UNIVERSITÄT KASSEL
FACHBEREICH ELEKTROTECHNIK/INFORMATIK
FACHGEBIET ANGEWANDTE INFORMATIONSSICHERHEIT

Master Thesis

Development of an Evaluation Method for Cryptanalysis of Classical Ciphers in CrypTool 2

Bastian Heuser
Matrikelnummer: 30220193

UNI KASSEL VERSITÄT
Fachgebiet Angewandte Informationssicherheit
Fachbereich Elektrotechnik/Informatik
Universität Kassel

August 8, 2017

Prüfer:
Prof. Dr. Arno Wacker
Dr. habil. Sebastian Petersen
Betreuer:
M. Sc. Nils Kopal

Contents

Acronyms

CT2	CrypTool 2	1
TVG	TestVectorGenerator	3
CAA	CryptAnalysisAnalyzer	3
CCA	CylinderCipherAnalyzer	3
CA	CipherAnalyzer	8
CC	CylinderCipher	10

List of Figures

List of Tables

List of Listings

List of Lists

1 Introduction

Testing, analysis, and verification of software is usually performed through standardized test vectors. Most fields have their own typical test vectors, which makes evaluations comparable. In the field of cryptanalysis of classical encryption algorithms and machines, there are very little test vectors available. Most test vectors have to be generated first hand by the developers and researchers of cryptanalytic methods. This makes the evaluations hardly comparable. Then, the amount of test vectors would be too big to download and store it, taking all the different key formats and ciphers into account. With this thesis, we introduce and implement a solution for this issue. The emphasis of this thesis is the development of a seeded random test vector generator component for CrypTool 2 (CT2)[1], as well as a cryptanalysis analyzer component, which is able to analyze cryptanalytic methods and produce comparable results. These components are designed to provide a basis of test vectors and to give the key tools for directly evaluating and comparing cryptanalytic methods in CT2.

1.1 Motivation

Standardized test vectors are the main tool for the analysis, verification, and evaluation in most software fields. With voice over IP and cellular, one of the most common test vectors are the ‘Harvard sentences” [23]. With data compression, the “Canterbury Corpus” [18] is the standard for testing. With image processing, one of the most used test vectors is the famous “Lenna” image [3].

Using with these same sets of test vectors makes the evaluations comparable. In the field of cryptanalysis, most test vectors are directly provided by the authors of algorithms and the according papers. Although these test vectors are necessary correct implementations to developers, they are not designed to make the results comparable. As mentioned above, the field of cryptanalysis of classical encryption algorithms and machines has very little test vectors available. If developers and researchers generate their own sets of test vectors, their results are only limited or mostly not comparable. Regarding all the different key formats and ciphers, the amount of test vectors has to be quite large. Storing and distributing of a large database of test vectors might be problematic. The motivation of the test vector generator component is, therefore, to make the generation customizable for each case and reproducible. This would make a large database unnecessary.

[1]All statements about CT2 refer to the Version CT 2.1 Beta 1 of July 2017.

The other current issue is poor-quality evaluation of cryptanalytic methods. There is no standardized way of evaluating or analyzing of these methods. Having a standardized evaluation software would make comparable results much easier to produce.

1.2 Problem Definition

For this master thesis we have built two CT2 components concerning the analysis of classical cryptology and cryptanalysis. The first component is a test vector generator that generates seeded random test vectors. The second component is an analyzer for cryptanalytic methods that is able to measure different performance aspects of these methods. These results are comparable and reflect the performance of each method. We have chosen and implemented suitable metrics and techniques for this analysis.

1.3 Goals

The following overall goals for both components and this thesis can be derived from the given problems and the given task:

- (G01) Generation of test vector sets for all classical encryption methods in CT2
- (G02) Evaluation of cryptanalytic methods with variable text length, variable key length, and variable algorithm parameters
- (G03) Development of a general evaluation method for cryptanalysis of classical ciphers in CT2
- (G04) Evaluation of the cryptanalytic CT2 component "CylinderCipherAnalyzer" using (G03)
- (G05) Visualization of the functionality of both components on the basis of (G04)

These five goals build the basis for this thesis. From these five goals and our first conceptual thoughts about the vector generation and analysis, we formulated the fifteen requirements in Section 3.2 on Page 15.

1.4 Structure of the Thesis

We have already explained the motivation and the requirements for the two new components TestVectorGenerator (TVG) and CryptAnalysisAnalyzer (CAA).

In Chapter 2, the fundamentals needed to understand this thesis are defined in detail: What the terms cryptology, cryptography, and cryptanalysis mean, what we mean by classical cryptology, what test vectors have to look like, and the basics about good cryptanalysis evaluation. Moreover, we introduce CT2 and its modular component structure and define necessary functions of the two new components.

The details about the component's functionalities are described in Chapter 3 as well as the concept and design of the components together with a demonstration of their usage using screenshots.

Chapter 4 is describes the implementation in detail. We show some parts of the algorithms we use, how we transform the design into code, and why we make which decisions. Code listings visualize the implementation of the features and the logic behind the components.

In Chapter 5 we provide a detailed guide for using the CAA with other components. We show code listings of our evaluation implementations in the CT2 component CylinderCipherAnalyzer (CCA) and explain the key points for using the full functionality of the CAA.

We test and visualize the functionality of the CAA in Chapter 6 by evaluating the hill climbing algorithm, used in the M-94 cylinder.

At last, Chapter 8 gives an overview about what we have accomplished, which problems we have faced, and which perspectives there are. Besides that, we present ideas for the further development.

2 Fundamentals

This chapter introduces the necessary fundamentals to understand the intention of what we have developed. First, we describe the term “cryptology”, which can be split into classical and modern cryptology. Then we explain what “cryptanalysis” means and demonstrate ways to evaluate cryptanalytic methods. After that, we introduce “CrypTool 2” as well as its two components “CylinderCipher” and “CylinderCipherAnalyzer”, which we have used for the “CryptAnalysisAnalyzer” component, developed in this thesis.

2.1 Cryptology

Cryptology is the study of cryptosystems. It is divided into the two big fields “Cryptography” and “Cryptanalysis”. Cryptography on the one hand deals with securing information, which is done through especially developed cryptographic encryption algorithms among other things. Cryptanalysis on the other hand serves the purpose of simultaneously evaluating and proving the security of these cryptographic procedures. Another part of cryptanalysis is breaking encrypted texts to yield the plaintext, without knowledge of the cryptographic key.

2.1.1 Cryptography

Cryptography is the science and art of writing secure messages. Its main goal is keeping information confidential, disregarding if it is on paper or digital, stored inside a safe, on a hard disk, or in a human brain. It could be sent in a letter, as part of an email, or per voice. It does not even have to be text, it could be any kind of information. Confidentiality can be achieved through encryption. In cryptology, the unencrypted data is referred to as plaintext p. The basic encryption method E takes the two parameters p and key k. The result is the encrypted text, called ciphertext c. Decrypting (D) c with k yields the plaintext p (see Equation 2.1).

$$\begin{aligned} \mathtt{c} &= \mathtt{E_k(p)} \\ \mathtt{p} &= \mathtt{D_k(c)} \end{aligned} \tag{2.1}$$

2.1.1.1 Classical Cryptography

Classical cryptography covers the classical ciphers developed and used before the computer age. These range from manual "pen & paper ciphers" (e.g. ADFGVX), over mechanical cipher machines (e.g. M-209), and electro-mechanical cipher machines (e.g. Enigma), up to electronic ciphers (e.g. T-310/50). It operates directly on traditional characters, which are letters or digits. In contrast to modern cryptography, classical cryptography often tried to keep the entire cryptosystem secret ("security through obscurity"[1]). Apart of a few cipher machine algorithms like Enigma or Sigaba, most algorithms cannot provide confidentiality considering todays analysis capabilities.

2.1.1.2 Modern Cryptography

Succeeding classical cryptography, modern cryptography makes use of computers. It is founded on various mathematical concepts, like number-, probability-, and complexity theory. The operands of modern cryptography are no longer characters, but bit sequences. This renders all cost functions based on letter frequency analysis useless. Following the "Kerckhoffs' Principle", a cryptosystem still has to be secure if everything about the cryptosystem except for the key is public knowledge [12]. Modern cryptographic algorithms try to satisfy this principle and rely only on the key to remain secret to stay secure.

2.1.2 Cryptanalysis

Today, scientists are still interested in classical ciphers. These ciphers are mostly vulnerable to so-called heuristics, like hill climbing, genetic algorithms, Tabu search, simulated annealing, etc. There is a variety of papers and publications that cover these attacks and procedures (e.g. [15][7][6][8]).

2.1.2.1 Heuristic Algorithms

Characterizing aspects of heuristic algorithms are their decision based procedure and searching the best possible approximate solution[2]. Usually, they are non-deterministic, as there is always some randomization involved (the starting point is typically random). Heuristic algorithms only check a fraction of the solution space, which makes them much faster, but also less accurate, less precise, and incomplete. Apart of being faster than a brute force attack that searches the whole

[1] "Security through obscurity" basically intents to raise the effort for the attacker breaking a process or method. It is still possible to break it with this amount of effort.

[2] The approximation is only possible for the analysis in classical cryptography. Heuristic algorithms are useless for analyzing modern cryptographic algorithms, because a very small change in the correct key leads to a random decrypted ciphertext, not providing any information.

solution space, heuristic algorithms are able to find approximate solutions where exhaustive search algorithms cannot find any exact solutions. The approximate solution may even be the best solution.

In cryptanalysis, heuristic algorithms are widely used. On the one hand, when searching for a key, the approximate solution is often sufficient to break a ciphertext. On the other hand, the complexity of a brute force algorithm would often be too big to be used practically. Hill climbing algorithms for example usually try to find the best global solution through modifying the key, comparing the results, searching best local solutions, and comparing them to find the best overall solution. The important thing about classical cryptography and approximating the best key is that the closer the decrypted ciphertext gets to the plaintext, the closer the used key is getting to the correct key. Otherwise, the comparison of different decrypted ciphertexts would retrieve no information.

Other examples of heuristic algorithms are genetic algorithms and Tabu set searches. Genetic algorithms hold a pool of solution candidates which are mutated after a certain period. There are many parameters available for optimization. Tabu set searches save a set of the last wrong candidates. These candidates are not tried again, until they are removed from the set (after a certain time period or number of candidates in the set).

2.1.2.2 Cryptanalysis Evaluation

To be able to evaluate the strength and efficiency of these newly developed cryptanalytic algorithms and methods, researchers and cryptanalysts use the following approaches:

1. Breaking unbroken historical ciphers (e.g. original Enigma messages)
2. Evaluation through self compiled test vectors (test ciphertexts)
3. Evaluation through externally obtained test vectors (test ciphertexts)

Breaking historical recorded ciphers is a rather unsuitable tool to show the strength of an analytic method. It only shows that the method is able to break the specific ciphertext of the historical cipher, but does not offer much information about the general strength of the evaluating analytic method. Evaluating self-compiled test vectors is a more appropriate method. This way, it is possible to use ciphertexts of different lengths, different keys, and other variables to test the limits of an analytic method. The problem of comparability with other methods remains, however, as the other methods used different generated test vectors.

Therefore, the best comparable evaluation tool is the usage of externally obtained (maybe even standardized) test vectors. That way, all evaluated analytic methods use the same data set and allow valid comparability between the algorithms. The main issue is the availability of these test vector sets. Other domains of computer science are the complete opposite with standardized test vector sets, as mentioned in Section 1.1.

2.1.2.3 Evaluation Metrics

A primary goal of research is to show the limits of a method and to compare it with other methods. With this in mind, common metrics for evaluating cryptanalysis of classical ciphers are the following:

1. Probability to successfully break a ciphertext
2. Runtime of the algorithm
3. Necessary quantity of decryptions
4. Decryptions per time entity (analysis speed)

These metrics are evaluated with various parameters, according to the respective cryptanalytic algorithm:

1. Size of the key space or key length
2. Length or quantity of ciphertext
3. Length or quantity of plaintext
4. Used cost function
5. Number of algorithm restarts (e.g. for hill climbing algorithms)
6. Population size (e.g. for genetic algorithms)
7. Tabu set size (e.g. for Tabu searches)

2.2 CrypTool 2

According to [13], CT2 is the successor of the well known e-learning application for cryptography and cryptanalysis 'CrypTool'. It is an open-source project that enables learners, teachers, and developers with interest in the cryptography to try out and apply different ciphers and CipherAnalyzers (CAs) on their own. The modern user interface allows to build both simple and very complex cryptographic algorithms, using simple and intuitive drag and drop methods. This happens through a graphical programming language that was particularly developed for this purpose. Users are enabled to combine the algorithms with each other to create and test their new algorithms and procedures, without special programming knowledge. CT2 is based on the .NET-Framework and the Windows Presentation Foundation (WPF). Furthermore, the architecture of CT2 is completely component-based and modular which simplifies the development of new functionalities. As part of the

CT2 project, a variety of cryptographic algorithms have been developed as components, like AES, SHA1, and the Enigma.

2.2.1 Components

The main feature of CT2 are its components. These components are elementary tools for processing data inside a workspace. Components may have input and output connectors to receive and transmit data between each other. These inputs and outputs may use various data types (we have implemented our own, described in Section 5.2 on Page 5.2). Plugging different components together through connectors is done via a specifically developed graphical programming language. It is designed to enable the average computer user to be able to use it without much knowledge. The component user interface is divided into five different views at most, namely “Presentation”, “Settings”, “Log”, “Data”, and “Help”.

2.2.1.1 Presentation

In the presentation of a component, the process of the component may be visualized. These presentations reach from a best list of heuristic results up to the complex visualization of the Enigma itself. Some presentations also offer direct user interactions.

2.2.1.2 Settings

The settings view lists all the settings of a component, for some components sorted into groups. These settings cover inputs like a drop down menu, a numerical up down input field, or a simple string input amongst others. The settings have a default value and may or may not be changeable while and after the execution of the component.

2.2.1.3 Log

The log lists all messages that occur while executing. These messages include errors and warnings.

2.2.1.4 Data

The data view shows the input and output data of the component.

2.2.1.5 Help

Clicking on help opens the documentation page of the according component. It especially supports unexperienced users to be able to use the component. Together with pre-designed templates for the components, anyone can understand the usage and functionality of the component.

2.2.1.6 CylinderCipher Component

The reason why we have to explain the CylinderCipher (CC) and CCA CT2 components here, is that they are the main components we intended to analyze in this thesis. The TVG and the CAA have been developed to work with the CC and CCA components first, and than extended to work with other components as well.

The first cylinder cipher or "Wheel cipher" named "Jefferson disk" was first invented by Thomas Jefferson in 1795 [4]. A cylinder cipher consists of multiple different lettered disks that can be arranged in any order to en- or decipher letters. Each disk is assigned with a number and a letter so that the order of the disks can be arranged following code words or numbers. It became publicly known, as Commandant Etienne Bazeries independently developed it one century later [11]. His invention was used under the name "M-94" by the United States Army from 1922 to 1945 with 25 disks. The original Bazeries cipher used only 20 disks. In the CT2 component CC, the user is able to choose between the M-94 (25 disks) and the Bazeries cipher (20 disks).

The aluminum disks of a cylinder cipher each contain the Roman alphabet on the outside of the disk. Each disk has a different order of the alphabet, which is mostly random. Once put on a rod and screwed together at the end, the disks cannot be interchanged. The order of the disks corresponds to the key in use. With 25 disks (M-94) there are 25! possible keys, which is similar to an 84-bit key size. Each disk encrypts one letter at a time. Hence, a 25 disk cipher like the M-94 encrypts 25 letters in one step. This is done by rotating each disk individually until the 25 plaintext letters align in one horizontal line. After completing that, one of the other 25 lines can be chosen as ciphertext. Upon decryption, the ciphertext letters are aligned in a horizontal line and one of the other 25 lines should be readable as plaintext.

The CylinderCipher (CC) component is able to encrypt or decrypt. Furthermore, the separator between each key symbol, the separator between the disk order and the offset, how to handle invalid characters, and the case sensitivity can be chosen through the settings of the component. Its input connectors are plaintext and key, both as strings. The key input contains the disk positions and the offsets of each disk. The components output is the ciphertext.

2.2.1.7 CylinderCipherAnalyzer Component

The CCA gives the user the possibility to choose between the extensive offset search and the hill climbing of the disk positions. Executing the offset search on two processor cores takes multiple hours, while the hill climbing algorithm mostly runs for seconds, depending heavily on the restarts. Evaluating the offset search with various input values on one computer would take weeks. That is the reason why the evaluation task for this thesis was reduced on the hill climbing algorithm only. The hill climbing algorithm only works with correctly given offsets. The algorithm restarts are adjustable by the user.

For each restart, the algorithm generates a random key and permutes the disk in two nested loops. In the first loop, the input ciphertext is decrypted with the current run-key (that is set to the current best key at the end of the inner loop). Then in the inner loop, two key elements next to each other are swapped and the ciphertext is decrypted in place with this new key. After that, the cost function chosen by the user is calculated. This uses either 3-grams, 4-grams, or both. If the current key costs are better than the best key costs, the current key, key costs, and the according plaintext become the best values. At the end of the inner loop, the two swapped elements are reverted back to the original run-key. In the next loop, two other elements are swapped and continued as described. Moreover, both loops are surrounded by a third one. The third loop repeats the inner loops as long as they return improved results. After the third loop, before the next restart, the local best key costs are compared to the global best key costs. If they are better, they become the global ones, as well as the according key and plaintext become the global best key and global best plaintext. After all restarts are through, the ciphertext is decrypted with the global best key. This yields the best plaintext that is returned in the end. As the letters have been mapped to numbers at the very beginning, they have to be mapped back to text before the return call.

Other settings of the CCA are the number of processor cores, which cylinder cipher should be used (M-94 or Bazeries), and which language is assumed for the according plaintext is in (necessary for the cost function; currently English, German, and Spanish are supported). The two input connectors of the component are the ciphertext and the offsets. The outputs are the best key and its according plaintext.

2.3 Used Technologies

All CT2 components in the context of this thesis have been implemented in C#. The used integrated development environment is Microsoft Visual Studio 2013.

3 Concept and Design of the Test Series Components

In this chapter, we formulate fifteen requirements in total for the TestVectorGenerator (TVG), the CryptAnalysisAnalyzer (CAA), and this thesis in general. We describe the concept of the TVG with all its settings and options and also explain its design. Furthermore, we took a look at related scientific papers concerning the cryptanalysis of classical ciphers and the concept of the CAA including a process model of its evaluation. Multiple screenshots show the components in action and describe their usage.

3.1 The Analysis Setup of the Test Series

The complete analysis can be divided into multiple tasks, displayed as a water fall model in Figure 3.1.

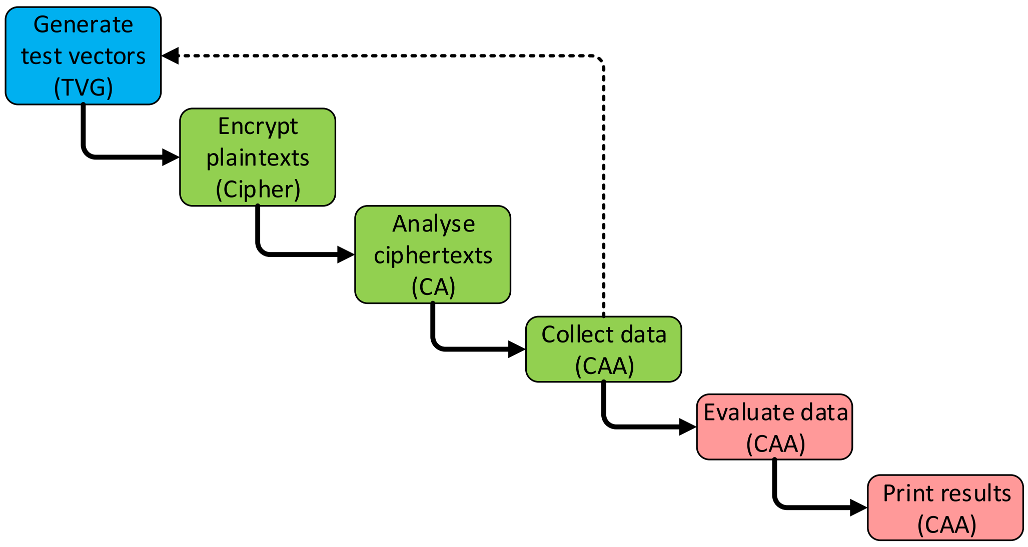


Figure 3.1: Waterfall model of the evaluation tasks

In order to analyze and evaluate, we have to collect the data of the CipherAnalyzer (CA) component we want to analyze. Before getting this data back from the CA component, we have to generate test vectors and feed them to the component.

So starting at the top, the first step is generating test vectors based on the input values. This is done in the TVG. The plaintext has to be encrypted by the according *Cipher*, returning the ciphertext. The CA component is fed with the test vectors including the ciphertext, does its analysis, and returns its best key and plaintext candidates. At this point, the CAA will collect the data and store it. This is done for each repetition of generating test vectors and awaiting the CAs results. After the last key has been processed and saved, the actual evaluation in the CAA starts. The three states of the CAA are highlighted in green (state one), in blue (state two), and in red (state three). The step where the CAA gets the current test vector from the TVG and provides the CA with it is not shown in this visualization. This step belongs to state two, so the CAA would be highlighted in green in this step.

Figure 3.2 visualizes the process flow in a very simplified way, showing the different components of the evaluation setup. The three states of the CAA are highlighted with the same colors, while the CAA is highlighted in yellow (for more details see the algorithm in List 3.10 on Page 35 and the detailed process flow in Figure 5.1 on Page 67).

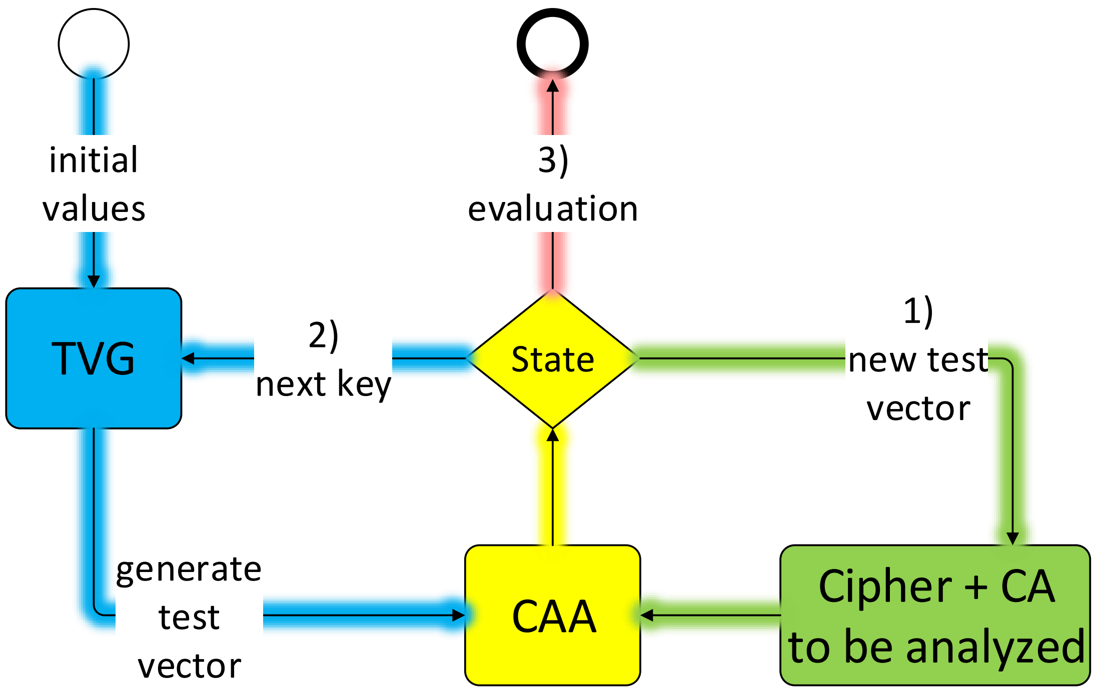


Figure 3.2: Simplified component setup

The most interesting aspect we want to measure are average values of multiple test runs. The information we want to get is the dependency of these averages on different base values. The three base values we chose are ciphertext length, key length, and runtime. Because the runtime highly depends on the used hardware, ideally it should only be used on the same hardware, or on similar hardware at least. This makes the runtime a bad comparison metric in general. It can be completely disabled in the CAA and is designed as an optional feature, as it can be convenient in some cases. Section 3.5.4 goes in detail about the evaluation metrics that we have collected and evaluated. For the sake of completeness, here is a list of all possible metrics: success probability, correctly decrypted percentage, ciphertext length, necessary decryptions, key length, algorithm restarts, population size, Tabu set size, and runtime.

The main process during the evaluation calculates the averages of all the provided metrics (not all metrics are used in each algorithm) in dependency on the ciphertext length, key length, and (if activated) the runtime. The overall averages are displayed via the evaluation output connector of the CAA, together with the amount of keys and ciphertexts for each length. The detailed averages are used in the GnuPlot outputs, described in the next section (3.5.3).

3.2 Requirements

Based on what we have described about the generation of test vectors and the goals in Section 1.3, we have derived the following requirements for the TVG:

- (R01) Accepting a text as input to generate plaintext test vectors
- (R02) Generation of plaintext test vectors
- (R03) Generation of natural language keys taken from input text
- (R04) Generation of simple random keys
- (R05) Generation of random keys through reverse regex
- (R06) Accepting a seed which makes all generations reproducible
- (R07) Settings for the plaintext length, key lengths, and number of test runs
- (R08) Generation of test vector sets (pairs of plaintext and key) for all classical encryption components (ciphers) in CT2

From our description about cryptanalysis and evaluation metrics, we can conclude these requirements that the CAA has to meet:

- (R09) Accepting test vectors from the TVG
- (R10) Feeding test vectors to the Cipher and accept evaluation input from the CA
- (R11) Evaluation of CAs with variable text and key length and variable algorithm parameters
- (R12) Visualization of the testing process

- (R13) Generation of GnuPlot script files and data files that visualize the evaluation results

Other requirements for this thesis are:

- (R14) Evaluation of the CA “CylinderCipherAnalyzer”
- (R15) Visualization of the functionality of the TVG and CAA on the basis of the evaluation in (R14)

3.3 TestVectorGenerator – Concept and Design

The TVG is the CT2 component that is meant to generate pairs of plaintexts and keys as test vectors. Through requirement (R08), TVG has to offer a lot of settings to be able to customize the plaintext and the key. The following sections explain the complete concept behind these settings and ideas.

In order to get plaintexts for testing, the user simply has to provide one large text, a seed value, and the amount of text-key pairs. The keys may be taken from the provided text as well, resulting in natural language keys. Alternatively, they may be randomly generated with a few options or even generated through reverse regex. This makes complex key formats possible. Through this setup, a large reproducible amount of test vectors can be generated, without having to store all of them directly. Only the input values are necessary to reproduce exactly the same set of test vectors.

3.3.1 Inputs and Outputs

The very first requirement for the component is the acceptance of an input text, so we have implemented an input connector that expects a text as string. All generations have to be reproducible, therefore, we have built another input connector that expects a seed as string. The input text and initial seed together with the settings are the only values necessary to reproduce the test vector series (if not using the reverse regex generation). If the reverse regex generation is used, a regex pattern has to be provided through the third input connector. In this mode, a specific alphabet might be provided as well, using the fourth input. The regex pattern input and the alphabet input are also necessary for reproducing the test vector series, if reverse regex is chosen as the key type. The alphabet can also be used for the random key generation, if the simple provided random key formats are insufficient. The seed is updated automatically for each test vector, which results in a new generation of both plaintext and key in the TVG. All input connectors of the TVG are listed in List 3.1.

List 3.1: TestVectorGenerator input connectors

- input text [string] (mandatory)
- seed [string] (mandatory)
- regex pattern [string]
- alphabet [string]

The main output connectors are the generated pair of plaintext and key, both as string. In order to know when the last key has been generated in other components, we implemented an output for the total number of keys (as integer). This value is also used to calculate the progress in other components. For debugging purposes, there is a debug output connector as string. List 3.2 itemizes all output connectors.

List 3.2: TestVectorGenerator output connectors

- plaintext [string]
- key [string]
- total number of keys [int]
- debug output [string]

3.3.2 Settings

The general settings of the TVG and their data type are presented in List 3.3.

The number of test runs is the total number of single test runs per plaintext-key pairs. This number is divided into the different plaintext lengths and key lengths. By the "uppercase only" setting, all letters are formated into uppercase in the plaintext and in the natural and random keys. The settings "delete spaces", "replace ß by sz", "replace umlauts", and "number handling" also modify the plaintext and the natural language keys. The setting "number handling" gives the user the possibility to either ignore, remove, or replace all numbers by their uppercase notation in English or German ("ZERO", "ONE", or "NULL", "EINS")[1].

[1]The CCA currently supports Spanish as third language, but we focused on English and German. All other languages may be implemented at any time.

List 3.3: TestVectorGenerator general settings

- number of test runs [int]
- uppercase only [boolean]
- delete spaces [boolean]
- replace ß by sz [boolean]
- replace umlauts [boolean]
- number handling [enumeration]
- show extended settings [boolean]

The option “show extended settings” shows or hides the settings “maximum text length”, “text length increase”, “uppercase only”, “delete spaces”, “replace ß by sz”, “replace umlauts”, “period symbol handling”, “period replacer”, “number handling”, and “key symbol separator”.

Available settings for the plaintext generation are listed in List 3.4.

List 3.4: TestVectorGenerator plaintext settings

- minimum plaintext length [int]
- maximum plaintext length [int]
- text length increase [int]
- period symbol handling [enumeration]
- period replacer [string]

The plaintext length range is set by the minimum and maximum plaintext length settings. The “text length increase” is the setting for the number of characters the plaintext is extended by from one test run to another (also see Section 3.3.4). The handling of periods is similar to the number handling, but apart of ignoring and removing the periods; they can also be replaced by a replacer symbol that is chosen by the user.

List 3.5 shows the settings for the key generation.

List 3.5: TestVectorGenerator key settings

- generation type [enumeration]
- minimum key length [int]
- maximum key length [int]
- key symbol separator [string]
- key format [enumeration]
- unique symbol usage [boolean]

The key generation provides three different types: natural language keys (from the given text), simple random keys, and reverse regex keys. Section 3.3.5 explains the different types in detail. The key length is bounded by the minimum and maximum key length settings. Each key symbol can be separated by the symbol given in the "key symbol separator" setting. The key format decides which kind of natural language or random key is generated. The options for the natural language key format are: "sentences from text", "numeric key derived from text". Random keys may have one of these formats: "letters", "numbers", "binary", and "use input alphabet". The details about that can also be found in Section 3.3.5.

3.3.3 Input Text Preprocessing

In order to get keys and plaintexts from the given text, our first step is separating the text by periods and putting the sentences into an array. This step also involves all the symbol replacements. First, we replace "?" and "!" with periods. Then we replace all newlines with spaces and delete all symbols that are not in this list: "A-Za-z0-9äöüÄÖÜß. " (i.e. every symbol that is no letter, number, umlaut, "ß", period, or space). After that, multiple spaces are reduced to one. The next step is removing all spaces after a period and removing the very last period of the text (splitting at the periods would create an empty entry after the last period). Finally, the last steps are the replacements specified in the settings, i.e. dealing with numbers, "ß", umlauts, and converting all letters to uppercase, according to the users specifications.

Before generating anything, we initialize a pseudo-random number generator with the input seed.

3.3.4 Plaintext Generation

For each generation, the pseudo-random number generator is requested to return the next random value in the range of the sentence array. Consequently, each generation is dependent on the very first seed. This random index of the array of sentences returns a sentence that we use as start sentence. The next step is checking if one of the plaintexts generated yet starts with the exact same sentence. We continue to randomly choose a start sentence until no other plaintext starts with this sentence to make the plaintexts unique. After that, the following sentences in the array are added to the first sentence, until the current text length is reached. At that point, the plaintext string is cut to the current text length, if it is longer than that.

The text length is increased continuously during generation. How many plaintexts per text length are generated is calculated in the beginning (Equation 3.1). If the text length increase setting is set to 0, the text length remains the same for all plaintexts. One plaintext per length will be generated, if the minimum and maximum text lengths are equal or if the text length increase setting is set to a higher value than maximum minus minimum key length.

$$\#(plaintexts) = \frac{NumberOfTestRuns}{\left(\frac{MaxTextLength - MinTextLength + LengthIncrease}{LengthIncrease}\right)} \tag{3.1}$$

For statistical purposes, every text length is generated the same number of times. So if the number of test runs cannot be equally distributed among the text length between the minimum and maximum with the given text length increase, the last text lengths will be dropped. This is the case in Equation 3.2. The number of test runs is set to 100 and the plaintext lengths should be 25 to 150 with an increase in length of 5 between them.

$$\#(plaintexts) = \frac{100}{\left(\frac{150 - 25 + 5}{5}\right)} = 3.85 \tag{3.2}$$

In this case, we will generate four plaintexts with each text length, starting at the minimum length 25. So the maximum text length is going to be 120, not 125. In order to get four plaintexts up to the length of 125, 104 test runs would be necessary.

3.3.5 Key Generation

The key generation is split into three different types, to be able to satisfy both inexperienced users (or simple use cases) and many ciphers in CT2. The three types are natural language keys, random keys, and reverse regex keys. One important aspect to mention is that every key and every plaintext is only used once in one test vector pair. They are not used interchanged.

3.3.5.1 Natural Language Key Generation

The natural key generation works similarly to the plaintext generation, by using the sentence array and searching sentences of matching lengths. It uses the same pseudo-random number generator, the same array of sentences, and the same way to generate a start sentence. But instead of concatenating multiple sentences, we try to find a sentence of the requested length. If that fails for the whole array, a longer sentence is cut to the requested length. Before counting the length of a sentence, all spaces are removed if the user has specified that in the settings. Which length to look for is determined by the minimum and maximum key length settings. The algorithm starts at the start sentence and continues its search through the sentence array, checking every sentence length against the key length range. If the sentence length is in the range, if this length of key has not been generated the necessary number of times yet, and if the sentence has not been used yet, it is chosen for the current key and added to the key list. The number of keys per length is calculated as shown in Equation 3.3. If there are too few test runs, however, exactly one key will be generated per length. If the minimum and maximum key lengths are equal, all keys will have the same length.

$$\#(keys\ per\ ciphertext\ length) = \frac{NumberOfTestRuns}{MaxKeyLength - MinKeyLength} \tag{3.3}$$

If the array of sentences is searched once and there are still keys missing, longer keys are to search for and cut to the necessary lengths. After that, the key is transformed into a unique number key if the setting is set. This is done by starting with the first letter in the alphabet and assigning the 0 to it. Following the order of the alphabet, the keys are assigned continuously increasing numbers, which are all unique. Transforming the letter key “letter” leads to the numeric key “2 0 4 5 1 3”. The transformation is started at the first letter of the alphabet, the *e*, so the first *e* becomes the 0, the second e becomes the 1. The next letter in the alphabet is the *l*, which is assigned the 2 and so on. The numbers are separated by a white space, because numbers greater than 9 become two or more digits long and have to be separated from other numbers.

The last step of the natural language key separation is the same one as for every other key generation type: Adding the separator symbol between each key symbol.

3.3.5.2 Random Key Generation

The random key generation uses the pseudo-random number generator as well, but does not need the input text, sentence array, or a start sentence. The generator is used to generate the number of random numbers in the range of the used alphabet that the key length defines. The alphabet may be input directly through the alphabet input or chosen through the selected key format. The available formats are “letters”, “numbers”, “binary” ([01]), and “use input alphabet”. Before the generation starts, the alphabet is split at the spaces and put into an array. Using the “unique symbol usage” setting, results in removing the currently chosen alphabet symbol directly from the alphabet array. So, unique keys can only be as long as the alphabet, because otherwise repetition would be necessary. Generating number keys that use digits multiple times can be arbitrarily long, but in order to generate keys from unique numbers that are longer than 10, an input alphabet with more symbols than the digits 0 to 9 has to be specified. The other random keys are simply built by using the generated number in the range of the alphabet array, taking the alphabet symbol at that particular index, and not removing the symbol from the alphabet.

3.3.5.3 Reverse Regex Key Generation

The most complex key generation method is the reverse regex generation. The user is enabled to formulate a regex pattern for which a matching random string is generated. This allows more complex keys to be generated. A simple example is shown Listing 3.1.

```
[ab]{4,6} // [set of characters]{range of key lengths}
```

Listing 3.1: Example for a simple regex pattern

This regex pattern generates strings consisting of the letters *a* and *b* of lengths from 4 to 6, like “aaaa”, “ababab”, and “bbabb”. Any symbols outside of the brackets will be ignored by the generator. The full functionality of the reverse regex generator can be found in the documentation [27].

In order to generate the key lengths specified in the minimum and maximum key length settings, we have implemented an additional variable into the regex: *$length*. The *$length* variable is simply replaced by the length that are specified in the TVG settings before the actual reverse regex generation starts. Listing 3.2 visualizes how this variable is used.

```
[ab]{$length}
```

Listing 3.2: Example for a regex pattern with $length

Another feature that is only missing in the reverse regex generation, is that uses alphabet symbols once, to get unique keys. For that purpose, we have added a second additional variable: *$unique*. This *$unique* variable comes in the form as shown in Listing 3.3. It generates a key, based on the parameters specified between the round brackets. In between, the alphabet letters to be used are specified between the square brackets (A-Z), the length of the key between the curly brackets (3), and optionally the number of following symbols in the key (1), which are not separated by the given separator symbol in round brackets.

```
$unique([A-Z](1){3})
```

Listing 3.3: Example for a regex pattern with $unique

The inner pair of round brackets with bracket 1 in Listing 3.3 may also be omitted. The default value is 1, so the separator symbol is inserted between every single key symbol. With a separator of "," this would generate strings like "I,V,J", "A,B,C", and "M,X,U", without using any alphabet symbol twice.

The complete string in the format *$unique(...)* is then replaced with the generated key. For multiple *$unique* variables, the process is repeated until all variables have been replaced by keys. After the replacement, the regex pattern still works for the Xeger generation like in Listing 3.1. The reason for that is that the parts of the pattern that have been replaced through according random keys are not inside of any brackets and are, therefore, ignored by the reverse regex generation. The generation is launched after replacing all *$length* and *$unique* variables. The reverse regex generation is based on an external library, explained in Section 3.4.1.

3.4 TestVectorGenerator – Application

This section will provide some details about the reverse regex generator we have used and show a test vector generation for the Enigma in CT2 with a detailed description.

3.4.1 Xeger – Reverse Regex Generator

The reverse regex generation is realized through an external library, called "Xeger". It is a "Java library for generating random text from regular expressions" developed by Wilfred Springer in 2009 [26]. It was partly based on the automaton package from Anders Møller, developed since 2001 [1]. Xeger has been improved by Roberto Ramírez Vique since 2013 [22] amongst others and has been made available in C# through the project "Fare – [F]inite [A]utomata and [R]egular [E]xpressions" by Nikos Baxevanis in 2011 [19]. Fare directly depends on Xeger and the automaton package.

We have included Fare into the CT2 repository, so it can be used by all the other components as well.

3.4.2 Test Vector Generation in Practice

The TVG is able to generate pure test vectors without having to evaluate them directly. This section shows how the settings of the TVG have to be adjusted to get the desired results. In order to show a more complex example using the reverse regex key generation, we have chosen the Enigma cipher to explain the key generation in detail. We also display the CT2 template with the TVG (Figure 3.3) that generates these Enigma keys.

3.4.2.1 Reverse Regex Keys for the Enigma

The Enigma machine accepts keys of the following format: "V, I, III / 2, 5, 20 / PW, HF, OC, XU, GV, BN, TL, SD, KA, RM, JE, QI, YZ". We translated this format into a regex pattern that works with the TVG. This regex pattern is shown in Listing 3.4.

```
$unique([I|II|III|IV|V|VI|VII|VIII]{3}) / $unique([1-26]{3}) /
    $unique([A-Z](2){26})
```

Listing 3.4: Generation of Enigma keys

We had to resort to our *$unique* variable in order not to choose any rotor more than once or to occupy a plug twice. This pattern generates the three parts of an Enigma key, separated by " / " (mind the spaces). The first part are the three rotors, uniquely chosen from I to VIII. The second part are the three random starting positions from 1 to 26. The last part of the pattern generates pairs of letters which represent the plugboard connections between the letters. In this example, we generate pairs from all 26 letters, hence 13 pairs. Less pairs are also possible to generate with the TVG and accepted by the Enigma. The applied separator "symbol" is ", " (build by two characters), so a key looks like this: "V, I, III / 2, 5, 20 / PW, HF, OC, XU, GV, BN, TL, SD, KA, RM, JE, QI, YZ".

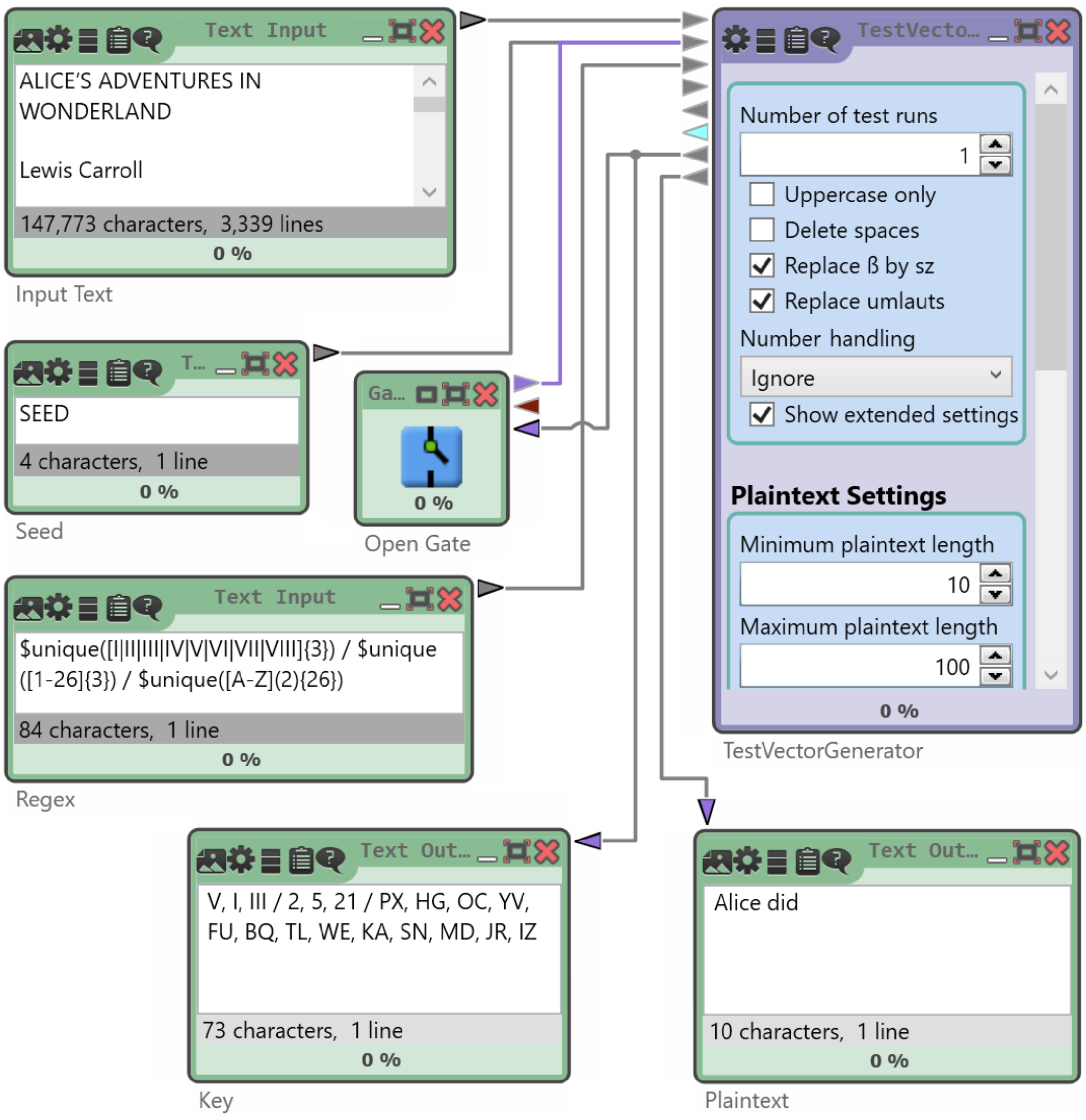
To show this complex key generation in practice, we have built a CT2 template (Figure 3.3) that generates Enigma keys with the pattern in Listing 3.4. We generate five different Enigma keys, using the input seed "SEED". Listing 3.5 contains the generated keys with these input parameters.

```
V, I, III / 2, 5, 21 / PX, HG, OC, YV, FU, BQ, TL, WE, KA, SN,
    MD, JR, IZ
IV, V, VIII / 6, 9, 2 / GJ, QS, AI, DB, RY, HV, UE, LF, ZK, NM,
    XO, PT, CW
I, III, VIII / 2, 15, 6 / SE, XY, LI, GZ, NF, VC, UW, AT, MK, BJ,
    QR, PO, DH
```

```
II, VIII, III / 19, 21, 3 / AC, VP, JQ, SN, WH, FZ, ML, DU, GE,
    KB, XT, OR, YI
IV, III, V / 17, 3, 21 / YF, VR, NC, OJ, XT, PD, IB, WZ, KE, HQ,
    SM, AL, UG
```

Listing 3.5: Enigma keys from the test vectors

Using the exact same input parameters, these keys are reproducible. The screenshot in Figure 3.3 displays how the different necessary components are connected.



ALICE'S ADVENTURES IN WONDERLAND

Lewis Carroll

147,773 characters, 3,339 lines
0 %

SEED
4 characters, 1 line
0 %

0 %

$unique([I|II|III|IV|V|VI|VII|VIII]{3}) / $unique([1-26]{3}) / $unique([A-Z](2){26})
84 characters, 1 line
0 %

Number of test runs: 1
[ ] Uppercase only
[ ] Delete spaces
[✓] Replace ß by sz
[✓] Replace umlauts
Number handling: Ignore
[✓] Show extended settings

Plaintext Settings
Minimum plaintext length: 10
Maximum plaintext length: 100
0 %

V, I, III / 2, 5, 21 / PX, HG, OC, YV, FU, BQ, TL, WE, KA, SN, MD, JR, IZ
73 characters, 1 line
0 %

Alice did
10 characters, 1 line
0 %

Figure 3.3: Screenshot of the Enigma test vector generation in CT2

The screenshot shows the inputs for the TVG, the gate that hands the last key over to the seed input as the new seed, the key output and plaintext output, and the TVG itself. This test vector generation can be done for all classical ciphers that are currently available in CT2[2], and also for some of the modern ones, by adjusting the settings accordingly.

3.4.2.2 Natural Language Key Generation in Practice

Another scenario is the generation of natural language keys of a length of 20 to 25 characters from the input text. We show the key output in Listing 3.6.

```
ITS NO BUSINESS OF MINE
HE PRONOUNCED IT ARRUM
THERES PLENTY OF ROOM
NOT THE SAME THING A BIT
OH YOU FOOLISH ALICE
PRESENTLY SHE BEGAN AGAIN
```

Listing 3.6: Natural language keys

The keys have the lengths of 20 to 25 and are all complete sentences. Using the input seed “NATURAL”, these keys can be reproduced.

All test vectors are reproducible by using the same inputs. The initial seed can be varied for different test vector sets.

3.4.2.3 Plaintext Generation in Practice

Using the CT2 template shown in Figure 3.3, we generate plaintexts that are 10 to 100 characters long. This shows the generation of real English sentences taken from the input text. We use the book “Alice in Wonderland” [14] as input text (consisting of about 150,000 characters)[3]. The input seed is set to “SEED”. The generated plaintexts are listed in Listing 3.7.

```
Alice did
He took me for his h
And shes such a capital one fo
Why should it muttered the Hatter Does Y
Why did they live at the bottom of a well The Dorm
his scaly friend replied There is another shore you know upo
said Alice who always took a great interest in questions of
   eating and
What was that inquired Alice Reeling and Writhing of course to
   begin with the Mo
Alice said to herself as well as she could for sneezing There was
   certainly too much of it
pleaded Alice And be quick about it added the Hatter or youll be
   asleep again before its done Once u
```

Listing 3.7: Generated plaintexts of length 10 to 100

[2]Ciphers like the CylinderCipher (CC) and the Solitaire Cipher require the splitting of different key parts in the CT2 workspace.

[3]Too short input texts result in an error and the abortion of the evaluation.

All generated plaintexts have lengths from 10 to 100 and are all readable (the first text ends with a space as 10^{th} character). Starting the whole generation again with the same parameters outputs the exact same plaintexts for that test vectors. This is the desired functioning of the TVG for this scenario.

3.5 CryptAnalysisAnalyzer – Concept and Design

The CAA is an evaluation tool for cryptanalysis procedures of (classical) ciphers. It expects the test vector input generated by the TVG and hands this input over to the *Cipher*. From there the encrypted test vector is connected to the cryptanalytic procedure, which returns its results to the CAA. In this way, the CAA gets all the necessary information to analyze and evaluate how the procedure has performed. Over the whole test vector series, average values are calculated and visualized, and GnuPlot graphs are prepared. These results can be exported by the user. They make cryptanalytic procedures comparable by multiple analytic metrics, on a very customizable basis. The following sections explain all that functionality in depth.

3.5.1 Inputs and Outputs

In order to receive test vector input from the TVG and from the CA and output data for the *Cipher* and CA components for the average value visualization and the GnuPlot graph, a lot of connectors are required. All the input connectors are listed in List 3.6.

List 3.6: CryptAnalysisAnalyzer input connectors

- input text [string]
- seed [string]
- key [string] (mandatory)
- plaintext [string] (mandatory)
- total keys [int] (mandatory)
- ciphertext [string] (mandatory)
- best key [string]
- best plaintext [string]
- evaluation container [EvaluationContainer]

Input text and seed are equal to the inputs of the TVG. They are optional and used for logging purposes. The key and plaintext inputs come from the TVG and are mandatory, as well as the total keys. The ciphertext comes from the *Cipher* that is used to encrypt the plaintext. Best key and plaintext as well as the *EvaluationContainer* come from the cryptanalytic component that is analyzed. They return all necessary information for each calculated test vector. We have developed the *EvaluationContainer* data type particularly for this purpose. It is documented in detail in Section 5.2 on Page 59.

The CAA's output connectors are listed in List 3.7

List 3.7: CryptAnalysisAnalyzer output connectors

- key [string]
- plaintext [string]
- minimal correct percentage [double]
- trigger next key [string]
- evaluation output [string]
- GnuPlot script [string]
- GnuPlot data [string]

Each key and plaintext test vector is handed over to the used *Cipher* component. If the minimal correct percentage is given to the CA component, the plaintext has to be provided, too, in order to have a result to check against. The next key is triggered repeatedly when a new *EvaluationContainer* has been detected. The key is triggered from the TVG by giving it a new seed (dependent on the original seed). The evaluation output visualizes the evaluation process and prints the final average values. The GnuPlot outputs provide a complete GnuPlot setup to draw the examples. GnuPlot is not included into CT2 and has to be used externally.

3.5.2 Settings

Most of the parameters for a test series are adjusted through the settings of the TVG. The CAA settings mostly control the GnuPlot output. These are the settings that belong to the evaluation itself:

List 3.8: CryptAnalysisAnalyzer evaluation settings

- minimal correct percentage [double]
- calculate runtime [boolean]

The minimal correct percentage is the minimal percentage that has to be correct for a plaintext to be treated as correct and is directly connected to the cryptanalytic component to evaluate. The runtime calculation can be enabled through a check box. If it is disabled, it will not be available in the evaluation and GnuPlot outputs.

The GnuPlot settings are presented in List 3.9.

List 3.9: CryptAnalysisAnalyzer GnuPlot settings

- X-axis [enumeration]
- Y-axis [enumeration]
- second Y-axis [enumeration]
- second Y-axis average [boolean]
- normalize Y-range factor [int]

The GnuPlot has at least two axes whose values can be chosen through the first two settings. The second Y-axis is a third axis for an additional graph in the same plot, also dependent on the chosen X-axis value. The Y-axis average can be enabled through a check box to get a straight line for the second Y-axis in the plot. The last setting does not manipulate any data. It is a setting to turn the attention to the main area of data points in the plot. It changes the initial GnuPlot range that is shown in the plot to the average value. But we do not calculate the standard average, we calculate a new average of all points that are within the given factor of the old average. All points outside of this margin will be dropped in the calculation of the new average value. One simple example for that would be: 9 points have values ranging from 2 to 6, but one has 80. This would give an average of about 11.6. The GnuPlot plot would show the range of 2 to 80, making it very hard to distinguish between 90% of the plotted data points. The user is now able to decide, whether he wants to see all data points in the plot, by setting the factor very high (e.g. 1000). But if the factor is on the default setting, which is 4, the data point with the value of 80 will be ignored in the initial plot. This is the case, because 80 is above 4 times the original average ($4 * 11.6 = 46.4$). We call it initial plot, because GnuPlot is able to be adjusted afterwards (through scrolling for example) so there is no data lost. In our example, the new average value will be about 4 (depending on the specific values), to which we adjust the initial plot. This yields a lot more presentable plots in our experience.

3.5.3 GnuPlot

The GnuPlot output is divided into two different string outputs. One is the script output that defines the specific settings for the GnuPlot to be drawn. The other one is a pure data output, providing the values for the plot (separated into multiple columns). The calculated average values are used in the generation of both outputs. In the GnuPlot script, the values and their names are used to generate settings like the range of the plot, the legend, and the GnuPlot data file columns. The data file simply holds the average values in different columns. These two files are put out from the CAA through the two GnuPlot outputs as strings. They have to be copied into files and used through GnuPlot. The naming of the files and the GnuPlot usage is described on top of each file. We define different file names by combining the selected values in abbreviation, like "Succ_PercDecr_Rest_PerCiphLen", which stands for "success, percent decrypted, and number of restarts per ciphertext length". This way, multiple files can be stored in the same folder without having to choose new proper names each time. The GnuPlot command to use the script is given in Listing 3.8. This command has to be entered into the GnuPlot command line, after navigating to the directory with the working copy.

```
load 'Succ_PercDecr_Rest_PerCiphLen.p'
```

Listing 3.8: Instruct GnuPlot to load a script file

Script files get the file ending ".p" while data file names end with ".dat". Listing 3.9 shows some excerpts of the GnuPlot script file, loaded in Listing 3.8.

```
set y2label "Restarts"
set y2range [0:650]
set title "Success, Percent Decrypted, and Restarts per
   Ciphertext Length\nfor 500 Restarts, using 3-grams and
   stopping at 80%"

plot    "Succ_PercDecr_Rest_PerCiphLen.dat" using 1:2 title
   'Success' with linespoints ls 1
```

Listing 3.9: GnuPlot script file excerpts

The commands “set label”, “set range”, and “set title” are in the generated plot. Label and range are set independently for each axis, which is the secondary Y-axis in this case.

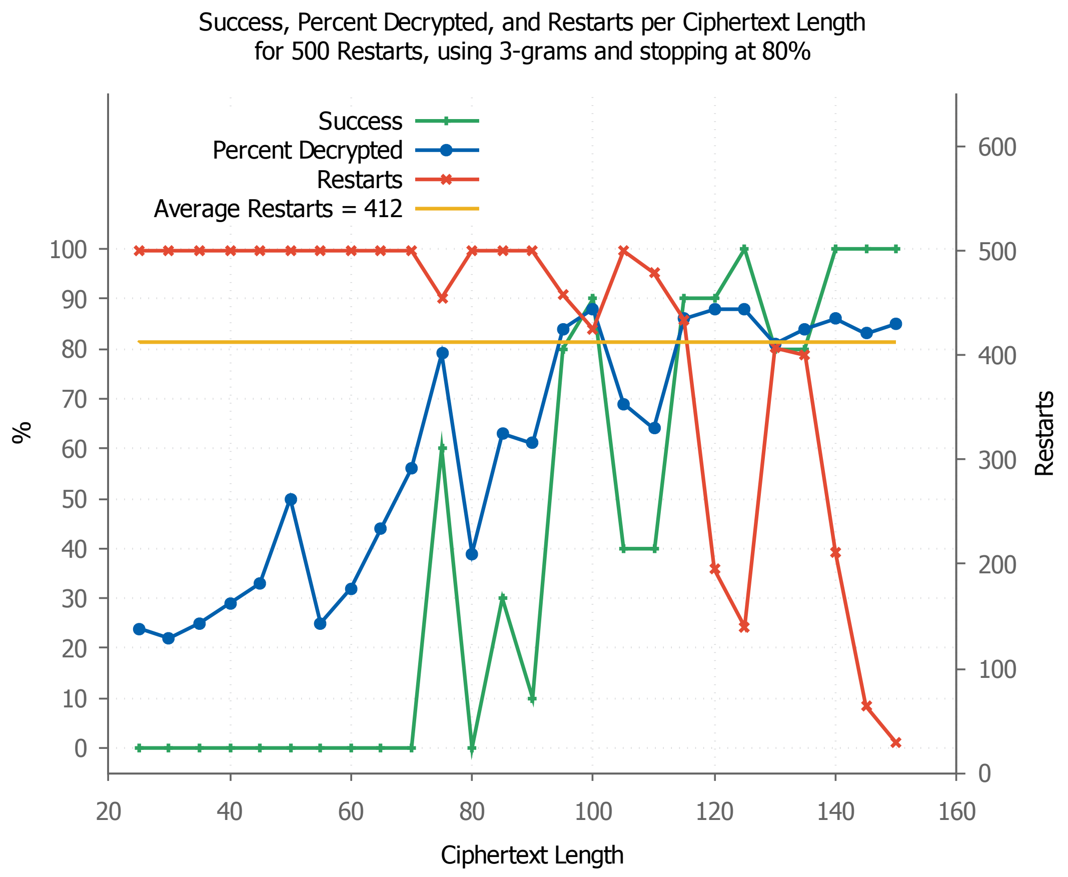


Figure 3.4: GnuPlot generated with the settings from Listing 3.9

The command “plot” takes values from the selected data source and plots them – here with the title “Success” as legend. The “1:2” selects column two in dependency of column one, so column two becomes the Y-axis value and column one the X-axis value. The line style (“ls”) number one is a combination of line color, type, and point symbols we created. Using different line styles makes the graphs more easily distinguishable.

Once the files have been saved and the script file has been loaded, GnuPlot draws a plot with the selected graphs in it. Figure 3.4 displays one exemplary GnuPlot plot. All the GnuPlot settings are changeable live while executing. This is especially interesting once the execution has come to an end. For the same test series and evaluation, GnuPlot outputs for all different combinations of values can be generated (one x-y combination at a time), without having to redo the whole testing process.

3.5.4 Meta Analysis Method

Table 3.1 visualizes which evaluation metrics are used by 9 scientific papers, published between 1995 and 2016. Legend and explanation can be found below.

	Paper indices								
	1	2	3	4	5	6	7	8	9
Algorithm runtime					√			√	√
Success probability	√		√	√		√	√	√	√
Correctly decrypted (%)	√		√			√	√	√	
Ciphertext length	√	√	√	√	√	√	√		
Plaintext length[4]									√
Necessary decryptions	√	√	√	√	√		√		√
Key space	√	√	√	√	√		√	√	√
Key length	√	√	√	√	√				
Cost function	√	√	√	√	√	√	√	√	√
Algorithm restarts							√		
Population size[5]						√		√	
Tabu set size[6]									

Table 3.1: Analysis of evaluation metrics of common cryptanalytic papers

Legend of the paper indices in Table 3.1:

1. Ciphertext-only cryptanalysis of Enigma [9]
2. Breaking Short Vigenère Ciphers [24]
3. Cryptanalysis of columnar transposition cipher with long keys [17]
4. Solving the Double Transposition Challenge with a Divide-and-Conquer Approach [15]
5. Breaking Short Playfair Ciphers with the Simulated Annealing Algorithm [7]
6. A parallel genetic algorithm for cryptanalysis of the polyalphabetic substitution cipher [6]
7. Efficient Cryptanalysis of Homophonic Substitution Ciphers [8]
8. Genetic Algorithms and Mathematical Programming to Crack the Spanish Strip Cipher [5]
9. Automated Known-Plaintext Cryptanalysis of Short Hagelin M-209 Messages [16]

[4]For known-plaintext attacks only
[5]For genetic algorithms only
[6]For Tabu search algorithms only

We examined the 9 papers of Table 3.1 to find out which evaluation metrics were analyzed by the authors. Only paper nine describes a known-plaintext attack, which explains why the plaintext length is only evaluated in this one case.

The first paper [9] describes a ciphertext-only cryptanalysis of the Enigma machine, recovering the different security measurements of the machine separately.

In the second paper [24], logic plays the major role. The key space is reduced in multiple steps by logically excluding impossible solutions and linguistic combinations. So its main focus is not the decryption itself but the prearrangements, which is why values like runtime, success, and decrypted percentage are not evaluated.

The papers three [17], four [15], seven [8], and nine [16] describe hill climbing attacks, although only paper seven evaluates the number of restarts. Paper three, four, and seven have more similarities among each other than paper nine. Paper four does not mention the percentages of the correctly decrypted ciphertexts by its algorithm while paper seven lacks the key length in its evaluation. All other evaluation metrics are the same.

Paper five [7] describes a simulated annealing algorithm to break short Playfair ciphers.

The papers six [6] and eight [5] describe genetic algorithms, both evaluating their population size. Paper eight also measures the algorithm runtime and calculates the key space, while paper six takes the ciphertext length into account. The rest of their metrics are equal.

The last paper [16] is different to the other hill climbing papers in the table, mostly because it describes a known-plaintext attack. It does not mention the correctly decrypted percentages or the key length. The number of restarts is not evaluated either. Instead of the ciphertext length, only the plaintext length is interesting and, therefore, evaluated. The runtimes for different scenarios are mentioned and the runtime is limited for each attack.

We could not find a single paper describing a Tabu search on classical ciphers, so the row is empty.

Metrics used in the papers that we cannot easily evaluate by the output of the CA CT2 component are cost function and key space. They can be disregarded at the moment for our automated evaluation[7]. Most papers evaluate success probability, correctly decrypted percentage, ciphertext length, necessary decryptions, and key length.

Algorithm restarts are a major factor for the performance of hill climbing algorithms; consequently, we have added it to our evaluation metrics. The population size is evaluated in both of the referenced papers dealing with genetic algorithms, so we have also added population size. We have also added the Tabu set size (in

[7]Setting an using these metrics could be automated, we mention that in Section 8.4.

case there is an according CT2 component in the future) and omitted the plaintext length, because we focused on ciphertext-only cryptanalysis.

3.6 CryptAnalysisAnalyzer – Application

This section provides multiple pseudo-code examples of the evaluation processes and explains some aspects in detail. Moreover, we show a screenshot of the complete evaluation setup in CT2 and explain the M-94 key generation.

3.6.1 The CryptAnalysisAnalyzer in Depth

The evaluation process in the CAA can be expressed through an algorithm as shown in List 3.10.

The three states are visualized in a colored flow diagram with the according components and actions in Figure 5.1 on Page 67. Figure 3.1 on Page 13 shows a more abstract visualization of the evaluation process.

The most important thing about using the CAA in three different states is to check that all necessary input connectors have been provided with new input values. The graphical programming language in CT2 will provide each value separately, causing the CAA component to execute the *Execute*-method. This means that we have to dismiss these changes until all necessary inputs provide the desired inputs.

Another important information is that each CT2 component will only start to process if all input values are provided with data. Hence, the CAA has to be supplied with default input values on all outputs that are not provided with data until its first execution. These values are the returning values of the *Cipher* and CA components. The CAA has to provide the first test vector to them without having any input for the best key and plaintext for example. That is why we put empty strings into the input values "ciphertext", "best key", and "best plaintext". The evaluation container input is fed with an empty *EvaluationContainer* by the "EvaluationContainerInput" component (see Section 3.6.2). When triggering the next key, the CAA has been provided with input values for all of its inputs. The empty input values are all overridden by actual values before collecting the data.

In the end, the CAA stays active and listens to possible changes in the GnuPlot settings. If these are changed, the GnuPlot outputs will be regenerated accordingly.

List 3.10: Evaluation process flow

```
State 1: Produce test data
  a) Wait for test vector input (disregard other input)
       i. Hand test vector over to Cipher and CA components
      ii. Count test vectors
     iii. Switch to state 2

State 2: Collect data
  a) Wait for new ciphertext, best key, best plaintext, and Evaluation-
     Container input; only proceed if all new, disregard other input
       i. Create ExtendedEvaluationContainer
      ii. Save container in dictionary with the ID as key
     iii. Visualize the data collection process through the evaluation
          output
      iv. Trigger next key and switch to state 1 if key count != total
          keys
       v. Break data collection loop and switch to state 3 if key count
          == total keys

State 3: Evaluate
  a) Calculate all evaluation metric values per ciphertext and plaintext
     (and runtime if activated)
  b) Divide values through the number of test runs to get the averages
  c) Generate evaluation output of the total average values, keys per
     length, and ciphertexts per length
  d) Generate GnuPlot script and data outputs from the detailed aver-
     age values according to current GnuPlot settings
  e) Wait for live changes of the GnuPlot settings
       i. Regenerate GnuPlot outputs on GnuPlot settings changes
```

3.6.2 EvaluationContainerInput

The *EvaluationContainerInput* is a CT2 component with the single purpose of generating an *EvaluationContainer* that can be used as an input in other components. The reason for that is that the CT2 data type *EvaluationContainer* has to be fed with an initial value when it is used as an input connector. The CAA uses the *EvaluationContainer* as an input, therefore it needs an initial value. For this purpose, the *EvaluationContainerInput* just has to generate an empty *EvaluationContainer*. Its functionality might be extended by a developer with more comprehensive needs.

3.6.3 Complete Evaluation Setup

The complete evaluation setup of the CCA is visualized in Figure 3.5.

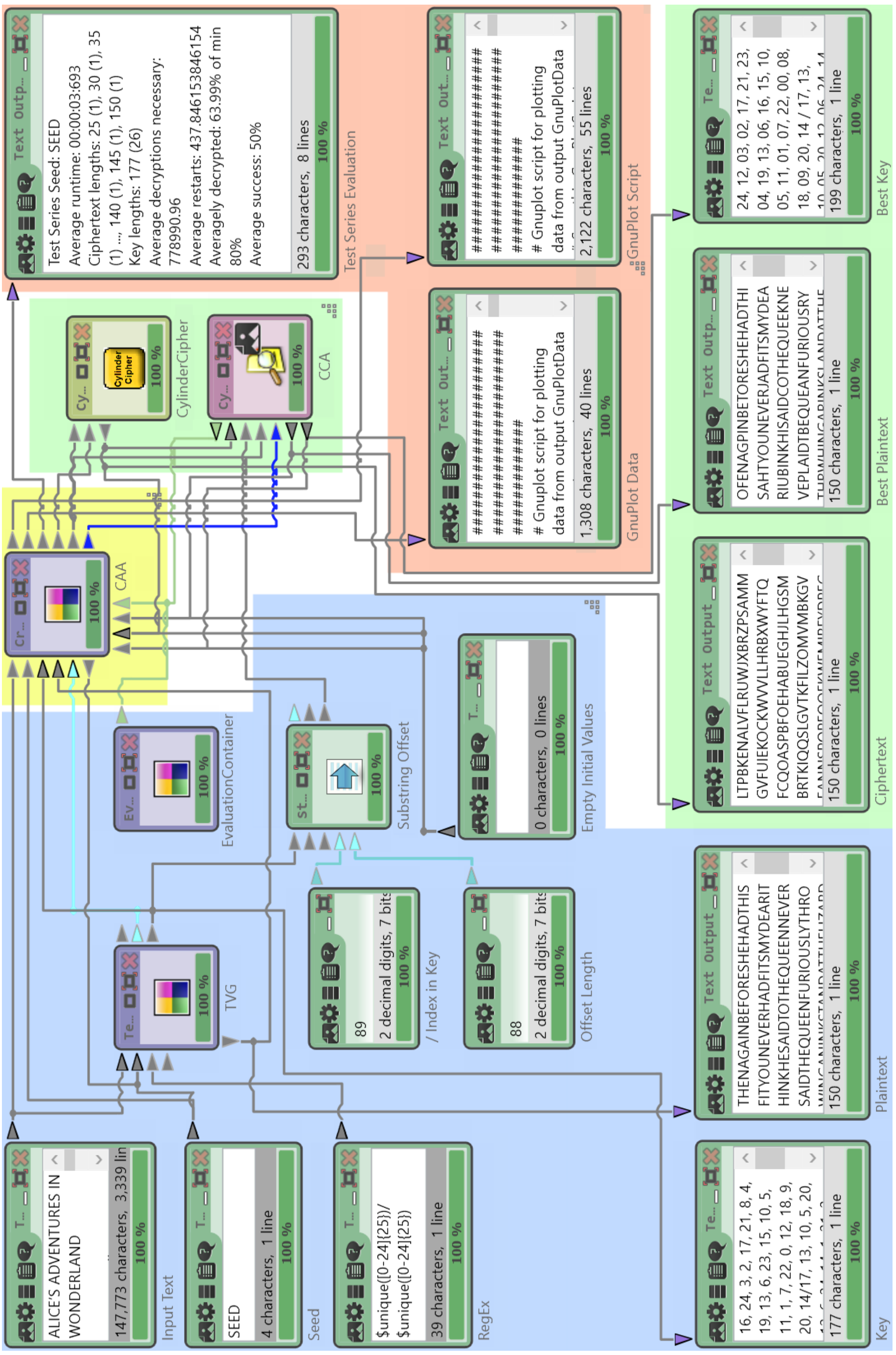
The screenshot reflects the complexity of the evaluation in CT2. The colors symbolize the three states of the CAA and follow the same color scheme as Figure 5.1 on Page 67 and our textual algorithm in List 3.10. State one is highlighted in blue, state two is highlighted in green, and state three is highlighted in red, while the CAA is highlighted in yellow (because all three states belong to it).

The regex pattern of this setup is shown in Listing 3.10. Following Chapter 5, other classical CA components of CT2 can be evaluated like that. After implementing the collection of the evaluation data in the analyzer component, the CylinderCipher and CCA components can directly be replaced by the ones to analyze. The offset calculation might be unnecessary and therefore omitted. Some similar adjustments might be necessary for other complex keys. The complete setup of the evaluation is explained in detail in Section 5.4 on Page 66. The regex pattern of this setup is shown in Listing 3.10. Following Chapter 5, other classical CA components of CT2 can be evaluated like that. After implementing the collection of the evaluation data in the analyzer component, the CylinderCipher and CCA components can directly be replaced by the ones to analyze. The offset calculation might be unnecessary and therefore omitted. Some similar adjustments might be necessary for other complex keys. The complete setup of the evaluation is explained in detail in Section 5.4 on Page 66.

```
$unique([0-24]{25})/$unique([0-24]{25})
```

Listing 3.10: Generation of M-94 keys

These keys in Listing 3.10 are specifically designed for the M-94, as their length is 25. The other cylinder cipher of the component ("Bazeries") works with 20 letter keys, as it has only 20 disks. The pattern generates 25 numbers from 0 to 24, each only used once. This is done twice and separated by a "/". The separator symbol we use is ", " (mind the space). This gives us keys like this one: "14, 1, 5, 2, 4, 19, 15, 22, 9, 7, 16, 3, 23, 20, 8, 18, 0, 13, 21, 11, 17, 6, 10, 12, 24/15, 13, 3, 8, 7, 9, 21, 17, 4, 11, 16, 6, 22, 10, 20, 18, 0, 1, 12, 5, 23, 19, 2, 14, 24".



Test Series Seed: SEED
Average runtime: 00:00:03:693
Ciphertext lengths: 25 (1), 30 (1), 35 (1) ..., 140 (1), 145 (1), 150 (1)
Key lengths: 177 (26)
Average decryptions necessary: 778990.96
Average restarts: 437.846153846154
Averagely decrypted: 63.99% of min 80%
Average success: 50%
293 characters, 8 lines 100 %

```
####################
####################
############
# Gnuplot script for plotting
data from output GnuPlotData
```
1,308 characters, 40 lines 100 %

```
####################
####################
############
# Gnuplot script for plotting
data from output GnuPlotData
```
2,122 characters, 55 lines 100 %

24, 12, 03, 02, 17, 21, 23, 04, 19, 13, 06, 16, 15, 10, 05, 11, 01, 07, 22, 00, 08, 18, 09, 20, 14 / 17, 13,
199 characters, 1 line 100 %

OFENAGPINBETORESHEHADTHI SAHTYOUNEVERJADFITSMYDEA RIUBINKHISAIDCOTHEQUEEKNE VEPLAIDTBEQUEANFURIOUSRY
150 characters, 1 line 100 %

LTPBKENALVFLRUWJXBRZPSAMM GVFUIEKOCKWVVLLHRBXWYFTQ FCQOASPBFOEHABUEGHJLHGSM BRTKIQQSLGVTKFILZOMVMBKGV
150 characters, 1 line 100 %

0 characters, 0 lines 100 %

89
2 decimal digits, 7 bits 100 %

88
2 decimal digits, 7 bits 100 %

THENAGAINBEFORESHEHADTHIS FITYOUNEVERHADFITSMYDEARIT HINKHESAIDTOTHEQUEENNEVER SAIDTHEQUEENFURIOUSLYTHRO
150 characters, 1 line 100 %

16, 24, 3, 2, 17, 21, 8, 4, 19, 13, 6, 23, 15, 10, 5, 11, 1, 7, 22, 0, 12, 18, 9, 20, 14/17, 13, 10, 5, 20,
177 characters, 1 line 100 %

ALICE'S ADVENTURES IN WONDERLAND
147,773 characters, 3,339 lin 100 %

SEED
4 characters, 1 line 100 %

$unique([0-24]{25})/
$unique([0-24]{25})
39 characters, 1 line 100 %

Figure 3.5: Screenshot of the CCA evaluation in CT2

4 Implementation

In this chapter, we explain the details of our implementation with colored listings of parts of the code. First, we show the connectors and settings of the TestVectorGenerator (TVG), followed by the plaintext and key generation algorithms. Moreover, we display the key aspects of the CryptAnalysisAnalyzer (CAA) evaluation and how the GnuPlot outputs are generated.

4.1 TestVectorGenerator

The TVG generation is designed to be as customizable as possible to be able to generate keys for many different scenarios. In order to achieve this, a lot of settings and extra inputs were necessary. All of the settings have to be displayed correctly and factored in the generation. This section goes more into detail about the implementation of these different aspects.

4.1.1 Inputs and Outputs

Each input and output connector is defined as a public method with a *PropertyInfo* in the line above (see Listing 4.2) in the TVG class. For each connector, we have implemented a private variable like shown in Listing 4.1.

```
private int _seedInput;
```

Listing 4.1: Private variable example

All global private variables are recognizable through their underscore in the beginning. The public connector methods function as setter and getter methods for these private variables. Internally in the TVG class, the private variables are used directly.

```
[PropertyInfo(Direction.InputData, "SeedInput", "SeedInput
   tooltip description", true)]
public string SeedInput {
    get { return this._seedInput.ToString(); }
    set {
        int seed = SHA1AsInt32(value);
        if (_seedInput != seed) {
            this._seedInput = seed;
            _newSeed = true;
        }
        OnPropertyChanged("SeedInput");
    }
}
```

Listing 4.2: Seed input method

Listing 4.2 shows the complete connector method of the seed input. It is the only method that does not simply store the given value if it is different to the old one and return the stored value. All the other input connectors do exactly that. In this method, the seed string input is hashed and then converted to a 32-bit integer. This old integer value is only overridden if the new one is different. The conversion is necessary, because the pseudo-random generator (*System.Random*) takes an integer as seed. In order to use any key format as a seed for the second test vector, the input connector has to accept string. This is realized through this conversion. The get method calls the *ToString*-method of the integer and returns the string.

The conversion from the string input seed to the integer seed is done by the method in Listing 4.3.

```
public static int SHA1AsInt32(string stringToHash) {
    using (var sha1 = new SHA1Managed()) {
        return BitConverter.ToInt32(sha1.ComputeHash(
           Encoding.UTF8.GetBytes(stringToHash)), 0);
    }
}
```

Listing 4.3: Hash of string converted to integer

All $UTF8$ characters are accepted and put into a byte array. This array is hashed by a $SHA1$ method. The hash algorithm $SHA1$ was chosen because its hashes are distributed randomly very evenly. There are other hash algorithms which provide that, but we just chose one that is easy to use in C#. The last step before returning the integer value is converting the hash of type byte array into a 32-bit integer. This is realized through Microsoft's *BitConverter* method.

Another important line of code is the *OnPropertyChanged* call once an output connector variable is updated. Listing 4.4 shows the *OnPropertyChanged* call that has to be called after changing the *PlaintextOutput* for the actual connector in CT2 to update this value.

```
OnPropertyChanged("PlaintextOutput");
```

Listing 4.4: OnPropertyChanged call

The *OnPropertyChanged* method is displayed in Listing 4.5.

```
private void OnPropertyChanged(string propertyName) {
    EventsHelper.PropertyChanged(PropertyChanged, this,
       propertyName);
}
```

Listing 4.5: OnPropertyChanged method

The *PropertyChanged* event (inside the brackets) is defined as a *PropertyChangedEventHandler* event. It is fired through the *PropertyChanged* method in Microsoft's *EventHelper* class like shown in Listing 4.5.

4.1.2 Settings

The TVG settings are defined in the *TestVectorGeneratorSettings* class. Every setting is defined like shown in Listing 4.6 and corresponds to a private variable like the one in Listing 4.7.

```
private int _numberOfTestRuns = 1;
```

Listing 4.6: Private settings variable example

Most of the variables are directly initialized with a default value. This default value is shown in the component's settings in CT2, because of the binding to the setting.

```
[TaskPane("Number of Test Runs",
   "NumberOfTestRunsTooltipCaption", null,
   generalPaneIndex, false, ControlType.NumericUpDown,
   ValidationType.RangeInteger, 1, Int32.MaxValue)]
public int NumberOfTestRuns {
    get { return _numberOfTestRuns; }
    set {
        if (_numberOfTestRuns != value) {
            _numberOfTestRuns = value;
            OnPropertyChanged("NumberOfTestRuns");
        }
    }
}
```

Listing 4.7: Number of test runs setting

The *TaskPane* keyword transforms the public method that follows this line into a setting of this CT2 component. Followed by the name and the tooltip there are the group of the setting, its index in the settings list, if it is changeable live while executing, the type of setting, the range of valid values, the minimum value, and the maximum value. The setting shown in Listing 4.7 is not changeable live, is displayed as a numeric box with up and down keys, has the valid range of an integer, has the minimum value one, and the maximum value of an integer. The method functions as a getter and setter of the private variable *_numberOfTestRuns* and calls the *OnPropertyChanged* method on changes.

We divided the settings into the regions “General TaskPane Settings”, “Plaintext TaskPane Settings”, and “Key TaskPane Settings”. This is helpful to navigate through the vast amount of settings (18 in total). We have explained the “show extended settings” option in Section 3.3.2. This and the updates for the visibility of the period replacer input and the key format input are updated via separate methods.

4.1.3 Plaintext Generation

The first thing we check in the plaintext generation is the number of generated plaintexts in the plaintext list against the number of test runs. If they are equal, the generation will not be executed, to prohibit a loop of generating too many test vectors.

The next step is the random selection of a starting sentence. Listing 4.8 shows a code sequence of that.

```
_startSentence = _rand.Next(0, _inputArray.Length);
```

Listing 4.8: Randomly selecting new starting sentence

The *_rand* variable is defined as a *System.Random*, initialized with the seed. We randomly pick an index of the input array which contains one sentence of the input text per array value. As long as this sentence matches the beginning of any plaintext in the list, we generate a new sentence. We also prevent an infinite loop in our code (not in the example).

Listing 4.9 visualizes the generation of the plaintext after choosing a start sentence (the length increasing of the plaintexts is not shown).

```
for (int i = _startSentence; i != _startSentence - 1; i = i
   == _inputArray.Length - 1 ? 0 : i + 1) {
    _plaintextOutput = _plaintextOutput +
       replaceSpaces(replacePeriods(_inputArray[i]));
    if (_plaintextOutput.Length >= _currentTextLength) {
        string finalPlaintext =
           _plaintextOutput.Substring(0,
           _currentTextLength);
        _plaintextList.Add(finalPlaintext);
        _plaintextOutput = finalPlaintext;
        // [...]
        break;
    }
}
```

Listing 4.9: Plaintext generation

The for loop runs up to the last index of the input array and starts at zero afterwards. The plaintext is appended by the next sentence until it is at least as long as the length currently searched for. Each sentence is processed according to the settings, regarding the periods and spaces. This has to be done before appending it and counting its symbols, as that process changes its length. Once the necessary length is reached, the plaintext is cut to the specific length, added to the plaintext list, and stored in the private plaintext output variable. Not shown in the example is the increase of the counter value, for the generated texts of that length and the following steps. After that, the value of the counter is compared to the desired number of texts with that length. If that number is reached, the counter is reset to zero and the text length currently searched for is increased by the value of the text length increase setting.

4.1.4 Natural Language Key Generation

The natural language key generation is based on randomly selecting a start sentence of the requested key length or cutting a longer one to that length. The start sentence is generated the same way as for the plaintext generation. The next step is initializing a *ConcurrentDictionary* with two integer dimensions. The first integer (also called the key of the dictionary) is the key length, and the second one stores the occurrences of keys of that length. We chose the *ConcurrentDictionary*, because it allows to add or update the value.

This kind of search for a sentence as key is divided into two phases:

1. search all sentences for the exact requested length
2. if nothing found, search all sentences for a longer length

We will limit this section to the first part (represented in Listing 4.10), as the second one is too complex to show.

```
if (sentenceLength >= _settings.MinKeyLength &&
   sentenceLength <= _settings.MaxKeyLength &&
   lengthOccurrences < _settings.KeysPerLength &&
   !_keyList.Contains(sentence)) {
    _keyList.Add(sentence);
    _occurrences.AddOrUpdate(sentenceLength, 1, (id, count)
       => count + 1);

    // if letters should be replaced by numbers, do so
    if (_settings.KeyFormatNaturalLanguage ==
       FormatType.numbers) {
        if (_settings.UniqueSymbolUsage)
            sentence = ConvertToUniqueNumericKey(sentence);
        else
            sentence = ConvertToNumericKey(sentence);
    } else {
        sentence = AddSeparator(sentence);
    }
    _keyOutput = sentence;
    return;
}
```

Listing 4.10: Natural key generation algorithm

The complete code in Listing 4.10 runs inside a loop over the input array. The if condition checks the current start sentence to fit the range between the minimum and maximum key length values, specified in the settings. Additionally, we have to check that the number of occurrences of that length has not reached the requested keys per length and that the key list does not contain the sentence already. If all of that is true, the sentence is added to the key list and the occurrences for this particular key length are incremented by one. The next step is the conversion to a numeric key, if specified in the settings, or simply adding the separator symbol to the sentence. In the end, the final sentence is stored in the private key output variable and the method is exited.

4.1.5 Random Key Generation

The random key generation is completely separate from the input text. Based on the pseudo-random number generator, we generate the symbols of an alphabet, which may be input through the alphabet input or chosen from a drop-down menu in the settings. The symbols are strings and can be chosen arbitrarily, even multiple characters are possible. The number generator is fed with the ranges of the alphabet and generates the specified number of alphabet string symbols. This generation can be with multiple occurrences of each symbol or only one. For the unique occurrence of symbols, the alphabet is shortened by the current symbol, which will not be generated again in this key.

The different built-in modes are Roman alphabet, digits, binary, unique Roman alphabet, unique digits, and the given alphabet through the alphabet input (the alphabet symbols have to be separated by spaces). The gist of what we have just described is summarized in Listing 4.11.

```
if (_settings.KeyFormatRandom == FormatType.letters) {
    if (_settings.UppercaseOnly)
```

```
        alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".Select(c => c.ToString()).ToList();
// [...]
}
// [...]
else {
    alphabet = _alphabetInput.Split(' ').ToList();
}
// [...]
int length = _settings.MinKeyLength + _lastKeyLengthIndex / _settings.KeysPerLength;
int separatorRepeat = 1;
// [...]
string randomKey = "";

for (int j = 0; j < length; j++) {
    int i = _rand.Next(0, alphabet.Count);

    string symbol = alphabet.ElementAt(i);
    if (_settings.UniqueSymbolUsage)
        alphabet.RemoveAt(i);

    if (randomKey == "")
        randomKey = symbol;
    else if (j % separatorRepeat == 0)
        randomKey = randomKey + _settings.Separator + symbol;
    else
        randomKey = randomKey + symbol;
}

return randomKey;
```

Listing 4.11: Random key generation

The first part shows how the string list *alphabet* is initialized. The two given examples are the Roman alphabet and the input alphabet, both split at the spaces and converted to a list. The key length is calculated by dividing the last key index by the number of keys to generate per length plus the minimum key length. We cut off the floating point result at the point to get an integer. The integer will increase when the last key index becomes a multiple of the number of keys per length. This way, the number of keys per length will be generated exactly. The separator symbol is repeated by the default value, one. The new string *randomKey* is set to the first alphabet symbol that is generated. The symbol is removed from the alphabet list. In the following rounds of the loop, the next symbol is appended to the key, separated by the separator symbol at the defined frequency.

Once the key reaches the requested length, it is returned.

4.1.6 Reverse Regex Key Generation

The usage of the reverse regex class *Fare* (C#), which wraps *Xeger* (Java) is very simple.

The regex input pattern has to be preceded with an “@”. Pattern and the pseudo-random number generator are used to initialize a new *xeger* variable. This variable contains the method *Generate* which returns the finished key, generated to match the regex pattern. Afterwards we have to check if this key has already been generated and continue the generation until it is new. In the end, the key is added to the key list and stored in the private key output variable.

In the omitted part of the code, we check for our own variables *$length* and *$unique* to replace them by values. *$length* is replaced by the current key length, which is generated the same way as in line 9 of Listing 4.11. The *$unique* variable contains parameters and has to be parsed. After determining the alphabet, the key length, and the separator repeat, we call the same method to generate this part of the key as for the random key generation. Each *$unique* variable is completely replaced by the according generated string, before starting the reverse regex generation.

4.2 CryptAnalysisAnalyzer

The CAA has a lot of connectors, whose functionality we have explained in Section 3.3.1. Their implementation is very similar to the TVG connectors (see Section 4.1.1). Accordingly, this section focuses on the evaluation.

4.2.1 Settings

Most of the settings look very similar to the TVG settings. For the purpose of updating the GnuPlot outputs live through changing a setting, we have to execute methods from the CAA class inside the settings class. Listing 4.12 shows the drop-down menu setting of the X-axis.

```
[TaskPane("X-axis", "Values to show on the X-axis",
   "GnuPlot", gnuPlotPaneIndex, true, ControlType.ComboBox,
   new String[] {
            "Ciphertext length", "Key length", "Runtime"})]
public XAxisPlot XAxis {
    get { return this._xAxis; }
    set {
        if (value != _xAxis) {
            this._xAxis = value;
            OnPropertyChanged("XAxis");

            if (_CAA.GnuPlotScriptOutput != null) {
                _CAA.SetGnuPlotVariables();
                _CAA.GenerateGnuPlotDataOutput();
                _CAA.GenerateGnuPlotScriptOutput();
                _CAA.RefreshEvaluationOutputs();
            }
        }
    }
}
```

Listing 4.12: X-axis setting

The *TaskPane* command makes a CT2 setting out of the public *XAxis* method. The *XAxisPlot* enumeration type provides the possibilities *ciphertextLength*, *keyLength*, and *runtime*. Apart from setting the local variable and firing the *OnPropertyChanged*, the update of the *XAxis* setting also executes multiple CAA methods. First, the existence of a GnuPlot script output is checked, to prohibit the execution of these methods before any output is available. If there is an output, all GnuPlot variables are updated, the data and script outputs for GnuPlot are regenerated, and the *EvaluationOutput* is refreshed.

In order to call methods in the CAA class, we need its current running instance in the settings. We get it there by using the constructor of the settings, at their point of initialization of the CAA. We use this CAA instance variable to refresh the outputs of the CAA when changes occur in the CAA GnuPlot settings.

4.2.2 Meta Analysis

The Meta Analysis is the complete process of collecting the data from the different CT2 components, evaluating it, and returning useful formatted results. The three main states in which the CAA operates are: "Distributing Test Vectors", "Collecting Data", and "Evaluation" (which includes the calculation and output). The next sections will give insights in the implementation of the different states.

4.2.2.1 State 1: Distributing Test Vectors

The functionality of the first state is more rudimentary. The test vectors of the TVG simply have to be distributed to the *Cipher* and CA components. Listing 4.13 presents how we have implemented that.

```
// If both plaintext and key are new, send them to the
   outputs
if (_newKey && _newPlaintext) {
    // consume new values
    _newKey = false;
    // [...]
  _keyCount++;
    // [...]
  // Send the plaintext and key (and min correct
     percentage) to the encryption method
  OnPropertyChanged("MinimalCorrectPercentage");
  PlaintextOutput = PlaintextInput;
  KeyOutput = KeyInput;
}
```

Listing 4.13: Distributing the test vectors

The plaintext, key, minimal necessary percentage, and the evaluation values are handed to the output connectors, once the plaintext and key are neither null or zero, nor equal to the old output values. The key counter is increased and the progress visualized through the *EvaluationOutput* (not shown).

4.2.2.2 State 2: Collecting Data

Once all evaluation data for one test run has reached the CAA, the *CollectEvaluationData* method is started. The complete condition for that can be taken from Listing 4.14.

```
// Wait for the analysis method to send evaluation data. If
   the evaluation input is set, together with the best key
   and plaintext, collect the evaluation data
else if (_newEvaluation && _newBestKey && _newBestPlaintext
   && _newCiphertext && _keyCount <= _totalKeysInput &&
   BestKeyInput != " " && BestPlaintextInput != " ") {
    // consume new values
    _newEvaluation = false;
    // [...]
    // gather all available evaluation data
    CollectEvaluationData();

    // trigger next key if key count is less than total
       keys...
    if (_totalKeysInput > 0 &&
        _keyCount < _totalKeysInput) {
        TriggerNextKey = KeyInput;
        OnPropertyChanged("TriggerNextKey");
    } else {
        // ...evaluate if not
        // [...]
        Evaluate();
        RefreshEvaluationOutputs();
    }
}
```

Listing 4.14: Checking if all variables are set

To know whether all necessary information has reached our component, we check the following key points:

- Is there an evaluation input that has a value set?
- Is that evaluation input different from the last one or the first evaluation input (is the last evaluation input null)?
- Is the current key smaller or equal to the total keys?
- Are the best plaintext and key values set?
- Are the best plaintext and key values not just one white space?

The last point became necessary, because CT2 was not accepting empty strings as initial value for the best plaintext and key, so we had to put at least one white space into the string. Once it is overridden with an actual value and all the other conditions are met, the current evaluation input is saved as the last input for the next iteration. We omitted the output generation in this segment of the code, as it is too long and not really relevant. Then the *CollectEvaluationData* method is called, which archives all data of each test run to prepare for the calculations in state 3. Listing 4.15 contains the shortened method's code.

The last if conditions decide whether this is the last key to evaluate and start the evaluation (and refresh the evaluation outputs), or it triggers the next key by setting the *TriggerNextKey* output to the current key.

We omitted one aspect in Listing 4.14. That is the intermediate generation of GnuPlot outputs (which includes the complete evaluation calculation beforehand). Currently, we trigger the generation on every ciphertext length update. This gives the user the possibility to view the results up to that point which is especially interesting for longer testing scenarios.

```
public void CollectEvaluationData() {
  // [...]
    double percentCorrect =
       _bestPlaintextInput.CalculateSimilarity(
       _plaintextInput) * 100;
    bool success = percentCorrect >=
       _settings.CorrectPercentage ? true : false;

    ExtendedEvaluationContainer testRun = new
       ExtendedEvaluationContainer(_evaluationInput,
        /*...*/);
    _testRuns.Add(_evaluationInput.GetID(), testRun);
    _evaluationCount++;
}
```

Listing 4.15: Collecting the evaluation data

The first step is calculating the percentage, to which the best plaintext matches the original plaintext. If this percentage is at least as high as the minimum necessary percentage specified in the settings, the *success* variable is assigned with the value *true*. After that, an extended *EvaluationContainer* is initialized with all the values of the current test run. Aside from the evaluation input container, these are the initial seed, key count, original key, original plaintext, the ciphertext, best key, best plaintext, minimum percentage, the current correct percentage, and the success. This container is added to the test runs dictionary under the unique id as key. The last steps are incrementing the evaluation counter and resetting the evaluation inputs.

4.2.2.3 State 3: Evaluation

Evaluating the overall average values of the calculation would be sufficient for the simple evaluation output. Looking into the details of how the examined algorithm performs under certain conditions, however, is much more complex. For the purpose of evaluating the average values for each evaluation metric we provide in relation to the three base values ciphertext length, key length, and runtime; all these related values have to be gathered separately. Our implementation of that involves one dictionary per relation (e.g. the relation between success and key length are gathered in the dictionary *_successPerKeyLength*). The first step of counting the metric values for each base value is visualized in Listing 4.16.

```
// counting and sorting the data into the dictionaries
foreach (KeyValuePair<int, ExtendedEvaluationContainer>
   entry in _testRuns) {
    // current test run values
    ExtendedEvaluationContainer testRun = entry.Value;
    int keyLength = testRun.GetKey().Length;
    int ciphertextLength = testRun.GetCiphertext().Length;
    int currentSuccess = 0;
    if (testRun.GetSuccessfull()) currentSuccess = 1;
    // [...]
    // count the successful runs
    if (testRun.GetSuccessfull())
        _successCount++;
    DictionaryExtention.AddOrIncrement<int>(
       _successPerKeyLength, keyLength, currentSuccess);
    // [...]
    // count the overall decryptions and decrypted
       percentages
    _decryptedCount += currentlyDecrypted;
    _decryptionsCount += decryptions;

    // count the decryptions and decrypted percentages per key and ciphertext lengths
    DictionaryExtention.AddOrIncrement<int>(_percentDecryptedPerKeyLength, keyLength, currentlyDecrypted);
    // [...]
    // update key value dictionaries: keyLengths and ciphertextLengths
    DictionaryExtention.AddOrIncrement(_keyLengths, keyLength, 1);
    // [...]
}
```

Listing 4.16: Counting evaluation values

For each test run in the *_testRuns* dictionary, we add every contained metric value to the current base value of the according dictionary. Also the three base values and their occurrences per length or amount are gathered in three according dictionaries.

We have developed the class *DictionaryExtention* containing multiple *AddOrIncrement* and divide methods for dictionaries. The usage of it is visualized in Listing 4.17.

```
DictionaryExtention.AddOrIncrement<K>(this Dictionary<K, int> dict, K key, int newValue)
DictionaryExtention.DivideAndRound<K>(this Dictionary<K, int> dict, K key, int divide, int round)
```

Listing 4.17: DictionaryExtention usage

The *AddOrIncrement* method takes one dictionary with an arbitrary key and an integer as value, a key of the same type, and a new integer value. If the given dictionary does not have an entry for that key, it is created and the *newValue* is assigned as value. If the key exists, however, its value is incremented by the *newValue*. The *DivideAndRound* method takes the same kind of dictionary and key, but also the integer to divide by and the integer to round to afterwards. This method also exists especially for percentages, multiplying the result with 100. The method without rounding (*Divide*) leaves out the *round* integer parameter.

All methods are also implemented for the data type *double*.

Calculating Averages

After all test runs have been processed and all values have been added to the according dictionaries, the next step is calculating the averages of those values separately. This is done by dividing the sum by their number of elements, as shown in Listing 4.18.

```
// calculate the overall average values
_averagePercentDecrypted =
   Math.Round((double)_decryptedCount / _testRuns.Count, 2);
// [...]
if (!_noRuntime) {
    // calculate the overall average values
    _averageRuntime = _runtimeCount / _testRuns.Count;
    // [...]
    _sortedRuntimes = from entry in _runtimes orderby
        entry.Key ascending select entry;
}
// [...]
// if the current key length count can be retrieved,
   calculate the average values
foreach (var pair in _keyLengths) {
    int keyLength = pair.Key;
    int count = pair.Value;

    // if the count is greater 1, we have to divide through
        count to get the average
    if (count > 0) {
        // calculate the detailed average values
        DictionaryExtention.DivideAndRoundPercent<int>(
            _successPerKeyLength, keyLength, count, 2);
        DictionaryExtention.Divide<int>(
            _decryptionsPerKeyLength, keyLength, count);
        // [...]
        if (!_noRestarts)
            DictionaryExtention.Divide<int>(
                _restartsPerKeyLength, keyLength, count);
    }
}
// [...]
if (_keyCount == TotalKeysInput)
    BuildEvaluationOutputString();
SetGnuPlotVariables();
GenerateGnuPlotDataOutput();
GenerateGnuPlotScriptOutput();
```

Listing 4.18: Calculating average values

The overall average percentage of the best plaintext matching the original plaintext is calculated by dividing the sum of all percentages through the number of percentages. We also round the value to two decimal places. This is also done for the runtime, which has to be sorted additionally, to draw a readable GnuPlot graph in the end. For each of the three base values, all sum values have to be divided as well, done in a *foreach* loop. Listing 4.18 only contains part of the loop of the key length calculation. The ciphertext length and runtime averages are calculated in similar loops. Some values are rounded to two decimal places to make the GnuPlot data file more readable in the end. Because of the intermediate generation of the GnuPlot outputs, we have to check whether the evaluation is already done before generating the evaluation output string. Lastly, the GnuPlot output generation methods are triggered.

Generating Evaluation Output

At this point, the evaluation is completed and all the outputs have to be formatted. Listing 4.19 is a snippet of the *BuildEvaluationOutputString* method, representing the most important parts.

```
public void BuildEvaluationOutputString() {
    // build the average runtime string
    string averageRuntimeString = "";
    if (!_noRuntime)
        averageRuntimeString = new
           DateTime(TimeSpan.FromMilliseconds(
           _averageRuntime).Ticks).ToString("HH:mm:ss:FFFF");
    // [...]
    // build the complete displayed evaluation output string
    _evaluationOutput = "";
    if (!_noRuntime)
        _evaluationOutput += "Average runtime: " +
           averageRuntimeString + "\r";
    // [...]
    _evaluationOutput += "Averagely decrypted: " +
       _averagePercentDecrypted + "% of min " +
       _settings.CorrectPercentage + "%\r";
}
```

Listing 4.19: Generating evaluation output

If the runtime is enabled, we build a new formatted string using the *DateTime* type. The complete evaluation output string is compiled from all the overall average values. The number of keys and ciphertexts per length is formatted beforehand (not shown).

Generating GnuPlot Data

The GnuPlot file has to be generated in one complete string and handed to the output container. The string variable is named *_gnuPlotDataOutput* and filled with all values line by line (see Listing 4.20).

```
public void GenerateGnuPlotDataOutput() {
    // generate the GnuPlot data output string
    _gnuPlotDataOutput = "##### [...] #####" + NewLine;
    _gnuPlotDataOutput += "# Gnuplot script for plotting
       data from output GnuPlotData" + NewLine;
    // [...]
    if (_settings.XAxis == XAxisPlot.ciphertextLength)
        AddCiphertextLengthValues();
    else if (_settings.XAxis == XAxisPlot.keyLength)
        AddKeyLengthValues();
    // [...]
}
```

Listing 4.20: Generating GnuPlot data output

At the top of both GnuPlot files, we include a header to explain how to use them. The actual appending of data is done in separate methods, according to the current GnuPlot settings.

For each entry in the *_ciphertextLengths* dictionary, all other chosen (in the settings) values are added in the same line, but in a different column. The columns are separated by multiple tab symbols (\t). The values have to be retrieved from the respective dictionary. A warning is printed on failure, the value is added on success.

The appending of all other values looks very similar, also for the other two base values key length and runtime.

Generating GnuPlot Script

To have the GnuPlot plots designed in a legitimate way and look useable directly, we put effort and testing into the GnuPlot settings, defined in the GnuPlot script. Apart from the different line styles and colors, we also adjusted the x and y ranges of the plot, to show the relevant part of the plot. We have shortly summarized the *GenerateGnuPlotScriptOutput* method in Listing 4.21.

```
public void GenerateGnuPlotScriptOutput() {
    // [..]
    // # second y-Axis settings
      // [...]
      _gnuPlotScriptOutput += "set y2label \"" + _val3 +
         "\"" + NewLine;

        // calculate normalized average Y-values
        _normalizedAverageYValues =
           CalculateNormalizedAverage(_yValuesArray,
           _settings.NormalizingFactor);
        int min = CalculateMinValue(_lowestYValue,
           _normalizedAverageYValues, "y2");
        int max = CalculateMaxValue(_lowestYValue,
           _highestYValue, _normalizedAverageYValues, "y2");
    // [...]
        if (min != max)
            _gnuPlotScriptOutput += "set y2range [" + min +
               ":" + max + "]" + NewLine;
    // [...]
    // # plotting
    // [...]
    _gnuPlotScriptOutput += "# plotting" + NewLine;
    _gnuPlotScriptOutput += "plot    \"" + _evalMethod +
       ".dat\" using 1:2 title '" + _val1 + "' with
       linespoints ls " + style;
    // [...]
}
```

Listing 4.21: Generating GnuPlot script output

The GnuPlot script is categorized into the areas “Header”, “General settings”, “Style settings”, “Plot settings”, “X-axis settings”, “Y-axis settings”, “Second x-axis settings”, and “Plotting”. The header is very similar to the data file header. The general settings remove old labels and settings, the style settings contain all the defined lines styles. The plot settings contain information like the title and the border of the plot. In the axis settings the axis labels, ranges, and scales are defined for each axis. The plotting settings contain the plotting commands, specifying the file name, the data column, the title, and the line style.

The *CalculateNormalizedAverage* method counts all values that are higher than the next value by the factor specified in the settings. Their value is subtracted from the overall sum of all values and the new average is calculated (we call it the normalized average).

We calculate the minimum and maximum values of the range to show in the plot based on the normalized average of all values, moving the emphasis of the plot towards the vast majority of all values. Through this optimization, very high values are outside of the default range of the plot, and can be accessed through scrolling in GnuPlot. The most important aspect of this is that all the other values become much more distinguishable in the plot. One example that happened during testing is that one or two values of the decryptions per ciphertext length exceeded the average value of the others by a factor of more than 10. This resulted in a squeezed plot with all the other values nearly indistinguishable close to each other.

This was completely resolved through this optimization. Listing 4.22 shows the maximum calculation.

```
public int CalculateMaxValue(double lowest, double highest,
   double avg, string axis) {
    int max = (int)highest;
    // [...]
    if (_settings.Y2Axis == Y2AxisPlot.decryptions ||
       (_settings.Y2Axis == Y2AxisPlot.runtime &&
       axis.Equals("y2")) || axis.Equals("x")) {
        // calculate distances to average
        int lowestToMean = (int)avg - (int)lowest;
        int highestToMean = (int)highest - (int)avg;

        if (highestToMean > lowestToMean * 2)
            max = (int)(avg + lowestToMean * 2);
    }
    return max;
}
```

Listing 4.22: Calculating the GnuPlot range maximum

The maximum value is only normalized if the second y-axis is set to decryptions or runtime, or for the x-axis. The distance between the lowest value and the normalized average is compared to the distance between the highest value and this average. If the *highestToMean* value is more than twice as big as the *lowestToMean*, the maximum value is set to the normalized average plus twice the *lowestToMean* distance.

4.3 Graphical Programming Precautions

The special graphical programming language in CT2 has some negative aspects, too. Once a single input variable changes, the *Execute*-method of that component is executed. This can cause race conditions and stop the process flow. In order to wait for all necessary inputs, it is best to introduce boolean variables. They should be set to true for a new value and only back to false when the new value has been consumed by the algorithm. Listing 4.23 shows the declaration and the important parts in the code of the *_newPlaintext* boolean variable.

```
private bool _newPlaintext = false;
// [...]
if (value != this._plaintextInput)
{
    this._plaintextInput = value;
    this._newPlaintext = true;
    OnPropertyChanged("PlaintextInput");
}
// [...]
if (_newKey && _newPlaintext)
{
    // consume new values
    _newPlaintext = false;
    // [...]
```

Listing 4.23: Boolean variables to check new values

Boolean variables like *_newPlaintext* should be implemented for each input variable of every component involved in the analysis.

5 Evaluation Methodology

This chapter is designed to be a guide for a developer in CT2 who wants to use the CryptAnalysisAnalyzer (CAA) with full evaluation capabilities to analyze a given CipherAnalyzer (CA) component. We have built the CAA as a meta framework designed to analyze and evaluate classical CAs in CT2 in a standardized way. Our main goal is making classical CAs performance evaluations comparable through reproducible test vectors. In order to provide all necessary information for the CAA to evaluate the CA component in depth, the component has to implement an evaluation input and output connector. The basic functionality of the CAA is only able to calculate the runtime and success probability relative to the provided ciphertexts and keys. This chapter explains the important key points when using the CAA based on the CCA acting as CA.

5.1 Evaluation Input and Output Connectors

Through the evaluation inputs *CorrectPlaintext* and *MinimalCorrectPercentage*, the developer can hand the plaintext and a percentage value over to his CA. This percentage provides how many characters of the provided correct plaintext and decrypted ciphertext have to be equal at least, to count as a successful decryption. The evaluation output is implemented as a complex data type named *Evaluation-Container*. It can be filled with various values (see Section 5.2).

5.2 EvaluationContainer

List 5.1 itemizes all implemented values in the *EvaluationContainer*.

List 5.1: Data type EvaluationContainer

- unique ID
- the necessary runtime
- the necessary number of decryptions
- the number of hill climbing restarts
- the population size
- the Tabu set size

5.2.1 ID

The ID is used as the key in the dictionary with the *ExtendedEvaluationContainer* (containing all evaluation data for this one test run) as value. This way, the value in the dictionary may be updated later on through this unique ID. The idea behind this structure and the unique ID is that many CT2 components send their best plaintext and best key not only once, but multiple times if they find a better one. The best way is sending the best plaintext and key only once at the end of the analysis. The process and intermediate best values are usually visible through the component presentation anyway.

The ID is just the hash code of the ciphertext; as the ciphertext may only be used once in the TestVectorGenerator (TVG), they are unique in one test vector series. The data type is integer.

5.2.2 Runtime

The runtime is an optional value, as it differs greatly depending on the hardware. If the runtime is to be measured in the evaluation, both the time at the start and the end of the analysis have to be captured. The elapsed runtime, is the start time, subtracted from the end time. The data type *DateTime* has to be transformed into a *TimeSpan*[1] and added to the *EvaluationContainer*.

5.2.3 Quantity of Decryptions

One of the most meaningful values is the number of decryptions, necessary to get the best plaintext and key. The decryptions should always be measured and

[1]The data type *TimeSpan* stores a span of time, by storing the individual parts of the time span separately (like seconds, minutes, hours, and days).

handed over to the *EvaluationContainer*. This can be done by just counting the decryptions in a global variable in the CAA. If the analysis operates multi-threaded, a separate counter for each thread must be used. These have to be added up to get the total decryptions. The data type is integer.

5.2.4 Hill Climbing Restarts

Hill climbing algorithms usually optimize their results through random restarts. What we are looking for in our evaluation is the number of restarts necessary to obtain the yielded best plaintext and key. In order to achieve this, it is practical to count the restarts in a global variable (one counter per thread) and stop the algorithm once the correct percentage is reached (The stopping is explained in detail in Section 5.3.4). If the algorithm is not a hill climb approach, the *EvaluationContainer* can be used without a restart value. The data type is integer.

5.2.5 Population Size

Genetic algorithms have a population of candidates for the correct key, which are mutated and improved during the analysis. The smaller the population size, the more efficient is the algorithm. So this value is very meaningful for genetic algorithms. The population size variable can just be updated as long as the algorithm runs, and passed to the *EvaluationContainer* with the other values at the end.

If the analysis does not use a genetic algorithm, the *EvaluationContainer* can be initialized without a population size. The data type is integer.

5.2.6 Tabu Set Size

Tabu search algorithms store a list or set of solutions that were dismissed in the past while searching the solution space. The solutions in this set are skipped in the future (usually for a defined amount of time) to avoid cycles. The smaller this Tabu set is, the fewer steps the search algorithm has to make. Therefore, the Tabu set size yields a very expressive measure for the efficiency of Tabu search algorithms. If the analysis does not use a Tabu search algorithm, the *EvaluationContainer* can be initialized without a Tabu set size. The data type is integer.

5.3 Evaluation of the CylinderCipherAnalyzer

The CCA is a hill climbing algorithm that has been equipped with the evaluation features during this thesis. It provides the values ID, runtime, decryptions, and restarts through the *EvaluationContainer*.

5.3.1 Enabling and Disabling

The evaluation may be disabled simply by not providing the correct plaintext or the minimal correct percentage and making the stopping (Section 5.3.4) dependent on them. If the evaluation should also be possible without providing these input values and, therefore, without stopping prematurely, a component setting is a good way to achieve the selection. Through this setting, the user is able to en- or disable the evaluation. Listing 5.1 shows the usage of this setting with a decryptions counter in the CCA.

```
<enable setting with decryptions counter>
```

Listing 5.1: Using the enable-setting

5.3.2 Additional Settings

For the purpose of making the user able to decide whether to use the best plaintext comparison and to stop the algorithm early, we have implemented a simple boolean setting into the CCA. If this setting is disabled, the algorithm does not compare the best plaintext with the original one and runs until all the defined restarts are done. The second additional setting is a numeric input field named *ComparisonFrequency*. It defines the number of improvements that have to occur until the current best plaintext is compared to the original plaintext once. Our experiments have shown that the best frequency for the CCA is 75. Comparing at every improvement does improve the results, but leads to a higher runtime (for more details see Section 6.2). This will most likely be different for other algorithms.

5.3.3 Collecting the Data

Determining the ID is trivial, only the hash code of the ciphertext needs to be calculated (shown in Listing 5.2).

```
int ID = Ciphertext.GetHashCode();
```

Listing 5.2: Calculating the ID

The current time can be acquired from *DateTime.Now*. Doing so twice and subtracting the first value from the second one yields the elapsed time between the two calls. The format *DateTime* contains separate values for milliseconds, seconds, minutes, and so on. In order to get a *TimeSpan*, these values can be handed over into the constructor of a new *TimeSpan* (shown in Listing 5.3).

```
var elapsedtime = DateTime.Now.Subtract(_startTime);
_runtime = new TimeSpan(elapsedtime.Days,
   elapsedtime.Hours, elapsedtime.Minutes,
   elapsedtime.Seconds, elapsedtime.Milliseconds);
```

Listing 5.3: Calculating the runtime

As the CCA is implemented as multi-threaded, values that are counted while operating have to be counted separately per thread. Listing 5.4 visualizes the counting per thread and the adding up after the execution. The ciphertext is decrypted at two positions in the algorithm, so each time the counter has to be increased. In the end, the ciphertext is decrypted one last time with the global best key, to return the best plaintext. Therefore, we added one more decryption in the end.

```
// per thread
_numberOfDecryptions[thread]++;
// [...]
// after execution
for (var i = 0; i < threads; i++) {
  // [...]
    _totalNumberOfDecryptions += _numberOfDecryptions[i];
}
// adding 1 for the last decryption
_totalNumberOfDecryptions++;
```

Listing 5.4: Determining the decryptions

Equal to the decryptions, the restarts have to be counted separately per thread (see Listing 5.5).

```
_numberOfRestarts[thread]++;
// [...]
for (var i = 0; i < threads; i++) {
  // [...]
    _totalNumberOfRestarts += _numberOfRestarts[i];
}
```

Listing 5.5: Determining the restarts

The CCA neither uses a population nor a Tabu set. Both values should be updated if the current value is bigger than the evaluation variable value. Hence, the output will be the maximum value.

5.3.4 Stopping if Percentage Reached

The whole point of providing the correct plaintext and the minimal percentage is being able to check the current best plaintext periodically against the correct plaintext and being able to stop if they match at least the given percentage. If the algorithm runs in a loop, this loop must be broken at that point. Listing 5.6 visualizes how we implemented this in the CCA.

```
double currentlyCorrect =
   CorrectPlaintextInput.CalculateSimilarity(bestPlaintext)
   * 100;

if (currentlyCorrect >= MinimalCorrectPercentage) {
    _finished = true;
    restarts = 0;

    // if the globalbestkey is null, set global values to
       current ones
    if (globalbestkey == null) {
        globalbestkeycost = bestkeycost;
        globalbestkey = bestkey;
        globalbestplaintext = bestplaintext;
    }
}
```

Listing 5.6: Stopping the CylinderCipherAnalyzer

In order to check the similarity of the current best plaintext and the correct plaintext, we simply added the class *SimilarityExtensions* to the analyzer and used its method *CalculateSimilarity* (shown in Listing 5.7). *SimilarityExtensions* implements a method to compute the Levenshtein distance between two strings. This distance is then divided by the longer string length and subtracted from 1 to retrieve the similarity (percentage between 0 and 1).

```
public static double CalculateSimilarity(this string
   source, string target) {
    // [...]
    double stepsToSame = ComputeLevenshteinDistance(source,
       target);
    return (1.0 - (stepsToSame /
       (double)Math.Max(source.Length, target.Length)));
}
```

Listing 5.7: Calculating the string similarity

It is very important when and how often this string similarity is computed. Checking it in every iteration will slow down the analyzer drastically. It is advised only to compute it on each improvement, or even only on every few improvements, if the algorithm is still slowed down too much by it. Checking if the algorithm should stop is visualized in Listing 5.8.

```
if (_settings.StopIfPercentReached &&
   MinimalCorrectPercentage != 0 &&
   !String.IsNullOrEmpty(CorrectPlaintextInput))
{
    // [...]
    double currentlyCorrect =
       CorrectPlaintextInput.CalculateSimilarity(
       <currentBestPlaintext>.ToString()) * 100;

    if (currentlyCorrect >= MinimalCorrectPercentage)
    {
        _finished = true;
        restarts = 0;
        // [...]
    }
}
```

Listing 5.8: Checking if percentage was reached

We check if the setting to stop the algorithm at the correct percentage is activated and if both the minimal percentage and the correct plaintext are available. Afterwards, we calculate the percentage of equality between the two texts and set the *_finished* variable to true on success.

5.3.5 Passing the Data to EvaluationContainer

Listing 5.9 shows the public data output *EvaluationOutput* of the CCA. The *EvaluationContainer* provides multiple different constructors for different combinations of evaluation values.

```
public EvaluationContainer EvaluationOutput {
    get {
        // calculate runtime and ID
        // [...]
        return new EvaluationContainer(ID, _runtime,
           _totalNumberOfDecryptions,
           _totalNumberOfRestarts);
    }
}
```

Listing 5.9: Initializing and returning new EvaluationContainer

5.3.6 Resetting the Evaluation Variables

To make sure that the collected values are correct, they should be reset before each execution. Listing 5.10 shows our code in the CCA. All variables that are directly incremented during the execution should be initialized before that, like the elements in the array *_numberOfDecryptions*.

```
_runtime = new TimeSpan();
_totalNumberOfRestarts = 0;
_totalNumberOfDecryptions = 0;
_numberOfRestarts = new int[_settings.CoresUsed];
_numberOfDecryptions = new int[_settings.CoresUsed];
for (int t = _settings.CoresUsed - 1; t >= 0; t--) {
    _numberOfRestarts[t] = 0;
    _numberOfDecryptions[t] = 0;
}
```

Listing 5.10: Resetting evaluation variables

5.4 The Component Setup in CrypTool 2

Figure 5.1 shows the complete component setup for the analysis in CT2.

The encryption element *Cipher* in our example is the CylinderCipher component. The CCA component is the *Analyzer* element (the complete name is CipherAnalyzer (CA) to be more precise) in the model. Both have to be replaced by the components to analyze[2].

The CAA operates in three different states: 1. Produce test data, 2. Collect data, and 3. Evaluate. The first and second state alternate. In the beginning, it is waiting in state one for the TVG to pass over a test vector. State one is highlighted in blue and corresponds to state one in List 3.10 on Page 35. The key and plaintext are handed over to the *Cipher*, which encrypts the plaintext using the key and returns the ciphertext to the CAA. The ciphertext is also passed to the CA. If the plaintext comparison is activated in the CAA, the minimal percentage to reach for a successful decryption and the plaintext are given to the *Analyzer*, too. Any necessary additional information can be added in the settings of the components in CT2, such as the offset calculation for the CCA's hill climbing algorithm.

After that, the CAA is waiting in state two for the results of the CA. State two is highlighted in green and corresponds to state two in List 3.10. Once all values are passed over (ciphertext, best key, best plaintext, *EvaluationContainer* in our example), the CAA creates an *ExtendedEvaluationContainer*, storing all the

[2]Only the CA is analyzed, the *Cipher* is necessary to produce the ciphertext test vector.

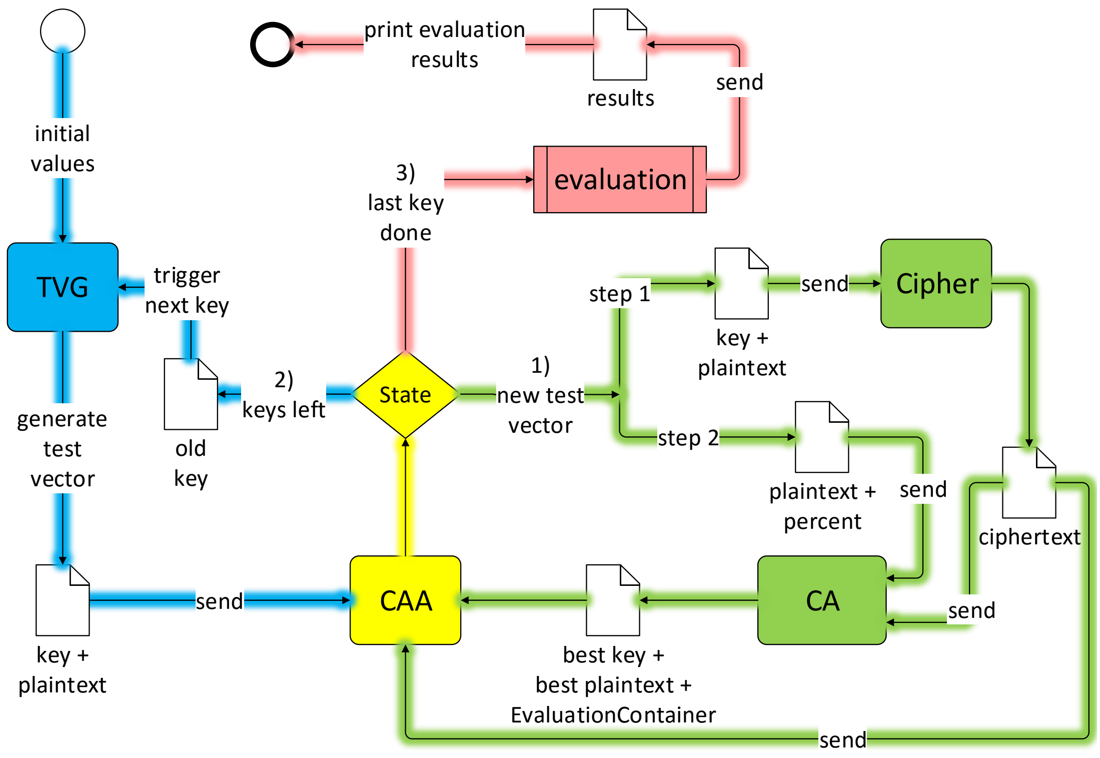


Figure 5.1: Setup of all the evaluation components in CT2

additional information. This container is stored in a dictionary with the ID from the test vector as the dictionary key. We call that the collection of evaluation data. If the processed test vector has not been the last one, the CAA triggers a new test vector from the TVG by providing the last key as the new seed. After triggering a new key, the CAA goes back to state one.

State three is entered after the last key has been processed. State three is highlighted in red and corresponds to state three in List 3.10. In this state, the average values are calculated and the evaluation and GnuPlot outputs are generated from these values.

We highlighted the CAA yellow, because all three states belong to it, and it is active in all three of them (Figure 3.1 on Page 13 shows the different colored states of the CAA itself).

6 Evaluation of the CylinderCipherAnalyzer

In this chapter, the main focus of this chapter lies on the evaluation of the CylinderCipherAnalyzer (CCA) using the CryptAnalysisAnalyzer (CAA). We go into detail about the different aspects of the analysis. The main aspects are the comparison of various numbers of restarts, using the early stopping at 80% versus calculating the specified number of restarts, and 3-gram versus 4-gram based cost functions. Both cost functions are built into the CCA, as well as a combination of both. In order to keep this analysis reasonably short, we evaluated both cost functions only separately. A 3-gram analysis assigns values to pairs of three letters and compares the values to those of English text. A 4-gram analysis uses pairs of four letters accordingly.

Summarizing the evaluation, we can state that the CCA works good in general, as well as our components TVG and CAA. However, we found a minor flaw in the CCA, that leads to much better results for multiples of 25 (see Section 6.1 for a detailed explanation).

6.1 3-Gram Overall Evaluation Results

The key length for the CCA is fixed to 25 for the M-94 model. A typical range of the plaintext length is around 25-150 characters, as 25 is too short to be broken by most CAs and 150 showed 100%success on average with all tested number of restarts. Consequently, the results show the section where the most progression of the success happens. We have chosen an increase of five characters per test run. For the purpose of balancing out outliers, we have evaluated each length 10 times. Other parameters are the number of restarts, stopping the calculation at a fixed percentage, and using 3-grams instead of 4-grams for the calculation.

First, we have discussed the influence of the number of restarts, using 3-grams, and ignoring the option to stop early on success. We have collected the different success rates (on the y-axis) in Figure 6.1 depending on the ciphertext length (on the x-axis) and various number of restarts (one number of restarts per graph). The tested restarts are 50, 100, 250, 500, 1000, 2500, and 5000 with ciphertext lengths from 25 to 150. The increase in length between two texts is five. We have evaluated each text length 10 times and have generated the average values, to make outliers less significant.

The plot displayed in Figure 6.2 is the only plot with the percentage as x-axis, containing all seven graphs - because it is harder to grasp as the ones with only

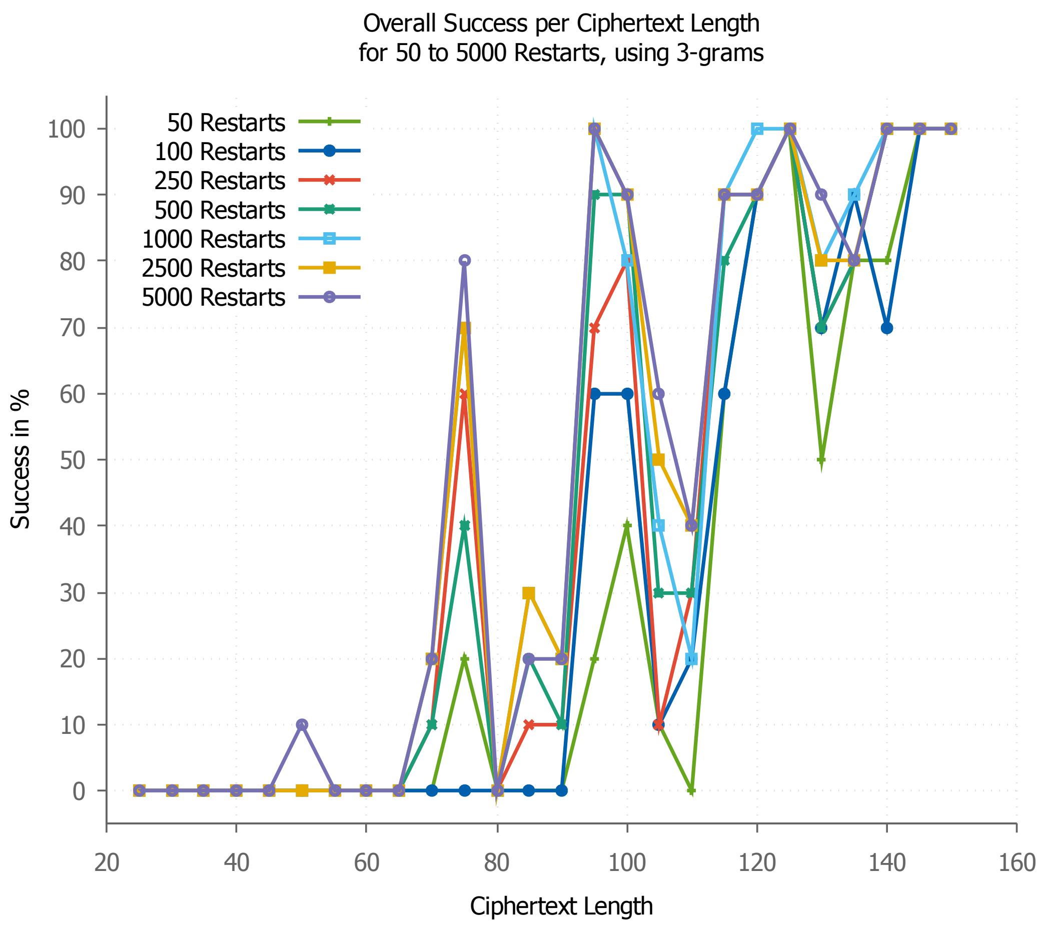


Figure 6.1: Overall success for different restarts per ciphertext length

three graphs. However, we want to show the detailed success progression at least once. All following graphs are much less overlapping. Figure 6.1 visualizes that the algorithm seems to be very ineffective from ciphertext lengths of 25 to 65 and has its best values at 75, 95, 125 and 150[1]. With 50 and 100 restarts, the algorithm hardly finds any results for the ciphertext lengths of 25 to 90. In total, 100 restarts outperform 50 and more restarts yield higher success than less restarts. "Waves" occur in the plot because of the way the algorithm is implemented. In this implementation, some text lengths are appended by multiple X characters up to lengths that are dividable by 25 (i.e. 50, 75, 100, 125, 150). This causes all text lengths not dividable by 25 to be a lot worse than those that are multiples of 25.

It is interesting to see that the improvements through the increased number of restarts are not always proportional, and many areas seem to be random. This is mainly caused by the specific algorithm implementation we have explained above. Two equal test runs may produce significantly different results using a heuristic.

[1]We have been surprised about the second value being 95 and not 100. We do not have an explanation for that.

The effect should be lessened through more repetitions. Average runtimes up to almost a minute, 10 repetitions for each of the given 26 different ciphertext lengths, and repeating the process multiple times with different parameters take a lot of time.

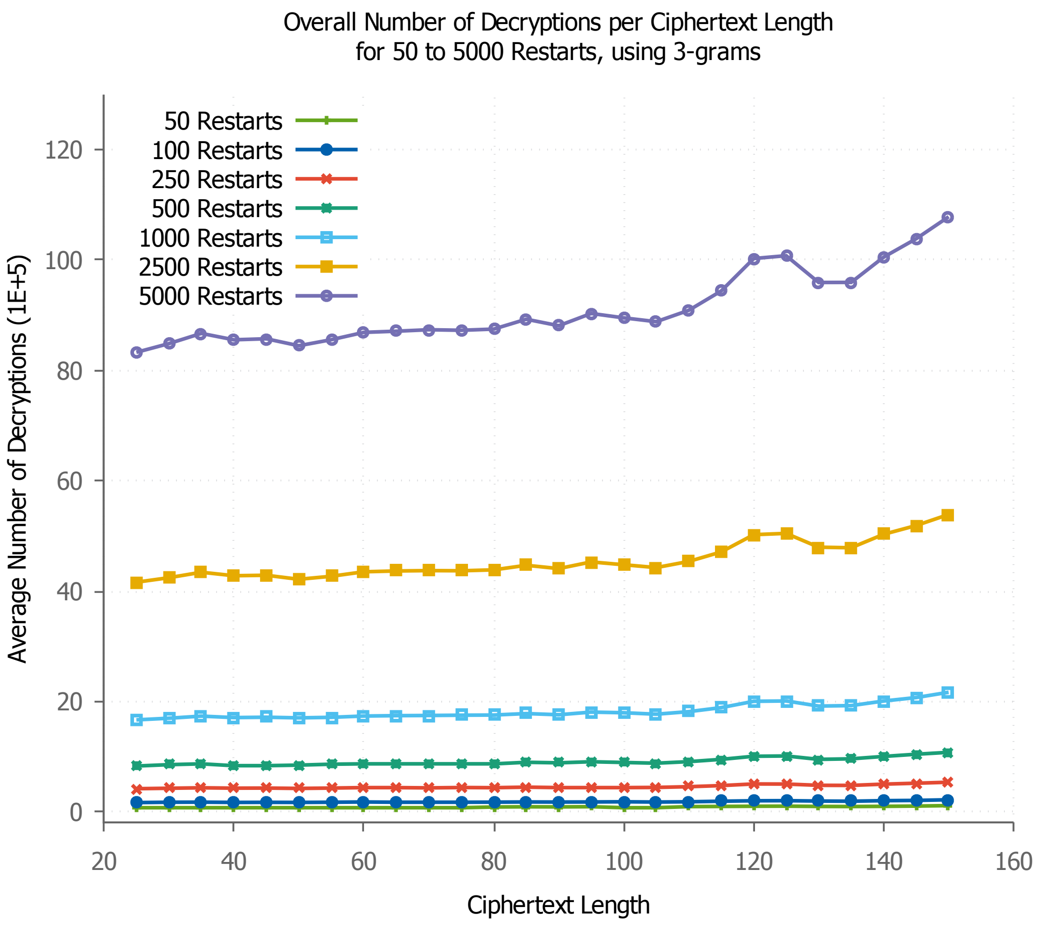


Figure 6.2: Overall decryptions for different restarts per ciphertext length

It is clear to see that a higher number of restarts leads to more success. That is mainly caused by the higher number of decryptions going along with more restarts (see Figure 6.2).

	Restarts						
	50	100	250	500	1000	2500	5000
$\varnothing Success$ [%]	28.85	31.92	38.85	40.00	43.85	44.62	45.77
$\varnothing Decrypted^2$ [%]	57.55	58.64	62.31	63.19	64.01	64.53	65.67
$\varnothing Decryptions$ [N]	$91*10^3$	$18*10^5$	$46*10^4$	$91*10^4$	$18*10^5$	$46*10^5$	$91*10^5$
$\varnothing Runtime$ [s]	0.5	1	2.5	4.7	8.9	22.3	46.6

Table 6.1: Average values of various restarts using 3-grams

Table 6.1 visualizes the average values of four metrics using restarts from 50 to 5000 and 3-grams. While the decryptions and the runtime increase nearly proportional to the restarts, the success and the percentage of correctly decrypted ciphertext increases rather sparsely. Especially above 500 restarts, the percentages do not increase significantly. That is one reason why we have chosen 500 restarts for the detailed comparison with the early stopping[3].

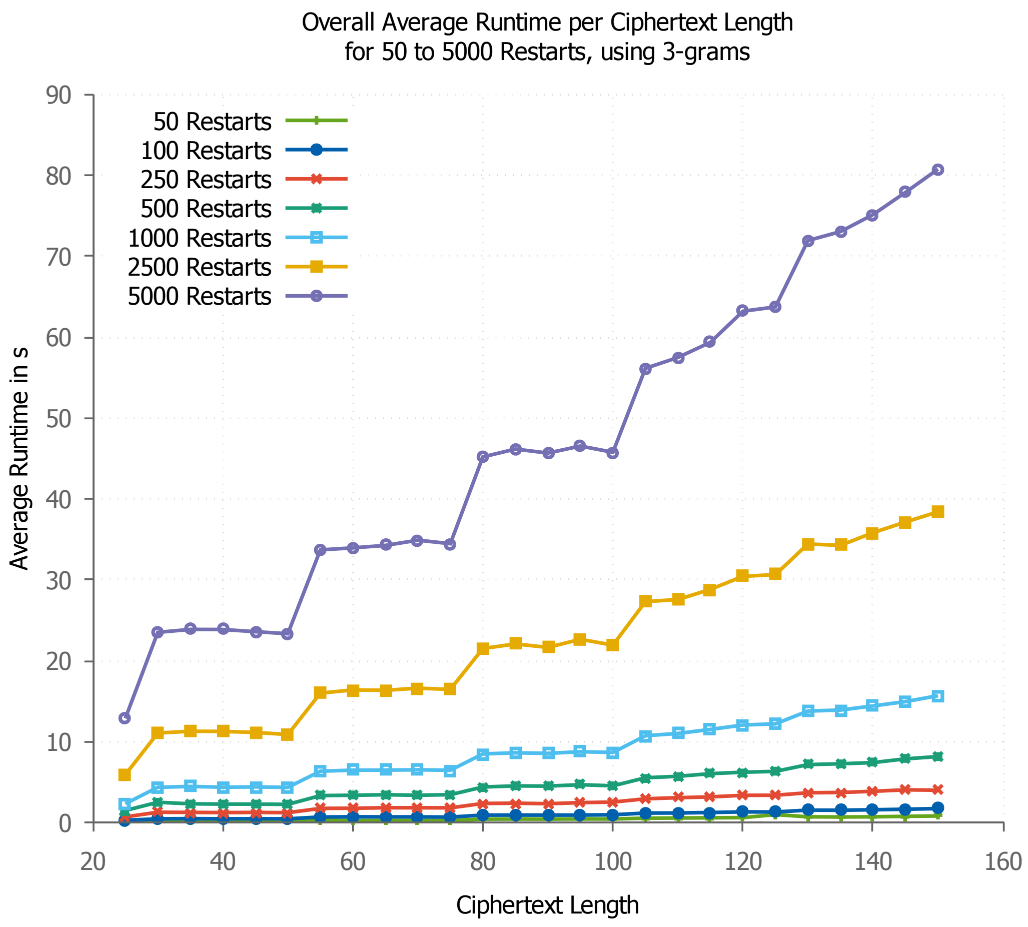


Figure 6.3: Overall runtime for different restarts per ciphertext length

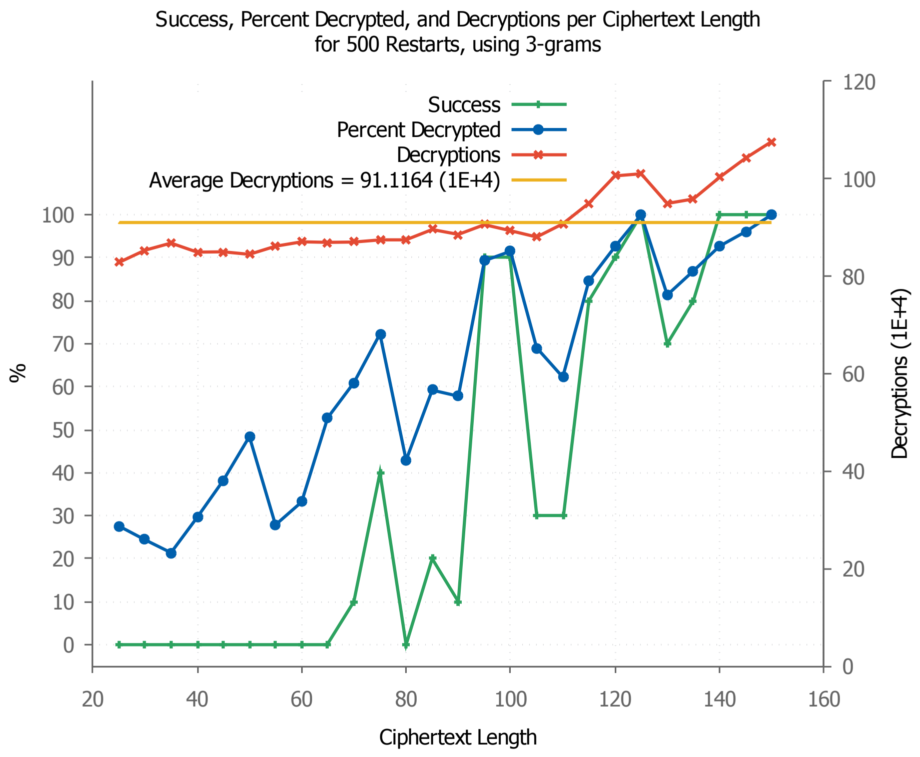
The longer the algorithm runs and calculates the costs for new keys, the better the results will be (see Figure 6.3).

6.2 3-Gram 500 Restarts Evaluation Results

The first plot in Figure 6.4 displays the success, percentage of decryption, and the number of decryptions in dependency on the ciphertext length for 500 restarts.

[2]Correctly decrypted on average

[3]Another reason for choosing 500 restarts over 1000 restarts is the increase in runtime of about 18 minutes. This makes 500 restarts much easier to test under various circumstances.



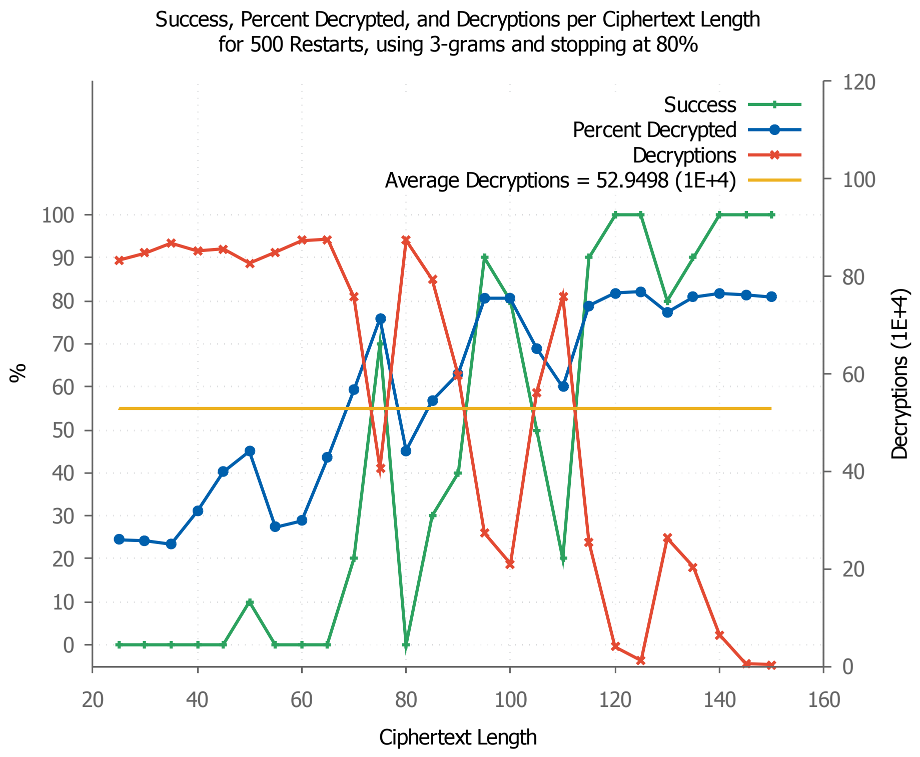


Figure 6.4: Average success for 500 restarts using 3-grams with and without stopping early

This analysis is based on 3-grams and going through all 500 restarts. We will compare the effort to another analysis with early stopping activated, contrasting the number of decryptions through the second plot in Figure 6.4 and the restarts through Table 6.2. The success and percentage of correct decryption are shown in both figures.

The option to stop the algorithm early at a specified percentage of correctness has been implemented in the context of this thesis. It is based on comparing the current best plaintext with the original plaintext on improvements of the key costs.

While the success is very similar or even improving, the number of decryptions drops by almost 42% and the number of restarts drops by 39%. This is a huge improvement over the original algorithm.

The drawback, however, is the increase in runtime by over 121%. We did not recognize this drastic increase until we compared the actual average values from the evaluation output (shown in Table 6.2).

	Not stopping	stopping
∅Success [%]	40.00	45.00
∅Decrypted correctly [%]	63.19	58.59
∅Decryptions [*N*]	$91 * 10^4$	$53 * 10^4$
∅Restarts [*N*]	500	305
∅Runtime [*s*]	4.7	10.4

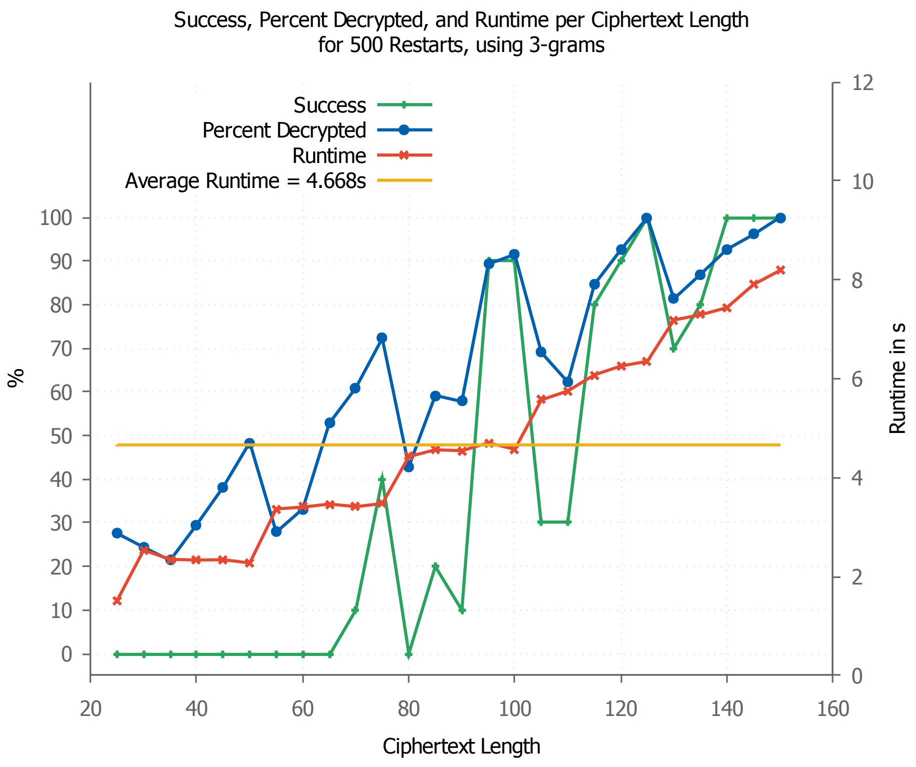
Table 6.2: Average values of 500 restarts using 3-grams and stopping

We reacted by implementing an additional setting, to only compare the similarity between the best plaintext and the original plaintext on every x^{th} improvement of the key costs. We evaluated different frequencies of comparing the best plaintext with the original plaintext on improvements. The comparison in Table 6.3 shows that every 100^{th} improvement returns the lowest average runtime.

	Not	Improvements to compare after					
	stop.	1	50	75	100	125	150
∅Success [%]	40.00	45.00	40.38	41.92	40.00	41.15	40.00
∅Decrypted[4] [%]	63.19	58.59	60.83	60.71	60.75	61.34	61.75
∅Decryptions [*N*]	$91{*}10^4$	$53{*}10^4$	$72{*}10^4$	$73{*}10^4$	$77{*}10^4$	$75{*}10^4$	$80{*}10^4$
∅Restarts [*N*]	500	305.0	406.6	411.6	431.4	424.0	446.6
∅Runtime [*s*]	4.668	10.43	3.832	3.813	3.768	3.999	4.349

Table 6.3: Comparing different comparison frequency performances

[4]Correctly decrypted on average



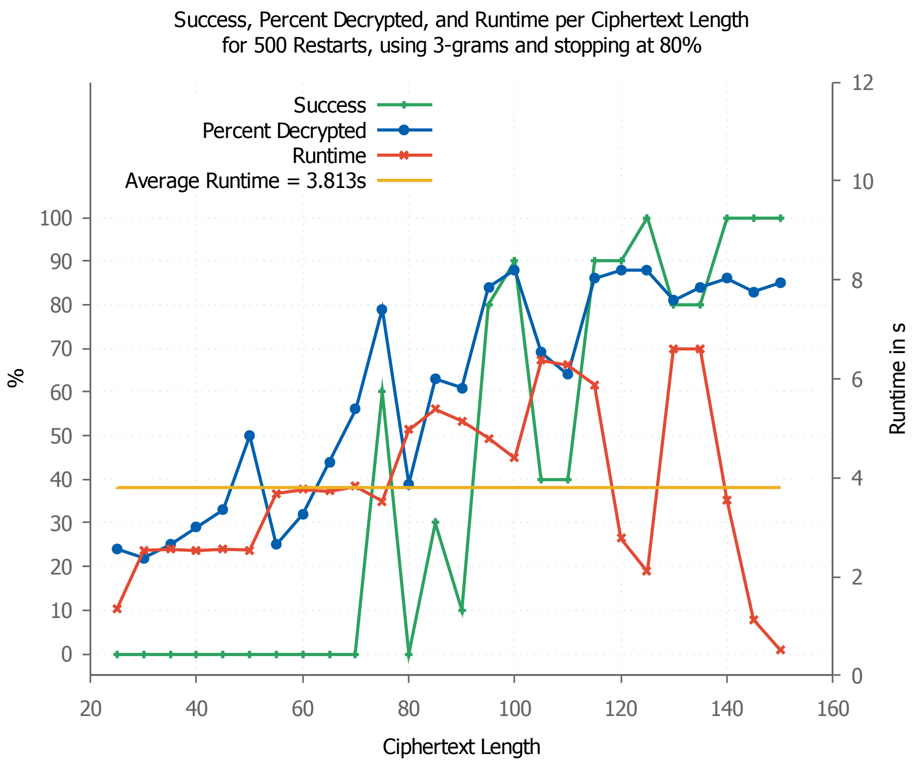


Figure 6.5: Average runtime for 500 restarts using 3-grams with and without stopping early

Table 6.3 highlights the differences between the original algorithm and its modified versions. The original algorithm does not stop until all restarts are finished; the first modification compares the best plaintext with the original plaintext at each improvement, while the other improvements experiment on the frequency of comparison. When the best plaintext matches the original one at least by the given percentage (80% in our scenario), the algorithm is stopped and this best key and plaintext are used as result. The table shows that the success with stopping is at least the same as without stopping, while the percentage of correct decryption is always lower. The lower decrypted percentage is the result of saving decryptions, restarts, and runtime through the stopping. The success is much more important than the specific percentage of correct decryption, because usually the rest of the ciphertext is already readable at a percentage of 80%. While the comparison at each improvement returns the highest success, it also increases the runtime drastically. If the only goal is maximizing success, this definitely is the way to go.

As we want to speed up the algorithm through the early stopping, we have compared the other frequencies of comparison. The fastest algorithm is the one with a frequency of 100. While the second fastest (75) is only 1.19% slower, it returns 4.80% more successful results. So we have adjusted the frequency to 75 and repeated all test runs involving stopping. Figure 6.5 was produced using the latest modification of the algorithm with comparing the plaintext similarities at every 75^{th} improvement.

6.3 4-Gram Overall Evaluation Results

Using 4-grams instead of 3-grams returns very similar results. Figure 6.6 visualizes the difference in success between 50, 500, and 5000 restarts.

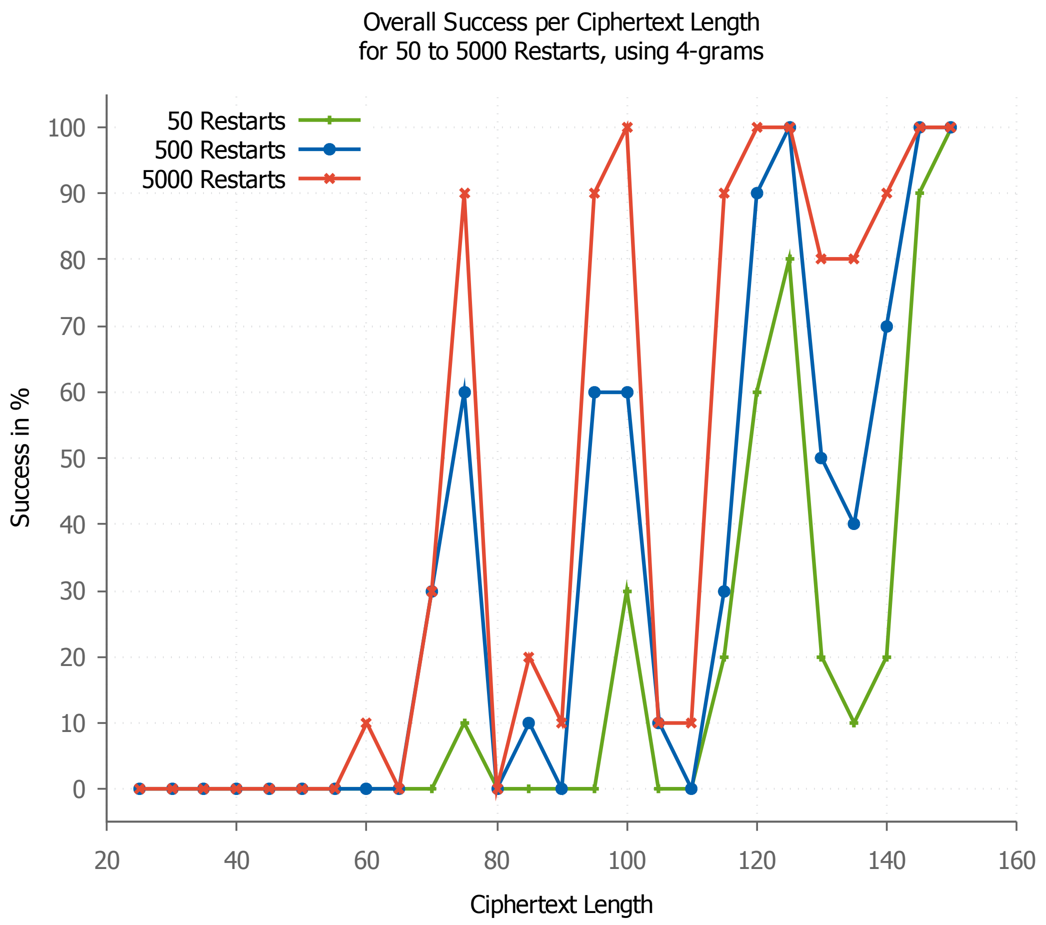


Figure 6.6: Overall success for different restarts per ciphertext length using 4-grams

As with 3-grams, all test runs start to find first usable results for very small ciphertext length. At a ciphertext length of 75 characters, the results start to get better. More restarts also clearly yield better results using 4-grams.

Looking at the 4-gram analysis, the average success and the percentage of correct decryption increase the most between 50 and 500 restarts on the one hand. On the other hand, there are 10 times more decryptions and the runtime is nearly separated by a factor of 10 between the three values. This corresponds to the improvements in the 3-gram analysis. What is more interesting is the difference in average values between the 3-gram and 4-gram analysis, listed in Table 6.4.

[5]Correctly decrypted on average

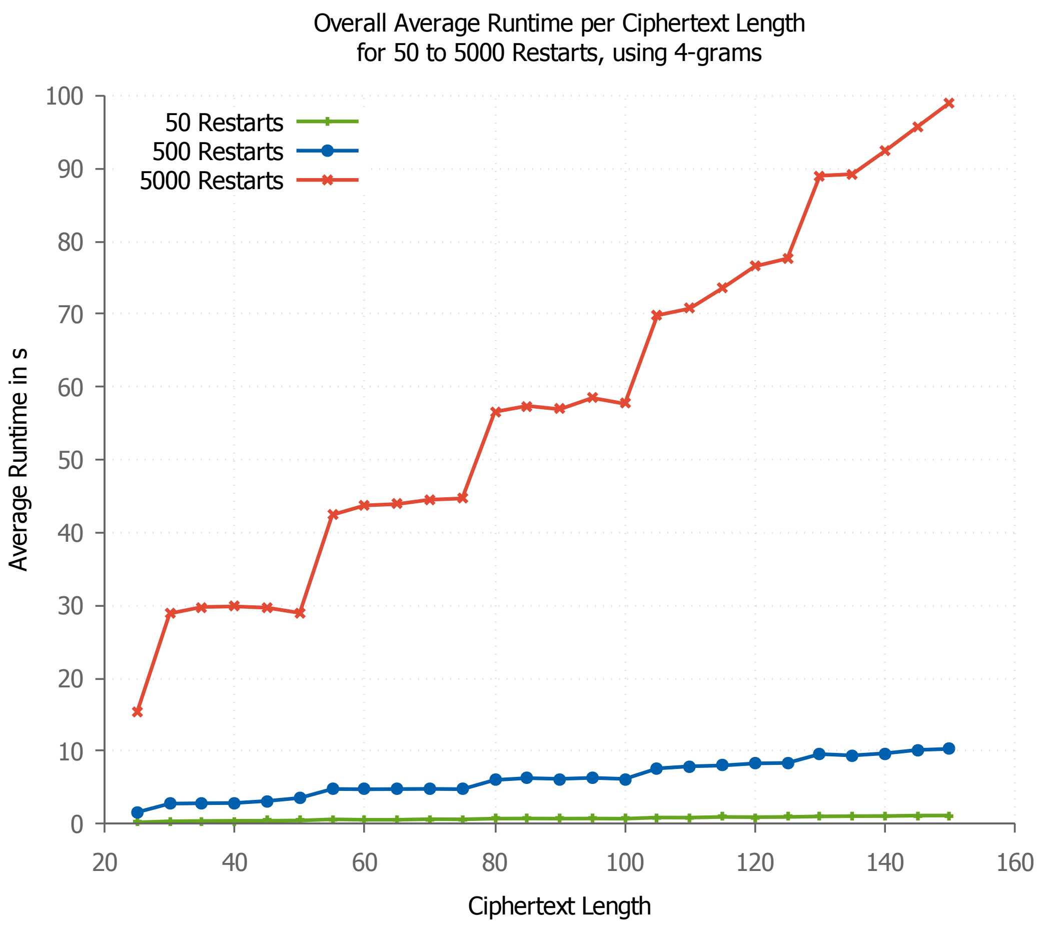


Figure 6.7: Overall runtime for different restarts per ciphertext length using 4-grams

	3-Grams			4-Grams		
Restarts	50	500	5000	50	500	5000
∅*Success* [%]	28.85	40.00	45.77	16.92	31.15	42.69
∅*Decrypted*[5] [%]	57.55	63.19	65.67	50.39	59.99	63.85
∅*Decryptions* [*N*]	$91{*}10^3$	$91{*}10^4$	$91{*}10^5$	$89 * 10^3$	$89{*}10^4$	$89{*}10^5$
∅*Runtime* [*s*]	0.5	4.7	46.6	0.7	6.2	57.8

Table 6.4: Average values of various restarts using 4-grams

In the direct comparison, the 3-gram analysis retrieves the much higher success, while having less runtime and doing slightly more decryptions. Comparing 50, 500, and 5000 restarts, the 3-gram analysis retrieves 23% more success on average.

Additionally, the runtime of the 4-gram analysis increases more than the one of the 3-gram analysis. This is good to see in the 5000 restarts graph in Figure 6.7. This increase in runtime, especially when evaluating with a very high number of restarts, can be prohibited through the early stopping option. The next section compares the stopping and non-stopping results using 4-grams.

6.4 4-Gram 500 Restarts Evaluation Results

The last aspect we want to analyze is the difference between the 4-gram analysis with 500 restarts with and without stopping early.

What is clear to see between the two plots in Figure 6.8 are the similar success and decrypted percentages, while the average number of decryptions drops at higher success rates.

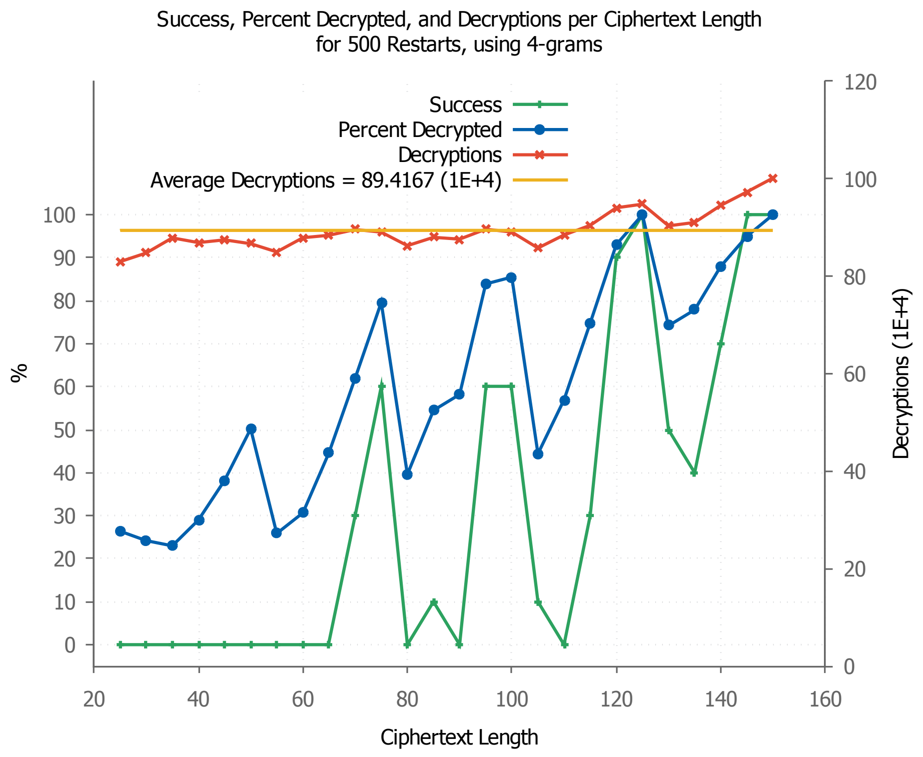
	Not stopping	stopping
$\varnothing Success\ [\%]$	31.15	32.31
$\varnothing Decrypted\ correctly\ [\%]$	59.99	57.08
$\varnothing Decryptions\ [N]$	$89 * 10^4$	$81 * 10^4$
$\varnothing Restarts\ [N]$	500	454.7
$\varnothing Runtime\ [s]$	6.2	5.2

Table 6.5: Average values of 500 restarts using 4-grams and stopping

As Table 6.5 states, the average number of decryptions dropped by almost 10%, the average number of restarts by 9%, and the average runtime decreased by 1s (~ 15%). Interestingly, the success rate improved by nearly 4% through the stopping option, used with comparing the best plaintext to the original plaintext at every 75th improvement.

Figure 6.9 presents the comparison of the average runtime between the 4-gram analysis without stopping with the analysis with stopping.

The runtime clearly drops most for ciphertext length above 115 characters. Overall we can state that the stopping option also improves the 4-gram results of the algorithm.



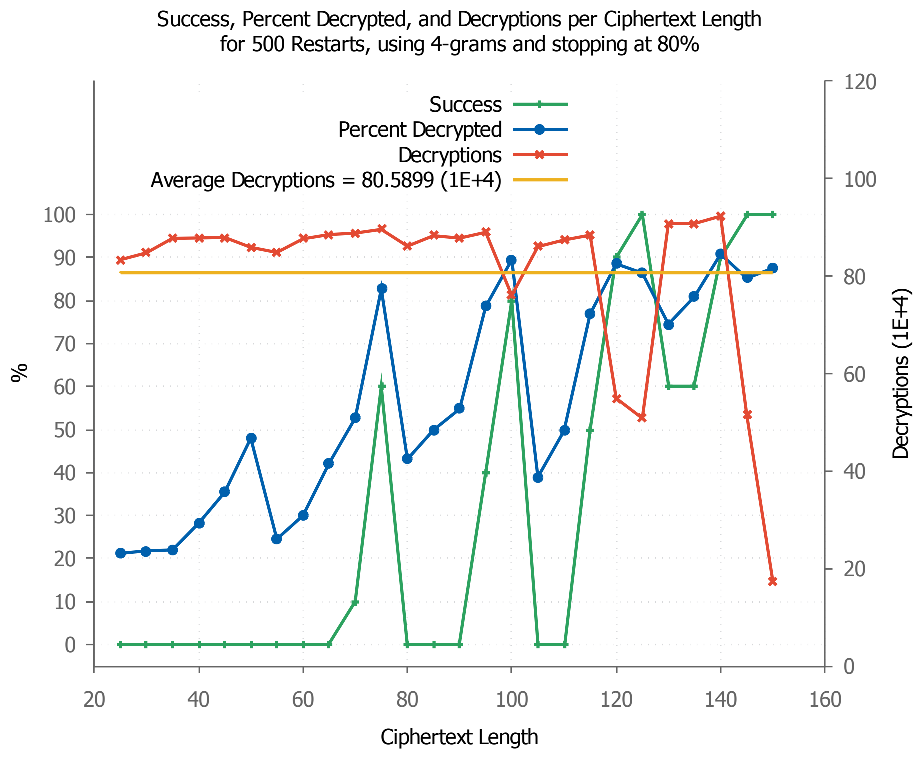
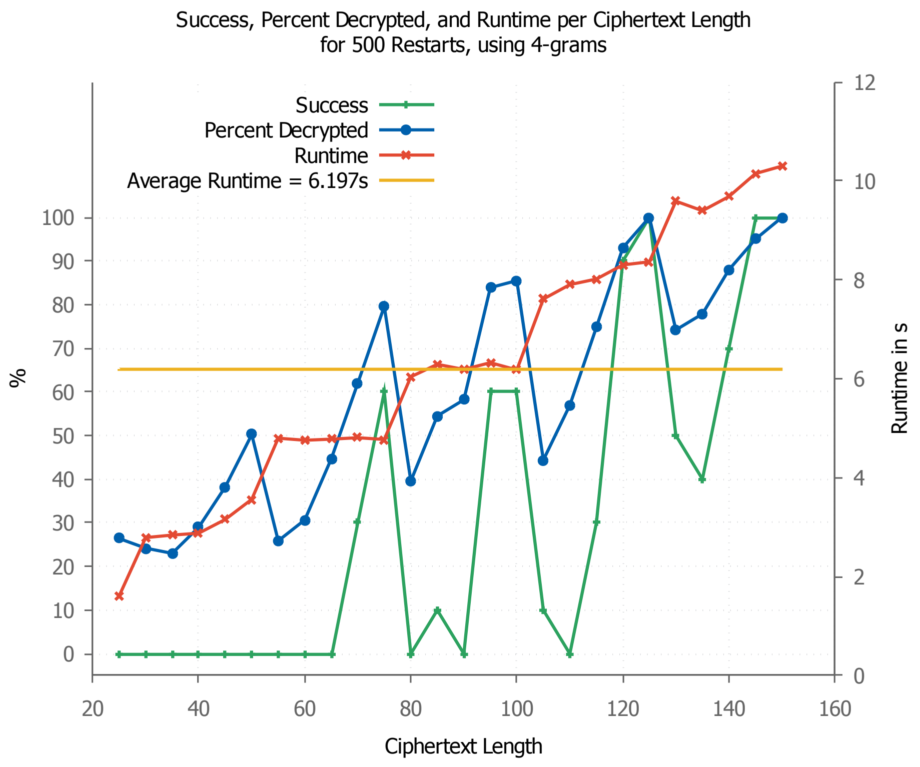


Figure 6.8: Average success for 500 restarts using 4-grams with and without stopping early



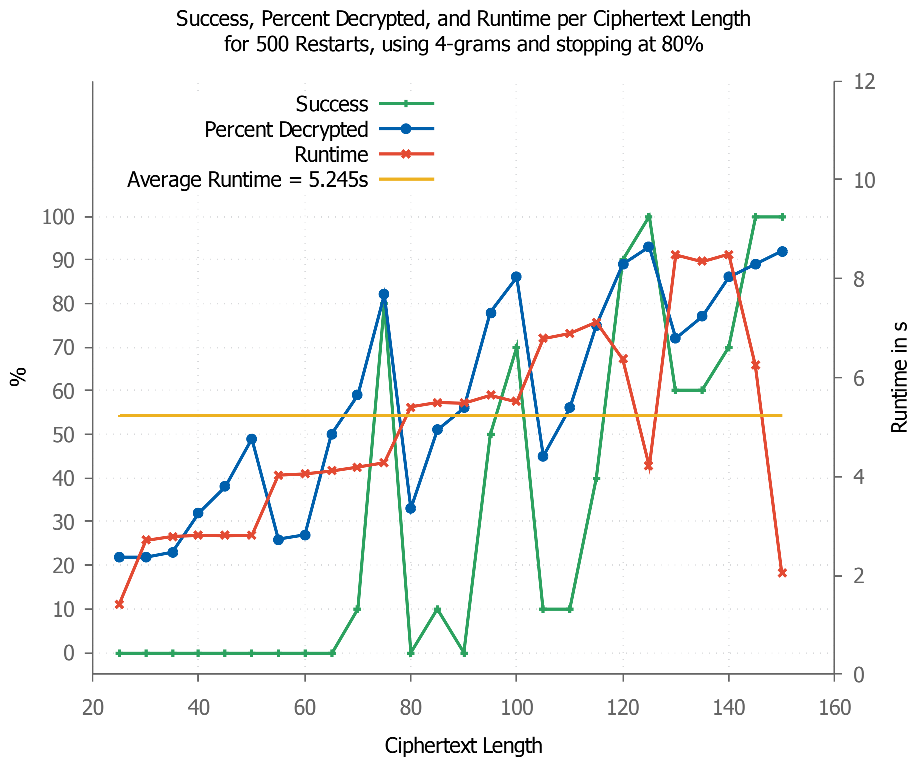


Figure 6.9: Average runtime for 500 restarts using 4-grams with and without stopping early

7 Related Work

One of the most used software concerning testing in cryptology is the “Cryptographic Algorithm Validation Program (CAVP)” of the National Institute of Standards and Technology (NIST) [20]. They also generate and use test vectors, but they focus on modern cryptography and particular implementations of algorithms. Besides, they do not test cryptanalytic algorithms. The Website “Cryptography.io” [10] provides test vectors where CAVP lacks them, but also no cryptanalytic algorithms. The field of cryptanalysis (especially for classical ciphers) does neither seem to have any test vector generators available nor some publicly available predefined test vectors. One fraction of cryptanalysis has test suits, namely the pseudo-random number generators. The standard there is the “DieHarder” test suite [2].

It is interesting to take a look at the test vectors and evaluation metrics of other IT branches, such as image processing, graphical rendering, data compression, or voice over IP. The main difference is that most of them have a predefined amount of test data that does not have to be extended or adjusted.

Regarding image processing or graphical rendering, there are very limited standardized test vectors necessary for testing [3][21]. Also in data compression, the Canterbury corpus is the one test vector to analyze and compare different data compression algorithms [18]. The Harvard Sentences are the standardized test vectors for most voice transmission like voice over IP [23]. They consist of 72 fixed sets of 10 sentences. These sentences are available as spoken audio files in various languages. Although they could be recorded in all possible languages, the main testing is done with the original English sentences. This makes the number of test vectors limited enough to provide all of them through a website [25]. Classical cryptology does not only have many more different ciphers that need different kinds of keys. It also has varying demand of plaintext formats and lengths. The task of providing predefined test vectors becomes much more complex. This might be one of the reasons why there are no standardized test vectors in the field of classical cryptology. Our TestVectorGenerator (TVG) is designed to solve this issue.

8 Conclusion and Future Work

In this chapter, we summarize what we have achieved in this thesis. In addition, we justify in detail the defined requirements and the achievements with the developed software. We also explain the main problem during the development and evaluation. Then, we take a look at aspects to improve in the future, as the software was developed prototypically.

8.1 Conclusion

In the context of this thesis, we have developed the CrypTool 2 (CT2) components TestVectorGenerator (TVG) and CryptAnalysisAnalyzer (CAA). We have explained the concepts behind them and we have showed their usage. Chapter 5 is intended to be a guide for developers who want to use our components. For the results in Chapter 6, we have collected data from over 10,000 test runs in over 50 test series[1]. We have evaluated this data and we have edited it to yield vivid plots.

8.2 Requirements Conclusion

The following sections describe which requirements (see Section 3.2 on Page 15) we have met and how we have met them:

[1]Each test series consists out of 260 test runs (26 different ciphertext lengths multiplied with 10 runs for each). We have evaluated 7 different restarts for the 3-grams (each involving the 260 test runs once), all with and without stopping, and 3 different numbers of restarts for the 4-grams, with and without stopping. Besides, we have evaluated many stopping test runs with various frequencies of comparison. This is the net amount of test runs. A lot of test series had to be done multiple times. For each stopping test series we saved 6 different files (3 script files and 3 data files) and for each non-stopping test series we saved 4 different files. The overall average number of decryptions per test run is above 2 million if all numbers of restarts are evaluated equally often. Because we evaluated 500 restarts much more often (with an average of about 0.8 - 0.9 million decryptions), the average number of restarts has been between 1 and 2 million. Resulting, the overall number of decryptions done for our analysis, is well above 20 billion.

(R01) Accepting a text as input to generate plaintext test vectors

Section 3.3.1 describes all input connectors of the TVG component, the first being the input text as string. This satisfies requirement (R01).

(R02) Generation of plaintext test vectors

Natural language plaintext is required for the cost functions of many algorithms. We describe the generation in Section 3.3.4 and in more detail in Section 4.1.3. Finally, Section 3.4.2.3 shows some generated plaintexts. This requirement has been implemented successfully.

(R03) Generation of natural language keys taken from input text

The natural language key generation works similar to the plaintext generation. It is described in Section 3.3.5.1; generated keys are displayed in Section 3.4.2.2. These keys show the correct functioning of the natural language key generation.

(R04) Generation of simple random keys

Section 3.3.5.2 explains the simple random key generation; Section 4.1.5 lists the excerpt of code that generates the random keys. The main focus of this simple to use random key generation is making it usable by inexperienced users as well. Our implementation through simple key format selection with drop down menus satisfies requirement (R04).

(R05) Generation of random keys through reverse regex

We show the usage and some of the specific options in Section 3.3.5.3. The implementation is explained in Section 4.1.6. Section 3.4.2.1 shows 10 generated Enigma keys using the reverse regex generation. Through *Xeger* and the additional *$length* and *$unique* variables, the reverse regex generation is very powerful. Therefore, requirement (R05) is fulfilled.

(R06) Accepting a seed which makes all generations reproducible

In Section 3.3.1, we explain the input connector for the seed. The particular parts where the seed is used to initialize a pseudo-random number generator are listed and explained in Section 3.3.3 and following. Additionally, Section 4.1.1 goes more into detail about the implementation of the seed input connector.

(R07) Settings for the plaintext length, key lengths, and number of test runs

All settings are listed and explained in Section 3.3.2, including settings for the plaintext length, key lengths, and the number of test runs. Section 4.1.2 shows how we have implemented such a setting. So requirement (R07) is fulfilled.

(R08) Generation of test vector sets for all classical ciphers in CT2

We have taken a look at the key formats of different classical ciphers of CT2. The *$unique* and *$length* options in the regex key generation enable the user to generate very complex keys (explained in Section 3.3.5.3), especially in combination with the power of *Xeger* (see Section 3.4.1). This allows the key format of all classical ciphers that are currently available in CT2. In order to use the evaluation of the CAA with other components, they currently have to provide a decrypted ciphertext, the according key, and an evaluation container. These outputs can be implemented following Chapter 5.

(R09) Accepting test vectors from the TestVectorGenerator

This is the first requirement for the CAA. We have implemented string inputs for the key and plaintext test vectors, as well as for the total amount of keys (explained in Section 3.5.1). Therefore, we have satisfied requirement (R09).

(R10) Feeding test vectors to the Cipher and accept evaluation input from the CipherAnalyzer (CA)

This is mainly controlled by the *Execute* method of the CAA, which controls the current state the CAA is in. In state 1, it excepts test vectors, gives them to the output connectors, and switches to state 2. In state 2, the results are awaited through the evaluation input connectors (for more information see Section 3.6.1 for a descriptive algorithm of the CAA and Section 4.2.2 for in depth code excerpts).

(R11) Evaluation of cryptanalytic methods with variable text and key length and variable algorithm parameters

Most of the settings are chosen in the TVG. The CAA adds one more algorithm parameter, the minimal percentage for a correct decryption. The other algorithm specific settings can be chosen in the particular *Cipher* and CA to analyze. In the CCA, we have also added extended options for the evaluation (described in Section 5.3.2). Other component specific settings may be implemented as necessary. Requirement (R11) has been fulfilled.

(R12) Visualization of the testing process

We set the value of the current progress of the CAA while evaluating, but through the graphical programming interface, the percentage is set to a very high value once the component is left. Because the majority of processing time is spent in the CA to analyze, the percentage shown in the beginning appears to be too high and does not change much. Therefore, we have implemented a progress bar into the evaluation output connector string, as well as information about the current key and the last test run results (see Section 3.5.1 for more details). This feature meets requirement (R12).

(R13) Generation of GnuPlot script files and data files that draw a plot visualizing the evaluation results

At the end of state 2 of the CAA (see Section 3.6.1) and on each plaintext length change, we trigger the evaluation and generate GnuPlot script files and data files. These script and data file string outputs can be copied into separate files and directly used by GnuPlot (explained in Section 3.5.3). The plots are formatted to fit the specifics.

(R14) Evaluation of the CA “CylinderCipherAnalyzer”

Chapter 6 goes into detail about our evaluation of the CCA. We have evaluated multiple variable settings using various metrics. These results are visualized and compared in four sections. This satisfies requirement (R14).

(R15) Visualization of the functionality of the TVG and CAA on the basis of the evaluation in (R14)

The evaluation in (R14) shows the intended functioning of both components with each other. Furthermore, Section 3.4.2 shows the TVG in scenarios of pure test vector generation. So requirement (R15) is met.

8.3 Difficulties

Most of the programming progressed over time without real difficulties. One problem kept returning due to the graphical programming language in CT2 (mentioned in Section 4.3). The first obstacle was providing empty initial values for every input connector. We had to implement the *EvaluationContainerInput* component, which only provides an empty *EvaluationContainer*. Without that, the components kept waiting for all connected inputs to transfer a value.

In a very late stage of programming, the analysis of the CCA kept building up race conditions and stopping randomly at some points. The problem was the consuming behavior of the input connector values of our components. The solution to that was implementing one boolean variable for each input connector that is set to true for a new input value. After consuming this value, the variable is set back to false. This was necessary for every used input connector of every involved component that is part of the circular evaluation flow in the interface. This resolved the problem so far that the evaluations in Chapter 6 became possible. Unfortunately, the analysis still stops at some point, after a few hundred test runs in one test series. Resolving this issue was too complex within the scope of this thesis and not absolutely necessary to use the components for evaluations.

8.4 Future Work

The developed prototypes have a few drawbacks that should be improved in the future. The main problem remains the unresolved stopping of the evaluation in CT2 at some point that has to do with the graphical programming. Another aspect to advance might be the progress bar accuracy of the CAA and through that, the overall progress in the user interface.

The current implementation of the CAA requires the input of the best found plaintext, the according key and ciphertext, and of an evaluation container. In order for the evaluation to work without any of these parameters, their dependency has to be made optional in the CAA. Reducing the analyzed parameters like that leads to a reduced evaluation (e.g. not providing the number of decryptions through an evaluation container removes this parameter from the evaluation).

As the settings in the different *Cipher* and CA components are very complex, it would be convenient to transmit the current settings over to the CAA and log them. Giving the CAA the functionality to modify these settings would be even more convenient and allow more complicated testing scenarios.

Currently, the initial values for the CAA have to be entered externally. We had to create a dummy component that is only able to provide an empty *EvaluationContainer* for the CAA. It would be straight forward to be able create the empty initial values in the CA component and pass them over to the CAA, or even directly in the CAA.

Besides these improvements, it would be very convenient for large evaluations to have more automation regarding the multiple files to export, in order to store all evaluation output. The built-in GnuPlot settings could also be improved, to enable the user to select the plot ranges manually, or to format certain values in specific ways in the plot. Another helpful feature would be starting GnuPlot directly from CT2 with the correct working directory and load command, to automate the plotting process.

In oder to make the work flows more transparent and increase the performance, the Cipher and CA components could be used as embedded functions of the CAA.

Nevertheless, the biggest aspect to improve upon certainly remains in the development of other evolved templates for the evaluation of many different CT2 components. This will be an ongoing process of various contributors of CT2.

Bibliography

[1] A. Møller, *dk.brics.automaton.* Website, 2001. [http://www.brics.dk/automaton/; accessed June 06, 2017].

[2] R. G. Brown, D. Eddelbuettel, and D. Bauer, *Dieharder: A Random Number Test Suite.* Website, 2017. [http://webhome.phy.duke.edu/~rgb/General/dieharder.php; accessed July 25, 2017].

[3] C. Rosenberg, *The Lenna Story - www.lenna.org.* Website, November 03, 2001. [http://www.cs.cmu.edu/~chuck/lennapg/; accessed April 02, 2017].

[4] C. Wollerton, *Wheel Cipher.* Website, February 13, 2017. [https://www.monticello.org/site/research-and-collections/wheel-cipher; accessed July 17, 2017].

[5] F. A. Campos, A. Gascón, J. M. Latorre, and J. R. Soler, *Genetic Algorithms and Mathematical Programming to Crack the Spanish Strip Cipher*, Cryptologia, 37 (2013), pp. 51–68.

[6] A. Clark and E. Dawson, *A PARALLEL GENETIC ALGORITHM FOR CRYPTANALYSIS OF THE POLYALPHABETIC SUBSTITUTION CIPHER*, Cryptologia, 21 (1997), pp. 129–138.

[7] M. J. Cowan, *Breaking Short Playfair Ciphers with the Simulated Annealing Algorithm*, Cryptologia, 32 (2008), pp. 71–83.

[8] A. Dhavare, R. M. Low, and M. Stamp, *Efficient Cryptanalysis of Homophonic Substitution Ciphers*, Cryptologia, 37 (2013), pp. 250–281.

[9] J. J. Gillogly, *CIPHERTEXT-ONLY CRYPTANALYSIS OF ENIGMA*, Cryptologia, 19 (1995), pp. 405–413.

[10] Individual Contributors, *Test vectors.* Website, 2017. [https://cryptography.io/en/latest/development/test-vectors/; accessed July 16, 2017].

[11] J. J. G. Savard, *The Bazeries Cylinder.* Website, 2012. [http://www.quadibloc.com/crypto/ro020101.htm; accessed July 17, 2017].

[12] A. Kerckhoffs, *La cryptographie militaire*, Journal des sciences militaires, IX (1883).

[13] N. Kopal, O. Kieselmann, A. Wacker, and B. Esslinger, *CrypTool 2.0 - Open-Source-Kryptologie für Jedermann*, Datenschutz und Datensicherheit (DuD), 38 (2014), pp. 701–708.

[14] L. Carroll, *Alice's Adventures in Wonderland by Lewis Carroll.* Website, June 27, 2008. [`http://www.gutenberg.org/ebooks/11`; accessed July 26, 2017].

[15] G. Lasry, N. Kopal, and A. Wacker, *Solving the Double Transposition Challenge with a Divide-and-Conquer Approach*, Cryptologia, 38 (2014), pp. 197–214.

[16] ——, *Automated Known-Plaintext Cryptanalysis of Short Hagelin M-209 Messages*, Cryptologia, 40 (2016), pp. 49–69.

[17] ——, *Cryptanalysis of columnar transposition cipher with long keys*, Cryptologia, 40 (2016), pp. 374–398.

[18] M. Powell, *The Canterbury Corpus.* Website, November 20, 2001. [`http://corpus.canterbury.ac.nz/`; accessed April 02, 2017].

[19] N. Baxevanis, *Fare - [F]inite [A]utomata and [R]egular [E]xpressions.* Website, December 2011. [`https://github.com/moodmosaic/Fare`; accessed June 06, 2017].

[20] NIST, *The Cryptographic Algorithm Validation Program (CAVP).* Website, January 28, 1996 (updated June 01, 2017). [`http://csrc.nist.gov/groups/STM/cavp/`; accessed July 16, 2017].

[21] P. Knowles, *Common 3D Models/Meshes used in Computer Graphics Research.* Website, December 08, 2014. [`http://goanna.cs.rmit.edu.au/~pknowles/models.html`; accessed July 16, 2017].

[22] R. R. Vique, *Xeger.* Website, 2013. [`https://github.com/robertrv/xeger`; accessed June 06, 2017].

[23] S. Zhang, *The 'Harvard Sentences' Secretly Shaped The Development Of Audio Tech.* Website, March 10, 2015. [`https://www.gizmodo.com.au/2015/03/the-harvard-sentences-secretly-shaped-the-development-of-audio-tech/`; accessed April 02, 2017].

[24] T. Schrödel, *Breaking Short Vigenère Ciphers*, Cryptologia, 32 (2008), pp. 334–347.

[25] VoIP Troubleshooter LLC, *The Open Speech Repository.* Website. [`http://www.voiptroubleshooter.com/open_speech/index.html`; accessed July 16, 2017].

[26] W. Springer, *Xeger.* Website, November 26, 2009. [`https://code.google.com/archive/p/xeger/`; accessed June 06, 2017].

[27] ——, *XegerLimitations.wiki.* Website, November, 26, 2009. [`https://code.google.com/archive/p/xeger/wikis/XegerLimitations.wiki`; accessed June 06, 2017].

Versicherung an Eides statt

Ich, Bastian Heuser, Matrikelnummer 30220193, wohnhaft in 64347 Griesheim, versichere an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen übernommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe.

Ich versichere an Eides statt, dass ich die vorgenannten Angaben nach bestem Wissen und Gewissen gemacht habe und dass die Angaben der Wahrheit entsprechen und ich nichts verschwiegen habe.

Die Strafbarkeit einer falschen eidesstattlichen Versicherung ist mir bekannt, namentlich die Strafandrohung gemäß § 156 StGB bis zu drei Jahren Freiheitsstrafe oder Geldstrafe bei vorsätzlicher Begehung der Tat bzw. gemäß § 163 StGB bis zu einem Jahr Freiheitsstrafe oder Geldstrafe bei fahrlässiger Begehung.

Griesheim, August 8, 2017

Bastian Heuser