Adrian Schjelderup Evensen

Reconstruction of clock sequences in irregularly clocked pseudorandom sequence generators

A bit-parallel approach to breaking the Binary Rate Multiplier

Master's thesis in Information Security
Supervisor: Prof. Slobodan Petrovic
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Kunnskap for en bedre verden

Adrian Schjelderup Evensen

Reconstruction of clock sequences in irregularly clocked pseudorandom sequence generators

A bit-parallel approach to breaking the Binary Rate Multiplier

Master's thesis in Information Security
Supervisor: Prof. Slobodan Petrovic
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Kunnskap for en bedre verden

Preface

This Master's thesis was written in MIS4900 as the final subject of *Master in Information Security (Part-time)* (MISD) at NTNU Gjøvik. Prof. Slobodan Petrovic proposed the topic and has supervised work on the thesis. The field of the study is cryptography and more specifically on *stream ciphers* based on non-linear combination of Linear Feedback Shift Registers (LFSR) by means of irregular clocking.

The topic of this thesis entails assessing the possibilities and effectiveness of reconstructing the clock-control sequence of the clocking LFSR in a *Binary rate multiplier* (BRM) based stream cipher system. There have been studies on applying the so-called *generalised correlation attack* [1] to find candidate sequences for the clocked LFSR by calculating the constrained Levenshtein distances between an intercepted ciphertext and the raw output of the clocked LFSR [2]. In [3] the first part of the attack was implemented both in software and in a Field Programmable Gate Array (FPGA). The software developed in this thesis builds on the existing software in an attempt to complete the full attack and recover the initial state of both the clocking, and the clocked, LFSR.

It is assumed that the reader is familiar with general computer architecture, basic computer programming and general mathematics including statistics and binary arithmetic. Any other pre-requisite aspects are thoroughly explained in Chapter 2.

Adrian Schjelderup Evensen
2021-06-01

Acknowledgements

Thank you to Prof. Slobodan Petrovic for excellent guidance and feedback during the work on this thesis. Also a big thank you to Hilde Bakke and other staff members at the Faculty of Information Technology and Electrical Engineering for great support throughout many years of studies at NTNU Gjøvik.

Finally, a very special thanks to my fantastic fiancé and two daughters for their unrelenting support throughout the process.

Adrian Schjelderup Evensen

Abstract

Stream ciphers implementing irregularly clocked Linear Feedback Shift Registers (LFSRs), whose outputs are combined in a non-linear boolean function, are popular since they produce output sequences with extremely long periods, high linear complexities and excellent statistics.

However, it is known that such schemes can be broken by means of the so-called generalised correlation attack [1]. Resistance to such an attack depends on the characteristics of the combining boolean function. In addition, if the clocked LFSR is relatively long (over 30 bits), it is difficult to execute the attack since it involves computation of the constrained edit distance for every tested initial state of the clocked LFSR and the time complexity of this calculation is quadratic in the length of the intercepted output sequence of the generator.

Instead of computing constrained edit distance, it is possible to find the best embedding of the intercepted ciphertext sequence of the generator in the output sequence of the attacked LFSR for every initial state of that LFSR. This is possible to achieve by performing a constrained approximate search for the intercepted sequence as shown in [2].

The next and final step in the cryptographic attack on such ciphers is the reconstruction of the clock control sequence and the initial state of the clocking LFSR. In the case of the classical, *dynamic programming*-based approach, this is done by backtracking through the matrix of partial constrained edit distances.

In this thesis, the possibilities of clock sequence reconstruction in the bit-parallel approximate search scenario is investigated and a novel attack combining the use of multi-threaded bit-parallel approximate search and a brute-force attack is developed and presented. Several experiments are also conducted in an attempt to find arbitrary optimal input parameters in the approximate search phase of the attack.

The idea that theorems follow from the postulates does not correspond to simple observation. If the Pythagorean theorem were found to not follow from the postulates, we would again search for a way to alter the postulates until it was true. Euclid's postulates came from the Pythagorean theorem, not the other way around.

RICHARD W. HAMMING

Contents

Figures

Chapter 1

Introduction

1.1 Background

Cryptographic systems implementing stream ciphers using irregularly clocked Linear Feedback Shift Registers (LFSR) whose outputs are combined in a non-linear boolean function are in widespread use as they produce output sequences with long periods, high linear complexities and excellent statistics. Despite their wide adoption it has been shown that such systems can be attacked through the *Generalised correlation attack* developed by Golic, et al.[1]. Resilience against this attack depends on the combining boolean function and the length of the clocked LFSR. For longer LFSRs (e.g. over 30 bits) it becomes difficult to execute the attack since it involves the computation of the *constrained edit distance* for every tested initial state of the clocked LFSR and the time complexity of this calculation is quadratic in the length of the intercepted output sequence of the generator.

It has been shown [2] that these attacks become more efficient by replacing the original constrained edit distance computation procedure with a bit-parallel constrained search in order to find the best embedding of the intercepted noisy output sequence of the generator in the output sequence of the controlled LFSR when it runs without clocking. This first step of the attack has been implemented in both 64bit CPUs and FPGA with good performance results for stream cipher systems based on the irregular clocking of an LFSR through the Binary Rate Multiplier (BRM) decimating function [3].

The next and final step in the cryptanalysis of these types of cryptographic systems is the recovery of the initial state of the *clocking* LFSR. In its simplest form it can be done by evaluating every initial state of the clocking LFSR against every *candidate* for the clocked, or decimated, LFSR chosen in the first step. This would result in a worst case scenario of

$$|C| \times (2^p - 2)$$

iterations where $|C|$ is the total number of candidates for the initial state of the clocked LFSR ($R2$) and p is the polynomial degree of the clocking LFSR ($R1$). The total time complexity of the attack depends on the decision rules for the selection

of the candidates in the candidate set, C, the algorithm for deducting possible initial states for the clocking LFSR, the efficiency of the program and the computational power available. In this thesis these topics are addressed and a novel software implementation using a multi-threaded brute-force approach for reconstructing the clocking LFSR (and thus the whole cryptosystem) is presented.

1.2 Related research

In their paper, Golic et al. [1] introduce the generalised correlation attack. The attack is a modification on Siegenthalers [4] ciphertext-only correlation attack on regularly clocked keystream-generators by using the Hamming-distance measure [5]. As the Hamming-distance metric is not applicable on irregularly clocked keystream-generators due to the difference in length in the intercepted ciphertext bitstream and the undecimated bitstream, Golic et al. proposed a similar attack by replacing the Hamming-distance with the constrained Levenshtein-distance (CLD) [6]. They present experimental results for the probability distributions $P(D|H_0)$ and $P(D|H_1)$, where H are the hypotheses:

H_0: The observed sequence of ciphertext is not produced by the chosen initial state.

H_1: The observed sequence of ciphertext is produced by the chosen initial state.

Petrovic [2] built on Golics research by exploring the possibility of utilising a *Shift-OR* bit-parallel search algorithm to compute the constrained Levenshtein distance in order to find candidates for the initial state of the clocked LFSR in an efficient manner.

In his thesis, Øverbø [3] developed a software, implementing an Approximate row-wise bit-parallel (ARBP) search algorithm, using the *shift-AND* mode of operation, for both 64bit CPUs and *Field programmable gate arrays* (FPGA) and researched the performance difference between the two. It was shown that the FPGA implementation performs much better (26-346 times faster), due to natively being able to perform bit-parallel operations on larger registers than a regular 64bit CPU. The implementations are written in the C-programming language and *Verilog*.

1.3 Scope

This thesis seeks to build on the previously discussed related research in order to complete the cryptanalysis of a stream cipher system utilising BRM for irregular clocking by developing and implementing an algorithm to recover the initial state of the clocking LFSR and thus breaking the whole cryptosystem. It is also within the scope of the thesis to quantitatively research the optimal decision rule for the hypotheses H_0 and H_1, resulting in the $R2$ initial state candidate set C, through the following parameters:

p: The polynomial degree of the LFSR register.
m: The length of the known plaintext.
n: The length of the intercepted ciphertext.
k: The error threshold for determining H_0 and H_1.

The work on this thesis will involve building on the CPU-based software already developed in [3]. First, the alternative *Shift-OR* approximate search algorithm with different degrees of constraints will be implemented and tested. Finally, these new algorithm implementations will be combined in an algorithm for testing each element of the chosen set of possible initial states for $R2$ against every possible initial state for the clocking LFSR using a *brute-force* approach in an efficient manner in CPU. Several experiments will also be done with multi-threading and utilisation of highly capable cloud-based virtual hosts.

Implementation of the finalised algorithm in FPGA or GPU (Graphics Processing Unit) is not within the scope of this thesis and may be a topic for future research. However, the computational performance in the CPU implementation of the algorithms will be improved upon, making it scalable and possibly applicable in practical attack implementations in the future.

1.4 Contributions

The research and development done in this thesis will contribute by first evaluating the previous research in order to see if the optimal parameters for generating the ideal set of candidates for $R2_{init}$ can be determined. The thesis will also contribute by attempting to further improve the efficiency of existing software by implementing a *Shift-OR* approach with different degrees of constraints, as proposed by Petrovic [2]. Another important contribution will be the implementation of multi-threading in the application, which was mentioned as a suggestion for further studies in [3]. In theory, this will significantly improve the efficiency of both the candidate selection for $R2_{init}$ and the subsequent procedure for recovering the clocking LFSRs initial state, $R1_{init}$. Ultimately, an algorithm for recovering the initial state of the clocking LFSR, $R1$, will be developed and presented.

1.5 Research questions

In this section the research questions for the thesis are presented. Throughout the thesis the questions will be referred to by their numeric denomination (e.g. **RQ#**).

RQ1 - Is it possible to define an arbitrary optimal decision rule for selecting the set of candidates for the initial state of $R2$?
RQ2 - Can a practical attack on irregularly clocked stream ciphers based on BRM be implemented and tested using brute-force in software?
RQ3 - Is it possible to recover the initial states of both the clocking ($R1$) and clocked ($R2$) LFSR in less than $(2^p - 2)^2$ brute-force iterations?

RQ4 - Is it possible to improve the performance of the overall algorithm by introducing orchestrated multi-threading using cloud based infrastructure?

1.6 Thesis outline

After this introductory chapter follows a thorough explanation of the underlying theory for irregularly clocked stream ciphers and the cryptoanalysis of such systems. It seeks also to explain the concepts of the search algorithms which have been used in previous research upon which this thesis builds. In Chapter 3 the methodologies employed throughout the project are presented. Chapter 4 explains the development process and final implementation of the algorithms along with the rationale behind how it was developed. In Chapter 5 the final product and the experimental results of the thesis are presented. It culminates with the demonstration of the full implementation of the attack on a BRM based cryptographic system in software. In the final two chapters the implementation and its results are discussed and a conclusion of the thesis along with suggestions for future research is presented.

Chapter 2

Theory

This chapter intends to supply the reader with the necessary pre-requisite knowledge for the rest of the thesis on the topics of stream ciphers, general cryptanalysis and search algorithms.

2.1 Stream ciphers

Since Shannons demonstration of the so-called *one-time pad* in 1949 [7], cryptographic researchers have been implementing more practical and usable *stream ciphers* to secure various communications channels. These ciphers, in contrast to the more modern and widespread *block ciphers* only encipher one symbol at a time (most often bits in modern communications) whereas block ciphers encipher multiple symbols at a time, in blocks. Stream ciphers are, however, well suited for implementations in hardware where fast communications are needed. As only one character is enciphered at a time, it is also more resilient against communication errors than block ciphers, where a single symbol transmission error may render a whole block of ciphertext unreadable. For the remainder of this thesis we will refer to *symbols* as *bits*, as we are only working with digital implementations of stream ciphers.

In general, a stream cipher system can be described with with following simple formula:

$$X_i \oplus Y_i = Z_i$$

Where X_i represents the i-th bit of the plaintext, Y_i represents the i-th bit of the key stream and Z_i the resulting i-th bit of the output ciphertext. The plaintext and key stream bits are combined with the **XOR** (bitwise addition, modulo 2) boolean operation in order to produce the corresponding ciphertext bit.

Some famous implementations of stream ciphers include the *Vernam cipher* and the more secure specification of it, the *one-time pad*. The one-time pad is a special implementation of the Vernam cipher which was proven to be unconditionally secure by Claude E. Shannon [7] if and only if the following criteria are met:

1. The keystream must be produced with perfect randomness.
2. The keystream must be of equal or greater length than the plaintext.
3. The key material, as a whole or in part, must only be used once.
4. The key material must be kept completely secret between the transmitting and receiving party.

As these criteria are very hard to fulfill in practice, such cryptographic systems are in limited use, however some government agencies are said to have used the one-time pad for their most secure communication channels, e.g. the "red telephone" used between the state leaders of the United States of America and the Russian Federation [8].

For modern widespread communication channels however, the one-time pad is infeasible to use as it depends heavily on perfect key material generation and management. Thus, a more practical approach to keystream generation is needed. This can be achieved by using Linear Feedback Shift Registers (LFSR) to generate pseudorandom key streams which will be discussed in depth in the next section.

Modern implementations of stream ciphers are used in Bluetooths E0-algorithm and in GSMs A5/1-algorithm, as shown in Figure 2.1.

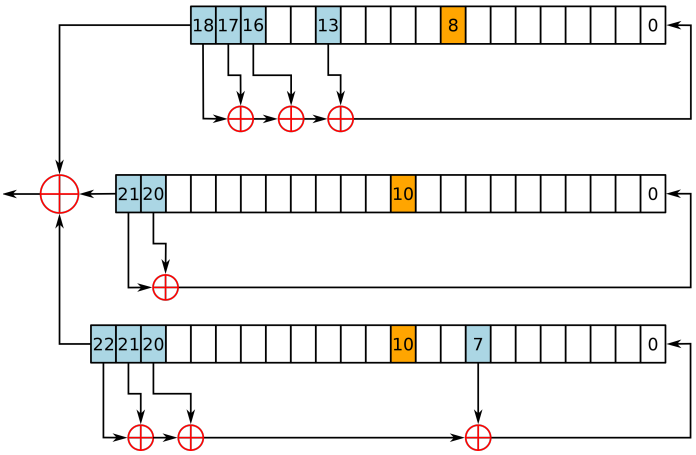

Figure 2.1: A5/1 key generator for GSM [9].

While stream ciphers can be highly effective when securing communications, if they are not properly implemented they can be highly predictable and thus breakable unless non-linearity is introduced. The details and importance of this will be discussed in detail in the next few sections of this thesis.

2.2 Linear feedback shift registers

As discussed earlier, a perfectly random key stream is hard to achieve and impractical to use in practical implementations of stream ciphers. Perfect randomness is especially hard to achieve in high-throughput systems as it requires perfectly random input parameters of the same bandwidth as the key stream output. Therefore,

in most practical implementations of cryptographic systems, *pseudorandom* noise generators are used. Such generators aim to provide a noise sequence which can be used as a key stream which is sufficiently random enough to be secure. In order to determine if a pseudorandom noise-sequence is sufficiently random we can check it against Golomb's three randomness postulates [10]:

Given s, a periodic sequence of period N;

RP1 In the cycle s^N of s, the numbers of 1's differ from the number of 0's by at most 1.

RP2 In the cycle s^N, at least half the runs have length 1, at least one-fourth have length 2, at least one-eight have length 3, etc., as long as the number of runs so indicated exceeds 1. Moreover, for each of these lengths, there are (almost) equally many gaps and blocks.

RP3 The auto correlation function $C(t)$ is two-valued. That is for some integer K,

$$N \times C(t) = \sum (2s_i - 1) \times (2s_{i+t} - 1) = \begin{cases} N, \text{ if } t = 0 \\ K, \text{ if } 1 \leq t \leq N-1 \end{cases}$$

The *Feedback shift register*, and more specifically the *Linear Feedback Shift Register* (LFSR) is the main building block of most modern stream cipher systems as it can produce pseudorandom noise sequences of large periods with a sufficient degree of entropy and with good statistical properties. LFSRs are also well-suited for hardware implementations and because of their structure, they can be readily analysed using algebraic functions [11].

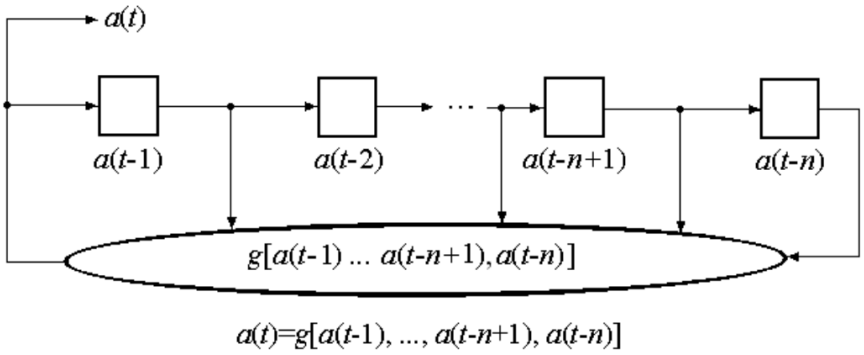

Figure 2.2: Illustration of an LFSR

The LFSR illustrated in Figure 2.2 consist of *stages*,

$$c_i, i \in 0, 1, ...L-1$$

from left to right, where L is the length of the register. The length of the register is the amount of bits the register can store. Each stage in the register holds a binary

value, $c_i \in \{0,1\}$, at any given time during execution and the initial state of these bits represent the input key to the cryptosystem. The zero-initial state $C_{0\ldots p-1} = 0$ however, can not be used as it would result in an all 0 output sequence, thus there are $2^p - 2$ valid states of the LFSR.

Each position in the register has an input, and can also have an output, or *tap*, which feeds into the linear combining function g. The system also has a clock which *steps* the register. During each step of the generator the following operations are performed:

1. The value of stage 0 is output and forms part of the output sequence.
2. A *shift* is performed on the whole register which results in the content of each stage i being moved to stage $i+1$.
3. The value of stage $n-1$ is fed back and combined with the other *taps* to form the next output and feedback value to stage 0.

It is easy to see that the maximum amount of bits the LFSR can produce before repeating itself is $N = (2^p - 2)$. In order to ensure that we achieve the maximum output length that also satisfies the requirements for a usable pseudorandom sequence, we need to ensure that the LFSRs *feedback polynomial* $C(D)$ is *primitive*. We can describe an LFSR as;

$$\langle p, C(D) \rangle \text{ where } C(D) \in \mathbb{Z}_2[D] \text{ is the feedback polynomial.}$$

In the definition above, $\mathbb{Z}_2[D]$ represents the elements of the binary extension field GF(2^p). The polynomial must satisfy the non-singularity condition, which implies that the degree of the polynomial $C(D)$ must be p, e.g.;

$$C(D) = 1 + c_1 D + c_2 D^2 + \ldots + c_p D^p$$

Furthermore, the polynomial $C(D) \in \mathbb{Z}_2[D]$ of degree p is *primitive* if and only if D is a generator of $\mathbb{F}^*_{\mathbb{D}^1}$, where $\mathbb{F}^*_{\mathbb{D}^1}$ is the multiplicative group of all the non-zero elements in $\mathbb{F}^*_{\mathbb{D}^1} = \mathbb{Z}_2[D]/(C(D))$. In order to test whether an irreducible polynomial is primitive we can use the following algorithm [11]:

Algorithm 1: Testing whether a polynomial is primitive

input : a prime p, a positive integer m, the distinct prime factors $r_1, r_2, \ldots, r_t$ of p^{m-1}, and a monic irreducible polynomial $f(x)$ of degree m in $\mathbb{Z}_p[x]$.

output: The answer to the question: "Is $f(x)$ a primitive polynomial?"

for *i=1; i<=t;* **do**
  Compute $l(x) = x^{(p^m-1)/r_i} mod f(x)$;
  if $l(x) = 1$ **then**
    return("Not primitive");
  else
    return("Primitive");
  end
end

Once a primitive polynomial has been found, a usable cryptosystem can be constructed. However, the output bitstream of a single LFSR $\langle p, C(D)\rangle$ is linear and predictable and thus weak against algebraic and statistical attacks which will be discussed in more depth in Section 2.4 of this chapter.

Non-linearity has to be introduced into the cryptosystem in order for it to be more resilient against such attacks. This can be achieved by combining the output of two or more LFSRs in a nonlinear combination generator, employing irregular clocking, or both. An example of a nonlinear combination generator is the A5/1 algorithm in GSM, as illustrated in Figure 2.1 which combines three separately clocked LFSRs of different lengths. A generalised model of such generators are given in Figure 2.3. In this thesis we will examine stream cipher systems employing *irregular clocking* and more specifically the *Binary rate multiplier* variant of such cryptosystems.

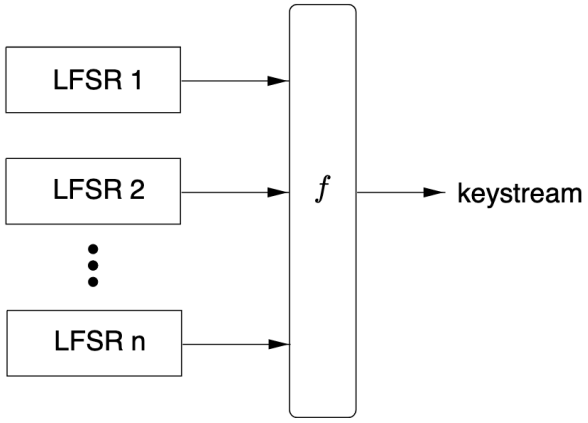

Figure 2.3: Generalised illustration of nonlinear combination of multiple LFSR [11].

2.3 Irregularly clocked stream ciphers

Earlier we saw that we can define an LFSR as $\langle p, C(D)\rangle$, where p represents the length and also the degree of the primitive polynomial $C(D)$. In addition to this, in order to make the system functional we need a *clocking function* to *step* the generator. A single LFSR implementation is normally restricted to the clock of the internal system clock or cycles in a *Central Processing Unit* (CPU), which make them predictable. In order to introduce non-linearity and to make such systems less predictable *irregular clocking* can be employed. Irregular clocking aims to add uncertainty, or entropy, to when the LFSR is clocked and thus the resulting output key stream. In GSMs A5/1 key stream generation algorithm, as shown in Figure 2.1 for example, the clocking of the three registers are dictated by the state of the 8th, 10th and 10th bit, respectively in each LFSR. The A5/1 implementation

uses a *majority bit* function to determine which registers are clocked. For every cycle the three bits are checked, and only the two registers with the same bit-value are clocked. Thus, for every cycle only two of the three registers are clocked and there is a uniform probability of each register clocking of 3/4. Another approach to irregular clocking is to have one LFSR clock another LFSR, e.g. the output of the first LFSR dictates the operation of the next. Examples of such generators are the *Alternating step generator*, the *Shrinking generator* and the *Binary rate multiplier*, the latter of which will be the main focus of this thesis.

2.3.1 Binary rate multiplier

As discussed earlier, non-linearity has to be introduced into the key stream generator in order for a stream cipher system to be secure. The *Binary rate multiplier* (BRM) accomplishes this by introducing the concept of *decimation*. Decimation is defined in the Cambridge dictionary [12] as

> "*... the act of killing something in large numbers, or reducing something severely.*"

Luckily perhaps, the second part of the definition is what will be discussed here.

The binary rate multiplier, or 0/1-clocking as it can also be called, consists of two separate LFSRs; the *clocking* LFSR and the *clocked* or *decimated* LFSR, henceforth referred to as $R1$ and $R2$ respectively, throughout this thesis. The LFSRs themselves work in the conventional way described in the previous section, however, $R1$s purpose is only that of deciding whether $R2$ is clocked or not. The two LFSR are initialised with their own separate initial states $R1_{init}$ and $R2_{init}$. In general terms the generator can be described with the model shown in Figure 2.4. In the figure, X_n represents the bit stream generated by $R2$ and Y_n the bit stream generated by $R1$. The resulting key stream, Z_n is the result of the decimating function $f(n)$.

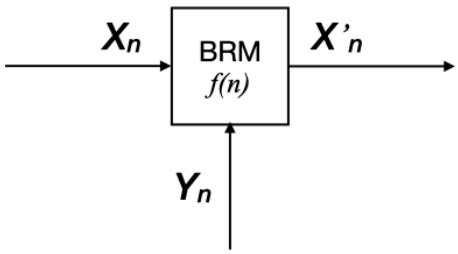

Figure 2.4: Generalised model of the BRM key stream generator.

The operation of the BRM generator can be explained as follows:

1. $R1$ is clocked once.
2. If the output of $R1$ is 0, then $R2$ is also clocked once and its output bit is forwarded as part of the key stream. If the output of $R1$ is 1, $R2$ is clocked twice where the first of the generated bits is discarded (or decimated) and the second forwarded onto the key stream.

The following equations can also be used to illustrate the BRM generators operation:

$$z_n = x_{f(n)}, n = 0, 1, 2, ...$$

$$f(n) = n + \sum_{i=0}^{n} y_i$$

In their theorem, Chambers et al. [13] showed that under the following assumptions;

- $R1$ has a primitive feedback polynomial of degree m, with a period of $M = 2^m - 1$
- $R2$ has a primitive feedback polynomial of degree n, with a period of $N = 2^n - 1$
- All prime factors of M divide N
- The greatest common divisor $(\sum_{i=0}^{M-1} x_i, N) = 1$

Then the linear complexity, $\mathcal{L}$, is $\mathcal{L} = nM$ and the period, $\mathcal{P}$, $\mathcal{P} = NM$. In the case where both $R1$ and $R2$ are of equal length, the requirements of the theorem is satisfied. In this thesis it is always assumed that $R1$ and $R2$ are of equal length when addressing the BRM.

The statistical model of the BRM in a stream cipher system is shown in Figure 2.5 and its operation is described in more detail in Algorithm 2.

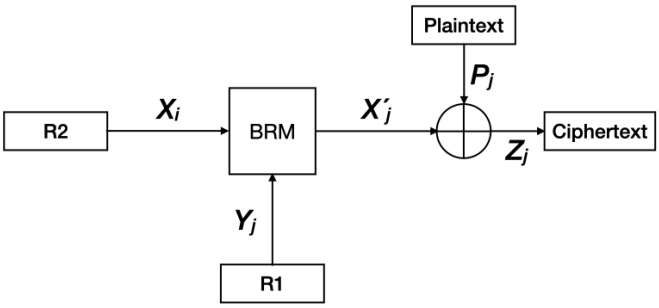

Figure 2.5: BRM as a key stream generator in a stream cipher system.

As we can see from Algorithm 2, given a plaintext of length m, $R1$ will produce m bits of output, while $R2$ should produce approximately $n = 2 \times m$ bits, given that $R1$ satisfies Golomb's first postulate (**RP1**) which dictates that in a generated pseudorandom bit sequence the number of 1's should differ from the number of

Algorithm 2: A stream cipher system based on BRM

input : a bit sequence from $R2$; $X_{i=0\ldots n-1}$, $R1$; $Y_{j=0\ldots m-1}$ and the plaintext $P_{j=0\ldots m-1}$
output: the ciphertext $Z_{j=0\ldots m-1}$
for $(j < m)$ **do**
  Step R1 $\rightarrow Y_j$
  if $Y_j = 0$ **then**
    Step R2 $\rightarrow X_{i+1}$
    $X_{i+1} \rightarrow X'_j$
  else
    Step R2 $\rightarrow X_{i+1}$
    Step R2 $\rightarrow X_{i+2}$
    $X_{i+2} \rightarrow X'_j$
  end
  $X'_j \oplus P_j \rightarrow Z_j$
  j++
end
return$(Z_{j=0\ldots m-1})$

0's by at most 1 bit. In some implementations of the BRM generator, the decimated bits may be forgotten by the machine or program running it, however in the case of the implementations described in this thesis a record of these bits is kept in order to be able to mount the attack which will be covered later. Specifically, in the implementations described in Chapter 4 both sequences $X_{i=0\ldots n-1}$ for $R2$ and $Y_{j=0\ldots m-1}$ for $R1$ are fully generated before being combined in the BRM function to produce the key stream and encrypt the plaintext.

A simple example for $m = 5$ and the zero plaintext $P = 0^{m-1}$ is given below;

$$
\begin{aligned}
P_{j=0\ldots4} &= 0,0,0,0,0 \\
X_{i=0\ldots8} &= \underline{1},1,\underline{0},\underline{1},\underline{0},1,\underline{0},1,1 \\
Y_{j=0\ldots4} &= 0,1,0,0,1 \\
X'_{j=0\ldots4} &= 1,0,1,0,0 \\
Z_{j=0\ldots4} &= 1,0,1,0,0
\end{aligned}
$$

The underlined digits of X are the ones being used as the key stream X'. As the input plaintext consists of only zeroes, the resulting ciphertext $Z = X'$.

2.4 Cryptanalysis of stream ciphers

In this section, different aspects of cryptanalysis and cryptographic attacks are explained. The main focus will be on the statistical attack and more specifically

the *generalised correlation attack*, however some other variations are also briefly discussed for comparison.

2.4.1 Cryptanalysis fundamentals

Cryptanalysis is the art of analysing cryptographic implementations and algorithms from different points of view. Sometimes the cryptanalyst assumes "*a priori*" knowledge, which means knowledge that the analyst assumes about the system before applying the attack. The fundamental problem of the cryptanalyst is to recover the plaintext and/or determine the key which can be used to decipher said plaintext. We can classify the types of analysis approaches under the following categories:

- Ciphertext-only attack
- Known plaintext attack
- Chosen plaintext attack
- Chosen ciphertext attack

We assume in most cases that the analyst as a minimum has full knowledge of the design of the system being analysed, and that he or she has access to, or the ability to capture, ciphertext produced by the system. The most difficult approach is where the analyst only has knowledge of the ciphertext and the task of breaking the system becomes significantly easier with more knowledge. The attack proposed in this thesis is a *known plaintext attack*, as we assume *a priori* knowledge of the plaintext, or parts of it, the ciphertext and the design of the cryptosystem. In the following sections we will briefly examine some well known general analysis strategies and attacks on cryptographic systems.

2.4.2 Brute force attacks

The so-called *brute force* attack assumes only knowledge of the cryptosystem and some of its corresponding ciphertext. This is the simplest form of cryptographic attack. The attack is based on the trail-and-error method of trying every possible key in the key space, $\mathcal{K}$, of the system and see if it deciphers the intercepted ciphertext into something meaningful. The parameters affecting the feasibility of such an approach is the key-length, the computational efficiency of the crypto algorithm and the computational power available to the attacker.

Any modern cryptographic implementation should never be susceptible to this type of attack, but some older legacy algorithms that were made with smaller key sizes and made for less efficient computers can still be vulnerable. In some cases, as with the approach proposed in this thesis, the brute force attack can form parts of the overall attack. In this thesis for example, the total key space is first reduced through a statistical attack, before the brute force attack is initiated. Consider for example a BRM-based stream cipher system with equal length LFSRs. In general, the worst case key space to test in a brute force approach of a generator with

non-linear combination of multiple LFSRs can be described by the formula

$$N_b = \prod_{i=1}^{N} (2^{p_i} - 2)$$

where N is the amount of combined LFSR of the degree p_i. Thus in the case of BRM the worst case key space to test without any prior reduction (and not counting the zero-state) would be:

$$\mathcal{K} = (2^p - 2)^2$$

With a reduction of the possible key space of one of the LFSR, the total key space can be drastically reduced, as will be demonstrated later in this thesis.

Table 2.1 shows the amount of possible keys in a BRM key stream generator when the two LFSR are of equal length p and no reduction has been made to either LFSRs key space.

Table 2.1: Worst case key space in BRM with equal length LFSR

p	$\mathcal{L}$
11	4190209
16	4294836225
20	1.1×10^{12}
30	1.15×10^{18}
40	1.21×10^{24}
64	3.40×10^{38}
128	1.16×10^{77}

For comparison, the amount of atoms on Earth is estimated to be 1.33×10^{50} and the age of the sun 1.45×10^{45} seconds.

2.4.3 Algebraic attacks

Algebraic attacks aim to to *solve* the cryptosystem through a system of equations relating to the details of the system being attacked. In the case of a single LFSR this is trivial;

Consider the LFSR shown in Figure 2.6. It has the feedback polynomial $f(x) = 1 + x + x^4$ and output bits $y_i = 0...n$. The goal of the cryptanalyst is to find the key, e.g. the initial state of the register. We denote the initial state as the state of each stage, $a_s = \{a_0, a_1, a_2, a_3\}$, before the first clock cycle. Given that we have knowledge of the systems design and the output bit sequence, we can use the set of equations relating to what we know and solve for a_s.

$$y_0 = a_3 + a_0$$
$$y_1 = y_0 + a_1$$
$$y_2 = y_1 + a_2$$
$$y_3 = y_2 + a_3$$

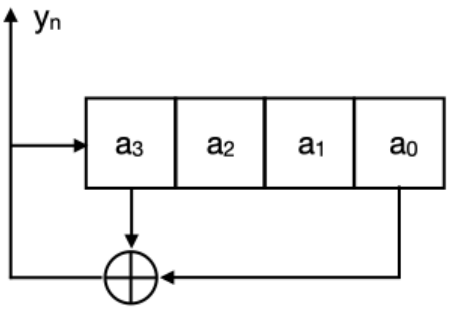

Figure 2.6: Simple LFSR with feedback polynomial $f(x) = 1 + x + x^4$

In the case of a non-linear combination of multiple LFSR however, the algebraic solution to the system becomes much more complex and in most cases becomes infeasible to solve. As this type of approach is not directly applicable to the further cryptanalysis in this thesis, we will not address this issue any further.

2.4.4 Statistical attacks

A statistical attack exploit potential weaknesses in the statistical characteristics of the output sequence of the pseudorandom generator. This may result in a complete decipherment of the underlying plaintext or at least a reduction of the effective key space that has to be tried, e.g. in a brute force approach. As a basic principle, the output of the boolean combining function must be *balanced* in the sense that the amount of 0's and 1's are approximately equal, adhering to Golomb's first postulate as described in Section 2.2, to avoid being breakable by statistical analysis. As we will see however, even systems following this principle may be susceptible to a special form of statistical attacks. Especially relevant to the cryptanalysis of Binary rate multiplier based systems, are correlation attacks, and more specifically the generalised correlation attack, both of which we will examine in the sections below.

Correlation attack

Even though a boolean combining operation may provide non-linearity, its output may be susceptible to statistical attacks, as is the case with the *Geffe generator*[14]. A powerful type of statistical attack is the *correlation attack*. This attack exploits situations where the output bit sequence of a pseudorandom generator correlates with one or more internal states. The difference, or *distance*, between two equal length bit sequences can be measured by the *Hamming distance* metric [5]. The Hamming distance is a measure of how many *errors*, hereby denoted k, there are between two sequences, as shown in the example below.

$$k = 2 \begin{cases} 0101\underline{1}1\underline{0}1 \\ 0101\underline{0}1\underline{1}1 \end{cases}$$

The expected distance of two random uncorrelated sequences of length n would be $k \approx \frac{n}{2}$. Thus, a significant deviation from this indicates that the two sequences are in some way *correlated*. An example of this can be made of the *Geffe generator* which consist of the non-linear combination function

$$F(x_1, x_2, x_3) = x_1x_2 \oplus (1 + x_2)x_3 = x_3 \oplus x_1x_2 \oplus x_2x_3$$

It possesses good statistical properties, and with long enough register lengths it is infeasible to obtain the key through a brute force approach. The output sequence of the combiner, F, itself is also balanced. However, with knowledge of the system and by examining the truth table of each component LFSR individually together with the output bit of the combiner, we can see that there is a *correlation bias*.

$$P(y = x_1) = \frac{3}{4}$$
$$P(y = x_2) = \frac{1}{2}$$
$$P(y = x_3) = \frac{3}{4}$$

x_1	x_2	x_3	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 2.7: Truth table of the Geffe generator correlated with its internal components.

As Figure 2.7 shows, the output bit y can be statistically correlated to the internal outputs $\{x_1, x_2, x_3\}$ of the three registers. Crucially, this knowledge can be used to make educated guesses as to what the internal states of the registers were when that bit was produced. Without exploiting this flaw in the Geffe system the number of possibilities to try in a brute force attack would be $N_b = \prod_{i=1}^{3}(2^{p_i}-1)$. However, with the advantage gained by the discovering the correlation we can reduce the number to $N_c = \sum_{i=1}^{3}(2^{p_i} - 1)$ possible keys.

In general terms, the measure of correlation, α, describes the similarity of the intercepted ciphertext sequence C_n of length L, the internal LFSR output sequence X_n^j, where j is the LFSR number and n is the bit number and the output of the combining function Z_n. Two *hypotheses* are defined, where

H_0 The intercepted sequence **cannot** be generated by the tested initial state.
H_1 The intercepted sequence **can** be generated by the tested initial state.

together with their corresponding mean and variance parameters (μ, σ^2). We also need to define a decision threshold, $\mathcal{T}$. The threshold is used to decide whether to accept a candidate initial state to the set of states believed to be generated by the tested initial state, according to the hypothesis H_1. If the measured correlation is larger than the threshold ($\alpha > \mathcal{T}$) then we accept H_1, if not, we assume H_0.

Depending on the chosen threshold value, the length of the intercepted sequence and the known plaintext there is a risk of both *false positives* and *false negatives*. It is highly likely for example that a set of candidates, $\mathcal{A}$, will contain a number of false positives in the form of candidates accepted to H_1 that is not the actual initial state we are seeking. A bigger problem however, would be that the true initial state be rejected by assuming H_0, possibly leaving the attacker spending lengths of time testing a candidate set which will never provide the actual initial state and thus the key.

From a independently and identical distributed binary sequence, we expect that $P(Z_n = 0) = P(Z_n = 1) = 0.5$. The correlation probability of the combining function F can be described as $P(Z_n = X_n^j) = q_j$. The correlation measure, or *Hamming distance*[5], is described with

$$\alpha = L - 2\sum_{n=1}^{L}(C_n \oplus X_n^j), j \in \{0, ..., N\}$$

The plaintext source P_n that is combined with the output of the generator to form the ciphertext $Z_n \oplus P_n = C_n$ is not random, so $P(Y_n = 0) = P(Y_n = 1) = p_0 \neq 0.5$. According Siegenthaler [4];

$$P(C_n \oplus X_n^j = 0) = p_e = 1 - (p_0 + q_j) + 2p_0q_j$$

The distributions $P(\alpha|H_1)$ and $P(\alpha|H_0)$ are binomial and for a large enough L they are assumed to be *Gaussian*. The parameters of the probability distributions of the two hypotheses we defined earlier are

$$P(\alpha|H_1)\begin{cases}\mu_\alpha = L(2p_e - 1)\\ \sigma_\alpha^2 = 4Lp_e(1-p_e)\end{cases}, P(\alpha|H_0)\begin{cases}\mu_\alpha = 0\\ \sigma_\alpha^2 = L\end{cases}$$

If p_e or q_j is 0.5 then the distributions are identical and the system has no bias and cannot be broken through the correlation attack.

In order to produce the minimal possible set of candidates $\mathcal{A}$ that also holds the actual initial state of the system the decision threshold, $\mathcal{T}$ must be set in a proper way. Ideally the resulting candidate set has no *false negative*, p_m and as few *false positives*, p_f as possible.

$$p_f = P(\alpha \geq \mathcal{T}|H_0)$$
$$p_m = P(\alpha < \mathcal{T}|H_1)$$

The full correlation attack can be described through the following algorithm:

- Determine the probabilities q_j for each internal LFSR.
- Choose a LFSR with high q_j for the attack.
- For each initial state of LFSR X^j the output sequence, $X^j_{n=0...L}$ is generated.
- The correlation measure is calculated.
    - If $\alpha \geq \mathcal{T}$ we assume that this may be a valid initial state candidate according to H_1 and add it to the set $\mathcal{A}$.

- If $\alpha < \mathcal{T}$ we assume that this is not a valid initial state and we discard it.

After iterating through all initial states, if the threshold has been set properly, we should be left with a set of possible candidates which is small compared to the total number of initial states of $X^j = (2^m - 1)$. The mean values of the distributions $P(\alpha|H_1), P(\alpha|H_0)$ should in that case be well separated.

The generalised correlation attack

When irregular clocking is introduced in a cryptosystem, the previously discussed *correlation attack* cannot be directly applied, as the Hamming distance cannot be applied to sequences of different length. Consider the case of the Binary rate multiplier, where the output of $R2$ is decimated by the output of $R1$; as $R1$ *decimates* the output of $R2$, the output sequence of $R2$ $X^2_{n=0\ldots L}$ differs in length from that of the generated key stream $Z_{m=0\ldots M}$ and resulting ciphertext $C_{m=0\ldots M}$. Thus a new method of attack is needed.

One such attack is the *Generalised correlation attack*, as demonstrated by Golic, et al. [1]. The generalised correlation attack is a special kind of statistical attack which is based on the *Levenshtein distance*[6] in place of the previously demonstrated Hamming distance. The modified attack involves measuring the *constrained edit distance* between the output of the decimated LFSR $R2$ $X^2_{n=0\ldots L}$ against the output of the clocking function $Z_{m=0\ldots M}$ where $M \approx \frac{L}{2}$. The constrained edit distance is the measure of how many edit operations, including *deletions*, *insertions* and *substitutions* are needed in order to transform one bit sequence into another. *Constraints* can be set for the allowed transformations, e.g. a maximum run of consecutive *deletions* that are allowed. The constraints are useful for example in the case of the Binary rate multiplier, where we know that no more than 1 bit can be decimated at a time.

In contrast to the traditional correlation attack, the parameters of the probability distributions $P(\alpha|H_1), P(\alpha|H_0)$ cannot be estimated in the same way as demonstrated for the normal correlation attack. In order to get a significant separation of distributions, we need to choose a threshold value based on experimentation. Choosing this threshold correctly can have a significant impact in the total time complexity of breaking the cryptosystem. This aspect is one that will be explored further through the research and experiments done in this thesis.

2.5 Approximate string matching

String matching, or search, is an important and well researched topic within computer science. The general problem is simple; find one or more occurrences of the string S in text T. As we identified when discussing the generalised correlation attack, it involves computing the edit distance between two sequences. This is a directly relatable to the problem of approximate string matching. Approximate

string matching is defined as the task of finding the *pattern* p in a text T when a limited number k errors, or differences, are allowed between the pattern and its occurrences in the search text [15]. By using the previously discussed *constrained Levenshtein distance* (CLD), $ed(x, y)$, between two strings x, y a measure of the correlation between the two can be calculated.

This is demonstrated in the following example;

$$p = annual$$
$$T = annealing$$
$$ed(p, T) = 4$$

We can see that, in this case we start with three matches, then an error which requires the *substitution* of the letter u. After the substitution two more matches are found for the letters "al" and finally the suffix "ing" is read, which is not in the pattern at all, and is therefore *deleted*. This results in one *substitution* and three *deletions* in total.

By setting an acceptable error level k the problem of approximate matching can be defined as the task of finding all occurrences in T of every p' that satisfies $ed(p, p') \leq k, 0 < k < m$.

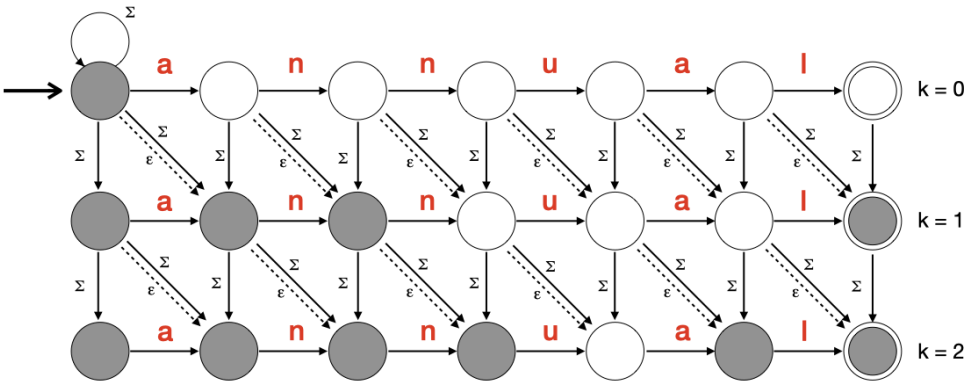

Figure 2.8: The approximate string matching of "annual" against "anneal", represented by an NFA[16].

The problem of approximate search can be represented as *Nondeterministic finite automata* (NFA), as shown by [17]. In Figure 2.8 we see the pattern matching of "annual" against "anneal" with $k = 2$ allowed errors, represented as an NFA. In the model, each row represents errors, horizontal paths represent exact matches and vertical paths insertions. The solid and dotted diagonal paths represent substitutions and deletions, respectively. We call the deletions ϵ*-transitions* (or empty transitions) as they differ from the others in that they only advance in the pattern, and not the text [18]. The initial self loop allows an occurrence to start anywhere in the text. Terminal states, or the end of an occurrence, are marked with a double circle. The gray circles represent the active states after each character is read. As

we can see, when a state is active all the lower rows (higher number of errors) are also active. In the example, after reading "anneal" the NFAs terminal state is at $k = 1$, meaning we found a match with only one substitution needed.

A more efficient way of implementing approximate string matching is by exploiting the inherent bit-parallelism present in the central processing units (CPUs) in modern day computers [19]. Most bit parallel implementations involve simulations of other models, such as the NFA. One such example is the *Shift-AND* and *Shift-OR* algorithms as we will discuss in the next section.

2.5.1 Bit parallelism

In modern computers, CPUs generally operate with 32 and 64-bit registers, the latter being the most usual and what we will be assuming in this thesis. This means that a CPU inherently can perform atomic operations on values of up to 64 bits. Arithmetic operations are executed in a section of the CPU called the *Arithmetic Logic Unit*, or ALU. Simple bitwise operations on values less than 64 bits are generally very fast in modern CPUs, but when the size of the operands are above 64 bits the execution time also increases, as more operations are needed to compute. For this reason, highly specific tasks, such as cryptographic functions, whether it be encryption, decryption or different attacks are often implemented in hardware or *Field Programmable Gate Arrays* (FPGA), as demonstrated by Magnus Øverbø in his thesis [3].

In this thesis however, we seek to do dynamic experimentation and analysis of the final part of the cryptographic attack on irregularly clocked stream ciphers, and will thus only consider CPU implementations.

2.5.2 Bit parallel search

As mentioned, the previously discussed NFA representations of the search problem can be represented as bit parallel algorithms. One such algorithm for *approximate matching* is the *BRP* or *ARBP* (Approximate Row-wise Bit Parallel search) algorithm [20], based on extending the *Shift-AND* algorithm for approximate search.

Shift-And

Some previous implementations of the generalised correlation attack on irregularly clocked stream ciphers have used the Shift-AND algorithm [3]. However it has also been shown that the *Shift-OR* algorithm can increase performance in theory [2]. In order to illustrate the differences, the Shift-AND algorithm is explained first, before implementing the subtle difference that makes the leaner *Shift-OR* algorithm. The algorithms operation in the exact match scenario will also be demonstrated before introducing the possibilities of errors, making it an approximate search algorithm.

Given a pattern p being searched for in the text T, the algorithm keeps a *status word* $D = d_m...d_1$. The status word represents the set of *prefixes* of the pattern that

match a *suffix* of the text being read. A set bit (1) in D indicates that the pattern $p_1...p_j$ is a suffix of $t_1...t_j$. The occurrence of an exact match is indicated by d_m being set.

First, the prefix table B which stores a bit mask for every character in the pattern is built. In the following example we have the search pattern **NTNU** and its mask $B[N]$, has $p_1 = p_3 = 1$ as shown in Figure 2.9.

$B =$

N	0 1 0 1
T	0 0 1 0
U	1 0 0 0
*	0 0 0 0

$D =$ 0 0 0 0

Figure 2.9: The prefix table B and the initial status word D for the pattern **NTNU** before running the **Shift-AND** algorithm.

When we start the search, the status word holds the value $D = 0^m$. In order to update this we need an update procedure which is run for each read character in the search text. The *Shift-AND* update procedure is represented by the formula below:

$$D' \longleftarrow ((D << 1) \mid 0^{m-1}1) \,\&\, B[t_{i+1}]$$

Here, the $<<$ operator is a left shift of the current status word, which moves all bits one step to the left. This results in the status indicating which positions of p were suffixes at the previous step, i. Next, the left shifted value is **OR**ed with $0^{m-1}1$, which marks the empty string * as a suffix. Lastly, the result is **AND**ed with the suffix mask value created initially, $B[t_{i+1}]$. By doing this the positions where t_{i+1} matches p_{j+1} is marked in the search word. Algorithm 3 shows the pseduocode for the whole process. We can demonstrate the stages of the algorithm by simulating a search for the pattern **NTNU** in the text **NOT NTNU**, as shown in Figure 2.11. The *Shift-AND* case is shown on the left side.

Shift-OR

The *Shift-OR* variant is very similar and produces exactly the same result as the *Shift-AND* algorithm. However, the status word update procedure has fewer operations and is, as a result, faster. The simplified update procedure is shown together with the previously discussed *Shift-AND* procedure below, for comparison:

$$D' \longleftarrow ((D << 1) \mid 0^{m-1}1) \,\&\, B[t_{i+1}] \qquad \textbf{Shift-AND}$$
$$D' \longleftarrow (D << 1) \mid B[t_{i+1}] \qquad \textbf{Shift-OR}$$

We can immediately see that the new procedure consists of only two binary operations, as opposed to the three used by the *Shift-AND* algorithm. This reduces

Algorithm 3: The **Shift-AND** algorithm for exact match [21]

input : $p = p_1 p_2 ... p_m, T = t_1 t_2 ... t_n$
output: Alert if an occurrence is found
Preprocessing:
for $c \in \sum$ **do**
  $B[c] \leftarrow 0^m$
end
for $j \in 1...m$ **do**
  $B[p_j] \leftarrow B[p_j] \mid 0^{m-j}10^{j-1}$
end
Search:
$D \leftarrow 0^m$
for $pos \in 1...n$ **do**
  $D \leftarrow ((D << 1) \mid 0^{m-1}1) \,\&\, B[t_{pos}]$
  if $D \,\&\, 10^{m-1} \neq 0^m$ **then**
    Occurrence found at $pos - m + 1$
end

the amount of operations by one third and thus should in theory reduce execution time by $\frac{1}{3} \approx 33\%$. In order for this to work however, when we calculate the prefix masks B we need to take the *complement* values from the original calculation, as shown in Figure 2.10. By changing the prefix masks, we can *OR* the left-shifted value of D directly with the mask corresponding with the character being read. E.g. for the example used earlier with the pattern **NTNU** now $B[N]$ has $p_1 = p_3 = 0$. Also the status word has to be inverted so that it is now initiated as $D = 1^m$. As opposed to the previous case, where a 1 in the MSB position indicates a match, a match is found when $d_m = 0$.

$B =$

N	1 0 1 0
T	1 1 0 1
U	0 1 1 1
*	1 1 1 1

$D =$ 1 1 1 1

Figure 2.10: The prefix table B and the initial status word D for the pattern **NTNU** before running the **Shift-OR** algorithm.

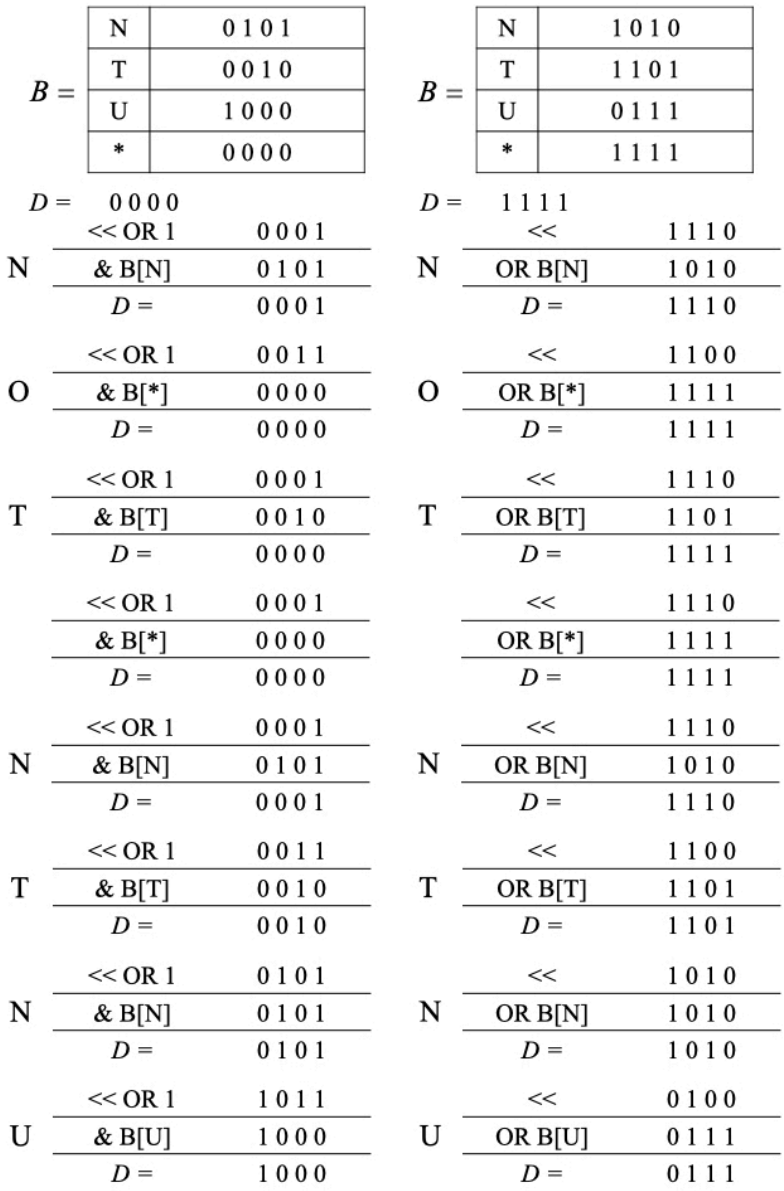

Figure 2.11: Side-by-side comparison of *Shift-AND* and *Shift-OR* input and ouput when searching for the pattern **NTNU** in the text **NOT NTNU**.

2.6 Approximate Row-wise Bit Parallel search

In this section the *Shift-OR* algorithm is extended to allow errors, making it an efficient approximate search algorithm. The original approach, using the extended version of *Shift-AND* was demonstrated in [22]. The extension involves setting a *threshold* value of number of allowed errors k, as demonstrated earlier in Figure 2.8. As opposed to the exact match scenario, $k+1$ search status words $R_{i=0..k}$ are used, each of which represent an approximate match with i errors and where $i = 0$ represents the exact match. The initial generation of these is simple, as each status word is constructed by left-shifting the exact match word $R_0 = 1^m$ by i positions, giving each error a "head start" to that of the exact match. The status word update procedure for R_0 remains the same, however a more complex procedure to accommodate for errors in $R_{i=1..k}$ is needed. The complete update function with errors is presented in the formula below [2]:

$$
\begin{aligned}
R'_0 &= ((R_i << 1) \mid B[t_j]) \\
R'_i &= ((R_i << 1) \mid B[t_j]) \ \& && \text{(match)} \\
&\quad ((R_{i-1} << 1) \mid \text{NOT } B[t_j] \ \& && \text{(sub)} \\
&\quad (R'_{i-1} << 1) && \text{(del)} \\
&\quad i = 1...k
\end{aligned}
$$

This procedure is *constrained* as it allows for *substitutions* and *deletions*, but not *insertions* or *reversals*. The scenario which is discussed in this thesis involves searching for the intercepted ciphertext (the pattern) in the undecimated bit sequence of the LFSR $R2$ (the search text). To properly emulate the BRMs decimation function, the case of *insertions* and *reversals* can be omitted. As we can see, the *substitution* procedure **OR**s with the complement of B ($\mid$ NOT $B[t_j]$). This operation makes sure that no *substitutions* occurs after a *match* in the previous row of the corresponding NFA.

An even further modified attack is proposed in [2] in order to reduce the amount of false positives when performing the generalised correlation attack. This is done by introducing even more constraints in order to eliminate "impossible" embeddings, e.g. where the amount of decimated (or skipped) bits exceeds that of the systems design, which is 1 in the case of the BRM system we are analysing in this thesis.

$$
\begin{aligned}
dm_i =& 1^m, i = 1...k \\
del =& 1^m \\
R'_0 =& ((R_i << 1) \mid B[t_j]) \ \& && \text{(match)} \\
R'_i =& ((R_{i-1} << 1) \mid \text{NOT } B[t_j] \ \& && \text{(sub)} \\
& (R_{i-1} << 1) \mid && \text{(del)} \\
& \text{NOT } ((del << 1) \mid 0^{m-1}1) \mid \\
& \text{NOT } ((dm_{i-1} << 1) \mid 0^{m-1}1) \mid \\
& (0^{m-1}1 << (m-1)) \\
& i = 1...k
\end{aligned}
$$

As we can see in the modified approach the procedures for *matches* and *substitutions* are unchanged but the *delete*-operation is significantly more complex. However, the omission of false positives should reduce the total set of candidates C which in turn would lead to lower execution times in the final stage, the reconstruction of the clocking LFSRs output sequence and initial state $R1_{init}$.

In Figure 2.12 a practical example of the *Shift-OR* approximate search algorithm with $m = 10, k = 2$ and the ciphertext 1010100101(bin) is shown. This output is generated by the program developed in this thesis, which will be discussed in detail in chapter Chapter 4.

```
        INIT STT:       11100010010
        DEGREE:         11
        LENGTH:         20
        POLYNOMIAL:     10010111001
        FINAL STT:      10101111100
        SEQUENCE:       01111000001001000111

Gen error R[0..2]
        R[0]    = 1111111111
        R[1]    = 1111111110
        R[2]    = 1111111100

Beginning search
-------------- 0
B[1]: 1010100101
R[0]: 1111111110
R[1]: 1111111100
R[2]: 1111111000

-------------- 1
B[1]: 1010100101
R[0]: 1111111110
R[1]: 1111111000
R[2]: 1111110000

-------------- 2
B[1]: 1010100101
R[0]: 1111111110
R[1]: 1111111000
R[2]: 1111110000

-------------- 3
B[0]: 0101011010
R[0]: 1111111101
R[1]: 1111110000
R[2]: 1111100000

-------------- 4
B[0]: 0101011010
R[0]: 1111111111
R[1]: 1111100000
R[2]: 1111000000

-------------- 5
B[0]: 0101011010
R[0]: 1111111111
R[1]: 1111100100
R[2]: 1110000000

-------------- 6
B[1]: 1010100101
R[0]: 1111111110
R[1]: 1111011000
R[2]: 1100000000

-------------- 7
B[0]: 0101011010
R[0]: 1111111101
R[1]: 1110110000
R[2]: 1000100000

-------------- 8
B[0]: 0101011010
R[0]: 1111111111
R[1]: 1111100000
R[2]: 1001000000

-------------- 9
B[1]: 1010100101
R[0]: 1111111110
R[1]: 1111011000
R[2]: 0110000000 [!]

-------------- 10
B[0]: 0101011010
R[0]: 1111111101
R[1]: 1110110000
R[2]: 1100100000

-------------- 11
B[0]: 0101011010
R[0]: 1111111111
R[1]: 1111100000
R[2]: 1001000000

-------------- 12
B[0]: 0101011010
R[0]: 1111111111
R[1]: 1111100100
R[2]: 1010000000

-------------- 13
B[0]: 0101011010
R[0]: 1111111111
R[1]: 1111101100
R[2]: 1110000000

-------------- 14
B[0]: 0101011010
R[0]: 1111111111
R[1]: 1111111100
R[2]: 1110000000

-------------- 15
B[1]: 1010100101
R[0]: 1111111110
R[1]: 1111111000
R[2]: 1101010000

-------------- 16
B[1]: 1010100101
R[0]: 1111111110
R[1]: 1111111000
R[2]: 1111110000

-------------- 17
B[1]: 1010100101
R[0]: 1111111110
R[1]: 1111111000
R[2]: 1111110000

-------------- 18
B[1]: 1010100101
R[0]: 1111111110
R[1]: 1111111000
R[2]: 1111110000

-------------- 19
B[0]: 0101011010
R[0]: 1111111101
R[1]: 1111110000
R[2]: 1111100000

Search done.
```

Figure 2.12: A practical example of the *Shift-OR* algorithm.

Chapter 3

Methodology

In this chapter, the methodology used throughout the work on this thesis is explained. A few of the essential disciplines and tools are also explained together with potential limiting factors that may affect the work on this thesis. Finally, a project timeline overview is presented.

3.1 Research methodology

The topic of the thesis requires elements from both a development project and a research project. In order to answer the research questions asked in the first chapter, a mixed-methods approach of both qualitative and quantitative methods will be utilised. Specifically a *mixed-methods exploratory design*[23] is the chosen methodology for achieving the goals of this thesis. Figure 3.1 below displays the overarching project design which will be followed. An exploratory sequential design involves an initial qualitative phase (Phase One) which has the purpose of developing the instruments for use in the the quantitative phase (Phase Two). In this project, the first phase will involve the initial research, assessment of earlier studies and, ultimately the software development itself. The second phase will involve quantitative testing of the developed software elements and data gathering.

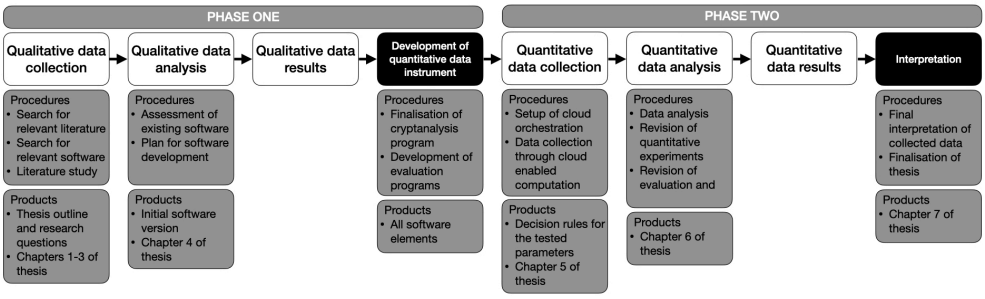

Figure 3.1: Mixed-methods exploratory design [24].

Through the initial, qualitative phase the works of earlier researchers on the topic

will be thoroughly examined and possibilities for improvement will be investigated. In particular this involves studying and evaluating the programming work done in [3]. Several iterations of software development will be undertaken in this phase, in order to qualitatively test different versions and algorithm variants with varying input parameters. The result of phase one is the final cryptanalysis program together with any additional evaluation software. This forms the instruments with which the quantitative experimentation and analysis will be carried out in phase two.

In order to prevent any compromise of the data gathered in the second phase, the functionality of the main program should not be changed after the initial phase is completed and the quantitative data gathering has commenced. However, some of the wrapping evaluation software or cloud templates may be adjusted through the course of the quantitative experiments.

3.2 Essential disciplines and tools

3.2.1 Software instrumentation

Software instrumentation is the act of measuring execution times and providing *trace* information in computer programs [25]. While optimal performance is not the most important part of the software development in this thesis, it is very important to analyse the performance of critical parts of the cryptanalysis program in order to understand execution times and time complexity of the full cryptographic attack. The results of software instrumentation may lead to a better understanding and also performance improvements throughout the development phase of this thesis, or give useful pointers for future research on the topic.

3.2.2 Analysis

The analysis activity in this project is divided in two parts. First, the qualitative analysis and assessment of existing literature and software which gives direction to the further software development. Secondly, the analysis of data gathered from the experiments run on the newly developed software. In a scenario such as this, where high complexity cryptographic calculations are involved it can be difficult to provide a large enough data set for analysis within the given time limit and financial constraints. A trade-off between realistic input parameters (for example higher orders of LFSR and longer intercepted bit sequences) and parameters that will provide enough quantitative data for sensible analysis will likely have to be made.

3.2.3 Tools

The software development in this thesis is done on a 2,4GHz 8-core Intel i9 MacBook Pro using the *VSCodium* IDE [26]. For virtual hosts used in the experiments,

Ubuntu 20.04 LTS [27] was chosen as the operating system. All data graphs in Chapter 5 have been generated in *DataGraph* [28] and the thesis itself written in LaTeX on *Overleaf*.

3.3 Limiting factors

3.3.1 Financial factors

As research question **RQ4** asks '*Is it possible to improve the performance of the overall algorithm by introducing orchestrated multi-threading using cloud based infrastructure?*' a paid account for the chosen cloud service will be needed. A financial cap is set to NOK3.000,- (approximately USD360$) for the experimental phase of this thesis. Compared to the predicted time complexity of simulating a full attack implementation with realistic input parameters, this limit may be too low, but a best effort will be made to produce representable results with the means available.

3.3.2 Legal factors

While several specific technologies employing *Stream ciphers* may be mentioned as examples in the thesis, none of them are directly subject to the attempt at a cryptoanalytical attack. If however discoveries are made that may impact any known technologies, these may be withheld from the thesis in order to perform a proper *responsible disclosure* [29] to whom it may concern. Any such discoveries will be discussed with the thesis supervisor before a decision is made.

Also, it is important that any third-party software libraries used in the project are checked for any applicable licensing models and if the model calls for it, ask for permission. Regardless of licensing model, credit will always be given where credit is due, to the original author.

3.4 Timeline

Figure 3.2 shows a *Gantt chart*[30] for the software development and thesis work. It has to be noted that the qualitative research phase started during the autumn semester of 2020. The final delivery deadline was also set to 1st of June, 2021.

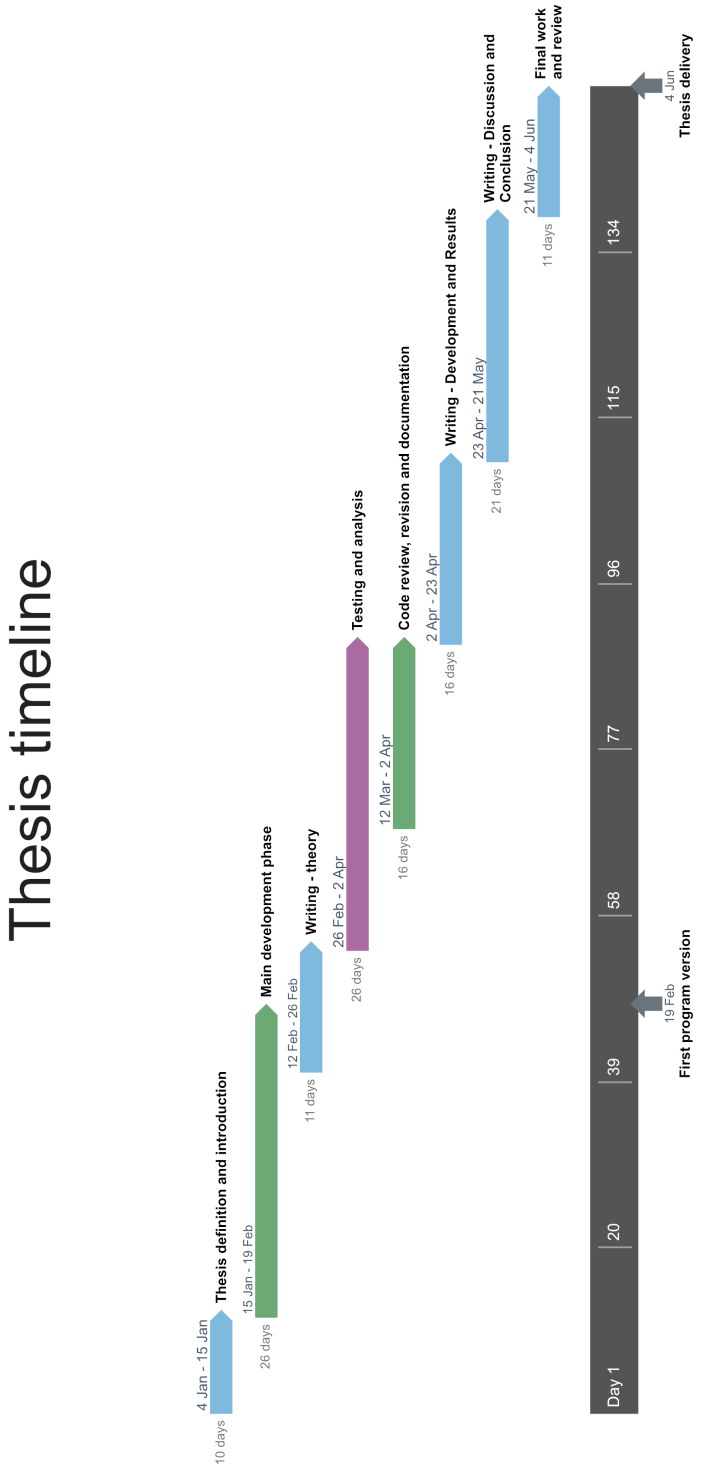

Figure 3.2: Thesis timeline

Chapter 4

Implementation

In this chapter the developed algorithms and their corresponding software implementations are presented. The testing and analysis process with which the results (which are presented in the next chapter) have been derived are also described. First, we address the different technologies with which the software is built, along with the rationale for using them. Next, the algorithms are presented, which contains the proposed improvement on generation of candidate sets, the clocking-LFSR reconstruction and the algorithms for evaluation of input parameters and performance. Finally, we look at the design of the different software elements. All implementations are done in software and written in the ***C***[31] and ***Go***[32] programming languages. Previous papers have addressed parts of the problem in both software and Field-programmable gate arrays (FPGA)[3]. While specifically designed FPGA implementations may perform better, the experimental nature of the work in this thesis requires more flexibility and as such, only software implementations have been considered. However, some of the tests have been done by running the software on high capacity virtual hosts in the cloud which will also be shown in this chapter.

4.1 Technologies

In this section, the different technologies that have been used to build the cryptanalysis program itself and the evaluation elements are presented and briefly discussed.

4.1.1 C99

The C programming language is the ideal language for high performance computer programs. It is a *low level* language, which means that it has a relatively thin abstraction layer to the resulting machine readable code which is produced when compiling, compared to other compiled languages such as **Golang**. This in turn means that the programmer has a higher level of granularity and control

over the resulting machine execution of the program. For the software implementations in this thesis the C99 version of C, defined in ISO/IEC 9899:1999 [33], has been used as it is the most widespread implementation and it is fully POSIX compliant. POSIX compliance is favourable as the software is intended to be able to run in a distributed manner on multiple hosts with potentially different operating systems. One major difference between C99 and the more recent C11 (ISO/IEC 9899:2011 [34]) is its implementation of *multi-threading*. C99 uses **pthreads** (POSIX threads), while C11 implements a new type of threads. While the new edition is somewhat improved and more easy to use, it will not compile on certain operating systems.

Multi-threading with pthreads

One important improvement that has been made in this thesis is the multi-threading of certain parts of the existing software, which will be shown later. With C99 being the chosen programming language, the multi-threading features was implemented using the native **pthreads**-library [35]. As this library offers no advanced control features for threading and depends on the programmer implementing these himself, a helper library was needed in order to orchestrate the threads properly. Two different modes of operation were considered; the producer-consumer model and the thread-pooling model. The first works by having a set amount of producer-threads running a function in parallel, and a different set of consumers handling the returned data from the producers. The latter is simply a model for setting a limit to how many threads can be made available to the program at one time. As we will see later, each thread reads and writes to and from a dedicated part of the memory stack, separated from each other, and so the option of thread-pooling was found to be the suitable choice for this application. The problem of limiting the number of threads running in a system may seem trivial, but alas, when the main function starts a *pthread* it releases it and does not keep track of how many are running. This can in some systems result in *segmentation faults* as operating systems tend to set different limits as to how many threads are allowed. It is also a point that more threads does not always mean faster execution times. The inherent multi-threading capacity in any CPU is constrained to that of its available cores, and so, while a system can have hundreds, even thousands of threads running, it does not make sense to run more threads than the number of cores available. In normal systems, we can say that a program should have $thread_{limit} = vCPU_{total} - 1$, allowing one core open for other system or user tasks. As the program in this thesis is meant for running on dedicated hosts, having no user interaction and a minimal of other system processes, we always assume $thread_{limit} = vCPU_{total}$. This setting has also been tested and verified through several *trail-and-error* runs of the program.

The thread-pooling library used in this thesis is based on the *MIT licensed*[36] code found at [37]. The library consists of the files **tpool.h** and **tpool.c** which can be found in Appendix A under Code listing A.1 and Code listing A.2.

It has to be noted that no parts of the source code has been written with security of the running hosts in mind. The combination of C and multi-threaded memory manipulation without carefully implemented security measures, may open the program to a number of possible exploits. As such, compile and run the program at your own risk, and never as *root* or *Administrator*.

4.1.2 GMPLib - GNU Multiple Precision Library

As mentioned earlier, CPUs are normally constrained by their register size, when executing arithmetic operations in the ALU. The normal register size in modern computers is 64 bits, and 32 bits in some legacy computers. In the case of bit-parallel computation in cryptanalysis, as in this thesis, the computed search patterns and text will in most cases exceed far beyond 64 bits and, as such, we will need a method of working around the CPUs inherent constraints. This can be done by using the *GNU Multiple Precision Arithmetic Library* (GMPLib) [38]. This library allows arithmetic operations on arbitrarily large numbers, and also contains a number of convenient functions for different operations and representations of large values. GMPLib is only limited by memory usage, which still has to be monitored and maintained by the programmer.

While the library offers very low-level operations and functions in its *mpn*-category, the *mpz*-category provides more easier to use functions and is assumed to provide sufficient efficiency at this stage of the research on this topic. If a more efficient practical attack is to be implemented in the future however, one may consider migrating the code base to use *mpn*-functions.

It has to be noted that the operations provided by this library are in no way *atomic*, in the sense that they do not represent one bit operation in the CPU or ALU. Each function will in most cases perform many low-level operations in order to calculate the results, and more operations will be needed as the size of the operands grow larger.

4.1.3 Golang

Go, or Golang as its also called, is a higher level programming language developed at Google Inc. It is a compiled language which aims to provide, among many other things, good and accessible implementations of multi-threading or *Go-routines* as they are called in Go [39].

It was chosen as the framework for orchestrating the experimental process of this thesis, mainly because of its ability to easily produce cross-compiled binary executables for use on multiple platforms and its self resolving dependency-system. Several *run-time interpreted* languages (such as Python and Bash) were considered for this task, but Golang was chosen. The Go-framework written for this thesis will be discussed in more detail later in this chapter, but the source code can be found in Code listing A.5 in Appendix A.

4.1.4 Amazon Web Services (AWS)

Amazon Web Services (AWS) is a well known cloud infrastructure provider run by *Amazon.com, Inc.* [40] which provides many different services, both for commercial use and for research purposes. Of special relevance to this topic was the ability to rent highly powerful virtual hosts with up to 96 cores (or vCPUs) at *spot pricing*. The AWS spot-price instance requests work by letting the customer send a request for a certain virtual host type, with availability within a certain timeframe. AWS accepts this requests and provides the requested computing power whenever there is "left-over" capacity in their systems. Spot-price instances are often up to 70% cheaper than the equivalent *on-demand* price which enables more quantitative data gathering to be done within the thesis given timeline and financial limits.

By default, AWS limits the amount of total vCPUs a user can utilise. A request was sent to AWS for a limit increase to $2 \times 96 = 192$vCPUs. This was however only partially fulfilled up to 172vCPUs.

AWS also conveniently allows for automatically running pre-programmed *Launch templates* being executed upon an accepted request. In this way, the workload can be executed as soon as the host is provided, without any user interaction. The instance is terminated on the behest of Amazon, the set requested time limit runs out, or the user terminates the request. The program continually writes results to a mounted persistent cloud storage, which is available regardless of the host being terminated. The host initialization routine written for the Ubuntu 20.04 LTS AWS template is given in Code listing A.7.

There are several alternatives to AWS available, including Google GCP [41], which may or may not provide the same type of functionality. No alternatives were considered or tested for use in this thesis, due to the authors previous familiarity with AWS.

4.2 Algorithms

In the following section descriptions and explanations of the programs algorithms and corresponding software implementations are given. This includes some of the existing algorithms, which was developed in [3], some of which has been modified, and also a couple of new ones.

The main cryptanalysis program (**main.c** - often just referred to as the "main program" in this thesis) is built on a number of different research papers as referenced in Chapter 2. It simulates a BRM-based pseudorandom generator and the encryption of a plaintext to produce a ciphertext. Next, it attempts to reverse the process with only knowledge of the ciphertext and plaintext, in order to derive the key.

In this section we will use the following symbols and abbreviations:

- $p \in \{11, 16, 20\}$ - the polynomial degree of both LFSR
- $n \in \mathbb{Z}^+$ - the search text length in bits

- $m \in \mathbb{Z}^+, \frac{n}{2} < m < n$ - the search pattern length in bits
- $R2_{init} \in \mathbb{Z}^+, 0 < R2_{init} < 2^p - 1$ - The initial state of the decimated LFSR, $R2$, in decimal value.
- $R1_{init} \in \mathbb{Z}^+, 0 < R1_{init} < 2^p - 1$ - The initial state of the clocking LFSR, $R1$, in decimal value.
- C - the set of $R2$ candidates where each candidate C_i consist of the tuple $R2_i$ and the corresponding LFSRs output sequence $R2_{0...n}$.
- $k \in \mathbb{Z}^+, 0 < k < m$ - the number of allowed errors during the $R2$ candidate selection.
- $a \in \mathbb{Z}^+, 0 <= a < (2^p - 1)$ - the number of accepted ciphertext collisions accepted when evaluating the $R1_{init}$ and $R2_{init}$ combinations.

For the experiments in this thesis, some of the algorithms "*cheat*" at certain points in order to evaluate its own performance. When *cheating*, the program assumes *a priori* knowledge that it would not have in an actual practical attack. An example of this is assuming the knowledge of $R2_{init}$ in order to evaluate the selected set C by checking if the real $R2_{init}$ has been added to C according to the H_1-hypothesis discussed in Section 2.4.4. Any source code or pseudo-code lines with such implementations are clearly marked, both in this thesis and in the attached source code.

The main program can be divided into the following overarching algorithms:

Step 1 - Create the target stream cipher system on which we will perform the attack.
Step 2 - Generate the set of $R2_{init}$ candidates, C, through ARBP search.
Step 3 - Test each candidate $C_i \in C$ against every $R1_{init}$ until a matching ciphertext is generated in order to recover the correct clock sequence.

In addition to the main program an orchestration algorithm has been implemented, which runs the main program repeatedly with different input values in order to perform the quantitative analysis.

The algorithm for calculating candidates for the initial state of the decimated LFSR, $R2$, was implemented in [3], but the main focus of the thesis was an implementation in FPGA, which is not a topic in this thesis. While the implementation in this thesis builds on previously developed software, many of the algorithms have been replaced or restructured.

In the following sections some of the most important code snippets are shown in the text, however, for brevity, not all functions are given in full. The full source code can be found in Appendix A.

4.2.1 Polynomials

As shown in Section 2.2 the fact that the polynomial of the component LFSRs are *primitive* is essential to the operation of the BRM cryptosystem. In the experiments performed in this thesis the polynomial degrees are limited to $p \in \{11, 16, 20\}$. The corresponding primitive irreducible polynomials are fetched from [42], which in turn in derived from [43]. The polynomials are presented in Table 4.1 and im-

plemented in the **polyMap()**-function in the main program. This function simply sets an **mpz_t**-variable by the decimal value based on the polynomial degree given as the first argument to the program. Then the bits of the binary representation of the variable represents the LFSRs *taps*.

Table 4.1: Primitive irreducible polynomials $p = \{11, 16, 20\}$.

p	Polynomial	Decimal
11	$x^{11} + x^8 + x^6 + x^5 + x^4 + x^1 + 1$	1209
16	$x^{16} + x^9 + x^8 + x^7 + x^6 + x^4 + x^3 + x^2 + 1$	33262
20	$x^{20} + x^9 + x^5 + x^3 + 1$	524564

4.2.2 BRM simulation

At the base of the main program is the simulation of the Binary rate multiplier. Recall Algorithm 2 given in Section 2.3.1. In the software implementation of the main program given in Code listing A.3 in Appendix A, we can see that this is the first task to be completed immediately after the variable settings and input validations. As we can see from the software implementation of the algorithm, given in Code listing 4.1, it differs slightly from Algorithm 2 in that it pre-generates the output sequence of both LFSR instead of stepping it bit by bit. This may not be a suitable implementation for a practical implementation of a stream cipher system but it works for the pre-defined nature of the input data that is being used in this thesis.

Code listing 4.1: BRM Simulation

```
1  ...
2   mpz_init(PLAINTEXT);
3   mpz_set_ui(PLAINTEXT, plain);      //Prepare the plaintext
4
5   // SIMULATE THE BRM SYSTEM
6   mpz_t R1SEQ;  mpz_init(R1SEQ);
7   lfsrgen(R1SEQ, deg, m, pol, R1STATE, 0, NULL);  //Generate R1 output sequence
8
9   mpz_t R2SEQ;  mpz_init(R2SEQ);
10  lfsrgen(R2SEQ, deg, n, pol, R2STATE, 0, NULL);  //Generate R2 output sequence
11
12  mpz_init( CIPHER );
13  genEncrypt( CIPHER, R1SEQ, R2SEQ, PLAINTEXT, m);    //Calculate ciphertext
14
15  mpz_clear( R1SEQ );         //Cleanup LFSRs
16  mpz_clear( R2SEQ );
17 ...
```

Here we see that all sequences used in the source code are generated using the *GMP-library*, and defined and initialized by the data type **mpz_t** (a *multiple precision integer*) and the command **mpz_init**, which sets its initial value to 0. The **mpz_set_ui** command given in line 3 of the code sets the **mpz_t** variable given as

the first argument to the **unsigned integer** given in the second parameter. For the experiments done in this thesis, the empty plaintext, PLAINTEXT $= 0^m$ is used.

Next, the function **lfsrgen()** is run for both $R1$ and $R2$ to produce their output bit sequences $R1_{i=0...m}, R2_{i=0...n}$, based on the given $R1_{init}$ and $R2_{init}$. We can recall that the variable $m, \frac{n}{2} < m < n$ in order for it to have enough clocking bits for the decimation process in the next step. Generally, if $R1$ satisfies Golomb's first postulate, $m = \frac{n}{2} + 1$ should always be sufficient. This boundary is always checked as part of the input validation of the program.

The function **genEncrypt()** takes the previously generated sequences as its input and calculates the decimated bit sequence, which in turn is **XOR**ed with the plaintext and returned as the ciphertext and stored in the variable **CIPHER**. The **CIPHER** variable is kept throughout the program and is considered as the "intercepted" ciphertext when performing the full attack demonstration.

Finally, both the initially generated sequences of the clocking and clocked LFSR are cleared from memory by the **mpz_clear** command. Normally, this is also where the program would "forget" the initial states of the two LFSR, but as mentioned earlier it will use these to validate results later on, in the evaluation phase.

4.2.3 R2 Candidate generation

After the BRM system has been simulated and the ciphertext generated, we can start the attack on the cryptosystem. For the attack we assume the *a priori* knowledge of the "intercepted" ciphertext, the plaintext and the design of the cryptosystem, making it a **known-plaintext attack**, as discussed in Section 2.4.

The attack starts by performing the *generalised correlation attack* to generate a set of possible candidates C for the decimated LFSR, $R2$. Recall the two hypotheses discussed in Section 2.4.4.

H_0 The intercepted sequence **cannot** be generated by the tested initial state.
H_1 The intercepted sequence **can** be generated by the tested initial state.

The search pattern is the ciphertext generated in the previous step of the program. The search text is the output sequence generated by $R2$ when iterating through every possible initial state $R2_i, i = 1...(2^p - 1)$. The error threshold k is also defined. The candidate generation algorithm is given in pseudo-code in Algorithm 4. Note that the pseudo-code is written in order to give the reader an overview of the program flow, and as such, some of the functions and their arguments are given implicitly. The full details can be found in the source code in Appendix A.

As we can see in Algorithm 4, the pre-processing consists of running the **genPrefixes()**-function, which creates the prefix table, as demonstrated earlier in Figure 2.10. As we are only considering the binary alphabet for this application, the resulting array **B** will always consist of two masks; the search pattern itself and its complemented value.

Algorithm 4: The candidate generation algorithm

input : CIPHER, k, p
output: the set of candidates C_i and the candidate counter u
Preprocessing:
$B \leftarrow$ genPrefixes(CIPHER)
Candidate generation:
for $i = 1...(p^2-1)$ **do**
  Create thread:
  R2SEQ $\leftarrow$ lfsrgen(R2$_i$)
  match $\leftarrow$ arbp_search(B, R2SEQ, k)
  $C_i[SEQ] \leftarrow R2SEQ$
  if $match = True$ **then**
    $C_i[match] \leftarrow$ True
  else
    $C_i[match] \leftarrow$ False
  end
end
for $i = 1...(p^2-1)$ **do**
  if $C_i[match] = True$ **then**
    file $\leftarrow$ write($C_i[SEQ]$)
    if $i = R2_{init}$ **then**
      found $\leftarrow 1$ // **CHEAT**
    $u \leftarrow u+1$
end

Next, the program loops through every possible initial state of $R2$ and for each iteration the output sequence of $R2$ is generated to form the search text which is used in the **arbp_search()**-function.

There are two essential differences in this implementation of the ARBP search algorithm compared to the earlier proposed implementation in [3]. First of all, every iteration of the first **for**-loop spawns its own *pthread* as long as there are threads available in the *thread pool*. As discussed earlier, the thread pool is limited to the amount of available CPU *cores*, or *vCPUs*[1].

The second important difference, is the implementation of the *Shift-OR* algorithm in the **arbp_search()**-function for updating the status words. In Code listing 4.2 and Code listing 4.3 we can see a side-by-side comparison of the *Shift-OR* and the *Shift-AND* code for exact match, or $k = 0$, respectively.

Code listing 4.2: OR $k = 0$

```
mpz_lshift( tmp1, m );
mpz_ior( tmp1, tmp1, B[Tii] );
```

Code listing 4.3: AND $k = 0$

```
mpz_lshift( tmp1, m );
mpz_setbit( tmp1, 0 );
mpz_and( tmp1, tmp1, B[Ti] );
```

In the code, **tmp1** represents a temporary variable holding the current status word. It is directly manipulated and written back through the given **mpz_***-commands. The variable **B[Ti]** is the pre-calculated prefix for the last read letter in the search text. In the *Shift-OR* implementation however, the inverse prefix is used. This is denoted **B[Tii]**. As can be seen in the code, it exploits the fact that, in the binary case, the two prefixes for 0 and 1 will always be the inverse of each other. The variable initialisation is shown in the code snippet below. This is a much more efficient way to get the complement than using the relatively costly **mpz_com()**-function [44] of the GMP library.

```
int Ti = mpz_tstbit( TEXT, pos );
int Tii = (Ti + 1) % 2;
```

As expected, the C-implementation of the *Shift-OR* algorithms exact match algorithm is one operation shorter than the previous *Shift-AND* implementation, as shown in Section 2.5.2. The exact match operation is performed $n \times (2^p - 2)$ times (-2 because the 0-state is not used) during the $R2$-candidate generation. It is easy to see that the performance of this operation is less important than that of the approximate match operation with k-errors, as this is executed $n \times (2^p - 2) \times k$ times.

When the *Shift-OR* algorithm for approximate match is implemented with the same constraints (OR1) as the previous implementation of *Shift-AND*, we see from the code samples below that also the equivalent status word update function with errors is shorter[2]. However, as pointed out in [2], setting a constraint that makes sure a *substitution* edit-operation occurs after a *match* is more relevant in this particular application of the algorithm. The first constrained version (OR2), as it is

[1] This limit was determined through a trail-and-error approach by measuring the runtime of the program on a dedicated host.

[2] This version of the program is compiled by default when running the **make**-command

explained in Section 2.5.2, is shown in Code listing 4.6[3] Note especially the OR-operation with **R[i]** and **B[Tii]**. As mentioned, **B[Tii]** is the complement value of the prefix. Due to this operation, the constrained implementation can not use the same shortcut as the simplified one, which is combining the *deletion* and *substitution* operations. This results in the same amount of operations as the original *Shift-AND* algorithm.

While performance has not been the main focus of this thesis, a best effort has been made to make these operations as efficient as possible while still using the GMPLib functions. Detailed instrumentation and performance results are presented in Section 5.2.

Code listing 4.4: OR1 $k > 0$

```
mpz_set(tmp1, oldR);
mpz_and(tmp1, tmp1, newR); //Del+Sub
mpz_lshift(tmp1, m);
mpz_set(oldR, R[i]);
mpz_lshift(R[i], m);
mpz_ior(R[i], R[i], B[Tii]); //Match
mpz_and(R[i], R[i], tmp1);
mpz_set(newR, R[i]);
```

Code listing 4.5: AND $k > 0$

```
mpz_ior(tmp2, oldR, newR); //Del+Sub
mpz_lshift(tmp2, m);
mpz_setbit( tmp2, 0 );
mpz_set(tmp1, R[i]);
mpz_lshift(tmp1, m);
mpz_and(tmp1, tmp1, B[Ti]); //Match
mpz_ior(tmp1, tmp1, tmp2);
mpz_set(newR, tmp1);
mpz_set(oldR, R[i]);
mpz_set(R[i], newR);
```

Code listing 4.6: OR2 $k > 0$

```
mpz_lshift(newR, m); //Del
mpz_set(tmp1, oldR);
mpz_lshift(tmp1, m);
mpz_ior(tmp1, tmp1, B[Ti]); //Sub
mpz_set(oldR, R[i]);
mpz_lshift(R[i], m);
mpz_ior(R[i], R[i], B[Tii]); //Match
mpz_and(R[i], R[i], tmp1);
mpz_and(R[i], R[i], newR);
mpz_set(newR, R[i]);
```

If we further constrain the *Shift-OR* as suggested as the *modified attack* in [2] the code obviously gets even more complex. An experimental implementation of this algorithm has been implemented in the program (OR3) and can be used by setting both the **-DSHIFTOR** and **-DCONSTRAINTS2** flags in the makefile before compilation. As mentioned in Section 2.5.2, this approach requires an additional register $dm_i = 1^m, i = 1...k$ which also increases the amount of operations and memory used for each run of ARBP. The suggested implementation is given in Code listing 4.7. The first part shows the initialisation of the new status array dm_i. The process exploits the fact that $R[0] = 1^m$ at this point in the code which makes the initialisation more efficient than using multiple **mpz**-functions in order to set the bit mask. Note that the new *deletion*-operation requires significantly more operations (12) than the previously discussed OR1 and OR2 approaches. This may however be worth it if in fact the further constrained algorithm provides a candidate

[3]This version of the program is compiled by compiling it with the **make or2**-command.

set C with fewer false positives than that of the less constrained implementations.

Code listing 4.7: OR3 $k > 0$

```
mpz_t* dm = malloc( (K)*sizeof(mpz_t) );
mpz_t del;
mpz_init(del);
int d = 0;
while (d<K) {
 mpz_init(dm[d]);
 mpz_set(dm[d], R[0]);
 d++;
}
mpz_set(del, R[0]);

...

// Deletion
mpz_set(tmp2, oldR);
mpz_lshift(tmp2, m);

mpz_lshift(del, m);
mpz_setbit(del,0);
mpz_com(tmp3, del);

mpz_ior(tmp2, tmp2, tmp3);

mpz_lshift(dm[i-1], m);
mpz_setbit(dm[i-1], 0);
mpz_com(tmp3, dm[i-1]);

mpz_ior(tmp2, tmp2, tmp3);

mpz_setbit(tmp2, (m-1));

mpz_set(dm[i], del);

...
```

As we see in Algorithm 4 and Code listing A.3, all results C_i are stored in memory, regardless of whether they are a considered a possible match (H_1) or not (H_0). As we can not know beforehand how many candidates there will be in the set, we always assume the worst case of $2^p - 2$ and allocate the necessary memory on the stack before executing the search algorithm. However, there is a boolean field in the candidate struct, **match** that is set to **True** if the initial state has one or more matches within the set error threshold, k. While memory usage has not been an issue while running the experiments related to this thesis, the practice of allocating more memory than needed may be improved upon in future developments of the program.

After all initial states have been tested and stored in memory, the array of $2^p - 2$ possible candidates is quickly iterated through once again in order to check which are valid candidates (**match = True**) and to write these to a file on disk, together with their corresponding output sequence. Here, the program "cheats" by checking whether the actual initial state $R2_{init}$ is selected according to H_1 for evaluation purposes. It is easy to imagine that the best set we can end up with contains

only one entry, namely $R2_{init}$. While this is a highly unlikely case, the aim turns to finding the smallest possible set, that also contains $R2_{init}$. In this loop, the total number of candidates are counted and stored in a temporary variable **u**, which is helpful in determining the next steps. The file is saved as a CSV-formatted list with the *.cand*-extension which contains the initial state in decimal and the output sequence in a *base-62* encoding as defined in [45]. This feature is primarily intended for adding the possibility of distributing the steps of the cryptographic attack between different hosts; e.g. host1 creates candidate-files and multiple hosts test clock sequence reconstructions. This particular form of load balancing was not tested in this thesis, but may be relevant in future studies on the topic.

Finally, we evaluate the results found in the previous routine based on the variable **u**. This is done according to the following decision flow:

Algorithm 5: Decision flow to determine whether or not to continue to the next step

```
input  : found, u - the amount of candidates in C_i
output: the next step
switch u do
    case 0 do
        No candidates found
        return −1
    end
    case u = (2^p − 2) do
        Full set - All initial states are candidates
        return −2
    end
    case found != True do
        The actual initial state R2_init is not in the set //CHEAT
        return u
    end
    otherwise do
        continue
    end
end
```

As we can see, any set that satisfies $0 < u < (2^p - 2)$ and $R2_{init} \in C_i$ are considered valid and allows the algorithm to continue. In the cases of invalid sets, the program exits with different return values in order for the user running the software to determine the cause. The return codes and program are structured in a machine readable way in order to enable wrapping of the program. As will be demonstrated later in this thesis, the wrapping program for evaluation and orchestration depends on this return value in order evaluate and determine the next step. In order to get a more human-readable output from the program, it can be compiled with the **make debug** command. In the case of a valid set the programs execution continues and initiates the clock sequence reconstruction algorithm.

4.2.4 Clock sequence reconstruction

As stated in Section 1.5, **RQ2** asks whether *a practical attack on irregularly clocked stream ciphers based on BRM can be implemented and tested using brute-force in software*. In the case of a valid candidate set produced from the previous algorithm it is easy to perform a *brute-force attack* by testing every initial state of $R1$ against the elements of the candidate set C_i as shown in Algorithm 6.

Algorithm 6: Brute force attack on the system

```
input  : C_i and the intercepted ciphertext
output: matches
for R2_i ∈ C_i do
    X ← C_i[SEQ]
    Create thread:
    for j <= (2^p − 2) do
        Y ←lfsrgen(R1_j)
        Z ←genEncrypt(X, Y)
        if Z = CIPHER then
            R2_i and R1_j produces the ciphertext
        else
            R2_i and R1_j does not produce the ciphertext
        end
    end
end
```

As we can see, the algorithm iterates recursively through all candidates of C. For every candidate, it reads the stored output sequence $X = C_i[SEQ]$ and then initiates another multi-threaded loop to iterate through every candidate for $R1_{init}$, $R1_j = 1...(2^p - 1)$. The looped function is called **match_R1()** in the source code. The function first creates the output sequence for the clocking LFSR, $Y =$ **lfsrgen(**$R1_j$**)**, then decimates the $R2$ candidate sequence and encrypts the known plaintext to produce a ciphertext, Z. When this is done, the only thing that remains to do is to determine whether or not the intercepted ciphertext matches the newly generated one.

A problem that may appear, especially when analysing shorter sequences, is *collisions*. A collision occurs when a multiple keys, when encrypting the same plaintext, produce the same ciphertext. This in turn results in the program returning a *false positive* combination of $R2_i$,$R1_j$ where $R1_j \neq R1_{init}$ or $R2_i \neq R2_{init}$, or both. It is easy to imagine that the shorter the known plaintext m is, the more likely a collision will be as the total combinations of possible output sequences become lower. In the practical application of longer sequences of hundreds or thousands of bits, collisions are increasingly unlikely. As the actual input $R1_{init}$ and $R2_{init}$ is still stored in memory, as mentioned earlier, it is possible to "cheat" in order to detect and analyze the occurrence of collision. In the program, a collision is detected by determining whether the produced sequence $Z =$ CIPHER is

in fact produced by $R1_{init}$ and $R2_{init}$, which would not be possible in a practical approach. For the sake of the experiment, the program iterates through every possible $R1$ initial state in order to be able to quantify the amount of collisions. This metric may be useful for future implementations of a practical approach.

In a practical approach, when $R1_{init}$ is truly unknown, a possible algorithm may simply be to iterate *sequentially* or *randomly* through the states $R1_{j=1...(2^p-1)}$ given the knowledge of which combination of error threshold k and known plaintext length m is likely to not produce collisions. Another, more solid approach is to use the already implemented *collision threshold* value, a, when running the program. This value can be used by keeping the loop running, even though a match has been found, until we reach a threshold of

$$a < \frac{2^p - 2}{\text{matches}}$$

4.3 Evaluation algorithm

The main program discussed in the previous section simulates the BRM-based cryptosystem and performs the attack with a given error threshold for the ARBP search algorithm. As we have seen, the total execution time for the brute-force approach proposed in Section 4.2.4 depends heavily on the resulting candidate set. In research question **RQ1** of this thesis we ask if an *arbitrary optimal decision rule for selecting the set of candidates for the initial state of* $R2$ can be determined. E.g. given a known plaintext of length m and an intercepted ciphertext of length n, is it possible to define an ideal threshold value k in order to produce the minimal set C which satisfies $R2_{init} \in C$ with the total size of $C, |C|$ approaching 1?

In an attempt to answer this, a special evaluation algorithm was developed. The algorithm simply aims to execute the main program multiple times with dynamically set input parameters, depending on the results from the previous run. The algorithm is implemented in **Golang** as a *wrapper* around the main program and the source code can be found in Code listing A.5 in Appendix A.

The algorithm executes dynamically, in a *seeking* manner, in order to find the best settings for the given input data. A snippet of the code is given in Code listing 4.8 for explanatory purposes. Some parts have been omitted for brevity (marked by ...).

Code listing 4.8: Seeking algorithm for analysis

```
for m <= stop_m {
    n=2*m
 bottom := 0
 var resmap = make([]int, x-1)
 var i int = int(float32(m)/float32(k))

 SeekErrors:
  for true {
   if resmap[i] == 1 {
    i++
    continue SeekErrors
   }
   search_start := time.Now()
   cmd := exec.Command("./main", strconv.Itoa(deg), strconv.Itoa(x), strconv.Itoa(n
       ↪ ), strconv.Itoa(i), strconv.Itoa(r1), strconv.Itoa(r2), strconv.Itoa(
       ↪ col_accept), strconv.Itoa(cpus))
...
   if res >= 1 {
    status = "invalid_no_r2"
    fmt.Printf("[m=%d,n=%d,k=%d] FAILED: Set of %d/%d contains no actual R2STATE...
        ↪ \n", x, n, i, cands,max)
   } else if res == -1 {
    status = "invalid_zero_set"
    bottom = 1
    fmt.Printf("[m=%d,n=%d,k=%d] FAILED: Zero candidates... \n", x, n, i)
   } else if res == -2 {
    status = "invalid_full_set"
    fmt.Printf("[m=%d,n=%d,k=%d] FAILED: Too many candidates... \n",x, n,  i)
   } else if res == -3 {
    status = "invalid_collisions"
    fmt.Printf("[m=%d,n=%d,k=%d] FAILED: Set of %d/%d contains collisions... \n",x,
        ↪  n, i,cands,max)
   } else if res == 0 {
    status = "valid"
    fmt.Printf("[m=%d,n=%d,k=%d] SUCCESS: Set of %d/%d is valid!\n",x, n, i,cands,
        ↪ max)

   }
   resmap[i] = 1
   logline = fmt.Sprintf("%d,%d,%d,%d,%s,%d,%d\n",x,n,i,cands,status,duration,cpus)
   writeLog(fname, logline)

   if res >= 0 && bottom == 0 {
    i--
   } else if res == -2 && bottom == 0 {
    i--
   } else if res >= 1 {
    i++
   } else if res == -1 {
    i++
   } else if res == -2 {
    break
   } else if res == -3 {
    break
   }
  }
 m=m+20
}
```

The program takes as its input all the same parameters as the main program, as listed in the beginning of Section 4.2. In addition, a value **stop_m** is set, which tells the program where to stop the analysis cycle. In the supplied code, the **m** variable is increased by 20 for each iteration, which was chosen for the experiments in this thesis. An initial error threshold i is also determined based on the input m and the initial error threshold factor k, $i = \frac{m}{k}$ where k instead of being an absolute value of the amount of errors accepted, represents a ratio of the bit sequence length m (represented by the temporary variable x in the code. The number of vCPUs, which determines the size of the aforementioned thread pool, are automatically determined by the simple Go-function given below.

Code listing 4.9: Automatic detection of cpus

```
1 cpus := runtime.NumCPU()
```

The output of the evaluation algorithm is a CSV-formatted log file which is written to in an iterative manner while the algorithm is executing. A sample of a log file is shown in Code listing 4.10 with headers shown in the first row. The values stored in the file are the input values m, n and k and the following resulting output values

- **#cand** - number of candidates in C_i.
- **status** - a status word indicating the validity of the set.
- **runtime** - the time taken, in nanoseconds, to execute the full algorithm (including the brute force of $R1$).
- **#cpus** - the amount of vCPUs available to the program. This is essential in order to create context to the execution time, when comparing results from different hosts.

The degree, p, the known plaintext length m and the initial states $R1_{init}$,$R2_{init}$, along with the chosen maximum n value are all stored in the filename. The collision acceptance threshold, a is also shown in the filename, although it is always set to $a = 0$ in the experiments performed in this thesis. The filename is stored in the format $p\_R1_{init}\_R2_{init}\_m\_mmax\_a.log$.

Code listing 4.10: Sample of CSV-formatted log file

```
1 m,n,k,#cand,status,runtime(ns),#cpus
2 38,77,12,1977,valid,8637,16
3 38,77,11,1675,valid,6213,16
4 38,77,10,983,valid,4907,16
5 38,77,9,463,invalid_no_r2,23132,16
6 38,77,8,118,invalid_no_r2,27642,16
7 38,77,7,79,invalid_no_r2,24260,16
8 38,77,6,0,invalid_zero_set,21678,16
9 38,77,13,2047,invalid_full_set,24122,16
```

As we can see in the sample, the algorithm starts with the settings:

$$m = 38, n = 77, i = k = \frac{m}{3} = \frac{38}{3} \approx 12$$

As this returns a valid set of $|C| = 1977$ candidates out of $2^p - 2 = 2046$ possible, the algorithm reduces the error threshold i in the next iteration to see if it is

possible to create a smaller, valid set. This is true for $i = 11, i = 10$, but when $i = 9$ we see that the "cheating" mechanism determines that in the set of $|C| = 463$, the actual $R2_{init}$ is not present. The algorithm continues reducing i until it reaches the "bottom", the zero-set, which is indicated by setting the **bottom** variable to **1**. When the zero-set is found, the algorithm needs to determine at what point it reaches the opposite end, the full-set, when the number of candidates equals $2^p - 2$. When both the "top" and "bottom" has been found, m is increased and the seeking process repeated until $m =$ **stop_m** is reached.

4.4 Task separation

The attack described in this thesis consists of multiple stages, as described in Section 4.2. Step 2 (the candidate generation for $R2$) and step 3 (the clock sequence reconstruction) are especially time consuming. Both in a practical attack and when researching this topic, it can be useful to separate these tasks. For example, a researcher may want to generate one or multiple candidate sets with different input settings and then run multiple instances of the clock sequence reconstruction at a later stage. This is solved in the program by saving the candidates from step 2 in the **.cand**-files explained in Section 4.2.3. Furthermore, the program can be compiled with the **-DREADCAND** option through the **make read**-command on the hosts that will read the files. This option changes the program to expect an input file instead of generating the set. Note that the program still needs to given all the input parameters, matching that of the **.cand**-file, except for the error threshold k, which is only relevant for the ARBP-process which is now skipped. This could be improved upon in the future by having the program parse the correct settings from the filename of the input file, but is not important at this stage of the research.

4.5 Cloud orchestration

As the previously described set of programs have been designed with the possibility of being load distributed over multiple hosts, it is possible to run it with high capacity in the cloud. While the full potential of cloud computing still remains to be exploited for this particular application, a function for load distributing the analysis program in the cloud has been implemented.

This works by having the program read "jobs" from a shared job-file which is stored in a storage that is common for all the running hosts. Specifically for the tests that have been run for this thesis, this is implemented in AWS by mounting a shared *Elastic File System* (EFS)[46] on all tasked hosts, which contains the job-file. The EFS is AWS' equivalent of the more commonly known *Network File System* (NFS). The jobfile is formatted in the following manner:

Code listing 4.11: Sample of CSV-formatted job file

```
p,m,n,k,stop_n,R1init,R2init,a,mode
...
11,32,801,3, 900,100,1000,0,or1
11,32,901,3,1000,1000,1000,0,or1
16,32,64,3,100,1000,1000,0,or1
16,32,100,3,200,1000,1000,0,or1
...
```

The running instances reads the top line of the file as its input parameters, deletes the line and writes the file back to the shared storage. In this procedure there remains a risk of a *race condition*, however it is deemed unlikely, as the jobs takes long time to execute and the job file is not accessed very often. However, this may have to be taken into account if starting up multiple instances at exactly the same time, or if it is shared among a high number of hosts simultaneously.

As mentioned in Section 4.1.4, we utilise the *spot instance* function in AWS by automatically launching instances according to the template defined in Code listing 4.12.

Code listing 4.12: AWS Launch template Bash script

```
#!/bin/bash
apt-get update
apt-get install -y libgmp-dev build-essential nfs-common golang-go
cd /home/ubuntu
su ubuntu -c "git␣clone␣-b␣evaluate␣https://github.com/philedem/BRM_Code"
cd BRM_Code/evaluation/src
mkdir data
mount -t nfs4 -o rsize=1048576,wsize=1048576,hard,timeo=60,retrans=2,noresvport \
fs-xxxxxxxx.efs.eu-north-1.amazonaws.com:/ /home/ubuntu/BRM_Code/evaluation/src/
    ↪ data
chown ubuntu data
chgrp ubuntu data
su ubuntu -c "go␣run␣analysis.go"
```

As the script shows, it pulls the whole software package related to this thesis from its Github repository. It also has to install the programs dependencies, as the host (Ubuntu 20.04 LTS) operating system is a completely clean install on initial launch. Next the **mount** command is run to mount the shared EFS to the operating system. Finally the analysis program is run with the **go run** command which self-resolves any Golang dependencies, compiles and immediately runs the program. As the analysis program itself first compiles the main program **main.c** with the provided *mode* (the last value in the job-line), this procedure should be easily adaptable to many different Linux and UNIX based platforms.

Chapter 5

Results

In this chapter the results from the experimentation process will be presented. The first part briefly presents the results of the initial qualitative phase, where the software implementation was developed. The rest of the chapter shows the results of quantitative testing of the software in different modes of operation. Some of the results will be briefly commented upon in this chapter, however the overall results are further discussed and analysed in Chapter 6.

5.1 Software development

As mentioned in Chapter 3, the project has followed an exploratory mixed-methods design. As such, the first phase entailed the qualitative data collection and analysis of the existing and evolving software. This was performed alongside several iterations of the development cycle. The results from this phase are represented as the software design itself as presented in Chapter 4, derived from what has been implemented before and the theories found in the literature. While the performance aspect of the software implementation has not been the main focus in this thesis, a best-effort approach has been used in order to make the program as efficient as possible under the constraints of using the C programming language with the GMP Library in CPU. The most important details of the program performance is presented in the subsequent section of this chapter.

The results of the qualitative phase also includes some preliminary results derived from repeated experimental testing with different values of static variables throughout the development iterations. One such variable is the chosen error ratio, related to the length of the known-plaintext. This metric is an important variable which marks the starting point for the analysis program which was described in the previous chapter. According to theory, if given a long enough bit sequence m the amount of decimated bits should be $k \approx \frac{m}{2}$. However, it was found that for the AND and OR1 algorithms the best starting point is closer to $k \approx \frac{m}{4}$. For the more constrained algorithms OR2 and OR3, the error ratio was found to be best somewhere in between $\frac{m}{3} < k < \frac{m}{2}$. This is likely due to the fact that with fewer constraints, the fewer *false positives* are produced. This should then in turn lead

to a better selection of the $R2$-candidate set, C. This theory will be tested in the quantitative analysis part of this chapter.

Through the testing done on the finished multi-threaded software it is apparent that even when running the algorithms on the full 172 vCPUs allocated by AWS, running it with polynomial degrees higher than $p = 16$ is still highly ineffective. Therefore, the quantitative experiments will primarily focus on the testing of $p = \{11, 16\}$ with the polynomials presented in Section 4.2.1. A few tests may however be done with $p = 20$ for comparison. It has also been determined that sequence lengths of $m = 100...300$ appear suitable for gaining enough data for quantitative analysis over the chosen polynomial values. While a realistic attack may involve much longer sequences, depending on the known plaintext, the range determined here is likely in the lower end of the spectrum. For example, 300bits equal $\frac{300}{8} = 37.5$ ASCII formatted characters or just fractions of a second of a 8kbps telephone call. Still, the results found in this thesis should give an indication of the potential of attacking longer sequences.

5.2 Performance

In this section the results related to the performance of the cryptoanalytical attack are presented. First, the results of changing from *Shift-AND* to *Shift-OR* are presented. Next, we look at the performance improvement pertaining to the implementation of multi-threading in the main program. Finally, the overall performance of the full brute-force algorithm to reconstruct $R1$ is presented. The execution time measurements in these experiments are based on a real-clock offset, by calculating the difference in start time and end time on the system running the tests. This method was chosen because the measurement based on CPU operations R_{ops} become highly impractical when measuring multi-threaded programs. All execution times are presented displayed in seconds (s) in the thesis and in nanoseconds (ns) or microseconds (μs) in the attached raw results in Appendix B.

5.2.1 ARBP performance

One of the contributions in this thesis is the implementation of the *Shift-OR* algorithm into the software. The program can still however be compiled with the *Shift-AND* algorithm as previously implemented in [3] by compiling it with the **make and**-command in place of the standard **make** which defaults to *Shift-OR*.

The performance comparison of the *Shift-AND* and two variants of the *Shift-OR* implementation is shown in the graphs below. The measurements were made on a 2,4GHz Intel Core i9 MacBook Pro. The experiment was deliberately done in single-core mode by disabling the implemented multi-threading in order to get a more realistic comparison of the three implementations. The method of instrumentation was done by inserting a *real-clock* time measurement in the program around the section being examined, as shown in Code listing 5.1. The input parameters and settings used in the experiment is shown in Table 5.1.

Table 5.1: Input parameters for the experiment.

deg	{11,16}
m	100...300 (interval of 20)
n	$2 \times m$
k	$m \div 4$
$R1_{init}$	100(dec) = 00001100100(bin)
$R2_{init}$	100(dec) = 00001100100(bin)

Code listing 5.1: Generic example of instrumentation

```
1 struct timeval start, end;
2 gettimeofday(&start, NULL);
3 ...
4 // [measured code block]
5 ...
6 gettimeofday(&end, NULL);
7 long seconds = (end.tv_sec - start.tv_sec);
8 long micros = ((seconds * 1000000) + end.tv_usec) - (start.tv_usec);
```

One important note is that the setting of $k = \frac{m}{4}$ is not the correct setting for every algorithm and will in many cases create an empty or otherwise invalid set C. The resulting C is however of no importance for this performance measurement, as the most important aspect of performance comparison is that all the algorithms perform the same amount of operations. As we have seen from the theory and implementation earlier, all candidates for $R2_{init}$ are tested in all cases, so the amount of operations will always be $O(n \times (2^p - 2) \times k)$.

In Figure 5.1, the dotted line represents the *Shift-AND* measurements (AND), the dashed line the equivalent *Shift-OR* implementation (OR1), and the solid line the additionally constrained *Shift-OR* implementation (OR2).

As we can immediately see, the OR2 execution time is significantly higher even though their implementations use the exact same amount of operations in their respective status word update functions, as was shown in Code listing 4.5 and Code listing 4.6. This effect is attributed to the fact that the latter contains an extra **mpz_lshift()**-operation, which as the name suggests, left-shifts the input value. The extra left-shift operation is due to the addition constraint in the *substitution* operation which requires us to OR the left-shifted value of the previous status word with the complement of **B[i]**. The GMP Library has no native implementation of the left-shift operation, therefore a function for this was implemented in [3]. Within this function, several calls are made to different GMPLib-functions, making it a computationally costly function to use. This could possibly be improved upon in future studies but has not been a focus in this thesis.

The execution times of the equivalent implementations of AND and OR are very similar. Although, while barely discernible in the graph, the data shows that *Shift-OR* performs slightly better in this experiment. As the experiment was run on an operating system that also has to handle other tasks, other system processes may have introduced a slight bias to either one even though the systems

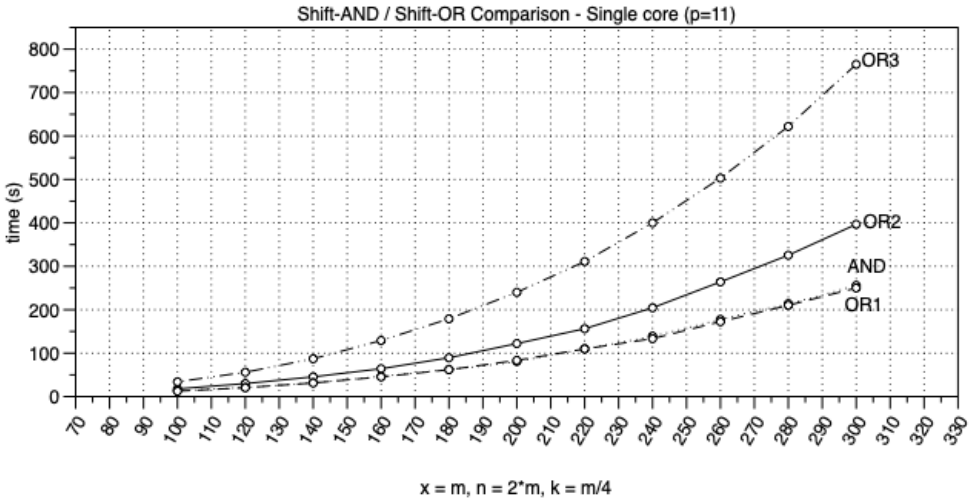

Figure 5.1: The execution time comparison of the *Shift-AND*, *Shift-OR* and *Shift-OR* with additional constraints when $p = 11$.

runtime conditions were kept as similar as possible for the two experiments. The apparent equivalence of the two does however emphasise the importance of the previous observation that the **mpz_lshift()**-operation is very costly in its current implementation. It is also noteworthy, although quite evident from the previously discussed theory, that execution times are non-linear in nature due to the amount of allowed errors k has to increase as m is increased, as illustrated in the formula

$$O(n \times (2^p - 2) \times k) \longrightarrow O((m \times 2) \times (2^p - 2) \times (\frac{m}{i}))$$

where i is the temporary ratio of k, initially set to 4 in this experiment.

The same test was also run for the comparison of *Shift-AND* and the equivalent *Shift-OR* algorithm with $p = 16$ as shown in Figure 5.2. As this is much more time consuming to run in single-core mode, only the two fastest version were tested for performance. We can however assume, based on this, that the relationship would be similar to what was shown in Figure 5.1.

From the Figure 5.2 we can observe that the execution times with $p = 16$ are much higher than that of the results observed when $p = 11$, as expected. When we look at the equation below, we would expect the $p = 16$ execution time to be approximately 32 times slower.

$$(2 \times 100) \times (2^{11} - 1) \times \left(\frac{100}{4}\right) = 200 \times 2047 \times 25 = 10235000$$

$$(2 \times 100) \times (2^{16} - 1) \times \left(\frac{100}{4}\right) = 200 \times 65535 \times 25 = 327675000$$

$$\frac{327675000}{10235000} \approx 32$$

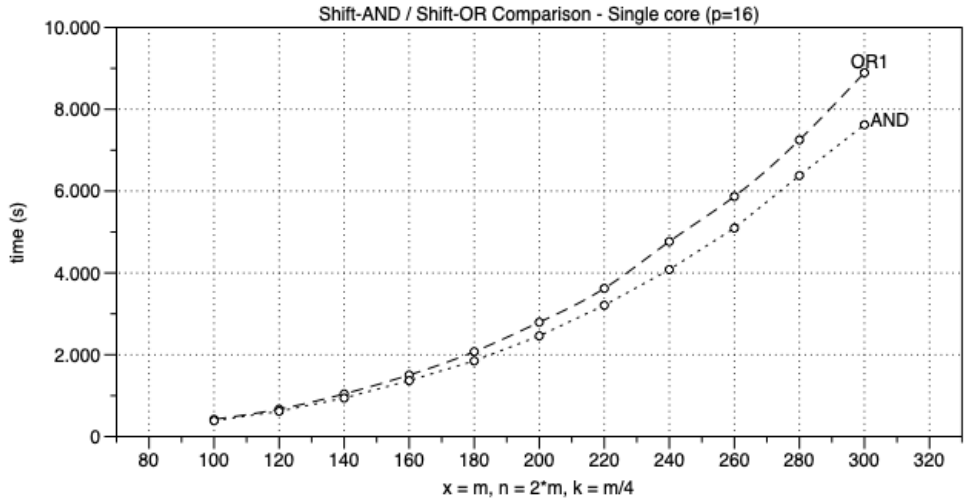

Figure 5.2: The execution time comparison of the *Shift-AND* and *Shift-OR* (OR1) when $p = 16$.

The equations below show the measurement comparisons of $p = 11, p = 16$ for some selected values of m.

$$t(100)_{11} = 12.42s, 12.42s \times 32 \approx 397.44s(\text{OR1})$$
$$t(100)_{16} = 421(\text{OR1})$$
$$\frac{421}{12.42} \approx 34$$
$$t(200)_{11} = 83.41s, 83.41s \times 32 = 2669.12s(\text{OR1})$$
$$t(200)_{16} = 2794(\text{OR1})$$
$$\frac{2794}{83.41} \approx 33$$
$$t(300)_{11} = 250s, 250s \times 32 = 8000s(\text{OR1})$$
$$t(300)_{16} = 8895(\text{OR1})$$
$$\frac{8895}{250} \approx 35$$

As we can see, the prediction of the algorithm being 32 times slower when running with $p = 16$ holds within an acceptable degree. The small deviation is attributed to the fact that the complexity only factors in the ARBP process of the program. Some parts, such as generation of LFSR output sequences will also contribute to an overall slower execution time.

5.2.2 Multi-threading

As we have seen in the previous section, the candidate generation algorithm (ARBP) is computationally costly to run. While there are surely ways to improve this by fine-tuning the code or implementing it directly in machine code, the most impactful way to improve the overall efficiency is probably by utilising multi-threading in CPUs. In theory, the original complexity of the ARBP process $O(n \times (2^p - 2) \times k)$ should be reduced to

$$O\left(\frac{n \times (2^p - 2) \times k}{c}\right)$$

where c is the amount of cores, or vCPUs, made available to the program.

We compare the results in the previous section with the multi-threaded version in the graphs below. The following tests were run on a dedicated virtual host in Amazon Web Services (AWS), on a **c5a.4xlarge**-instance. This instance type provides 16 vCPUs and 32GB of memory.

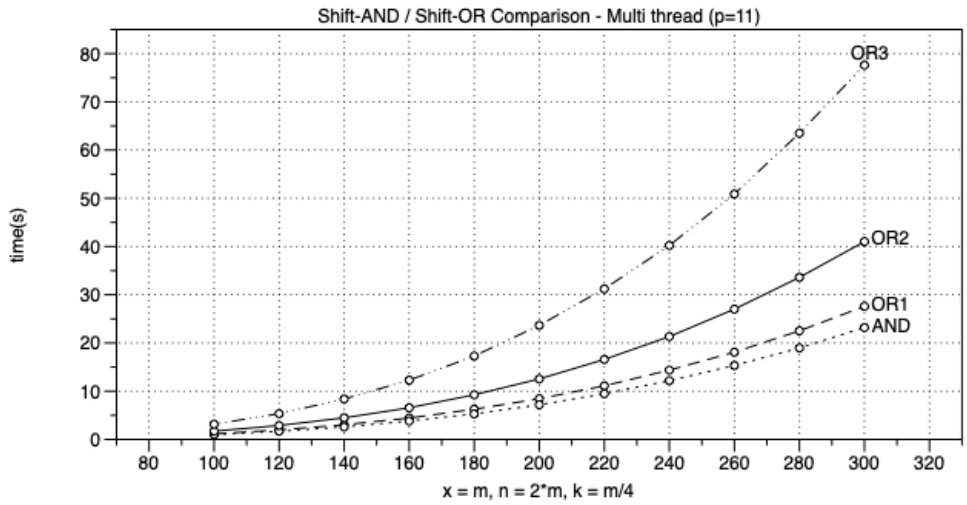

Figure 5.3: The execution time comparison of the *Shift-AND*, *Shift-OR* and *Shift-OR* with additional constraints when $p = 11$ with Multi-threading enabled.

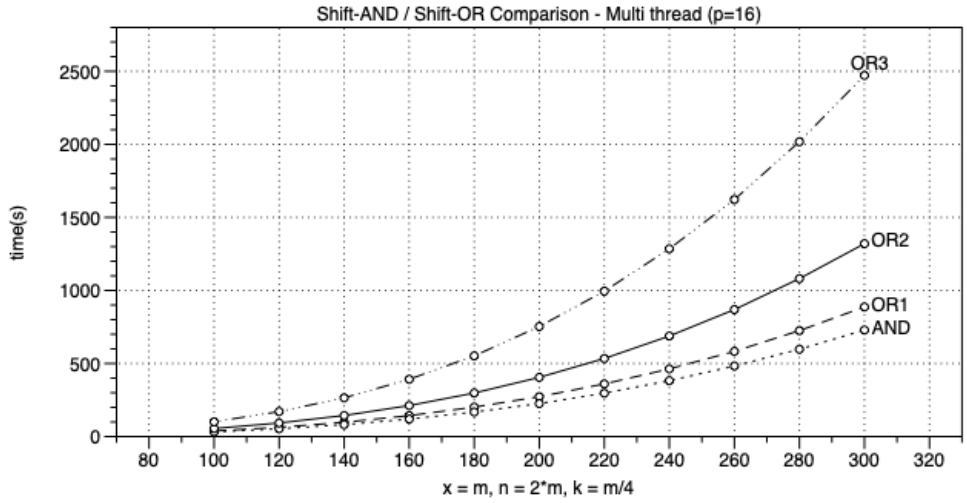

Figure 5.4: The execution time comparison of the *Shift-AND*, *Shift-OR* and *Shift-OR* with additional constraints when $p = 16$ with Multi-threading enabled.

As we can see in Figure 5.3 and Figure 5.4 the execution times have been drastically reduced. In the equations below, the single-core (denoted SC) measurements are compared with the multi-threaded (denoted MT) measurements for selected m values.

$$t(100)_{16} = 421s \quad \text{(OR1 SC)}$$
$$t(100)_{16} = 38s \quad \text{(OR1 MT)}$$
$$\frac{421s}{38s} \approx 11$$
$$t(200)_{16} = 2794s \quad \text{(OR1 SC)}$$
$$t(200)_{16} = 273s \quad \text{(OR1 MT)}$$
$$\frac{2794s}{273s} \approx 10$$
$$t(300)_{16} = 8895s \quad \text{(OR1 SC)}$$
$$t(300)_{16} = 887s \quad \text{(OR1 MT)}$$
$$\frac{8895s}{887s} \approx 10$$

The results show that the total average performance increase is around a factor of 10. This is somewhat less than what was expected from a 16 core dedicated host, but still a significant improvement from that of the single core implementation. The deviation from the theoretic performance is likely because of a bias introduced by the virtual operating systems system-tasks interfering with some of the programs threads. It could also be caused by the operation of the virtual cores allocated by AWS. As these are part of virtual resource pools some of the cores may be affected by outside factors such as the physical CPUs operating temperature or

power available.

5.3 Candidate generation

In this section, the results from the evaluation of the four different ARBP implementations are evaluated. Recall Section 2.4.4 on Statistical attacks. The process of generating the set C is essentially generating the output sequence of $R2_{i=1...n}$ of every possible initial state of the $R2$ LFSR and deciding on one of the two hypotheses :

H_0 The intercepted sequence **cannot** be generated by the tested initial state.
H_1 The intercepted sequence **can** be generated by the tested initial state.

The decision is based on the selected approximate search algorithm being used and the error threshold parameter, k. An additional parameter a was also introduced in this thesis to measure the amount of *collisions* when testing whether ciphertext collisions occur when testing the output sequences of R1 and R2. As explained earlier, while the more complex implementations of the *Shift-OR* algorithm are slower in execution time, they may make up for it by creating smaller *valid* candidate-sets C. Recall Chapter 2; The optimal set C is the minimal size $|C|$ which also contains the actual $R2_{init} \in C$ and ideally $|C| = 1, C_i = R2_i = R2_{init}$.

The preliminary experiment for determining the ideal settings of k compared to the values m, n was run by a slightly modified version of the program, which stops the program after the candidate-set C has been generated. The modified version excludes the check for possible *collisions* when running the full attack, as this will not be applicable before the clock reconstruction is introduced in the next section of this chapter. The tests were run simultaneously over multiple AWS hosts of the type **c5.18xlarge**, providing 72 vCPUs and 144GB of memory, each. The execution of the experiment was orchestrated through the *jobs*-file, as explained in Section 4.5. The *seeking* algorithm to dynamically change the value of k is explained in Section 4.3. The initial input parameters are given in the table below.

deg	{11,16}
m	100...300 (interval of 20)
n	$2 \times m$
k	$m \div 3$ initially
$R1_{init}$	100(dec) = 00001100100(bin)
$R2_{init}$	100(dec) = 00001100100(bin)

Table 5.2: Input parameters for the experiment.

In the first experiment the comparison of the three *Shift-OR* algorithms and the resulting candidate set size, $|C|$, is tested. The *Shift-AND* mode is excluded from this experiment, as we already know from the theory that it gives the same results

as the OR1 version of the *Shift-OR* implementation.

The graphs in Figure 5.5 and Figure 5.6 display the value $|C|$ on the Y-axis and the m-value in intervals of 20 on the X-axis for $p = 11$ and $p = 16$, correspondingly. Only the valid sets are shown as dots on the Y-axis, while any invalid sets are hidden. Each point is marked with its corresponding error threshold value k and a line is drawn through the minimal, and most ideal sets.

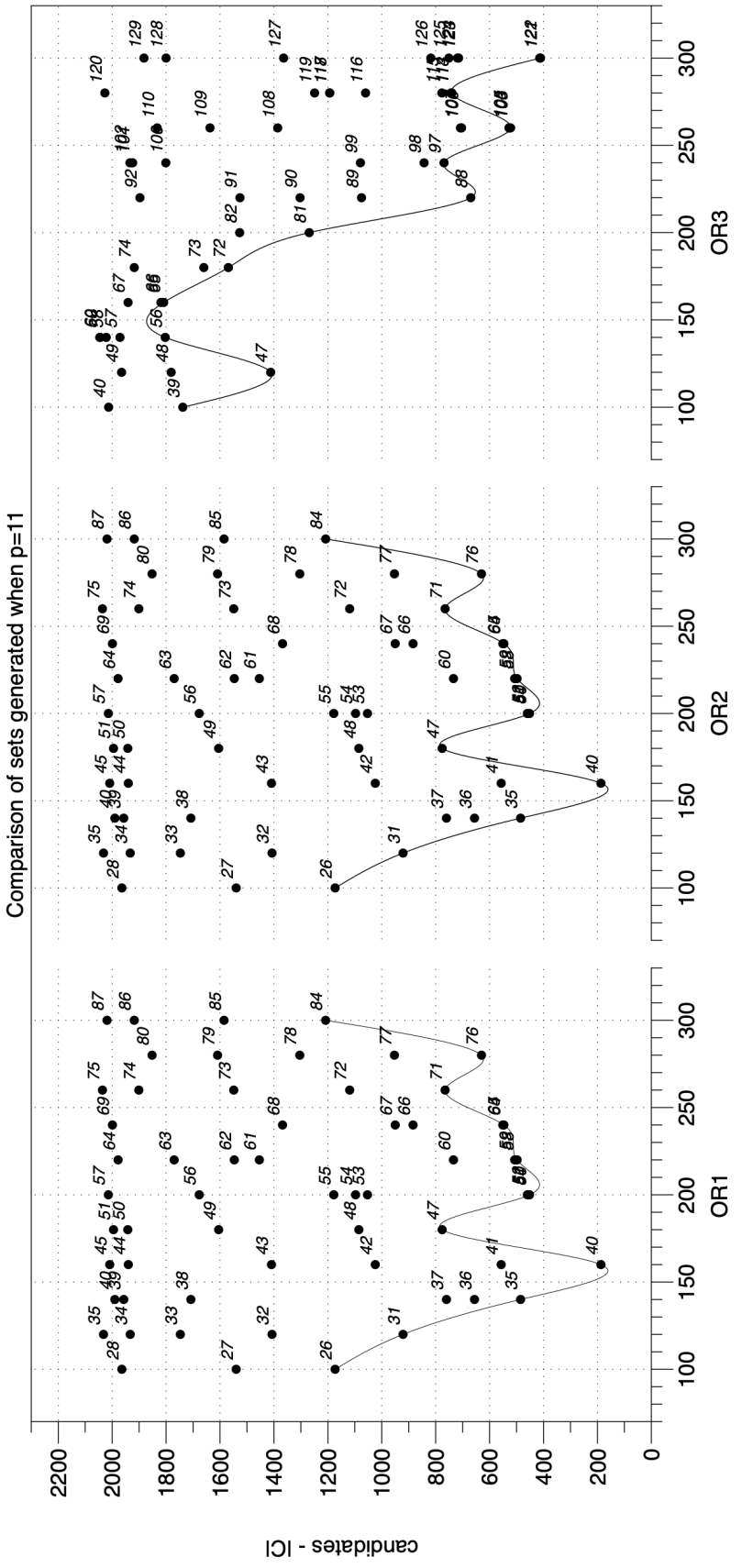

Figure 5.5: A comparison of the generated sets of $R2_{init}$ candidates with corresponding k values when $p = 11$.

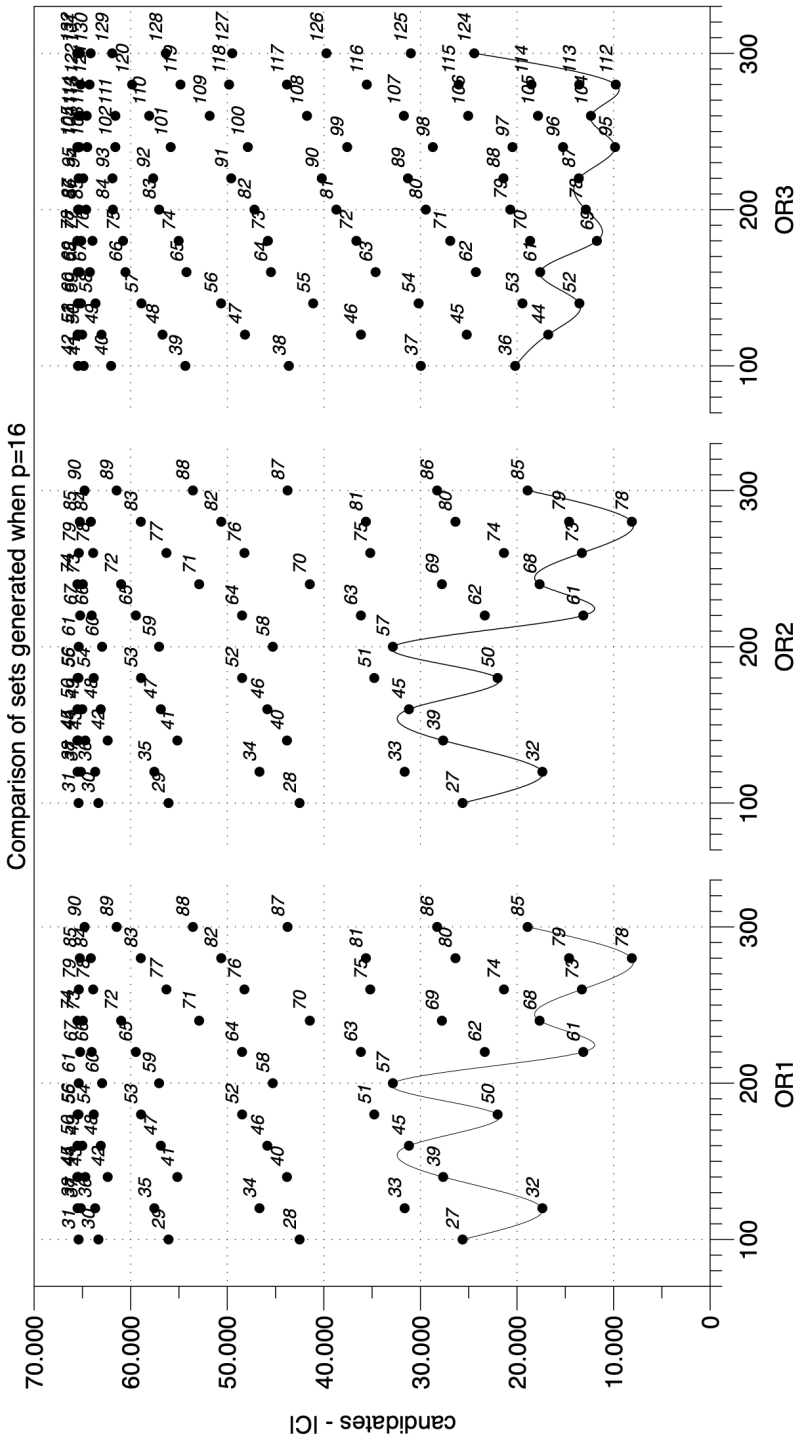

Figure 5.6: A comparison of the generated sets of $R2_{init}$ candidates with corresponding k values when $p = 16$.

The first observation we can make from the graphs is that the OR1 and OR2 graphs are the same, while OR3 differs significantly. As the constraints introduced in OR2, compared to OR1 only affect the special case where a substitution occurs after a match and we are analysing relatively small amounts of errors, this may be natural. This observation will be further addressed in the next chapter. However, as a result of this, in the further graphs OR1 is omitted and only OR2 and OR3 are compared.

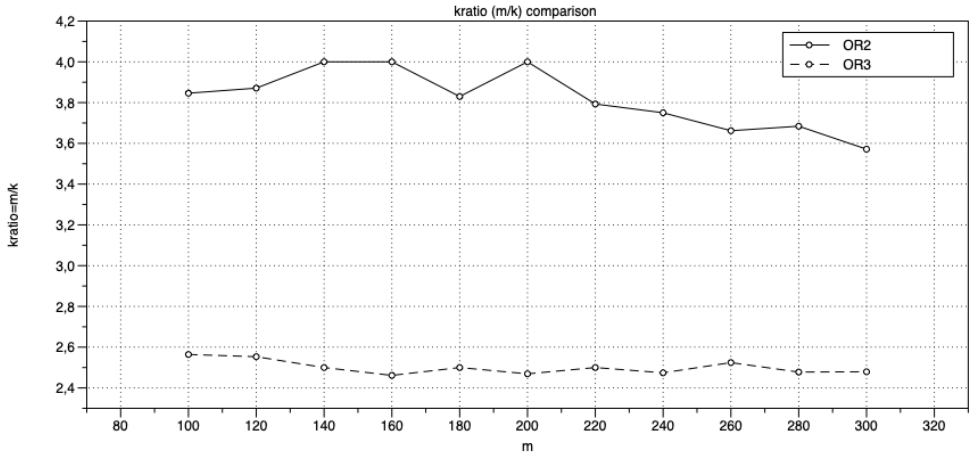

Figure 5.7: A comparison of the k_{ratio} m/k for $p = 11$.

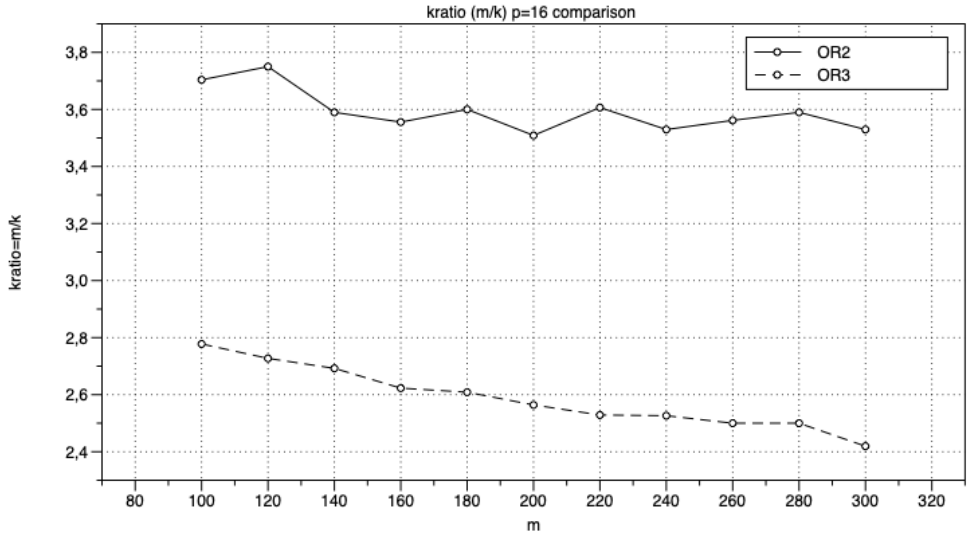

Figure 5.8: A comparison of the k_{ratio} m/k for $p = 16$.

The graphs in Figure 5.7 and Figure 5.8 shows the compared ratio $k_{ratio} = \frac{m}{k}$ for OR2 and OR3. These values show the relationship between k and m, in order to give an indication of how many errors need to be accepted by the ARBP algorithm

in order to produce the optimal set C. As we can see from the equation, the lower the value of k_{ratio}, the more errors need to be allowed. As we can see in the graph, the OR2 algorithm is much more permissive of errors. This is as expected as the more permissive constraints open up for more *false positives*, which should be showing through the C_{ratio} metric.

The next set of graphs show the relationship $C_{ratio} = \frac{|C|}{(p^2-2)}$. This relationship is a measure of the amount of candidates selected according to the H_1 hypothesis compared to the total possible candidates. Ideally this number is as low as possible, which indicates a more ideal set for the clock sequence reconstruction phase which is the next step of the cryptanalysis. The graphs show the C_{ratio} value for the minimal valid sets measured.

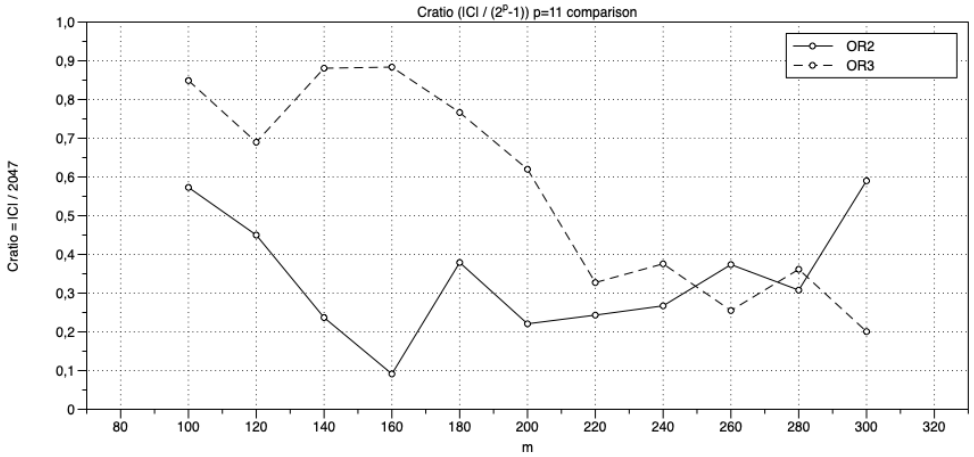

Figure 5.9: A comparison of the $C_{ratio} = |C|/(p^2-2)$ for $p = 11$.

Table 5.3 and Table 5.4 shows the average results of the candidate set generation. The table shows the average $|C|$ and the average values of k and $|C|$ with relation to the tested m-values for OR2 and OR3, respectively.

Table 5.3: The average values of C_{ratio}, k_{ratio} and $|C|$ for valid sets over the tested space $m = 100...300$ using the OR2 algorithm.

p	avg($\|C\|$)	avg(C_{ratio})	avg(k_{ratio})	CI(k_{ratio}) 95%
11	695	0.33939	3.81887	±0.08591
16	20709	0.31600	3.59314	±0.07400

Table 5.4: The average values of C_{ratio}, k_{ratio} and $|C|$ for valid sets over the tested space $m = 100...300$ using the OR3 algorithm.

p	avg($\|C\|$)	avg(C_{ratio})	avg(k_{ratio})	CI(k_{ratio}) 95%
11	1156	0.56455	2.50033	±0.02007
16	14790	0.22568	2.58796	±0.06467

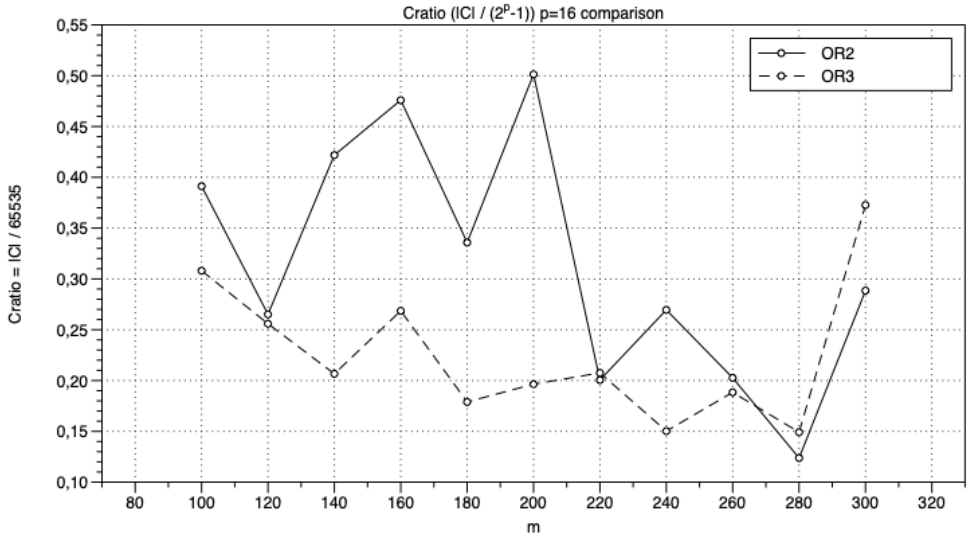

Figure 5.10: A comparison of the $C_{ratio} = |C|/(p^2 - 2)$ for $p = 16$.

As we can see, when measuring by the C_{ratio}-metric the OR3 algorithm outperforms the OR2 algorithm for $p = 16$, but performs worse in $p = 11$. The average k_{ratio} confirms what we saw in the previous graphs; that the OR3 algorithm allows fewer errors than that of OR2. While the C_{ratio}-metric says something about the quality of the selection, the average k_{ratio} is important to analyse which initial setting of k is to be used to increase the chance of generating the optimal set C. These results will be further discussed and analysed in Chapter 6.

5.4 Clock sequence reconstruction

The implementation of the clock sequence reconstruction is a brute-force attack, using the generated candidate set C and testing every possible $R1_{init}$, as explained in Section 4.2.4. The complexity of this attack $O((2^p - 2) \times |C|)$ is a linear relationship of the total possible initial states of $R1_{init} \in \mathrm{GF}(2^p)$ and the total candidates in C, $|C|$. The program executes the attack using multi-threading by letting each thread perform the brute-force procedure for one candidate, C_i.

As mentioned earlier, in the classical *dynamic programming*-approach to cryptanalysis of irregularly clocked stream ciphers it is possible to backtrack through the matrix of errors when generating the candidate set for $R2$ in order to determine the clock sequence. The possibility of a similar approach for the bit-parallel search approach was investigated in the project planning for this thesis. However, such an attack becomes inherently difficult when working with binary operations as most are impossible to reverse without knowing the previous state. An approach to the problem that might work is to keep a separate matrix of transitions (*substitutions*, *deletions* and *matches*) used to reach a valid approximate match. Such an approach may lead to a significant performance improvement to the clock sequence recon-

struction phase at a decent time and memory tradeoff in the candidate generation phase.

5.4.1 Performance

The first performance measurement for the clock reconstruction was the test of execution times when performing a *full* brute force procedure with an arbitrary initial state. A *full* brute force means that it will always test every candidate in C against every possible initial state of $R1$. By doing this the computational complexity will always be the *worst case* $O((2^p-2)^2)$. This differs from the more practical attack where we use the reduced $R2_{init}$ candidate set C and also interrupt the brute force process as soon as a matching ciphertext is found. The full approach is still useful however, if one wants to assess the performance and the possibility of collisions.

For the experiment, we compile the program with the **make read** command, which tells the program to read a previously generated candidate-file (**.cand**) into memory. The goal of the test is to analyse how the performance decreases with higher degrees of LFSR, p and increasing lengths of ciphertext m. The length of the intercepted ciphertext m is an important factor, as the algorithm needs to generate an equally long candidate ciphertext and compare it with the intercepted ciphertext for every pair of $R1_{init}$ and $R2_{init} \in C$.

In order to make the results comparable, "fictional" candidate files have been crafted[1], which contains the full set $|C| = (2^p-2)$ for three different values of p and m. The test was run on a dedicated AWS host of the type **c5.24xlarge**, providing 96 vCPUs and 192GB of memory. The testing parameters were as given in Table 5.5.

Table 5.5: Input parameters for the experiment.

deg	{11,16}
m	{100, 150, 200}
n	$2 \times m$
$R1_{init}$	100(dec) = 00001100100(bin)
$R2_{init}$	100(dec) = 00001100100(bin)

Table 5.6: Brute force performance experiment 1

p	$m = 100$	$m = 150$	$m = 200$
11	1.70s	2.64s	19.96s
16	224.45s	2259.70s	23908.00s

[1]These sets were hand-crafted by the author, as the program normally will discard any full-set result from the ARBP-search procedure. Therefore, the crafted sets used in this experiment are supplied together with the source code of this thesis.

There was also an intention to test this experiment with $p = 20$, however, due to the time required to compute $p = 16$ and the given financial limits given in Section 3.3.1 this was not possible. This may however be an experiment to be considered for future research on the topic.

In order to investigate the results from the previous experiment further, another experiment was constructed in order to perform more detailed instrumentation of the implementation. Recall from Section 4.2.4, each thread i performs the following steps on one candidate R2 output sequence generated by $R2_i \in C$:

- **S1** Generate the output sequence of R1, $R1_{seq}$, given the initial state $R1_j$.
- **S2** Encrypt the known plaintext with $R1_{seq}$ and the tested $R2_i$ to generate a candidate ciphertext Z'.
- **S3** Compare the candidate ciphertext Z' with the intercepted Z ciphertext.

In this experiment the execution time of each of these steps, the total execution time for one iteration (denoted I in the tables) of the test ($R1_j \oplus R2_i = Z' \stackrel{?}{=} Z$) and the total thread (denoted T in the tables) execution time were measured in *microseconds* in a single-core setup on a 2,4GHz Intel Core i9 MacBook Pro. Specifically, the test is implemented by inserting a real-clock time measurement function into the threaded function at the different measurement points, as demonstrated in Code listing 5.1. As the instrumentation itself introduces a few extra operations and may cause a minor increase in execution times, only one element was measured at a time. In this experimentation the $p = 20$ version is also included as the analysis only requires partial execution of the whole reconstruction procedure.

The results in the following tables are presented as the average times (μs) over a sample (at least $> 5\%$) of execution times when $m = \{100, 150, 200\}$ for $p = \{11, 16, 20\}$, respectively.

Table 5.7: Brute force instrumentation times (μs), $p = 11$

m	T	I	$S1$	$S2$	$S3$
100	64706	39	38	2	0.03
150	103076	61	57	3	0.05
200	137497	78	72	4	0.02

Table 5.8: Brute force instrumentation times (μs), $p = 16$

m	T	I	$S1$	$S2$	$S3$
100	2349052	41	41	2	0.03
150	3917884	67	62	3	0.04
200	5062029	89	83	4	0.03

Table 5.9: Brute force instrumentation times (μs), $p = 20$

m	T	I	$S1$	$S2$	$S3$
100	104972172	47	45	1	0.06
150	163396701	71	68	2	0.05
200	217826629	95	93	3	0.03

The first observation we can make from these results is that **Step 1** is the significant factor in the execution times. This step consists of the **lfsrgen()**-function of the program.

Another interesting observation is that **Step 3** (the comparison of the output sequences) tend to be lower with longer sequences. This step consists of evaluating the **mpz_cmp()**-function of GMPLib. However, the measured times being as low as fractions of a microsecond, this is of very little significance when compared to the total iteration time (I) and may be caused by a bias in the system.

We can also see that, as one would expect, the thread-internal iteration times I are approximately linear as m increases. Meanwhile, the total thread times T deviate slightly from the expected linearity. This is likely due to the fact that the CPUs handling of threads may vary due to other factors on the system.

5.5 Practical attack demonstration

Finally, a novelty implementation of the full attack on a BRM-generated stream cipher is demonstrated. This version of the program is compiled without the "cheating" functions used earlier to analyse the program. This means that after the program has simulated the BRM and encrypted the plaintext it has no knowledge of the actual initial states of $R1$ and $R2$. The selected error threshold k is selected based on the findings in Section 5.3 and, contrary to the development version of the program, a simple *seeking* functionality is introduced which increases k if the candidate set is empty and decreases it if it is full. The brute force algorithm is executed in a *sequential* manner (iterating sequentially through the initial states, as opposed to *randomly*.) and will interrupt and report a match as soon as a match $Z' = Z$ is found regardless of this being a *false positive* in form of a ciphertext collision. The program ultimately returns the initial states of $R2$ and $R1$.

As the program has primarily been developed for testing purposes, running the demonstration requires that the **-DDEMO** flag is used when compiling in order to change to demonstration mode. This can be done by issuing the **make demo**-command upon compilation, however if different ARBP search algorithms are to be tested, this has to be manually specified in the *makefile*. Figure 5.11 shows a screenshot of a terminal session executing the practical demonstration with different input parameters.

```
→  src git:(evaluate) ✗ ./main 11 100 200 22 999 640 0 16
[k=23] Empty candidate set. Increasing error threshold
[k=24] Empty candidate set. Increasing error threshold
[k=25] Valid candidate set |C|=121 (of 2047). Starting brute force attack!
         - POSSIBLE MATCH FOUND for R1 init state 999 and R2 init state 640
A match was found in 11258969! Exiting.
→  src git:(evaluate) ✗ ./main 11 100 200 22 1001 640 0 16
[k=23] Empty candidate set. Increasing error threshold
[k=24] Empty candidate set. Increasing error threshold
[k=25] Empty candidate set. Increasing error threshold
[k=26] Valid candidate set |C|=459 (of 2047). Starting brute force attack!
         - POSSIBLE MATCH FOUND for R1 init state 1001 and R2 init state 640
A match was found in 13686478! Exiting.
→  src git:(evaluate) ✗ ./main 11 100 200 22 865 640 0 16
[k=23] Empty candidate set. Increasing error threshold
[k=24] Empty candidate set. Increasing error threshold
[k=25] Empty candidate set. Increasing error threshold
[k=26] Valid candidate set |C|=457 (of 2047). Starting brute force attack!
         - POSSIBLE MATCH FOUND for R1 init state 865 and R2 init state 640
A match was found in 13225340! Exiting.
→  src git:(evaluate) ✗ ./main 16 50 100 10 8650 640 0 16
[k=11] Valid candidate set |C|=2231 (of 65535). Starting brute force attack!
         - POSSIBLE MATCH FOUND for R1 init state 8650 and R2 init state 640
A match was found in 115603604! Exiting.
→  src git:(evaluate) ✗ ./main 16 50 100 8 8650 64000 0 16
[k=9] Empty candidate set. Increasing error threshold
[k=10] Valid candidate set |C|=392 (of 65535). Starting brute force attack!
         - POSSIBLE MATCH FOUND for R1 init state 8650 and R2 init state 64000
A match was found in 14633396880! Exiting.
→  src git:(evaluate) ✗ ./main 16 50 100 8 7294 12462 0 16
[k=9] Valid candidate set |C|=112 (of 65535). Starting brute force attack!
         - POSSIBLE MATCH FOUND for R1 init state 7294 and R2 init state 12462
A match was found in 2381752446! Exiting.
```

Figure 5.11: A screenshot of several runs of the practical demonstration.

Chapter 6

Discussion

In this chapter the results presented in the previous chapter is further analysed and discussed. The overall process and product will also be evaluated here, before the thesis is concluded in the next and final chapter.

6.1 Software development and testing

The development of the software was done with testing and evaluation for the future development of attacks on BRM-based systems as the main focus while delivering a "proof-of-concept" of the full attack sequence. As such, there are likely many improvements that can be made, both in terms of performance, efficiency and application of "best-practices" in the C-programming language. The work on the software involved building on the previously developed software in [3]. A complete rebuild and restructuring of the source code was considered but the fundamental functionality in the existing software was deemed to be solid and most of the helper functions was kept and revised. As we saw through the instrumentation done in Chapter 5 however, the functions **mpz_lshift()** and **lfsrgen()** were found to be a significant factor in increasing processing times of the different algorithms in the program. The high complexity of running evaluation experiments in this thesis has made it difficult to produce enough data for quantitative analysis due to the inherent time constraints and financial limits, as the development phase itself has taken much of the time allocated to the project. However, the software together with the "proof-of-concept" of task separation and cloud orchestration provided in this thesis will likely prove useful for future research on the topic and the algorithm implementations may be adaptable to other platforms, such as FPGA or GPU, in the next iteration.

The major contributions to the software made in the development phase of the work on this thesis has been:

- the software implementation of the *Shift-OR* algorithm (with three different levels of constraints) in the *Approximate Row-wise Bit Parallel String matching*-process which is part of the important $R2_{init}$ candidate set generation.

- the introduction of multi-threading in the candidate generation algorithm.
- the multi-threaded clock sequence reconstruction algorithm.
- altering the output of the main program to provide both sensible output codes and **.cand**-files which enable wrapping the program in other languages.

In addition to these elements a wrapper-program (**analysis.go**) was developed in order to enable and simplify the process of running larger-scale analysis of the software in the cloud. The main programs output is specifically designed to work with such wrappers, and future work based on this software may further build on this, and possibly also in other languages such as *Python*, etc.

Throughout the evaluation process, Amazon Web Services [40] were used to provide dedicated high capacity computing power. By using *spot requests*, as much experimentation as possible has been done within the set financial limits. As AWS only provides a small number of total CPUs allowed in spot instances by default, a request was made to Amazon for an increased limit to at least $2 \times 96 = 192$vCPUs. This was partially approved up to 172 vCPUs. While the experiments constructed and executed though cloud computing in this thesis could have been structured in a better way in order to answer the research questions given in Section 1.5, the possibilities of a practical attack load-distributed over many such hosts should be easy to implement from the supplied source code.

If further research into cryptanalysis of BRM-based stream ciphers is to be undertaken in the future it is highly recommended to assess the possibilities and advantages of implementing the ARBP-search generation algorithm and the clock sequence reconstruction procedure in GPU (Graphics processing unit), for example using Nvidias CUDA (Compute Unified Device Architecture) platform, rather than CPU. There are readily available implementations of GMPLib for CUDA, see for example [47], [48] and [49].

6.2 Performance

As presented earlier in the thesis, several different versions of the *Shift-OR* was implemented in software, based on [2]. As we have seen in Chapter 5, the performance results vary significantly over the four different implemented search algorithms. While the basic *Shift-OR* algorithm (OR1) is of less complexity than the *Shift-AND* algorithm and thus should be theoretically faster, it has been shown that the computational complexity of some of the functions used in the implementation cause it to be slower. Through instrumentation it was shown that the most significant factor of this was the **mpz_lshift()**-function which was implemented in [3], which itself consists of five **mpz_***-function calls.

The most significant improvement made to performance through the work on this thesis has been the introduction of *multi-threading*. It was shown in Section 5.2.2 that the total running time of the program was reduced by approximately a factor of 10 when running with $p = 16$ on a 16 vCPU dedicated virtual host

when compared to the equivalent single-core experiment. As has been shown in this thesis, the program is easily scalable when running in the cloud which makes it fully achievable to reach the same performance levels as was achieved in FPGA for the candidate generation algorithm (26-346 times faster)[3] at a reasonable cost.

6.3 Candidate generation

The process of generating the candidate set, C, is essential in the proposed attack on BRM-based stream ciphers. The total complexity of the brute-force attack, $O((2^p - 2) \times |C|)$, heavily depends on the size $|C|$ of the generated candidate set C. The best case complexity of the clock sequence reconstruction is $O((2^p - 2) \times 1)$ (C contains only the actual initial state of $R2$) and the worst case is $O((2^p - 2)^2)$ (C contains all initial states), as was tested in the experiment in Section 5.2.1. As such, a large portion of the research in this thesis has been dedicated to improving optimisation of the algorithm by the introduction and testing of OR2 and OR3 and tuning the parameters used in the ARBP-process.

As have been discussed in the theory, the additional constraints introduced in the OR2 version of the *Shift-OR* algorithm eliminates the possibility of a substitution occurring if a match occurred in the previous row of the error matrix. From the experiments done in this thesis, very few false positives stem from the additional constraint introduced in OR2 compared to OR1. As such, the added complexity introduced by OR-ing the shifted status word with the complement of the bit-mask may not be worth it in most cases. One possible bias here is that the algorithm has not been tested over a large set of different initial-state combinations and perhaps most importantly not for higher degrees of LFSR (over 30 bits).

The OR3 algorithm implemented in this thesis introduces even further constraints into the *deletion*-part of the algorithm. These constraints eliminate the possibility of a deletion occurring at the end, and also the possibility of multiple deletions occurring in a row. These constraints are coherent with the operation of the BRM stream cipher system. It has been shown that the use of this algorithm effectively reduces the $R2$ candidate set. The drawback is the added complexity, which makes the overall performance slower.

It is fair to say that more analysis (both qualitative and quantitative) has to be done on determining the optimal decision rules and input parameters to the candidate generation algorithm. While it was the intention with this thesis to do quantitative analysis of larger data sets for many different input values, the inherent time complexity of the problem, the timeline and financial limits prevented this. It is however the hope of the author that the algorithms and instruments developed through the work on this thesis will provide future researchers with the necessary tools to further explore this topic in order to further improve the candidate selection process.

6.4 Clock sequence reconstruction

As mentioned in Section 4.2 the algorithm has been implemented with "cheating"-operations in order to evaluate the performance and efficiency of the program. One aspect of this was to evaluate the possibilities of ciphertext *collisions* (a set of incorrect $R1$ and/or $R2$ that produces the correct ciphertext) by evaluating whether a reported ciphertext match was generated by the actual initial states. However, the probability of such collisions occurring are very low when operating on longer bit sequences ($m > 50$), which is normally the case. In the more realistic scenario where there is no knowledge of the initial states of $R1$ and $R2$, the behaviour can be changed to count matches, regardless of the matches being confirmed hits or possible collisions. If the program ends up with only one matching ciphertext it is likely to be the correct one. If the program reports more than one match, collisions have occurred and the program may have to be run again with different settings. To reach the optimum performance, the program can also be set to interrupt the process upon the first match, as was shown in the practical demonstration. This may however result in a *false positive result*.

The clock sequence reconstruction algorithm proposed in this thesis requires the whole ciphertext of length m to be generated for every pair of $R2_{init} \in C, R1_i$, which is very time consuming. One possible improvement that could be introduced here is by sequentially evaluating the generated bit sequence through a *fast-filtering* function as explained in [50]. This may provide a significant improvement, especially for longer known-plaintext sequences, $m > 1000$. However, as we have seen in Tables 5.7-5.9, the **lfsrgen()**-function (S1) accounts for approximately 99% of the execution time of each iteration (I), which may be sensible to improve upon first. We can also see from the results that the execution time increases approximately linearly compared to m. Thus, the implementation of filtering, which possibly would require running this function multiple times for every iteration may be counter-productive unless **lfsrgen()** is rewritten to be more efficient. A new version of **lfsrgen()** may perhaps provide a builtin option for filtering.

As expected, based on the theory, the higher p values significantly increase the computation times. The possibility of $p = 20$ was included and tested in the program. It is however clear that in order to attain the necessary amount of data for a thorough quantitative analysis of LFSRs of higher order than $p = 16$, a lot of computation power and time is needed.

The brute force approach, while usable for the low polynomial degrees of the component LFSRs used in this thesis, is likely not a viable solution for higher and more realistic degrees unless the performance is further improved through for example use of GPU arrays. As mentioned, in the initial research phase of this thesis the possibilities of backtracking through the candidate set in order to derive possible clock sequences was explored. The author of this thesis was however unable to see any practical solutions to this problem with the available literature and tools. One possible solution may however be building on the program developed in this thesis and by introducing a matrix of applied edit operations

it could be possible to backtrack through this matrix to derive the corresponding clock sequences at the cost of using more memory.

Chapter 7

Conclusion

This thesis is the result of a mixed-method exploratory research project on clock sequence reconstruction in irregularly clocked stream ciphers. It is the final step in the full bit-parallel attack on such ciphers, which has to the best of the authors knowledge, not been researched before in the known literature. The thesis provides and exploratory insight into a simple baseline process of brute force algorithm to break a stream cipher system based on BRM. The final program is heavily based upon the theory developed by Petrovic [2] and initial software implementation by Øverbø [3]. Many areas of the process remain to be further researched and improved upon. However, this thesis has demonstrated a framework for the attack making breaking the full system possible in CPU software for low polynomial degrees. It is left to future researchers to further improve performance and hopefully also develop a clock reconstruction algorithm that is more efficient than the brute force approach taken in this thesis. Specific ideas for future improvements are suggested in the final section of this chapter.

7.1 Answers to research questions

Recalling the research questions from Section 1.5, the thesis concludes with the following answers:

RQ1 - Is it possible to define an arbitrary optimal decision rule for selecting the set of candidates for the initial state of $R2$?

In the thesis several experiments have been done over a set of different input parameters for the different algorithm implementations. It was shown that for the less constrained version (OR2) of the candidate generation algorithm, an initial value of the error threshold k approximately $k \approx \frac{m}{3.6}$ is appropriate. The constrained OR3 algorithm requires a higher threshold due to the additional constraints, $k \approx \frac{m}{2.5}$. The thesis also introduces the idea of the *seeking* algorithm in order to evaluate the resulting candidate sets before continuing the attack.

It is clear however that a definitive conclusion on this question cannot be made until further quantitative analysis is done on larger data set and for a representable selection of different initial states.

RQ2 - Can a practical attack on irregularly clocked stream ciphers based on BRM be implemented and tested using brute-force in software?

As the demonstration in Section 5.5 showed, a practical attack is in fact feasible when the degree of the LFSRs are low enough or the attacker has access to large amounts of computing power. It was shown that the introduction of multi-threading makes the program capable of performing at the same levels as the FPGA implementations proposed in [3] when run in large virtual resource pools. As the software developed in the thesis is scalable and possible to load distribute, a practical attack on actual implementations may be possible for an actor with access to large computing resources.

RQ3 - Is it possible to recover the initial states of both the clocking ($R1$) and clocked ($R2$) LFSR in less than $(2^p-2)^2$ brute-force iterations?

As shown in this thesis, it is possible to recover the initial states of both $R1$ and $R2$ in $|C| \times (2^p - 2)$ brute-force iterations in the worst case, where $|C|$ represents the amount of $R2$ candidates in the candidate set C. By assuming no collisions occur during the process the average case of the attack would be closer to $\frac{|C|\times(2^p-2)}{2}$ iterations, as the program interrupts the process as soon as a match is found. The load distribution functions proposed in this thesis, which are easily implemented through the supplied software framework can further improve the overall brute-force performance. The thesis also suggests improvements to the candidate generation in order to generate the smallest possible valid set of candidates for $R2$ as the answer to **RQ1**.

RQ4 - Is it possible to improve the performance of the overall algorithm by introducing orchestrated multi-threading using cloud based infrastructure?

It has been shown that the deployment of the software and its wrapper programs in the cloud can drastically increase the performance at relatively low costs using AWS spot-instances. The introduction of the **.cand** output files which can be distributed to multiple hosts which in turn are issued "jobs" through the jobs-file enables scaling the attack to a degree only limited by the resources available to the attacker.

While the full potential of cloud based load distribution in the program remains to be exploited, it has been shown that it is highly effective and the provided software framework should make even further improvements easy to implement.

7.2 Future work

Throughout the thesis, pointers have been made to what can be improved upon or further researched in the topic of cryptanalysis of irregularly clocked pseudorandom generators in the future.

As a direct continuation of the research done in this topic, there remains a potential for further improvements on the arbitrary decision rules as asked in **RQ1**. While the results from applying different search algorithms for different values of k relating to m has been tested, it is probably necessary to test this over a representable set of initial states to determine if the results given in this thesis holds for the whole system. This thesis leaves future researchers with the tools for generating larger data sets which will be required for quantitative analysis on this topic.

In order to further improve the performance of the attack, a fully distributed approach using the **.cand**-file and the read version of the program on many virtual hosts in the cloud is possible to achieve given the framework provided in this thesis and may be a part of future studies on practical attacks. Several potential improvements in the program itself has also been pointed out through the instrumentation of the key functions of the algorithms. Specifically, the improvement of the **lfsrgen()** and **mpz_lshift()**-functions has the potential to significantly improve the overall performance of the program. Furthermore, the implementation of the proposed algorithms in GPU (see [47],[48] and [49]) may provide significant performance improvements compared to that of the CPU version developed in this thesis.

Finally, as the brute-force approach is the simplest and least effective form of cryptoanalytic attack, future researches are encouraged to attempt alternative methods of reconstructing the clock sequence in irregularly clocked pseudorandom sequence generators when employing bit-parallel search to generate candidates. A similar approach to that of the dynamic programming approach [51] may be possible and the possibilities of this should be further explored.

Bibliography

[1] J. Golic, 'A generalized correlation attack on a class of stream ciphers based on the levenshtein distance,' *Journal of Cryptography*, vol. 3, no. 1, pp. 201–212, 1991.

[2] S. Petrovic, 'Constrained approximate bit-parallel search with application in cryptanalysis.,' *RECSI XV: Seión 6. Seguridad y Análisis de Datos*, vol. 15, no. 6, pp. 174–179, 2018. [Online]. Available: https://nesg.ugr.es/recsi2018/docs/ActasXVRECSI.pdf.

[3] M. Øverbø, *Cryptoanalysis of Irregularly Clocked LFSR using approximate RBP search on FPGA*, 2019.

[4] Siegenthaler, 'Decrypting a class of stream ciphers using ciphertext only,' *IEEE Transactions on Computers*, vol. C-34, no. 1, pp. 81–85, 1985. DOI: 10.1109/TC.1985.1676518.

[5] R. W. Hamming, 'Error detecting and error correcting codes,' *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[6] V. I. Levenshtein, 'Binary Codes Capable of Correcting Deletions, Insertions and Reversals,' *Soviet Physics Doklady*, vol. 10, p. 707, Feb. 1966.

[7] C. E. Shannon, 'Communication theory of secrecy systems,' *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949. DOI: 10.1002/j.1538-7305.1949.tb00928.x.

[8] Wikipedia contributors, *Moscow-washington hotline— Wikipedia, the free encyclopedia*, [Online; accessed 17-April-2021], 2021. [Online]. Available: https://en.wikipedia.org/wiki/Moscow%E2%80%93Washington_hotline.

[9] Wikipedia contributors, *Stream cipher— Wikipedia, the free encyclopedia*, [Online; accessed 17-April-2021], 2021. [Online]. Available: https://en.wikipedia.org/wiki/Stream_cipher.

[10] T. Helleseth, 'Golomb's randomness postulates,' in *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg, Ed. Boston, MA: Springer US, 2005, pp. 242–242, ISBN: 978-0-387-23483-0.

[11] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 2001. [Online]. Available: http://www.cacr.math.uwaterloo.ca/hac/.

[12] Rationality., *The Cambridge Dictionary of Philosophy*. Cambridge University Press, 1999.

[13] W. G. Chambers and S. Jennings, 'Linear equivalence of certain BRM shift-register sequences,' *Electronics Letters*, vol. 20, pp. 1018–1019, 1984.

[14] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 2001, ch. 6.3.3, pp. 209–2012. [Online]. Available: http://www.cacr.math.uwaterloo.ca/hac/.

[15] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002, ch. 6.1, p. 145. DOI: 10.1017/CBO9781316135228.

[16] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002, ch. 6.3, pp. 150–152. DOI: 10.1017/CBO9781316135228.

[17] E. Ukkonen, 'Algorithms for approximate string matching,' *Information and Control*, vol. 64, no. 1, pp. 100–118, 1985, International Conference on Foundations of Computation Theory, ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(85)80046-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995885800462.

[18] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002, ch. 1.3.3, pp. 11–12. DOI: 10.1017/CBO9781316135228.

[19] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002, ch. 6.4, pp. 152–162. DOI: 10.1017/CBO9781316135228.

[20] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002, ch. 6.4.1, pp. 152–153. DOI: 10.1017/CBO9781316135228.

[21] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002, ch. 2.2.2, pp. 19–22. DOI: 10.1017/CBO9781316135228.

[22] S. Wu and U. Manber, 'Fast Text Searching: Allowing Errors,' *Commun. ACM*, vol. 35, no. 10, pp. 83–91, Oct. 1992, ISSN: 0001-0782. DOI: 10.1145/135239.135244. [Online]. Available: https://doi.org/10.1145/135239.135244.

[23] P. Leedy and J. Ormrod, *Practical Research: Planning and Design*. Pearson Education, 2015, ISBN: 9780133741957. [Online]. Available: https://books.google.no/books?id=jbSgBwAAQBAJ.

[24] J. W. Creswell and V. L. Plano Clark, *Designing and conducting mixed methods research / John W. Creswell, Vicki L. Plano Clark*, English, 2nd ed. SAGE Publications Los Angeles, Calif, 2011, xxvi, 457 p. : ISBN: 9781412975179 1412975174.

[25] T. Kempf, K. Karuri and L. Gao, 'Software instrumentation,' in. Sep. 2008, ISBN: 9780470050118. DOI: 10.1002/9780470050118.ecse386.

[26] VSCodium. (2021). 'VSCodium - Open source binaries,' [Online]. Available: https://vscodium.com/ (visited on 25/05/2021).

[27] Canonical. (2021). 'Ubuntu 20.04.2.0 LTS (Focal Fossa),' [Online]. Available: https://releases.ubuntu.com/20.04/ (visited on 25/05/2021).

[28] Visual Data Tools. (2021). 'DataGraph for macOS,' [Online]. Available: https://www.visualdatatools.com/DataGraph/ (visited on 25/05/2021).

[29] H. Cavusoglu, H. Cavusoglu and S. Raghunathan, 'Emerging issues in responsible vulnerability disclosure.,' in *WEIS*, 2005.

[30] S. A. Jessen, *Prosjektadministrative metoder*. Gyldendal, 2002, ISBN: 9788200128397.

[31] GNU. (2021). 'Status of C99 features in GCC,' [Online]. Available: https://gcc.gnu.org/c99status.html (visited on 04/05/2021).

[32] Google. (2021). 'Golang project description,' [Online]. Available: https://golang.org/project (visited on 04/05/2021).

[33] ISO, 'ISO C Standard 1999,' Tech. Rep., 1999, ISO/IEC 9899:1999 draft. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[34] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, Dec. 2011, 683 (est.) [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

[35] G. Ippolito. (2021). 'POSIX thread (pthread) libraries,' [Online]. Available: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html (visited on 04/05/2021).

[36] MIT. (2021). 'The MIT License,' [Online]. Available: https://opensource.org/licenses/MIT (visited on 04/05/2021).

[37] John. (2019). 'Thread pool in C,' [Online]. Available: https://nachtimwald.com/2019/04/12/thread-pool-in-c/ (visited on 10/04/2021).

[38] FSF. (2021). 'The GNU MP Bignum Library,' [Online]. Available: https://gmplib.org/ (visited on 04/05/2021).

[39] C. P. Tom Steele and D. Kottmann, *Black Hat Go*. No Starch Press, 2020, ISBN: 9781593278656.

[40] Amazon.com, Inc. (2021). 'About AWS,' [Online]. Available: https://aws.amazon.com/about-aws/ (visited on 04/05/2021).

[41] Google. (2021). 'Google Cloud Computing Services,' [Online]. Available: https://cloud.google.com/ (visited on 04/05/2021).

[42] Arash Partow. (2021). 'Primitive Polynomial List,' [Online]. Available: http://www.partow.net/programming/polynomials/index.html#deg20 (visited on 30/05/2021).

[43] R. Church, 'Tables of irreducible polynomials for the first four prime moduli,' *Annals of Mathematics*, vol. 36, no. 1, pp. 198–209, 1935, ISSN: 0003486X. [Online]. Available: http://www.jstor.org/stable/1968675.

[44] sethtroisi. (2019). 'libgmp/com.c at master - Github.com,' [Online]. Available: https://github.com/sethtroisi/libgmp/blob/master/mpz/com.c (visited on 12/05/2021).

[45] FSF. (2021). 'I/O of Integers (GNU MP 6.2.1),' [Online]. Available: https://gmplib.org/manual/I_002fO-of-Integers (visited on 04/05/2021).

[46] Amazon.com, Inc. (2021). 'Amazon Elastic File System,' [Online]. Available: https://aws.amazon.com/efs/ (visited on 10/05/2021).

[47] T. Nakayama and D. Takahashi, 'Implementation of multiple-precision floating-point arithmetic library for gpu computing,' Dec. 2011. DOI: 10.2316/P.2011.757-041.

[48] NVlabs. (2021). 'CGBN: CUDA Accelerated Multiple Precision Arithmetic (Big Num) using Cooperative Groups,' [Online]. Available: https://github.com/NVlabs/CGBN (visited on 20/05/2021).

[49] H. J. Liu and C. Tong, 'Gmp implementation on cuda – a backward compatible design with performance tuning,' 2011.

[50] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002, ch. 6.5, pp. 162–171. DOI: 10.1017/CBO9781316135228.

[51] G. Myers, 'A fast bit-vector algorithm for approximate string matching based on dynamic programming,' *J. ACM*, vol. 46, no. 3, pp. 395–415, May 1999, ISSN: 0004-5411. DOI: 10.1145/316542.316550. [Online]. Available: https://doi.org/10.1145/316542.316550.

Appendix A

Source code

A.1 Multi-threading libraries

Code listing A.1: <tpool.h> Thread pool header file [37]

```
// fetched from https://nachtimwald.com/2019/04/12/thread-pool-in-c/ 11.04.2021


#ifndef __TPOOL_H__
#define __TPOOL_H__

#include <stdbool.h>
#include <stddef.h>

struct tpool;
typedef struct tpool tpool_t;

typedef void (*thread_func_t)(void *arg);

tpool_t *tpool_create(size_t num);
void tpool_destroy(tpool_t *tm);

bool tpool_add_work(tpool_t *tm, thread_func_t func, void *arg);
void tpool_wait(tpool_t *tm);

#endif /* __TPOOL_H__ */
```

Code listing A.2: <tpool.c> Thread pool source file [37]

```
#include "tpool.h"
#include <pthread.h>
#include <stdlib.h>


struct tpool_work {
    thread_func_t      func;
    void               *arg;
    struct tpool_work *next;
};
typedef struct tpool_work tpool_work_t;

```

```
struct tpool {
    tpool_work_t    *work_first;
    tpool_work_t    *work_last;
    pthread_mutex_t  work_mutex;
    pthread_cond_t   work_cond;
    pthread_cond_t   working_cond;
    size_t           working_cnt;
    size_t           thread_cnt;
    bool             stop;
};

static tpool_work_t *tpool_work_create(thread_func_t func, void *arg)
{
    tpool_work_t *work;

    if (func == NULL)
        return NULL;

    work       = malloc(sizeof(*work));
    work->func = func;
    work->arg  = arg;
    work->next = NULL;
    return work;
}

static void tpool_work_destroy(tpool_work_t *work)
{
    if (work == NULL)
        return;
    free(work);
}

static tpool_work_t *tpool_work_get(tpool_t *tm)
{
    tpool_work_t *work;

    if (tm == NULL)
        return NULL;

    work = tm->work_first;
    if (work == NULL)
        return NULL;

    if (work->next == NULL) {
        tm->work_first = NULL;
        tm->work_last  = NULL;
    } else {
        tm->work_first = work->next;
    }
    return work;
}
static void *tpool_worker(void *arg)
{
    tpool_t      *tm = arg;
    tpool_work_t *work;

    while (1) {
        pthread_mutex_lock(&(tm->work_mutex));

        while (tm->work_first == NULL && !tm->stop)
```

```
73              pthread_cond_wait(&(tm->work_cond), &(tm->work_mutex));
74
75          if (tm->stop)
76              break;
77
78          work = tpool_work_get(tm);
79          tm->working_cnt++;
80          pthread_mutex_unlock(&(tm->work_mutex));
81
82          if (work != NULL) {
83              work->func(work->arg);
84              tpool_work_destroy(work);
85          }
86
87          pthread_mutex_lock(&(tm->work_mutex));
88          tm->working_cnt--;
89          if (!tm->stop && tm->working_cnt == 0 && tm->work_first == NULL)
90              pthread_cond_signal(&(tm->working_cond));
91          pthread_mutex_unlock(&(tm->work_mutex));
92      }
93
94      tm->thread_cnt--;
95      pthread_cond_signal(&(tm->working_cond));
96      pthread_mutex_unlock(&(tm->work_mutex));
97      return NULL;
98  }
99
100 tpool_t *tpool_create(size_t num)
101 {
102     tpool_t   *tm;
103     pthread_t  thread;
104     size_t     i;
105
106     if (num == 0)
107         num = 2;
108
109     tm             = calloc(1, sizeof(*tm));
110     tm->thread_cnt = num;
111
112     pthread_mutex_init(&(tm->work_mutex), NULL);
113     pthread_cond_init(&(tm->work_cond), NULL);
114     pthread_cond_init(&(tm->working_cond), NULL);
115
116     tm->work_first = NULL;
117     tm->work_last  = NULL;
118
119     for (i=0; i<num; i++) {
120         pthread_create(&thread, NULL, tpool_worker, tm);
121         pthread_detach(thread);
122     }
123
124     return tm;
125 }
126
127 void tpool_destroy(tpool_t *tm)
128 {
129     tpool_work_t *work;
130     tpool_work_t *work2;
131
132     if (tm == NULL)
```

```
133         return;
134
135     pthread_mutex_lock(&(tm->work_mutex));
136     work = tm->work_first;
137     while (work != NULL) {
138         work2 = work->next;
139         tpool_work_destroy(work);
140         work = work2;
141     }
142     tm->stop = true;
143     pthread_cond_broadcast(&(tm->work_cond));
144     pthread_mutex_unlock(&(tm->work_mutex));
145
146     tpool_wait(tm);
147
148     pthread_mutex_destroy(&(tm->work_mutex));
149     pthread_cond_destroy(&(tm->work_cond));
150     pthread_cond_destroy(&(tm->working_cond));
151
152     free(tm);
153 }
154
155 bool tpool_add_work(tpool_t *tm, thread_func_t func, void *arg)
156 {
157     tpool_work_t *work;
158
159     if (tm == NULL)
160         return false;
161
162     work = tpool_work_create(func, arg);
163     if (work == NULL)
164         return false;
165
166     pthread_mutex_lock(&(tm->work_mutex));
167     if (tm->work_first == NULL) {
168         tm->work_first = work;
169         tm->work_last  = tm->work_first;
170     } else {
171         tm->work_last->next = work;
172         tm->work_last       = work;
173     }
174
175     pthread_cond_broadcast(&(tm->work_cond));
176     pthread_mutex_unlock(&(tm->work_mutex));
177
178     return true;
179 }
180 void tpool_wait(tpool_t *tm)
181 {
182     if (tm == NULL)
183         return;
184
185     pthread_mutex_lock(&(tm->work_mutex));
186     while (1) {
187         if ((!tm->stop && tm->working_cnt != 0) || (tm->stop && tm->thread_cnt !=
                ↪ 0)) {
188             pthread_cond_wait(&(tm->working_cond), &(tm->work_mutex));
189         } else {
190             break;
191         }
```

```
192     }
193     pthread_mutex_unlock(&(tm->work_mutex));
194 }
```

A.2 Main program

Code listing A.3: <main.c> The main program source code

```
1  /**############################################################################
2   ** TITLE:   CipherSearch 3.0
3   ** CONTRIBUTORS: Magnus Overbo & Adrian S. Evensen
4   ** ABOUT:   Ciphersearch utilises the GMP library to manage arbitrary sized
5   **      numbers and perform an unconstrained approximate row-based bit
6   **      parallell search.  It searches for an intercepted bitsequence,
7   **      generated by a decimated LFSR encrypted with a message, and
8   **      tries to find this in all possible generated sequences which are
9   **      not decimated. The candidate set can then be attacked with
10  **      a brute force attack in order to break the cryptosystem.
11  **
12  ** Release:  20190313 - 64b version
13  ** Release:  20190328 - GMP library version for arbitrary bit size
14  ** Release:  20210531 - Add option for shift-OR implementation (default)
15  **        - Add brute force for clock reconstruction
16  **############################################################################**/
17
18  //----------------------------------------------------------------------------
19  // INCLUDES
20  //----------------------------------------------------------------------------
21  #include <stdio.h>
22  #include <unistd.h>
23  #include <stdbool.h>
24  #include <pthread.h>
25  #include <sys/types.h>
26  #include <sys/stat.h>
27  #include <assert.h>
28  #include <stdlib.h>
29  #include <sys/time.h>
30  #include <string.h>     //strlen
31  #include <stdint.h>     //64b Int
32  #include <inttypes.h>   //64b int
33  #include <gmp.h>        //arbitrary integer size
34  #include "tpool.h"
35
36  //----------------------------------------------------------------------------
37  // GLOBAL VARIABLES
38  //----------------------------------------------------------------------------
39  size_t num_threads = 16; // Thread pool limit. Generally set to # of vCPUs in cloud
        ↪ instances
40  const int ALPHASIZE = 2; // Alphabet size, 0 & 1
41  tpool_t *tm;
42  int col_acceptance;
43  int col;
44  int hit;
45  int m;
46  int n;
47  int deg;
48  mpz_t pol;
```

```
int slen;
int plain = 0;
mpz_t PLAINTEXT;
mpz_t CIPHER;
int R1STATE;
int R2STATE;
mpz_t max;
mpz_t* B;

//------------------------------------------------------------------------------
// STRUCTs
//------------------------------------------------------------------------------
struct LFSR {
 mpz_t POLYNOMIAL; //Polynomial definition
 mpz_t STATE;   //LFSR state
 int DEGREE;   //Polynomial degree
};

struct CANDIDATE {
 int istate;  // R2 initial state
 bool match;
 mpz_t X;  // Undecimated output
};

//------------------------------------------------------------------------------
// FUNCTION DECLARATIONs
//------------------------------------------------------------------------------
int search();              // Gen target system
int polyMap(int);             // Get LFSR polynomial
mpz_t* genAlphabet( int );              // Gen array of the alphabet
int lfsr_iterate( struct LFSR*);         // Gen next state & output
void lfsrgen(mpz_t, int, int, mpz_t, uint_least64_t, int, mpz_t*);  // Gen LFRS
mpz_t* arbp_search( mpz_t*, mpz_t, int, int, int );          // Main search
    ↪ function
mpz_t* genError(int);              // Gen init error table
void genPrefixes( mpz_t*, mpz_t, int );        // Generate the prefixes
void genEncrypt( mpz_t, mpz_t, mpz_t, mpz_t, int );           // Encrypt the
    ↪ plaintext
void mpz_lshift( mpz_t, int );          // Left shift bin seq by 1
char* pb( mpz_t, int, int );          // Print prepending zeros


// CSV Parse function fetched from
// https://stackoverflow.com/questions/12911299/read-csv-file-in-c
const char* getfield(char* line, int num)
{
    const char* tok;
    for (tok = strtok(line, ",");
            tok && *tok;
            tok = strtok(NULL, ",\n"))
    {
        if (!--num)
            return tok;
    }
    return NULL;
}

void match_R1(void *arg) { // Thread for brute force attempts
 struct CANDIDATE *cand = (struct CANDIDATE *)arg;
 mpz_t LCLK; mpz_init(LCLK);      //LFSR for dessimating
```

```
107  mpz_t CIPHER2; mpz_init( CIPHER2 );   //Gen test ciphertext
108  mpz_init(LCLK);
109  mpz_init(CIPHER2);
110  int i = 0;
111  int c = 0;
112
113  while ( i<mpz_get_ui(max) && col <= col_acceptance ) { // Iterate through all
         ↪ R1States until a hit is found or collision acceptance exceeded
114   #if defined DEBUG
115    printf("Generating␣clocking␣LFSR␣(R1)␣output␣sequence:␣\n");
116   #endif
117   lfsrgen(LCLK, deg, m, pol, i, 0, NULL);    // Clocking LFSR, R1
118
119   #if defined DEBUG
120    printf("Calculating␣the␣Decimated␣bitsequence␣and␣creating␣ciphertext:\n\n");
121   #endif
122
123   genEncrypt(CIPHER2, LCLK, cand->X , PLAINTEXT, m);  // Generate the candidate
         ↪ cipher
124
125   if ( mpz_cmp(CIPHER, CIPHER2) == 0 ) {    // Check if candidate cipher matches
126    #if defined DEMO
127    printf("\t␣-␣POSSIBLE␣MATCH␣FOUND␣for␣R1␣init␣state␣%i␣and␣R2␣init␣state␣%i\n",
         ↪ i, cand->istate);
128    col++;
129    #else
130    if (cand->istate == R2STATE && i == R1STATE) { // CHEATING. However, in a real
         ↪ approach, the collision indicator will work as a counter and indicate if
         ↪  there are 1 or more hits.
131     printf("\t␣-␣ACTUAL␣INIT␣STATES␣FOUND␣for␣R1␣init␣state␣%i␣and␣R2␣init␣state␣%i
          ↪ \n", i, cand->istate);
132     hit = 1;
133    } else {
134     printf("\t␣-␣COLLISION␣FOUND␣at␣R1␣init␣state␣%i␣and␣R2␣init␣state␣%i\n", i,
          ↪ cand->istate);
135     //run function to terminate all match_R1 instances.
136     col ++;
137    }
138    #endif
139   }
140   i++;
141  }
142  mpz_clear(LCLK);
143  mpz_clear(CIPHER2);
144 }
145
146 void search_thread(void *arg) { // Thread for ARBP search
147    struct CANDIDATE *cand = (struct CANDIDATE *)arg;
148  mpz_t TEXT;            // This variable stores the current search text
149  mpz_init(TEXT);
150  int ci = 0;
151
152  lfsrgen( TEXT, deg, n, pol, cand->istate, 1, B );  // Generate undecimated bitseq
         ↪ TEXT for current initial state
153
154  mpz_t* MATCH = arbp_search(B, TEXT, slen, m, n);  // Run the ARBP search on TEXT
         ↪ with slen errors allowed and return matches with CLD and position.
155
156  int j = 0;
157  while(j < n){           //For each position
```

```
158   if( mpz_cmp_ui(MATCH[j], m) < 0 ){     //If match is less than m
159    mpz_clear( MATCH[j] );
160    ci++;           //increment counter for print
161   }
162   j++;            //Next position
163  }
164
165  if (ci > 0) {           //If matches exist, add state to Candidate matrix.
166   cand->match = true;
167   mpz_init(cand->X); mpz_set(cand->X, TEXT);
168   ci = 0;
169  }
170  else {
171   cand->match = false;
172   mpz_init(cand->X); mpz_set(cand->X, TEXT);
173  }
174  free(MATCH);
175 }
176
177 //------------------------------------------------------------------------------
178 // Program start
179 //------------------------------------------------------------------------------
180 int main(int argc, char *argv[]){
181  struct timeval start, end;
182  gettimeofday(&start, NULL);
183  //-------------------------------------------------------------
184  // Parameter initialization and validation
185  //-------------------------------------------------------------
186  #if defined READCAND // Change argument requirements when reading file. These
       ↪ should in the future be changed to parse from filename instead
187   if( argc != 9 ){              //Check that required input parameters are met
188    printf("Incorrect␣number␣of␣arguments\n␣Usage:␣./main␣<polynomial␣degree>␣<
         ↪ search␣word␣length>␣<search␣text␣length>␣<init␣state␣R1>␣<init␣state␣R2>
         ↪ ␣<collision␣acceptance>␣<#␣CPUs>␣<.cand␣path>\n");
189    return 1;
190   }
191   deg = atoi( argv[1] );
192   m = atoi( argv[2] );
193   n = atoi( argv[3] );
194   slen = 0; // Doesn't matter, as we are not doing ARBP
195   R1STATE = atoi( argv[4] );
196   R2STATE = atoi( argv[5] );
197   col_acceptance = atoi( argv[6] );
198   num_threads = atoi( argv[7] );
199   char* candfile = argv[8];
200  #else
201   if( argc != 9 ){              //Check that required input parameters are met
202    printf("Incorrect␣number␣of␣arguments\nUsage:␣./main␣<polynomial␣degree>␣<search
         ↪ ␣word␣length>␣<search␣text␣length>␣<errors>␣<init␣state␣R1>␣<init␣state␣
         ↪ R2>␣<collision␣acceptance>␣<#␣CPUs>\n");
203    return 1;
204   }
205   deg = atoi( argv[1] );
206   m = atoi( argv[2] );
207   n = atoi( argv[3] );
208   slen = atoi( argv[4] )+1;
209   R1STATE = atoi( argv[5] );
210   R2STATE = atoi( argv[6] );
211   col_acceptance = atoi( argv[7] );
212   num_threads = atoi( argv[8] );
```

```
213 #endif
214
215 struct stat st = {0};
216 if (stat("./data", &st) == -1) {      // Create ./data directory if it does not
        ↪ exist.
217  mkdir("./data", 0777);
218 }
219
220 if (n<2*m) {
221  printf("Search␣text␣must␣be␣at␣least␣2␣*␣m...\n");
222  return 1;
223 }
224
225 if (slen >= m) {
226  printf("Error:␣Errors␣cannot␣be␣higher␣than␣search␣word␣(m)...\n");
227  return 1;
228 }
229
230 mpz_init( pol );
231 mpz_set_ui(pol, polyMap(deg));      //Get LFSR polynomial
232
233 mpz_init( max );
234 mpz_setbit(max, deg);        //Set max val, eg 2047 in 2^11
235 if (R1STATE > mpz_get_ui(max)) {
236  printf("Error:␣R1␣initial␣state␣(%d)␣is␣higher␣than␣LFSR␣capacity␣(%lu)\n",
        ↪ R1STATE, mpz_get_ui(max));
237  return 1;
238 } else if (R2STATE > mpz_get_ui(max)) {
239  printf("Error:␣R2␣initial␣state␣(%d)␣is␣higher␣than␣LFSR␣capacity␣(%lu)\n",
        ↪ R2STATE, mpz_get_ui(max));
240  return 1;
241 }
242
243 mpz_init(PLAINTEXT);
244 mpz_set_ui(PLAINTEXT, plain);      //Default value is 0
245
246 // SIMULATE THE BRM SYSTEM
247 mpz_t R1SEQ;  mpz_init(R1SEQ);
248 lfsrgen(R1SEQ, deg, m, pol, R1STATE, 0, NULL);  //Generate R1 output sequence
249
250 mpz_t R2SEQ;  mpz_init(R2SEQ);    //LFSR to be decimated
251 lfsrgen(R2SEQ, deg, n, pol, R2STATE, 0, NULL);  //Decimated LFSR
252
253 mpz_init( CIPHER );         //Gen intercepted ciphertext
254 genEncrypt( CIPHER, R1SEQ, R2SEQ, PLAINTEXT, m);
255
256 mpz_clear( R1SEQ );         //Cleanup LFSRs
257 mpz_clear( R2SEQ );
258
259 //------------------------------------------------------------------------------
260 // At this point, the target (CIPHER) has been created.
261 // The known-plaintext attack starts here.
262 //
263 // Step ONE: Generate prefixes and run ARBP search and choose candidates for R2
264 // Candidates are stored in an array in memory in the form:
265 // <initial state>, <R2 undecimated output sequence of length n>
266 //
267 // UNLESS: READCAND was defined when compiling, changing the modus of the
268 // program into clock sequence reconstruction only.
269 //------------------------------------------------------------------------------
```

```
#if defined DEMO
while (true) {
#endif

 tm   = tpool_create(num_threads);
 struct CANDIDATE* C = malloc( mpz_get_ui(max) * sizeof(struct CANDIDATE) );  //
     ↪ Initialize candidate array
 struct CANDIDATE* ptr = C;
 struct CANDIDATE* endPtr = C + (sizeof(*C)/sizeof(struct CANDIDATE) * mpz_get_ui(
     ↪ max));

 #if defined READCAND // If we expect to read an already generated .cand-file from
     ↪ previous generation
  printf("Reading␣file...␣\n");
  FILE* stream = fopen(candfile, "r");
  int l;
  char line[1024];
  while (fgets(line, 1024, stream)) // iterate through all lines in .cand file
  {
   char* tmp = strdup(line);
   l = atoi(getfield(tmp, 1));
   char* tmp2 = strdup(line);
   C[l].istate = l;
   mpz_set_str(C[l].X, getfield(tmp2, 2), 62); //Decode the base-62 sequence
  }
  // Start brute force attack
  struct timeval start, end;
  gettimeofday(&start, NULL);
  tm   = tpool_create(num_threads);
  col = 0;           // Reset collision counter
  for (int i = 0; i < (mpz_get_ui(max)-1); i++){  // Iterate through all initial
      ↪ states of R1
   C[i].istate = i;
   tpool_add_work(tm, match_R1, &C[i]);   // Launch brute-force threads
  }
  tpool_wait(tm);
  tpool_destroy(tm);
  // Attack done, summarize.
  if (hit > 0) {
   printf("Found␣%d␣ACTUAL␣hits␣(by␣cheating)␣and␣%d␣collisions.", hit, col);
  } else {
   printf("No␣hits␣found...");
  }
  gettimeofday(&end, NULL);
  long seconds = (end.tv_sec - start.tv_sec);
  long micros = ((seconds * 1000000) + end.tv_usec) - (start.tv_usec);
  printf("\nThe␣elapsed␣time␣is␣%ld␣seconds␣and␣%ld␣micros␣after␣R1␣reconstruction\
      ↪ n", seconds, micros);
 #else // Normal mode of operation
  char* FNAME;
  int found;

  B = genAlphabet( ALPHASIZE );    //Generate alphabet
  genPrefixes(B, CIPHER, m);     //Generate prefixes for the alphabet

  for (int i = 0; i < (mpz_get_ui(max)-1); i++){  // Iterate through all initial
      ↪ states of R2
   C[i].istate = i+1;
   tpool_add_work(tm, search_thread, &C[i]);  // Launch thread for ARBP
  }
```

```
324   tpool_wait(tm);          // Wait here until all threads are dead
325   tpool_destroy(tm);
326
327   int u=0;
328
329   // Open file for writing. May need to be moved back up if position of edits are
          ↪ to be written
330   FNAME = malloc(60*sizeof(char));     //Filename allocation
331   sprintf(FNAME, "./data/%d_%d_%d_%d_%d_%d.cand", deg, m, n, slen-1, R1STATE,
          ↪ R2STATE);
332   FILE* fh = fopen(FNAME, "w");      // Open output file for writing
333
334   while ( ptr < endPtr ) { // Iterate through all possible candidates
335    if (ptr->match == true){ // If this is a valid candidate
336     fprintf(fh, "\n%i,", ptr->istate); mpz_out_str(fh, 62, ptr->X); // Write it to
           ↪ file
337     #if !defined DEMO
338      if (ptr->istate == R2STATE) {
339       found = 1;
340      }
341     #endif
342     u++; // Increase candidate counter
343    }
344    ptr++;
345   }
346
347   fclose( fh ); // Close data file
348   if ( u == 0 ) { // Zero matches, empty dataset
349    remove(FNAME); // Discard file
350    #if defined DEMO
351     printf("[k=%d] Empty candidate set. Increasing error threshold\n", slen);
352     slen++;
353    #else
354     printf("%d,%d",-1,u);
355     return -1;
356    #endif
357   } else if( u >= mpz_get_ui(max)-1 ) {  // Full dataset, all candidates
358    remove(FNAME); // Discard file
359    #if defined DEMO
360     printf("[k=%d] Full candidate set. Decreasing error threshold\n", slen);
361     slen--;
362    #else
363     printf("%d,%d",-2,u);
364     return -2;
365    #endif
366   }
367   #if !defined DEMO
368   else if ( found != 1 ) { // Dataset not containing the actual candidate (cheat)
369    remove(FNAME); // Discard file
370    printf("%d,%d",u,u);
371
372    return u;
373   }
374   #endif
375   else {
376    // Dataset is VALID!
377    // R1 reconstruction phase starts here
378    // Start brute force attack with intercepted CIPHER and known PLAINTEXT with the
          ↪  generated R2 candidate set
379    #if defined DEMO
```

```
380    printf("[k=%d] Valid candidate set |C|=%d (of %lu). Starting brute force attack
           ↪ !\n", slen, u, mpz_get_ui(max)-1);
381    #endif
382    tm   = tpool_create(num_threads); // Create thread pool
383    col = 0;           // Reset collision counter
384    for (int i = 0; i < (mpz_get_ui(max)-1); i++){  // Iterate through all initial
           ↪ states of R1
385     C[i].istate = i+1;
386     tpool_add_work(tm, match_R1, &C[i]);   // Launch brute-force threads
387    }
388    tpool_wait(tm); // Wait here until brute force is done
389    tpool_destroy(tm);
390
391    #if defined DEMO
392    if (col>0) {
393     gettimeofday(&end, NULL);
394     long seconds = (end.tv_sec - start.tv_sec);
395     long micros = ((seconds * 1000000) + end.tv_usec) - (start.tv_usec);
396
397     printf("A match was found in %ld! Exiting.\n", micros);
398     return 0;
399    } else {
400     printf("No matches found... Increasing error threshold \n");
401     slen++;
402    }
403    #else
404    if (col > 0) {
405     printf("%d,%d",-3,u);
406     return -3;
407    } else {
408     printf("%d,%d",0,u);
409    }
410    return 0;
411    #endif
412
413
414   }
415  #endif
416
417 #if defined DEMO
418 }
419 #endif
420
421
422 exit(0);
423
424 }
425
426 /**############################################################################
427  **
428  ** Helper functions
429  **
430  **############################################################################**/
431
432 /*----------------------------------------------------------------------------
433  * Generates the cipher which then becomes the prefix.
434  * Y is the clocking LFSR
435  * X is the encrypting LFSR
436  * Implementation of the BRM
437 ----------------------------------------------------------------------------*/
```

```
438 void genEncrypt(mpz_t rop, mpz_t CLK, mpz_t DES, mpz_t PLAINTEXT, int m){
439  int i = 0;
440  int j = 0;
441  int x, y;
442  char* t;
443  mpz_t CIPHER; mpz_init(CIPHER);
444  while( i < m ){      //Counter for clocking lfsr
445   x = 0;
446   y = 0;
447   #if defined DEBUG
448    printf("\tCLK(%d)=%d␣", i, mpz_tstbit(CLK, i) );
449   #endif
450
451   if( mpz_tstbit(CLK, i) == 1 ){
452     j++;     //Decimate LFSR by skipping a bit
453     #if defined DEBUG
454      printf("\tDES(%d)=%d␣-->␣\tDES(%d)=%d", j-1, mpz_tstbit(DES, j-1), j,
          ↪ mpz_tstbit(DES, j) );
455     #endif
456   }
457   #if defined DEBUG
458   else{
459    printf("\t\t\tDES(%d)=%d", j, mpz_tstbit(DES,j));
460   }
461   #endif
462
463   //Val of dessimated LFSR output
464   if( mpz_tstbit(DES, j) == 1 ) x = 1; //grab value of bit
465   if( mpz_tstbit(PLAINTEXT, i) == 1 )  y = 1; //Grab value of bit
466   if( (y^x) == 1 ) mpz_setbit(CIPHER, i);   //Xor to get CIPHER
467
468   #if defined DEBUG
469   printf("␣^␣P(%d)=%d␣==␣C(%d)=%d\n", i, mpz_tstbit(PLAINTEXT, i), i, mpz_tstbit(
        ↪ CIPHER, i) );
470   #endif
471   i++;
472   j++; //Increment counters
473  }
474  #if defined DEBUG
475   printf("\n");
476   t = pb(CIPHER, m, 0);
477   printf("\tCIPHERTEXT:␣%s", t); mpz_out_str(stdout, 2, CIPHER);
478   free(t);
479   printf("\n\n");
480  #endif
481  mpz_set(rop, CIPHER);        //Set var to generated cipher
482  mpz_clear( CIPHER );
483 }
484
485 /*--------------------------------------------------------------------------
486  *
487  * Return an irreducible polynomial based on the given degree.
488  *
489 --------------------------------------------------------------------------*/
490 int polyMap(int deg) {
491  int pol;
492  switch(deg) {
493   case 11:
494    pol = 1209;
495    break;
```

```
  case 16:
   pol = 33262;
   break;
  case 20:
   pol = 524564;
   break;
 }
 return pol;
}

/*--------------------------------------------------------------------------------
 * Prepend bits to a binary representation of a number.
 * returns a char pointer to an array filled with missing bits
 * or nothing if it is already full.
--------------------------------------------------------------------------------*/
char* pb( mpz_t num, int len, int b ){
 size_t plen = len+1 - mpz_sizeinbase( num, 2 );
 if( plen == 0 )
  return "";
 char* pre = malloc( plen * sizeof(char) );   //Allocate plen length array
 int i = 0;
 while( i < plen-1 ){
  if( b == 1){
   pre[i++] = '1';
  }
  else{
   pre[i++] = '0';
  }
 }
 pre[i]='\0';
 return pre;
}

/*--------------------------------------------------------------------------------
 * Left shift MPZ_T variable to the left
 *  n number of shift
 *  rop is mpz_t value to shift
--------------------------------------------------------------------------------*/
void mpz_lshift( mpz_t rop, int len ) {
 mpz_t tmp;
 mpz_init( tmp ); //temp variable
 int i = len-1;
 while( i > 0 ){
  if(mpz_tstbit(rop, i-1) == 1){
   mpz_setbit( tmp, i );  //set bit
  }
  i--;
 }
 mpz_set(rop, tmp);     //Set original var to tmp
 mpz_clear(tmp);
}

/*--------------------------------------------------------------------------------
 * Generate initial m-bitmasks for the alphabet of ALPHASIZE
 * Shift-OR Complements 1
--------------------------------------------------------------------------------*/
mpz_t* genAlphabet( int ALPHASIZE ){
 mpz_t* B = malloc((ALPHASIZE)*sizeof( mpz_t ));
 #if defined DEBUG
  printf( "Generating masks for the alphabet\n" );
```

```
556  #endif
557
558  int i=0;
559  while( i < ALPHASIZE ){
560   mpz_init( B[i] );
561   mpz_set_ui( B[i], 0 );
562
563   #if defined DEBUG
564    printf("\tB[%d]␣",i);
565    mpz_out_str(stdout, 2, B[i]);
566    printf("\n");
567   #endif
568   i++;
569  }
570  #if defined DEBUG
571   printf( "\tDone\n\n" );
572  #endif
573  return B;
574 }
575
576 /*-------------------------------------------------------------------------------
577  * LFSR iteration function
578  *  Calculates the next state of the LFSR by first grabbing the MSB as output
579  *  value.  Then it calculates the AND of cur state and the polynomial before
580  *  running an XOR on all bits that are set in the temp var to generate the
581  *  feedback value.
582  *
583  *  Finally it left shifts the entire original state and sets the LSB to the
584  *  value of the feedback polynomial.
585  *
586  *  The feedback value is then set
587 -------------------------------------------------------------------------------*/
588 int lfsr_iterate( struct LFSR* lfsr) {
589  int i = 0;           //Counter
590  int fbck = 0;          //XOR calculated value
591  int ret = mpz_tstbit( lfsr->STATE, lfsr->DEGREE-1 );
592
593  mpz_t tmp;
594  mpz_init( tmp );
595  mpz_and(tmp, lfsr->STATE, lfsr->POLYNOMIAL); //AND taps the LFSR according to the
       ↪ given POLYNOMIAL
596
597  while( i < lfsr->DEGREE ){    //Calc feedback
598   fbck = fbck + mpz_tstbit( tmp, i );
599   i++;
600  }
601
602  fbck = fbck % 2; // XOR the feedback bits
603  mpz_lshift(lfsr->STATE, lfsr->DEGREE); //Left shift
604  if( fbck == 1 ) mpz_setbit( lfsr->STATE, 0 );
605  mpz_clear( tmp );
606  return ret;           //Return output character
607 }
608
609 /*-------------------------------------------------------------------------------
610  * Generate LFSR and output an n-length bitsequence
611  * With all arbitrary skips until first prefix is met
612 -------------------------------------------------------------------------------*/
613 void lfsrgen(mpz_t rop, int psize, int olen, mpz_t p,
614        uint_least64_t iv, int skip, mpz_t* B){
```

```
615  int i;           //Counter var
616  int initmatch = 0;        //Check if first prefix is found
617  struct LFSR lfsr;        //Create struct variable
618  char* t;
619
620  lfsr.DEGREE = psize;       //Set polynomial degree
621
622  mpz_init( lfsr.POLYNOMIAL );
623  mpz_set(lfsr.POLYNOMIAL, p);     //Set polynomial
624
625  mpz_init( lfsr.STATE );
626  mpz_set_ui(lfsr.STATE, iv);      //Set initial vector (seed)
627
628  #if defined DEBUG
629   t = pb(lfsr.STATE, psize, 0);
630   printf("\n\tINIT␣STT:\t%s",t); mpz_out_str( stdout, 2, lfsr.STATE );
631   printf("\n");
632   free(t);
633  #endif
634
635  mpz_t OUTPUT;
636  mpz_init( OUTPUT );
637  int tmpOUT = 0;        //temp char holder
638  i = 0;           //Zero out counter
639  while( i < olen ){        //Iterate the LFSR for the set output length
640   tmpOUT = lfsr_iterate(&lfsr);
641   if( initmatch == 0 && skip == 1) {   // Iterate until we match the first bit of
         ↪ the sequence
642
643    if( mpz_tstbit(B[tmpOUT], 0) == 1 ) {
644     initmatch = 1;      //Set state to found
645     if( tmpOUT == 1 )
646      mpz_setbit(OUTPUT, i);   //Set output to tmpvar
647
648     i++;        //inc counter
649    }
650   }
651   else {
652    if( tmpOUT == 1) mpz_setbit(OUTPUT, i);
653    i++;               //Next bit
654   }
655  }
656
657  //Print LFSR information
658  #if defined DEBUG
659  printf("\tDEGREE:\t\t%d\n", psize);
660  printf("\tLENGTH:\t\t%d\n", olen);
661  printf("\tPOLYNOMIAL:\t"); mpz_out_str( stdout, 2, lfsr.POLYNOMIAL );
662  t = pb(lfsr.STATE, psize, 0);
663  printf("\n\tFINAL␣STT:\t%s", t); mpz_out_str( stdout, 2, lfsr.STATE );
664  free(t); t = pb(OUTPUT, olen, 0);
665  printf("\n\tSEQUENCE:\t%s", t); mpz_out_str( stdout, 2, OUTPUT);
666  free(t);
667  printf("\n\n");
668  #endif
669  mpz_set(rop, OUTPUT);
670  mpz_clear(OUTPUT);
671  mpz_clear( lfsr.POLYNOMIAL );
672  mpz_clear( lfsr.STATE );
673 }
```

```
674
675 /*--------------------------------------------------------------------------
676  * Creates the prefixes
677 --------------------------------------------------------------------------*/
678 void genPrefixes( mpz_t* B, mpz_t P, int m ){
679  int  j = 0;
680
681  mpz_t tmp;
682  mpz_init(tmp);
683  mpz_set_ui( tmp, 1);
684
685  while( j<m ){
686   int ci = mpz_tstbit(P, j);    //Char value 0/1
687
688   #if defined DEBUG
689    printf("\tj=%d\tB[%d]␣|\t%s", j, mpz_tstbit(P,j), pb(B[ci],m,0) );
690    mpz_out_str(stdout, 2, B[ci]);
691   #endif
692
693   mpz_ior( B[ci], B[ci], tmp );   //current value or-ed with 10^(j-1)
694   mpz_lshift( tmp, m );
695
696   #if defined DEBUG
697    printf( "\n\t\t\t%s", pb(B[ci],m,0));
698    mpz_out_str( stdout, 2, B[ci]);  printf( "\n" );
699   #endif
700
701   j++;         //Next pattern character
702  }
703
704
705  mpz_clear( tmp );
706
707 }
708
709 /*--------------------------------------------------------------------------
710  * Creates the error list of K-size
711 --------------------------------------------------------------------------*/
712 mpz_t* genError(int K) {
713  mpz_t* R = malloc( K*sizeof(mpz_t) );  //Allocate memory for array
714  #if defined DEBUG
715   printf("Gen␣error␣R[%d..%d]\n", 0, K-1);
716  #endif
717  int k = 0;          //Set counter
718  while( k<K ){
719   int i = 0;
720   mpz_init( R[k] );
721   while( i < k ){
722    mpz_setbit( R[k], i );
723    i++;
724   }
725   #if defined SHIFTOR
726    mpz_t mask;
727    mpz_init(mask);
728    mpz_ui_pow_ui(mask, ALPHASIZE, m);
729    mpz_sub_ui(mask, mask, 1);
730    mpz_xor( R[k], R[k], mask );
731    //mpz_com(R[k], R[k]);
732   #endif
733
```

```
734   #if defined DEBUG
735   char* t;
736   t = pb(R[i],m,0);
737   printf("\tR[%d]\t=␣%s", k, t);
738   mpz_out_str( stdout, 2, R[k]);
739   printf( "\n" );
740   #endif
741   k++;
742  }
743
744  return R;
745 }
746
747 /*-------------------------------------------------------------------------
748  * Perform search on TEXT and PREFIX
749 -------------------------------------------------------------------------*/
750 mpz_t* arbp_search(mpz_t* B, mpz_t TEXT, int K, int m, int n) {
751  mpz_t tmp1;
752  mpz_init( tmp1 );         //Tmp variables
753  mpz_t tmp2;
754  mpz_init( tmp2 );
755  mpz_t tmp3;
756  mpz_init( tmp3 );
757  char* t;
758  char* b;
759              //Match for each position in text
760  mpz_t* R = genError( K );       //Gen error-table
761  mpz_t* MATCHES = malloc( n * sizeof(mpz_t) );
762
763  #if defined CONSTRAINTS2 // Additional matrix for the OR3 algorithm
764   mpz_t* dm = malloc( (K)*sizeof(mpz_t) ); // Initialize dm_i, i=1...k
765   mpz_t del;
766   mpz_init(del);
767   int d = 0;
768   while (d<K) {
769    mpz_init(dm[d]);
770    mpz_set(dm[d], R[0]);       // Set all dm_i to 1^m
771    d++;
772   }
773   mpz_set(del, R[0]);
774  #endif
775
776  mpz_t oldR, newR;         //Create and init variables
777  mpz_init( oldR );
778  mpz_init( newR );
779
780
781
782  #if defined DEBUG
783   printf( "\nBeginning␣search\n");
784  #endif
785
786  uint_least64_t pos = 0;       //Start search from char 1
787
788  while( pos < n ){         //Search entire TEXT
789
790   #if defined DEBUG
791    printf("--------------␣%llu", pos);
792   #endif
793
```

```
794   int Ti = mpz_tstbit( TEXT, pos );    //Grab current chars int value
795   int Tii = (Ti + 1) % 2;       //Trick to get the complemented value
796
797   mpz_clear( oldR ); mpz_init( oldR );   //Reset variables
798   mpz_clear( newR ); mpz_init( newR );
799
800   mpz_set( oldR, R[0] );       //Init oldR to cur R[0] (R[i])
801   mpz_set( tmp1, R[0] );
802
803   // Exact match
804   #if defined SHIFTOR
805    mpz_lshift( tmp1, m );      //lshift
806    mpz_ior( tmp1, tmp1, B[Tii] );    //OR with B[Ti]
807   #else
808    mpz_lshift( tmp1, m );      //lshift
809    mpz_setbit( tmp1, 0 );      //OR with 1
810    mpz_and( tmp1, tmp1, B[Ti] );    //AND with B[Ti]
811   #endif
812
813   #if defined DEBUG
814    b = pb(B[Ti],m,0);
815    printf("\n\nB[%d]:␣%s", Ti, b);
816    mpz_out_str(stdout, 2, B[Ti]);
817   #endif
818
819   mpz_set( newR, tmp1 );       //Set newR to tmp
820   mpz_set(R[0], newR);       //Set R[0] to R'[i]
821
822   // Approximate match algorithms
823   #if defined (DEBUG) && defined (SHIFTOR)
824    if(mpz_tstbit(R[0], m-1) > 0){
825     t = pb(R[0],m,1);
826    }
827    else{
828     t = pb(R[0],m,0);
829    }
830    printf("\nR[0]:␣%s", t);
831    mpz_out_str(stdout, 2, R[0]);
832    if( mpz_tstbit(R[0], m-1) == 0 ) printf("␣[!]"); // Print indicator if match
833    printf("\n");
834   #elif defined DEBUG
835    t = pb(R[0],m,0);
836    printf("\nR[0]:␣%s", t);
837    mpz_out_str(stdout, 2, R[0]);
838    if( mpz_tstbit(R[0], m-1) == 1 ) printf("␣[!]"); // Print indicator if match
839    printf("\n");
840   #endif
841
842
843   uint_least64_t i = 1;        //Calc matches with K allowed errors
844   while( i < K ) {
845    mpz_clear( tmp1 );mpz_clear(tmp2);mpz_clear(tmp3);
846    mpz_init(tmp1); mpz_init(tmp2);mpz_init(tmp3); //Reset and initialise temp
         ↪ variables
847
848    #if defined SHIFTOR
849     #if defined CONSTRAINTS2 // OR3 algorithm implementation
850     // Deletion
851     mpz_set(tmp2, oldR);
852     mpz_lshift(tmp2, m);
```

```
853
854     mpz_lshift(del, m);
855     mpz_setbit(del,0);
856     mpz_com(tmp3, del);
857
858     mpz_ior(tmp2, tmp2, tmp3);
859
860     mpz_lshift(dm[i-1], m);
861     mpz_setbit(dm[i-1], 0);
862     mpz_com(tmp3, dm[i-1]);
863
864     mpz_ior(tmp2, tmp2, tmp3);
865
866     mpz_setbit(tmp2, (m-1));
867
868     mpz_set(dm[i], del);
869     // Substitution
870     mpz_set(tmp1, oldR);    //tmp3 = oldR
871     mpz_lshift(tmp1, m);    //tmp3 = <tmp3> << 1
872     mpz_ior(tmp1, tmp1, B[Ti]);   // OR NOT B[]
873
874     mpz_set(oldR, R[i]);    //Store R[i] for next error
875
876     // Match
877     mpz_lshift(R[i], m);
878     mpz_ior(R[i], R[i], B[Tii]);
879
880     mpz_and(R[i], R[i], tmp1);
881     mpz_and(R[i], R[i], newR);
882
883     mpz_set(newR, R[i]);    //newR = <tmp1>
884
885     #elif defined CONSTRAINTS1 // OR2 algorithm implementation
886     // Deletion
887     mpz_lshift(newR, m);
888
889     // Substitution
890     mpz_set(tmp1, oldR);    //tmp3 = oldR
891     mpz_lshift(tmp1, m);    //tmp3 = <tmp3> << 1
892     mpz_ior(tmp1, tmp1, B[Ti]);   // OR NOT B[]
893
894     mpz_set(oldR, R[i]);    //Store R[i] for next error
895
896     // Match
897     mpz_lshift(R[i], m);
898     mpz_ior(R[i], R[i], B[Tii]);
899
900     mpz_and(R[i], R[i], tmp1);
901     mpz_and(R[i], R[i], newR);
902
903     mpz_set(newR, R[i]);    //newR = <tmp1>
904
905     #else // OR1 algorithm implementation
906     mpz_set(tmp1, oldR);
907     mpz_and(tmp1, tmp1, newR); //Del+Sub
908     mpz_lshift(tmp1, m);
909     mpz_set(oldR, R[i]);
910     mpz_lshift(R[i], m);
911     mpz_ior(R[i], R[i], B[Tii]); //Match
912     mpz_and(R[i], R[i], tmp1);
```

```
913     mpz_set(newR, R[i]);
914
915     #endif
916
917
918    #else // Shift-AND implementation
919     mpz_ior(tmp2, oldR, newR);   //tmp2 = (oldR|newR)
920     mpz_lshift(tmp2, m);    //tmp2 = <tmp2> << 1
921
922     mpz_setbit( tmp2, 0 );
923
924     mpz_set(tmp1, R[i]);    //Copy value
925     mpz_lshift(tmp1, m);    //tmp1 = R[i]<<1
926     mpz_and(tmp1, tmp1, B[Ti]);   //tmp1 = <tmp1> & B[Ti]
927
928     mpz_ior(tmp1, tmp1, tmp2);   //tmp1 = <tmp1> | <tmp2>
929
930     mpz_set(newR, tmp1);    //newR = <tmp1>
931     mpz_set(oldR, R[i]);    //Store R[i] for next error
932     mpz_set(R[i], newR);    //R[i] == R'[i]
933    #endif
934
935
936
937    #if defined (DEBUG) && defined (SHIFTOR)
938     if(mpz_tstbit(R[i], m-1) > 0){
939      t = pb(R[i],m,1);
940     }
941     else{
942      t = pb(R[i],m,0);
943     }
944     printf("R[%llu]:␣%s", i, t );
945     mpz_out_str(stdout, 2, R[i]);
946     if( mpz_tstbit(newR, m-1) == 0 ) printf("␣[!]");
947     printf("\n");
948
949    #elif defined DEBUG
950     t = pb(R[i],m,0);
951     printf("R[%llu]:␣%s", i, t );
952     mpz_out_str(stdout, 2, R[i]);
953     if( mpz_tstbit(newR, m-1) == 1 ) printf("␣[!]"); // Print indicator if match
954     printf("\n");
955
956    #endif
957
958     i++;          //Next error
959    }
960
961   mpz_init( MATCHES[pos] );
962   mpz_set_ui( MATCHES[pos], m );    //Init val of match at cur pos
963   int j = 0;        //Init counter
964   #if defined SHIFTOR
965   if( mpz_tstbit(newR, m-1) == 0 ){   //Check if R-table has a match
966   #else
967   if( mpz_tstbit(newR, m-1) == 1 ){   //Check if R-table has a match
968   #endif
969
970   while( j<K ){       //Loop R-table for matches (MSB set)
971     #if defined SHIFTOR
972     if(mpz_tstbit(R[j], m-1) == 0){  //Check if MSB zero
```

```
973     #else
974     if(mpz_tstbit(R[j], m-1) == 1){  //Check if MSB set
975     #endif
976      mpz_set_ui(MATCHES[pos], j); //Set match to the R-level (0-K)
977      j = K;            //Skip to end
978     }
979     j++;              //Next error value
980    }
981   }
982
983   #if defined DEBUG
984   printf("\n");
985   #endif
986   pos += 1;               //Next position in search text
987  }
988
989  #if defined DEBUG
990   printf("Search␣done.\n");
991  #endif
992  mpz_clear( oldR );
993  mpz_clear( newR );
994  free(R);
995
996  return MATCHES;
997
998 }
```

Code listing A.4: <makefile> The main programs makefile

```
1
2  main: clean
3   gcc -DSHIFTOR -o main main.c tpool.c -lgmp -lpthread
4
5  and: clean
6   gcc -o main main.c tpool.c -lgmp -lpthread
7
8  or2: clean
9   gcc -DSHIFTOR -DCONSTRAINTS1 -o main main.c tpool.c -lgmp -lpthread
10
11 or3: clean
12  gcc -DSHIFTOR -DCONSTRAINTS2 -o main main.c tpool.c -lgmp -lpthread
13
14 read: clean
15  gcc -DREADCAND -o main main.c tpool.c -lgmp -lpthread
16
17 demo: clean
18  gcc -DSHIFTOR -DCONSTRAINTS2 -DDEMO -o main main.c tpool.c -lgmp -lpthread
19
20 debug: clean
21  gcc -DSHIFTOR -DDEBUG -o main main.c tpool.c -lgmp -lpthread
22
23 clean:
24  rm -f main *.lib
```

A.3 Analysis framework

Code listing A.5: <analysis.go> Go framework for testing

```
package main

import (
 "fmt"
 "strconv"
 "os/exec"
 "log"
 "math"
 "strings"
 "time"
 "os"
 "io"
 "runtime"
 "encoding/csv"
)

func main() {


 data_path := "./data"          // Local system path to store data files
 cpus := runtime.NumCPU()

 // Get available jobs
 for {
  jobfile := fmt.Sprintf("%s/jobs.txt",data_path)   // Avilable jobs
  ret_job := getJob(jobfile) //

  deg, _ := strconv.Atoi(strings.TrimSpace(ret_job[0]))
  m, _ := strconv.Atoi( strings.TrimSpace(ret_job[1]))
  n, _ := strconv.Atoi( strings.TrimSpace(ret_job[2]))
  k, _ := strconv.ParseFloat( strings.TrimSpace(ret_job[3]), 32)
  stop_m, _ := strconv.Atoi( strings.TrimSpace(ret_job[4]))
  r1, _ := strconv.Atoi( strings.TrimSpace(ret_job[5]))
  r2, _ := strconv.Atoi( strings.TrimSpace(ret_job[6]))
  col_accept, _ := strconv.Atoi( strings.TrimSpace(ret_job[7]))
  mode := strings.TrimSpace(ret_job[8])

  make_cmd := exec.Command("make")

  if mode == "and" {
   make_cmd = exec.Command("make","and")
  } else if mode == "or2" {
   make_cmd = exec.Command("make","or2")
  } else if mode == "or3" {
   make_cmd = exec.Command("make","or3")
  }

  err := make_cmd.Run()
  if err != nil {
   log.Fatalf("FATAL␣ERROR:␣Build␣failed␣%s\n", err)
  return
  }

  n=2*m

  var status string
  var logline string

```

```
59  max := int(math.Exp2(float64(deg)))
60
61  fname := fmt.Sprintf("%s/%s_%d_%d_%d_%d_%d_%d.log",    // Logfile name
62   data_path, mode, deg, r1, r2, m, n, col_accept)
63
64
65  if n < m*2 {
66   fmt.Printf("Search␣text␣too␣short...\n")
67   continue
68  }
69
70  for m <= stop_m {
71   n=2*m
72   bottom := 0
73   var resmap = make([]int, m-1)
74   var i int = int(float32(m)/float32(k))
75
76   SeekErrors:
77    for true {
78     if resmap[i] == 1 {
79      i++
80      continue SeekErrors
81     }
82     search_start := time.Now()
83     cmd := exec.Command("./main", strconv.Itoa(deg), strconv.Itoa(m), strconv.Itoa
         ↪ (n), strconv.Itoa(i), strconv.Itoa(r1), strconv.Itoa(r2), strconv.Itoa
         ↪ (col_accept), strconv.Itoa(cpus))
84     b, _ := cmd.Output()
85     duration := time.Since(search_start)
86     s := strings.Split(string(b), ",")
87     res, _ := strconv.Atoi(s[0])
88     cands, _ := strconv.Atoi(s[1])
89
90     if res >= 1 {
91      status = "invalid_no_r2"
92      fmt.Printf("[m=%d,n=%d,k=%d]␣FAILED:␣Set␣of␣%d/%d␣contains␣no␣actual␣R2STATE
         ↪ ...␣\n", m, n, i, cands,max)
93     } else if res == -1 {
94      status = "invalid_zero_set"
95      bottom = 1
96      fmt.Printf("[m=%d,n=%d,k=%d]␣FAILED:␣Zero␣candidates...␣\n", m, n, i)
97     } else if res == -2 {
98      status = "invalid_full_set"
99      fmt.Printf("[m=%d,n=%d,k=%d]␣FAILED:␣Too␣many␣candidates...␣\n",m, n,  i)
100    } else if res == -3 {
101     status = "invalid_collisions"
102     fmt.Printf("[m=%d,n=%d,k=%d]␣FAILED:␣Set␣of␣%d/%d␣contains␣collisions...␣\n",
         ↪ m, n, i,cands,max)
103    } else if res == 0 {
104     status = "valid"
105     fmt.Printf("[m=%d,n=%d,k=%d]␣SUCCESS:␣Set␣of␣%d/%d␣is␣valid!\n",m, n, i,cands
         ↪ ,max)
106     // If success on first run, save the parameters and decrease. Skip ahead to a
         ↪  set value when done
107    }
108    resmap[i] = 1
109    logline = fmt.Sprintf("%d,%d,%d,%d,%s,%d,%d\n",m,n,i,cands,status,duration,
         ↪ cpus)
110
111    writeLog(fname, logline)
```

```
112
113     // m,n,k, invalid_full_set/invalid_zero_set/invalid_no_r2/invalid_collisions/
            ↪ valid
114     if res >= 0 && bottom == 0 {
115      i--
116     } else if res == -2 && bottom == 0 {
117      i--
118     } else if res >= 1 {
119      i++
120     } else if res == -1 {
121      i++
122     } else if res == -2 {
123      break
124     } else if res == -3 {
125      break
126     }
127
128    }
129   m=m+20
130  }
131 }
132 fmt.Printf("Done\n")
133 os.Exit(0)
134 }
135
136 func writeLog(fname string, logline string) {
137  f, err :=  os.OpenFile(fname, os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0666); // File
         ↪ to hold return values for later statistics
138  if err != nil {
139   panic(err)
140  }
141  if _, err := f.WriteString(logline); err != nil {
142   panic(err)
143  }
144  f.Close()
145
146 }
147
148 func getJob(jobfile string) []string {
149  csvfile, err := os.Open(jobfile)
150  if err != nil {
151   log.Fatalln("Couldn't␣open␣the␣csv␣file", err)
152  }
153  os.Remove(jobfile)
154  newfile, err  := os.OpenFile(jobfile, os.O_CREATE|os.O_EXCL|os.O_WRONLY, 0666)
155  r := csv.NewReader(csvfile)
156  i := 0
157
158  ret_job, err := r.Read()
159  if err == io.EOF {
160   fmt.Printf("No␣more␣jobs,␣shutting␣down...\n")
161   os.Exit(0)
162  }
163  if err != nil {
164   log.Fatal(err)
165  }
166
167  for {
168   job, err := r.Read()
169   if err == io.EOF {
```

```
    break
   }
   if err != nil {
    log.Fatal(err)
   }
   writer := csv.NewWriter(newfile)
   writer.Write(job)
   writer.Flush()
   i++
  }

  return ret_job

}
```

Code listing A.6: <jobs.txt> Sample job-file for analysis.go

```
11,40,80,4,40,100,100,0,or1
11,40,80,4,40,100,100,0,or2
11,40,80,4,40,100,100,0,or3
```

A.4 AWS Launch template

Code listing A.7: AWS Launch template Bash script

```
#!/bin/bash
apt-get update
apt-get install -y libgmp-dev build-essential nfs-common golang-go
cd /home/ubuntu
su ubuntu -c "git␣clone␣-b␣evaluate␣https://github.com/philedem/BRM_Code"
cd BRM_Code/evaluation/src
mkdir data
mount -t nfs4 -o rsize=1048576,wsize=1048576,hard,timeo=60,retrans=2,noresvport \
fs-xxxxxxxx.efs.eu-north-1.amazonaws.com:/ /home/ubuntu/BRM_Code/evaluation/src/
    ↪ data
chown ubuntu data
chgrp ubuntu data
su ubuntu -c "go␣run␣analysis.go"
```

Appendix B

Data

B.1 Evaluation data

As the quantitative evaluation data set contains large amounts of data, this is included separately in the provided **.zip**-file which also contains the source code related to this thesis. The filename is **evaluation.xlsx**.

B.2 Performance data

Single-core performance, p=11

Shift-AND (AND)

m	k	time (s)	C	valid
100	25	13.41	580	0
120	30	21.14	602	0
140	35	32.03	485	1
160	40	46.43	187	1
180	45	62.42	0	0
200	50	80.91	452	1
220	55	110.04	0	0
240	60	139.18	0	0
260	65	177.64	0	0
280	70	213.89	0	0
300	75	256.00	0	0

Shift-OR (OR1)

m	k	time (s)	C	valid
100	25	12.42	580	0
120	30	20.45	602	0
140	35	31.27	485	1
160	40	45.35	187	1
180	45	61.83	0	0
200	50	83.41	452	1
220	55	109.73	0	0
240	60	133.31	0	0
260	65	172.4	0	0
280	70	210	0	0
300	75	250	0	0

Shift-OR2 (OR2)

m	k	time (s)	C	valid
100	25	17.75	580	0
120	30	29.92	602	0
140	35	45.40	485	1
160	40	64.13	187	1
180	45	89.35	0	0
200	50	122.01	452	1
220	55	156.25	0	0
240	60	204.49	0	0
260	65	264.06	0	0
280	70	325.50	0	0
300	75	396.66	0	0

Shift-OR3 (OR3)

m	k	time (s)	C	valid
100	25	34	0	0
120	30	56	0	0
140	35	87	0	0
160	40	129	0	0
180	45	179	0	0
200	50	240	0	0
220	55	311	0	0
240	60	400	0	0
260	65	503	0	0
280	70	622	0	0
300	75	765	0	0

Single-core performance, p=16

Shift-AND (AND)

m	k	time (s)	C	valid
100	25	386	6488	0
120	30	617	3811	0
140	35	941	1732	0
160	40	1367	726	0
180	45	1849	612	0
200	50	2460	227	0
220	55	3208	0	0
240	60	4081	0	0
260	65	5095	296	0
280	70	6380	318	0
300	75	7622	338	0

Shift-OR (OR1)

m	k	time (s)	C	valid
100	25	421	6488	0
120	30	670	3811	0
140	35	1043	1732	0
160	40	1505	726	0
180	45	2075	612	0
200	50	2794	227	0
220	55	3621	0	0
240	60	4767	0	0
260	65	5868	296	0
280	70	7251	318	0
300	75	8895	338	0

Multi-core performance (16), p=11				
Shift-AND (AND)				
m	k	time (s)	C	valid
100	25	1.01	580	0
120	30	1.69	602	0
140	35	2.59	485	1
160	40	3.81	187	1
180	45	5.29	0	0
200	50	7.18	452	1
220	55	9.48	0	0
240	60	12.18	0	0
260	65	15.33	0	0
280	70	18.95	0	0
300	75	23.20	0	0
Shift-OR (OR1)				
m	k	time (s)	C	valid
100	25	1.12	580	0
120	30	1.98	602	0
140	35	3.03	485	1
160	40	4.44	187	1
180	45	6.26	0	0
200	50	8.44	452	1
220	55	11.11	0	0
240	60	14.37	0	0
260	65	18.09	0	0
280	70	22.52	0	0
300	75	27.59	0	0
Shift-OR2 (OR2)				
m	k	time (s)	C	valid
100	25	1.71	580	0
120	30	2.88	602	0
140	35	4.48	485	1
160	40	6.56	187	1
180	45	9.25	0	0
200	50	12.54	452	1
220	55	16.58	0	0
240	60	21.33	0	0
260	65	27.01	0	0
280	70	33.60	0	0
300	75	41.02	0	0
Shift-OR3 (OR3)				
m	k	time (s)	C	valid
100	25	3.16	0	0
120	30	5.34	0	0
140	35	8.38	0	0
160	40	12.28	0	0
180	45	17.29	0	0
200	50	23.65	0	0
220	55	31.22	0	0
240	60	40.22	0	0
260	65	50.88	0	0
280	70	63.5	0	0
300	75	77.61	0	0

Multi-core performance (16), p=16				
Shift-OR (OR1)				
m	k	time (s)	C	valid
100	25	37.77	6488	0
120	30	63.25	3811	0
140	35	97.65	1732	0
160	40	143.14	726	0
180	45	201.39	612	0
200	50	272.78	227	0
220	55	360.18	0	0
240	60	463	0	0
260	65	583	296	0
280	70	725	318	0
300	75	887	338	0
Shift-OR (OR2)				
m	k	time (s)	C	valid
100	25	55	6488	0
120	30	93	3811	0
140	35	144	1732	0
160	40	212	726	0
180	45	298	612	0
200	50	405	227	0
220	55	534	0	0
240	60	688	0	0
260	65	869	296	0
280	70	1081	318	0
300	75	1320	338	0
Shift-OR2 (OR3)				
m	k	time (s)	C	valid
100	25	101	0	0
120	30	171	0	0
140	35	265	0	0
160	40	392	0	0
180	45	552	0	0
200	50	753	0	0
220	55	995	0	0
240	60	1285	0	0
260	65	1622	0	0
280	70	2018	0	0
300	75	2472	0	0

Adrian Schjelderup Evensen

Reconstruction of clock sequences in irregularly clocked pseudorandom sequence generators

NTNU
Kunnskap for en bedre verden