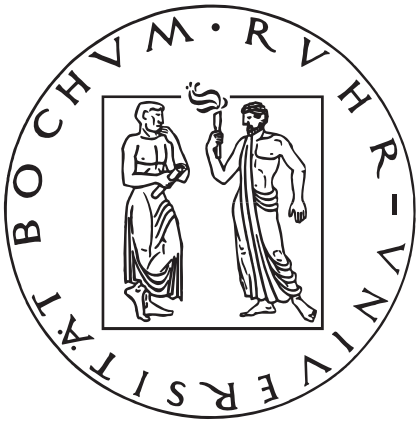
Entwicklung einer Toolbox zur Generierung und Analyse von Stromchiffren unter CrypTool 2.0

Schriftliche Prüfungsarbeit
für die Master-Prüfung des Studiengangs
IT-Security
an der Ruhr-Universität Bochum

vorgelegt von:

Sören Rinne



14. Januar 2010

Themensteller: Prof. Dr.-Ing. Frederik Armknecht
2. Prüfer: Prof. Dr. rer. nat. Jörg Schwenk

Erklärung

Ich erkläre, dass das Thema dieser Arbeit nicht identisch ist mit dem Thema einer von mir bereits für ein anderes Examen eingereichten Arbeit.

Ich erkläre weiterhin, dass ich die Arbeit nicht bereits an einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen der Entlehnung kenntlich gemacht. Dies gilt sinngemäß auch für gelieferte Zeichnungen, Skizzen und bildliche Darstellungen und dergleichen.

Datum	Unterschrift

Contents

List of Figures

List of Tables

1. Introduction

There is still a big industrial demand for secure and efficient stream ciphers. Stream ciphers are well suited to perform in narrow-bandwidth and voice-coding environments like the mobile/wireless communications sector. Stream ciphers do not expand messages, are tolerant to bit errors, and are generally faster in hardware compared to block ciphers.

In science, there is an huge interest to get a better understanding of designing a stream cipher, since many of the proposed schemes in the past have been under more or less serious attacks. The NESSIE (New European Schemes for Signatures, Integrity and Encryption) project[1], which was comparable to the NIST (National Institute of Standards and Technology) AES (Advanced Encryption Standard) process, is a good example: all six submitted ciphers failed to withstand cryptanalysis. Therefore in 2004 the ECRYPT eSTREAM project[2] was launched to identify promising new stream ciphers, that might become suitable for widespread adoption. The project was completed in 2008 and the eSTREAM portfolio now contains four ciphers suited for software implementation (Profile 1) and three ciphers for hardware applications (Profile 2). Since both profiles contained 48 ciphers at the beginning, this shows again, that many of the submitted stream ciphers did not endure cryptanalysis. Due to this lack of knowledge on building strong stream ciphers, we propose a toolbox, which allows the user to design and analyze stream ciphers easily. Our toolbox is not only meant for scientific use to design new stream ciphers, but also for didactic use to introduce students to the fundaments and underlying techniques of stream ciphers in an appealing way.

The well-known "Handbook of Applied Cryptography" (HAC) [10] states, that *"feedback shift registers, in particular linear feedback shift registers, are the basic components of many keystream generators"*. In the present, more than 10 years after the HAC has been released initially, this statement is still true even for some of the ciphers in the current eSTREAM portfolio. On that account our toolbox' main components are both linear and nonlinear feedback shift registers. These feedback shift registers produce a (periodic) binary sequence, which can be combined by a boolean function to a keystream. Another component, which is closely related to shift registers, is the Berlekamp-Massey algorithm (BMA). The BMA is an efficient algorithm for determining the linear complexity of a finite binary sequence. Or, in other words, the BMA computes the shortest linear feedback shift register (LFSR), which produces a given binary sequence. The third main component of the toolbox is the Boolean Function Parser (BFP). The BFP solves a function with given variables and calculates the output bit.

The proposed toolbox is part of the software CrypTool 2.0 – the modern successor of the widespread e-learning platform for cryptography and cryptanalysis CrypTool. With its modern plug'n'play interface and plugin-based architecture, CrypTool 2.0 emphasizes the toolbox character of our approach. Each component can be connected with any other component with the same connection type on a workspace inside the application. By combining different tools, a stream cipher can be copied or build from scratch inside CrypTool 2.0.

[1] http://http://www.cryptonessie.org
[2] http://www.ecrypt.eu.org/stream/

Our toolbox is a foundation for cryptanalysis components. Other works presented by Oruba [12] and Bourbahh [3] make use of functions described in this work. Oruba implemented the cube attack of Dinur and Shamir [6] in CrypTool 2.0 and uses our Boolean Function Parser component as a blackbox. Bourbahh implemented algebraic attacks which can be used to test and attack stream ciphers.

In our work, we present an implementation of a toolbox containing basic components for assembling stream ciphers. Our toolbox can be extended in future work with new plugins containing attacks or other primitive components at any time in an easy way.

1.1. Related Work

There has been a similar approach in providing an educational environment for stream ciphers in March 2009. The work "Stream cipher for education - Implementierung einer Lernumgebung in JCrypTool" of Christian Wagner [16] implements a component of stream ciphers in the Java variant of CrypTool named "JCrypTool".

We will discuss our toolbox in comparison to the here mentioned works later on in Chapter 6.3 on page 59.

Another work which focusses on the educational presentation of LFSRs is the project of Christoph Bertram [1]: "Entwurf und Implementierung eines Werkzeugs zum Rechnen mit Schieberegisterfolgen". His work includes also the BMA and implements logical operators to combine bitstreams of several LFSRs. The implementation is also made in Java and has a graphical user interface based on the model view control (MVC).

1.2. Organization of the thesis

This work is divided into five main parts:

1. Mathematical and Cryptographic Background (Chap. 2, p. 3)
2. An Introduction to CrypTool 2.0 (Chap. 3, p. 11)
3. The Implementation (Chap. 4, p. 17)
4. Practical Scenarios Based on the Toolbox (Chap. 5, p. 43)
5. Conclusion and Further Work (Chap. 6, p. 57)

The mathematical background of the components of our toolbox is presentend in Chapter 2. Since our focus is on the implementation, we only provide a rather short mathematical background. Chapter 3 provides an introduction to our host application CrypTool 2.0: we give an overview of its history first, then take a closer look at the underlying architecture, and finally introduce the user interface of CrypTool 2.0. The implementation of the components is described in Chapter 4 including a description of a generalized CrypTool 2.0 plugin. Each plugin is presented with its in- and outputs, settings, and main methods. Chapter 5 provides some kind of handbook with tutorials on using the toolbox. We build scenarios in which the user is introduced to the features of each of the plugins. Finally, we draw a conclusion in Chapter 6, compare our implementation with the work of others, and give some proposals on enhancing the toolbox.

The source code can be found at `http://www.soerenrinne.de/MA/sources.zip` and on the CD attached.

2. Mathematical and Cryptographic Background

In this chapter we introduce the underlying theory of our toolbox. We give some short mathematical background on stream ciphers, introduce feedback shift registers and the Berlekamp-Massey algorithm, and finally present three kinds of keystream generators.

2.1. Stream Ciphers

At first we would like to introduce the basic functionality of a **stream cipher**, as we aim to design them with the help of our toolbox. Ciphers in general can be divided into two main groups: block ciphers and stream ciphers. **Block ciphers** encrypt (or decrypt) blocks of plaintext of a defined length at a time. In contrast, stream ciphers do not operate on blocks, but on single characters. Usually these characters are binary digits. So stream ciphers can be seen as a special case of block ciphers with block length $= 1$.

Let us now look at a more formal description of stream ciphers: Suppose, we have the alphabet $\sum = \{0, 1\}$. A word is defined by a combination of symbols from the alphabet $\sum$. The set of all possible words over $\sum$ is denoted as $\sum^*$. In our case, plaintext, ciphertext, and key are all elements of $\sum^*$. The key set is $\sum^n$ for a key length of n bits. Words in $\sum^*$ will be encrypted symbol by symbol in the following way: Let $k = (k_0, \ldots, k_{n-1})$ be a key and $w = \sigma_0 \ldots \sigma_{m-1}$ a word of length m in $\sum^*$. To encrypt a word, we need to generate a keystream $z = z_0, \ldots, z_{m-1}$. The functions for encryption E_k and decryption D_k are given by the following equation:

$$E_k(w) = D_k(w) = \sigma_0 \oplus z_0, \ldots, \sigma_{m-1} \oplus z_{m-1}. \tag{2.1}$$

The word that we want to encrypt is XORed bitwise with the keystream. Note that encryption and decryption are completely the same operation. We define an encrypted word as **ciphertext** $y = y_1 \ldots y_m$ as

$$y_i = E_k(\sigma_i). \tag{2.2}$$

The keystream z used for encryption and decryption is the main issue for the security of stream ciphers. The central requirement, as stated in [13], is identified as the randomness of the keystream bits. To any person, that does not know the key, the keystream should appear like a random sequence.

A cipher that makes use of a complete random sequence as the keystream is the **One-Time Pad** (OTP). OTP cannot be broken even with infinite computational resources and therefore can be defined as **unconditional secure**. Why can we claim that? For every ciphertext bit we get an equation as follows:

$$\begin{aligned} y_0 &= \sigma_0 \oplus z_0 \\ y_1 &= \sigma_1 \oplus z_1 \\ &\vdots \end{aligned}$$

Each relation is a linear equation modulo 2 with two unknowns. They are impossible to solve if the keystream is truly random.

In real scenarios, it is not practicable to apply a truly random sequence, because the key has to be as long as the plaintext to be encrypted. Instead of true random number generators (TRNGs) pseudo-random number generators (PRNGs) are used. These generators produce a stream, that looks like it was generated randomly. This leads us to the main components of our toolbox: An example of devices that produce such pseudo-random streams are feedback shift registers (FSRs). FSRs consist of a shift register and a feedback function. In case of a linear feedback function, we refer to a **linear feedback shift register** (LFSR). Otherwise, if the feedback function is nonlinear, the FSR is called **nonlinear feedback shift register** (NLFSR). Both FSRs are presented in the following.

2.2. Linear Feedback Shift Registers (LFSRs)

An LFSR of length L consists of L stages numbered by $0, 1, \ldots, L-1$. Each **stage** (or delay element) has one input and one output and can store one bit of data. The whole LFSR is connected to a clock which controls the movement of data. During each clock cycle the following operations are performed:

(i) The data of stage 0 is output and forms a part of the output sequence,

(ii) the data of stage i is moved to stage $i-1$ for each $i, 1 \leq i \leq L-1$, and

(iii) the new data of stage $L-1$ is given by the feedback bit s_j, which is the result of XORing the previous data of a fixed subset of stages $0, 1, \ldots, L-1$.

The fixed subset of stages that are added together is affected by c_i, where each of them is either 0 or 1. The **feedback bit** s_j is the result of XORing the data of those stages $i, 0 \leq i \leq L-1$, for which $c_{L-i} = 1$. Figure 2.1 depicts the general setup of an LFSR.

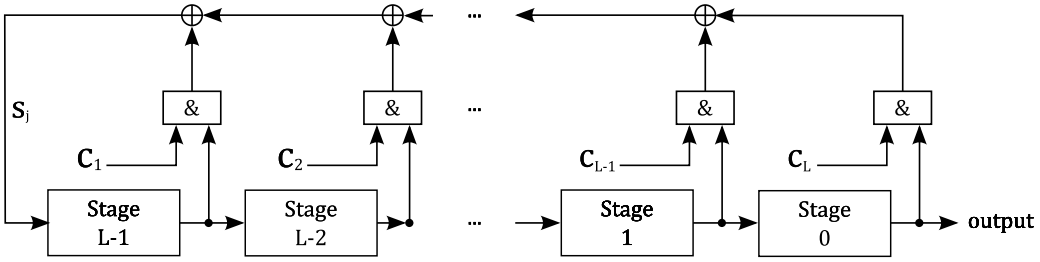


Figure 2.1.: Linear feedback shift register (LFSR) of length L [10]

The LFSR in Figure 2.1 is denoted by $\langle L, C(D) \rangle$, where $C(D) = 1 + c_1 D + c_2 D^2 + \ldots + c_L D^L \in \mathbb{Z}_2[D]$ represents the **feedback polynomial**. The initial content s_i of stage i for each $i, 0 \leq i \leq L-1$ is called **seed** (or initial state) of the LFSR. Now we can define our recursion, which produces the output sequence (keystream) $s = s_0, s_1, s_2, \ldots$, more generally as

$$s_j = (c_1 s_{j-1} + c_2 s_{j-2} + \ldots + c_L s_{j-L}) \ mod \ 2, \quad j \geq L. \tag{2.3}$$

If the seed is $[0, 0, \ldots, 0]$, the output sequence is the **zero sequence** $s = 0, 0, \ldots$.

The **linear complexity** of an infinite binary sequence s, denoted by $L(s)$, is defined as follows:

(i) If s is the zero sequence $s = 0, 0, \ldots$, then $L(s) = 0$,

(ii) if no LFSR generates s, then $L(s) = \infty$,

(iii) otherwise, $L(s)$ is the length of the shortest LFSR that generates s.

The linear complexity of a finite binary sequence s^n, denoted by $L(s^n)$, is the length of the shortest LFSR that generates a sequence having s^n as its first n terms.

Example 2.2.1 As given in [4], an example of an elementary keystream generator could work as follows: The keystream z is gained by setting

$$z_i = s_i, \quad 0 \leq i \leq n-1. \tag{2.4}$$

In case of $m > n$ we set

$$z_i = c_1 z_{i-1} + c_2 z_{i-2} + \ldots + c_n z_{i-n} \; mod \; 2, \quad n < i < m, \tag{2.5}$$

where $c_1, \ldots, c_n$ are fixed bits in our keystream generator. This equation is called a **linear recursion** of degree n.

Example 2.2.2 As a second example, we take a keylength of $n = 4$. The keystream is generated according to the recursion

$$z_{i+4} = z_i + z_{i+1} \; mod \; 2. \tag{2.6}$$

It can be followed from the recursion that $c_1 = c_2 = 0, c_3 = c_4 = 1$. Let the key ($\widehat{=}$ seed) be $k = (1, 0, 0, 0)$. We now get a keystream of

$$1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, \ldots$$

Notice that the keystream is periodic with a length of 15: $z_i = z_{i+15}$.

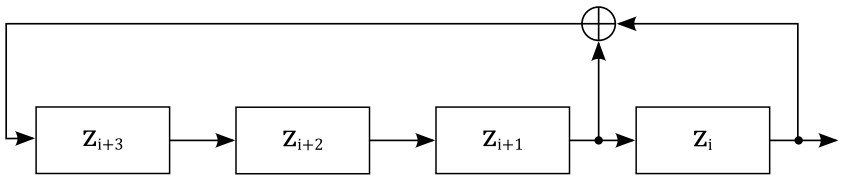


Figure 2.2.: An elementary stream cipher based on a linear feedback shift register (LFSR).

The registers in Figure 2.2 contain the next four values of the keystream. In every step the key from the right most register is used for encryption. The others are shifted to the right inside the next stage. The new left most stage results from adding the stages mod 2 whose coefficient c_i is 1. In our case c_3 and c_4 are 1, so this results to the new key z_{i+4} by adding z_i to z_{i+1} (mod 2), which corresponds to the recursion defined in Equation 2.6.

2.3. Berlekamp-Massey Algorithm

An efficient algorithm for determining the linear complexity $L(s)$ of a finite binary sequence is the **Berlekamp-Massey algorithm** (BMA). The algorithm takes n iterations, with the Nth iteration computing the linear complexity of the subsequence s^N consisting of the first N terms of s^n. The running time of the Berlekamp-Massey algorithm is $O(n^2)$, where n is the length of a binary sequence. The pseudo code of the BMA is shown in Algorithm 1.

Algorithm 1 Berlekamp-Massey algorithm [10]

Require: a binary sequence $s^n = s_0, s_1, s_2, \ldots, s_{n-1}$ of length n.
Ensure: the linear complexity $L(s^n)$ of $s^n, 0 \leq L(s^n) \leq n$.
1: *Initialization.* $C(D) \leftarrow 1, L \leftarrow 0, m \leftarrow -1, B(D) \leftarrow 1, N \leftarrow 0$.
2: **while** $(N < n)$ **do**
3: *Compute the next discrepancy d.* $d \leftarrow (s_N + \sum_{i=1}^{L} c_i s_{N-i}) \; mod \; 2$.
4: **if** $d = 1$ **then**
5: $T(D) \leftarrow C(D), C(D) \leftarrow C(D) + B(D) \cdot D^{N-m}$.
6: **if** $L \leq N/2$ **then**
7: $L \leftarrow N + 1 - L, m \leftarrow N, B(D) \leftarrow T(D)$.
8: **end if**
9: **end if**
10: $N \leftarrow N + 1$.
11: **end while**
12: **return** (L).

If s is an infinite binary sequence of linear complexity $L(s)$, and t is a finite subsequence of s of length at least $2L$, then the BMA on input t determines an LFSR of length L which generates s. This only applies if the return value in line 12 is modified to return both L and $C(D)$.

2.4. Nonlinear Feedback Shift Registers (NLFSRs)

As stated above, a **nonlinear feedback shift register** (NLFSR) is the analog counterpart of an LFSR. An NLFSR of length L consists of L stages, each of them storing again one bit of data, having one input and one output, and a clock which controls the movement of data. Only step (iii) of the operations performed each clock cycle is different (compare to Section 2.2):

(i) The data of stage 0 is output and forms a part of the output sequence,

(ii) the data of stage i is moved to stage $i-1$ for each $i, 1 \leq i \leq L-1$, and

(iii) the new data of stage $L-1$ is given by the feedback bit $s_j = f(s_{j-1}, s_{j-2}, \ldots, s_1, s_0)$, where the feedback function f is a nonlinear boolean function and s_{j-i} is the previous data of stage $L-1, 1 \leq i \leq L$.

A **boolean function** of n variables is a function with n binary inputs and one binary output.

Figure 2.3 depicts a general FSR. Note that if the feedback function f is a linear function, then the FSR is an LFSR.

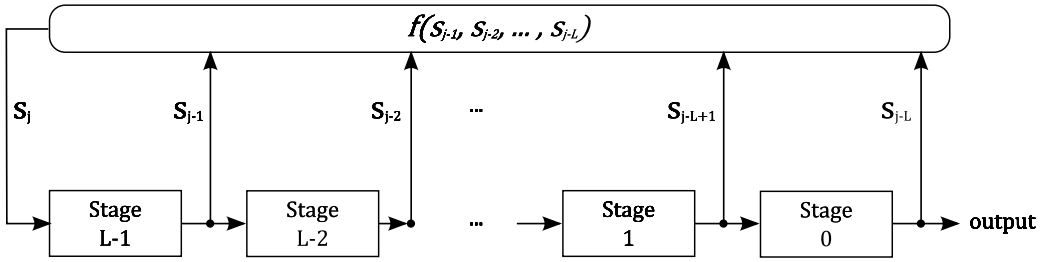


Figure 2.3.: A feedback shift register (FSR) of length L [10]

2.5. Using FSRs as Keystream Generators for Stream Ciphers?

As mentioned in the beginning, LFSRs are the basic building block in a lot of stream ciphers that have been proposed. They are well-suited for hardware implementation, produce sequences having large periods and good statistical properties, and are readily analyzed using algebraic techniques. The disadvantage of LFSRs is their linearity. They are easily predictable, as we want to show now.

We assume that the output sequence s of an LFSR has the linear complexity $L(s)$. The feedback polynomial $C(D)$ of an LFSR of length L that generates s can be efficiently determined using the BMA from any subsequence t of s (having length at least $2L$). After getting $C(D)$ from the BMA, an LFSR $\langle L, C(D)\rangle$ can be initialized (seeded) with any substring of t having length L to output the remainder of the sequence s. An adversary may obtain the required subsequence t by mounting a known or chosen-plaintext attack[1] on the stream cipher. If the adversary knows the plaintext subsequence $\sigma_0, \sigma_1, \ldots, \sigma_{n-1}$ and the corresponding ciphertext sequence $y_0, y_1, \ldots, y_{n-1}$, he is able to compute the corresponding keystream z by using

$$z_i = \sigma_i \oplus y_i, \quad 0 \leq i \leq n-1. \tag{2.7}$$

Since we do not want our stream cipher to be an easy victim of known-plaintext attacks, an LFSR should never be used by itself to generate a keystream. Using NLFSRs instead of LFSRs is a step forward, but not enough. There are three general techniques for utilizing LFSRs and NLFSRs in the design of stream ciphers by destroying the unwanted linearity properties as stated in [10]:

(i) using a nonlinear combining function on the outputs of several FSRs,

(ii) using a nonlinear filtering function on the contents of a single FSR, and

(iii) using the output of one (or more) FSRs to control the clock of one (or more) other FSRs.

In the following we present the three techniques in detail and give examples of stream ciphers that make use of them.

[1] See [10], §1.13.1

2.6. Nonlinear Combination Generators

A **nonlinear combination generator** computes a keystream by combining the outputs of several FSRs. Figure 2.4 shows a sample layout of a keystream generator using a nonlinear combining function and LFSRs.

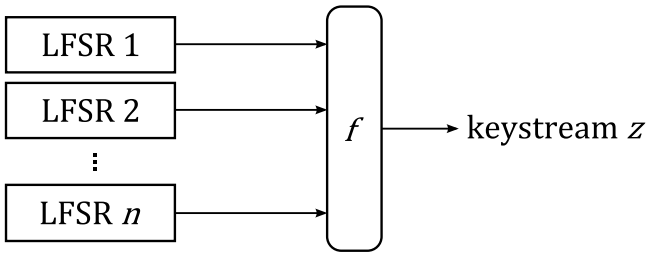


Figure 2.4.: Simplified layout of a keystream generator using a nonlinear combining function f.

Again, f is a boolean function with n binary inputs produced by the LFSRs and one binary output z_i at one clock cycle which forms the keystream sequence $(z_0, z_1, \ldots)$.

An example for this method is the stream cipher Achterbahn-128/80 [8] with a key size of 128 or 80 bits, which has been proposed to the ECRYPT eSTREAM Project. Achterbahn-128 uses 13 NLFSRs which output into a boolean combining function of degree 4. Achterbahn-80 uses 11 NLFSRs. All NLFSRs have a length L between 21 and 33. Figure 2.5 shows the layout as depicted in [8] (with slight changes concerning the arrangement of the components).

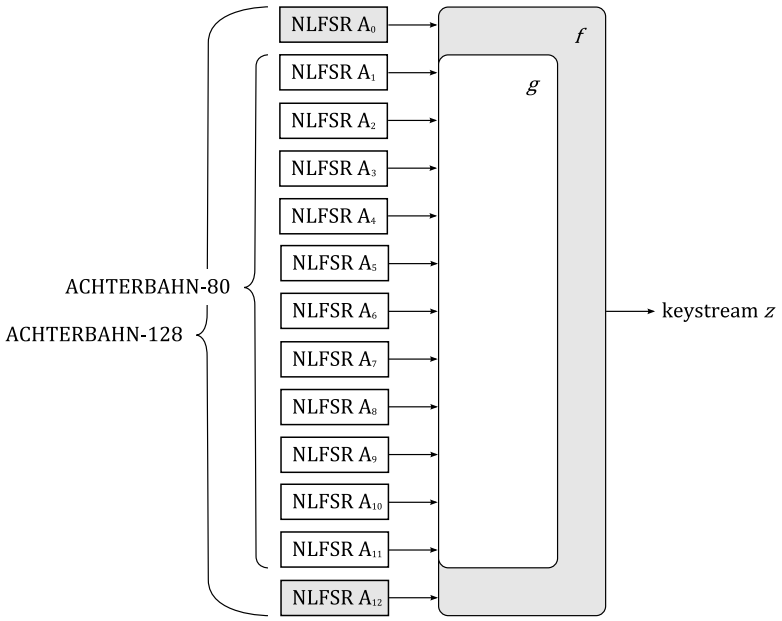


Figure 2.5.: Achterbahn stream cipher using a nonlinear combining function f or g.

2.7. Nonlinear Filter Generators

A **nonlinear filter generator** generates its keystream by combining the stages of an FSR in one function. Figure 2.6 shows a sample layout of a keystream generator of length $L = 4$ using a nonlinear filtering function f.

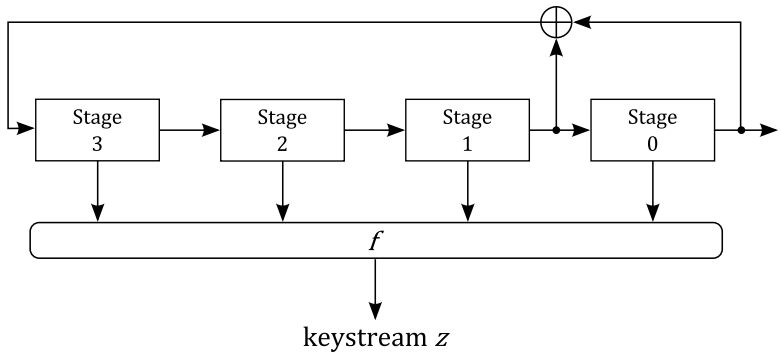


Figure 2.6.: Layout of a sample stream generator using a nonlinear boolean filtering function f.

In this case the boolean function f gains its variables from the stages of the LFSR.

An example for gaining the keystream out of the stages is the stream cipher TRIVIUM [5]. TRIVIUM has a key size of 80 bits and uses three NLFSRs with a combined length of 288 stages $s_1, \ldots, s_{288}$. A specific feature of TRIVIUM are the feedback functions that combine stages of two NLFSRs to compute the feedback bit s_j of a single NLFSR (see Figure 2.7).

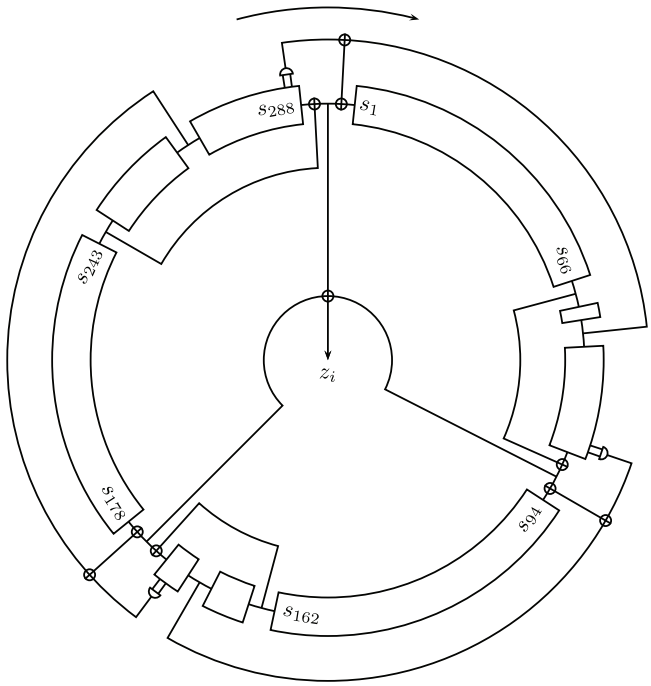


Figure 2.7.: Layout of TRIVIUM with its three intertwining FSRs [5].

2.8. Clock-Controlled Generators

Clock-controlled generators make use of a clock that steps one or more FSRs. The output of an FSR in combination with the clock signal can also be used to step FSRs. Figure 2.6 shows a sample layout of a clock-controlled keystream generator with LFSRs.

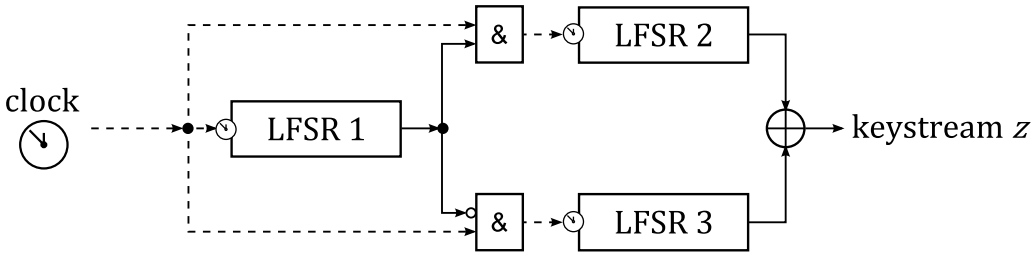


Figure 2.8.: The alternating step generator.

The above shown generator is a so-called alternating step generator. LFSR 1 is used to control the stepping of two downstreamed LFSRs, LFSR 2 and LFSR 3. The keystream is produced by computing the XOR of the output sequences of LFSR 2 and LFSR 3.

An example which makes use of irregular clocked LFSRs is the stream cipher A5/1 [2]. A5/1 consists of three LFSRs whose stepping is controlled indirectly by the two other LFSRs. If an LFSR steps or not depends on the majority function $m(s_{LFSR_i})$ of specific stages $s_{LFSR_{1,\ldots,3}}$ of all three LFSRs:

$$m(s_{LFSR_i}) = \begin{cases} 1, & \text{if } \sum_{i=1}^{3} s_{LFSR_i} > 1, \\ 0, & \text{else.} \end{cases} \tag{2.8}$$

An LFSRs steps if its stage s agrees with $m(s_{LFSR_i})$. The specific stage of an LFSR has to agree with one or both specific stages of the other two LFSRs to be clocked. Figure 2.9 depicts the A5/1 cipher.

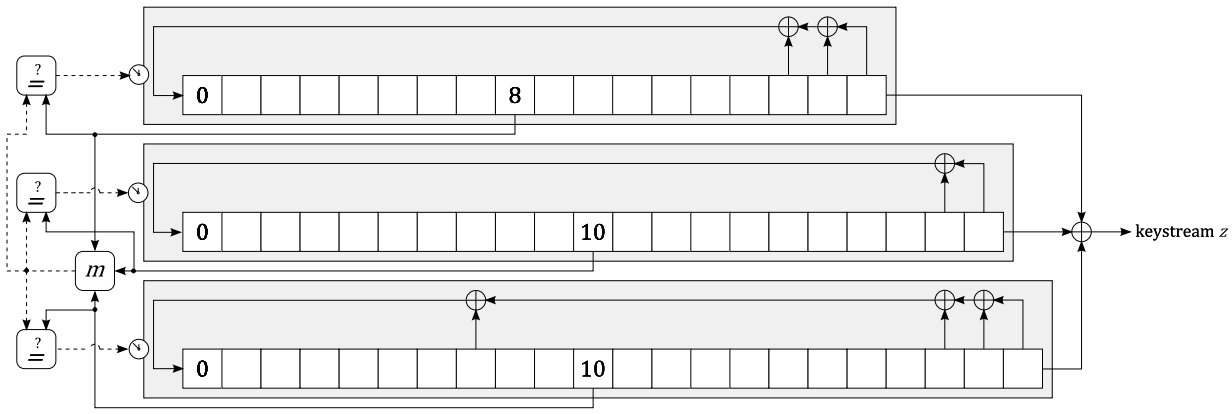


Figure 2.9.: Stream cipher A5/1 with three LFSRs whose steppings depend on the majority function m.

3. An Introduction to CrypTool 2.0

3.1. History of CrypTool

In our toolbox we implemented the above mentioned components that are commonly used to build a stream cipher. The toolbox is integrated in the application CrypTool 2.0. To get an idea what CrypTool 2.0 represents, we first have to start with its predecessor CrypTool.

The history of CrypTool started in 1998 [7], where it was developed in a bank for in-firm training. The software was part of an awareness program in which employees where taught to gain a certain understanding and awareness for IT security. Besides the company and its employees, the CrypTool project was presented to another audience. The founder of CrypTool, Bernhard Esslinger, started a cooperation with many universities, where CrypTool was used to improve education. The universities themselves started to develop CrypTool further and the project became a more and more media didactic approach by still orienting towards the common standards in IT security. To provide the project to a broader audience, CrypTool became freeware in the year 2000. Three years later, in 2003, the program was given to the community as open source. The program itself and the corresponding website with its enormous source of information concerning cryptography and cryptology were hosted by the University of Darmstadt. Since 2007 CrypTool is, besides German, also available in English, Polish and Spanish language. CrypTool is now developed by people from companies and universities in different countries from all over the world.

After 8 years in 2006, the planning of a novel successor of CrypTool began. Moreover, a technology changeover was needed because CrypTool 1.4.x was based on the outdated GUI library Microsoft Foundation Classes (MFC) which no longer met the project requirements. The project founders commissioned a lot of research on that topic including the development of visual prototypes and experimental user interfaces made by the University of Koblenz and the Aachen University of Applied Sciences. In 2007, two new projects where founded to take CrypTool to the next level: A Java-based approach named JCrypTool (also known as JCT) and a .NET approach named CrypTool 2.0 [15].

In November of 2007, the initial team gathering and the official project kick-off of CrypTool 2.0 took place at the University of Duisburg-Essen. CrypTool 2.0 (CT2) picked up an approach proposed by previous case studies in which the program should be more intuitive, more visual and more interactive. A modern plug and play interface which follows the Microsoft Office 2007 User Interface Design Guideline was selected as the base of CT2. CT2 should also enable the user to combine cryptographic functions, instead of just being able to use one function at a time as it was in CrypTool 1.

3.2. Architecture of CrypTool 2.0

Regarding this requirements, the underlying concept of the architecture was driven by the idea of creating a highly flexible architecture while maintaining a slim framework. Since CT2 is open source and should be developed by a community, an additional design-aspect was the integration of the possibility for easily

creating and adding new algorithms and cryptographic functions by open source developers. Therefore CT2 has a purely modular design, where almost each component is realized as a plugin. So the enhancement or even a complete exchange of a plugin is a straightforward progress which can be done with no interfering of any other part of the program. A plugin can now be upgraded and made public apart from the main program. To achieve this goal, three different aspects were strictly separated: The front-end, the core, and the plugins. Figure 3.1 with its three horizontal layers represents the three different aspects mentioned above.

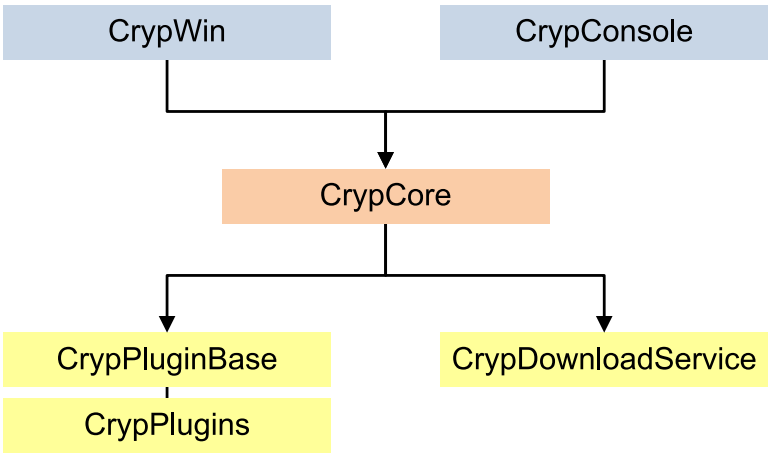


Figure 3.1.: The six main components of the CT2 architecture (in three layers)

The upper blue layer comprises two possible front-ends, i.e. CrypWin and the command-line version of CT2 namely CrypConsole. CrypCore, the heart of CT2, is situated in the middle and all plugin-related components in the lower yellow layer.

We now take a closer look at the six components, beginning with the two application interfaces in the blue layer. **CrypConsole** is a command line interface for automation which is not under development yet. **CrypWin** is a graphical user interface based on the Windows Presentation Foundation (WPF [11]). The GUI of CrypWin consists of two components: the main window with its primary interface and control elements for the application itself. Within the main window a visual programming control (editor plugin) provides visualization and workflow controls to enable intuitive manipulation and interaction of cryptographic primitives supported in CT2. These primitives are again plugins which are hold on a workspace applied by the editor plugin, where they can be added via drag and drop. Figure 3.2 visualizes this plugin concept.

The **CrypCore** component is the central point of administration for the plugins that implement functionality of CT2. To build a global plugin store, which is only amendable by system administrators, as well as a user specific plugin repository, a granular design and approach has been selected for this critical system component. This means that common functionality can be provided as a base to all users, which can be easily expanded by a specific user. By downloading and installing additional plugins, the user creates his local plugin store. The content of this store can only be accessed by the user who has actually downloaded the plugins.

The first component in the yellow layer of Figure 3.1, namely **CrypPluginBase**, is the connection between the plugin developer and core functionality of CT2. CrypPluginBase contains a collection of interfaces and attributes that must be implemented by the plugin developer when writing a CT2 plugin, which itself is

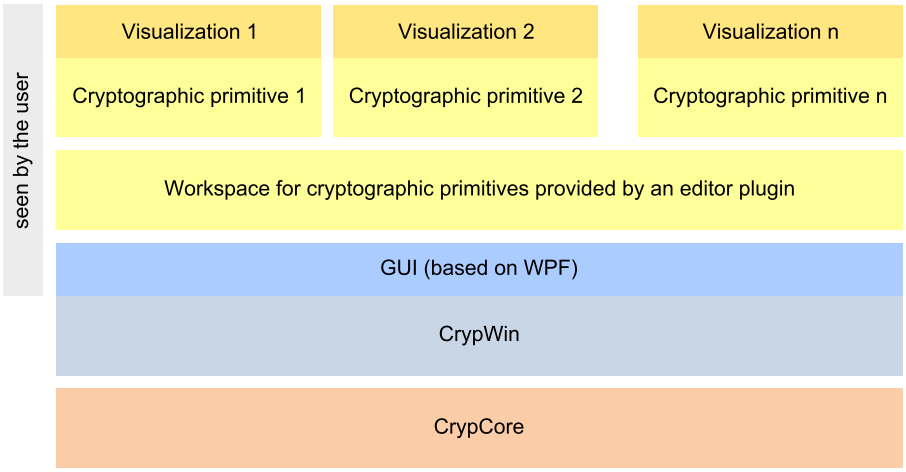


Figure 3.2.: GUI layers of CT2

part of **CrypPlugins**. Due to this design concept, a plugin developer can focus on realizing an algorithm by implementing the interfaces and does not have to care about the core.

Another component of the yellow layer is **CrypDownloadService** which will be a web service providing trusted plugins for download. This service is not implemented yet.

3.3. Graphical User Interface of CT2

At next, we give an overview of the graphical user interface (GUI) of CT2 depicted in Figure 3.3. We define the different elements of the user interface (UI), which are numbered and colored.

A - The ribbon tab

The ribbon tab at the top of the UI contains basic controls for CT2.

- (1) The control buttons can be used to build a new workspace and discard (or save) the old one, open a saved workspace, or save the current workspace.
- (2) These are the tabs of the ribbon tab, namely Home, Settings, and Algorithms. In the Settings tab one can change things like the detail level. The Algorithms tab is more or less a copy of the navigation pane.
- (3) If one wants to start or stop the chain of the plugins on the workspace, one can use these controls.

B - The algorithms pane

The algorithms pane contains all plugins which can be used in CT2. They are sorted into different categories.

- (4) The classic ciphers category contains algorithms like Caesar or Vigenère.
- (5) Other ciphers and hash functions, the cryptanalysis category, tools like TextInput, and editor specific plugins can be found here.

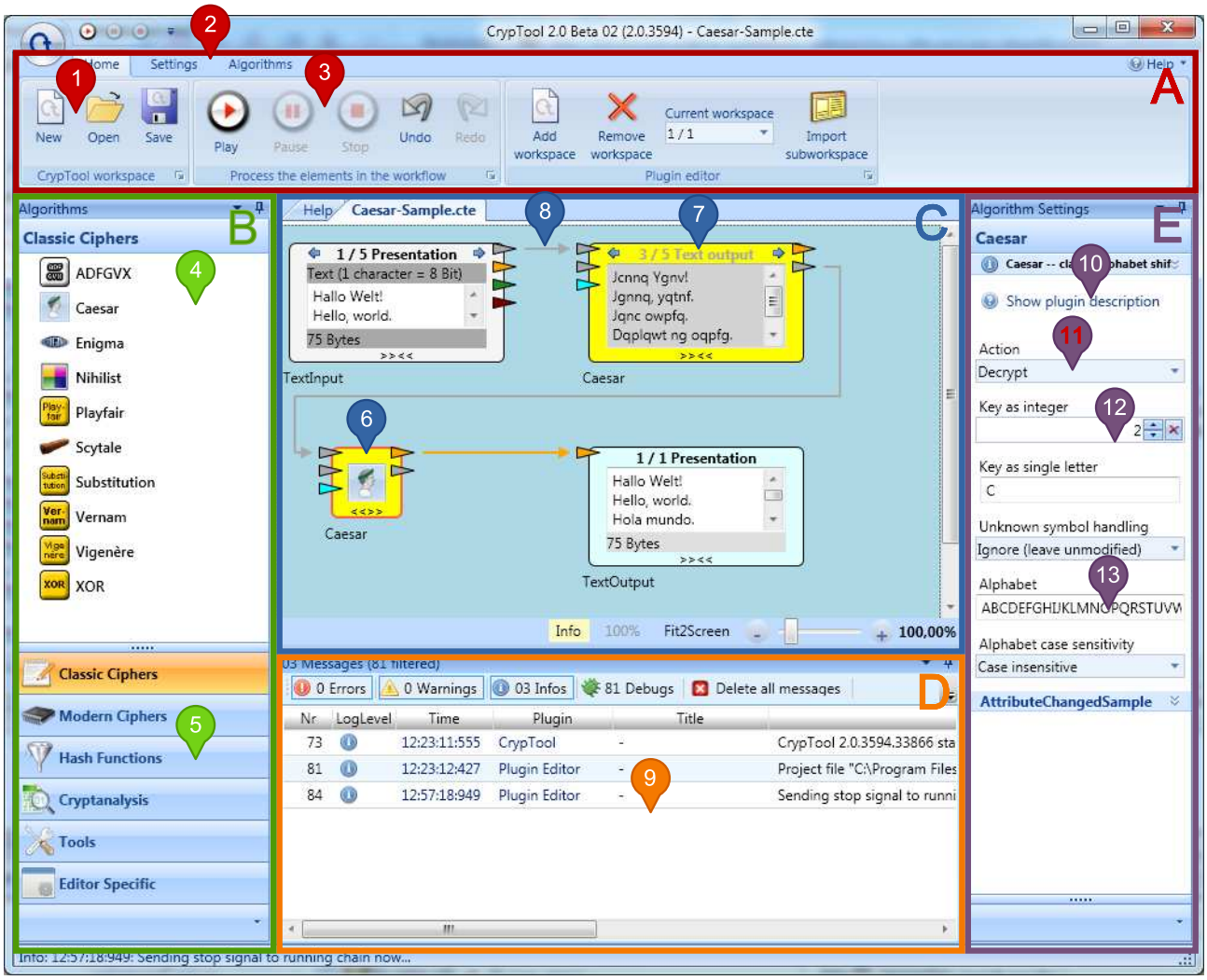


Figure 3.3.: CT2 GUI overview

C - The workspace

The workspace holds the plugins that are used at the moment. Here you can define connections between plugins and see the QuickwatchPresentation of the plugins. The QuickwatchPresentation contains by default the values of all in- and outputs and can be extended by a special presentation implemented by the plugin developer.

- (6) This is a plugin (here Caesar) with the so-called QuickwatchPresentation closed. You can open it by clicking on the arrows («») on the bottom of the plugin. If the QuickwatchPresentation is closed, the same icon as in the algorithms pane is shown.
- (7) This is a plugin (again Caesar) with quickwatch opened. You can cycle through the different in- and outputs and the QuickwatchPresentation created by the developer by using the blue arrows on top of the plugin.
- (8) The in- and outputs of a plugin are marked with triangles at the side of the plugin. On the left

side are the inputs and on the right side the outputs. The color of the triangles refer to the type of the in- or outputs as defined in the source code of the plugin. The connections between to plugins are denoted by arrows in the corresponding colour. You can only connect an output with an input of the same type (and color) except an input of type object, which accepts all output connections. To build a connection click on an output, hold the left mouse button, drag it onto an input and drop it there. In addition to the normal in- and outputs some plugins have an IControl-Master or -Slave connector, which can be found on top of a plugin (black marker). By using the IControl interface, one plugin can directly access a method inside another plugin to gain better performance. This interface can also be used to call a method from a second plugin while still executing the first plugin.

D - The log messages

The log messages of CrypTool are displayed at the bottom of the window. Here you can choose between one or more types of log messages, which can be Error, Warning, Info, and Debug messages. You can also export the messages as a HTML document or delete them.

- (9) The log messages chosen above are listed here.

E - The algorithm settings

In this pane you can configure the plugin you have selected on the workspace (here Caesar). The numbers here point to different types of elements which are available in the settings.

- (10) By clicking on the info link you get a detailed description of the plugin (if implemented by the author). The info panel above the link shows the author of the plugin, his affiliation and e-mail-address (if the author wants to offer the information), and some detailed information about the plugin file.
- (11) This is a dropdown menu implemented as `ControlType.ComboBox` in C#.
- (12) A number field implemented as a `ControlType.NumericUpDown` in C#
- (13) This is a simple text field (`ControlType.TextBox` in C#).

Chapter 4.1 shows how to implement these control types.

3.4. General Conventions Concerning CT2 Plugins

If we ever talk about inputs and outputs of a plugin later on in the implementation chapter, in most cases their names should tell if they are an in- or an output or an IControl interface. Additionally they are marked with an icon: anInput, anOutput, anIControl. Settings elements are also marked with an icon when listed: aSetting.

4. The Implementation

This chapter will be about the implementation of the toolbox in CrypTool 2. We first take a look at generally implementing a CrypTool 2 plugin, the prerequisites and the main interfaces provided by the application in Section 4.1. The next Section 4.2 goes deeper into the implementation of the unique plugins and shows how the components were realized in CrypTool 2. We also present a performance analysis of the Boolean Function Parser plugin.

4.1. Realization of Plugins in CrypTool 2.0

As a basic requirement for CT2 we need an SVN client for downloading and submitting the source code, Microsoft Visual Studio 2008 Professional Edition to load and compile the sources, and a .NET Framework of version 3.5 or higher installed to run the application. Further details on setting up the programming environment can be found at `http://cryptool2.vs.uni-due.de` in the developer area.

A CrypTool 2 plugin mainly consists of the following subparts:

- the main plugin functions,
- the plugin settings,
- the plugin presentation, and
- the plugin description.

This is also reflected by the folder structure as shown in Figure 4.1: While the main plugin functions are contained in the file *MyPlugin.cs*, the settings methods are contained in *MyPluginSettings.cs*. The presentation with the images can be found in the *Images* folder, and the description is located at the *DetailedDescription* folder. The *Poperties* folder contain Visual Studio properties files. The *MyPlugin.csproj* file is the Visual Studio project file.

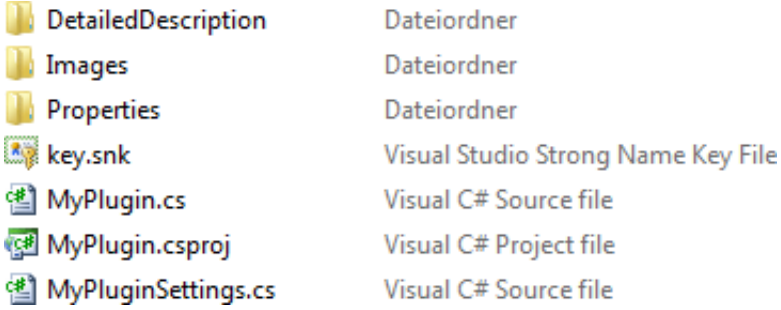


Figure 4.1.: MyPlugin folder structure

The main plugin functions are the basic interfaces which need to be implemented by the user. To get an impression, a basic plugin called *MyPlugin*, which produces a string containing the phrase "Hello world!" as output, is explained step by step in the following. We will present some lines of code and a description by turns.

MyPlugin.cs - Includes, namespace, and plugin information

```
1  using System;
   using System.Collections.Generic;
   using System.Linq;
   using System.Text;
   using Cryptool.PluginBase;
6  using Cryptool.PluginBase.IO;
   using System.ComponentModel;
   using Cryptool.PluginBase.Miscellaneous;

   namespace Cryptool.MyPlugin
11 {
     [Author("MyPlugin Author Name", "author.email@homepage.org", "Authors Affiliation", "http://
         www.homepage.of/affiliation")]
     [PluginInfo(true, "MyPlugin", "MyPlugin short description.", null, "MyPlugin/Images/icon.png
         ")]
```

The first eight lines of code of the plugin *MyPlugin* include essential references, some basic includes of types from the namespace `System` and the main `Cryptool.Pluginbase` interfaces. Line 10 extends the `Cryptool.Plugins` namespace with the name of the plugin `MyPlugin`. Lines 12 and 13 provide some information about the author and the plugin itself which are more or less self explaining. These settings can be seen inside CrypTool by clicking on the info button placed at the beginning of the algorithm settings pane, which is normally placed on the right hand side of CT2.

MyPlugin.cs - Class definition, private variables, and constructor

```
     public class MyPlugin : IOutput
15   {
       #region private variables

       private string output;
       private MyPluginSettings settings = new MyPluginSettings();
20
       #endregion private variables

       // constructor
       public MyPlugin()
25     {
       }
```

Line 14 defines the plugin class and from which class the plugin inherits. In this example we inherit from the interface `IOutput`, which means that our plugin can be described as a box which produces some output. Other examples of interfaces are `IInput`, `IThroughput`, or `ICryptographicHash`. Starting with line 16 we see the region of the private variables, which can only be accessed from inside the plugin. The only private variables in this state of our plugin is the definition of a settings variable and our output variable. We will take a closer look at the code of the settings class later on. Lines 24 to 26 define the constructor of our class which is also fairly empty at the moment.

MyPlugin.cs - Public interfaces

```
    #region public interface

    // output interface
30  [PropertyInfo(Direction.OutputData, "Output", "Output of MyPlugin.", "", true, false,
        DisplayLevel.Beginner, QuickWatchFormat.Text, null)]
    public string Output
    {
      get {
          return output;
35    }
      set {
          output = value;
          OnPropertyChanged("Output");
      }
40  }


    #endregion public interface
```

Starting with line 30 follows one of the main parts which has to be implemented by the plugin developer. In this section we have to define the interfaces of the plugin, through which it communicates with its environment, namely other plugins. The interfaces can have two directions, either coming from the outside (an input) or coming from inside the plugin (an output). The direction of our interface is determined in line 33 inside the property section. Since our interface produces an output, so we set the `Direction` property to `OutputData`. Other choices could also be `InputData`, `ControlMaster`, or `ControlSlave`, but we will get familiar with the `ControlMaster` and `-Slave` interfaces later on in Section 4.2.5. The properties following the direction are:

- the caption "Output",
- the tooltip of the plugin "Output of MyPlugin",
- an URL to a detailed description (normally an xaml file),
- a switch if this input is mandatory or not (if it is mandatory this plugin cannot be used unless this in- or output is connected),
- another switch which defines if this in- or output has a default value,
- the display level at which the input (or output) can be seen,
- and the format in which it will be converted by the quickwatch.

The display level can bet set in the settings of CrypTool. Possible format conversions of the quickwatch are `Base64`, `Hex`, `None`, and `Text`. Lines 34 to 43 implement the getter and setter of our output interface. The get-method returns our private variable `output`, the set-method saves the assign value. Line 38 fires an event which tells the editor, that the value of our output has been changed. The value will now be forwarded by the editor to the plugin connected to our output, for example another plugin which displays the value as a text.

MyPlugin.cs - IPlugin members - Part 1

```
  #region IPlugin Members
45
  #pragma warning disable 67
    public event StatusChangedEventHandler OnPluginStatusChanged;
```

```
    public event GuiLogNotificationEventHandler OnGuiLogNotificationOccured;
    public event PluginProgressChangedEventHandler OnPluginProgressChanged;
#pragma warning restore

public ISettings Settings
{
  get { return settings; }
}

public System.Windows.Controls.UserControl Presentation
{
  get { return null; }
}

public System.Windows.Controls.UserControl QuickWatchPresentation
{
  get { return null; }
}
```

The region called `IPlugin Members` is the most interesting part of the plugin. This region contains the interfaces which run the plugin. The part above starts with a pragma to temporary disable a compiler warning. Since we have not implemented some events yet we put them inside a `pragma warning disable` section. The `pragma` section is followed by the settings interface `ISettings` which returns our settings variable defined in line 19. We also have no implementation of a presentation styled by ourself, so `Presentation` and the `QuickwatchPresentation` both return null.

MyPlugin.cs - IPlugin members - Part 2

```
public void PreExecution()
{
}

public void Execute()
{
  Output = "Hello world!";
}

public void PostExecution()
{
}

public void Pause()
{
}

public void Stop()
{
}

public void Initialize()
{
}

public void Dispose()
{
}

#endregion
```

Part 2 of the `IPlugin Members` region contains the public functions which are called on certain events. These events are fired by the editor and then executed by all plugins present on the workspace. If a plugin is added to the workspace, the `Initialize` function of line 87 is called. By clicking on the play button the `PreExecution` and the `Execute` functions are called (line 66 and 70). As one would have supposed the `Pause` function is executed by clicking on the pause button (line 79). The code of the `Stop` function will be executed when a user clicks on the stop button respectively. Once a plugin gets removed from the workspace, the `Dispose` function will be called. Figure 4.2 shows the `IPlugin` call flow as described.

So the main part of our plugin, where the main actions take place, is the `Execute` function. For example an implementation of a cryptographic algorithm can be placed here. That algorithm may use other (private) functions added by the author. In our plugin we only output a string containing "Hello world!" by setting our output variable (note that we do not set the private variable `output` but call the set function of `Output`).

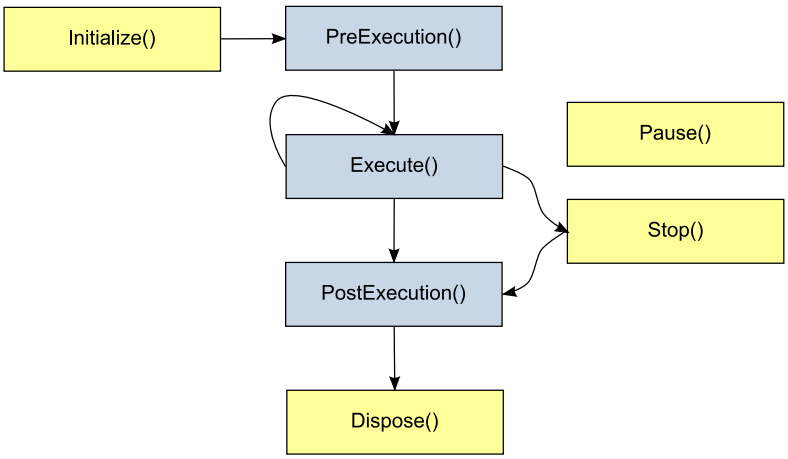


Figure 4.2.: Flow chart of the IPlugin interface.

In our example only the `Execute` function fires an event. Figuratively we tell the editor, that our output value has been changed and an output event gets fire, which will then be forwarded to the plugin connected to our output. The parameter `"Output"` refers to the definition of our output named "Output" in line 34.

MyPlugin.cs - Events

```
    #region INotifyPropertyChanged Members

    public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

100 protected void OnPropertyChanged(string name)
    {
      EventsHelper.PropertyChanged(PropertyChanged, this, new PropertyChangedEventArgs(name));
    }

105 #endregion
  }
}
```

The region specified in the lines 98 to 105 implements the event notification interfaces that tell the editor if something has changed inside our plugin. We define an event handler which will be activated by calling the function `OnPropertyChanged()` as seen before in line 38.

If we now load our project into the CrypTool 2 solution in Visual Studio, compile and run the application, add MyPlugin to the workspace, add a TextOutput plugin, and connect the plugins we will get the following result depicted in Figure 4.3 after pressing the play button.

We will now implement the settings class called *MyPluginSettings* of our plugin which allows us to change the output value of MyPlugin through the settings. In the following, we will present some lines of code and give a description by turns.

MyPluginSettings.cs - Includes, namespace, and private variables

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

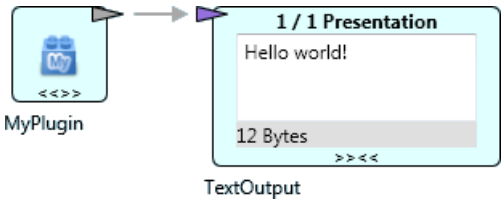


Figure 4.3.: Hello World in CrypTool 2: MyPlugin connected to a TextOutput after pressing Play.

```
5  using System.IO;
   using Cryptool.PluginBase;
   using System.ComponentModel;

   namespace Cryptool.MyPlugin
10 {
       public class MyPluginSettings : ISettings
       {
           #region Private Variables

15         private bool hasChanges = false;
           private string outputMessage = "Hello world! Greetings from the settings.";

           #endregion
```

Again the first lines of our settings file include some essential references. In line 9 we select the same namespace like in MyPlugin.cs. Our settings class inherits from the `ISettings` interface (line 11). Lines 15 and 16 contain our private variables, namely the switch `hasChanges` that is checked by the editor and a string variable containing our settings value.

MyPluginSettings.cs - Public interfaces

```
   #region Public Interface
20
   public bool HasChanges
   {
     get { return hasChanges; }
     set
25   {
       if (value != hasChanges)
       {
         hasChanges = value;
         OnPropertyChanged("HasChanges");
30     }
     }
   }

   [TaskPane("Output message", "Define the output of the plugin. For example 'Hello world!'",
       null, 0, true, DisplayLevel.Beginner, ControlType.TextBox)]
35 public string OutputMessage
   {
     get { return this.outputMessage; }
     set
     {
40     this.outputMessage = value;
       OnPropertyChanged("OutputMessage");
       if (value != outputMessage) HasChanges = true;
     }
   }
45
   #endregion
```

Lines 21 to 32 implement the get and set methods of a switch that tells the editor if a setting of our plugin has been changed. If the get method returns true, the save button in CrypTool 2 becomes available pointing the user to save the changes made by modifying the settings of a plugin. In line 34 we define a textbox in the settings pane of MyPlugin with the parameters caption, tool tip, settings group name, order of the settings item, if the settings item can be changed while the plugin is executed, the display level, and the type. As before we have a get and set method starting in line 37. In line 41 and 42 we propagate to our main MyPlugin class and to the editor if the settings have been changed by the user.

MyPluginSettings.cs - Events

```
    #region INotifyPropertyChanged Members

        public event PropertyChangedEventHandler PropertyChanged;

        protected void OnPropertyChanged(string name)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs(name));
            }
        }

        #endregion
    }
}
```

We implement a second event handler in order to handle our `OnPropertyChanged()` event.

Now we are able to change the output text of MyPlugin through the settings. What the settings pane of our plugin looks like is depicted in Figure 4.4. Figure 4.5 shows the plugin on the workspace with the output text that has been changed by the settings.

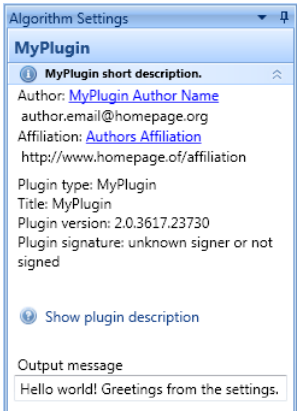


Figure 4.4.: Settings of MyPlugin with the plugin description and the output message setting.

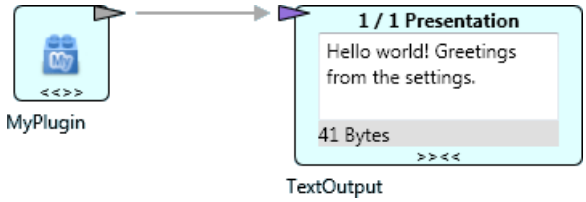


Figure 4.5.: MyPlugin with new output message.

4.2. Components of the Toolbox

In this section we will discuss implementation details of the components of the stream cipher toolbox. Each component is described in a top-down view, which means that we first describe the features, see how it actually looks like in the application, and then get to know how it was realized inside the plugin by presenting relevant functions.

4.2.1. LFSR Plugin

This plugin implements the functionality of a linear feedback shift register. Given a polynomial and a seed, it computes as many output bits as set in the settings of the plugin. Additionally the plugin displays a visual live representation of the LFSR similar to the Figure 2.3 on page 7. Moreover the plugin can be stepped by an external clock as depicted in Figure 2.8 on page 10. To get a specific internal stage the LFSR plugin also provides an additional output bit. The stage is marked orange in the QuickwatchPresentation (see Figure 4.8). If all stages are needed, the plugin is able to provide a boolean array that outputs the complete state of an LFSR. Summarized, the LFSR plugin has the following in- and outputs[1]:

- String *TapSequence* (optional): A string containing the taps as a sequence of the c_i or as a polynomial (e.g.: 1011, which is equal to the string x^4 + x^2 + x + 1. Note, that the last 1, which is x$\hat{0}$, is implicitly set to 1 inside the plugin).
- String *Seed* (optional): The initial binary value of all states (E.g.: 1001).
- Boolean *Clock* (optional): Optional external clock signal.
- String *Output*: The string containing the output stream with a length of the rounds (e.g.: 1001001, assuming that seven rounds have been performed).
- Boolean *Output*: Contains the current output bit as a boolean value.
- Boolean *Additional Output Bit*: Outputs a specific stage of the LFSR as a boolean value.
- Boolean[] *Boolean Array Output*: Outputs all stages of the LFSR. This can be used for example to build a nonlinear filter generator together with the BooleanFunctionParser plugin.

Figures 4.6, 4.7, and 4.8 show the plugin on the workspace in different states.

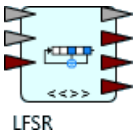


Figure 4.6.: LFSR plugin on the workspace with three inputs and four outputs.

The plugin has a lot of settings to fulfill the requirements of its diverse range of application. Figure 4.9 shows the settings pane in CT2. The pane of the LFSR plugin contains the following items:

- Draw LFSR (Button): Draws the QuickwatchPresentation of the LFSR, if the polynomial fits to the seed.
- Feedback polynomial (Text): Feedback polynomial or tap sequence.

[1] denotes an input, an output, an IControl interface, a settings element.

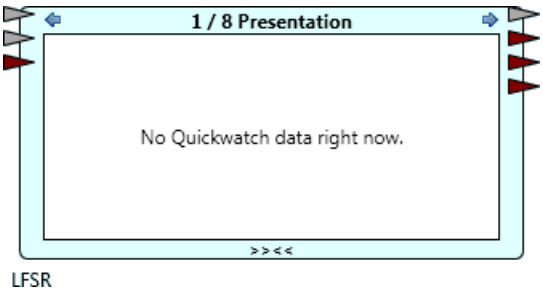


Figure 4.7.: Opened QuickwatchPresentation of LFSR plugin with nothing set.

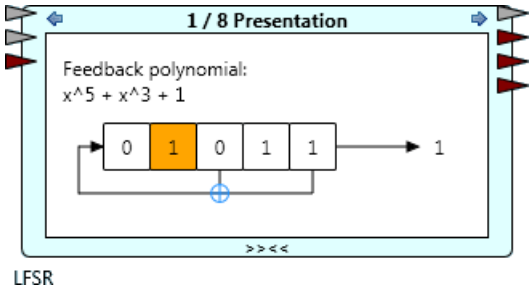


Figure 4.8.: Opened QuickwatchPresentation of LFSR plugin showing the presentation.

- Seed (Text): The seed (initial state).
- Period of LFSR (Label): Indicates the period depending on feedback polynomial and seed.
- Do not display Quickwatch (Checkbox): Does not update the QuickwatchPresentation on runtime when it is checked to get a better performance.
- Number of rounds (NumericUpDown): Number of output bits.
- Save the state of the LFSR (Checkbox): Saves the current state when stopping and restarting the plugin or saving the workspace.
- Output stages (Checkbox): Outputs all stages as a boolean array when checked.
- Generate additional output bit (Checkbox): Outputs a specific stage when checked.
- Additional output bit # (NumericUpDown): Defines the stage of the additional output bit.
- Use external clock (Checkbox): Makes use of the external clock input and then clocks the LFSR only on a true input.
- Always create output (Checkbox): Does output a bit even if the external clock is set to false. The output bit is the bit from the last round. It bit is false if we are in the first round.

We now present a list of the main methods of the LFSR plugin. Each method is briefly described. We omitted (at least some of) the standard IPlugin methods that all plugins contain.

`string BuildPolynomialFromBinary(char[] tapSequence)`
A method that builds a polynomial out of a binary sequence.

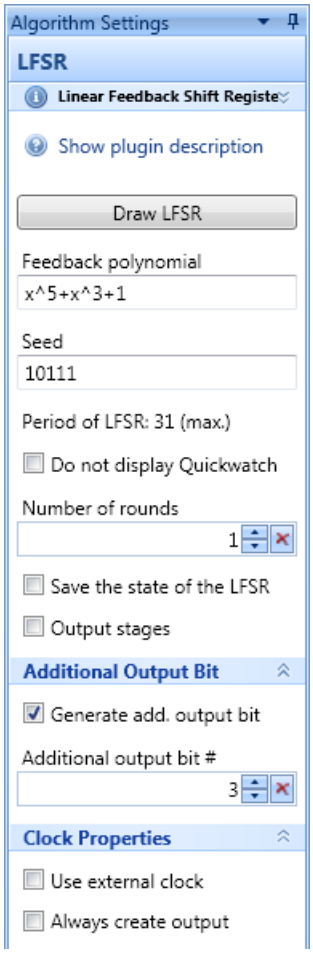


Figure 4.9.: Settings of the LFSR plugin.

Example: The given char array [1,0,0,1] results in the polynomial $x^4 + x + 1$. Again the last $+1$ is set implicitly.

`int checkForInputSeed()`
Loads the seed input or fills it with a dummy value.

`int checkForInputTapSequence()`
Loads the feedback (polynomial) input or fills it with a dummy value.

`int computePeriod()`
Computes the period of the LFSR and displays it in the settings. If the period is maximal, it is shown by adding "(max.)" to the period.

`bool IsPolynomial(String strPoly)`
Checks if a given string is a feedback polynomial using regular expressions.

`String MakeBinary(String strPoly)`
This is an equivalent to the `BuildPolynomialFromBinary` method. `MakeBinary` builds a binary string out of a polynomial.

Example: The given string `x^4 + x + 1` is converted to the string `10011`.

`String MakeBooleanArrayFromCharArray(char[] charArray)`
This method converts a char array into a boolean array.

`void preprocessLFSR(bool createLog)`
The complete setup of the LFSR takes place in this method. We check for valid inputs (tap sequence or polynomial and seed) and do some conversion if necessary. We also check if the length of the tap sequence or polynomial fits to the given seed and throw an error if the seed is too short or too long. We then determine if we get a clock signal from outside the plugin and finally draw the QuickwatchPresentation. This method is also called every time the user modifies the polynomial or seed in the settings tab. The boolean parameter indicates if we forward log messages to the editor or not.

`void processLFSR()`
In this method we update the QuickwatchPresentation and compute the new output bit. If the clock value is false, we output our old value or output false if it is the first round. The additional output bit is picked and set on the output, just as the boolean array output. Both is only done according to the settings of the LFSR plugin.

`char[] ReverseOrder(char[] tapSequence)`
`ReverseOrder()` is a method that inverts the order of a char array.

4.2.2. NLFSR Plugin

The NLFSR plugin is pretty much the same as the LFSR plugin. The only difference is in the `processNLFSR` function which makes use of a boolean function parser method to compute the new bit. The method to

solve a boolean expression is discussed later on in Section 4.2.5 on page 32. The QuickwatchPresentation is also a bit different compared to the LFSR presentation: The feedback function and the used stages are shown instead of the XOR icon (compare to Figure 4.8).

Figure 4.10 shows the plugin.

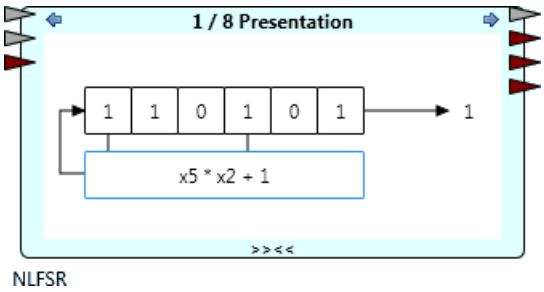


Figure 4.10.: NLFSR plugin with opened QuickwatchPresentation.

4.2.3. CLK Plugin

The CLK plugin acts as a clock for the LFSR and the NLFSR plugins. It fires an event based on a timely offset or based on an incoming event connected to the input. The event fired is a true or false boolean value, which leads to a stepping of a connected (N)LFSR steps one cycle. Figure 4.11 shows two CLK plugins that output a true and a false value each time the output event is fired.

Figure 4.11.: Two CLK plugins on the workspace with a true and a false boolean output.

The following in- and outputs are available at the plugin:

- Object *EventInput* (optional): Optional input used to step the clock one cycle on any input event. Must be set in the settings to have an effect.
- Boolean *Output*: Contains the current output value. Value can be set via settings.
- Int *Output of rounds*: Outputs the current round number as an integer value. This can be used to build a for-loop.

In the settings of the plugin the output value can be selected via the radio buttons (Figure 4.12). The checkbox below tells the plugin if it should fire an output on an input event, or if it should fire after a specific time. The timeout can be set by typing the amount of milliseconds in the textbox underneath the checkbox. The user can also define how many events the plugin should fire. If a FSR is connected, this is directly mapped to the rounds of the FSR. The following list summarizes the available settings:

- Set clock output to (RadioButton): Switches the output value between true and false.

- Use input event instead of clock (Checkbox): Fires a new output, not according to a timer but based on an input event.
- Set CLK timeout in milliseconds (Textbox): Sets the timeout if the input event is not selected.
- Rounds (NumericUpDown): Number of rounds to be stepped.

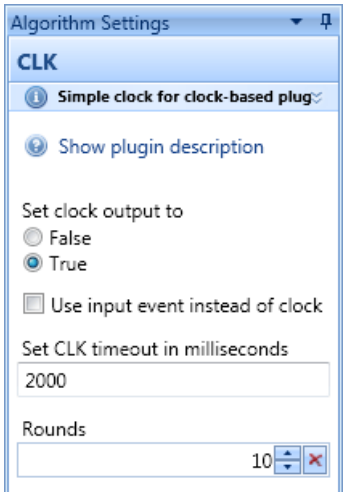


Figure 4.12.: Settings of the CLK plugin.

By opening the QuickwatchPresentation we see a clock which is actually a button. Setting the rounds to zero enables the user to fire an output event by clicking on the button. This can be used for example to step an LFSR by hand, which can be necessary in teaching.

The main methods included in the CLK plugin are explained now (again, we omit some of the standard `IPlugin` methods):

`void Execute()`
Fires an output event if it gets an input event. This only happens if in the settings the option "Use input event instead of clock" is checked. Otherwise it starts a timer by calling the `process()` function. In both cases the current round number is set on the round output.

`void PreExecution()`
In the preexecution the icon is set according to the output settings of the plugin. If the output should have a true value the icon depicts a green clock, otherwise a red clock.

`void process(int timeout)`
The process function starts a timer with the timeout given by the settings. Every time step it calls the function `sendCLKSignal()` until the number of clock cycles has reached its destined value set in the settings.

`void sendCLKSignal(object sender, EventArgs e)`
Produces the output event (true or false, depending on the setting of the plugin) and stops the timer of the process function if the number of clock cycles is reached.

4.2.4. Berlekamp-Massey Algorithm Plugin

The Berlekamp-Massey Algorithm (BMA) plugin runs the BMA on a given binary sequence string. Its output is the length L as an integer value and the corresponding polynomial $C(D)$ as a string. Both values are also presented in the QuickwatchPresentation of the plugin. The output polynomial can be used as an input for an LFSR plugin. The BMA plugin has no settings. Figure 4.13 shows the plugin with closed and opened QuickwatchPresentation.

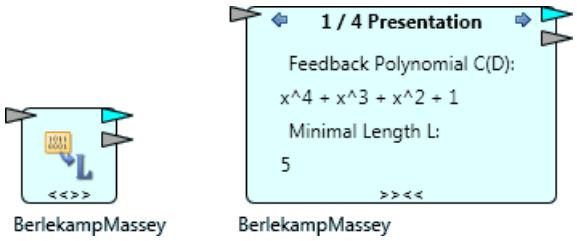


Figure 4.13.: BMA Plugin with the QuickwatchPresentation closed and opened.

The following in- and outputs are available at the plugin:

- String *Input sequence* (mandatory): The binary input sequence as a string.
- Int *Minimal Length L*: Outputs the minimal length L of the LFSR found by the BMA.
- String *Feedback Polynomial C(D)*: Outputs the feedback polynomial $C(D)$ found by the BMA.

The following functions are the main ones of the Berlekamp-Massey Algorithm plugin:

`int BerlekampMasseyAlgorithm(byte[] s)`
Implements the Berlekamp-Massey algorithm itself as described in Algorithm 1 on page 6. Besides the minimal length L this function also returns the corresponding polynomial $C(D)$. Since a function cannot return two values, L is the return value and the output `PolynomialOutput` is set directly to $C(D)$ by the function. To gain $C(D)$ we first have to build a polynomial of it, since the algorithm produces a binary sequence. 100101 for example is the equivalent of the polynomial $1 + D^3 + D^5$ (lowest bit first).

`string BuildPolynomialFromBinary(char[] tapSequence)`
A function that builds a polynomial out of a binary sequence.

Example: The given char array [1,0,0,1] results in the polynomial $x^4 + x + 1$.

`char[] ReverseOrder(char[] tapSequence)`
`ReverseOrder()` is a function that inverts the order of a char array. Since the BMA produces only the binary equivalent of an polynomial but with lowest power first, we have to invert its order before building a polynomial from it.

Example: The array [1,1,0,0,1] is inverted to [1,0,0,1,1].

`byte[] StrToByteArray(string stringToConvert)`

This function converts a string into a byte array. It is used to convert a binary sequence string from the input into a byte array that then can be processed by the BerlekampMasseyAlgorithm function mentioned above.

4.2.5. Boolean Function Parser Plugin

The Boolean Function Parser (BFP) plugin parses a boolean function and computes the single output bit. The function is given through an input string or can be set in the QuickwatchPresentation. It may contain variables `xi.j` and a memory bit `m`. The values for the `xi.j` come from the boolean array inputs or can also be set in the QuickwatchPresentation. The BFP plugin supports functions based on the operations AND (*), XOR (+) and parentheses. Since the BFP can get its values for the function variables from as many other plugins (NLFSR for example) as the user likes, the number of the boolean array inputs are variable. The number of inputs can be set in the settings. The memory bit is the output bit of the last run of the plugin.

The function variables `xi.j` are defined as follows:

- x denotes the beginning of a variable.
- i denotes the number of the boolean array inputs, starting with 0. Such an array consists of a variable numbers of bits.
- j denotes the array index of the boolean array input bits, also starting with by 0.

Example: The given variable `x0.2` denotes the third array index of the first boolean array input.

To refer to the data that is given in the QuickwatchPresentation we use the variable `xqj`, where j again stands for the array index.

Figure 4.14 shows the Boolean Function Parser on the workspace in the normal mode. The function used for the computation is given by the user through the QuickwatchPresentation. It contains the quickwatch variable `xq0` and the memory bit `m`. The data is also given through the quickwatch. The value of the memory bit is shown in the lower part of the presentation and is 0 (false) by default (on first run).

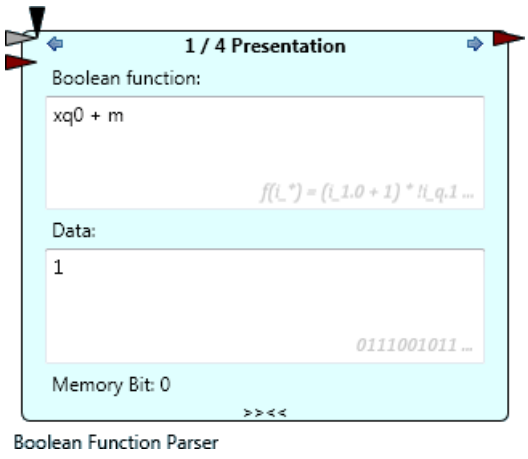


Figure 4.14.: Boolean Function Parser with QuickwatchPresentation opened (normal mode).

In addition to the normal inputs the BFP plugin has an **IControl slave** connector to provide direct access to its parsing function. With this interface it is possible for a plugin with an IControl master connector to let the BFP solve a function at runtime. In other words, a plugin A can use the parsing function of the BFP like it were implemented inside the plugin A itself. This allows a very fast communication compared to the normal way over in- and output connectors.

A plugin that can connect to the IControl slave connector is for instance the Cube Attack. If we change the setting of the BFP to **CubeAttack mode**, we get a slight different QuickwatchPresentation for passing input variables (see Figure 4.15). Here, we can define a function for the cube attack and a secret key whose bits are used as values for the variables of the funtion. A connected cube attack plugin is able to analyze the function and to reveal the secret key. In the CubeAttack mode a new variable `vj` for the public key bits (sent by the cube attack plugin) is available. The variables `xj` in this case refers to the data bits given as secret key bits. Since we have only two data arrays, we do not need a second index for variables as in textttxi.j.

Example: The given function `v0v1x0 + v0v1x2 + v2x0x1 + v0x0 + v0x1 + v1x2 + v2x1 + x0x2 + v0 + v1 + x0 + x1 + 1` together with the secret key bits `101` is determined by the cube attack during preprocessing phase. The key bits are correctly revealed during online phase. Note that the AND operator (*) inside the function can be omitted by the user.

To get more information about the cube attack and especially the plugin see [12].

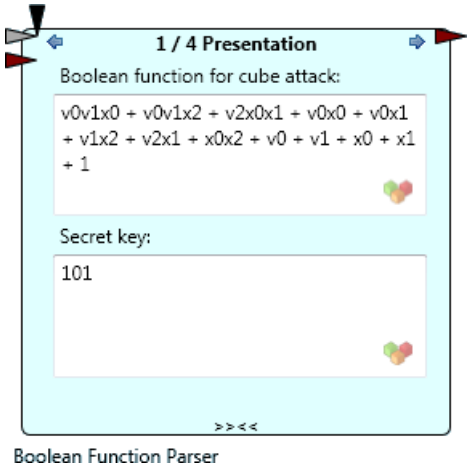


Figure 4.15.: Boolean Function Parser with QuickwatchPresentation opened (CubeAttack mode).

The following in- and outputs are available at the plugin:

- String *Boolean function* (optional): The boolean function to be solved can be given as a string.
- Boolean[] *Input i* (optional): The data bits to be placed in the function. This is a dynamic input and the number of these inputs can be set via settings.
- Boolean *Function Output*: This output contains the computed output bit of the function.
- IControl Slave *BFP Slave*: This is the IControl slave connector.

The settings of the plugin are rather short and contain only two values:

- Number of inputs (NumericUpDown): Sets the number of boolean array inputs.
- CubeAttack mode (Checkbox): Sets the plugin into the CubeAttack mode.

The main methods of the Boolean Function Parser plugin are as follows:

`void AddInput(string name, string tooltip)`
This method adds an input to the plugin.

`void CreateInputOutput(bool announcePropertyChange)`
The `CreateInputOutput` method creates the new number of inputs and announces the change to the editor if the parameter is set.

`bool EvaluateString(string function)`
This method solves a given function without any variables or with variables replaced by 0 or 1 and returns the one bit value as boolean. This method is discussed in detail later on.

Example: The given string "1+0*(1+1)", where + stands for the boolean operation XOR and * for AND, results in 1 and hence, the boolean value "true" is returned.

`void Execute()`
The `Execute` method checks if all boolean array inputs are fresh or if data is given in the QuickwatchPresentation and then parses the boolean function.

`object getCurrentValue(string name)`
Due to the variable count of the boolean array inputs this method returns the current value of an input by a given name.

`string makeStarsInText(Match m)`
This method adds missing * in the function for a given regular expression match. These operators are needed by the evaluation method.

Example: The given string "x1.0x0.0(x1.1+x1.2)" results in the string "x1.0*x0.0*(x1.1+x1.2)".

`int ParseBooleanFunction(bool[] inputVariables, bool[] dataTwo)`
This method is for the IControl interface. It reads the variables and the function, calls the replacing method, tests the function, and finally calls the evaluation method. It returns the computed result as integer.

`string ReplaceVariables(string strExpressionWithVariables, bool[] externDataOne, bool[] externDataTwo)`
`ReplaceVariables` is a method for replacing the variables `xi.j`, `xqj`, `xj`, and `vj` with the values given in one of the boolean arrays.

`string TestFunction(string strExpression)`
This method tests a function if it is a valid boolean function.

Boolean Function Parser Performance

The above mentioned `EvaluateString()` method mainly consists of StringBuilder-operations like `IndexOf`, `Substring`, `Remove`, and `Insert`. These operations are used to find parentheses and binary operators in the given string. If we find a "*" symbol for example, we try to get the digit before and after the symbol, compute the sum of them and then replace the three symbols by the computed sum. If we find an opening parenthesis, we look for the corresponding closing parenthesis and call the evaluation method recursively with the string inside the parentheses.

Working on strings in general is normally a relative costly method, whether we use the String class or the StringBuilder class. One may assume that we implemented a method that uses the reverse polish notation (RPN) which is very fast in evaluating expressions. We have tested our implementation against two freely available function parsers, **RPNExpression**[2] and **MathParser**[3]. The first one uses the RPN to evaluate an expression, the second one makes use of regular expressions. For testing the two approaches against our implementation we used a given scenario in which a cube attack is performed against a boolean function using our BFP. During a cube attack the BFP plugin is called a lot of times ($\sim$ 1300 times). We now measured the time used in all cases to set up the parser (introduce variables and corresponding values) and finally parse the function. We did that for every time the BFP method `ParseBooleanFunction()` is called, picked the maximum time used, and counted also the sum of all calls. Afterwards we measured the scenario again but inserting a for-loop inside the `ParseBooleanFunction()` method, so that the parser setup is done like before but now we solved the expression 65536 times, which is 2^{16}. Table 4.1 shows the results. **MyParser** was given as a name to our own implementation.

Parser	Max. time [ms]	Overall time [ms]	Max. time loop [ms]	Overall time loop [ms]
MyParser	27	8.80	2294	∞
MathParser	1	1.00	12	2949
RPNExpression	2	8.05	313	∞

Table 4.1.: Maximum and overall time of the different parsers measured in our first setup.

As one can see, the fastest parser in all cases is MathParser. Neither our own implementation nor the RPNExpression parser did finish the looped requests in a reasonable time. But we have only checked the maximum time used per one call and did not compute an average value for all calls. The overall time is not significant, because the number of calls from the cube attack differ from run to run and we did not count the calls in our first setup.

So we adjusted our scenario in a second setup and tested again our own implementation against both parsers in 5 different runs without any loop to get a mean value of a request. We set the following boolean function for the cube attack and performed the preprocessing phase: $v0v1x0 + v0v1x2 + v2x0x1 + v0x0 + v0x1 + v1x2 + v2x1 + x0x2 + v0 + v1 + x0 + x1 + 1$. The results are shown in Table 4.2 and are depicted in Figure 4.16.

The results show clearly that our naive implementation is nearly two times faster than the MathParser. We had the higher maximum time per one call in the test before (which may happened due to other rival processes in windows during runtime), but in average we solved an expression faster than both of the other parsers. The RPNExpression parser is two times slower than the MathParser.

[2] http://www.codeproject.com/KB/recipes/RPNExpression.aspx
[3] http://www.codeproject.com/KB/cs/MathParser.aspx

Parser	run 1 [ms]	run 2 [ms]	run 3 [ms]	run 4 [ms]	run 5 [ms]	⌀ [ms]
MyParser	0.352	0.377	0.346	0.342	0.355	0.3544
MathParser	0.809	0.719	0.761	0.738	0.786	0.7626
RPNExpression	1.764	1.670	1.579	1.608	1.610	1.6298

Table 4.2.: Average time used in about 1300 requests and average of all runs from the second setup.

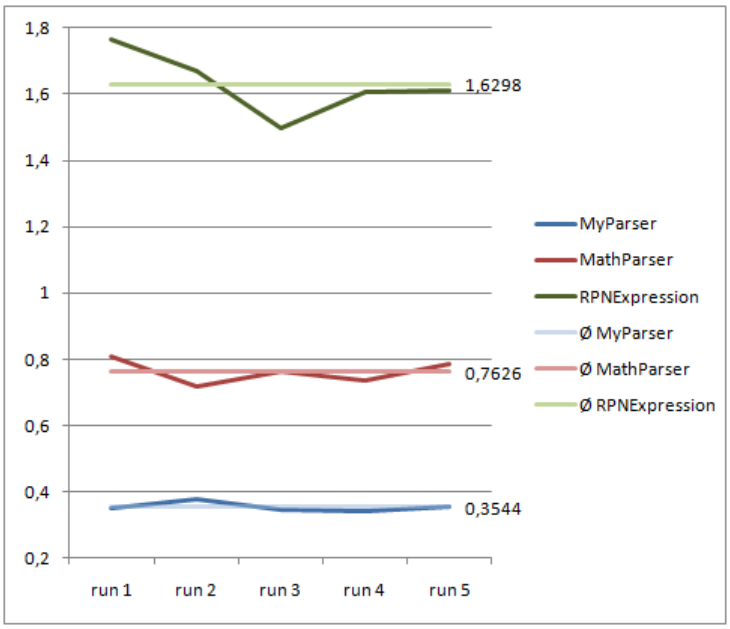


Figure 4.16.: Running time of our own parser versus the other parsers visualized.

The advantage of our implementation could be, that it only implements the needed functionality, namely the boolean operations AND and XOR and the ability of using parentheses. In contrast to this the other parsers support a lot more operations. But the results presented were made with a MathParser stripped-down to the minimal functionality our parser supports. So, in this setup, our own implementation using string operations seems to be the fastest way to solve an expression. We had no chance to strip-down the RPNExpression parser in an easy way, so we used the original class. All times mentioned are the averages from around 1300 requests sent by the cube attack during preprocessing phase.

After pointing out, that the RPN and regular expression approaches were slower, we could now further concentrate on which string operation we should use in our implementation - String or StringBuilder. An analysis for the replace method has been presented by Sam Allen on dotnetperls.com[4]. His study clearly states StringBuilder as the winner of his competition. Due to this analysis we used the StringBuilder class for our parser.

The results presented may vary on a different machine. For the sake of completeness here is a rough summary of our test setup:

- Test machine was a Lenovo T61, Intel Core2 Duo T7500 @2.2GHz

[4] http://dotnetperls.com/replace

- Operating system was Windows 7 Professional 32bit with 2GB RAM
- CrypTool was started out of Visual Studio 2008 Professional Edition in Debug mode (mixed platforms)
- A sample workspace was used with a Cube Attack plugin and a Boolean Function Parser plugin connected via IControl interface
- Settings of Cube Attack were:
  - Action: Preprocessing
  - Public Bit size, Secret Bit Size, Max Cube Size: 3
  - Constant Test, Linearity Test: 20
  - Output Bit: 1
  - Manual Public Bit Input: 0*0
  - Read superpolys from File: unchecked
  - Enable Log Messages: unchecked
- Settings of BFP were:
  - Cube Attack mode: checked
  - Boolean function for Cube Attack: v0v1x0 + v0v1x2 + v2x0x1 + v0x0 + v0x1 + v1x2 + v2x1 + x0x2 + v0 + v1 + x0 + x1 + 1
  - Secret key: 101

4.2.6. Auxiliary Plugins

Some auxiliary plugins have been evolved during the implementation of the plugins presented before. To build an alternating step generator as described in Section 2.8 on page 10 we needed some boolean operators, namely a binary AND and a NOT operator. Boolean operators were not implemented as plugins at the implementation phase of our work. Our code was merged into and enhanced by the Boolean Binary Operator plugin developed by Nils Kopal[5].

We also needed a plugin that appends the boolean outputs of a LFSR plugin. To get a binary sequence instead of single values we implemented the Appender plugin. The Appender takes boolean values or strings, and then appends and outputs them as a string. This can be used as an input for the Berlekamp-Massey Algorithm plugin for example. Figure 4.17 depicts the Appender plugin.

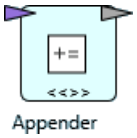


Figure 4.17.: Appender plugin.

[5] nils.kopal@cryptool.org

4.2.7. Class Diagrams of all Plugins

We now present a non commented list of the class diagrams of all plugins described above (generated by Visual Studio).

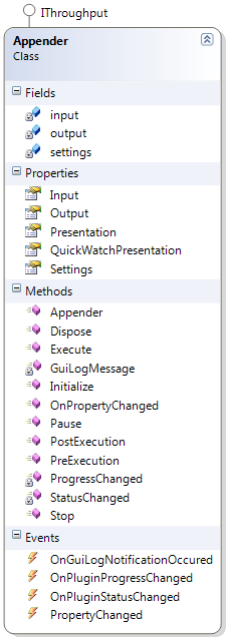


Figure 4.18.: Appender class diagram

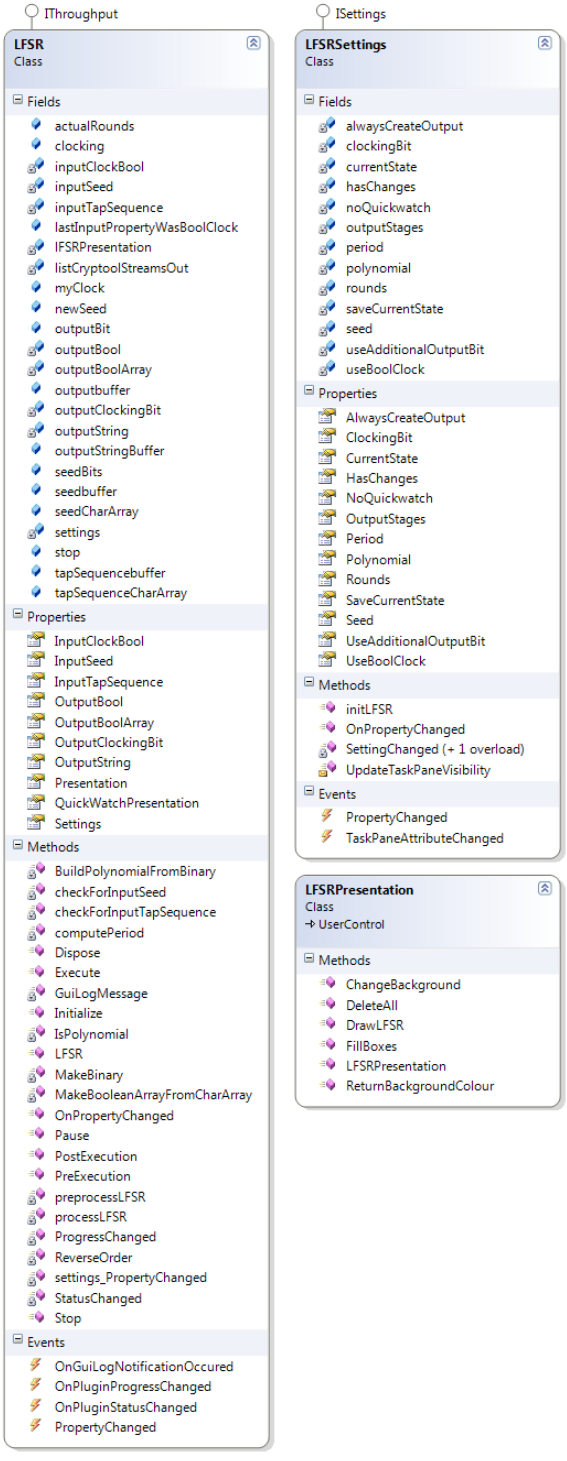


Figure 4.19.: LFSR class diagram

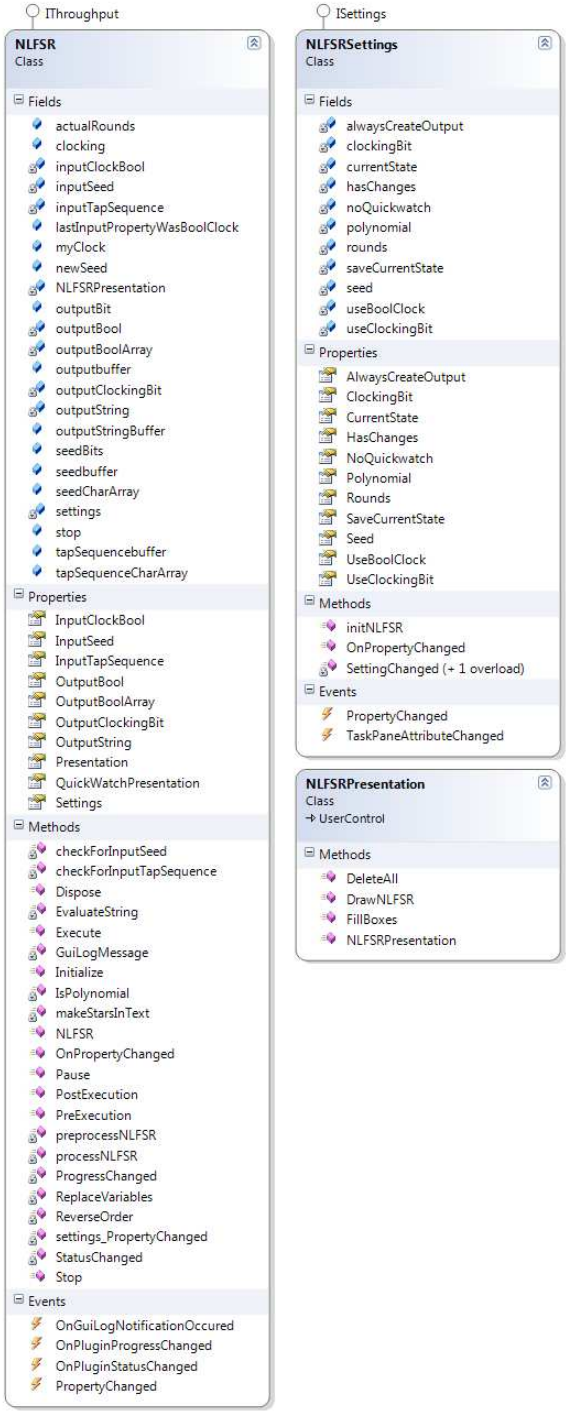


Figure 4.20.: NLFSR class diagram

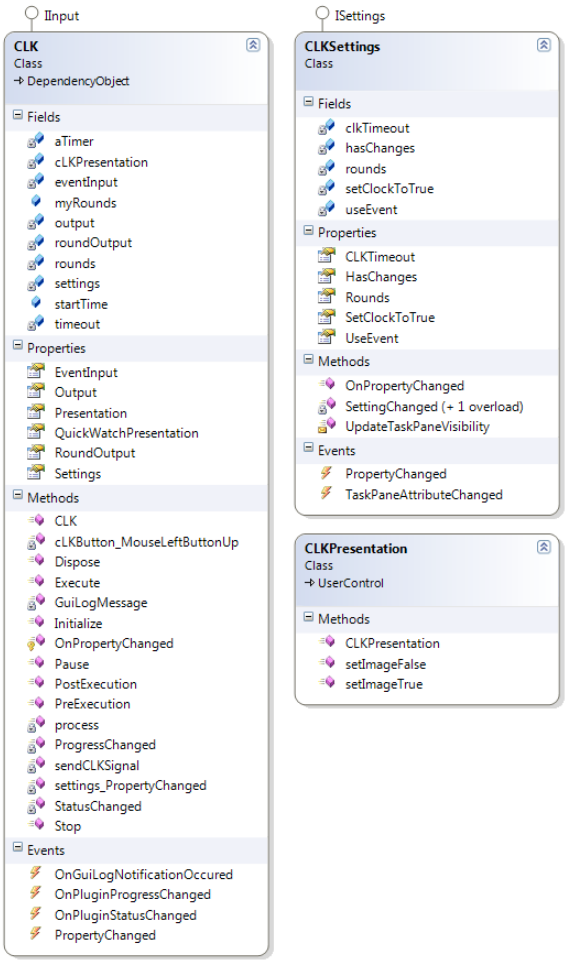


Figure 4.21.: CLK class diagram

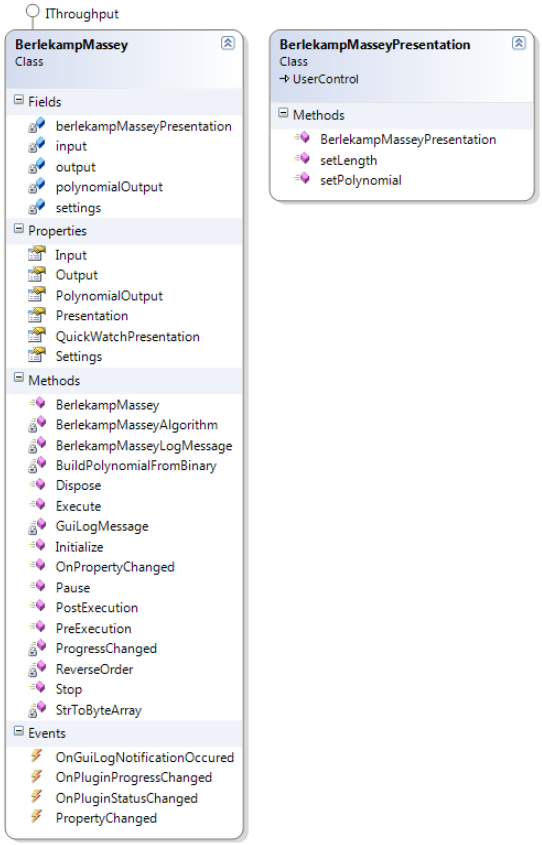


Figure 4.22.: BMA class diagram

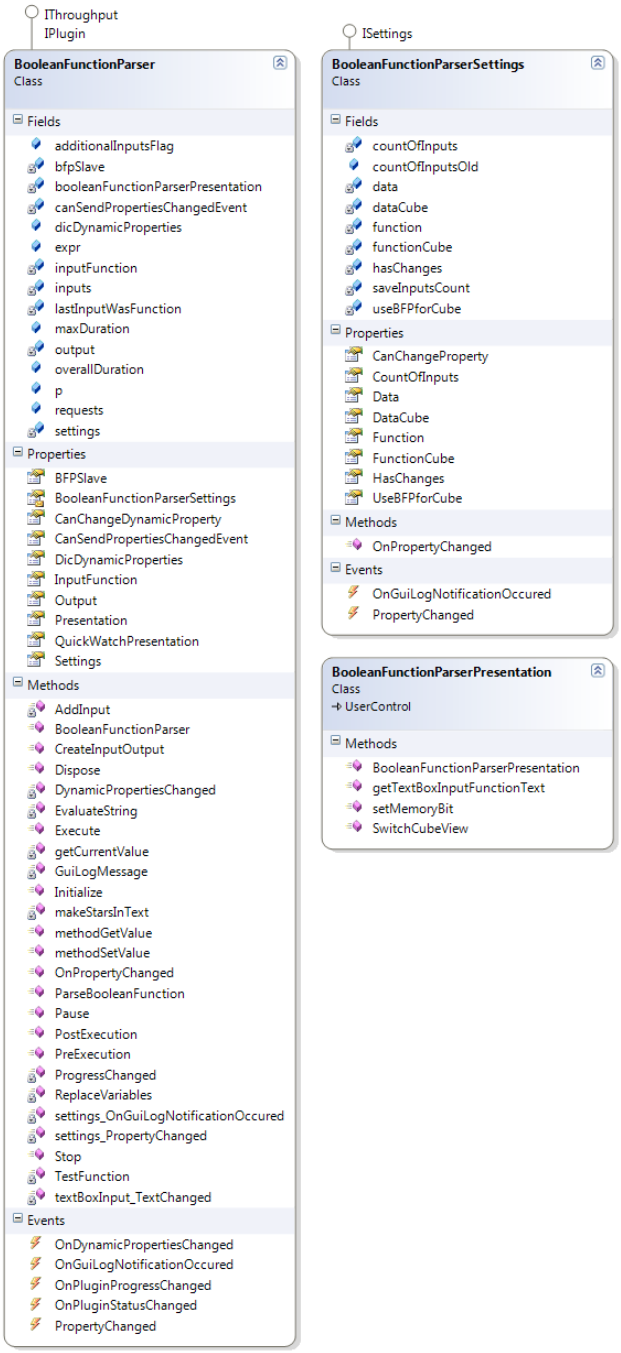


Figure 4.23.: BFP class diagram

5. Practical Scenarios Based on the Toolbox

In this chapter we will explain how the plugins presented before can actually be used inside CT2. Section 5.1 starts with an LFSR plugin scenario. Step by step we extend our scenarios by using more of our implemented plugins. We then show in Section 5.2 how to set up keystream generators in CT2 as described in Section 2.8. Finally we assemble the Achterbahn-80 cipher in Section 5.3 as described before in Section 2.5 on page 8.

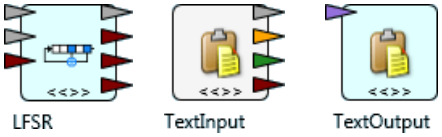
This chapter assumes that you are familiar with the GUI elements of CT2 as described in Section 3.3 on page 13.

5.1. Basic Usage of the LFSR Plugin

5.1.1. Using the plain LFSR plugin

Summary
In this basic scenario we want to setup a single LFSR with a polynomial and a seed and generate 20 output bits.

Plugins used in this scenario



Scenario walkthrough
At first, we have to add an LFSR plugin by double clicking or dragging the LFSR icon [] onto the workspace. The icon can be found in the algorithms pane under *Tools* → *Throughput*. We now open the QuickwatchPresentation by clicking on the arrows (««) at the bottom of the LFSR plugin. Since we did not enter any polynomial or seed yet, we only see the message "No Quickwatch data right now.".

Hence, we define a feedback polynomial in the settings pane of the LFSR plugin by entering the following function into the textbox: x^5+x^2+1. While entering, a red dot appears on the top left corner of the QuickwatchPresentation. That indicates, that our polynomial does not fit to the seed. Since we did not enter any seed at all, we can ignore that. As seed we enter "11100". By entering the last digit, our seed fits to the entered polynomial and in the QuickwatchPresentation a graphical representation of an LFSR appears. In addition to this, the user also gets a notification if the chosen polynomial is a good polynomial, which is shown by a label under the seed confirming that our LFSR has a maximum period. Since we want to generate 20 output bits we have to adjust the number of rounds in the settings. Click on the upper arrow until 20 rounds are set.

In order to get the output of our LFSR we have to add another plugin called TextOutput [], which can be found in *Tools* → *Output* inside the algorithms pane. Drag the icon just to the right hand side of the LFSR plugin. Now open the QuickwatchPresentation of the TextOutput again by clicking on the arrows. We now have to connect both plugins. To do so, click and hold the left mouse button on the first output of the LFSR plugin (grey marker) and drag it onto the input marker (purple) of the TextOutput plugin. When you see that the input marker starts to glow green release the mouse button. You now have successfully connected two plugins.

Let us now concentrate on generating an output. To start our workspace we have to click on the Play button located on the upper ribbon tab, but unfortunately the button is not available. This results from the fact, that our LFSR plugin is not able to start by itself. It has to be started by another plugin. THerefore we add the TextInput plugin [], which can be found under *Tools* → *Input* and connect its first output (grey marker) with the first input (grey marker) of our LFSR as described above. By adding a startable plugin like the TextInput the Play button above the workspace becomes available. We have connected the TextInput plugin with the first input of our LFSR plugin, which is an additional way to tell the LFSR which polynomial has to be used. Since we did not enter any function inside the TextInput plugin, it will be ignored and the polynomial set in the settings pane will be used.

Now we are done and after clicking on the *Play* button our LFSR generates 20 bits of output, which are displayed inside the TextOutput plugin. Figure 5.1 shows CT2 after following the steps described above.

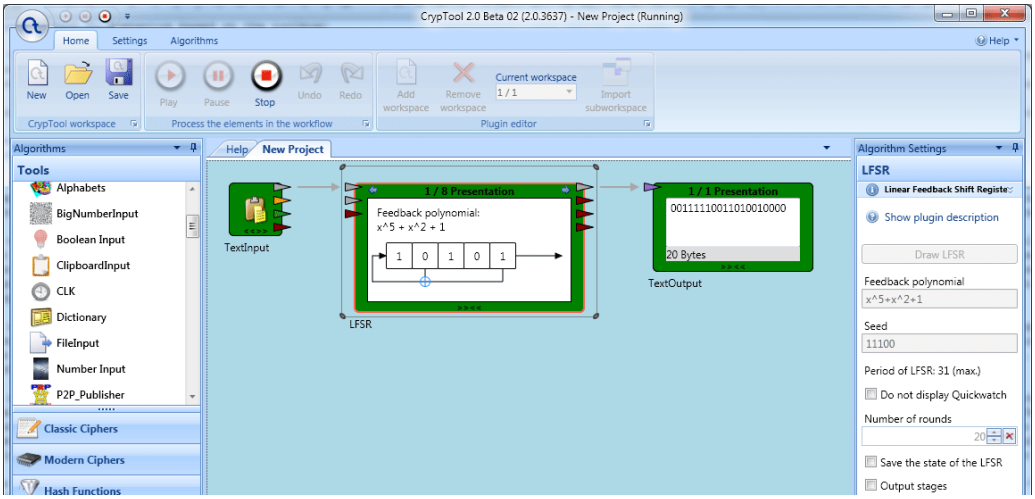


Figure 5.1.: LFSR plugin on workspace generating 20 output bits.

Note that the background of the plugins is now green, which indicates that each plugin did run successfully. In the further course, to change anything on your workspace, we first have to press the Stop button.

The workspace can be saved by clicking on the *Save* button located at the upper Home ribbon tab. Name it for example "ToolboxSample_1.cte".

Lessons learned
You are now able to connect plugins, set the settings of thr LFSR plugin and save a workspace.

5.1.2. Using an External Clock Signal to Step an LFSR

Summary

In this scenario we learn how to use an external clock to step an LFSR. In the following, we first use the CLK plugin to clock the LFSR, and then show how to clock another LFSR by the first one. This scenario is based on the previous.

Plugins used in this scenario

Scenario walkthrough

We assume, that you have opened CT2 with the workspace we have built in the previous scenario. We now add the CLK plugin [⏲], which is located in the algorithms pane under *Tools* → *Input*. Connect the first output (red marker) with the third input marker (red) of the existing LFSR plugin. The CLK plugin can now send a boolean value to the LFSR to control its stepping. Now select the LFSR plugin and open the Clock Properties section in the settings pane in order to check the "Use external clock" checkbox. Note that the ability to set the rounds in the settings of the LFSR plugin is gone. The LFSR now only steps on an external clock event. As we do not need the TextInput plugin anymore, select it and press delete on your keyboard to remove it from the workspace. Your workspace should now look similar to Figure 5.2.

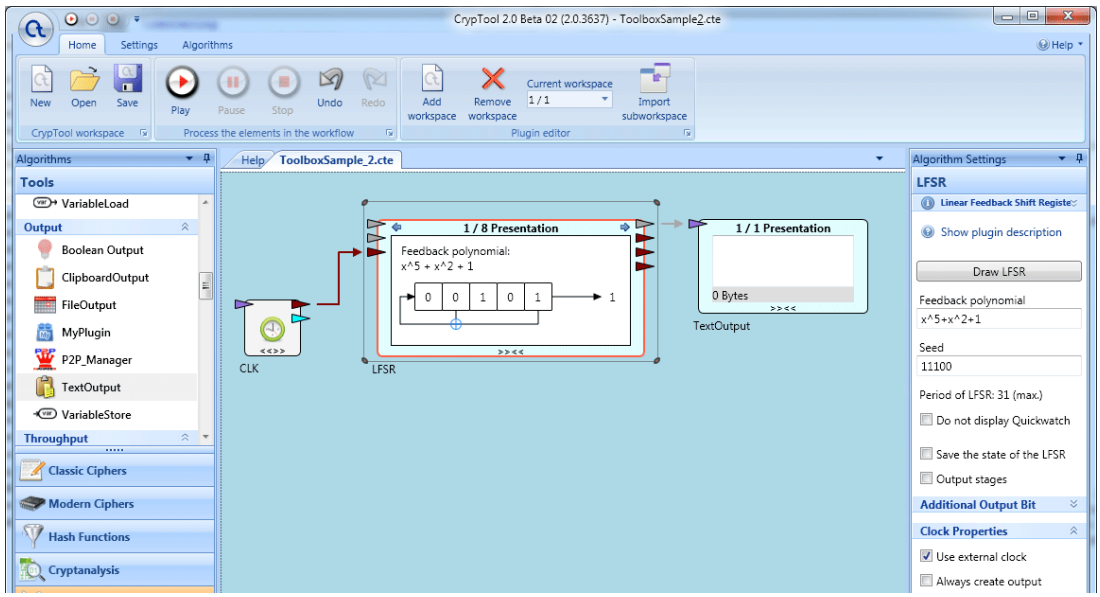


Figure 5.2.: CLK plugin connected to the LFSR.

By pressing the *Play* button, the CLK fires an output event every two seconds, which causes the LFSR to produce one output bit. Here, one can see how the bits are flipping in the QuickwatchPresentation (which was too fast to recognize in the first scenario in Section 5.1.1). After 10 Bits have been generated

the whole process ends.

The time between each new clock event and the number of clock events can be set in the settings of the CLK plugin. After pressing the *Stop* button in the upper ribbon tab, select the CLK plugin and have a look at the settings of the plugin. We change the timeout in milliseconds to 1500 and the number of rounds to 15.

We now add a second LFSR plugin from *Tools* → *Throughput* to the workspace, open the Quickwatch-Presentation and set the string "10010" as the feedback polynomial in the settings. Fill the seed with enough 0s and 1s, so that the QuickwatchPresentation displays the LFSR representation and the red dot disappears from the top left corner of the presentation. As you may notice, the two LFSRs look the same (aside from the seed). In this way it is possible to use a binary tap sequence instead of a polynomial to construct an LFSR.

Since we want our second LFSR to be clocked through the first LFSR, we check in the settings of the second LFSR in the Clock Properties section the "Use external clock" checkbox. As clock input we connect the second output of our first LFSR (red marker) with the third input of the second LFSR. However, in order to not only see the output of the first LFSR, we have to add another TextOutput plugin from the *Tools* and connect it with the first output (grey marker) of the second LFSR.

If you open the QuickwatchPresentation of the second TextOutput now and press *Play*, your workspace should look like Figure 5.3. You may save it as "ToolboxSample_2.cte".

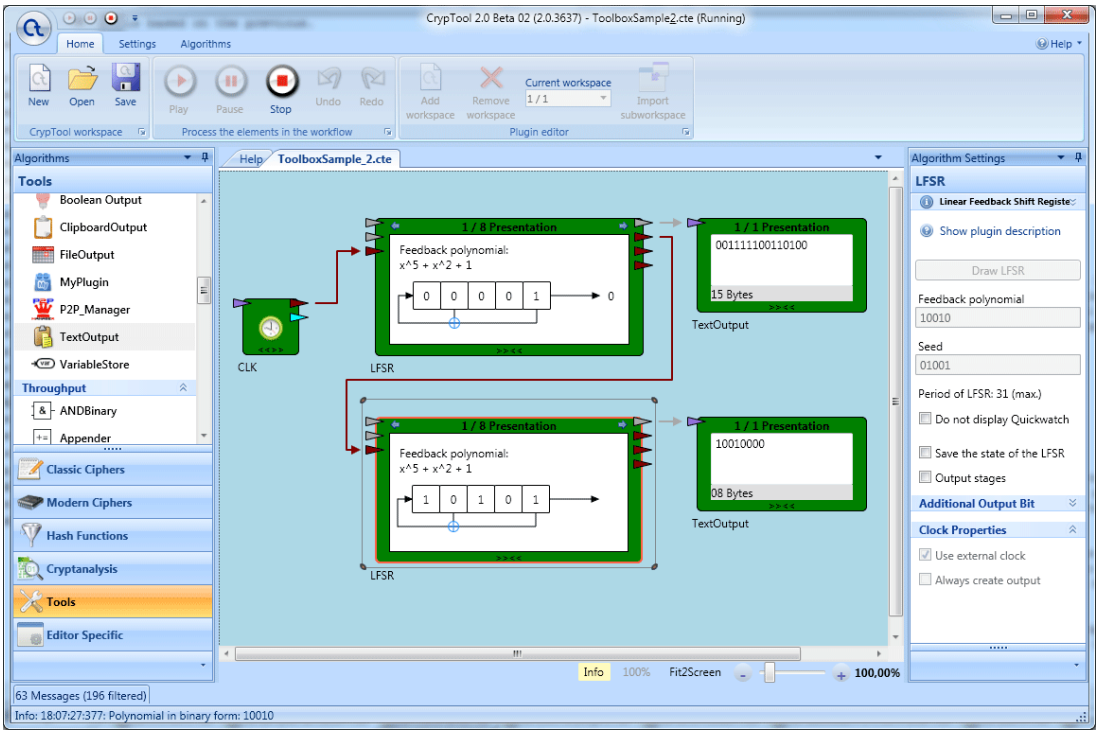


Figure 5.3.: CLK plugin connected to an LFSR, which itself clocks another LFSR.

As the result our first LFSR outputs 15 bits and the second LFSR only 8. This due to the fact that the number of 1s in the output of the first LFSR is exactly 8. Our LFSRs only step one cycle, if the clock event is true (=1). And hence the second LFSR is stepped only 8 times. Now go to the settings of the

second LFSR and check the box "Always create output" in the Clock Properties section (Do not forget to press *Stop* before changing something). This will force the LFSR to output a bit even on a false clock event. The bit which is returned as output is the bit from the last round (or zero, if there is no last round to repeat).

To speed up the plugin chain a bit, we use a feature of the CLK plugin that can be selected in the settings: "Use input event instead of clock". After checking it in the settings, the clock fires a new output event only after getting an input event. This allows us to fire a new clock event when the last member of our chain has finished processing. To tell the clock which one is the last in the chain, connect the first (or second) output marker of the second LFSR with the purple input marker of the clock. After pressing *Play* we will get 15 output bits out of both LFSRs. LFSR 1 should produce the sequence "001111100110100" and LFSR 2 the sequence "001001000000000". Figure 5.4 shows our current workspace.

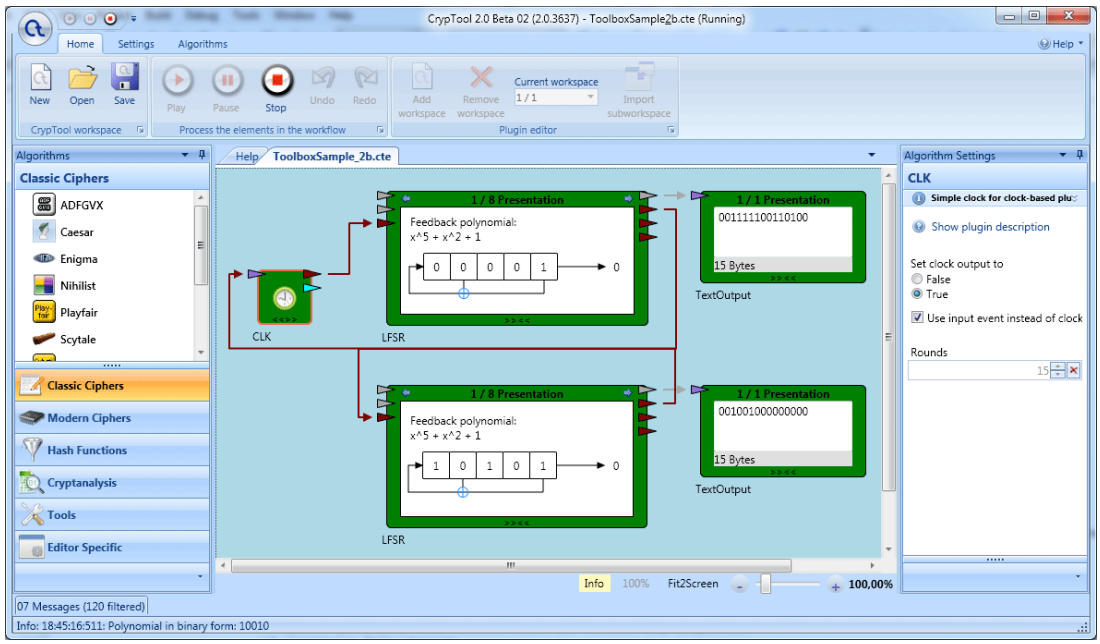


Figure 5.4.: CLK plugin with feedback connected to an LFSR, which itself clocks another LFSR.

Lessons learned

You now have a deeper understanding on using the LFSR plugin: You know how a polynomial or a tap sequence can be entered in the settings. You can now control an LFSR via an external clock signal, know that the output depends on the value of the clock signal, and are able to setup an LFSR that produces an output even on false clock events.

5.2. Building and Analyzing a Simple Keystream Generator

5.2.1. Building an Alternating Step Generator

Summary

In this advanced scenario we will assemble the alternating step generator presented before on page 10. For a better understanding we present the figure again:

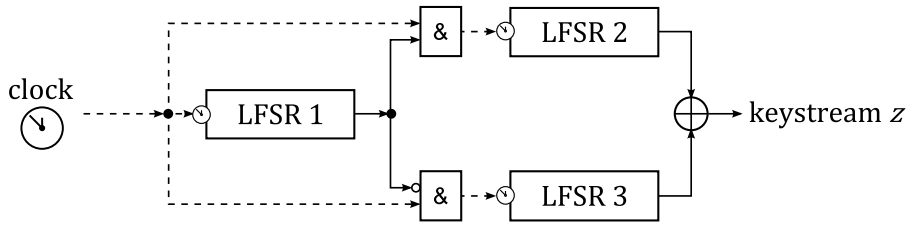
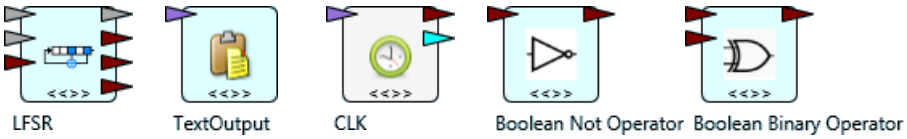


Figure 5.5.: The alternating step generator.

Plugins used in this scenario



Scenario walkthrough

Open a new workspace in CT2. Add 3 LFSRs to the workspace and change their names to LFSR 1, LFSR 2, and LFSR 3 by clicking on the name below the LFSR icon. Set the Clock Properties of all LFSRs to both "Use external clock" and "Always create output".

Now set the polynomials and seeds of the three LFSRs as follows:

- LFSR 1:

  Polynomial: x^3+x^2+1

  Seed: 001

- LFSR 2:

  Polynomial: x^4+x^3+1

  Seed: 1011

- LFSR 3:

  Polynomial: x^5+x^4+x^3+x+1

  Seed: 01001

Add a CLK plugin to your workspace and check the "Use input event instead of clock" checkbox in its settings. Set the rounds to 20. Connect the first output of the CLK plugin with the third input of LFSR 1, as well as the second output of LFSR 1 with the third input of LFSR 2. Now add a Boolean

Not Operator plugin [▷] (which can be found in *Tools* → *Throughput*) to the workspace and rename it to "NOT". Connect the second output of LFSR 1 with the input of the NOT plugin. As you may have noticed, it is possible to connect an output of a plugin to more than one other plugin. Add a Boolean Binary Operator plugin [D] to your workspace, rename it to "XOR" and set the "Operator Type" in the settings correspondingly. If it is not set yet, check the checkbox "Update needs both inputs". We will now connect LFSR 2 and LFSR 3 with the XOR. Choose the second output (red marker) of both LFSRs to connect them to one of the inputs of the XOR. Add a TextOutput plugin to the workspace, rename it to "Output Bits" and connect the output of the XOR with the TextOutput plugin. Finally, connect the output of the XOR with the input of the CLK plugin. Your workspace now should look like Figure 5.6.

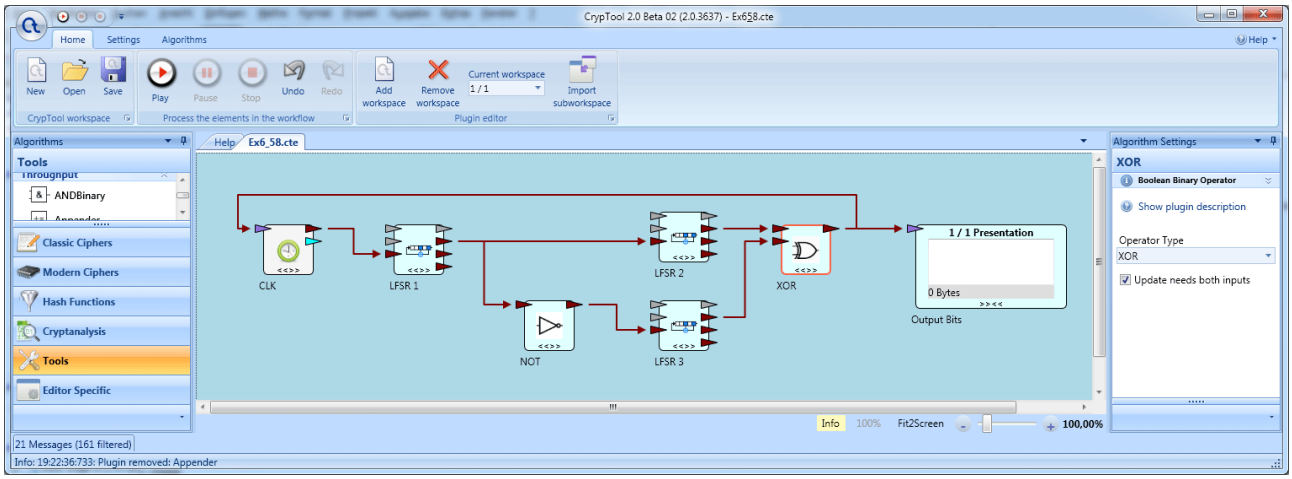


Figure 5.6.: The alternating step generator assembled in CT2.

After pressing *Play* you may notice, that the TextOutput plugin displays only "True" or "False". This results from the fact, that we only get single boolean values from the XOR plugin. To get a sequence of zeros and ones, we have to adjust the settings of the TextOutput as follows: The first action that has to be taken is to check the "Display Boolean as numeric value" checkbox. Then open the Append section to set the "Append n-breaks" to zero, and finally check the "Append text input" checkbox. After pressing the *Play* button again, your setup should now generate the following output sequence: 10111010101000010111.

Note, you can also start your workspace by simply pressing F5 on your keyboard. The key combination Shift + F5 stops it.

Lessons learned

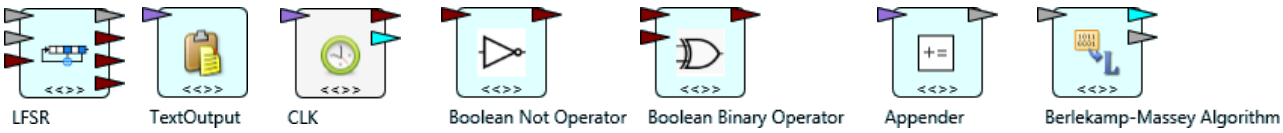
You are now able to assemble a complex binary generator with the use of elementary boolean functions.

5.2.2. Determining the Linear Complexity of the Generator Using the BMA

Summary

In this scenario we will determine the linear complexity of the above assembled alternating step generator. The complexity L can be measured by the Berlekamp-Massey algorithm.

Plugins used in this scenario



Scenario walkthrough

Open the above created workspace in CT2. Add the Berlekamp-Massey Algorithm plugin [] from *Tools* → *Throughput* to the right hand side of your workspace. Open the QuickwatchPresentation. As we want to check the linear complexity of the binary stream that is generated by our setup, we need to connect the XOR with the BMA plugin. As you may have noticed, the output of the XOR has a red marker (which states that this output has a boolean value) and the input of the BMA is grey (which represents a string). If you now try to connect these both you would not have success, since they are not of the same type. We need another plugin to "translate" the boolean value into a string.

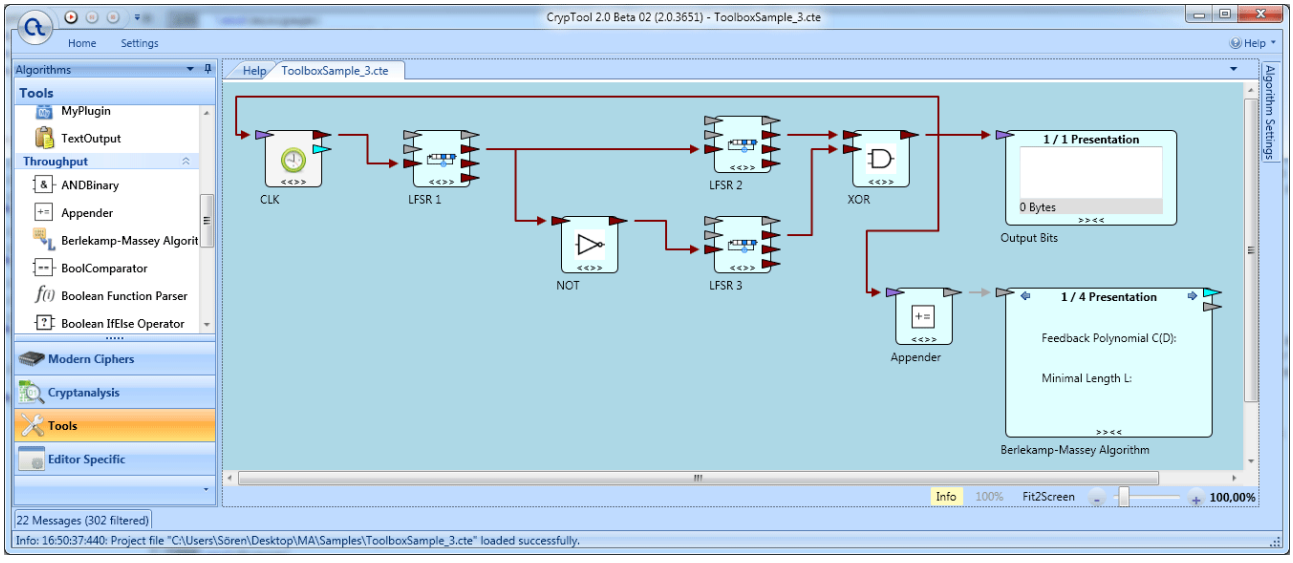


Figure 5.7.: Determining the linear complexity of the alternating step generator.

To get the XOR and the BMA plugin connected, we need to add the Appender plugin from *Tools* → *Throughput* to our workspace. Place it between the XOR and the BMA and connect all three plugins. The Appender now takes the single boolean values from the XOR, converts them into the characters '0' and '1', appends them internally to a string, and forwards the string to the BMA plugin. Since we expect a high linear complexity we need a long output sequence of our keystream generator and hence set the rounds in the settings of the CLK plugin to 150. If your workspace now looks similar to Figure 5.7, hit *Play* to start.

You may notice that the determined linear complexity and the corresponding polynomial change during the generation of the 150 bits, but the final complexity L determined by the BMA plugin should be 63 as depicted in Figure 5.8.

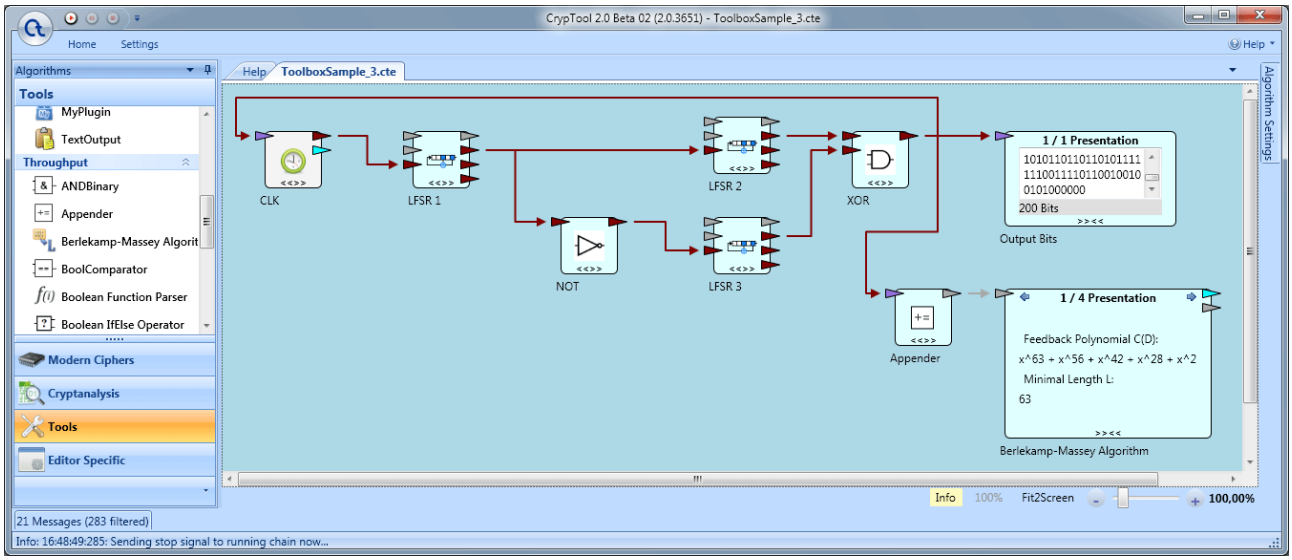


Figure 5.8.: Final result of the linear complexity of the alternating step generator.

We found out that our assembled generator has a complexity of 63 and that its output sequence could also be generated by a single LFSR with the same length by using the following feedback polynomial: x^63 + x^56 + x^42 + x^28 + x^21 + x^14 + 1. Hence, we generated a stream that could also be generated by a simple LFSR with a polynomial of degree 63. But we did it by combining three LFSRs with a maximal degree of only 5, which is a great success in destroying the linearity.

Using your gained knowledge you should now be able to determine the complexity of each of the single LFSRs.

Lessons learned:
You are now able to measure the linear complexity of an LFSR or an assembled generator by using the Berlekamp-Massey Algorithm Parser plugin.

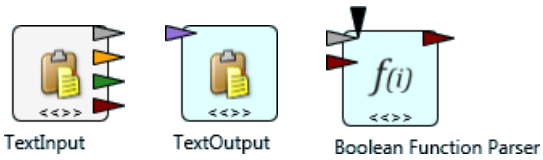
5.3. Using the Boolean Function Parser

5.3.1. Basic Usage of the Boolean Function Parser Plugin

Summary

In this basic scenario we will show how to use the Boolean Function Parser plugin. We will solve a boolean function by using three data sources for our variables: two external ones and data we will enter in the QuickwatchPresentation.

Plugins used in this scenario



Scenario walkthrough

At first create a fresh workspace in CT2 by clicking on the *New* button. Then add the Boolean Function Parser plugin [f(i)] from *Tools* → *Throughput* to your workspace. By opening the QuickwatchPresentation you can see two text fields: One for entering the boolean function and one to enter the data. Data means the values that replace the variables in our boolean function. We enter the following string as the boolean function: "x0.0 + x1.0". Since we want to use data from two external sources, we have to set the "Number of inputs" in the settings of the BFP plugin to two.

Now add two TextInput plugins, open the Quickwatch of them and enter "01" in the first and "10" in the second TextInput. Connect the last output (red marker) of each TextInput with one of the two boolean array inputs (red marker) of the BFP plugin. To get the output of the parser plugin we have to connect a TextOutput plugin to its output. Add a TextOutput to the workspace and connect the parser to it. As before check the "Display boolean as numeric value" checkbox in the settings of the TextOutput plugin. Press *Play* to run the chain. The BFP should output "1" and your workspace should look similar to Figure 5.9.

So why is the output "1"? Our output is the outcome of XORing "x0.0" and "x1.0". "xi.j" is a variable as described in Section 4.2.5 on page 32: x starts a variable, i denotes the input starting at zero, and j is the array index of the values (also starting at zero). So "x0.0" refers to the first value of the first input, which is "0". In the same way "x1.0" refers to the first value of the second input, which is "1". So our boolean function is interpreted as "$0 \oplus 1 = 1$". Now change the indices of our function in the QuickwatchPresentation of the BFP a bit to get familiar with the plugin. You could also try "*" instead of "+" and use parentheses.

So far, we have completely ignored the second text field in the QuickwatchPresentation of the BFP plugin. In this field you can also add boolean values to be processed in a boolean function. To access them use the variable "xqj", j starting again at zero. Try to build boolean functions using the quickwatch variable "xqj".

Another variable is the memory "m" of the Boolean Function Parser. You may have noticed the "Memory Bit" label in the presentation. This variable holds the last output of the BFP plugin. A possible boolean function using all described variables could look like as follows: "(x0.0 + x0.1) * 1 + m * xq0". Start and

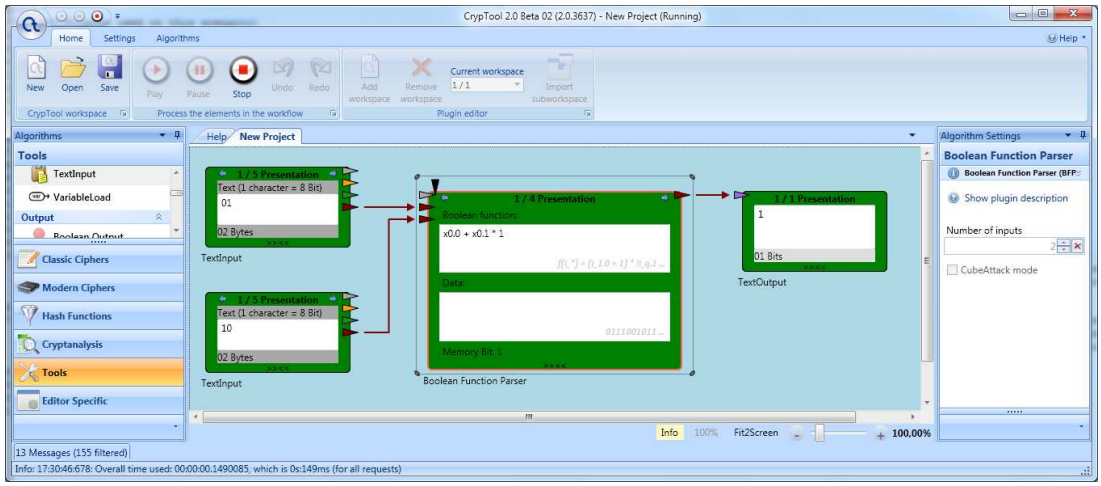


Figure 5.9.: Boolean Function Parser plugin sample.

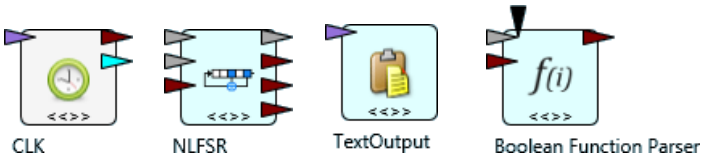
stop your workspace a few times to see the memory changing its value.

Lessons learned:
You are now able to use the Boolean Function Parser and compute binary boolean functions using values from external and internal sources.

5.3.2. Assembling the Cipher Achterbahn-80 in CrypTool 2.0

Summary
In this advanced scenario we will assemble the Achterbahn-80 cipher with the help of our toolbox. Basically we need a lot of NLFSRs whose outputs are combined by a boolean function.

Plugins used in this scenario



Scenario walkthrough
At first create a new workspace in CT2. Now add 11 NLFSRs from *Tools* → *Throughput* to the workspace and give them names from A1 to A11. Now change the clock properties of all 11 NLFSRs to "Use external clock". Fill out the polynomials and seeds as defined in the Achterbahn paper as follows[1]:

- A1:

  Polynomial: x0 + x1 + x5 + x6 + x8 + x13 + x15 + x1x3 + x1x7+ x1x13 + x4x12 + x5x11 + x6x12 + x7x9 + x1x11x14 + x1x4x11x14 + x1x7x11x14 + x1x4x10x11x14 + x1x7x9x11x14

[1]The polynomials and seeds can also be found at http://www.soerenrinne.de/Achterbahn/Achterbahn80PolynomialSeed.txt.

+ x1x10x11x12x14

Seed: 0111010011000100000010

- A2:

  Polynomial: x0 + x4 + x5 + x13 + x16 + x1x6 + x1x7 + x4x6 + x5x11+ x7x9 + x8x11 + x12x14 + x1x5x9x15 + x1x9x10x15 + x1x3x9x11x15 + x1x5x9x11x15 + x1x8x9x11x15 + x1x9x10x11x15

  Seed: 11001100000010101101011

- A3:

  Polynomial: x0 + x2 + x3 + x6 + x8 + x12 + x18 + x1x11 + x1x15+ x2x13 + x4x13 + x6x15 + x12x13 + x13x14 + x2x5x6 + x2x5x14+ x2x6x7 + x2x7x14 + x5x6x7 + x5x7x14 + x1x2x5x15 + x1x2x7x15 + x1x5x7x15 + x2x5x6x15 + x2x5x9x14 + x2x5x9x16 + x2x6x7x15+ x2x7x9x14 + x2x7x9x16 + x5x6x7x15 + x5x7x9x14 + x5x7x9x16

  Seed: 111110010011101110101111

- A4:

  Polynomial: x0 + x6 + x11 + x20 + x1x5 + x3x5 + x4x12 + x5x14 + x6x16+ x7x16 + x8x15 + x8x17 + x15x17 + x2x3x14 + x2x5x14 + x5x8x15+ x5x8x17 + x5x12x13 + x5x12x14 + x5x15x17 + x2x3x8x15 + x2x3x8x17 + x2x3x12x13 + x2x3x12x14 + x2x3x15x17 + x2x5x8x15 + x2x5x8x17 + x2x5x12x13 + x2x5x12x14 + x2x5x15x17

  Seed: 11011011001010000011110100

- A5:

  Polynomial: x0 + x4 + x5 + x15 + x16 + x17 + x21 + x2x4 + x2x18+ x3x6 + x4x13 + x12x13 + x3x4x10 + x3x4x15 + x3x10x14 + x3x14x15 + x4x10x15 + x10x14x15 + x3x4x10x13 + x3x4x13x15 + x3x7x10x11 + x3x7x10x14 + x3x7x11x15 + x3x7x14x15+ x3x10x12x13 + x3x12x13x15 + x4x10x13x15 + x7x10x11x15 + x7x10x14x15 + x10x12x13x15

  Seed: 011100110111001110111011101

- A6:

  Polynomial: x0 + x3 + x4 + x15 + x25 + x1x3 + x1x8 + x1x12 + x6x17+ x10x13 + x10x17 + x13x14 + x5x10x11x18 + x2x5x11x16x18 + x2x5x11x17x18 + x5x10x11x13x18 + x5x11x13x14x18 + x5x11x16x17x18

  Seed: 00110100001100000111011111 1

- A7:

  Polynomial: x0 + x1 + x5 + x20 + x25 + x1x2 + x2x17 + x4x12 + x10x15 + x10x18 + x14x16 + x16x20 + x7x9x18x19 + x7x9x10x15x19 + x7x9x10x18x19 + x1x2x7x9x13x19

  Seed: 11100010110011111101110110010

- A8:

  Polynomial: x0 + x2 + x10 + x11 + x17 + x18 + x21 + x24 + x1x4 + x8x21+ x10x21 + x13x19 + x6x15x19 + x8x9x18 + x13x14x16 + x13x14x19 + x13x15x19 + x6x14x15x16 + x6x14x15x19 + x8x9x18x19 + x13x14x15x16 + x13x14x15x19 + x8x9x14x16x18 + x8x9x14x18x19

  Seed: 11000100000110011100011100111

- A9:

  Polynomial: x0 + x1 + x7 + x10 + x12 + x18 + x28 + x2x8 + x4x7 + x4x18+ x10x12 + x10x19 + x10x22 + x14x22 + x3x5x7 + x3x7x8 + x5x7x8+ x1x3x5x9 + x1x3x5x21 + x1x3x8x9 + x1x3x8x21 + x1x5x8x9 + x1x5x8x21 + x3x4x5x7 + x3x4x5x18 + x3x4x7x8 + x3x4x8x18 + x3x5x9x21 + x3x8x9x21 + x4x5x7x8 + x4x5x8x18 + x5x8x9x21

  Seed: 10011000000101100110011101110 0

- A10:

  Polynomial: x0 + x2 + x5 + x6 + x15 + x17 + x18 + x20 + x25 + x8x18+ x8x20 + x12x21 + x14x19 + x17x21 + x20x22 + x4x12x22 + x4x19x22+ x7x20x21 + x8x18x22 + x8x20x22 + x12x19x22 + x20x21x22 + x4x7x12x21 + x4x7x19x21 + x4x12x21x22 + x4x19x21x22 + x7x8x18x21 + x7x8x20x21 + x7x12x19x21 + x8x18x21x22 + x8x20x21x22 + x12x19x21x22

  Seed: 10100110110100110110110011100 10

- A11:

  Polynomial: x0 + x3 + x17 + x22 + x28 + x2x13 + x5x19 + x7x19 + x8x12+ x8x13 + x13x15 + x2x12x13 + x7x8x12 + x7x8x14 + x8x12x13 + x2x7x12x13 + x2x7x13x14 + x4x11x12x24 + x7x8x12x13 + x7x8x13x14 + x4x7x11x12x24 + x4x7x11x14x24

  Seed: 00010100111001100101001111101101

Now add the Boolean Function Parser plugin from *Tools* → *Throughput* to the workspace, rename it to "Boolean Function G", and set the number of inputs to 11. Open the QuickwatchPresentation and fill in the following boolean function[2]:

x0.0 + x1.0 + x2.0 + x3.0 + x4.0 + x6.0 + x8.0 + x10.0 + x1.0x9.0 + x1.0x10.0 + x3.0x7.0 + x4.0x5.0 + x5.0x7.0 + x5.0x9.0 + x5.0x10.0 + x6.0x7.0 + x7.0x8.0 + x7.0x9.0 + x8.0x9.0 + x8.0x10.0 + x0.0x1.0x7.0 + x0.0x3.0x9.0 + x0.0x3.0x10.0 + x0.0x7.0x8.0 + x0.0x8.0x9.0 + x0.0x8.0x10.0 + x1.0x2.0x7.0 + x1.0x3.0x7.0 + x1.0x3.0x9.0 + x1.0x3.0x10.0 + x1.0x6.0x7.0 + x1.0x7.0x9.0 + x1.0x7.0x10.0 + x1.0x8.0x9.0 + x1.0x8.0x10.0 + x2.0x3.0x7.0 + x2.0x7.0x8.0 + x3.0x6.0x7.0 + x3.0x7.0x8.0 + x4.0x5.0x7.0 + x4.0x5.0x9.0 + x4.0x5.0x10.0 + x5.0x7.0x9.0 + x5.0x7.0x10.0 + x6.0x7.0x8.0 + x7.0x8.0x9.0 + x7.0x8.0x10.0 + x0.0x1.0x2.0x7.0 + x0.0x1.0x6.0x7.0 + x0.0x2.0x4.0x7.0 + x0.0x2.0x7.0x8.0 + x0.0x3.0x7.0x9.0 + x0.0x3.0x7.0x10.0 + x0.0x4.0x6.0x7.0 + x0.0x6.0x7.0x8.0 + x0.0x7.0x8.0x9.0 + x0.0x7.0x8.0x10.0 + x1.0x2.0x3.0x7.0 + x1.0x2.0x4.0x7.0 + x1.0x3.0x6.0x7.0 + x1.0x3.0x7.0x9.0 + x1.0x3.0x7.0x10.0 + x1.0x4.0x6.0x7.0 + x1.0x7.0x8.0x9.0 + x1.0x7.0x8.0x10.0 + x2.0x3.0x7.0x8.0 + x3.0x6.0x7.0x8.0 + x4.0x5.0x7.0x9.0 + x4.0x5.0x7.0x10.0

After setting the boolean function, connect the third output (red marker) of each of the NLFSRs with one of the inputs of the BFP plugin in correct order. Connect A1 with the first input, A2 with the second, and so on. To get the output of the BFP, we have to add a TextOutput from *Tools* → *Output* to our workspace. Rename it to "Keystream Output" and connect the output of the BFP with the TextOutput plugin. In the settings of the TextOutput check the "Display boolean as numeric value" checkbox and append the text input with 0 breaks ("Append" section in settings). Now the only thing left to do is clocking the 11 NLFSRs with a clock. To do so, add the CLK plugin from *Tools* → *Input* to your workspace and connect the clocking output (red marker) of the clock with the clocking input (third input, red marker) of

[2] See http://www.soerenrinne.de/Achterbahn/Achterbahn80GFunction.txt for a digital version

all 11 NLFSRs. In the settings of the CLK plugin, set the rounds to "16" and check the "Use input event instead of clock" checkbox. Finally connect the BFP output with the CLK input and click *Play* to start your assembly if it looks similar to Figure 5.10.

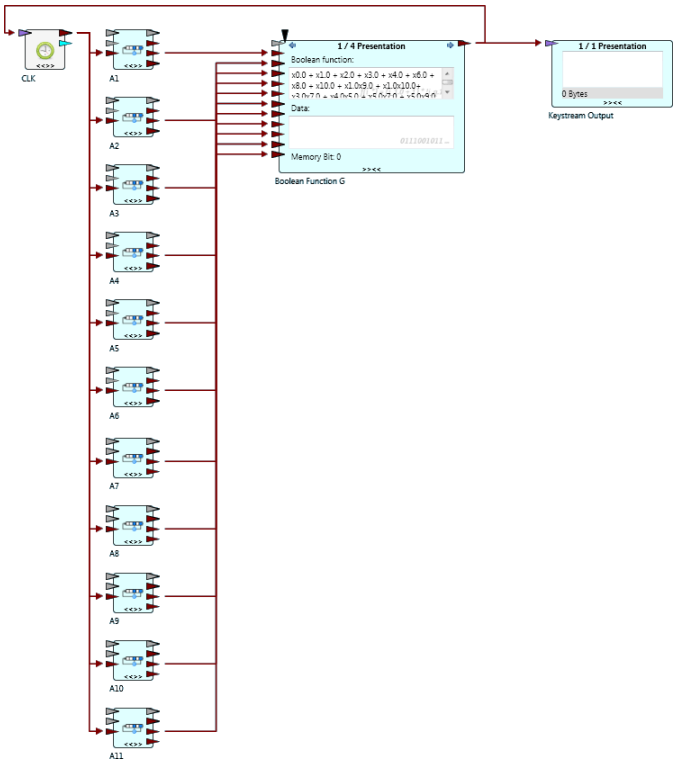


Figure 5.10.: Achterbahn-80 cipher in CT2

We now have successfully assembled the Achterbahn-80 cipher. The output should be "1001110011101111" which is "9CEF" in hex.

Lessons learned:
With the help of our toolbox you are now able to assemble complex ciphers.

Remarks on the Achterbahn-80 assembled in CT2:
Since our (N)LFSRs do not support to feed in a specific key as it is necessary in Achterbahn-80, we skipped the setup and warmup phase of the cipher. The seeds presented above are taken from the reference implementation of Achterbahn[3] after successfully running the setup and warmup phase. The key and initialization vector (IV) used to generate the keystream above are as follows:

- Key: 0102030405060708090A (hex)
- IV: 0102030405060708090A (hex)

[3] `http://www.matpack.de/achterbahn/achterbahn-128-80.zip`

6. Conclusion and Further Work

In this chapter we will present a summary of our work and the possibilities of our toolbox (Section 6.1). We will point out problems concerning the CT2 architecture in Section 6.2. In Section 6.3, we will compare our implementation to the work of others. Finally we present possible future work and enhancements in Section 6.4.

6.1. Achieved Goals

Our ambition was to implement a toolbox which allows the user to easily create and analyze stream ciphers. After analyzing the design of several stream ciphers from eSTREAM, the ECRYPT Stream Cipher Project[1], we were able to specify main components of stream ciphers that were used quite often: Feedback shift registers in conjunction with a combining boolean function. Linear and nonlinear shift registers appeared in various characteristics and combinations.

After stating the mathematical background of these main components we gave an introduction to the host application our toolbox is nested into: CT2, one of the successors of the well known e-learning platform for cryptography and cryptanalysis CrypTool. CT2 offers a plugin-based architecture which allows us to implement each component of our toolbox as a single plugin. This approach makes it very easy to extend the stream cipher toolbox with new tools. Another advantage is the compatibility with other plugins which are already part of CT2. Existing plugins can be combined with our newly implemented ones.

In the next part of our work we described and discussed the implementation of our plugins. We presented the in- and outputs of the plugins, the settings and the appearance in CT2, and finally the main methods of the source code.

To state the possibilities of our toolbox we then introduced several scenarios, in which the components were used to build keystream generators. These generators were an implementation in our toolbox of the ones that have been described before in the theoretical section. We were able to analyze the keystream generators, e.g. by determining the linear complexity. Furthermore, we built an existing stream cipher called Achterbahn-80 with the help of our toolbox.

All in all, we successfully implemented a toolbox which can be used to build keystream generators without any knowledge of a programming language. This toolbox can easily be used in teaching to demonstrate the inner assembly of a component like the LFSR or build primitive generators as described in the scenarios of Chapter 5.

[1] http://www.ecrypt.eu.org/stream/

6.2. Disadvantages of the (Current) CrypTool 2.0 Architecture

In the following, we present two drawbacks of the CrypTool 2.0 architecture. Section 6.2.1 provides a general performance analysis of CT2 and Section 6.2.2 is about concurrency on the CT2 workspace.

6.2.1. Performance Losses

By assembling a complex stream cipher with the help of our toolbox, we recognized some disadvantages of the host application. The overall performance of our implemented stream generators was very poor, which is due to the (current) editor architecture of CT2. The editor in which the different plugins can be assembled is event driven and more or less focused on the presentation for the user. This leads to a clearly arranged workspace in which details of a cryptographic function are depicted in an ostensive way but lacks in a good performance. The connection between two plugins, for example, flashes every time something is sent from one plugin to the other, which makes it easy to understand the correlation between the plugins of a complex workspace. But this vast use of the WPF architecture seems to slow down the whole workspace, making it nearly impossible to assemble complex systems with good performances inside CT2. So the design of the GUI of CT2 is very clear but the performance in our case is bad: About 30 seconds were needed to generate a 16 bit long keystream with our Achterbahn-80 setup of Section 5.3.2, which normally should take only milliseconds.

In other scenarios, where performance is needed, the CT2 team introduced the IControl interface mentioned before. In our toolbox we are not able to use the IControl interface, since our plugin chain is completely flexible. A Plugin A for example provides the input for a plugin B, whose output is the input for a plugin C. The IControl interface assumes, that plugin A provides the input for plugin B and additionally receives the output of plugin B for further use. The IControl interfaces is not designed to hand the output of plugin B over to a plugin C. Furthermore we have to implement the IControl interface inside a plugin for another *specific* plugin, which leads to a vast number of IControl interfaces at the plugins of our toolbox to be able to connect the components of our toolbox flexibly. So, to use the IControl, we already need to know in the implementation phase how the components of the toolbox will be assembled later on. Since this cannot be known by the developer, we need other ways to speed up the performance.

However, the CT2 team realized these problems on its last team meeting in December 2009 and is now planning to release a new editor for their application. The new editor should catch up on that issue.

6.2.2. Concurrency on the CrypTool 2.0 Workspace

Another problem we faced was the concurrency on a workspace. When assembling plugins in such a way, that two or more parallel plugin executions are performed, whose results are combined in a single plugin later on, concurrency occurs. The parallel plugins may have different runnning times which leads to chronological shifted time of arrival at the combining plugin. Normally the plugin would execute if any of the plugin inputs fires an event. This leads in our parallel case to a false outcome, because the values of the other parallel working plugins may not be fresh at that time. Figure 6.1 depicts that problem.

One approach is to introduce flags to every input of a plugin. Each flag is set, if a new input event has arrived. The plugin is now only executed when every input flag is set, which means that every input value is a new input value and the execute method can be called. In the execute method all flags are unset again. Figure 6.2 depicts that approach starting at step 3.

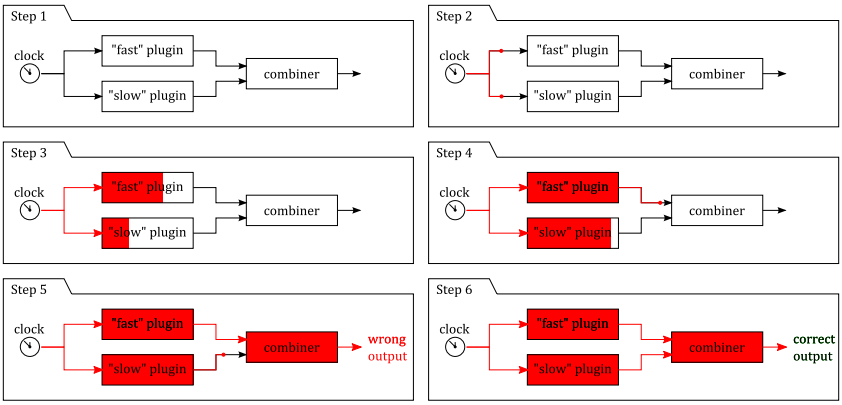


Figure 6.1.: Concurrency on a workspace with two parallel plugins and a combiner.

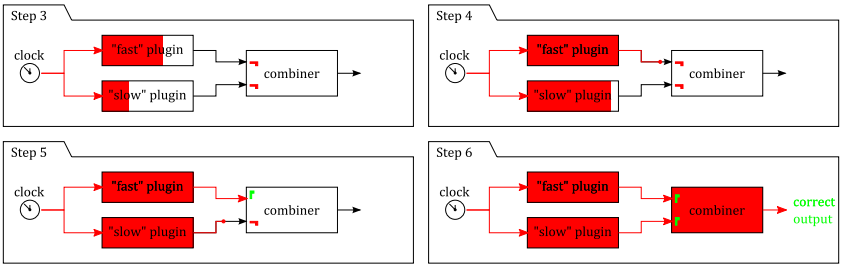


Figure 6.2.: Concurrency on a workspace with two parallel plugins and a combiner solved by flags.

The flag-approach is working in most of the cases like simple boolean combinations as AND, XOR, or the Boolean Function Parser plugin. This approach is based on the assumption, that every input (for example of an XOR) is needed to execute the plugin. But what about a case, in which we do not know how many input events we get? The cipher A5/1 is a good example for that problem. Again take a look at Figure 2.9 on page 10. The XOR behind the three LFSRs has three inputs. Handling three inputs in an XOR is not the problem. The problem is, that on each round two *or* three new inputs may contribute to the XOR function. So we do not know for how many inputs we have to wait until we are ready to execute the XOR. We need to know from which LFSR we can expect a new input, or, in other words, an LFSR that does not output in the current round needs to forward this information to the XOR plugin. Since the LFSRs only output boolean values which include only two states, we cannot output something like a "-1" to tell the XOR plugin that this specific output can be ignored. Therefore the LFSRs' outputs have to be changed to something like integer instead of boolean and the XOR plugin has to be extended in accepting three or more inputs and ignoring special inputs like "-1". This both leads to high implementation costs.

6.3. Our Toolbox Compared to Other Projects

As stated in Section 1.1, there are at least two other projects dealing with the presentation of LFSRs or stream ciphers in general.

The implementation of Wagner [16] solely focusses on LFSRs as a component of stream ciphers. The implementation is limited since it only supports up to three LFSRs to be connected by one specific aggregator

(XOR, a multiplexer, or a majority function) at a time due to GUI limitations. These connected LFSRs are called a "scheme", which unfortunately cannot be saved for further use. Another disadvantage is that the LFSRs cannot be stepped by an external clock (e.g. to generate irregular clocked LFSRs). Nearly all possible further development proposed in his work and the before mentioned disadvantages (like saving complete workspaces) are implemented in our toolbox.

The work of Bertram [1] also focusses on LFSRs and implements the BMA as we did. In contrast to our work the BMA is shown step by step, which may be added to the QuickwatchPresentation of our BMA plugin (see also Section 6.4). The auto-correlation function can also be computed. To combine several LFSRs in Bertram's work, one has to compute the bitstreams of the LFSRs first, and then type in a function that combines these streams. Given three output streams named "f0", "f1", and "f2" from three LFSRs, the user has to provide a function similar to the following: "and(f0, xor(not(f1),f2))". This combination can be done on the fly and in a more intuitive way in our implementation by using the Boolean Binary Operators plugins or combining the three outputs directly with the Boolen Function Parser plugin. Clocking an LFSR by another one is also not part of the implementation of Bertram, but all LFSRs and bitstreams can be saved in files for later use as in CT2.

6.4. Enhancements to be Made

The following enhancements may be added to the plugins of our toolbox or the toolbox itself:

Introduce a key into LFSRs and NLFSRs
The (N)LFSR plugin for example lacks of a possibility to introduce a key to the FSR like it is needed in the setup phase of Achterbahn-80. In that phase the whole key is fed bitwise into the register instead of the feedback bit.

External variables in feedback function
A possibility to integrate external variables into the feedback function, like in the cipher Grain [9] or TRIVIUM, is missing, too.

Too many outputs
The (N)LFSR plugins have a lot of different outputs which may confuse a beginner. The outputs could be implemented in a dynamic way concerning both the number and the type. Since the dynamic property is still under construction, we added this feature only to our BFP plugin.

Add FCSRs
In general, our toolbox may be extended by an FCSR plugin. An FCSR is a feedback with carry shift register which solves some insufficiency of LFSRs and are more resistant to known attacks as described in [14].

Enhance QuickwatchPresentation of BMA plugin
The QuickwatchPresentation of the BMA plugin could be enhanced by a step-by-step presentation of the Berlekamp-Massey algorithm as shown in Bertram [1].

A. Bibliography

[1] C. Bertram. Entwurf und Implementierung eines Werkzeugs zum Rechnen mit Schieberegisterfolgen, April 2006.

[2] E. Biham and O. Dunkelman. Cryptanalysis of the A5/1 GSM stream cipher. *Lecture Notes in Computer Science*, pages 43–51, 2000.

[3] A. Bourbahh. Entwicklung einer Toolbox in Cryptool 2.0 zur Analyse algebraischer Angriffe. Master's thesis, Ruhr-University of Bochum, 2010.

[4] J. Buchmann. *Einführung in die Kryptographie*. Springer, 4th edition, 2008.

[5] C. de Cannière and B. Preneel. Trivium - A Stream Cipher Construction Inspired by Block Cipher Design Principles. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/030, 2005. `http://www.ecrypt.eu.org/stream`.

[6] I. Dinur and A. Shamir. Cube Attacks on Tweakable Black Box Polynomials. In *EUROCRYPT*, volume 5479, pages 278–299. Springer, 2009.

[7] B. Esslinger and CrypTool-Team. *Cryptology with CrypTool v1.4.30 Beta04 Practical Introduction to Cryptography and Cryptanalysis Scope, Technology, and Future of CrypTool*. CrypTool-Project, August 2009. Available from: `http://www.cryptool.com/download/CrypToolPresentation-en.pdf`.

[8] B. Gammel, R. Göttfert, and O. Kniffler. The Achterbahn Stream Cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/002, 2005. `http://www.ecrypt.eu.org/stream`.

[9] M. Hell, T. Johansson, and W. Meier. Grain - A Stream Cipher for Constrained Environments. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. `http://www.ecrypt.eu.org/stream`.

[10] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC, 1996.

[11] Microsoft Corporation. Windows Presentation Foundation (WPF), 2009. Available from: `http://msdn.microsoft.com/de-de/netframework/aa663326.aspx`.

[12] D. Oruba. Developing a Toolbox for CrypTool 2.0 for Analyzing Cube Attacks. Master's thesis, Ruhr-University of Bochum, 2009.

[13] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2009.

[14] D. Stegemann and G. Tabarz. Building Stream Ciphers from FCSRs. 2008.

[15] A. Wacker, M. Saternus, B. Esslinger, and P. Südmeyer. Cryptool 2 website, November 2009. Available from: `http://cryptool2.vs.uni-due.de`.

[16] C. Wagner. Stream Cipher for Education - Implementierung einer Lernumgebung in JCrypTool. 2009.