Implementation and Didactical Visualization of the ChaCha Cipher Family in CrypTool 2

Bachelor's Thesis

Ramdip Gill

Supervisor
Priv.-Doz. Dr. Wolfgang Merkle

Second Supervisor
Prof. Dr. Frederik Armknecht

Heidelberg, December 11, 2020

Faculty of Mathematics and Computer Science
Heidelberg University

ABSTRACT

This thesis is about the implementation of the ChaCha plug-in in CrypTool 2.

The thesis introduces the ChaCha cipher family, explains what the plug-in is capable of, and gives insight into the development process of the plug-in.

ChaCha is used in the Transport Layer Security protocol (TLS) since 2014 and so very relevant for applied modern cryptography. Because of the importance of ChaCha its internal design should be made more accessible to the broader public. This is the actual goal of the plug-in.

The goal is achieved by focusing on an in-depth but easy to understand visualization of the encryption process. CrypTool 2 is the most popular e-learning platform in the field of cryptology, used in schools, universities, and companies. Incorporating this plug-in into CrypTool 2 helps to reach a broad audience.

ZUSAMMENFASSUNG

Diese Bachelorarbeit befasst sich mit der Implementierung des ChaCha Plugins für CrypTool 2.

Die Arbeit stellt die Familie der ChaCha-Chiffren vor; erklärt, wozu das Plugin in der Lage ist; und gibt Einblick in den Entwicklungsprozess des Plugins.

ChaCha wird seit 2014 im Transport Layer Security-Protokoll (TLS) verwendet und ist daher für die angewandte moderne Kryptographie sehr relevant. Aufgrund der Bedeutung von ChaCha sollte sein internes Design der breiten Öffentlichkeit zugänglicher gemacht werden. Dies ist das eigentliche Ziel des Plugins.

Das Ziel wird erreicht, indem man sich auf eine detaillierte, aber leicht verständliche Visualisierung des Verschlüsselungsprozesses konzentriert. CrypTool 2 ist die beliebteste E-Learning-Plattform im Bereich der Kryptologie und wird in Schulen, Universitäten und Unternehmen eingesetzt. Durch die Integration des Plugins in CrypTool 2 wird so ein breites Publikum erreicht.

Acknowledgment

First of all, special thanks to Dr. Wolfgang Merkle who was willing to be the adviser from my home university, the University of Heidelberg. He introduced me to Prof. Dr. Frederik Armknecht from the University of Mannheim and thus laid the foundation for all of this. If not for Dr. Merkle, I don't know who else could have been the adviser for a bachelor's thesis in my preferred field, the field of cryptography. Most likely, I would have written my thesis in a different field.

I am also very grateful to Prof. Dr. Frederik Armknecht that he accepted me whom he did not know at all beforehand. He offered me a wide variety of interesting subjects from which I could chose. In the end, I have chosen to develop a plug-in for *CrypTool 2* (CT2), an open-source e-learning platform for cryptography and cryptanalysis.

Therefore, I want to extend my gratitude to the team behind CT2 whose support and welcomeness meant a lot to me. Prof. Bernhard Esslinger is the overall coordinator and Dr. Nils Kopal is the technical lead developer, both from the University of Siegen. Whereas Prof. Dr. Armknecht gave me important feedback from a user's perspective since he uses CT2 in his lectures, Prof. Esslinger and Dr. Kopal always took their time to answer technical questions of mine. Additionally, Prof. Esslinger was always eager to remind me about things that were easy to miss like adding text to tooltips which I didn't even know they existed.

Finally, I also want to thank the people at my new employer Abusix, Inc. They made it possible for me to focus on my thesis by offering very flexible working hours. I am also very grateful that I could work on my thesis on their premises since going into the office was always a guarantee for a very productive day.

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. The principles and recommendations "Verantwortung in der Wissenschaft" of Heidelberg University have been followed.

first and last name

city, date and signature

Contents

1 Introduction

Applications of cryptology, the science behind creating encryptions (cryptography) and breaking them (cryptanalysis), date back far into ancient times. The first known example of cryptography, a substitution cipher to conceal a formula for pottery glaze, is from 3500 BC [Bin20]. Ever since, advancements in technology pushed the boundary for secure ciphers. Nowadays, in the "Age of Information", keeping sensitive information private has never been so important and will only get more important with widespread adoption of new technologies such as the Internet of Things. This is why research into new encryption standards has to continuously take place.

The ChaCha cipher family by Daniel J. Bernstein is the result of such research and was published in 2008 [Ber08]. Because AES-GCM does not perform very well on devices without hardware acceleration such as wearable or mobile devices, Google started to replace AES-GCM in the Transport Layer Security (TLS) cipher suite of its browser Chrome with ChaCha20 for symmetric encryption and Poly1305 for authentication in 2014. Additionally, ChaCha is by design immune to previous TLS attacks such as padding-oracle or timing attacks and thus improves the security of HTTPS connections [Goo14].

This usage in TLS makes the ChaCha cipher family very attractive to include it in *CrypTool 2* (CT2), a free open-source e-learning platform which mainly targets students. It uses visual programming to teach cryptographic concepts and includes visualizations of many different ciphers, attacks and more. Thanks to its underlying architecture, one can easily write plug-ins for it using C#, WPF and XAML. In fact, this was already done multiple times before by students as part of their bachelor thesis.

This thesis will describe the implemented ChaCha plug-in which includes an in-depth visualization of its internals. The visualization also makes it possible to study the diffusion property of the cipher by letting the user alter the input values. The development process and underlying architecture will also be outlined to understand the reasoning behind some design decisions.

2 Related Work

This chapter discusses relevant work for the ChaCha plug-in implementation and was therefore reviewed during the work on this thesis.

2.1 Salsa20 Cipher Family

The ChaCha cipher family is based on the 256-bit stream cipher family Salsa20. Salsa20/20, the 20 rounds variant, was developed by Daniel J. Bernstein in 2005 [Ber05b] and submitted to eSTREAM, a European project to "to promote the design of efficient and compact stream ciphers suitable for widespread adoption" [Eur12].

It uses only add-rotate-XOR (ARX) operations for encryption which prevents timing attacks since they run in constant time on basically all platforms [Ber05a]. Beside 256-bit keys, it also supports 128-bit keys. It internally uses a hash function which transforms a 512-bit state, consisting of the key, four 32-bit constants, a 64-bit initialization vector and a 64-bit counter, into a keystream block. For following keystream blocks, only the counter is incremented in the initial state before applying the hash function. This means that Salsa20 shares the same implementation advantages as block ciphers in counter mode, in particular the ability to generate output blocks in any order and in parallel [Ber05b].

Bernstein later introduced other variants with 8 and 12 rounds, named Salsa20/8 and Salsa20/12, to let users decide for a faster, but less secure cipher. Other round variants like 9, 10 or 11 were not introduced because the difference in speed would be insignificant [Ber06]. The ChaCha cipher family received the same round variants.

There is also a variant of Salsa20 called XSalsa20 which supports 192-bit initialization vectors. Since its implementation varies quite a bit from the Salsa20/r variants and Bernstein introduced XSalsa20 as part of a new cipher family (based on Salsa20), this cipher is of no relevance for this thesis [Ber11]. There is a XChaCha20 variant but it "is currently not widely implemented outside the libsodium library

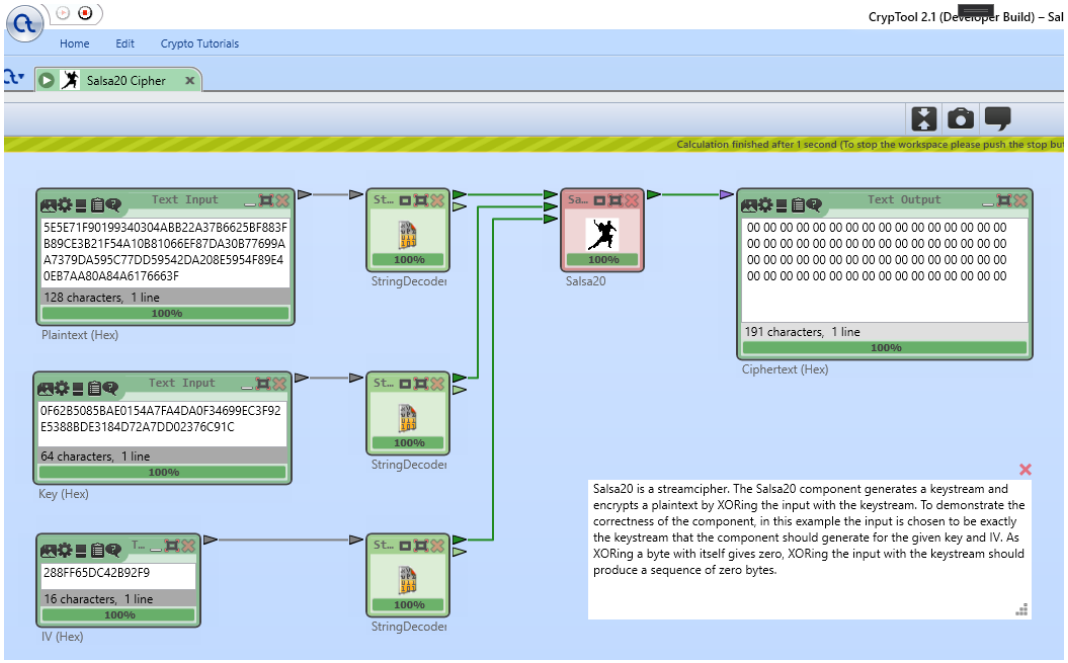


Figure 2.1: CT2 template for the already existing Salsa20 plug-in

[a software library for cryptography], due to the absence of formal specification" [Lib18].

The specification of Salsa20 is very relevant for ChaCha because the specification for ChaCha only mentions the differences [Ber08]. Therefore, to implement ChaCha, one has to also read through the specification of Salsa20. However, Chapter 3 will summarize the specification of ChaCha without assuming prior knowledge about Salsa20.

2.2 Salsa20 CrypTool 2 Plug-in

CrypTool 2 already has a plug-in for the Salsa20 cipher but without a visualization. Figure 2.1 shows the CT2 template for the plug-in. Templates are prepared workspaces for a plug-in where all necessary components to run the plug-in are already included and properly connected.

During the work on the ChaCha visualization, we discussed if the code could be reused to create a visualization for the Salsa20 cipher family. However, this would at least need adaption of the XAML code since the state is built up differently. Also

the quarter-round function is slightly different which also needs to be reflected in the visualization.

Nonetheless, it should be possible to reuse most of the codebase used for the ChaCha visualization, especially the underlying navigation system and the storage and retrieval mechanism for the intermediate results. This will further be discussed in Section 5.2.

2.3 Other CrypTool 2 Cipher Visualizations

This section is all about other existing CT2 cipher visualizations and which ideas originated from them. The plug-ins were also created by students during their bachelor's thesis.

2.3.1 AES Visualization

Matthias Becher created a visualization for the AES cipher in 2016 [Bec16]. It was the visualization of which the most inspiration was taken from for the ChaCha visualization. Reading through his bachelor's thesis was very useful since he encountered similar problems. For example, he wrote the following in his bachelor's thesis:

"The first big decision that had to be made was whether the states after each encryption operation would be calculated during the visualization or precalculated and stored at the start of the execution. One feature the plug-in should have was to not only jump ahead to later operations but also to go back to previous ones. That means if the values were calculated during the visualization every time you went back they would have to be recalculated from the start. Therefore, I decided to precompute and store results of each operation in an array of byte arrays.." [Bec16]

We came to the exact same conclusion that the values need to be precalculated for the reasons he mentioned.

Looking through his visualization, his usage of background coloring was very useful to catch the user's attention. This is shown in Figure 2.2. Therefore, there is a similar mechanic during the ChaCha hash function visualization where a light blue background is put onto the state elements which are used as the quarter-round input. Also during the quarter-round execution, background coloring is extensively used to indicate where the user should pay attention.

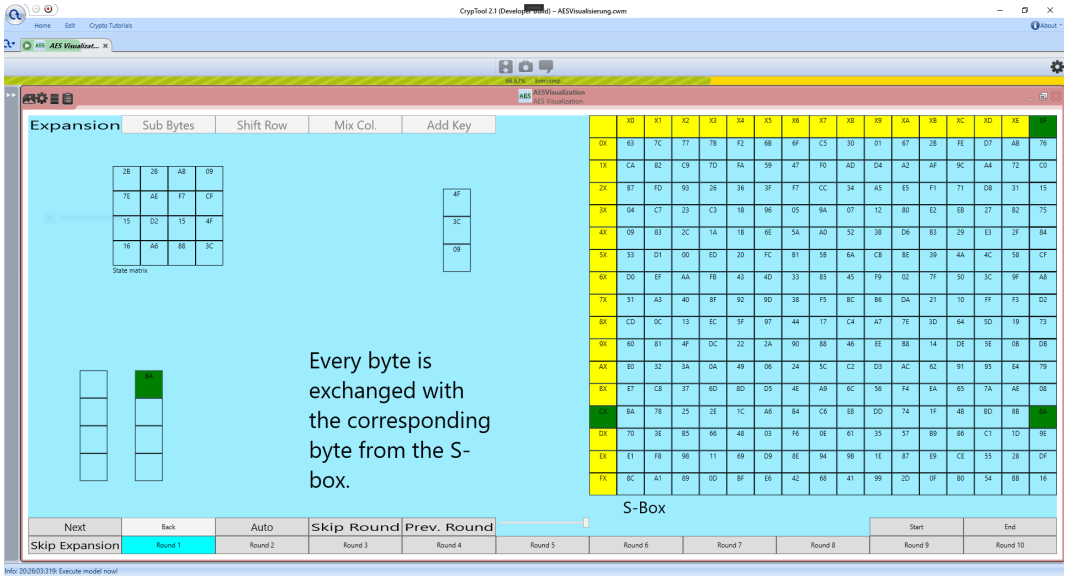


Figure 2.2: AES visualization plug-in

Another thing adopted from his visualization was the navigation in the top-left corner where the page navigation for the ChaCha visualization is placed.
What is different regarding navigation is to not show so many buttons all the time to the user. It was quite overwhelming to see all the buttons in the bottom navigation bar on the start even though they were disabled. Therefore, on pages which have no actions, there are no buttons in the bottom row. For the ChaCha hash function, which needed more detailed navigation, the navigation bar looks similar expect that it uses arrow buttons and text inputs instead of buttons for every single round. This decreased the amount of buttons while maintaining a similar degree of navigation.

Further, it was confusing that the “Back” button during the “Expansion” or “Encryption” step was disabled. For the ChaCha plug-in, the user should be able to navigate to any step in the visualization fairly simple. To achieve this, the page navigation in the top-left corner stays the same on every page and tells the user on which page he currently is with bold font. This means the user knows that if he wants to go to a different page, he needs to select the page there. Additionally, every single action on each page is numbered together with an action text input and how many actions a page has in total. The text input makes it possible for the user to immediately jump to an action. If he does not know the number of the action he wants to go, there are also buttons labeled with descriptive names on

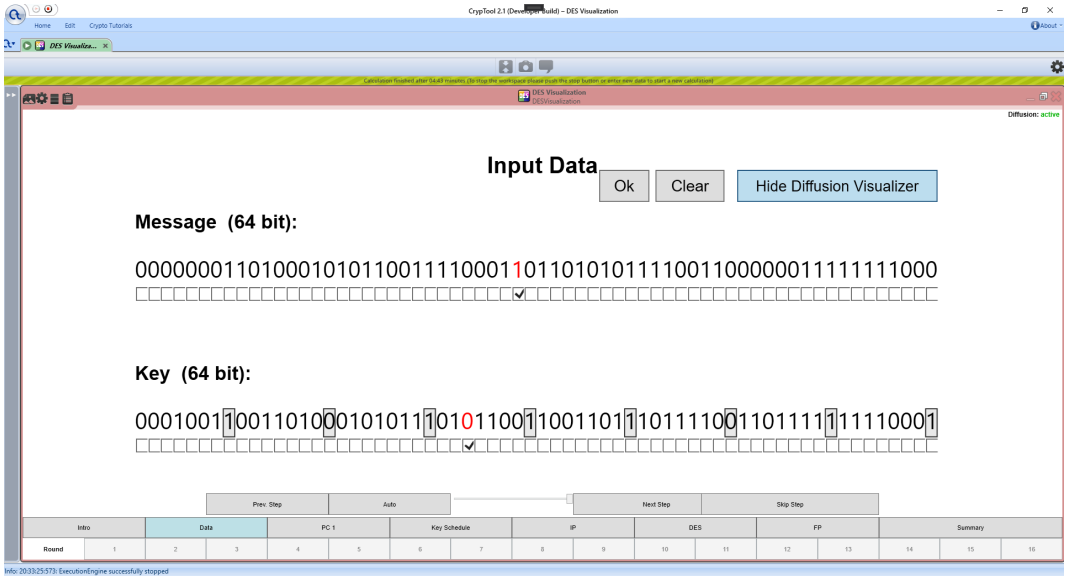


Figure 2.3: DES visualization plug-in

the pages with actions. For example, the page about the state setup has buttons to immediately go to the start or end of the key encoding step.

2.3.2 DES Visualization

The DES visualization was created by Lars Hoffman. His approach to visualizing the diffusion had the most influence on the diffusion visualization in the ChaCha plug-in. In Figure 2.3, you can see the page on which the user can flip bits to activate diffusion. The message and key with flipped bits will then be used to show the diffusion property of DES.

Throughout the visualization, all values are shown in binary. This makes it possible to just mark flipped bits red since if a bit is marked red, we immediately know the value of the diffusion run (we just flip the red bit).

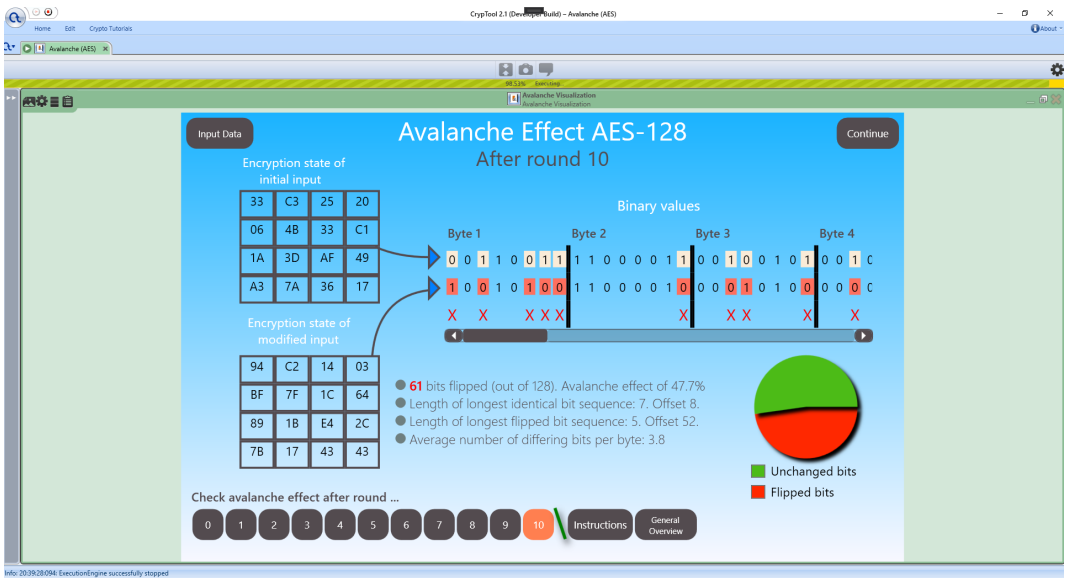
It was tried to use the same coloring approach but since the ChaCha cipher uses longer keys and 512-bit blocks compared to the 64-bit blocks of DES, hex strings needed to be used for the values to save canvas space. This lead to a loss of information about the concrete values if only marking red the hexadecimal characters which are different. Therefore, the usage of red color was combined together with showing both values in two rows. In the row for the altered value, the difference is still marked red for easier visual recognition. This is shown later in Figure 4.11.

2.3.3 Avalanche Visualization

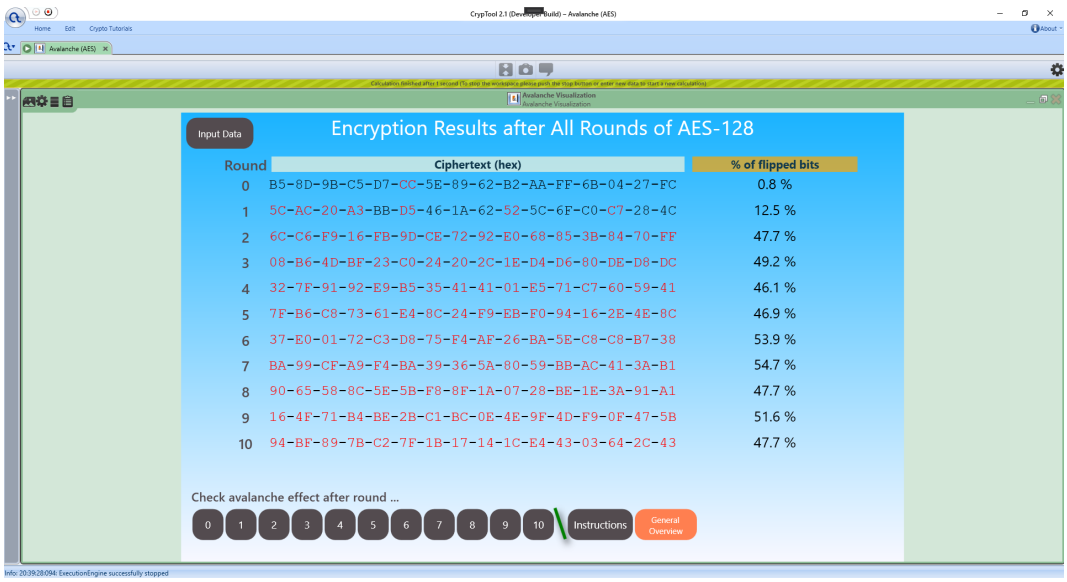
The Avalanche visualization plug-in was created by Camilo Echeverri in 2016 [Ech16]. The most noticeable part about it is probably that it is visually very appealing. Camilo seemed to be very experienced in creating nice user interfaces. Especially the pie chart for the amount of flipped bits with its shadow (Figure 2.4a) and the gradient background stood out. It made it clear that the user interface of the ChaCha visualization should also be visually appealing and not just functional.

The overview over all rounds (Figure 2.4b) was also a remarkable feature. It was discussed that the ChaCha visualization should include a similar overview since seeing how many rounds were needed for half of all bits to be flipped is useful for studying the diffusion property of a cipher. Unfortunately to date, this did not make it into the version of the plug-in but can be added in the future (see Section 5.2).

On the other hand, scrollbars as seen in Figure 2.4a should be prevented because the user should see everything he needs immediately.



(a) AES-128 avalanche visualization: End of all rounds



(b) AES-128 avalanche visualization: General overview

Figure 2.4: Avalanche visualization plug-in

3 ChaCha Specification

To help with understanding the plug-in visualization and for the sake of completeness, this chapter will summarize the specification of the ChaCha cipher.

ChaCha is a 256-bit stream cipher based on Salsa20, both developed by Daniel J. Bernstein. It was designed to improve diffusion per round while maintaining or even increasing the performance compared to Salsa20. This makes it more secure than Salsa20 with the same amount of rounds. It was developed in the year 2008, three years after Salsa20 [Ber08].

The specification can be broken apart into five main points:
The *quarter-round* function, the *little-endian* function, a hash function which utilizes the two other mentioned functions, the setup of the 512-bit state and finally, the encryption/decryption process, bringing all the individual pieces together.

3.1 Quarter-Round Function

The ChaCha quarter-round function takes in four 32-bit unsigned integers which we will name a, b, c and d. It also returns four 32-bit unsigned integers.
It modifies its input values as described in the following pseudo-code:

```
quarterround(a,b,c,d):
    a += b; d ^= a; d <<<= 16
    c += d; b ^= c; b <<<= 12
    a += b; d ^= a; d <<<= 8
    c += d; b ^= c; b <<<= 7
    return a, b, c, d
```

Remark. We define one row, consisting of one 32-bit addition, one XOR and one shift operation, as one *quarter-round step*. This naming convention will be reused in Section 4.2.2.

3.2 Little-Endian Function

The little-endian function takes in one 32-bit unsigned integer and reverses its byte order; also returning a 32-bit unsigned integer. For example, the hexadecimal value `0x12345678` would be transformed into `0x78563412` with this function.
It can be implemented as follows:

```
littleendian(x):
    x0 = (x >> 24) & 0xff
    x1 = (x >> 16) & 0xff
    x2 = (x >> 8) & 0xff
    x3 = x & 0xff
    return (x3 << 24) | (x2 << 16) | (x1 << 8) | x0
```

Remark. Its naming has nothing to do with system endianness, but was just named like this by Bernstein for unknown reasons (most likely because reversing the byte order is what needs to be done when transmitting data between systems of different endianness).

3.3 ChaCha Hash Function

The ChaCha hash function takes in 16 32-bit unsigned integers and returns 16 32-bit unsigned integers. The input vector $(y_0, y_1, y_2, \ldots, y_{15})$ can be written as a 4×4 matrix:

$$\begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{pmatrix}$$

Bernstein uses this matrix representation in his paper to help with understanding why he calls some rounds *column rounds* and others *diagonal rounds* (one round consists of four quarter-rounds):
The ChaCha hash function first iterates through all columns and then through all diagonals of the matrix; applying the quarter-round function to the four entries of each column/diagonal. After the first four quarter-rounds it therefore has changed all columns of the matrix. This is what Bernstein calls a column round in his paper. After the next four quarter-rounds, it changed all diagonals of the matrix which Bernstein analogously calls a diagonal round.

To summarize, the following quarter-rounds make up one column round:

quarterround(y_0, y_4, y_8, y_{12})
quarterround(y_1, y_5, y_9, y_{13})
quarterround(y_2, y_6, y_{10}, y_{14})
quarterround(y_3, y_7, y_{11}, y_{15})

whereas the following quarterrounds make up one diagonal round:

quarterround(y_0, y_5, y_{10}, y_{15})
quarterround(y_1, y_6, y_{11}, y_{12})
quarterround(y_2, y_7, y_8, y_{13})
quarterround(y_3, y_4, y_9, y_{14})

After a set amount of rounds (8, 12, or 20), the input vector is added to the vector on which the rounds were run. Then the byte order of each matrix entry is reversed using the little-endian function.

Having explained the basic structure of the ChaCha hash function, the following pseudo-code should complete the readers comprehension of it:

```
chachahash(y):
    z = copy(y)
    for(i = 0; i < ROUNDS; i += 2) {
        // column round
        y[0], y[4], y[8], y[12] = quarterround(y[0], y[4], y[8], y[12])
        y[1], y[5], y[9], y[13] = quarterround(y[1], y[5], y[9], y[13])
        y[2], y[6], y[10], y[14] = quarterround(y[2], y[6], y[10], y[14])
        y[3], y[7], y[11], y[15] = quarterround(y[3], y[7], y[11], y[15])
        // diagonal round
        y[0], y[5], y[10], y[15] = quarterround(y[0], y[5], y[10], y[15])
        y[1], y[6], y[11], y[12] = quarterround(y[0], y[5], y[10], y[15])
        y[2], y[7], y[8], y[13] = quarterround(y[0], y[5], y[10], y[15])
        y[3], y[4], y[9], y[14] = quarterround(y[0], y[5], y[10], y[15])
    }
    for(i = 0; i < 16; i += 1) {
        y[i] += z[i]
        y[i] = littleendian(y[i])
    }
    return y
```

3.4 ChaCha State Matrix

ChaCha internally uses a 512-bit state for keystream generation. The ChaCha hash function modifies this state to generate a keystream block.
This section will explain how the state is setup. It is then passed in 32-bit blocks to the ChaCha hash function.

The state is made up of a 128-bit constant, a 256-bit key section and a 128-bit nonce section. To demonstrate that Bernstein had no hidden intent with picking his constants (nothing-up-my-sleeve number), he defined the constants to be "expand 16-byte k" for 128-bit keys and "expand 32-byte k" for 256-bit keys in ASCII.

In the 128-bit key variant, the key is concatenated with itself to create a 256-bit key. This concatenated key is then used for the state setup. If the key is already 256-bit, we do nothing and just use it as it is for the state setup.

The nonce section, which consists of a block counter and the initialization vector, is where the original version and the IETF version differ. In the original version, a 64-bit counter and a 64-bit initialization vector is used whereas the IETF version is using a 32-bit counter and a 96-bit initialization vector.
This means that the IETF version only partitions the nonce differently. Their reasoning to have a longer initialization vector was that with a 32-bit counter, one can encrypt messages up to 256 GiB which should be enough and therefore one could make use of a bigger initialization vector. Since we need to make sure that an initialization vector is never reused with the same key, we can use a bigger initialization vector to make it more secure in cases where the same key is used by multiple senders. This is done by partitioning the 96-bit word into one 32-bit and one 64-bit section. The 32-bit section could be a unique value per sender and the last 64 bits could be a counter which is incremented for every message [Nir+18].

All state parameters are encoded by first splitting them into 32-bit blocks whose byte order is reversed except for the counter, whose byte order is first completely reversed and afterwards split into 32-bit blocks. These 32-bit blocks are then ordered as follows to form the 512-bit state matrix:

$$\begin{pmatrix} \texttt{CONSTANT} & \texttt{CONSTANT} & \texttt{CONSTANT} & \texttt{CONSTANT} \\ \texttt{KEY} & \texttt{KEY} & \texttt{KEY} & \texttt{KEY} \\ \texttt{KEY} & \texttt{KEY} & \texttt{KEY} & \texttt{KEY} \\ \texttt{COUNTER} & \texttt{COUNTER/IV} & \texttt{IV} & \texttt{IV} \end{pmatrix}$$

Example. State parameters (all numbers are in hexadecimal):

```
key (256-bit)  01:02:03:04 05:06:07:08 09:0a:0b:0c 0d:0e:0f:10
               11:12:13:14 15:16:17:18 19:1a:1b:1c 1d:1e:1f:20
IV             00:11:22:33 44:55:66:77
Counter        00:00:00:00 00:00:00:01
```

Since we used a 256-bit key, we will use the ASCII constants "expand 32-byte k". Their byte representation is:

```
Constants      65:78:70:61 6e:64:20:33 32:2d:62:79 74:65:20:6b
```

The resulting state matrix:

$$\begin{pmatrix} 61{:}70{:}78{:}65 & 33{:}20{:}64{:}6e & 79{:}62{:}2d{:}32 & 6b{:}20{:}65{:}74 \\ 04{:}03{:}02{:}01 & 08{:}07{:}06{:}05 & 0c{:}0b{:}0a{:}09 & 10{:}0f{:}0e{:}0d \\ 14{:}13{:}12{:}11 & 18{:}17{:}16{:}15 & 1c{:}1b{:}1a{:}19 & 20{:}1f{:}1e{:}1d \\ 00{:}00{:}00{:}01 & 00{:}00{:}00{:}00 & 33{:}22{:}11{:}00 & 77{:}66{:}55{:}44 \end{pmatrix}$$

3.5 Encryption/Decryption

To encrypt or decrypt a input text, it is XOR'ed with the keystream.
To generate the keystream, the ChaCha hash function is continuously used to create 512-bit keystream blocks until we have enough to XOR every byte of the input text. The input to the ChaCha hash function is the 512-bit initial state as explained in the previous section. After each keystream block, the counter is incremented to have a different initial state as the input each time.

Since we are operating on streams, if the input message is not a multiple of 512-bit, the bits of the last block of the input message are left-aligned and the remaining bits of the keystream are dropped. This means that the output will always be the exact same length as the input.

There is no difference between encryption or decryption because XOR is the inverse to itself.

4 Plug-in

This chapter describes the goals of the plug-in and how the implementation accommodates these goals. It also includes the thought process behind some architectural decisions that were made.

At the end, alternative solutions that were taken into consideration but eventually abandoned are described. They complement the reasoning about the final architecture.

4.1 Goals

This section lists which goals the plug-in should meet. Each goal will be marked with a number which will later be reused in Section 5.1 to summarize how the goals were met.

(*G*1) **Easy-to-understand visualization of the encryption process**
The main goal of the plug-in was to teach students how the ChaCha cipher family encrypts messages. Therefore, the visualization should be easy to follow without much prior knowledge about ciphers.

(*G*2) **Visualization of the diffusion property**
To get a better understanding how the cipher hides the relationship between the ciphertext and the plaintext, the plug-in should contain a visualization of the diffusion property. To achieve this, the user should be able to flip bits of the input values.

(*G*3) **Support for all variants of the cipher family**
The plug-in should support 128-bit and 256-bit keys and the default 64-bit counter and 64-bit initialization vector. Since the Internet Engineering Task Force (IETF) introduced a slightly modified version of the cipher which has a 32-bit counter and a 96-bit initialization vector, the plug-in should also support these values. Finally, the user should also be able to choose between 8, 12 or 20 rounds.

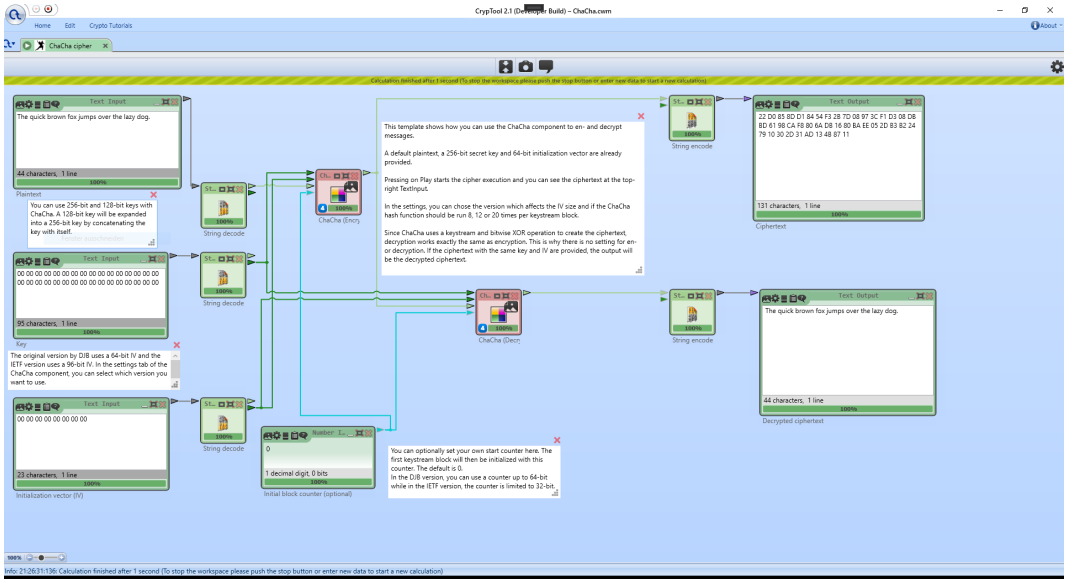


Figure 4.1: CT2 template for the ChaCha plug-in

4.2 Implementation Details

This section details how the goals described in the previous section were achieved. In the first subsection, the key features of the plug-in are described to give a rough overview what a user can expect from the actual plug-in implementation.
The second subsection will then explain the user interface which is used to navigate through the plug-in and communicate to the user what is currently happening inside the cipher.
The last subsection then goes into technical details to explain how the plug-in internally was designed to make the user interface behave as it does.

4.2.1 Key Features

The plug-in offers the user the ability to input his own plaintext, key, initialization vector and initial counter using the concept of visual programming (around which CT2 is built) as one can see in Figure 4.1. The counter is optional and defaults to zero. These values are then used to visualize the cipher execution comprehensively. Descriptions complement the visualization by providing information about what is happening.
As mentioned in Section 4.1, if the user wants to see the diffusion property of the cipher, he can alter the key, initialization vector and counter on a dedicated page

inside the plug-in.
The version (original DJB version or IETF version) and how often the ChaCha hash function should be run per keystream block can be chosen in the plug-in settings.

4.2.2 User Interface

This subsection will describe the layout and functionality of the user interface and the reasoning behind it. First, the parts of the user interface which all pages have in common will be explained. Afterwards, we will expand upon the interface differences between the individual pages.

General interface structure

All pages have a common interface layout which consists of three sections. Each section is inside an own dedicated row.
The first section implements the navigation to move between pages. It also shows the title of the current page.
The second section shows the content of the current page. The content is not always static since it can change by using the action navigation bar in the bottom section. This navigation bar includes buttons to go to the previous or next action, a slider for quicker action navigation, a text input to go to a given action and a indicator on which action the user currently is and how many actions there are in total on the current page. In Figure 4.4, you can see this navigation bar for the first time.

A slider was chosen over alternative solutions like individual buttons because a slider enables the user to quickly navigate through a page. This advantage is best noticeable on the page about the ChaCha hash function where there are more than 3000 actions per keystream block. Fitting more than 3000 buttons on a single page was not feasible but would need pagination features such as showing the next set amount of buttons which would make quick navigation impossible.
The user can also enter a number into the text input to the right of the slider to go to an action. But this text input is only more helpful than the slider if the user already knows the number of the action he wants to go to. If he does not, the slider creates a much better user experience.
If a page does not have actions, the action navigation is hidden but the space is still reserved for it to have a consistent canvas size for all pages.

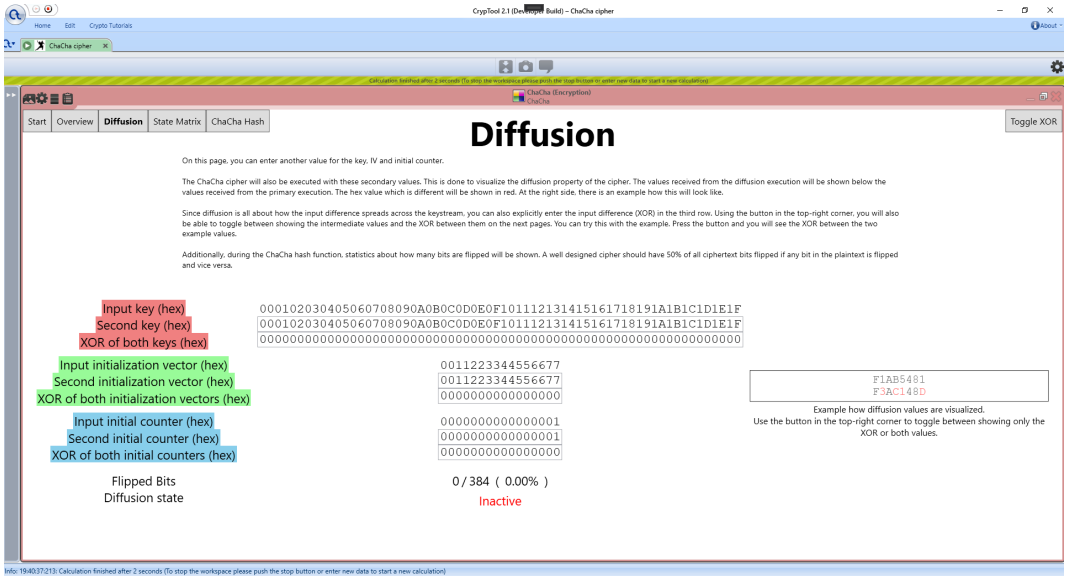


Figure 4.2: Diffusion page in its initial state

What follows are sections which go into greater detail about each page which does not have fully static content. These pages are the Diffusion page, the State Matrix Initialization page and the ChaCha Hash Function page.

Diffusion page

The Diffusion page (Figure 4.2) is the dedicated page, as mentioned in Section 4.2.1, where the user can alter the key, initialization vector (IV) and initial counter in hexadecimal. They will be used to visualize the diffusion property. This means that during cipher execution, hexadecimal letters which are different are marked red. An example of this is shown on the Diffusion page at the right side.

Hexadecimal text inputs were chosen because alternatives like individual checkboxes for each bit (like in the DES visualization) or binary text inputs would have taken up a lot of canvas size because the key, counter and IV could be together 384-bit (if using a 256-bit key). This lost canvas size would have taken away the canvas size for other features like the example or would have made the page very crowded.

Unfortunately, using hexadecimal input fields made it harder to flip specific bits. This feature was necessary since when studying the diffusion property of a cipher, one is more interested in the difference of two values (XOR) during two cipher runs instead of the concrete values. The solution for this was to introduce a third input

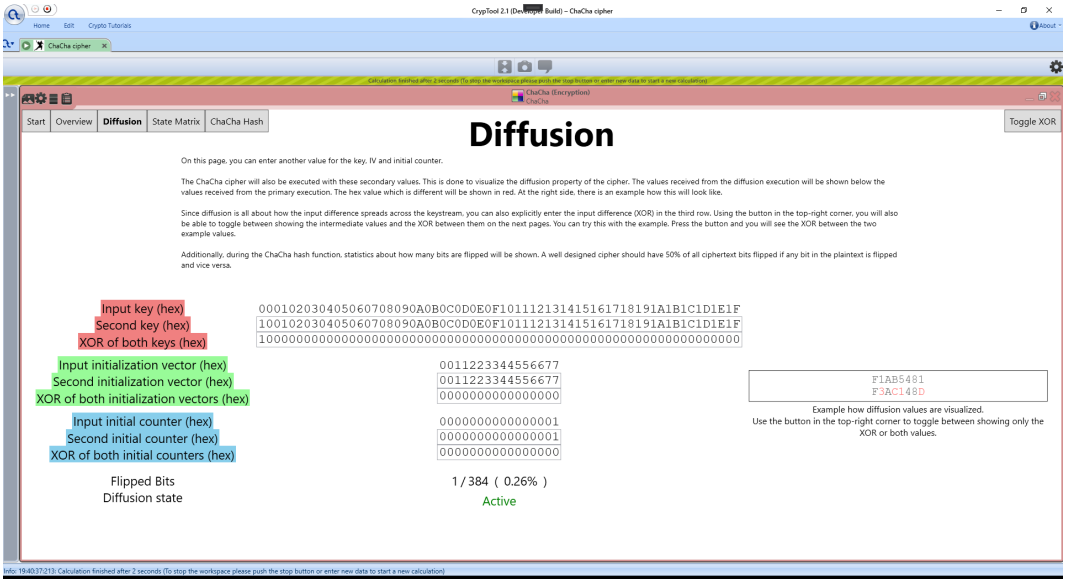


Figure 4.3: Diffusion page in its active state

field where the user can explicitly input the XOR between the input value and the altered value. Additionally, if diffusion is active, the user can toggle if he wants to see the XOR and both concrete values during the cipher execution by using the button in the top-right corner of each page.

To help with the input, the input fields show an error message if the user entered invalid characters or a too large string since the hex strings for each value must be of equal size as the input value. If the hex string is too short, it gets left-padded with zeroes to align it with the primary value.

Last but not least, the amount of flipped bits is shown in the second to last row together with a state indication in the last row if the diffusion is active or not. The diffusion is active if at least one bit was flipped. This is shown in Figure 4.3.

State Matrix Initialization page

The State Matrix Initialization page shows the setup of the initial 512-bit state for the first keystream block.

In Figure 4.4, you can see the page in its initial state. In the top-left, you see the state. It is empty at the beginning because the initialization has not started yet. On the top-right, you see descriptions which inform the user about what the next steps are. At the bottom-left, the encoding will be visualized. At the bottom-right, the state parameters with their original values before encoding are shown. At the

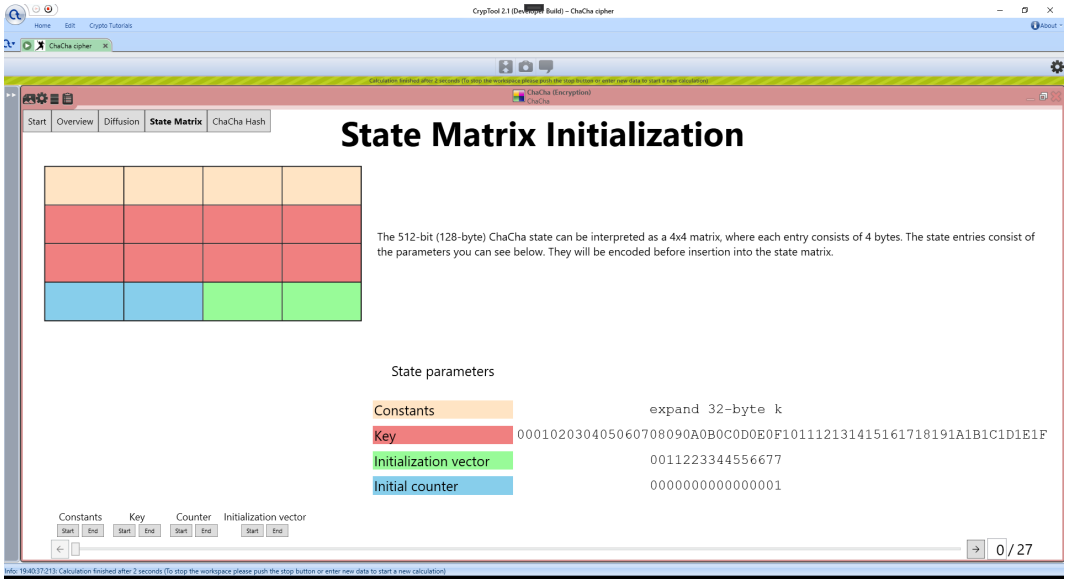


Figure 4.4: State Matrix Initialization page in its initial state

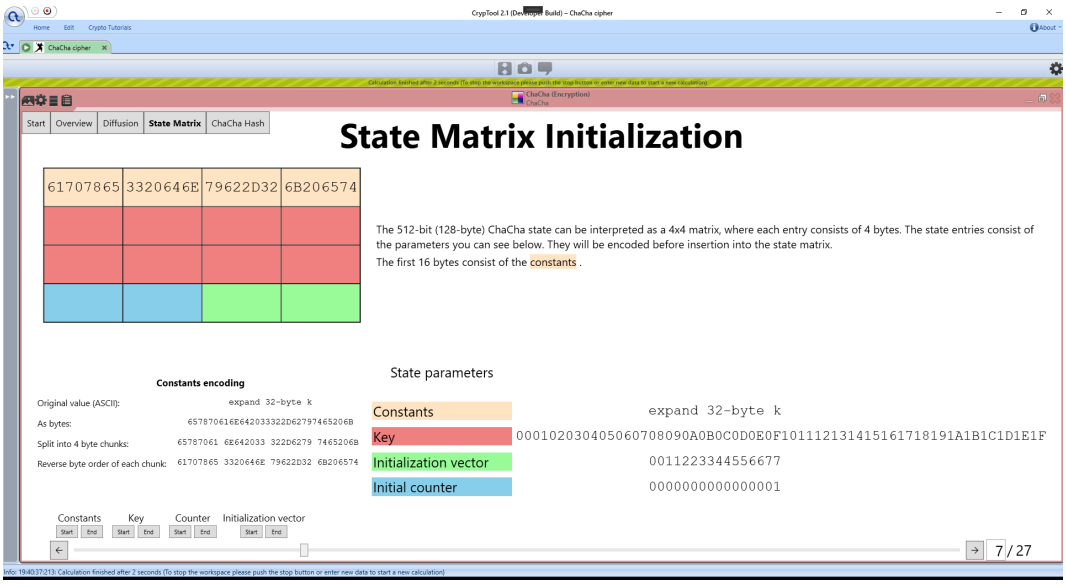
bottom above the slider are individual buttons to go to the start and end of each state parameter encoding.

Only the construction of the state for the very first keystream block is shown on this page. This was done like this because interrupting the page flow by jumping back to the visualization of the state matrix initialization after each keystream block was not reasonable. Further, only the counter would be different for each state so setting up the state again would introduce a lot of noise to the visualization. The information which is provided to the user in the first and only state matrix initialization should be enough for him to construct all following initial states without further guidance. To achieve this, the focus for this page was on developing a comprehensive visualization of the encoding for each state parameter (constants, key, IV, counter).

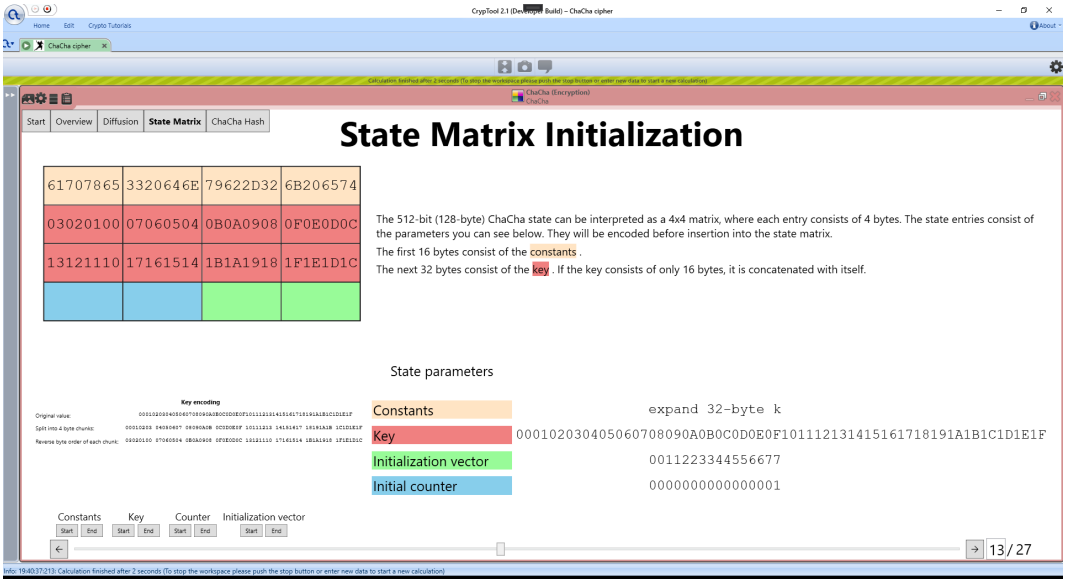
In Figure 4.5, you can see the page state during the end of each state parameter encoding. For each encoding step which corresponds to a row in the encoding section, a page action has been implemented. This means that the user can follow along each encoding step-by-step in his own speed and is not overwhelmed by a lot of information.

First, the constants are encoded. Since they are shown in ASCII format in the state parameters section, they are first decoded to show the actual byte values. Then the bytes are split into 4 byte chunks whose order is then reversed in the last step.

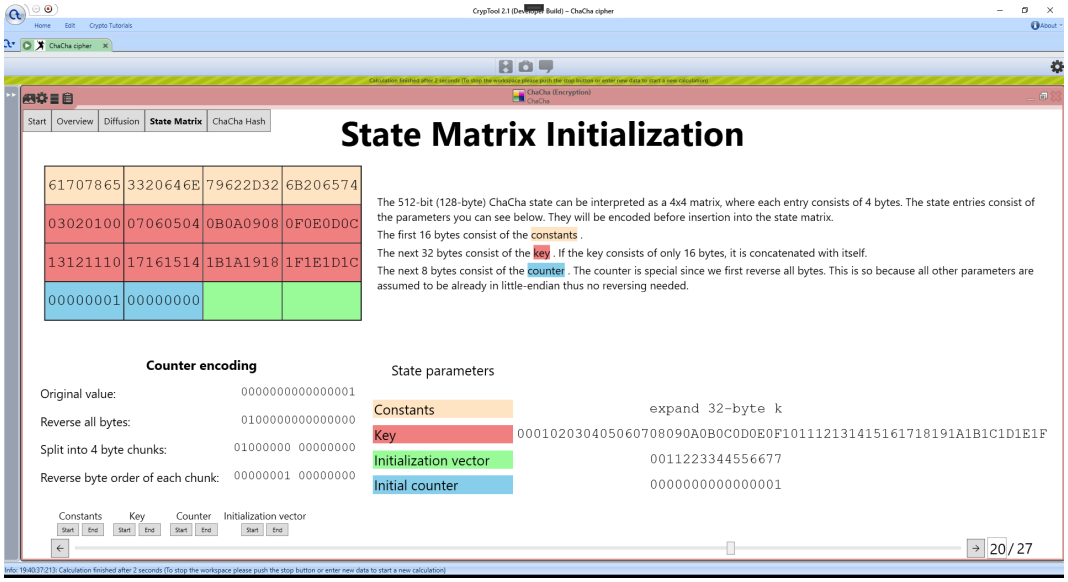
Afterwards, the key is encoded. It is just split into 4 byte chunks whose order is then reversed.
When encoding the counter, the complete byte order is first reversed, and then it is split into 4 byte chunks whose byte order is then again reversed.
The last parameter is the IV. It is encoded exactly the same as the key.



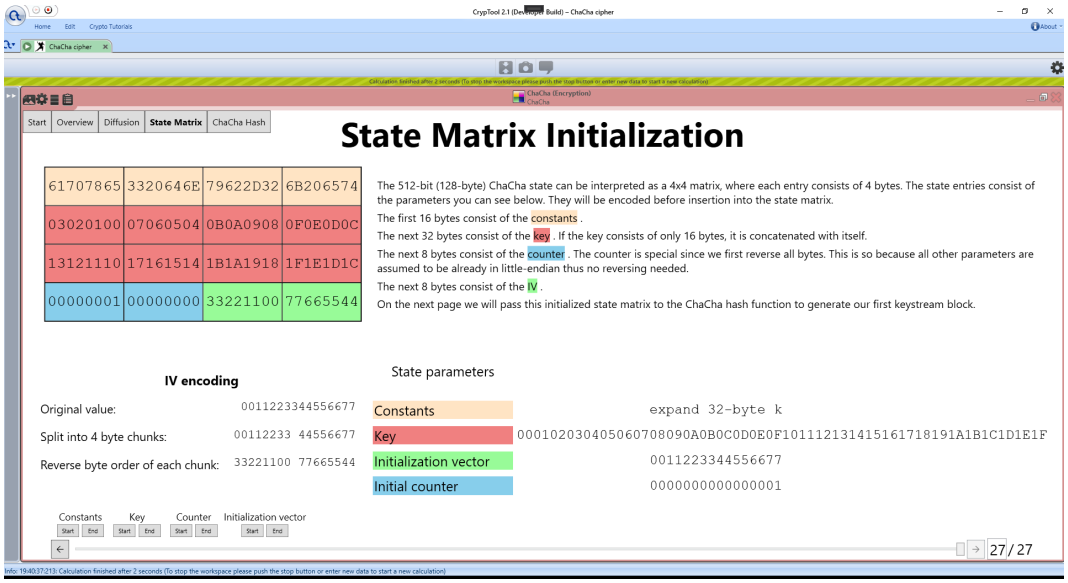
(a) Constants encoding



(b) Key encoding



(c) Counter encoding



(d) IV encoding

Figure 4.5: Encoding of the state parameters on the State Matrix Initialization page

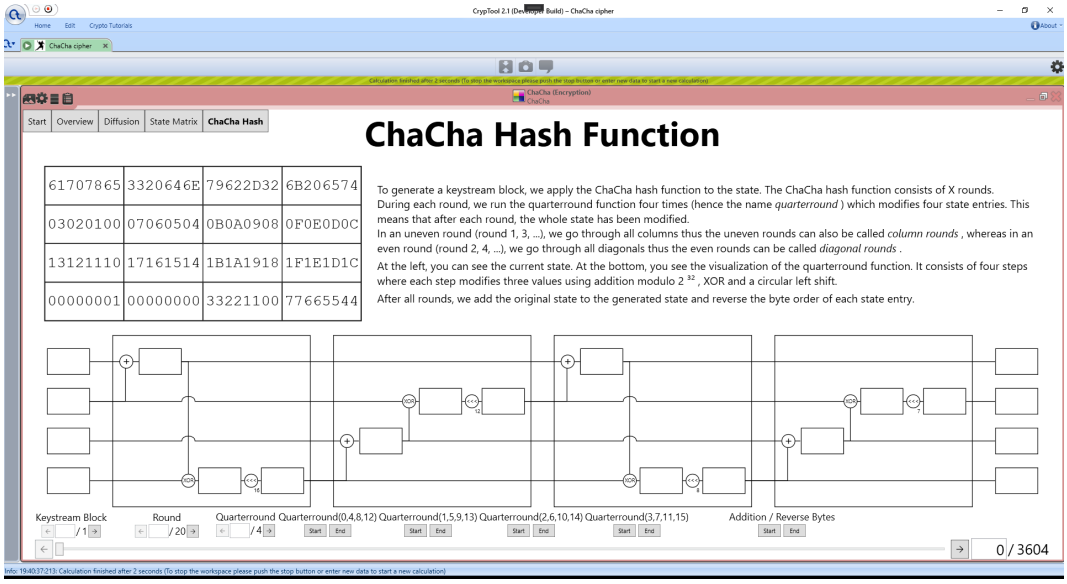


Figure 4.6: ChaCha Hash Function page in its initial state

ChaCha Hash Function page

The ChaCha Hash Function page (Figure 4.6) visualizes the generation of a keystream block by using two kind of visualizations: The quarter-round visualization (Figure 4.7) and the addition / reverse bytes step visualization at the end of the hash function (Figure 4.9).

At the top of the page, you can see the current state together with a description about the ChaCha hash function. At the bottom, if we are still in the ChaCha hash function loop (as described in Section 3.3), you can see the visualization of the quarter-rounds (Figure 4.7). The quarter-round visualization is split into four cells on either side and four boxes in the middle. The boxes on the two ends contain the four input and output values of the quarter-round. The boxes in the middle visualize one quarter-round step as was described in Section 3.1.

A circuit diagram was used to visualize the quarter-round function similar to the one found in the ChaCha section on the Wikipedia page about Salsa20 (see Figure 4.8). This way, the intermediate values could just be put on the circuit lines and following along the visualization was very clear because all next steps are already shown.

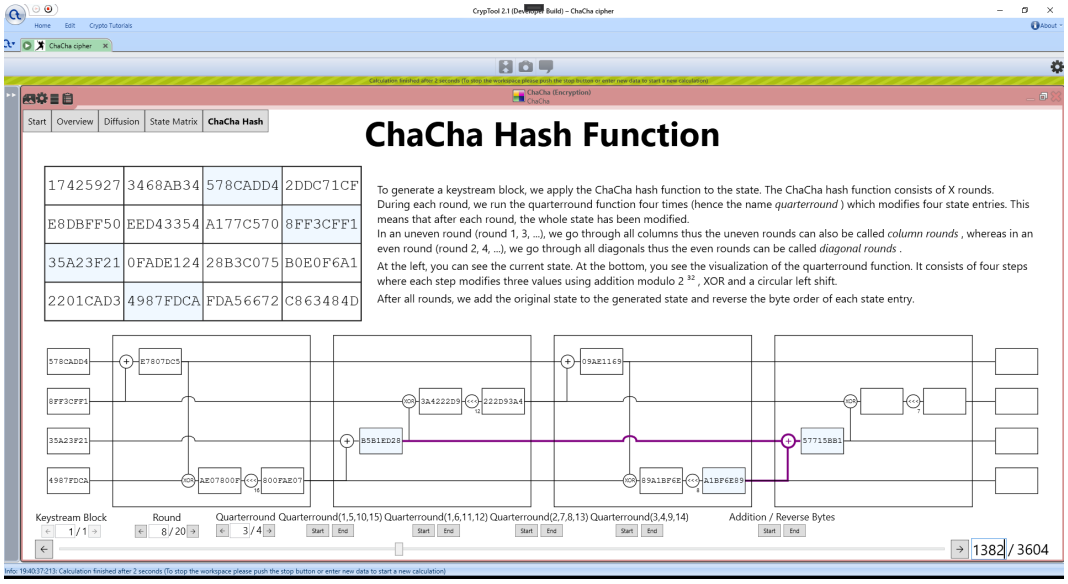
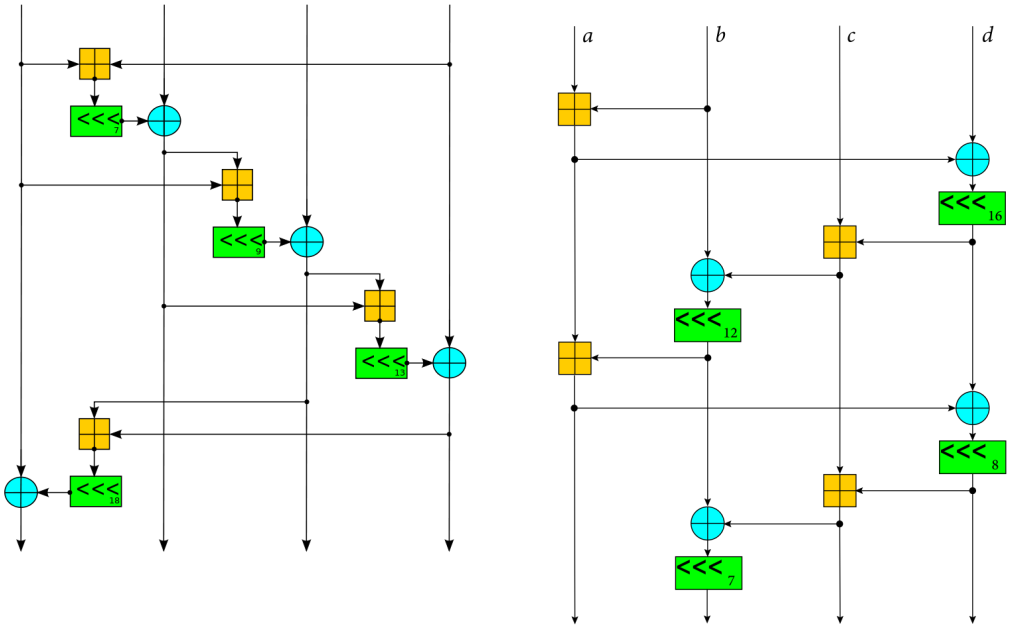


Figure 4.7: Quarter-round visualization

If we are finished with the loop, we can see the original state at the left, the result of adding the original state with the state after all rounds in the middle, and at the right the result of reversing the byte order of each state entry of the state in the middle (Figure 4.9). This is the final state and thus one 512-bit block inside the keystream with which the input message is later XOR'ed .

Below the visualization and just above the slider for the page actions is another navigation bar. This helps the user to quickly navigate through the ChaCha hash function. He can use the arrow buttons to traverse through the keystream blocks, rounds or quarter-rounds or enter a number to directly go to a specific step.
Since entering a number into each text input or using the arrows will only bring the user to the start of each step, he can use the buttons to the right of the quarter-round input to jump to the end of specific quarter-rounds of the current round.
These buttons also show the state indices that will get updated during their quarter-round in parentheses. Since column and diagonal rounds take turns, these labels show the state indices according to the current round.

Figure 4.10 shows exemplary how the page looks like at the end of each quarter-round of a diagonal round. As you can see, the corresponding state entry is highlighted with a light blue background. This light blue background is also used throughout the visualization to catch the user's attention about which UI elements will be updated in the next step.



(a) Salsa quarter-round circuit diagram **(b)** ChaCha quarter-round circuit diagram

Figure 4.8: Quarter-round circuit diagram
Source:
(a): `https://en.wikipedia.org/wiki/Salsa20`
(b): `https://en.wikipedia.org/wiki/Salsa20#ChaCha_variant`

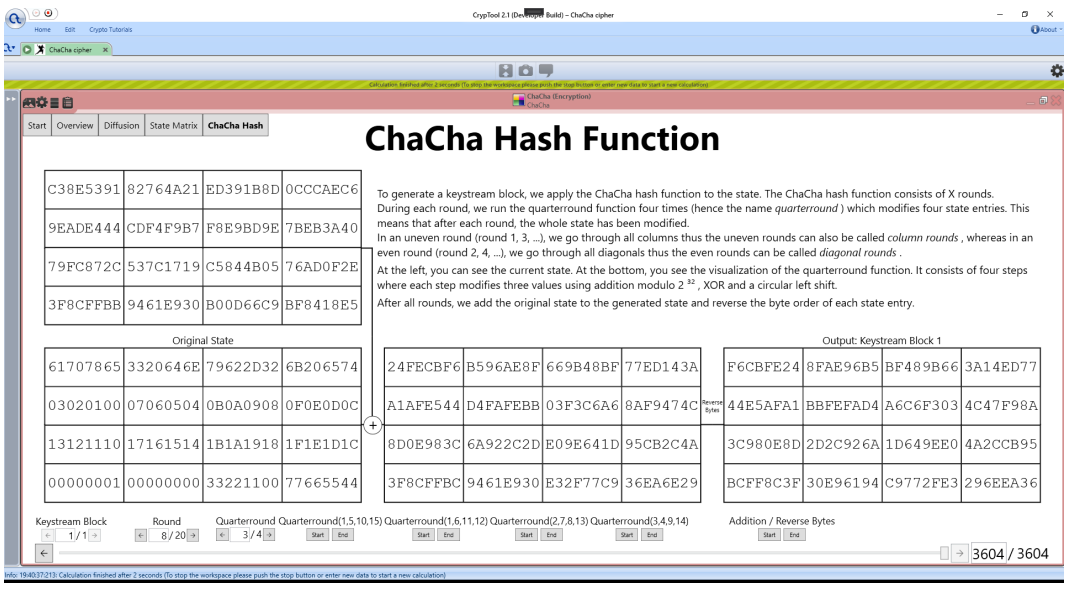
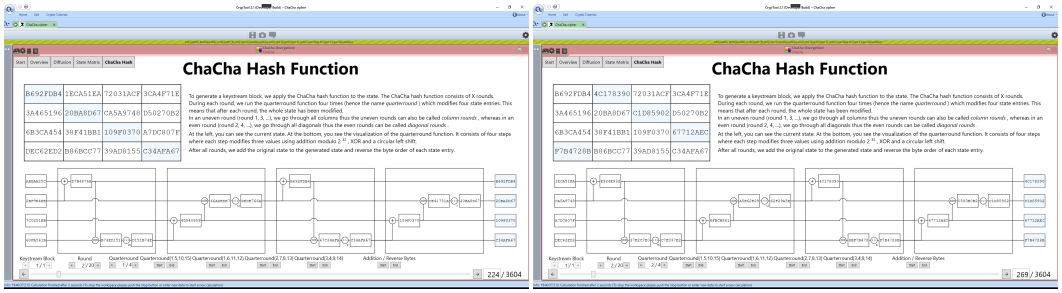
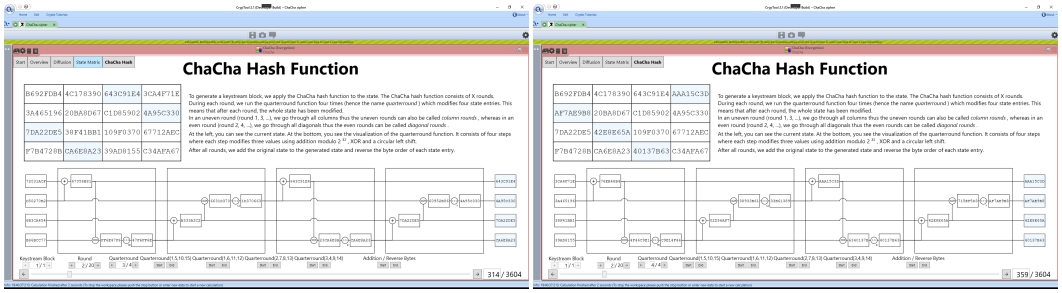


Figure 4.9: Addition and little-endian step visualization



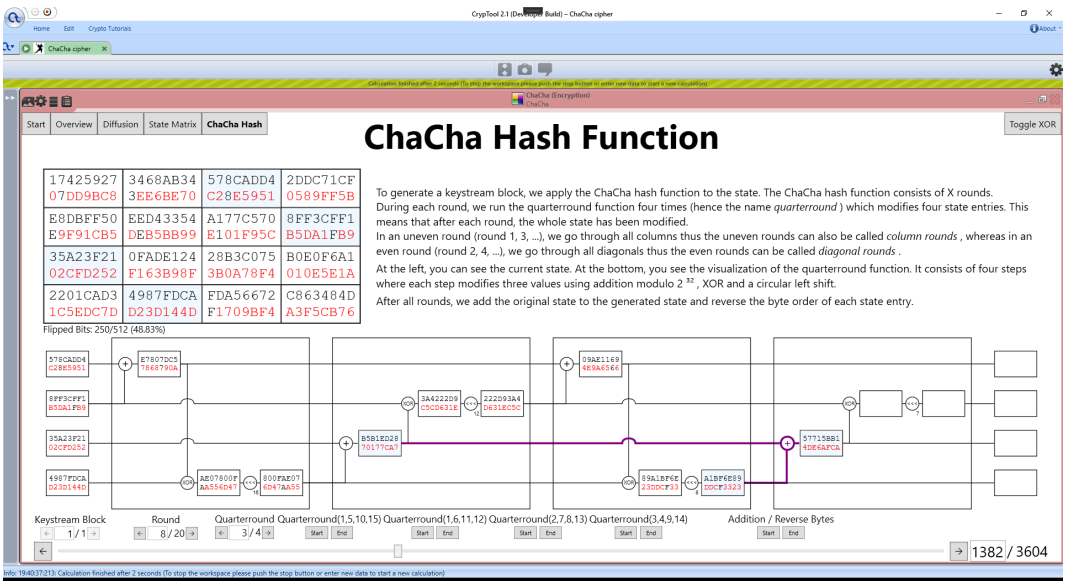
(a) End of first quarter-round (diagonal round) **(b)** End of second quarter-round (diagonal round)



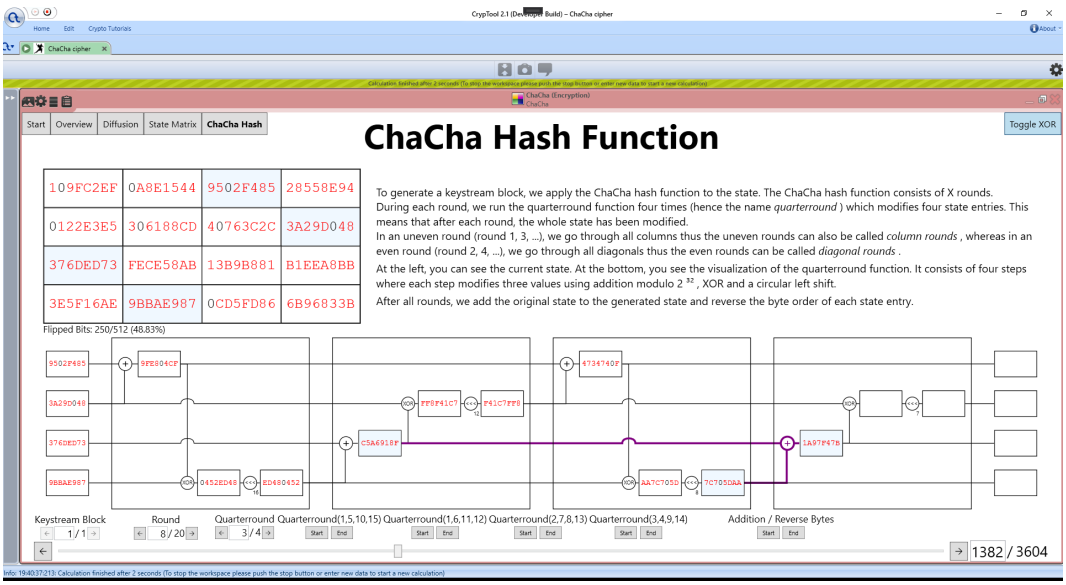
(c) End of third quarter-round (diagonal round) **(d)** End of fourth quarter-round (diagonal round)

Figure 4.10: End of each quarterround execution of a diagonal round

Figure 4.11 shows the quarter-round visualization as an example how diffusion is visualized throughout the plug-in. Figure 4.11a shows the visualization with the XOR button in the top-right corner not toggled; thus showing the values from both cipher runs whereas Figure 4.11b shows the visualization with the XOR button toggled.



(a) Quarter-round visualization with diffusion (showing both values)



(b) Quarter-round visualization with diffusion (showing XOR)

Figure 4.11: Quarter-round visualization with diffusion

4.2.3 Architecture

This section will go more into detail how the software was layed out.
The software architecture, which was written using WPF (Windows Presentation Foundation), XAML and C#7.0, can be split into two parts.
The first part is about the Model-View-ViewModel (MVVM) architecture to create the user interface which was explained in the previous section, using WPF built-in tools such as data binding, templates, converters and validation rules.
The second part is about the action navigation system which plays a huge role regarding performance. It powers the slider and buttons in the bottom row of each page which has actions. The page navigation is handled by the first part because it is very simple and uses MVVM design patterns.

MVVM architecture

The architecture was built with the MVVM design pattern in mind. As the name suggest, MVVM is all about separating the code into three parts: Models, Views and View Models.

Models hold the raw application data. In our case, this would be the classes which hold the values generated by the ChaCha cipher. It should be completely unaware of any view or view Model.

Views define how the data should be presented. They consist mainly of XAML code with as little code-behind as possible. They do not maintain their own state but rather use data binding to synchronize themselves with the data inside view models. Therefore, views are aware of view models and in fact depend on them to show any relevant data.

View models connect the model data with the views. However, they do not directly manipulate the views but just define properties and methods which implement the logic for user interactions. The views then use data binding to be notified about any property changes or call methods if a button is clicked etc. This means that view models should not rely on any code inside a view. This essentially decouples the backend from the frontend.

In Figure 4.12, an example for how this data binding looks in code is provided. What you see is the view code for the page navigation.

Another technique (that you can also see in the mentioned figure) is the use of templates. Data templates are used to attach views to view models and control templates to implement the general UI structure with the three sections mentioned

```
<ItemsControl ItemsSource="{Binding Pages}">
    <ItemsControl.ItemsPanel>
        <ItemsPanelTemplate>
            <StackPanel Orientation="Horizontal" />
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <Viewbox Style="{StaticResource UniformViewbox}">
                <Button
                    Command="{Binding DataContext.ChangePageCommand, RelativeSource={RelativeSource AncestorType={x:Type UserControl}}}"
                    CommandParameter="{Binding .}"
                    IsEnabled="{Binding DataContext.NavigationEnabled, RelativeSource={RelativeSource AncestorType={x:Type UserControl}}}">
                    <Label Content="{Binding Name, Mode=OneWay}">
                        <Label.Style>
                            <Style TargetType="{x:Type Label}">
                                <Setter Property="FontWeight" Value="Normal" />
                                <Style.Triggers>
                                    <DataTrigger Value="True">
                                        <DataTrigger.Binding>
                                            <MultiBinding Converter="{StaticResource CompareStrings}">
                                                <Binding
                                                    RelativeSource="{RelativeSource AncestorType={x:Type UserControl}}"
                                                    Path="DataContext.CurrentPage.Name"
                                                    Mode="OneWay" />
                                                <Binding Path="Name" Mode="OneWay" />
                                            </MultiBinding>
                                        </DataTrigger.Binding>
                                        <Setter Property="FontWeight" Value="Bold" />
                                    </DataTrigger>
                                </Style.Triggers>
                            </Style>
                        </Label.Style>
                    </Label>
                </Button>
            </Viewbox>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

Figure 4.12: Implementation of page navigation using data binding

```
<ControlTemplate x:Key="PageContentTemplate" TargetType="{x:Type ContentControl}">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="90*" />
            <RowDefinition Height="5*" />
        </Grid.RowDefinitions>
        <Grid Grid.Row="0">
            <ContentPresenter />
        </Grid>
        <Grid Grid.Row="1">
            <Viewbox Style="{StaticResource UniformViewbox}">
                <StackPanel
                    Orientation="Horizontal"
                    Visibility="{Binding HasActions, Converter={StaticResource FalseToCollapsed}, FallbackValue={x:Static Visibility.Collapsed}}">
                    <Button Command="{Binding PrevActionCommand}" Style="{StaticResource PrevButton}" />
                    <Slider
                        Margin="3,0,3,0"
                        VerticalAlignment="Center"
                        Minimum="0"
                        Maximum="{Binding TotalActions, Converter={StaticResource SubtractOne}}"
                        Width="1200"
                        Value="{Binding CurrentActionIndex, Mode=OneWay}"
                        Name="ActionSlider"
                        TickFrequency="1"
                        IsSnapToTickEnabled="True" />
                    <Button Command="{Binding NextActionCommand}" Style="{StaticResource NextButton}" />
                    <TextBox
                        Name="ActionInputTextBox"
                        Style="{StaticResource Input}"
                        Width="{Binding ElementName=TotalActionsText, Path=ActualWidth}" />
                    <TextBox Style="{StaticResource Text}" Text="/" />
                    <TextBox
                        Name="TotalActionsText"
                        Style="{StaticResource Text}"
                        Text="{Binding TotalActions, Mode=OneWay, Converter={StaticResource SubtractOne}}" />
                </StackPanel>
            </Viewbox>
        </Grid>
    </Grid>
</ControlTemplate>
```

Figure 4.13: Implementation of action navigation inside a control template

in Section 4.2.2. This makes it possible to define the UI layout for all pages in one place and also reuse a single implementation of the page and action navigation across all pages. This way, view modifications down the road which applied to all pages were easy and fast. In Figure 4.13, you can see the view implementation of the action navigation bar which is only visible on pages which do have actions.

Since the most interesting part about the MVVM architecture is probably how the diffusion visualization works, this feature will be explained more in-depth in the following.

Diffusion implementation using MVVM

If the user enters something into the input fields of the Diffusion page, validation of the input occurs. The validation checks if the input contains only valid hexadecimal characters and if the size is not too large. This is done by using two-way bindings and extending the `ValidationRule` class which is a built-in module of WPF. Only if the input is valid, the value is saved into a property of the underlying view model. This way, we can be sure that we always have sane data with which we later can execute the cipher with.

Furthermore, converters which are also built into WPF are used to convert data into user-friendly formats. Converters are always attached to bindings, so inside the two-way bindings, byte arrays are converted into hex strings and vice versa using custom converters.

Data binding works by notifying a view if a variable has changed. WPF provides an interface named `INotifyPropertyChanged` which helps to implement these data binding notifications. It provides a method named `OnPropertyChanged` and an event named `PropertyChangedEventHandler` which must be called with the name of the variable to raise such an notification. The data binding system then updates the variables in the view.

The notification raising and the implementation of the `INotifyPropertyChanged` interface can be seen in Figure 4.14. The setter of `DiffusionInputKey` raises multiple notifications because other variables which depend on `DiffusionInputKey` also need updating. (The first call of `OnPropertyChanged` in the setter has no argument because the attribute `CallerMemberName` in the function signature automatically enters the name of the caller if no argument is provided.)

On page exit, `ChaChaPresentationViewModel`, which is the view model which implements navigation between the different pages (which have their own view mod-

```
private byte[] _diffusionKey; public byte[] DiffusionInputKey
{
    get
    {
        return _diffusionKey;
    }
    set
    {
        if (_diffusionKey != value)
        {
            _diffusionKey = value;
            OnPropertyChanged();
            OnPropertyChanged("DiffusionActive");
            OnPropertyChanged("FlippedBits");
            OnPropertyChanged("FlippedBitsPercentage");
        }
    }
}
```

(a) Data binding notification in the setter of `DiffusionInputKey`

```
internal abstract class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public virtual void OnPropertyChanged([CallerMemberName] string propertyName = "")
    {
        PropertyChangedEventHandler handler = this.PropertyChanged;
        if (handler != null)
        {
            var e = new PropertyChangedEventArgs(propertyName);
            handler(this, e);
        }
    }
}
```

(b) `ViewModelBase` implementation with `INotifyPropertyChanged` interface

Figure 4.14: Data binding notification implementation

els), calls `Teardown` on the page we are leaving. This triggers the ChaCha execution with the given altered values.

Before execution, a flag was set which instructs the list getters into which the ChaCha component would save its intermediate values to return a different list. This means that the ChaCha component is agnostic of where it ultimately saves its intermediate values. During the execution, the intermediate values for the diffusion run are therefore saved in parallel lists. This is shown exemplary for the quarter-round input values in Figure 4.15. The other pages then simply bind to these lists to display the intermediate values of the diffusion run. Converting the

```
public class ChaCha : ICrypComponent, IValidatableObject, INotifyPropertyChanged
{
    // ...

    private (uint, uint, uint, uint) Quarterround(uint a, uint b, uint c, uint d)
    {
        QRInput.Add((a, b, c, d));
        (a, b, d) = QuarterroundStep(a, b, d, 16);
        (c, d, b) = QuarterroundStep(c, d, b, 12);
        (a, b, d) = QuarterroundStep(a, b, d, 8);
        (c, d, b) = QuarterroundStep(c, d, b, 7);
        QROutput.Add((a, b, c, d));
        return (a, b, c, d);
    }

    // ...

    private List<(uint, uint, uint, uint)> _qrInputDiffusion; public List<(uint, uint, uint, uint)> QRInputDiffusion
    {
        get
        {
            if (_qrInputDiffusion == null) _qrInputDiffusion = new List<(uint, uint, uint, uint)>();
            return _qrInputDiffusion;
        }
    }

    private List<(uint, uint, uint, uint)> _qrInput; public List<(uint, uint, uint, uint)> QRInput
    {
        get
        {
            if (DiffusionExecution) return QRInputDiffusion;
            if (_qrInput == null) _qrInput = new List<(uint, uint, uint, uint)>();
            return _qrInput;
        }
    }

    // ...

    public bool DiffusionExecution { get; set; }

    public void ExecuteDiffusion(byte[] diffusionKey, byte[] diffusionIv, ulong diffusionInitialCounter)
    {
        DiffusionExecution = true;
        Xcrypt(diffusionKey, diffusionIv, diffusionInitialCounter, (ChaChaSettings)Settings, InputStream, new CStreamWriter());
        DiffusionExecution = false;
    }
}
```

Figure 4.15: Saving of intermediate values during ChaCha execution in lists

32-bit unsigned integers into hex strings and marking their characters which are different red is then done in the view models.

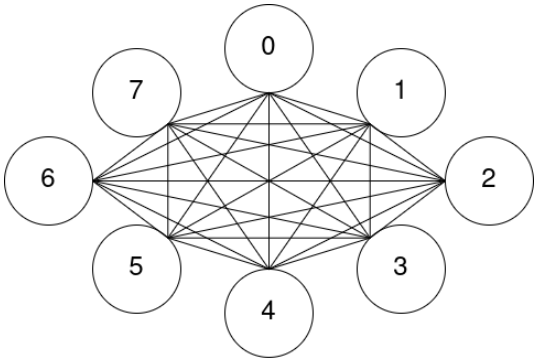


Figure 4.16: Navigation paths between actions if no reset state was used.

Centralized navigation system

The final design of the navigation system is characterized by reflecting on the problems previous iterations had. In this subsection, only the current, final implementation of the navigation system will be described. Previous implementations with their problems are described in Section 4.3.

The core idea of the navigation system was to decrease hops between actions without having too many possible transitions since for every transition from action A to another action B and vice versa, code needs to be written to perform the transition. Figure 4.16 shows how a navigation system would look like where the maximum amount of hops is minimized to one. Total transitions are $2n(n-1)$ and for every new action, transitions increase by $2(n-1)$ (n is the total number of actions). The factor 2 comes from the fact that for every transition from action A to B, a transition back from B to A is also needed. We can conclude that such an architecture does not scale very well from the perspective of a developer who needs to write all of that transitional code.
On the other hand, decreasing the amount of possible transitions by introducing more hops would degrade performance due to computational overhead, so a system with a very high average amount of hops would not scale very well regarding performance.

We came to the conclusion that the best trade-off would be to have a navigation system where every action has a transition to a centralized state and this central state then has a transition to all other actions. This would result in a maximum of two hops between any two actions. Further, adding a new action would only add two new transitions to the system. This system design is shown in Figure 4.17.

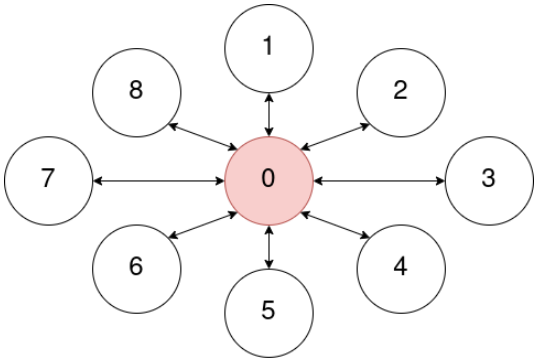


Figure 4.17: Navigation paths between actions in the centralized navigation system. The colored state is the reset state which corresponds to the initial state.

During implementation, it turned out that the amount of new transitions for each action could be decreased to one. Defining the initial state as the central state meant that transitioning from any action to it could be done by just resetting the whole page to the first action. Due to this, the central state can also be called the reset or initial state.

Unfortunately, the issue of code duplication quickly arised with this design. Since most of the time, the next page state was only a slight modification of the previous page state, the code for following actions was almost the same. To mitigate this issue, we started to think about the actions as *sequences*. A sequence of actions would mean that every action inside a sequence is an extension of all previous actions of the same sequence. Extension in this context means that if action A extends action B, action A contains at least all the code of action B. To use terminology of set theory, one could also say that A is a superset of B; viewing individual code statements as objects.

To implement this concept, an extension method for the `Action` type in C# was created together with an interface to create action sequences. Figure 4.18 shows the method which extends the built-in `Action` type.

This concept was further developed by introducing *nested sequences*. This means that a sequence could be started inside another sequence. A nested or child sequence would extend all the actions from its parent sequence. From the perspective of the nested sequence, it is no different than if all the actions from the parent sequence were created inside it. When ending a nested sequence, from the perspective of the parent sequence, the nested sequence never existed. Figure 4.19 shows a test which asserts that this nesting does indeed work as expected.

```
internal static class ActionExtensions
{
    /// <summary>
    /// Extension method to extend actions.
    /// The last parameter is the action from which we want to derive a new, extended action.
    /// This is why we call it first in the new returned action.
    /// </summary>
    public static Action Extend(this Action action, Action toExtend)
    {
        return () =>
        {
            toExtend.Invoke();
            action.Invoke();
        };
    }

    /// <summary>
    /// Syntactic sugar for Extend chaining.
    /// In other words: Action.Extend(x, y) is equivalent to Action.Extend(x).Extend(y)
    /// </summary>
    public static Action Extend(this Action action, params Action[] toExtend)
    {
        return toExtend.Aggregate(action, (acc, curr) => acc.Extend(curr));
    }
}
```

Figure 4.18: Extension method for the `Action` type to extend actions

```
[TestMethod]
public void TestNestedSequentialExecutionOrder()
{
    ActionCreator actionCreator = new ActionCreator();
    List<Action> Actions = new List<Action>();
    List<int> actual = new List<int>();

    actionCreator.StartSequence();
    Actions.Add(actionCreator.Sequential(() => { actual.Add(0); })); // Should add 0
    Actions.Add(actionCreator.Sequential(() => { actual.Add(1); })); // Should add 0, 1

    actionCreator.StartSequence();
    Actions.Add(actionCreator.Sequential(() => { actual.Add(2); })); // Should add 0, 1, 2
    Actions.Add(actionCreator.Sequential(() => { actual.Add(3); })); // Should add 0, 1, 2, 3
    actionCreator.EndSequence();

    Actions.Add(actionCreator.Sequential(() => { actual.Add(4); })); // Should add 0, 1, 4
    Actions.Add(actionCreator.Sequential(() => { actual.Add(5); })); // Should add 0, 1, 4, 5
    actionCreator.EndSequence();

    foreach (Action a in Actions)
    {
        a.Invoke();
    }

    List<int> expected = new List<int>(new int[] { 0, 0, 1, 0, 1, 2, 0, 1, 2, 3, 0, 1, 4, 0, 1, 4, 5 });

    CollectionAssert.AreEqual(expected, actual);
}
```

Figure 4.19: Test method for nested sequences

Nesting sequences was very helpful for implementing the actions for the ChaCha Hash Function page. Since each keystream block started with a cleared quarter-round visualization and the initial state visible in the state matrix, it made sense to declare a sequence for each keystream block. Similar was true for the beginning of each round and quarter-round. Therefore, the sequences for the rounds were nested inside the keystream block sequence and the sequences for the quarter-rounds were nested inside the round sequences. This meant that I would not introduce too much computational overhead inside the transitional code (even though only one transition was executed by design) since I could just reset the sequence if it made no longer sense to include the code of previous actions. This was the case when a (quarter-)round or keystream block was finished.

Being able to nest sequences essentially prevented the system degrading back to a linear navigation system as will be described in Section 4.3. The system would have less transitions but would still execute the same code as a linear navigation system would when moving from action 0 to any other action; undoing any positive effect this approach of minimizing transitions could have had.

The weakness of this design was that there was still some overhead when navigating through the page in a linear manner since the system is basically designed to always go back to the first action first and from there to the desired action. This leads to a lot of unnecessary resetting because as mentioned, most of the times the next page state is only a slight modification of the previous one. But this is the trade-off that was had to be made for a good overall performance. To summarize, this design had a higher overhead for small steps but did scale much better for bigger steps which were necessary for the keystream, round and quarter-round navigation.

The following section will make it clear why so much thought was put into the navigation system design. It will show why previous less thought-out architectures failed and includes plots about the performance of each architecture at the end.

4.3 Encountered Problems

This section will discuss the main problems that were encountered during implementation. To briefly summarize, they mainly consisted of how the system behind the interface should be designed to have the best or at least a reasonable performance.

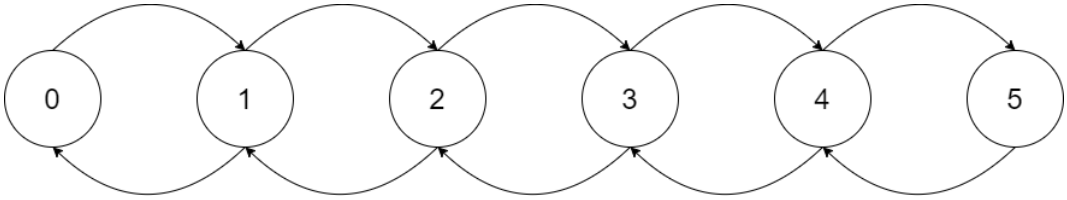


Figure 4.20: Navigation paths between actions in the linear navigation system

As described in Section 2.3.1, to enable a fluent navigation experience where the user can navigate very freely through the visualization (using backward and forward navigation), the intermediate values need to be pre-calculated. But as will be explained on the next pages, this was not all that was needed to ensure such an experience.

We hope that the description of these problems and the solutions we have found may help future students in writing their own plug-ins.

Linear navigation system

The performance of the plug-in was essentially coupled to how the navigation system was designed. Other things like the aforementioned calculation of the intermediate values for the visualization were compared to the navigation system design insignificant because they are created by the ChaCha cipher anyway and must just be saved somewhere to not lose them. This means that storing them was only a necessary but not sufficient condition for a overall good performance.

This was noticeable for the first time early during the implementation of the action input field. This input field would enable the user to skip from any action to any other action. During testing, the performance of the navigation system turned out to be a problem. Skipping 100 actions took about one second during which the UI was unresponsive. As can be seen in Figure 4.24, this time increased linearly. Therefore, it was quite clear that something needed to be done against this, especially because the page with the most actions had over 3000 actions. All measurements were taken by starting at action 0 and skipping to the action specified at the x axis.

The root cause of the problem was the navigation system design which was called in hindsight ***linear navigation system***. It consisted of defining actions which build upon each other. This means that if we are at action 0 (initial state of the page) and want to go to action 5, we need to execute all the code inside the action

definitions between 0 and 5 to arrive at the page state as it should be at action 5. This is resembled in Figure 4.20.

This linear navigation system design was used to have a smooth implementation experience since one has to only write the actual page state changes between to actions. Code duplication was prevented because pages most of the time only changed a little bit between steps and thus the page state of each action is best described using differences. This complemented how the user experiences the visualization. The actions are numbered in a sequence and thus are inherently linear. Because of this, reflecting this linear nature in the system design made a lot of sense.

Since this "design flaw" was not the leading cause of the performance problem (going through a for-loop of size 3000 does not directly lead to performance issues), we will continue with briefly explaining how the actual code responsible for the transitions looked like.

During the transition between two page states, the state of the page elements which will change is saved such that we can undo the changes if the user decides to navigate back. This enabled “automatic action undoing” and thus to skip writing transitional code for backwards navigation. A function was written which retrieved the state corresponding to the transition and then applied it. This function worked for all backwards navigation without further intervention which was very convenient during development.

The problem with that architecture was that the state saving and the execution logic inside the action definitions were changing a lot of page elements directly by accessing them via their name that was given to them in the XAML code. This issue combined with the restricted, linear pathing between actions resulted in that significant performance loss that was described in Figure 4.24.

Linear navigation system with caches

After identifying the two underlying issues, a ***linear navigation system with caches*** was implemented. As the name suggests, cache entries were implemented to be able to navigate in constant time between an action and an action which was cached. To not only increase performance during these transitions (action to cached actions) but between all transitions, we check before each transition, if first moving to a cached entry would decrease the amount of hops needed to go to our destination. If this is true, we first go to the cached entry and then to our destination. Figure 4.21 demonstrates that such a navigation system needs less hops between any two given actions.

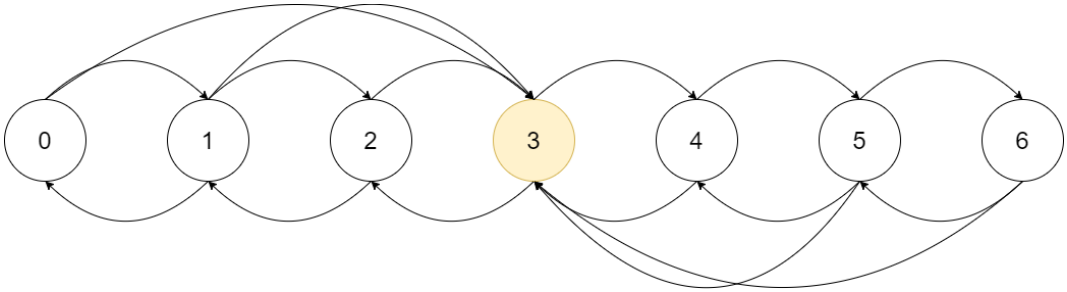


Figure 4.21: Navigation paths between actions in the linear navigation system with caches. The colored state has a corresponding cache entry.

The cache entries consisted of instructions to restore the complete state of a page at the action index for which this cache entry was for. They contained instructions for the complete state instead of only the difference between two actions because now, moving to that cache must initialize the page, independent at which action / page state we were before. This means that there was some overhead in initializing the page because essentially, the whole page was cleared and then the the page elements with their appropriate content were initialized, potentially leading to some unnecessary code execution because the content prior to cleaning was already the one we needed.

Nonetheless, creating a cache entry for every start of a round increased performance significantly as can be seen in Figure 4.27. The maximum response time went down from more around 40 seconds in the system without any caches to around 1.5 seconds (!). This could further be decreased to in average 250 milliseconds by creating cache entries for every start of a quarter-round (see Figure 4.27). Since it was not feasible to time every single data point, the data points marked as squares were just interpolated using previously timed data points. This means that for example, for the interpolated cache data points, the average of all previous measurements for skips to cached actions was taken.

Further, to decrease the load on the CPU while dragging the action slider, a very simple asynchronous navigation was implemented. It was implemented by using a stack as a buffer for the values received from the slider during dragging. Every 50 ms, the last value from the buffer is read and the page moves to that action. Afterwards, the buffer is cleared.
This did only enhance the slider but not the performance because at its core, it used the same navigation logic; just asynchronously. Nonetheless, it improved the user experience during dragging significantly which I found quite impressing for how

minimal the code for it is. In fact, the whole code for the asynchronous navigation can be seen in Figure 4.22. Before, dragging the slider lead to weird artifacts where the slider would jump forwards and backwards because the navigation was executed while dragging. Therefore, dragging the slider conflicted with the navigation system going to a certain action which would reset the slider to a certain position. Executing the navigation after dragging was finished was not an option because that basically made the slider useless.

One of the major drawbacks for this enhanced design with caches was that the automatic action undoing was no longer possible. Since we can not guarantee that the state between two actions has been saved, we cannot use our undo function. Therefore, the code for backwards navigation needed to be written manually.

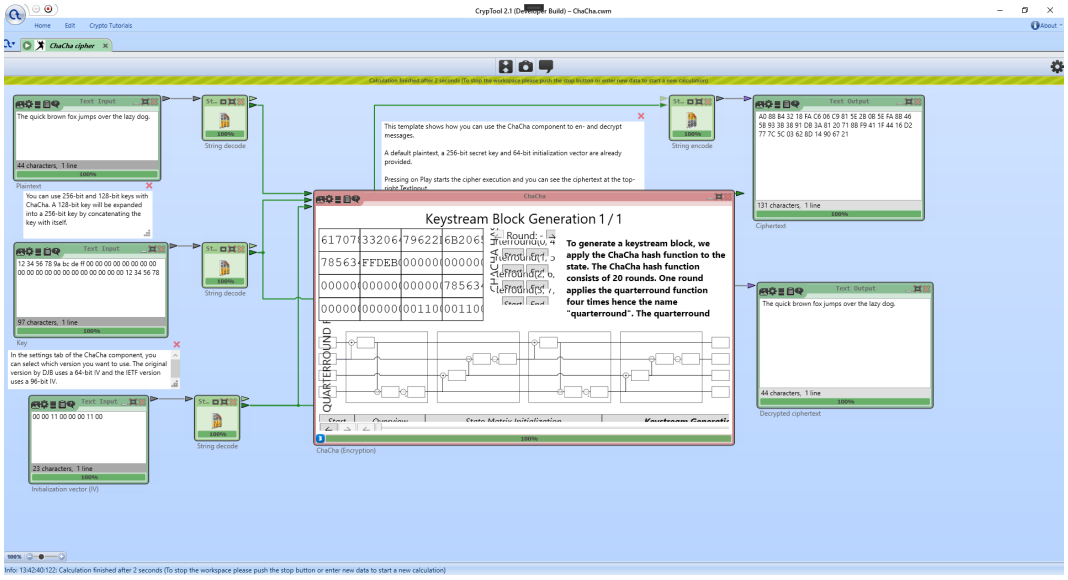
Final architecture

During the implementation of other features such as the diffusion, it was noticed that having to implement for each page action the forward and backwards code, the code became quite error-prone. It was hard to notice bugs because it was not feasible to check every single action from both directions manually and creating a testing framework just because of this was out of scope.
Some navigation bugs were easy to notice because due to the overall still linear nature of the navigation system, errors did propagate. This means that a error in a previous action most likely did break the page state on future actions because they depend on each other.
Nonetheless, this did not help in tracking down the bug because it was not known on which action the error happened. This only gave more reason to reiterate on the navigation system again.

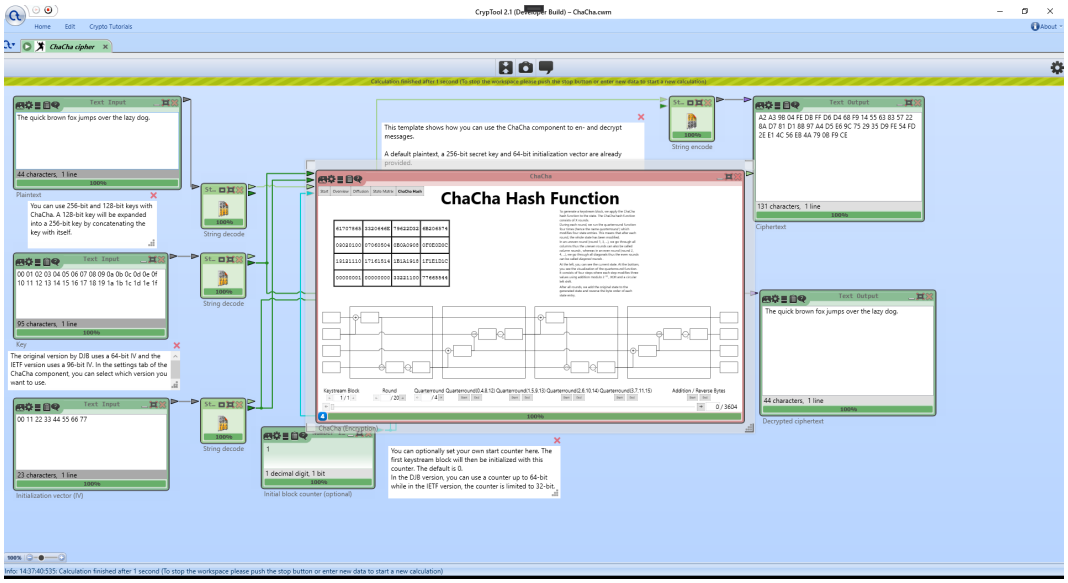
To decide if the code only needs to be refactored or indeed rewritten from scratch, all current problems with the existing codebase were summarized. They not only consisted of performance problems but also with visualization problems. For example, resizing the window did not appropriately scale the UI elements as can be seen in Figure 4.23a. Additionally, the current navigation system was quite restricted in what kind of UI elements it supported without further hassle. Since during the last navigation system update, the existing code was only updated, the core design was still all about directly manipulating and creating UI elements on demand. The existing functions revolved around the UI elements currently in use thus they could not easily be reused for different, new UI elements; leading to the mentioned limited support of the navigation system.

```
#region Asynchronous action navigation

private readonly Stack<int> AsyncMoveCommands = new Stack<int>();

private CancellationTokenSource ActionNavigationTokenSource;

public void QueueMoveActions(int n)
{
    QueueMoveToAction(n + CurrentActionIndex);
}

public void QueueMoveToAction(int n)
{
    lock (AsyncMoveCommands)
    {
        AsyncMoveCommands.Push(n);
    }
}

private async void StartActionBufferHandler(int millisecondsPeriod)
{
    // first stop action thread if one exists
    StopActionBufferHandler();
    ActionNavigationTokenSource = new CancellationTokenSource();
    CancellationToken cancellationToken = ActionNavigationTokenSource.Token;
    Task ClearActionBuffer()
    {
        return Task.Run(() =>
        {
            int n = CurrentActionIndex;
            lock (AsyncMoveCommands)
            {
                if (AsyncMoveCommands.Count != 0)
                {
                    n = AsyncMoveCommands.Pop();
                    AsyncMoveCommands.Clear();
                }
            }
            MoveToAction(n);
        }, cancellationToken);
    }
    while (true)
    {
        try
        {
            var delayTask = Task.Delay(millisecondsPeriod, cancellationToken);
            await ClearActionBuffer();
            await delayTask;
        }
        catch (TaskCanceledException)
        {
            break;
        }
    }
}

private void StopActionBufferHandler()
{
    ActionNavigationTokenSource?.Cancel();
}

#endregion Asynchronous action navigation
```

Figure 4.22: Asynchronous navigation subsystem



(a) Bad scaling property



(b) Fixed scaling property

Figure 4.23: Example for bad scaling property in previous plug-in versions

Essentially, the problem was that a lot of the code-behind was highly coupled to the XAML code. This was done like this because it was very straight-forward to do so and lead to fast results. At first, the trade-off between loss of maintainability/flexibility in the future and not having to spend precious time to learn design patterns for WPF applications seemed worth it. This thinking was grounded in the reason that the code for this bachelor's thesis would not get regular updates long into the future thus high maintainability or being easy to extend was not a priority. This assumption turned out to be false when realizing that the amount of "code smells" that could be handled was already too high about one month before the date of handing in the thesis. Every code change kept increasing the accumulated technical debt; killing any motivation still left to work on the existing codebase. Continuing like this for another month seemed impossible. Therefore, a list of all current problems was created together with what requirements a new software architecture would need to meet to solve them:

1. **(Inconsistent) Performance**
The underlying linear design was crippling the performance for the reasons already mentioned. The introduction of caches did only fight the symptoms and not solve the main issue. Further, it made the performance confusing for the users. Sometimes, it took close to no time at all to move to a certain action (action was cached) and on other times, it took quite a lot of time to move to a different action (action was not cached).
Requirement: Moving to any action should be done in $O(1)$. This means, it should not matter how many actions we needed to skip to arrive at the destination.

2. **Error-prone design for action creation**
Writing new actions was error-prone because code needed to be written for forward and backwards navigation which introduced mental overhead because it depended on the code of all previous actions. This also lead to error propagation. Errors were easily noticeable by users but were hard to track down to their origin.
Requirement: New actions should be able to be written without having to write backwards navigation code. Backwards navigation should be handled automatically and thus be "error-free by design".

3. **No coherent system design**
Adding new code without following a design pattern made it hard to grasp the system architecture over the long run. Additionally, the high coupling of the backend (code-behind) with the frontend (XAML) made it harder to implement new features in one part without needing to modify the other part. The system essentially got very rigid and over time, even seemingly small changes took quite some time to implement.
Requirement: The new system should make it clear what piece of code is responsible for what and thus be highly modular. This should also make it clear where new code must be added to implement a new feature without increasing technical debt; while also decreasing the possibility to introduce bugs since code is less coupled. To summarize, the new architecture should strive for high cohesion, but low coupling.

All these problems were solved using the MVVM design pattern with a new navigation system design. To implement them, the existing codebase was rewritten from scratch, which was time-consuming (took about two weeks) but in the end worth it. Figure 4.25 shows the performance of the final navigation system. As one can see, the time it takes to skip actions is constantly very low and only varies between a few milliseconds; making the user interface very responsive.

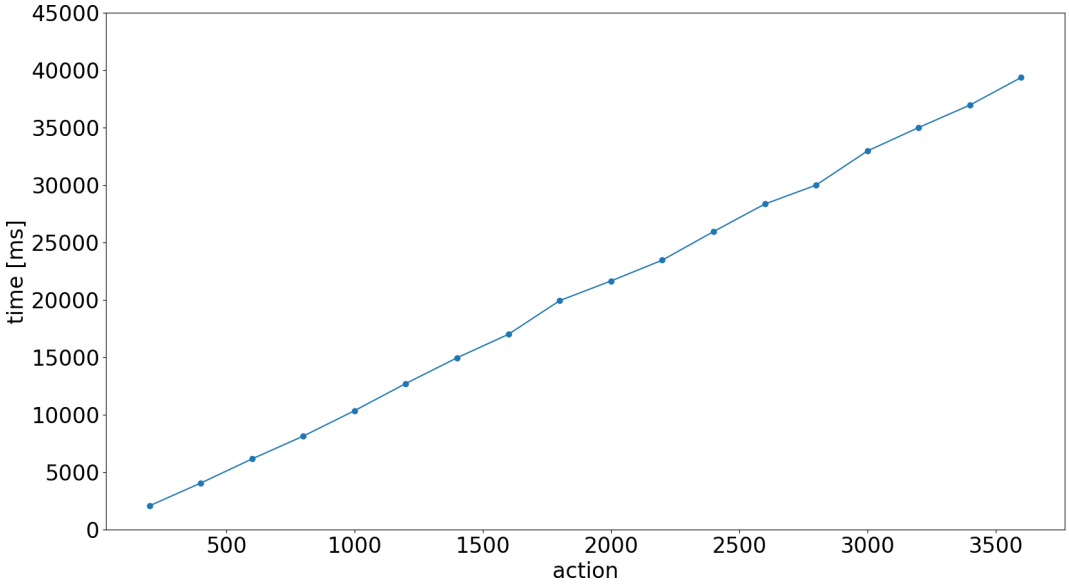


Figure 4.24: Performance of linear navigation system

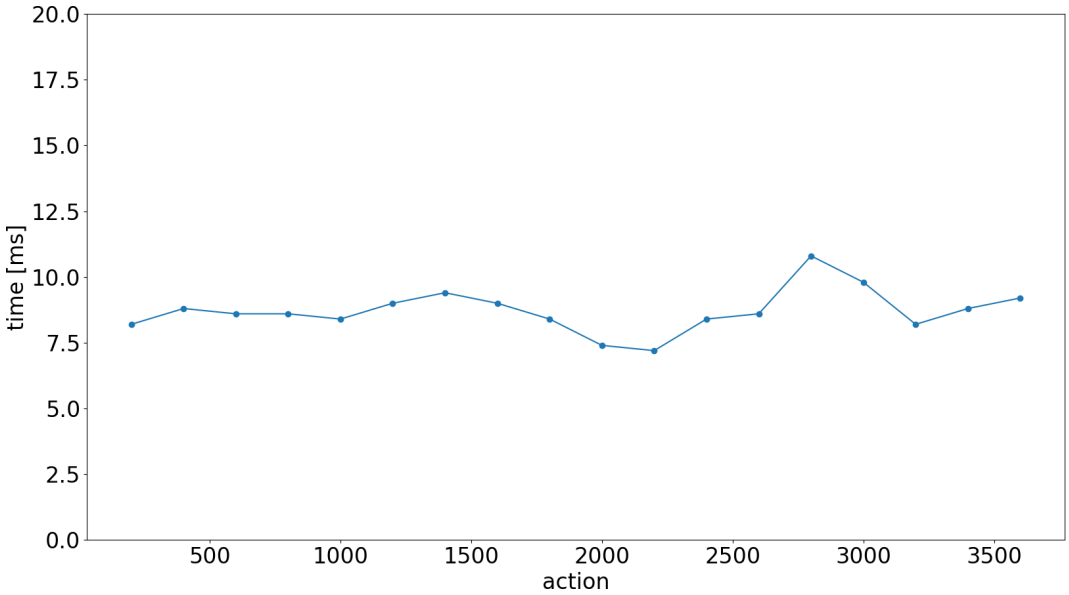


Figure 4.25: Performance of final (centralized) navigation system

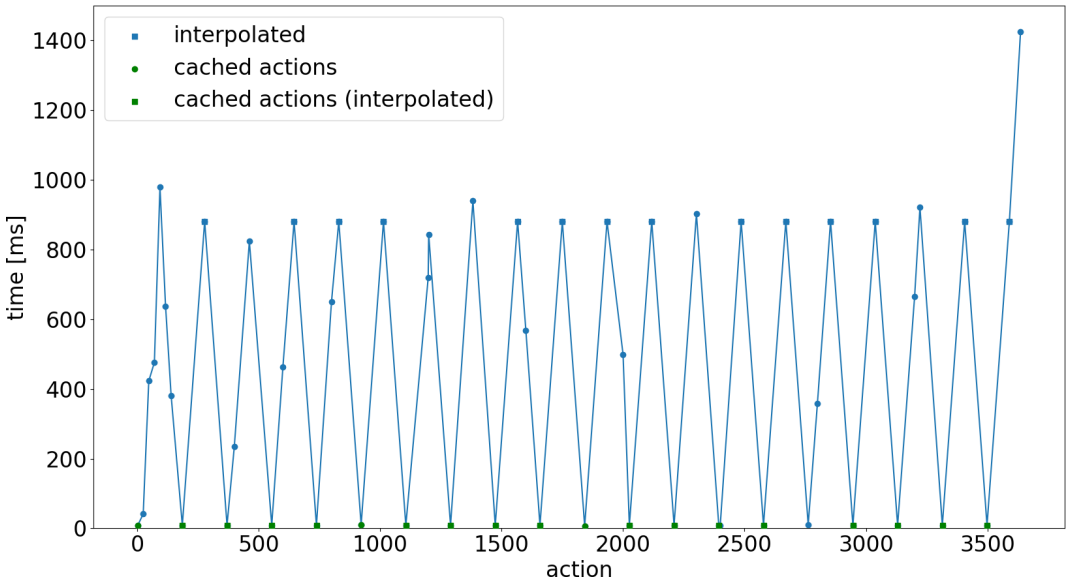


Figure 4.26: Performance of linear navigation system with caches for each round

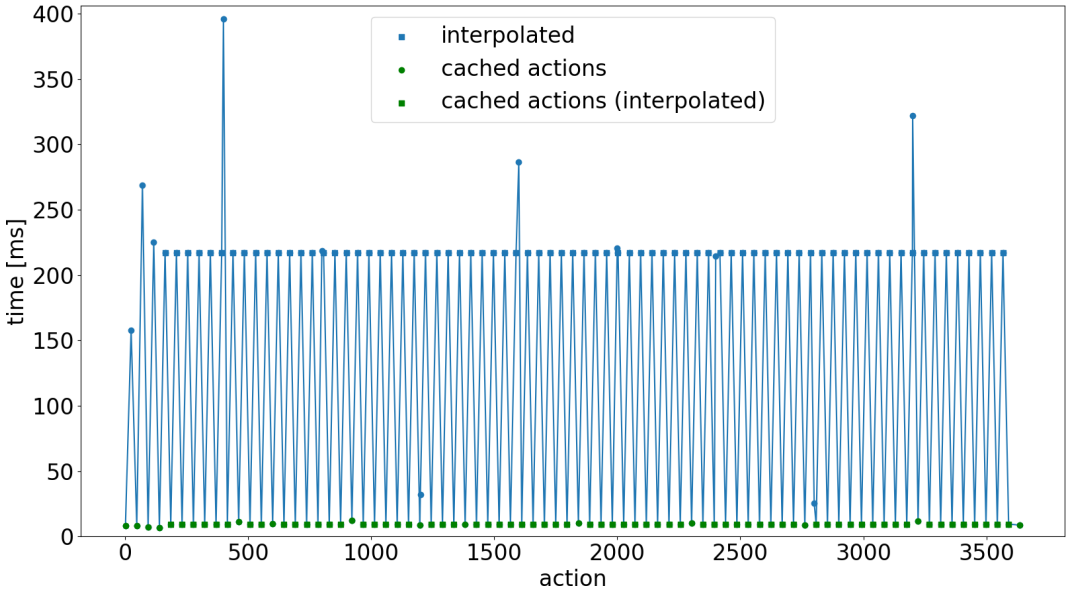


Figure 4.27: Performance of linear navigation system with caches for each quarter-round

5 Conclusion

This chapter finishes the thesis by summarizing its main points. It also includes suggestions how the plug-in could be further improved in the future.

5.1 Summary

This section gives a high level overview over the main points of this thesis.
In Section 4.2, the implementation is described. The section gave a detailed overview over how the plug-in was designed to meet the goals defined in Section 4.1. Section 4.3 described previous architectures and which problems they had.
The main point of this thesis is the description how the current implementation meets these goals and how the problems which occurred during development were solved.

Goals

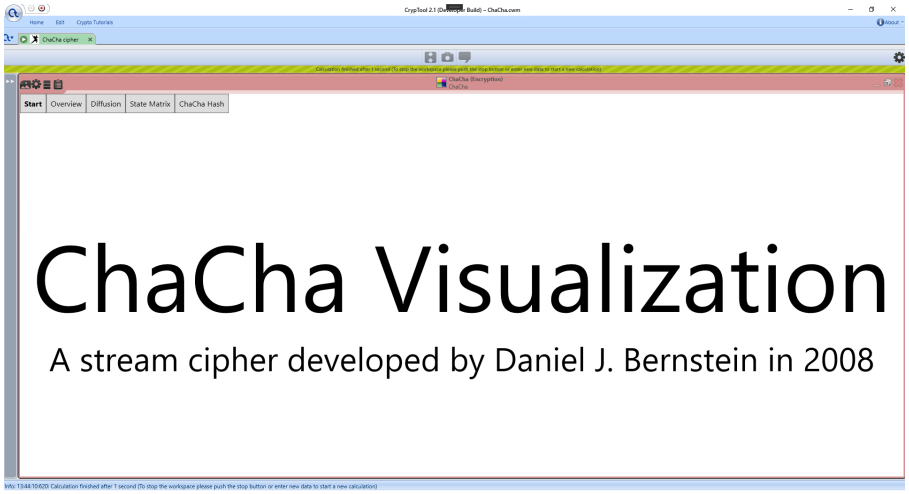
Three goals which the plug-in should meet were defined.

(*G*1) **Easy-to-understand visualization of the encryption process**
The first goal was met by creating a simple but thought-through user interface for the plug-in visualization.

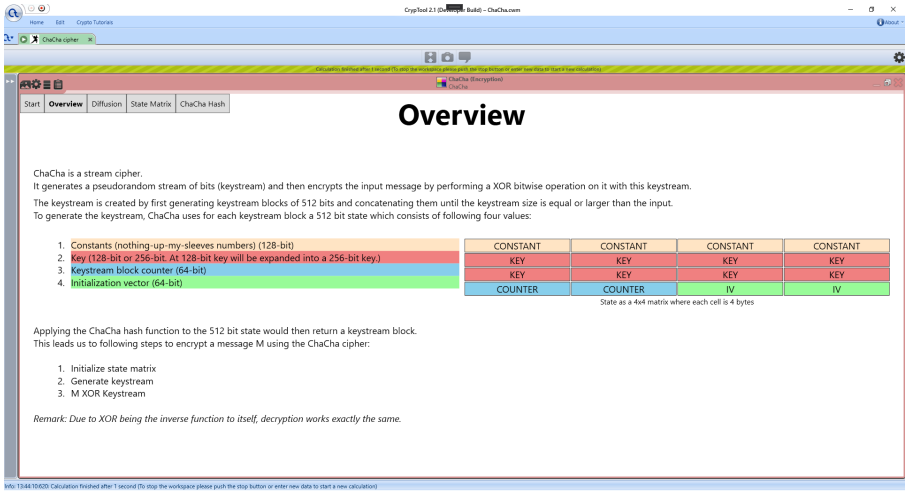
The visualization is split into five pages in the following order:

- Landing page
- Overview page
- Diffusion page
- State Matrix Initialization page
- ChaCha Hash Function page

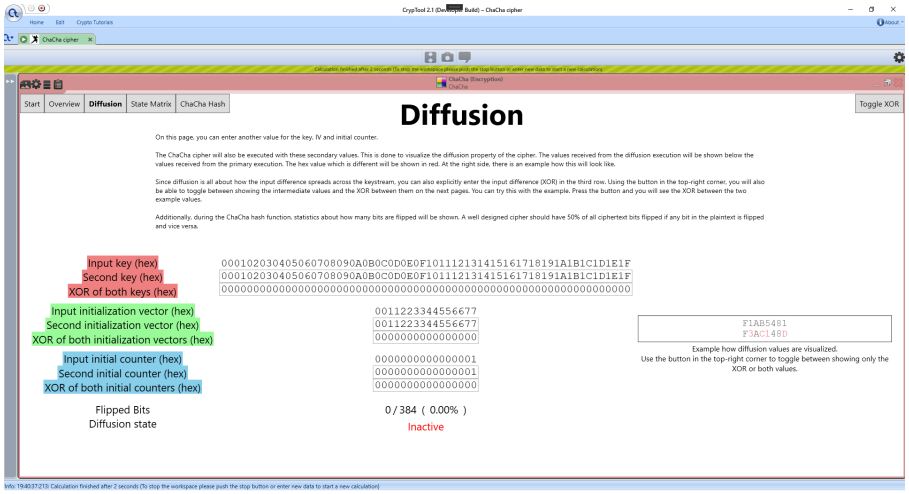
Splitting the visualization up into five different pieces (Figure 5.1) made it easier to let the user focus on a single step of the encryption process. This also made it possible to have a new page layout for each step without confusing the user



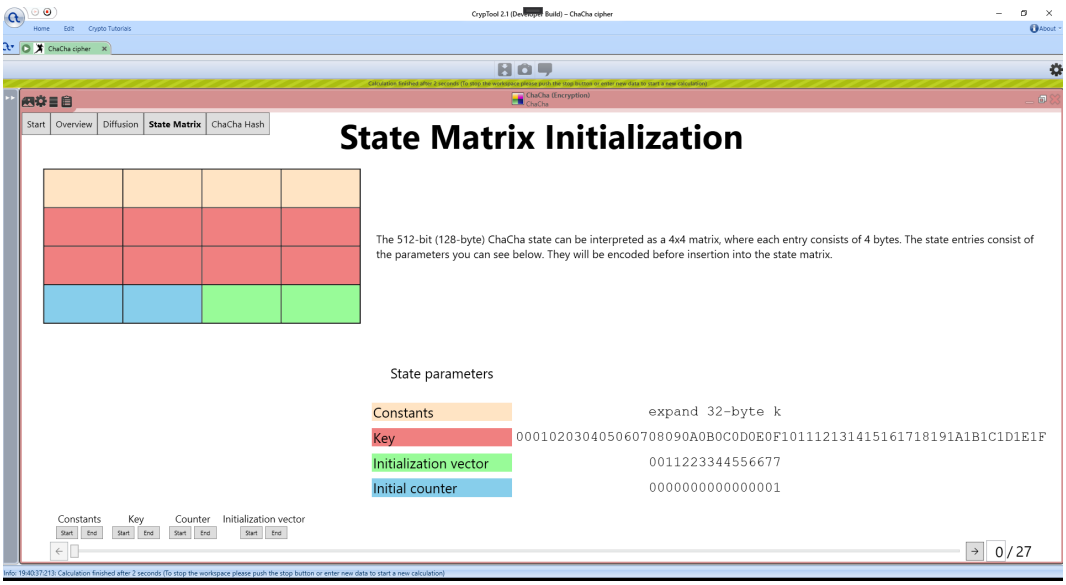
(a) Landing page



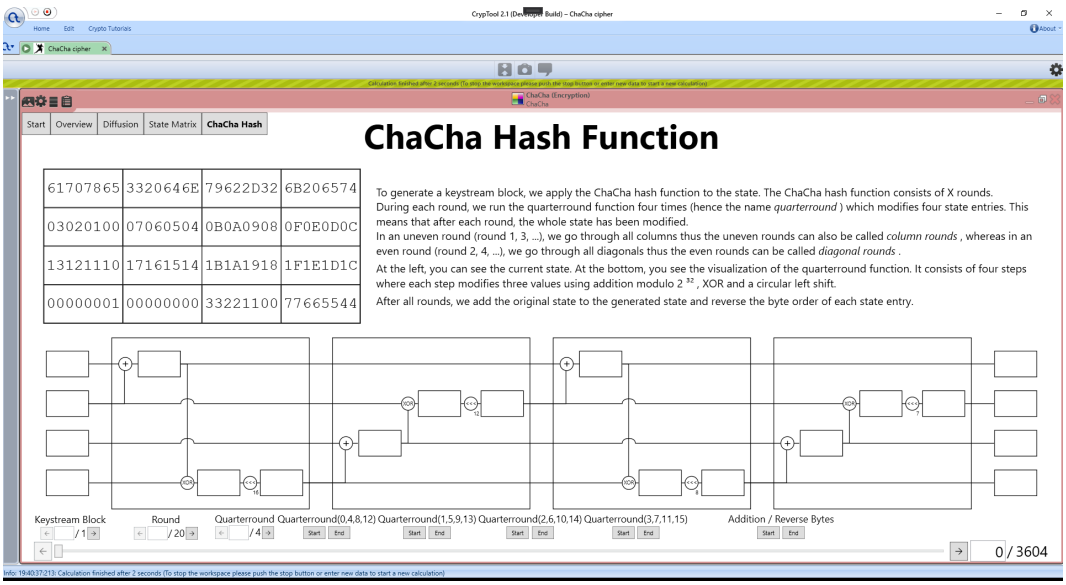
(b) Overview page



(c) Diffusion page



(d) State Matrix Initialization page



(e) ChaCha Hash Function page

Figure 5.1: All pages of the plug-in in their initial state

because each page is presented as an individual piece. A new page layout for each step was useful because we then were not restricted to a single layout for the whole visualization. Presenting the pages as individual pieces starts with the clear cut from the landing page to the overview page using the page navigation bar in the top-left of each page.

At the landing page, only the page navigation bar at the top-left is shown. This means the user must use it to advance. After using it, it is clear to the user that this is the place where he can navigate between pages.

Each page tells the user with descriptions what it is about. If a page has actions, the user has access to a slider with buttons in the bottom with which he can navigate within a page. At the bottom-right, the total number of actions a page has is shown as well as a text input which shows the current action index.

Using that text input makes it possible to skip to a certain action. Since the user will not know which action corresponds to which page state when using the plug-in for the first time, additional buttons are shown above the slider which bring the user immediately to important intermediate steps. Looking at the current action index, the user now knows what he has to type in there to immediately jump to that action again. This indirectly helps to better understand the cipher because it enables students and lecturers to talk about a specific step using the action index number.

The state setup and the hash function were visualized with attention to detail. During the state setup, each parameter is encoded separately and each encoding step is described. During the hash function, the quarter-round circuit diagram makes it possible to show the intermediate results in an intuitive way by placing them above the circuit lines. Intermediate results were important to let the user comprehend every single step of the keystream block generation. Further, background coloring helps to show which elements are used to calculate the next value or which state entries are currently in use by the quarter-round function.

(*G*2) **Visualization of the diffusion property**
The diffusion property is visualized by letting the user enter alternative values on the Diffusion page. Since the ChaCha keystream is independent of the plaintext, the user can only use alternative values for the key, counter and IV.

It was important to easily be able to flip bits. Thus, the user can not only enter the explicit alternative value but also the difference between the two values (XOR).

An example on the same page informs the user what he can expect for the next pages if he activates diffusion by flipping at least one bit. Using the button in the

top-right corner enables the user to choose between two vies:
A view which shows both values (as shown in Figure 4.11a) and a view to only show the XOR between both values (as shown in Figure 4.11b)
The values are always shown in hexadecimal. If a hexadecimal character is different between both cipher runs, it is marked red for easier visual recognition. The two different views were important because when studying the diffusion property of a cipher, one is more interested in the difference between two keystreams instead of their actual values. On the other hand, having access to the concrete values could also be useful thus both views were implemented instead of just one.

Additionally, the percentage of flipped bits is shown below the state at the end of each quarter-round in the ChaCha hash function page.

(*G*3) **Support for all variants of the cipher family**
The support to choose the number of rounds and the version (original version by Bernstein or IETF version) was implemented by adding appropriate settings to the plug-in. The inputs are then validated accordingly. This means that if the user has chosen Bernstein's original version, he must enter a 64-bit counter and a 64-bit IV. If he has chosen the IETF version, he must enter a 32-bit counter and a 96-bit IV. The plug-in will not start with wrong inputs. Instead, it will show an error message with the expected size for the input and its actual size.

The visualization works not much different with a 128-bit or 256-bit key. It is mentioned that if using a 128-bit key is used, it will be concatenated with itself. This is done in the step where the encoded key is put into the state.
To reflect the chosen version, only the the overview page and the state setup are slightly different. The overview page shows the correct parameter sizes and the last row of the state in the overview page and state setup page are partitioned accordingly.

Main problems

Two main problems were encountered while developing the plug-in.

(*P*1) **Architecture**
The first problem was to find out how the architecture behind the user interface should be laid out to not hinder further development. This means that it was not straight-forward to know how all systems (user interface, the cipher implementation, navigation, storage and retrieval of intermediate values) should interact without introducing hard-to-debug bugs in the long run.

(*P*2) **Performance**
The second problem was the overall performance of the plug-in. It was caused by the desire to let the user navigate to any step within a page. Letting the user navigate from any step to any step meant that the navigation system must support a lot of possible transitions in a reasonable time.

Solutions

Here are the solutions that were found for the two problems described above.

(*S*1) **MVVM design pattern**
The first problem was solved by following a design pattern. The MVVM design pattern was chosen because it was very popular across the WPF community [Smi09].

Using a design pattern helped in solving a lot of problems in a very obvious way. Features which previously were implemented with a lot of code smells (indicators that usually correspond to a deeper problem within an architecture) could now be implemented properly without increasing technical debt.
For example, before using a design pattern, the navigation system was spread across the whole codebase. With the MVVM design pattern, the whole navigation system could be written as a single interface. An abstract view model class then implemented this interface. All other view models which had actions then just extended this base class without having to duplicate any navigational logic.

As mentioned in Section 4.2.3, extensive usage of data binding helped in decoupling the view code from the underlying architecture. Therefore, if some things in the view should change, almost no code in the architecture has to change if the MVVM design pattern is properly implemented. Previously, the view code was very rigid. Changing it took a lot of effort because it kept breaking the architecture behind it.

(*S*2) **Centralized navigation system**
The second problem was solved by reflecting on the performance problems previous navigation system implementations had and their underlying issues.

The problem of the first navigation system was that it was designed to execute the transitions in a linear manner (hence the name *linear navigation system*). This meant that skipping a lot of actions would take a lot more time than skipping only a few actions because a lot more transitions needed to be executed for the larger skip. This resulted in taking more roughly 40 seconds to skip 3000 actions on the ChaCha hash function page.

This was tried to solve by introducing caches. Even though the cache implementation brought the average response time down to 250 ms, it did not solve the problem to a satisfactory degree. It introduced inconsistencies regarding the performance because with it, it was not obvious to the user why some skips took longer than others. Before, one could easily see that larger skips took longer than smaller.

As described in Section 4.2.3, as a result, the navigation system was optimized to have a consistently good performance for any skip. This means that the overhead of the navigation system design should be approximately equal for any skip. This was possible with the centralized navigation system. Any transition would start with first going back to the first action and then from there straight to the destination action. Therefore, the start action does not matter because moving to the first action from any action is done in $O(1)$ and moving from the first action to any action is done in $O(1)$. This improved performance dramatically. The average response time was now about 10 ms.

5.2 Future Work

This section lists five suggestions how the plug-in could be further improved to make it even more useful for the audience of CrypTool 2. The last point is a todo the author will implement during the next 4 weeks to make the plug-in fully complete.

1. **Better overview over flipped bits at the end of each round**
At the end of the Avalanche visualization, the author provided an overview over the percentage of flipped bits at the end of each round. This overview is very useful because it shows how the amount of flipped bits goes up very fast to around 50% and then stays near it which is exactly what one would expect from a good cipher.

   Using a plot instead of the simple text which is updated at the end of each round was considered but due to canvas and time constraints, this idea was not further pursued.

   Nonetheless, integrating such an overview into the Diffusion page should be possible. This would need cipher execution while still on the page instead on page exit but this would not be a big problem since a button to start cipher execution would suffice. Therefore, this plot could be an addition instead of replacing the text below the state in the ChaCha hash function visualization.

2. **Improve performance during diffusion**
All measurements in Section 4.2.3 were done with inactive diffusion since the performance during active diffusion improved in a similar manner. This means that thanks to the centralized navigation system architecture, moving to any action is still done in constant time even if diffusion is active.

The problem is that it takes around one or two seconds for each move (instead of around 10 ms if diffusion was inactive) which is quite annoying. We suspect that this is the case because the red color is implemented by creating an inline element for every character and marking it red if it is different.

An idea is to create a single element for every possible combination of color (black and red) and hexadecimal character ([0-9A-F]). Therefore, we would not need to create a new inline element for each character but could reuse the same in multiple places. If the performance issue results from the memory allocation, this approach should solve it.

However, first attempts resulted in weird bugs. So fixing this will probably be the most difficult point in this list and could result in once again having to rethink some major design decisions.

3. **Automatic navigation**
The AES visualization includes a button labeled with "Auto". It lets the visualization run without further user interaction needed. A slider was provided to adjust the speed (see Figure 2.2).

Such a button could be useful for the ChaCha visualization, too, especially for the page about the ChaCha hash function with its many actions.
Since asynchronous navigation is already in use for the action slider, implementing this feature could easily be integrated within the existing navigation system. Only page switches could maybe need some clever solutions since the action navigation is handled within each page thus navigating out of a page may not be straightforward.

In the AES visualization though, the automatic navigation does stop between each step (which roughly corresponds to a page in our visualization) so switching the page automatically may even not be desired.

4. **Salsa20 visualization**
As mentioned in Section 2.2, using the now existing codebase for ChaCha visualization to create a Salsa20 visualization would definitely increase the value gained from this thesis.

This would at least need adaption of the XAML code for the state matrix initialization and the quarter-round since ChaCha and Salsa20 differ in these aspects from each other. It would most likely even increase the value of the ChaCha visualization since both ciphers could then be compared side-by-side. Comparing them and their diffusion property should be very easy due to the very similar visualization.

5. **Localization and online help**
Currently, most texts are only localized in English. Only the memo fields, component labels and plaintext value are also available in German.
This will be changed in the near future. Then also an online help entry for the ChaCha plug-in will be available. The online help appears when pressing F1.

Bibliography

[Bec16] M. Becher. "Visualization of AES as a CrypTool 2 Plugin". B.Sc. Thesis. 2016. URL: https://www.cryptool.org/assets/img/ctp/documents/BA_Becher.pdf (visited on 12/11/2020) (cited on page 5).

[Ber05a] D. J. Bernstein. *Salsa20 security*. 2005. URL: https://cr.yp.to/snuffle/security.pdf (visited on 12/11/2020) (cited on page 3).

[Ber05b] D. J. Bernstein. *Salsa20 Specification*. 2005. URL: https://cr.yp.to/snuffle/spec.pdf (visited on 12/11/2020) (cited on page 3).

[Ber06] D. J. Bernstein. *Salsa20/8 and Salsa 20/12*. 2006. URL: https://cr.yp.to/snuffle/812.pdf (visited on 12/11/2020) (cited on page 3).

[Ber08] D. J. Bernstein. *ChaCha, a variant of Salsa20*. 2008. URL: https://cr.yp.to/chacha/chacha-20080120.pdf (visited on 12/11/2020) (cited on pages 1, 4, 11).

[Ber11] D. J. Bernstein. *Extending the Salsa20 nonce*. 2011. URL: http://cr.yp.to/snuffle/xsalsa-20110204.pdf (visited on 12/11/2020) (cited on page 3).

[Bin20] Binance Academy. *History of Cryptography*. 2020. URL: https://academy.binance.com/en/articles/history-of-cryptography (visited on 12/11/2020) (cited on page 1).

[Ech16] C. Echeverri. "Visualization of the Avalanche Effect in CT2". B.Sc. Thesis. 2016. URL: https://www.cryptool.org/assets/img/ctp/documents/BA_Echeverri.pdf (visited on 12/11/2020) (cited on page 8).

[Eur12] European Network of Excellence for Cryptology (ECRYPT) II. *eSTREAM: the ECRYPT Stream Cipher Project*. 2012. URL: https://www.ecrypt.eu.org/stream (visited on 12/11/2020) (cited on page 3).

[Goo14] Google. *Speeding up and strengthening HTTPS connections for Chrome on Android*. 2014. URL: https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html (visited on 12/11/2020) (cited on page 1).

[Lib18] Libsodium. *XChaCha20*. 2018. URL: https://libsodium.gitbook.io/doc/advanced/stream_ciphers/xchacha20 (visited on 12/11/2020) (cited on page 4).

[Nir+18] Y. Nir, A. Langley, Dell EMC, and Google Inc. *ChaCha20 and Poly1305 for IETF Protocols*. 2018. URL: https://tools.ietf.org/html/rfc8439 (visited on 12/11/2020) (cited on page 14).

[Smi09] J. Smith. *Patterns - WPF Apps With The Model-View-ViewModel Design Pattern*. 2009. URL: https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern (visited on 12/11/2020) (cited on page 54).

List of Figures

All figures and screenshots are either created by myself or their origin is indicated. Screenshots are created with “Snipping Tool" (MS Windows screenshot tool). Performance plots are created with matplotlib v3.3.3 (`https://matplotlib.org/`). Profiling was done in Release mode and with an Intel® i3-4130 @ 3.40 GHz. Architecture diagrams are created with diagrams.net (`https://www.diagrams.net/`). Code screenshots are created with Carbon (`https://carbon.now.sh/`).