSR}_s$, with a new starting point of the equations for each repetition. If the initial state was guessed wrongly, experimental results show that the probability is high (above 90%) that none of the iterations are reported as consistent. On the other hand, if the initial state was guessed correctly, the results show that the probability is very low (1% or less) that none of the iterations are reported as consistent. This applies to noise levels in the range 0 to 0.20.

However, as the results show, it is not clear how many equations are needed in the ciphertext only scenario in order to reduce the number of false consistency alarms. In the scenario without noise, as shown in Section 3.2, it was shown that for our chosen registers of length 4, when the number of equations exceeds the number of unknowns + the length of $\mathtt{LFSR}_s$ + 2, no false consistency alarms occured. [1] However, when noise is introduced, false consistency alarms cannot be avoided entirely simply because of the impact of the noise. This is reflected in the probability explained above. As we increase the number of equations (i.e. increase the $\mathtt{delta}$ variable), this probability will increase and approach 100% when the initial state is guessed wrongly. This is confirmed especially by the results for the registers of length 4, for example for $\mathtt{delta} = 0$ the results were very bad for noise levels above 0.10, but by increasing $\mathtt{delta}$ by 2, very good results were obtained even for noise level 0.20.

The same pattern can be observed for our registers of length 7, but in this case the results were good enough with $\mathtt{delta} = 0$. Thus, a possibility for future research would be to determine how many equations are needed given the feedback polynomials.

Molland [3] improved the Linear Consistency Test for irregularly clocked generators in the scenario without noise in terms of runtime complexity. It was shown that the complexity of the

[1] This could be verified because it was possible to test all combinations of initial states of the registers.

LCT attack could be reduced to 2^{l_s}. In Section 3.3.1 it is argued that if the attack of Molland could be extended by the method proposed in this thesis, the resulting ciphertext only attack would have a runtime complexity of $c \times 2^{l_s}$, where c is the number of repetitions of the consistency test in order to be able to distinguish a correctly guessed initial state from a wrongly guessed initial state. It is shown that, with this assumption, when the length of LFSR_s is sufficiently shorter than the length of LFSR_u, which is common in practical implementations, the time complexity of the esentially algebraic LCT attack is orders of magnitude less than that of correlation attacks. It should be noted that this is an estimation, and further research is needed in order to verify or disprove this, for example by implementing the LCT attack of Molland extended by the method proposed in this thesis, and run experiments for longer register lengths.

Bibliography

[1] Dawson, E., Clark, A., Golic, J., Millan, W., Penna, L., & Simpson, L. 2000. The lili-128 keystream generator.

[2] Chambers, W. G. & Jennings, S. M. Nov 1984. Linear equivalence of certain brm shift-register sequences. *Electronics Letters*, 20(24), 1018–1019.

[3] Molland, H. 2004. Improved linear consistency attack on irregular clocked keystream generators. In Roy and Meier [29], 109–126.

[4] Zeng, K., Yang, C. H., & Rao, T. R. N. 1989. On the linear consistency test (lct) in cryptanalysis with applications. In *CRYPTO 89: Proceedings on Advances in cryptology*, 164–174, New York, NY, USA. Springer-Verlag New York, Inc.

[5] Jennings, S. M. *A Special Class of Binary Sequences*. PhD thesis, University of London, 1980.

[6] Massey, J. L. & Rueppel, R. A. 1984. Linear ciphers and random sequence generators with multiple clocks. In *EUROCRYPT*, 74–87.

[7] Fluhrer, S. R. & Lucks, S. 2001. Analysis of the e0 encryption system. In *Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography*, SAC '01, 38–48, London, UK. Springer-Verlag.

[8] Petrovic, S. & Fuster, A. September 2004. Clock control sequence reconstruction in noisy generators with irregular clocking. In *Proceedings of the 3rd IASTED International Conference on Communication Systems and Networks*, 231–236, Malaga (Spain).

[9] Vernam, G. S. January 1926. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Transactions of the American Institute of Electrical Engineers*, XLV, 295 –301.

[10] Shannon, C. E. 1949. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28, 656–715.

[11] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. Stream ciphers. In *Handbook of applied cryptography* [15], chapter 6, 191–222.

[12] Zeng, K., Yang, C.-H., Wei, D.-Y., & Rao, T. R. N. 1991. Pseudorandom bit generators in stream-cipher cryptography. *Computer*, 24(2), 8–17.

[13] Beker, H. & Piper, F. Linear shift registers. In *Cipher Systems - The Protection of Communications* [30], chapter 5, 175–215.

[14] Massey, J. January 1969. Shift-register synthesis and bch decoding. *Information Theory, IEEE Transactions on*, 15(1), 122 – 127.

[15] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. 1997. *Handbook of applied cryptography*. CRC Press.

[16] Coppersmith, D., Krawczyk, H., & Mansour, Y. 1994. The shrinking generator. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '93, 22–39, London, UK. Springer-Verlag.

[17] Günther, C. 1988. Alternating step generators controlled by de bruijn sequences. In *Advances in Cryptology - EUROCRYPT 87*, Chaum, D. & Price, W., eds, volume 304 of *Lecture Notes in Computer Science*, 5–14. Springer Berlin / Heidelberg.

[18] Jönsson, F. & Johansson, T. 2002. A fast correlation attack on lili-128. *Inf. Process. Lett.*, 81(3), 127–132.

[19] Canteaut, A. 2005. Correlation attack. In *Encyclopedia of Cryptography and Security*, Tilborg, H. C. v., ed. Springer, Secaucus, NJ, USA.

[20] Siegenthaler, T. January 1985. Decrypting a class of stream ciphers using ciphertext only. *Computers, IEEE Transactions on*, C-34(1), 81 –85.

[21] Meier, W. & Staffelbach, O. January 1989. Fast correlation attacks on certain stream ciphers. *J. Cryptol.*, 1, 159–176.

[22] Johansson, T. & Jönsson, F. 1999. Improved fast correlation attacks on stream ciphers via convolutional codes. In *EUROCRYPT*, 347–362.

[23] Chepyzhov, V., Johansson, T., & Smeets, B. 2001. A simple algorithm for fast correlation attacks on stream ciphers. In *Fast Software Encryption*, Goos, G., Hartmanis, J., van Leeuwen, J., & Schneier, B., eds, volume 1978 of *Lecture Notes in Computer Science*, 124–135. Springer Berlin / Heidelberg.

[24] Palit, S., Roy, B., & De, A. 2003. A fast correlation attack for lfsr-based stream ciphers. In *Applied Cryptography and Network Security*, Zhou, J., Yung, M., & Han, Y., eds, volume 2846 of *Lecture Notes in Computer Science*, 331–342. Springer Berlin / Heidelberg.

[25] Golic, J. D. & Mihaljevic, M. J. 1991. A generalized correlation attack on a class of stream ciphers based on the levenshtein distance. *Journal of Cryptology*, 3, 201–212.

[26] Golic, J. & Petrovic, S. April 1996. Correlation attacks on clock-controlled shift registers in keystream generators. *Computers, IEEE Transactions on*, 45(4), 482 –486.

[27] Molland, H. & Helleseth, T. 2004. An improved correlation attack against irregular clocked and filtered keystream generators. In *CRYPTO*, Franklin, M. K., ed, volume 3152 of *Lecture Notes in Computer Science*, 373–389. Springer.

[28] Wagner, D. 2002. A generalized birthday problem. In *CRYPTO 02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, Yung, M., ed, volume 2442 of *Lecture Notes in Computer Science*, 288–304, London, UK. Springer-Verlag.

[29] Roy, B. K. & Meier, W., eds. *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*. Springer, 2004.

[30] Beker, H. & Piper, F. 1982. *Cipher Systems - The Protection of Communications*. Northwood Publications.

A Implementation details

This chapter includes the source code of the most important classes and methods of the attack. Some of the code is intentionally left out, such as the class that implements the shift registers (class $Lfsr$). Moreover, code presented earlier in the thesis (such as $ExperimentLctNoise$) is not repeated here.

A.1 LctNoise.cs

A.1.1 Method CreateDecimationVector()

This method creates the decimation vector, which is stored in the c array of the LctNoise class. The decimation vector corresponds to $u_{guessed}$ in Section 3.1. The elements of the c array are structs of type $Cell$, with member variables val and $varidx$. Table 26 shows the contents of the c array when CreateDecimationVector is run for the sample introduced in Section 3.1.

Thus, the values of $c[i].val$ which are different from 2 are obtained from the intercepted sequence (the output from the generator with or without noise), and the position of these values in the c array are determined by the decimating sequence s. So the val part of the c array contains the u sequence, except for some unknown values, denoted by '2'.

When the system of equations is to be created (see Section A.1.2), a value of 2 in $c[i].val$ indicates that a new variable should be created, and the index of the variable is stored in the corresponding $c[i].varidx$.

Listing A.1: Method LctNoise.CreateDecimationVector. Code provided by Slobodan Petrović.

```
public void CreateDecimationVector()
{
    int j = 0;

    //LFSR implementation where the contents of the register is stored in a circular
        buffer
    Lfsr r1 = new Lfsr(256, nConsR1, nTapsR1, r1seq.GetLength(0) - 1, r1seq, r1fdb,
        r1taps);

    for (int i = 1; i <= lenA; i++)
    {
        int fdb = r1.Shift(0);
        b[i] = r1.GetBuf()[r1.GetTaps()[1]];
        ncc = 0;
        for (int n = 1; n <= nTapsR1; n++)
        {
            ncc += r1.GetBuf()[r1.GetTaps()[1]] * (int)Math.Pow(2, nTapsR1 - n);
        }
        j += ncc + 1;
        c[j + r2fdb[nConsR2]].val = a[i];
    }

    lenDecVec = j;
}
```

i	c[i].val	c[i].varidx	$u_{guessed}$
0	-	-	
1	2	4	
2	2	3	
3	2	2	
4	2	1	
5	2	0	x_5
6	0	0	0
7	1	0	1
8	1	0	1
9	1	0	1
10	2	0	x_6
11	0	0	0
12	2	0	x_7

Table 26: Example contents of the decimation vector (c array of the LctNoise class), illustrating the relationship between c array and $u_{guessed}$ from Section 3.1.

A.1.2 Method CreateEquations(int k)

The method creates a system of equations from the decimation vector, using the feedback polynomial of the clocked register $LFSR_u$ (denoted by $r2$ in the implementation). Let the system of equations be defined by

$$\mathbf{Ax} = \mathbf{b}$$

Then, the coefficient matrix **A** is stored in the two dimensional array $amat$ and the **b** vector is stored in the array $cvec$ (members of $LctNoise.cs$).

The number of equations created (neq) is determined by the condition $while((neq < (nvar + r1fdb[nConsR1] + delta))\&\&(neq < (m-1)))$. Here, $nvar$ stores the number of unknowns, $r1fdb[nConsR1]$ the degree of the feedback polynomial and m the number of rows in the matrix. $delta$ is a parameter that is to be set when running an experiment, in order to be able to increase the number of equations created.

The argument k determines the starting point of the first equation - that is, when $k = 0$, the equations are created starting from $c[5]$ (see Table 26 and Section A.1.1). Similarly, when $k > 0$, the starting point is $c[k+5]$.

Listing A.2: Method LctNoise.CreateEquations

```
public void CreateEquations(int k)
    {
        int j;

        neq = 0;
        nvar = r2fdb[nConsR2]; int i = nvar;  //set i = nvar = degree of feedback
            polynomial
        int numneg; int sum;

        //to avoid a large number of (unused) variables, only the variables needed are
            created
        if (k > 0)
        {
            i = k + r2fdb[nConsR2];
            nvar = 0;
            for (int r=i; r > k; r--)
            {
                if (c[r].val == 2)
                {
                    nvar++;
                    c[r].varidx = nvar;
                }
            }
        }

        while ((neq < (nvar + r1fdb[nConsR1] + delta)) && (neq < (m - 1)))
        {

            i++;
            if (i >= lenDecVec)
            {
                Console.WriteLine("ERROR:␣i␣>=␣lenDecVec");
                Console.WriteLine("i={0},␣lenDecVec={1}", i, lenDecVec);
                throw new Exception("ERROR:␣i␣>=␣lenDecVec");
            }

            if (c[i].val == 2) //A new variable is to be added to the system
            {
                nvar++;
                c[i].varidx = nvar;
                pos[1] = -c[i].varidx;
            }
            else
            {
                pos[1] = c[i].val;
            }
            for (j = 1; j <= nConsR2; j++)
            {
                if (c[i - r2fdb[j]].varidx > 0)
                {
                    pos[1 + j] = -c[i - r2fdb[j]].varidx;
                }
                else
                {
                    pos[1 + j] = c[i - r2fdb[j]].val;
                }
            }
            numneg = 0;
            for (j = 1; j <= nConsR2 + 1; j++)
            {
                if (pos[j] < 0) numneg++;
            }

            sum = 0;

            int[] equation = new int[nvar + 1];

            for (j = 1; j <= nConsR2 + 1; j++)
            {
                if (pos[j] < 0) equation[Math.Abs(pos[j])] = 1;
                else sum ^= pos[j];
            }

            //Add a new equation to the system if not added previously
            if (AddEquation(neq + 1, equation, sum))
            {
                neq++;
            }
        }
}
```

The method *LctNoise.AddEquation* (Listing A.3) checks whether the equation (in *equation* and *sum*) already exists in the matrix in one of the rows above *row*. If not, the equation is added to *amat* and the sum to *cvec*. The method returns true if the equation was added.

Listing A.3: Method LctNoise.AddEquation

```
private bool AddEquation(int row, int[] equation, int sum)
{

    bool exists = false;
    for (int i = 1; (i <= neq) && !exists; i++)
    {
        exists = true;

        if (cvec[i] != sum) exists = false;

        if (exists)
        {
            for (int j = 1; j <= nvar; j++)
            {
                if (amat[i][j] != equation[j]) exists = false;
            }
        }
    }

    if (!exists)
    {
        for (int j = 1; j < equation.GetLength(0); j++)
        {
            amat[row][j] = equation[j];
        }
        cvec[row] = sum;
    }

    return !exists;
}
```

A.1.3 Method IsConsistent()

The method checks whether the system of equations in *amat* and *cvec* is consistent, and returns true if it is. As explained in Section 3.2, the system is consistent if and only if the right hand side of the system of equations (that is, *cvec* in the implementation) is contained in the subspace spanned by the column vectors of the A matrix (*amat*). Thus, to check the system for consistency, the *IsInLa* method of class *Matrix* is invoked. This method is further explained in the next section.

Listing A.4: Method LctNoise.IsConsistent

```
public bool IsConsistent()
{
  Matrix mAmat = new Matrix(GetAmat());

  bool consistent = mAmat.IsInLa(GetCvec());
```

```

    return consistent;
}
```

A.2 Matrix.cs

A.2.1 Method IsInLa(int[] vector)

The method checks whether $vector$ is contained in the set $L(A)$, that is, in the set of all linear combinations of the column vectors from the matrix. The $positions$ variable determines the columns to combine, i.e. positions=7 (0...0111 binary) means that the 3 last columns of the matrix should be combined to yield the sum $v_{n-2} + v_{n-1} + v_n$ (where the v's are the numbered column vectors of the matrix).

The equality between some linear combination ($actualLinearComb$) and $vector$ is checked bit by bit starting from $vector[0]$ - such that if $vector[i]$ does not equal the i'th bit of $actualLinearComb$, this linear combination is discarded and a new one is checked starting from vector[0]. If all bits match, i.e. $vector$ equals $actualLinearComb$, then $vector$ is in $L(A)$ and the method returns true. On the other hand, if $vector$ does not equal any linear combination, the method returns false.

The equality check between $vector[i]$ and the i'th bit (denoted by $current$) of $actualLinearComb$ is performed as follows:

1. Compute the bitwise AND between row i of the matrix and $positions$. Store the result in the and variable.
2. Compute the sum of and by counting the number of 1's in and. Set $current = 1$ if odd number of 1's, $current = 0$ otherwise.
3. Compare $current$ to $vector[i]$.

In order for the method to be efficient, the computation of the number of 1's in the binary form of a decimal number uses the optimized algorithm in http://blogs.media-tips.com/bernard.opic/2007/09/03/compter-le-nombre-de-1-dans-la-forme-binaire-d-un-nombre/ (last visited 2011/04/26).

Listing A.5: Method Matrix.IsInLa

```
public bool IsInLa(int[] vector)
        {
            int max = (int) Math.Pow(2, GetN());

            //convert each row of the matrix to a decimal number
            int[] matrixDec = GetDecimalMatrix();

            int m = GetM();

            //for each decimal number in [1, 2^numCols>
            for (int positions = 1; positions < max; positions++)
            {
                int i = -1;
                int current = 0;
                int and;

                //for each matrix row (i)
                do
                {
                    i++;
                    and = matrixDec[i] & positions;

                    //compute the number of 1's in the binary form of 'and'
                    int c = 0;
                    while (and != 0)
                    {
                        c += (int)(and & 1L);
                        and >>= 1;
                    }

                    current = c % 2; //1 if odd number of 1's, 0 otherwise

                //continue as long as the corresponding bits of the linear combination
                    and vector are equal
                } while (current == vector[i] && i < (m - 1));


                if (i == (m - 1) && current == vector[i])
                {
                    return true; //vector is found in L(A)
                }
            }

            //vector is not found in L(A)
            return false;
        }
```

B Graphs

B.1 Length=4 registers

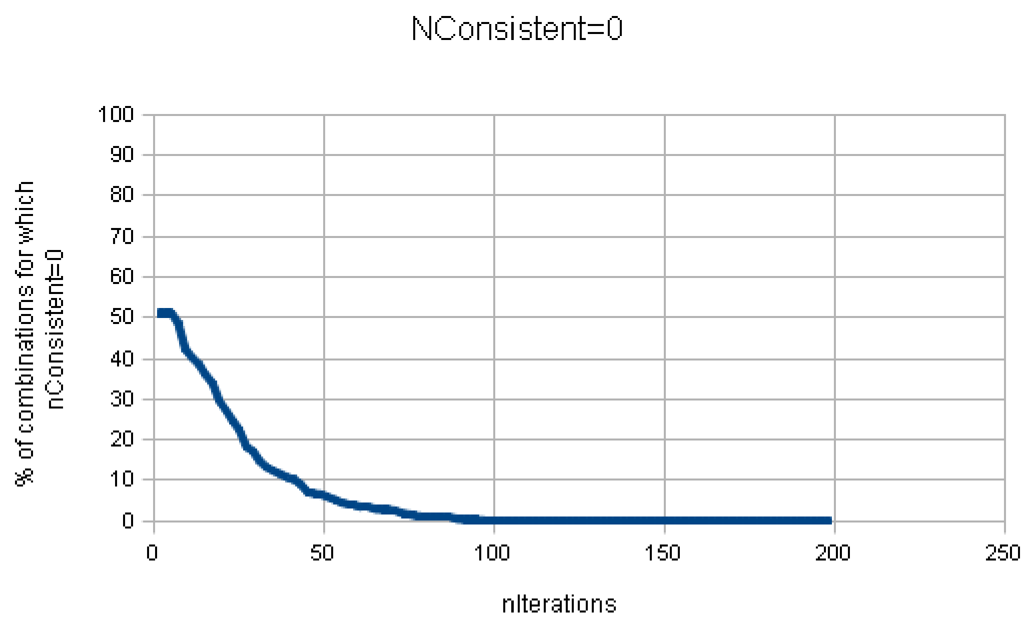

Figure 27: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.05, nComb=1000, delta=1.

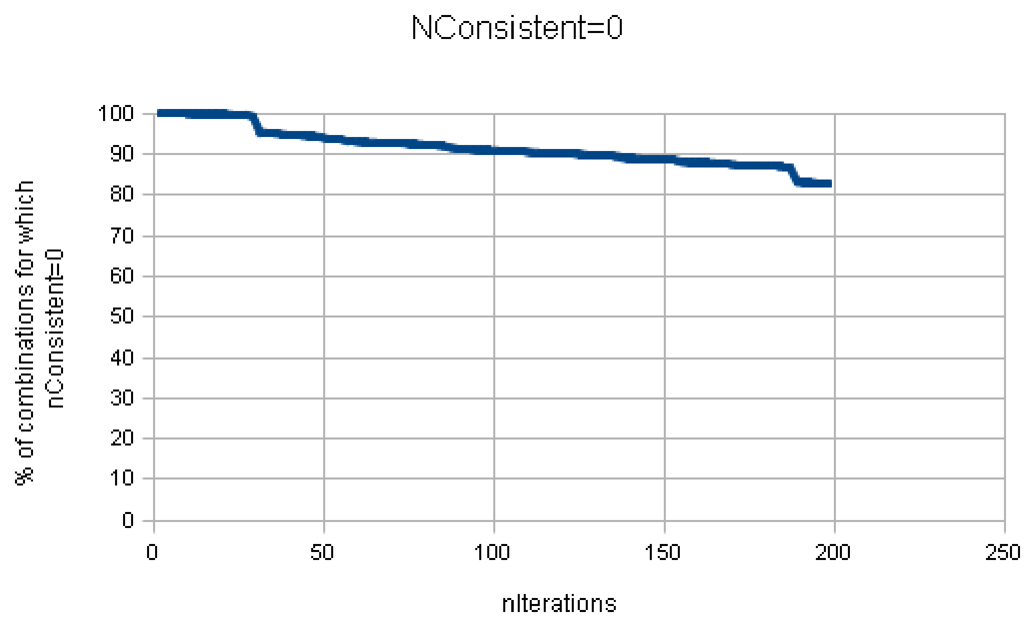

Figure 28: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.05, nComb=1000, delta=1.

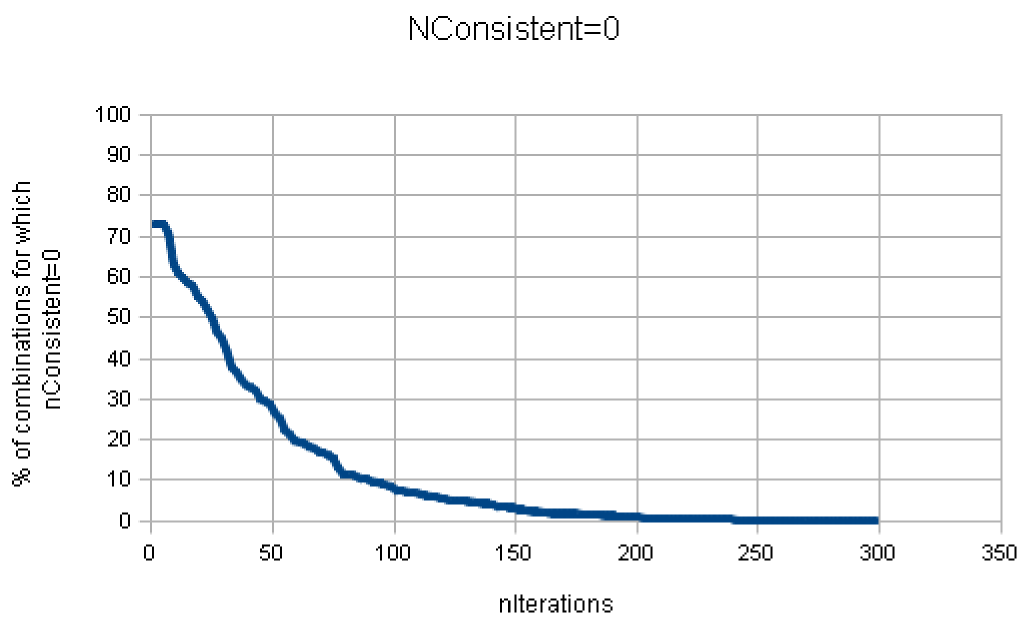

Figure 29: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.10, nComb=1000, delta=1.

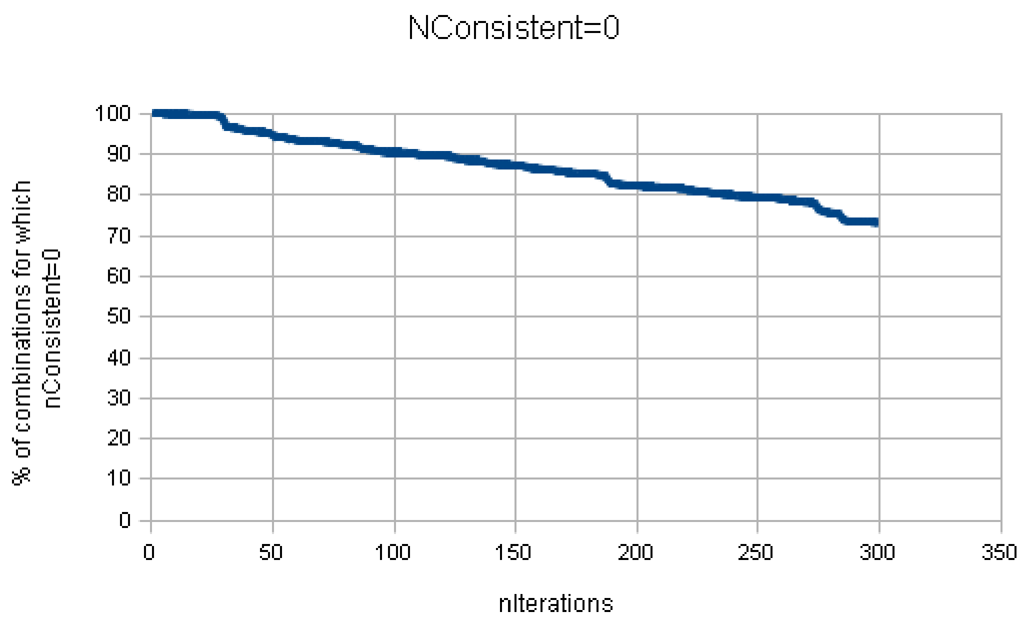

Figure 30: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.10, nComb=1000, delta=1.

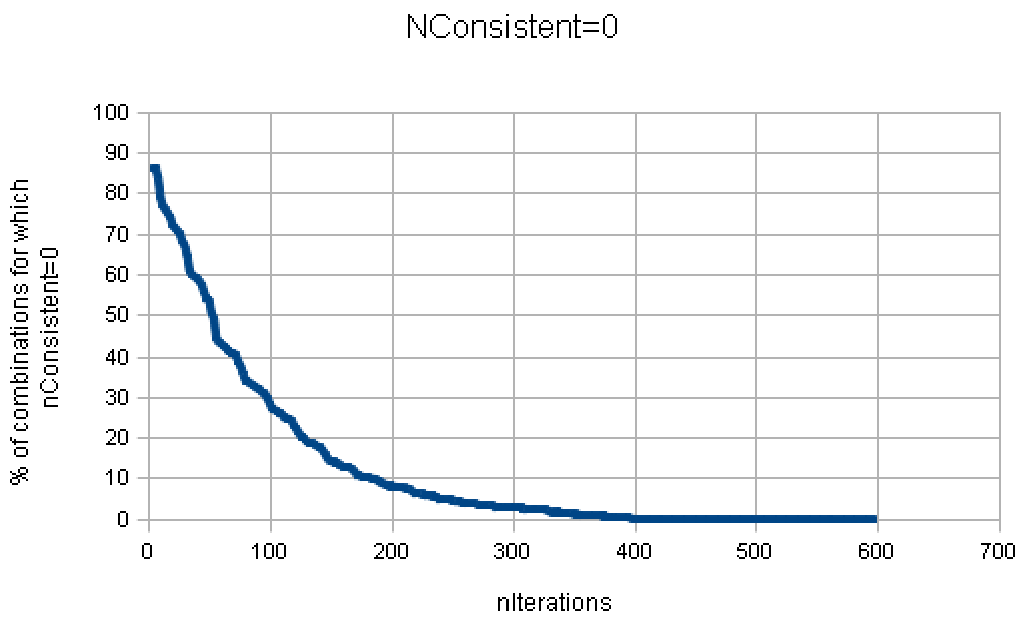

Figure 31: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.15, nComb=1000, delta=1.

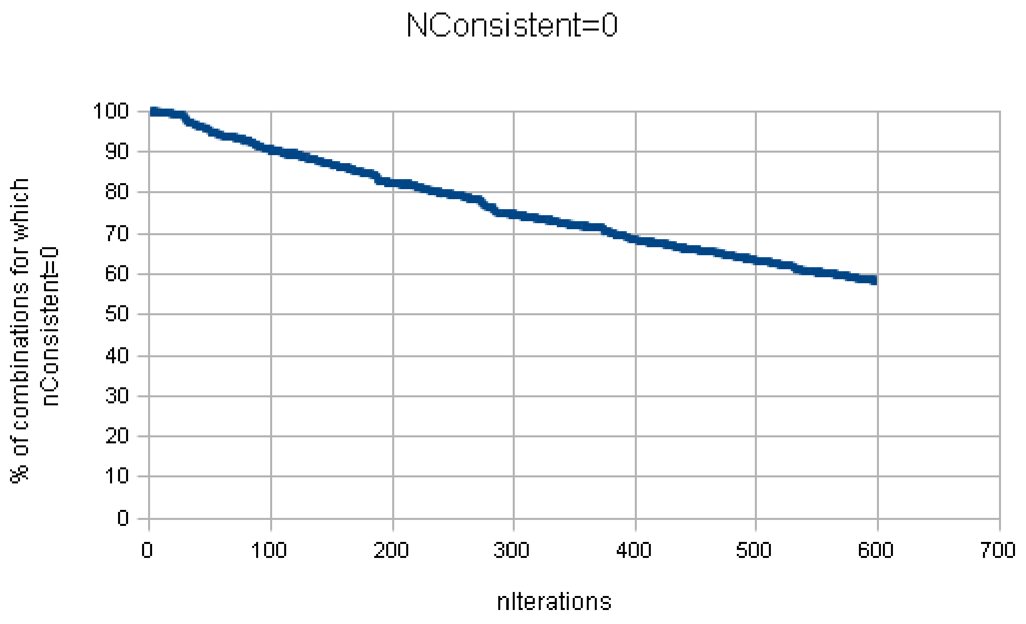

Figure 32: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.15, nComb=1000, delta=1.

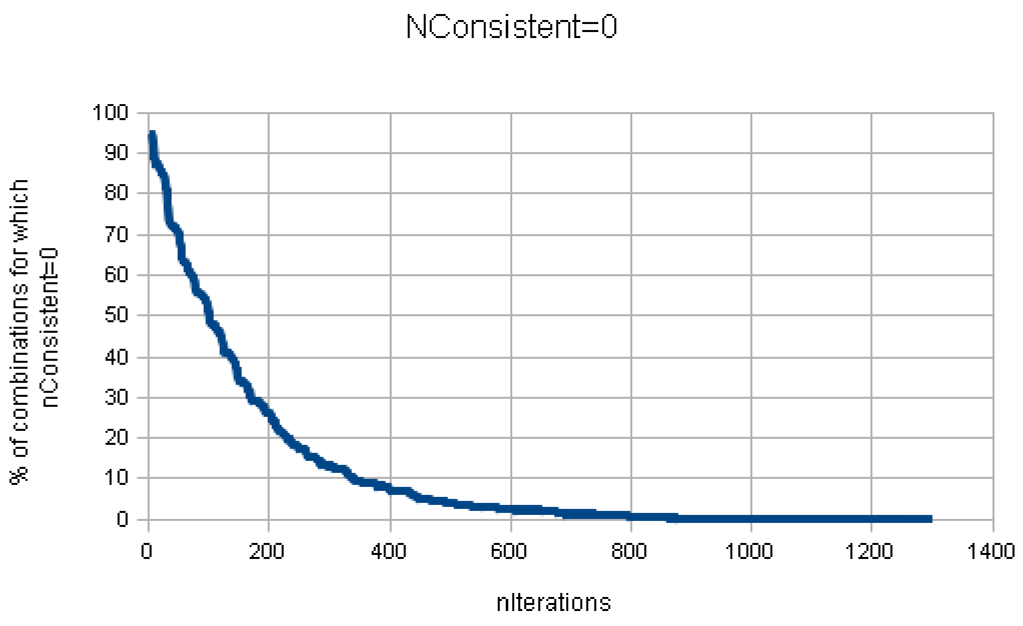

Figure 33: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.20, nComb=1000, delta=1.

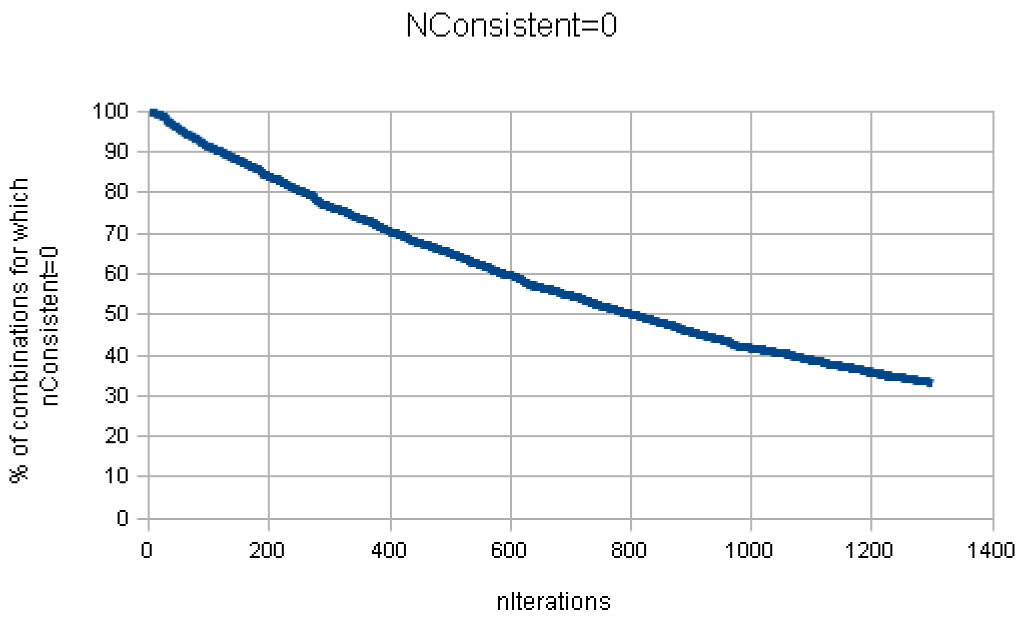

Figure 34: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.20, nComb=1000, delta=1.

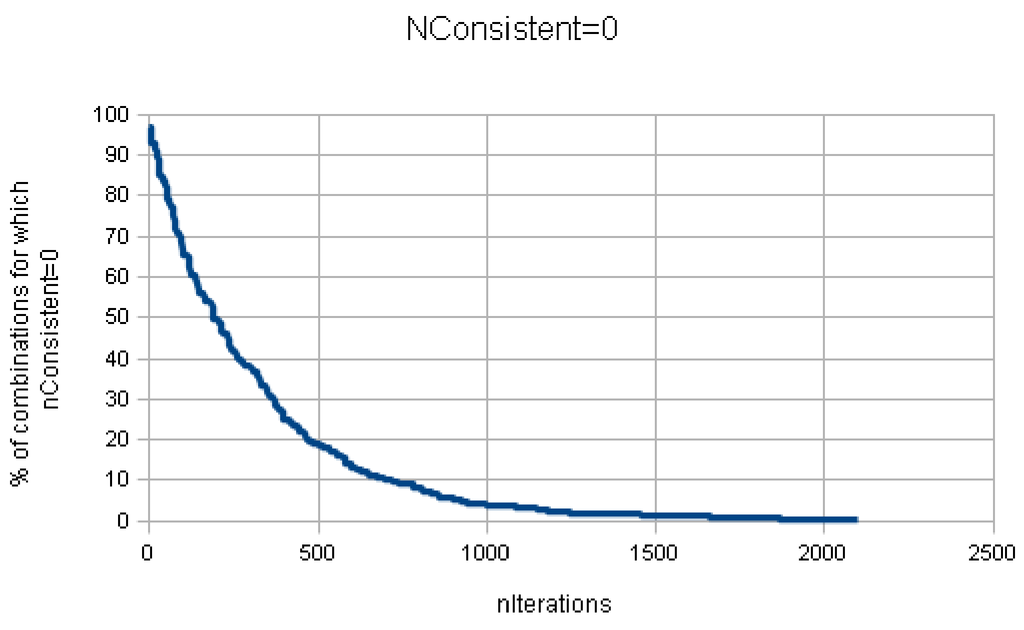

Figure 35: Case 1: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.25, nComb=1000, delta=1.

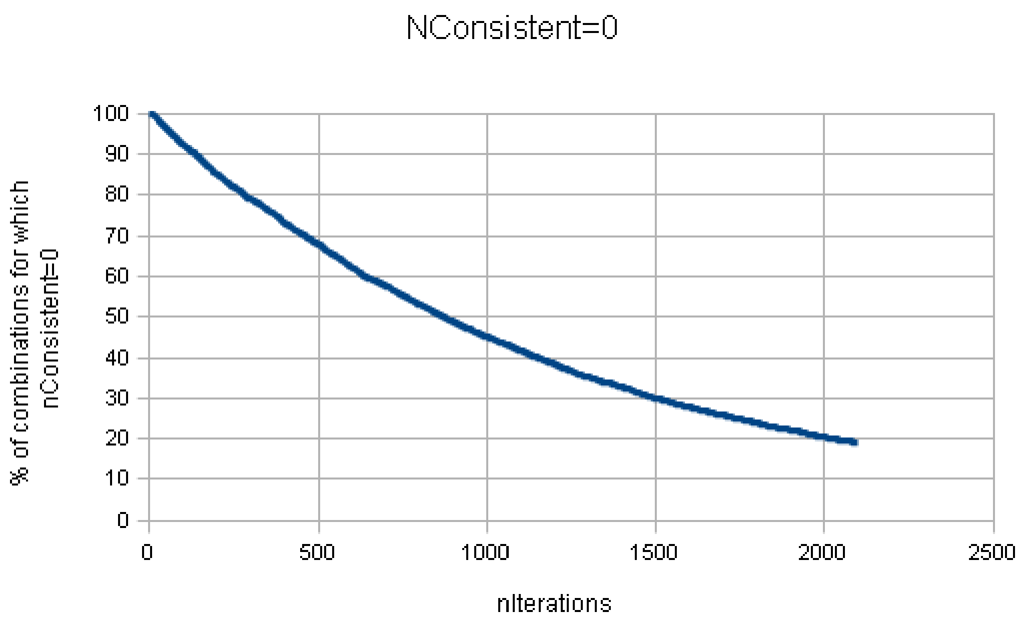

Figure 36: Case 2: $f_s(x) = 1 + x + x^4$ and $f_u(x) = 1 + x^3 + x^4$. Noise level=0.25, nComb=1000, delta=1.

B.2 Length=7 registers

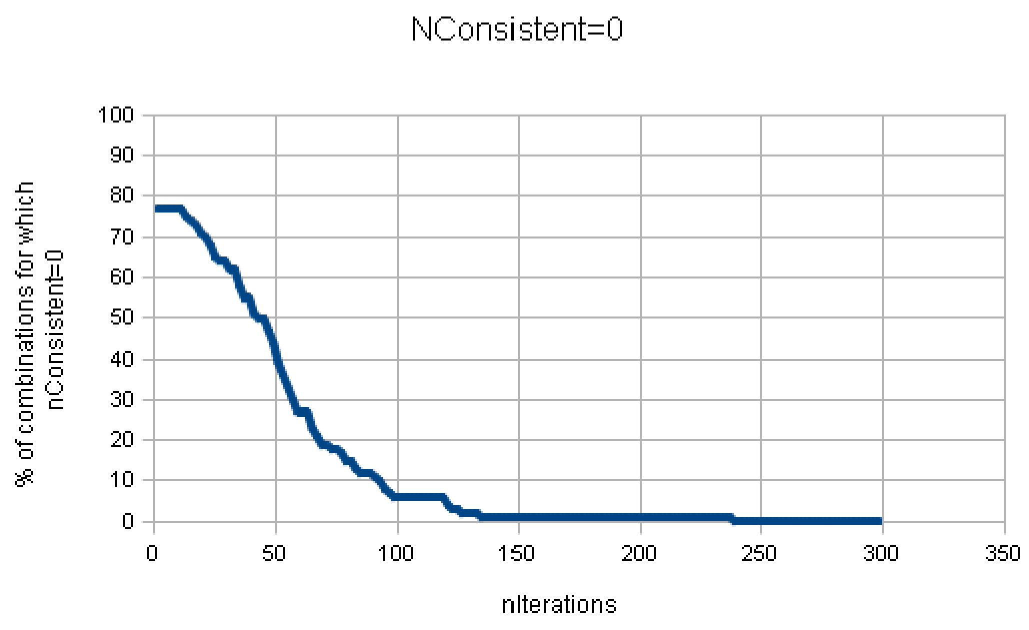

Figure 37: Case 1: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.05, nComb=100, delta=0.

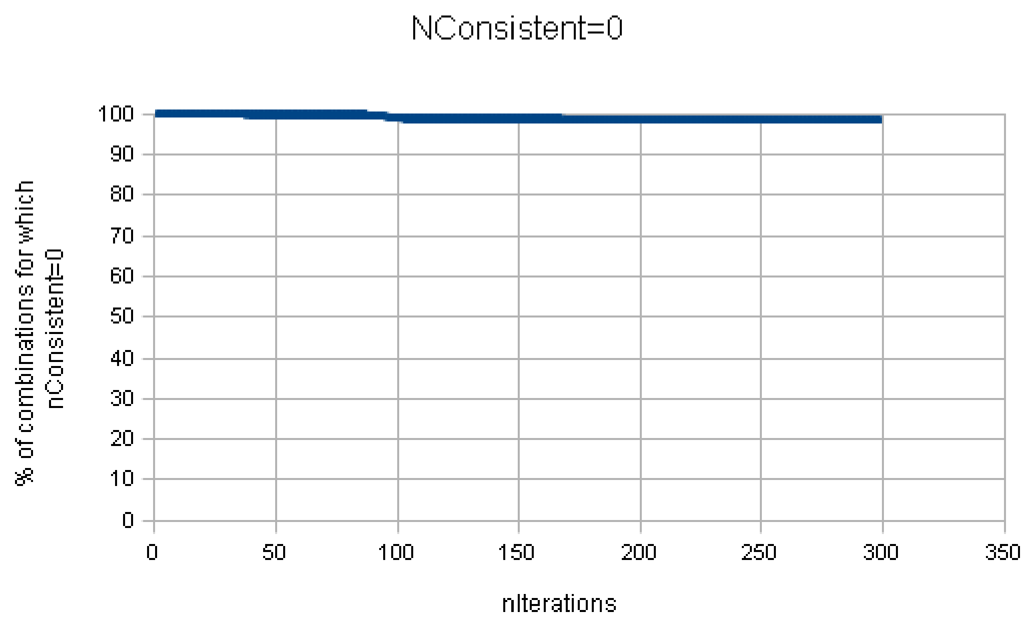

Figure 38: Case 2: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.05, nComb=100, delta=0.

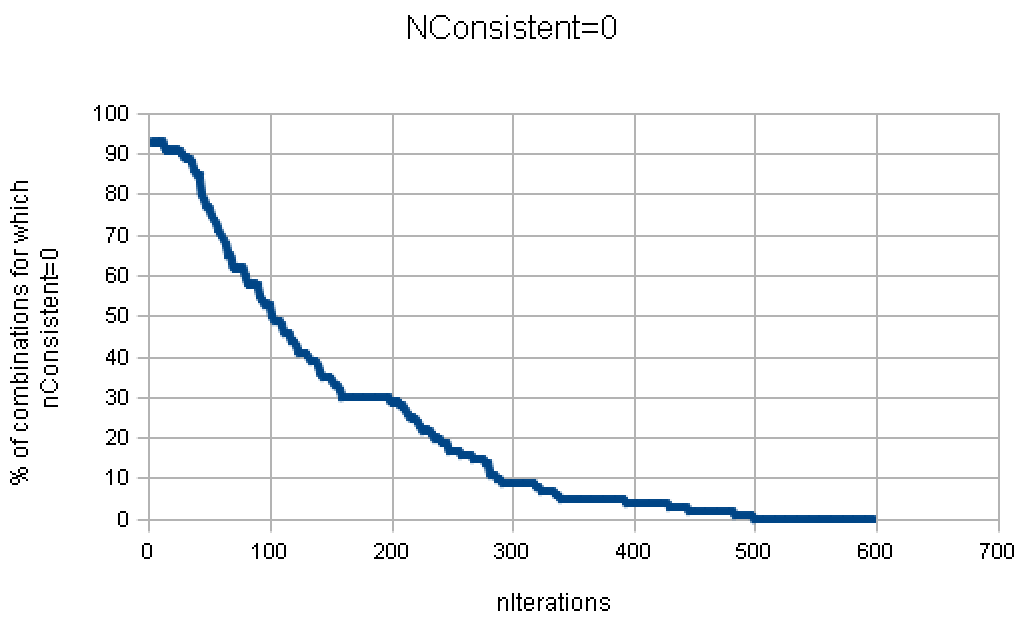

Figure 39: Case 1: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.10, nComb=100, delta=0.

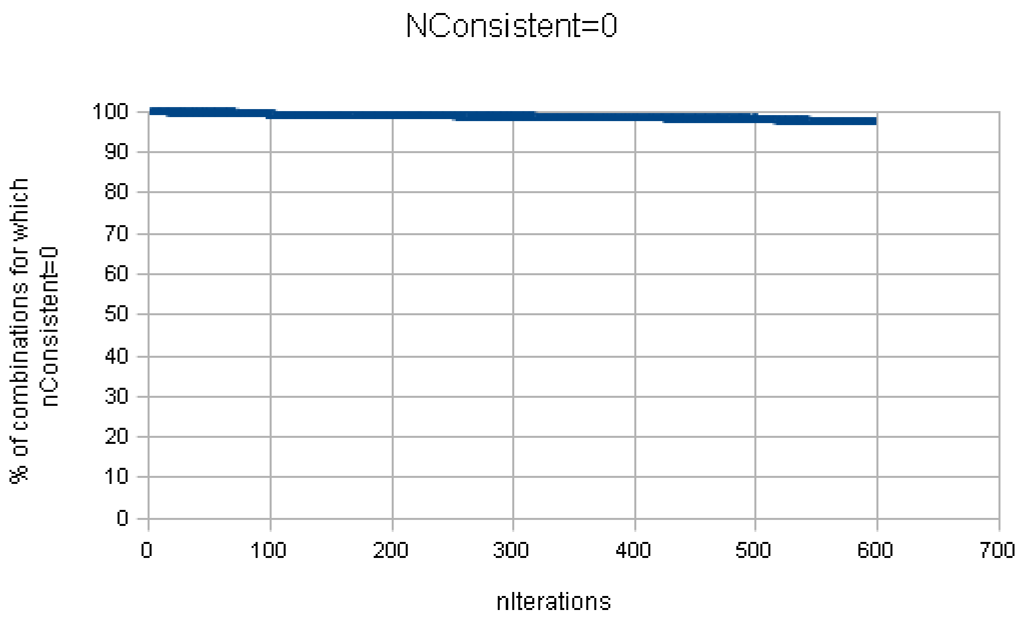

Figure 40: Case 2: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.10, nComb=100, delta=0.

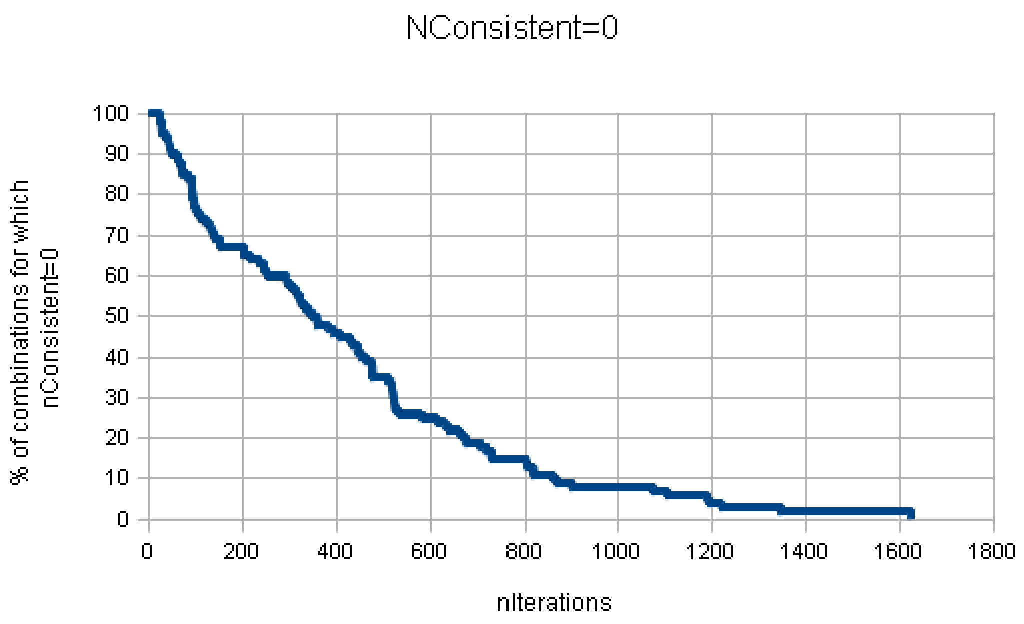

Figure 41: Case 1: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.15, nComb=100, delta=0.

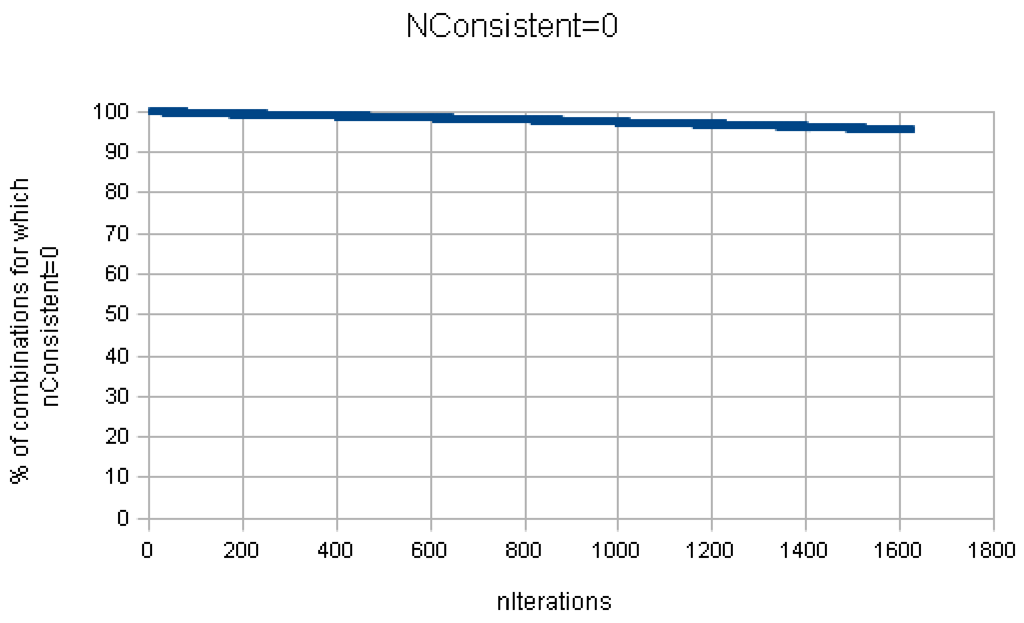

Figure 42: Case 2: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.15, nComb=100, delta=0.

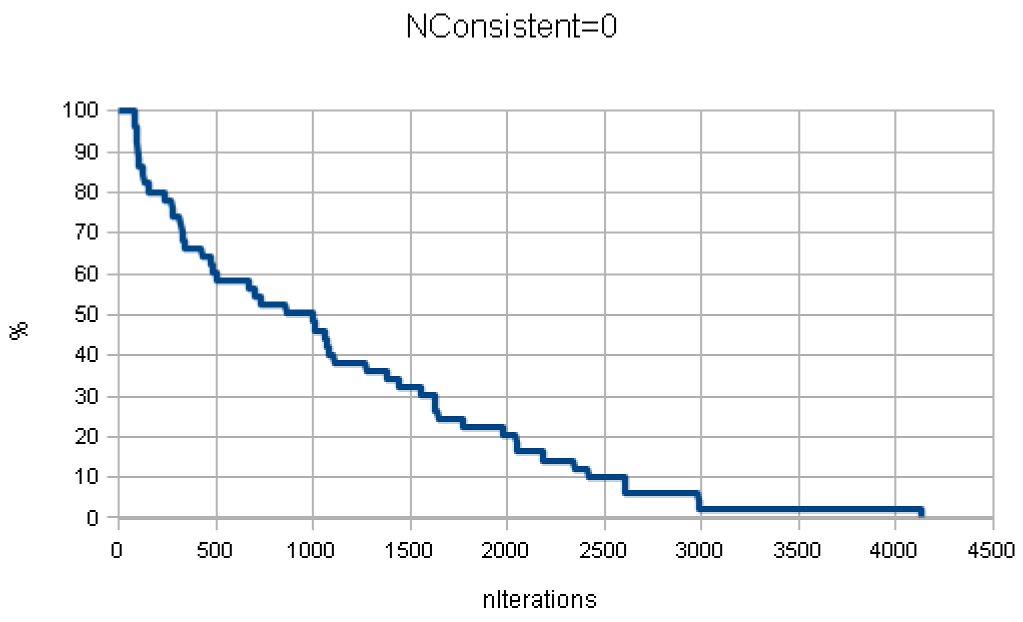

Figure 43: Case 1: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.20, nComb=50, delta=0.

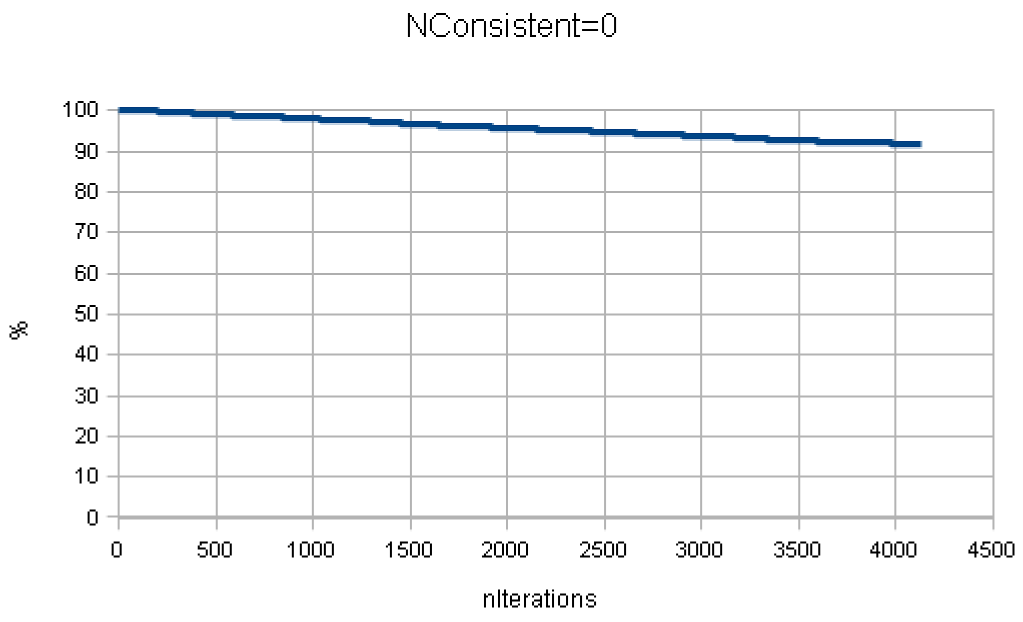

Figure 44: Case 2: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.20, nComb=50, delta=0.

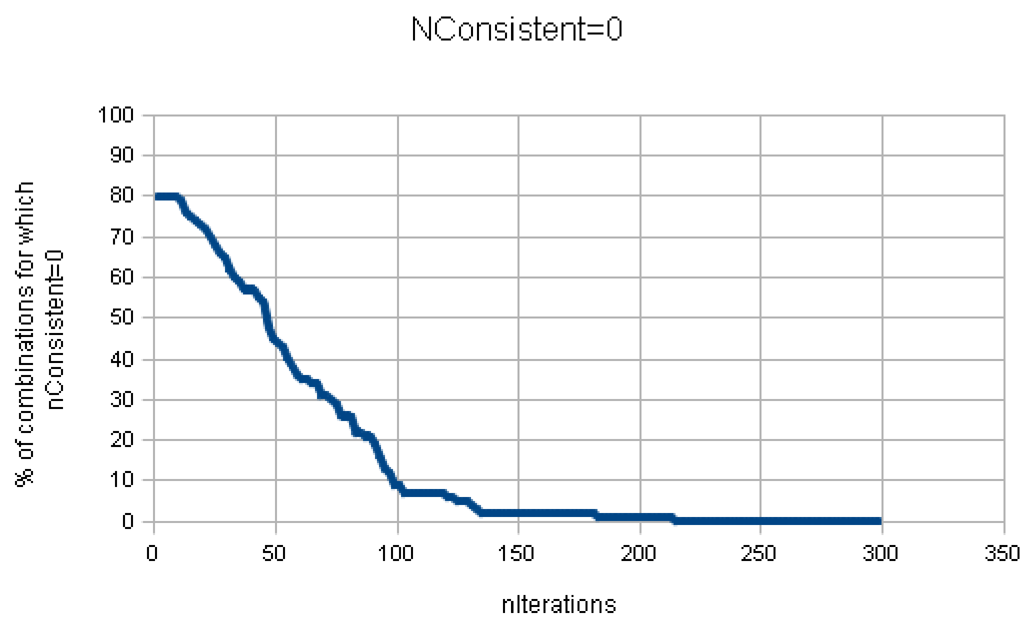

Figure 45: Case 1: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.05, nComb=100, delta=1.

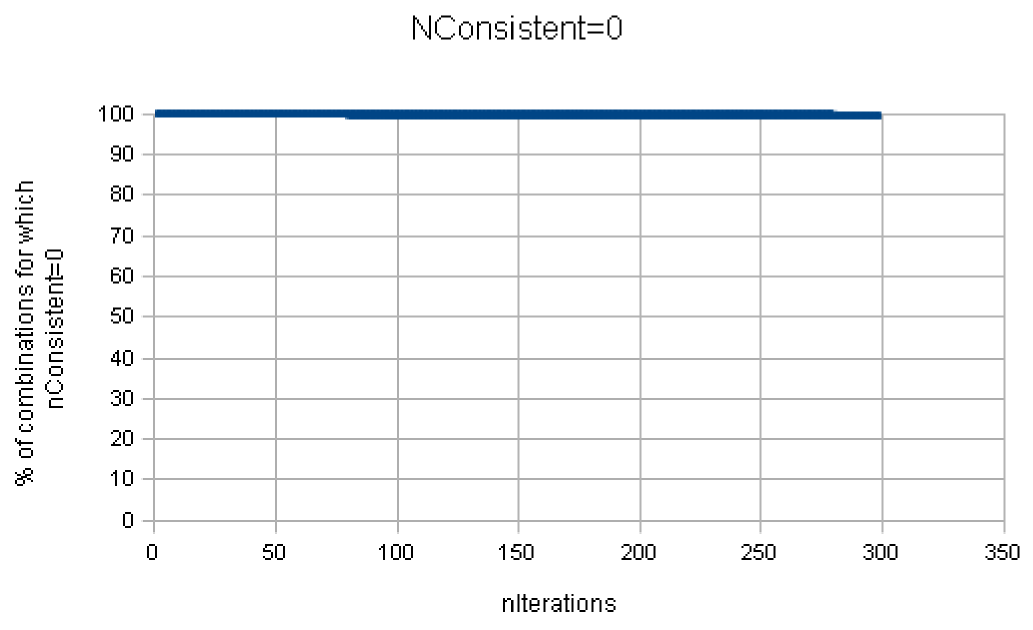

Figure 46: Case 2: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.05, nComb=100, delta=1.

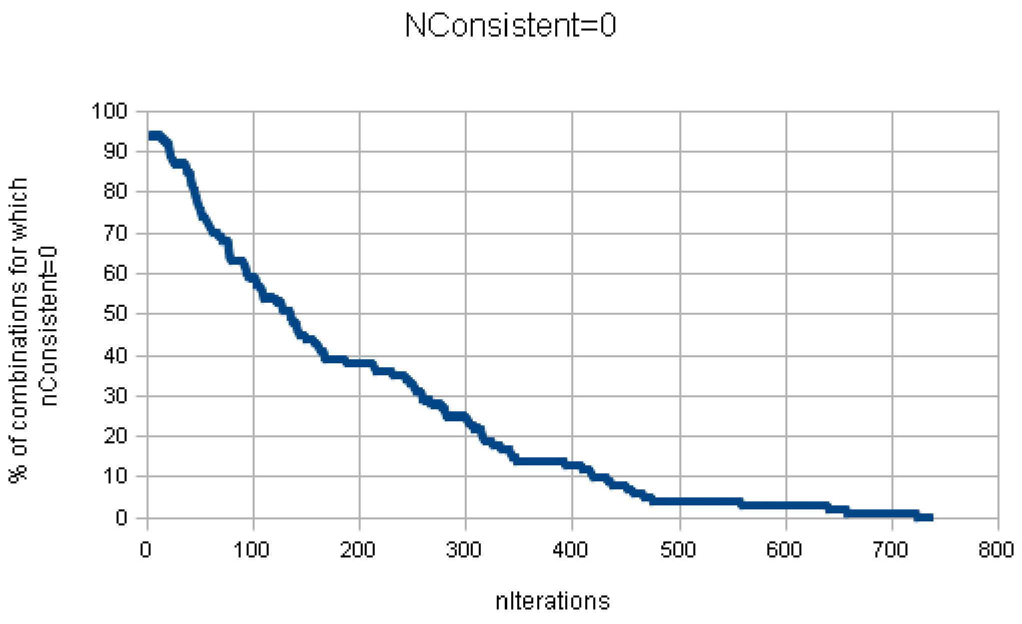

Figure 47: Case 1: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.10, nComb=100, delta=1.

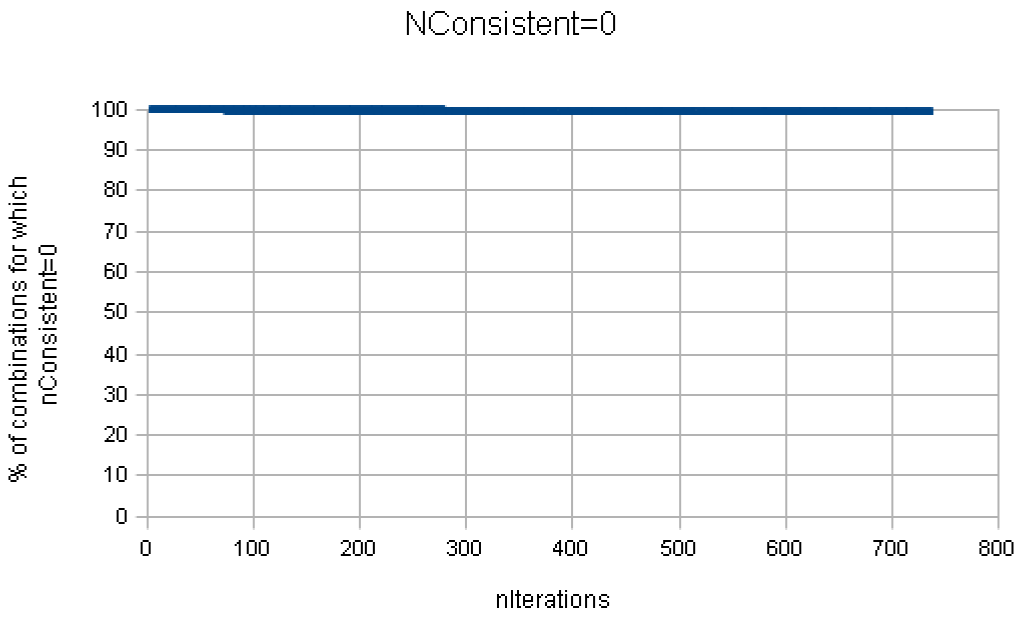

Figure 48: Case 2: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.10, nComb=100, delta=1.

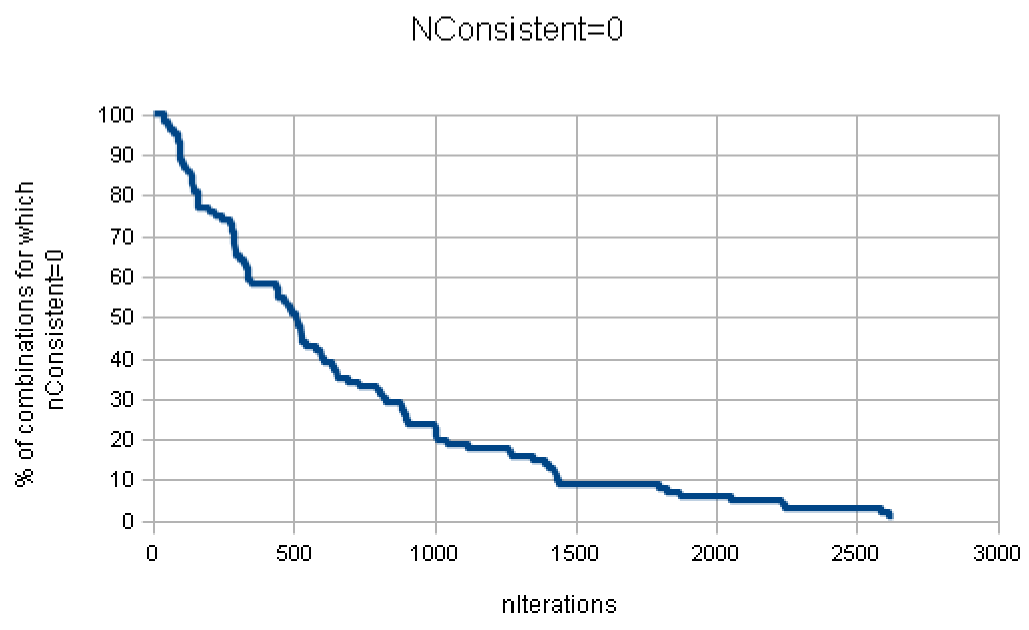

Figure 49: Case 1: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.15, nComb=100, delta=1.

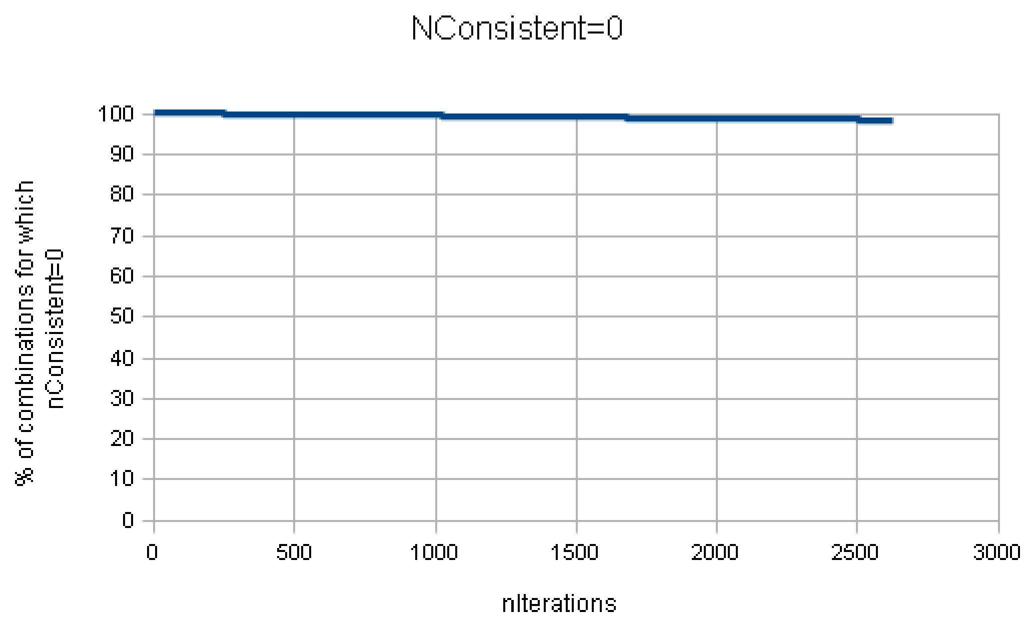

Figure 50: Case 2: $f_s = 1 + x^4 + x^7$, $f_u = 1 + x^6 + x^7$. Noise level=0.15, nComb=100, delta=1.

B.3 Length=12 registers

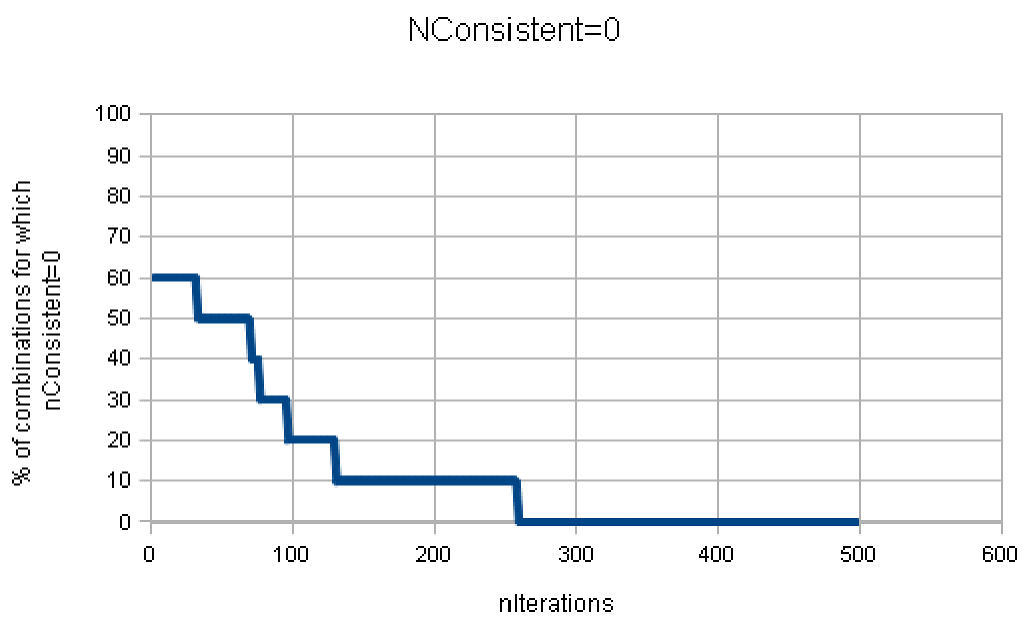

Figure 51: Case 1: $f_s = 1+x+x^4+x^6+x^{12}$, $f_u = 1+x^2+x^3+x^9+x^{12}$. Noise=0.05, nComb=10, delta=0.

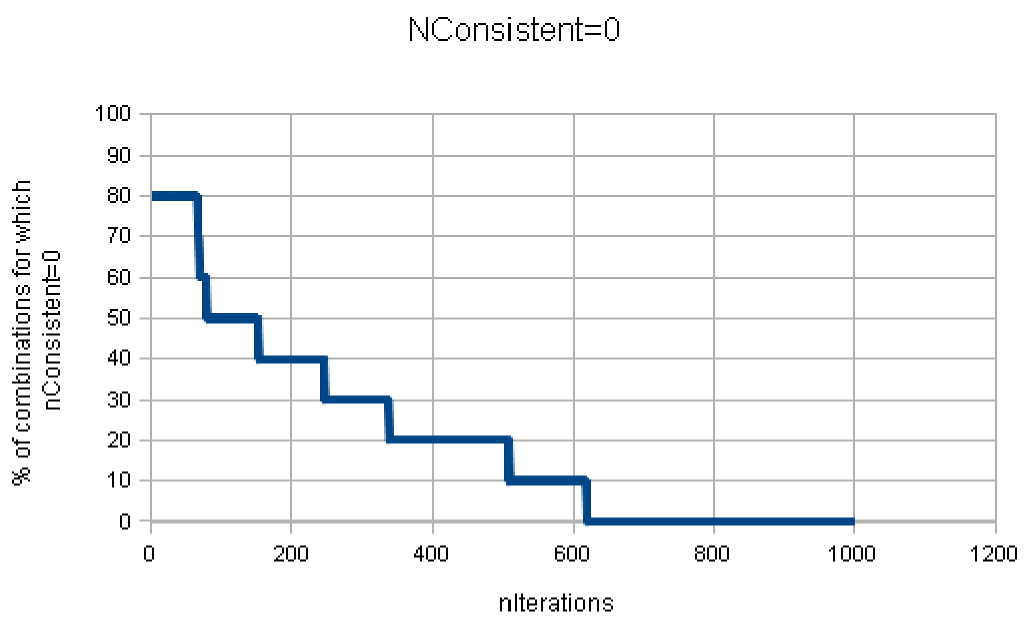

Figure 52: Case 1: $f_s = 1+x+x^4+x^6+x^{12}$, $f_u = 1+x^2+x^3+x^9+x^{12}$. Noise=0.10, nComb=10, delta=0.

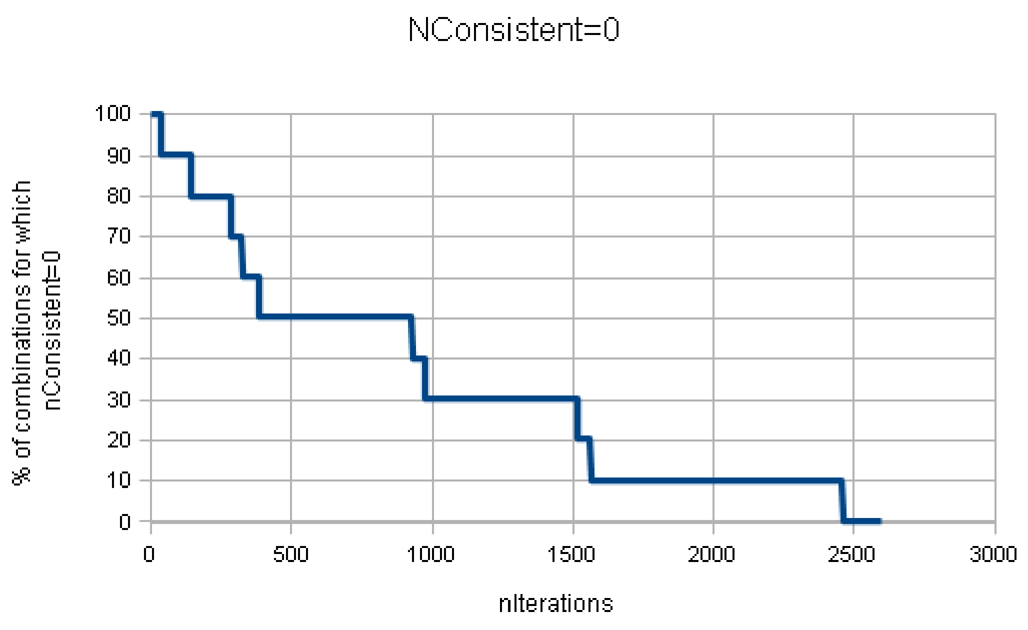

Figure 53: Case 1: $f_s = 1+x+x^4+x^6+x^{12}$, $f_u = 1+x^2+x^3+x^9+x^{12}$. Noise=0.15, nComb=10, delta=0.

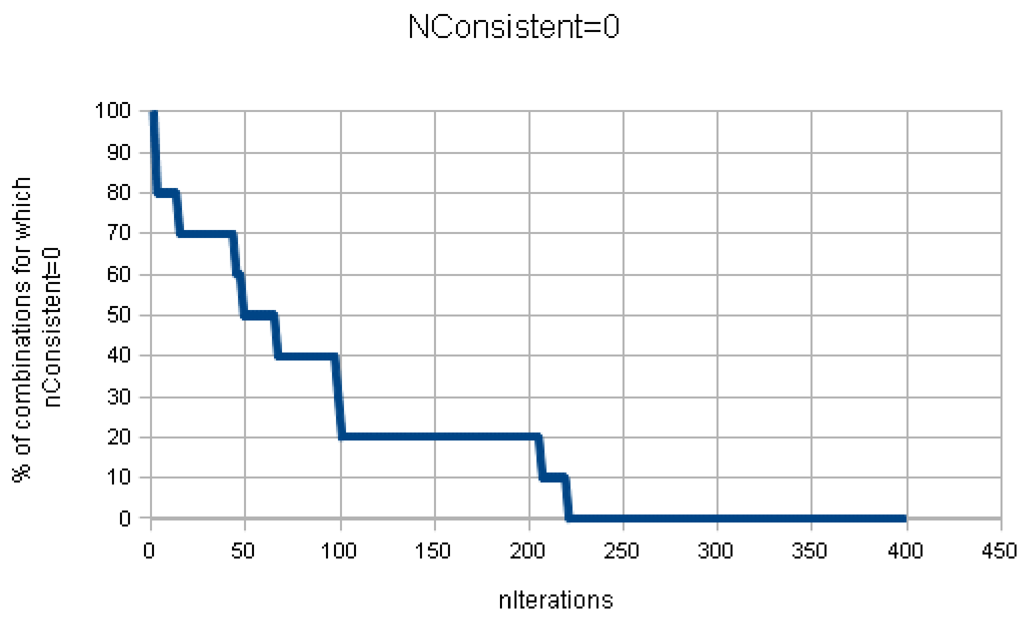

Figure 54: Case 1: $f_s = 1+x+x^4+x^6+x^{12}$, $f_u = 1+x^2+x^3+x^9+x^{12}$. Noise=0.05, nComb=10, delta=1.

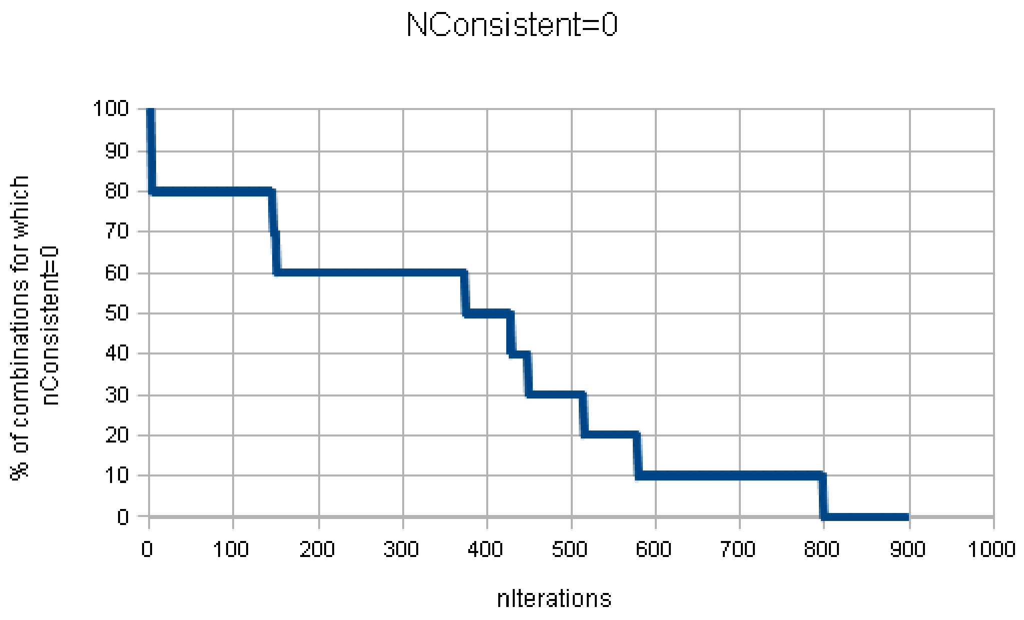

Figure 55: Case 1: $f_s = 1+x+x^4+x^6+x^{12}$, $f_u = 1+x^2+x^3+x^9+x^{12}$. Noise=0.10, nComb=10, delta=1.

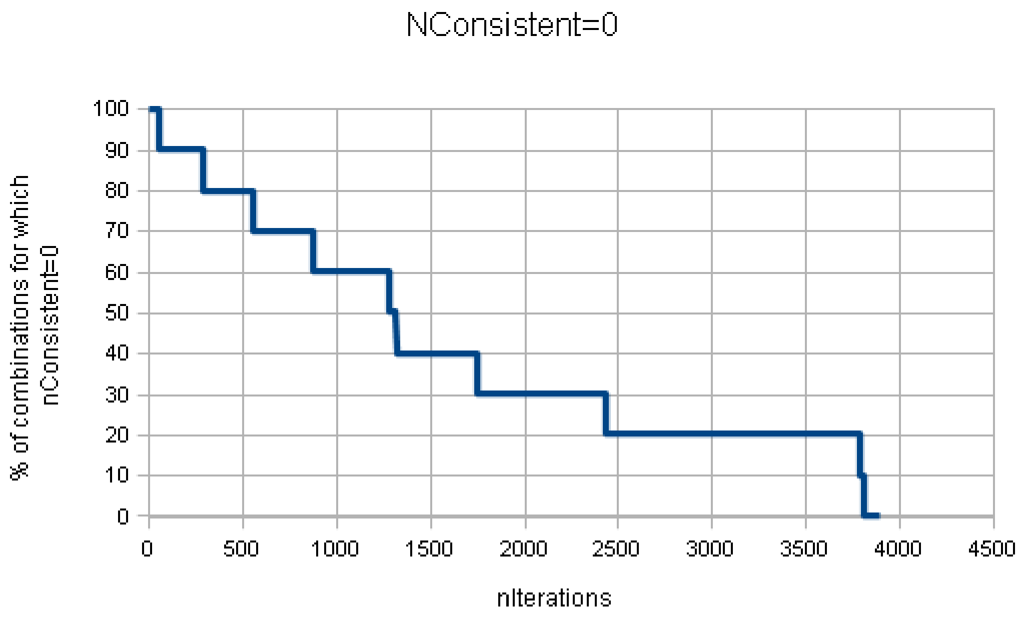

Figure 56: Case 1: $f_s = 1+x+x^4+x^6+x^{12}$, $f_u = 1+x^2+x^3+x^9+x^{12}$. Noise=0.15, nComb=10, delta=1.