Christoph Hechenblaikner

Automated Cryptanalysis of New Authenticated Ciphers

Master Thesis

Graz University of Technology

Institute for Applied Information Processing and Communications
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch

Advisor: Dipl.-Ing. Dr.techn. Florian Mendel
Assessor: Dipl.-Ing. Dr.techn. Florian Mendel

Graz, October 2014

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

In a standard cryptanalytic process, preliminary analysis of ciphers based on search tools plays an important role, since it allows the cryptanalyst to get an overview over properties of a cipher and gives indication for potential weaknesses in cipher components. Even though there exist tools to perform this preliminary analysis using various analysis methods, the effort a cryptanalyst has to invest into setting up the environment for those tools is enormous. This is mainly due to the fact, that certain chains of tools have to be set up in order to obtain valid and useful results when each of those tools requires an own representation of the analysed cipher. So a lot of effort has to go into establishing these tool representations.

Especially in a context of cryptographic competitions, where a potentially large set of ciphers needs to be analysed, this effort is significant, which results in the need for a degree of automation in the process.

This thesis introduces a framework that allows cryptanalysts to conduct an automated preliminary analysis on a potentially very high number of cipher designs when focusing on designs for authenticated encryption. The framework automatically parses the C-reference implementation, delivered in the submission to cryptographic competitions, into an abstract cipher representation. This representation can be combined with a tool specific adapter to transform the abstract cipher representation into the tool representation needed. This process works independent of the cipher given as input and can therefore be automatically applied to many different submissions. This saves the cryptanalysist time, which can be used to conduct in deep dedicated analysis methods and construct attacks based on the preliminary results. This leads to a potentially higher overall analysis degree of ciphers submitted to cryptographic competitions.

The framework introduced, was demonstrated using three cipher submissions to CAESAR competition (NORX, MORUS and KETJE) and one toolchain using methods for differential characteristics search and validation.

Key words: cryptanalysis, preliminary analysis, authenticated ciphers, automated analysis, differential cryptanalysis

Acknowledgements

Firstly, I would like to take the chance to thank my advisor, Florian Mendel, for his great work in advising, helping and encouraging me in different phases of the thesis as well as for his commitment and drive to make me benefit the most from this thesis. His patience, advice and guidance has enormously helped on the way of completing this thesis.

I would like to thank my girlfriend Julia, who has greatly supported me during the time working on this thesis and still supports me in everything I do. Without her, this would have not been possible.

Thank you to my mother, Katharina, for all her support during my studies, reaching a very important milestone in the completion of this thesis.

And finally, I'd like to thank my father, Georg, whose life and legacy has greatly inspired me in pursuing my current paths.

Christoph Hechenblaikner

Contents

List of Figures

List of Tables

List of Symbols

$\gg$	right shift of bitstrings
$\ggg$	right rotation of bitstrings
$\ll$	left shift of bitstrings
$\lll$	left rotation of bitstrings
$D_K(C)$	C is decrypted with the key K
$E_K(P)$	P is encrypted with the key K
AES	Advanced Encryption Standard
AST	abstract syntax tree
CAESAR	Competition for Authenticated Encryption: Security, Applicability, and Robustness
IR	intermediate representation
NIST	National Institute of Standards and Technology
SHA-3	Secure Hash Algorithm 3 (Keccak)

Chapter 1

Introduction

1.1 Motivation

Protecting digital information and securing global communication taking place over potentially insecure channels such as the internet has become more important than ever. Millions of people communicate using the internet (an arguably rather insecure channel) every day and exchange critical information over this network. The goal of IT-Security and Cryptography (or Cryptology) is to provide tools to protect this information and provide a framework for secure communication as well as to prevent any potential attacker from influencing on an ongoing communication. This is necessary to ensure the right for secrecy of information and privacy of the communication partners.

Besides just ensuring secrecy of data in a communication, factors such as guaranteeing a certain origin of information or making sure the information has not been altered on it's way state major goals of cryptography in practical applications. What is the value of critical data transmitted over an insecure channel securely, when the recipient cannot make sure it was created or sent by the intended communication partner? And what is the value in the trust to a certain communication partner if it cannot be ensured the data has not been manipulated by someone else on it's way? Consider the practical example of a system that delivers medical reports from doctors or hospitals to patients over the internet. Even if the communication takes place in a fashion that provides secrecy of data while being transmitted, a patient cannot be sure that the report was issued by the intended medical professional or hospital and has no proof that the data has not been manipulated by anyone in transit. Therefore, such information can be considered worthless without additional mechanisms taking care of those concerns.

Requirements like these raise the need for other cryptographic goals beyond the obvious secrecy of data. So information security manifests itself in various ways, caused by the wide variety of different applications.

Besides confidentiality of data, authenticity and integrity state a standard requirement that has to be fulfilled in every communication. In applications where symmetric cryptography is used, a variety of mechanisms have been developed to meet these requirements based on combining known symmetric primitives. In general the need for having confidentiality, authenticity and integrity in every communication raises the requirement of encryption schemes taking care of all of them in one primitive. Such primitives are represented by authenticated encryption ciphers.

The wide field of application for these authenticated ciphers come with high requirements towards security, speed as well as properties regarding their implementation in hardware and software. In order to establish a high quality of authenticated ciphers, new authenticated encryption schemes need to be found that address future needs in this area.

Cryptographic competitions have proven themselves as excellent tools for finding such new designs as well as to gain deep analysis by the cryptologic community. The CAESAR competition states such a challenge with the goal of identifying a portfolio of new strong authenticated ciphers. Changes in the number and quality of submissions to such competitions reveal major problems related to analytical effort in order to provide a sufficient analysis on the submitted designs. Therefore, an automated way of performing preliminary analysis of cipher designs in general (and authenticated ciphers in this particular case) would state a major advantage for such competitions.

This thesis introduces a framework for performing such an automated analysis especially designed to meet the requirements of cryptanalysis of new authenticated encryption schemes. It therefore allows cryptanalysts to easily conduct a preliminary automated cryptanalysis to a wide range of ciphers using various tools. This results in a much lower effort for cryptanalyst to establish a preliminary analysis and therefore enables them to spend more time on conducting targeted dedicated analysis to specific components of design. This enables them to deal with changing circumstances related to cryptographic challenges and creates an environment that allows deeper custom analysis and therefore higher confidence in the cryptographic schemes.

1.2 Introduction to Cryptography

This section gives a very brief overview of cryptography and it's main goals. Techniques of how to establish these goals will be explained in later chapters of this thesis.

When talking about cryptography and it's definition, one generally has to distinguish between three terms often used interchangeably:

a) **Cryptology** names the overall study of communications over potentially insecure channels and all problems and challenges related to it [28].

b) **Cryptography** names the study of creating systems that solve the problem of securing a communication over an insecure channel [28]. Other sources, such as [21], define it as the study of mathematical techniques for fulfilling the basic cryptographic goals.

c) **Cryptanalysis** deals with analysing and especially breaking of such systems [28].

All of these terms are related to the security of data and fundamental methods of how to establish, analyse and proof it. All of them describe how a communication can be protected from potential attackers at certain levels.

To explain the basic goals of cryptography, assume the communication illustrated in Figure 1.1. Two users, Alice and Bob, want to communicate (exchange messages) over a potentially insecure channel (e.g. the internet). Further, assume an attacker Eve, who wants to intercept and attack the communication. The basic goals of cryptography are now protecting this communication in various ways.

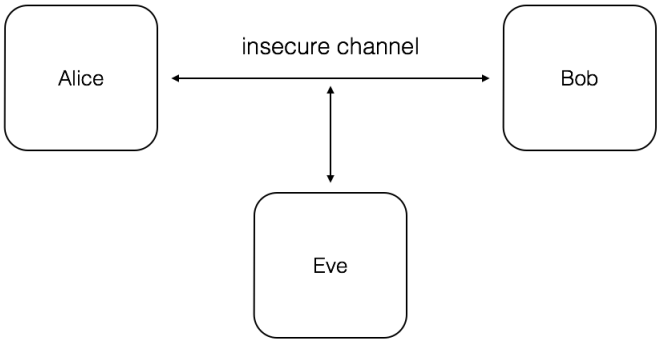

Figure 1.1: Basic setting to define cryptographic goals. Alice wants to communicate with Bob in the presence of an attacker Eve.

In general we can state the following basic goals of cryptography [21]:

1. **Confidentiality** is the basic goal of keeping information from all but those explicitly authorized to have it. The terms secrecy and privacy are often used to describe confidentiality. This could, for example, be protecting the used communication media or the information transmitted in a way such that it cannot be read by unauthorized users. In modern cryptography, this goal is realized by encrypting data with a secret key under the assumption that only those authorized to read the message have access to this key. The information transmitted, is useless to anyone except for those in possession of the secret key. So Alice would encrypt the message using a secret key, and share this key as well as the encrypted data with Bob. Eve, not in possession of the key, cannot gain any information from reading the encrypted message.

2. **Integrity** states the basic goal of being able to detect altering of messages when transported from the sender to the receiver. This means that any manipulation of

Year	Competition	Primitive
1998	Advanced Encryption Standard	Block Cipher
2005	eSTREAM	Stream Cipher
2008	SHA-3	Hash Function
2014	PHC	Password Hashing Scheme
2014	CAESAR	Authenticated Cipher

Table 1.1: Cryptographic competitions over the last 16 years [2].

the message can be detected by the receiver without any additional communication channel in place. If Eve would intercept and change the message on it's way from Alice to Bob, Bob could detect it and for example ignore the message.

3. **Authenticity** names the identification of either entities taking part in a communication or the transmitted information itself. Therefore, two types of authenticity can be distinguished here:

   a) Entity Authentication: In this case the partners entering a communication identify each other and make sure they are communicating with whom they think they do (Alice can prove that she is talking to Bob and vice versa).

   b) Data Origin Authentication: Data transmitted during such a communication should be authenticated. (Bob can ensure that the message was sent with certain characteristics intentionally from Alice). Note that this implies data integrity (changing data means changing the source of data)

4. **Non-Repudiation** addresses the desire of communication partners to proof that certain actions were previously committed in a communication. Assume Alice and Bob digitally sign a contract over the internet. At a later point both of them want to prove that the other one has signed the contract or in other words, prevent the other party from denying it.

1.3 Cryptographic Competitions

In the course of the past 16 years, cryptographic competitions in symmetric key cryptography have established as an excellent form of collecting and evaluating cryptographic primitives in various forms. Starting with the *AES Competition* in 1998, various competitions were hold by different organizations to establish standards or recommendations. Some of those challenges and the type of cryptographic primitive called for are listen in Table 1.1 in chronological order.

In general all cryptographic competitions follow the same idea and the same process / phases. In a call for submissions, the hosting organisation calls the cryptographic community to submit proposals for a new cryptographic primitive. The submissions and

the developed ciphers have to comply with basic rules and requirements defined by the organization to ensure alignment of the submissions with the overall goal motivating the competition. In most cases this motivation is based on practical problems with existing primitives or acts as a response to current threats or attacks compromising important security measures. The cryptographic community then submits designs alongside with reference implementations and security and performance analysis. After all proposals have been submitted, the research community conducts cryptanalysis on the submissions in order to either break weak designs or provide deep analysis of good submissions. This usually filters the submitted ciphers quite well as this takes place in several rounds of publishing attacks. After several (predefined) rounds, the organization running the competition selects a portfolio of winner primitives (of defined size). In some competitions, these winners are then standardized as for example in case of NIST's *AES* or *SHA-3* competition. Other competitions do not drive standardisation even though finalists of such competitions might be standardised by other organisations.

1.4 Problems of Cryptographic Competitions

A trend that can be observed in those cryptographic competitions is the heavily increasing number of submissions. Figure 1.2 illustrates the increasing number of first round submissions to the before discussed competitions. It can be seen that the submissions have increased about 367% from 15 to 55 submission between 1998 (AES) and 2014 (CAESAR). Even though this development can be generally seen as very positive, there are problems related to it. The huge analytical effort needed to provide a preliminary level of cryptanalysis has become enormous. In addition, the general quality of submissions increased significantly. This means a larger number of submissions require deep analysis, since none of them can be broken early in the competition. This effect of more submissions and higher submission quality results in a strong rise of the cryptanalytic effort needed to provide good analysis of all ciphers.

Figure 1.3 shows the number of first-round and second-round candidates of the AES, SHA-3 and CAESAR competition (ongoing). It can be observed, that in earlier competitions, the number of submissions went down relatively fast, due to broken schemes and discovered weaknesses (AES: 60%, SHA-3: 73%). In contrast to that, the CAESAR candidates have only gone down by seven (12 %) in the first six months (about the time of a usual round) and it is not expected to fall much lower. Therefore, the cryptographic community now needs to provide in-depth analysis of 48 instead of 5 (AES) or 14 (SHA-3) ciphers which states an incredible workload on the community.

Even though there are automatic tools available to analyse parts of the proposed primitives in various ways, the application of such analysis tools is still linked to a lot of manual effort. This effort mainly results from the different in- and output file formats, algorithm representation and interface structures that analysis tools rely upon. Figure 1.4 illus-

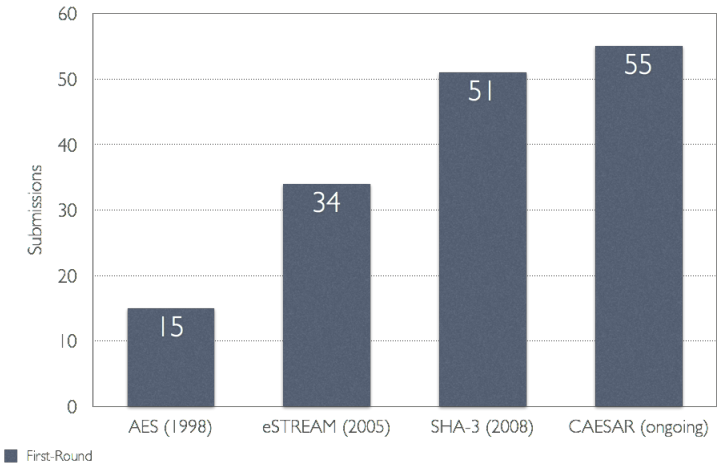

Figure 1.2: Development of first round submissions to cryptographic competitions since 1998 [2].

trates this situation. Assume a cryptanalyst wants to apply n tools to analyse a primitive. In this case she would have to create

- n representations of the algorithm (e.g. different languages, formats, etc.) based on the original reference implementation and the specification.
- n input format representations.
- n procedures for conducting the analysis based on the different interfaces of the tools (e.g. c-interface, command line etc.)

This results in a huge overhead to set up a toolchain as illustrated in Figure 1.4 and establish a preliminary level of cryptanalysis to base dedicated analysis methods on.

This thesis introduces an automated framework that takes as an input the submission reference implementation and transforms the proposed algorithm into an abstract format. The framework further provides mechanisms to easily transform this abstract representation into any representation format needed in the toolchain of such an analysis. This transformation into the abstract representation as well as the transformation in other formats is performed fully automatic and is therefore intended to perform automatic preliminary cryptanalysis on a large number of cryptographic submissions while providing the possibility to easily extend the framework with new tools or advanced transformations. We refer to Chapters 4 and 5 for further information on this principle and the framework itself.

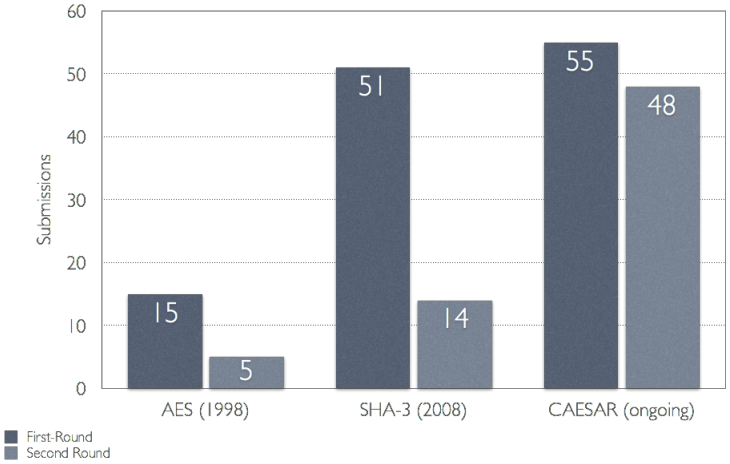

Figure 1.3: Development of first round vs. second round submissions in cryptographic competitions. [2].

1.5 Overview

The goal and scope of this thesis is to create a basic framework that allows to conduct automated analysis on authenticated cipher submissions using various analysis tools. The remainder of this thesis is organized as follows. Chapter 2 provides a basic understanding of what authenticity and integrity mean, how they are achieved in symmetric and asymmetric cryptographic systems and how dedicated designs achieve the given goals. Further, it gives a short introduction to design principles of dedicated authenticated ciphers and introduces the three selected submissions to the CAESAR competition, which were used to demonstrate the framework presented in this thesis.

Chapter 3 illustrates basic methods and principles of cryptanalysis and particularly those relevant to authenticated ciphers. Further it introduces the reader to the goals and different types of cryptanalysis as well as tools that can be used to perform certain subtasks in the analysis process. In addition, it describes typical workflows during analysis of authenticated ciphers, points out potential problems and how an automated workflow for conducting an automated analysis can help solve them.

Chapter 4 introduces an automated analysis workflow that can perform a fully automatic preliminary cryptanalysis based on submission documents to the CAESAR competition. It explains the basic tasks, goals and challenges of performing such an automated analysis and states examples for some of the formats and frameworks used in this process.

Chapter 5 finally describes the components and processes of the implemented automated analysis framework, shows how it accomplishes the tasks and goals and overcomes the challenges stated in Chapter 4 and describes how it was applied to the three selected

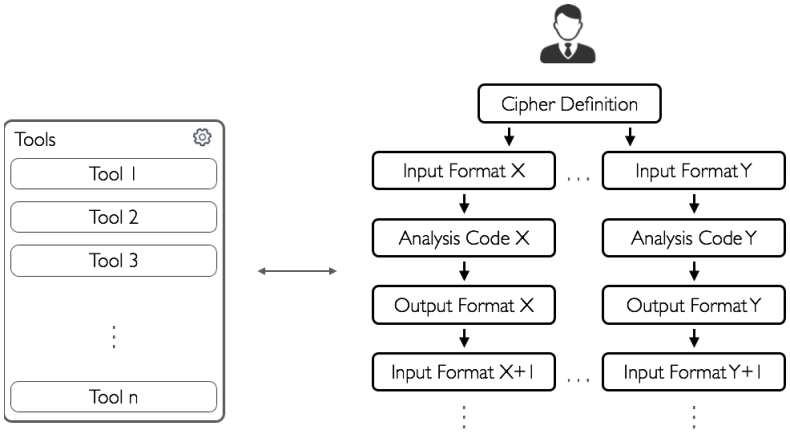

Figure 1.4: Typical setting for applying different cryptanalysis tools to a cryptographic primitive.

CAESAR submissions. Further, it shows customization possibilities and how new tools can be easily integrated into the framework to extend it.

Finally, Chapter 6 provides an overview of the solutions provided by this framework and the results gained from the conducted analysis. It discusses the results, their current scope in this thesis and limitations at this point. Further, an overview of next tasks in the development of an automated analysis framework is provided and directions of future work are discussed.

Chapter 2

Authenticated Encryption

This chapter provides an overview of authenticated ciphers. To understand the idea, principles and basic structures of this type of cryptographic primitive, a short introduction to conventional cryptographic systems, their basic components as well as their weaknesses and drawbacks regarding authenticity of data and entities is provided. Further, this chapter provides an overview of existing symmetric authenticated ciphers as well as a short introduction to the CAESAR competition. After providing a short introduction to relevant design approaches of authenticated ciphers, it gives a short description of the cipher designs addressed and analysed in this thesis.

2.1 Conventional Cryptographic Systems

This section provides a brief overview of the principles and goals of today's conventional cryptographic systems. It provides a short introduction to how those basic goals are achieved and why new cryptographic primitives, such as authenticated encryption, are desired.

2.1.1 Basics and Primitives

Conventional cryptographic encryption systems solve the problems stated in Chapter 1 using two main categories of primitives: symmetric and asymmetric cryptography.

Symmetric Cryptography

Symmetric cryptographic systems assume a shared secret key among the participants of a communication over an insecure channel. This means that the key has been exchanged between the two entities in a way such that both are in possession of the key and can trust

its secrecy. Participants use the same secret key K_S to encrypt and decrypt data. Thus, these systems are called symmetric. All primitives used in order to fulfill the basic goals and requirements of cryptography are based on such a symmetric approach for encrypting and decryption data with the same key. Even though multiple keys might be used in a communication to achieve different goals, only one key should be used to perform a particular sub-task in such a system. For example, two different (symmetric and secret) keys K_C and K_A should be used for ensuring confidentiality (K_C) and authenticity (K_A) in a communication in order to ensure the claimed security of the system. Figure 2.1 illustrates the basic process of encrypting a message using symmetric encryption.

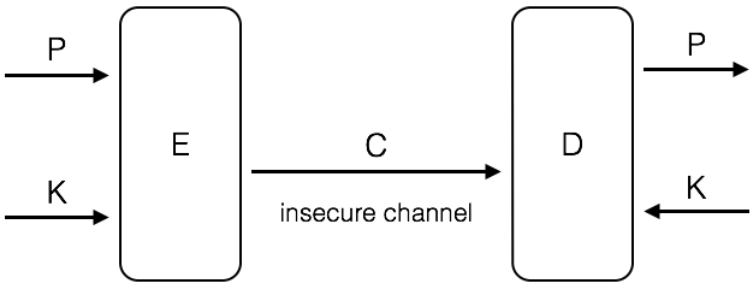

Figure 2.1: Basic principle of symmetric encryption. P ... plaintext, C ... ciphertext, K ... secret shared key, E ... encryption, D ... decryption

The basic operations in symmetric cryptographic systems are:

- Encryption: A plaintext message M is encrypted using a key K resulting in a ciphertext C. The information contained in the message M cannot be extracted from the ciphertext without applying decryption with the same key K.

$$C = E_K(M) \tag{2.1}$$

- Decryption: A ciphertext message C, which was encrypted using the key K, is decrypted using the same key K resulting in the plaintext message M. This operation forms the inverse to the encryption operation.

$$M = D_K(C) = D_K(E_K(M)) \tag{2.2}$$

In symmetric cryptographic systems, the basic cryptographic goals are fulfilled in the following way:

- **Confidentiality**: In symmetric cryptographic systems, the goal of confidentiality is reached by assuming two parties share a secret key that is used to encrypt messages. Anyone not being in possession if the secret key shall not be able to intercept and read encrypted messages. In such systems, the distribution of keys

among all participants states one of the major challenges that this assumption relies on.

- **Authenticity and Integrity**: The goals of authenticity and integrity are commonly accomplished using so called Message Authentication Codes (MACs). It has to be noted that these mechanisms only provide a certain level of *data-origin* authentication rather than an identity authentication of the sender. We refer to Chapter 1 for more information about the different types of authenticity. In Section 2.2, further information on authenticity in symmetric systems is provided.
- **Non-Repudiation**: Non-repudiation (preventing the denial of committed actions) cannot be solved using symmetric primitives alone. Since a key is shared among various participant in a communication, it can never be assured that a certain action was committed by a specific member of the group. In practice, a trusted third party is needed for tracking commitment and prove actions of participants at a later point.

Asymmetric Cryptography

Asymmetric cryptographic systems, in contrast to the previously described symmetric approaches, rely upon the principles of two keys being present for each participant of a communication. This keypair consists of a public key K_{pu} and a private key K_{pr}, which belong together and are generated at the same time. In contrast to symmetric systems, only the private key K_{pr} has to be kept secret and is not shared among users. The public key K_p is considered public information and can be freely distributed over insecure channels.

Asymmetric cryptographic systems address the problem of not being able to easily distribute symmetric secret keys is an easy way. When using symmetric encryption, the participants need to exchange or agree on a secret key *before* the secure communication takes place. A secure channel for either protecting keys transmitted or protecting the key agreement process is needed. In reality this is a very difficult requirement to fulfill, since the establishment of such a channel states a goal of the whole process in the first place. So the problem is reduced to establishing a secure channel for less data or shorter time. In some cases this might not be possible using only symmetric cryptography.

Asymmetric cryptographic systems now solve this problem by being able to distribute public keys over a potential insecure channel. Therefore, the need for a secure channel is shifted towards the problem of verifying that the received public key belongs to the intended communication partner. Since now two keys are involved, the operation set is larger than in the symmetric case.

The basic operations in asymmetric cryptographic systems are:

- Encryption: A plaintext message M is encrypted for user A with his/her public key K_{pu_A} resulting in a ciphertext C_A. The information contained in the message

M cannot extracted from the ciphertext without applying decryption with user A's private key K_{pr_A}. Note that the ciphertext C_A can not be decrypted by a key other than K_{pr_A} and therefore is encrypted for user A.

$$C_A = E_{K_{pu_A}}(M) \tag{2.3}$$

- Decryption: A ciphertext message C_A, which was encrypted with user A's public key K_{pu_A}, is decrypted with user A's corresponding private key K_{pr_A} resulting in the plaintext message M. This operation forms the inverse to the encryption operation.
$$M = D_{K_{pr_A}}(C_A) = D_{K_{pr_A}}(E_{K_{pu_A}}(M)) \tag{2.4}$$

- Signing: A message M is signed by user A by applying the previously discussed decryption operation to the message M using his / her private key K_{pr_A} resulting in the signature value S_A. This signing can only be performed by a user in possession of the corresponding private key. Therefore, this operation proves that a specific user applied the operation to a message.
$$S_A = D_{K_{pr_A}}(M) \tag{2.5}$$

- Signature Verification: A signature value S_A, signed by user A, is verified by applying the before mentioned encryption operation on S_A using the public key of A resulting in M_A. If $M_A = M$ holds for a known message M, the message M must have been signed using A's private key.
$$M_A = E_{K_{pu_A}}(S_A) = E_{K_{pu_A}}(D_{K_{pr_A}}(M)) \tag{2.6}$$

In asymmetric cryptographic systems, the basic cryptographic goals are fulfilled in the following way:

- **Confidentiality**: In asymmetric cryptographic systems, confidentiality is established encrypting data with the public key K_{pu} of the communication partner and using the own private key K_{pr} to decrypt message encrypted with the corresponding public key. Rather than relying on the presence of a commonly shared and secret symmetric key, in such a cryptographic system public keys used for encryption are publicly shared over insecure channels. The problem of having to establish a common secret key is now shifted towards verifying that the public key received really corresponds to the private key of the communication partner and has not been altered by an attacker (public key authenticity). This verification can be performed using various kinds of methods, where the most common is represented by digital certificates and PKIs.

- **Authenticity**: In an asymmetric cryptographic system, identity authenticity can be established by signing messages or data blocks. Under the general assumption

that a users private key is only accessible to him/her and the public key can be verified to belong to the user's identity, the above mentioned signing and verification operations ensure this identity authenticity.

- **Non-Repudiation**: Since every participant in an asymmetric cryptographic system is in possession of a private/public - keypair, no common key is shared between entities within the system. Thus, non-repudiation can be ensured using digital signature schemes. A message signed by user A can always be proven to be signed by A through the verification operation and only A can sign messages due to the possession of the private key K_{pr_A}.

2.1.2 Differences

Symmetric and asymmetric cryptographic systems show various differences in underlying principles as well as characteristics in an application scenario. The following lists some of the most important differences:

- **Underlying principles**:
  - In general, asymmetric cryptography is based on mathematical trapdoor-one-way functions and number theoretical problems. A trapdoor-one-way-function is thereby define as a function $f(x)$ that is very easy to calculate when generally being very hard (or resource intensive) to invert. This means that $y = f(x)$ is very easy to calculate when $x = f^{-1}(y)$ is generally infeasible to calculate. Further, such functions provide a trapdoor which is defined as information that softens the inversion of $f(x)$ and turns it into an easy problem. Therefore, calculating $x = f^{-1}(y)$ is very hard without and very easy with the trapdoor. In such systems, the encryption is linked to calculating the function value, when the decryption is linked to solving the inversion-problem of the function and the trapdoor is given as the private key. In this case the public key represents the parameters of the one-way-function f when the private key represents the trapdoor to solving the inversion problem under the parameters defined by the public key. Candidates for such functions are: prime multiplication / factorization, modular exponentiation / logarithm calculation or squaring / square root calculation in finite fields.

  - Symmetric ciphers are, on the other hand, based on principles which combine a message M and a symmetric key K to a value C in a way, such that it is infeasible to unveil M from C without the knowledge of K. Approaches of such non-linear combinations are based on elements like *Feistel* structures [21] or *Substitution-Permutation-Networks* [21].

- **Speed**: Asymmetric encryption schemes are, due to the nature of their underlying problems, extremely slow compared to symmetric schemes. Symmetric ciphers

are on the other hand very fast and very efficient to implement in hardware and software, which makes them very attractive for practical applications.

- **Key-Length**: Keys of asymmetric cryptographic systems are usually significantly longer that those of symmetric encryption while providing the same level of security.
- **Fulfillment of cryptographic goals**: As mentioned earlier, not all cryptographic goals can be met using symmetric encryption without any additional mechanisms. The goal of ensuring non-repudiation can not be fulfilled in an symmetric systems without having a trusted third party in place. Further, identity authenticity cannot be ensure by symmetric schemes in constrast to asymmetric systems.

Especially, the big differences in speed and key-length between the two types of systems often suggest combinations of them referred to as hybrid encryption schemes. These schemes use principles of both types to ensure fast primitives as well as short key-lengths. A simple example of such a combination is *key-wrap* [26]. In this technique, a symmetric cipher is used to encrypt data, but asymmetric encryption is used to *wrap* (protect) the symmetric key used.

2.2 Authenticity in Symmetric Cryptographic Systems

As discussed earlier, asymmetric cryptographic systems are capable of providing identity and data authenticity under basic assumptions. In areas where symmetric cryptography is required, data authenticity has to be ensured using additional mechanisms and primitives (Note that identity authenticity can not be ensured in such systems). Systems ensuring both, confidentiality and authenticity, can categorized into the following two groups.

2.2.1 Generic Composition

An approach of creating such a cryptographic mechanism is combining two symmetric primitives: an encryption scheme and a MAC. A MAC is a keyed hash function h that takes as input a message M and a secret key K and calculates the MAC-value as $T = h(M, K) = h_K(M)$, where T is referred to as the *authentication tag*. If the message $M||T$ is transmitted during a communication the receiver can verify integrity and authenticity by simply calculating $T_X = h_K(M)$ based on the message and the shared secret key. If $T_X = T$ holds, the message has not been altered in transport and the data was authenticated by someone in possession of the shared secret key.

Since authenticated encryption has the goal of delivering confidentiality as well as integrity and authenticity, an encryption primitive has to be combined with such a MAC. In general, there exist three approaches for this combination. The following briefly de-

scribes them and their security level. All the statements made about the security level rely on the assumption of secure symmetric encryption schemes as well as MACs.

- **Encrypt-And-Mac**: A message M is encrypted using an encryption key K_E to ensure confidentiality of the message. Further, a MAC value T is calculated from M using the authentication key K_A to ensure authenticity and integrity of the transmitted message. Both operations (encryption and MAC) are applied to the transmitted message M itself and are combined to the resulting ciphertext C.

$$C' = E_{K_E}(M) \tag{2.7}$$
$$T = h_{K_A}(M) \tag{2.8}$$
$$C = C'||T \tag{2.9}$$

  Security: This combination is generally considered insecure. Even though it provides authenticity (directly linked to strength of MAC scheme), it lacks confidentiality. This is due to the fact that the value $T = h_{K_A}(M)$ leaks information about the plaintext (e.g. repeats). Since h_{K_A} needs to be deterministic by definition, it reveals information about the plaintext and therefore breaks confidentiality. Even though plaintext integrity is ensured using the MAC, ciphertext integrity is not established using this method. This means that it does not detect or prevent the case of an attacker finding an altered ciphertext decrypting to M. [5]

- **Mac-Then-Encrypt**: When following this approach, the before-mentioned chaining of operation takes place by first calculating the MAC value (T) of the message M, and then applying encryption on the message and the authentication tag.

$$T = h_{K_A}(M) \tag{2.10}$$
$$C = E_{K_E}(M||T) \tag{2.11}$$
$$\tag{2.12}$$

  Security: This combination now considers the before-mentioned leak of plaintext information to the authentication tag T and counteracts by encrypting it. Even though confidentiality can be fixed this way, it does still not provide ciphertext integrity and therefore shows the same problem as the *encrypt-and-mac* method. [5]

- **Encrypt-Then-Mac**: In this method, a message M is encrypted using an encryption key K_E (confidentiality). The output of this encryption process is then used as input for the MAC operation using the authentication key K_A resulting in the authentication tag T. The two operations are now chained while encryption being first.

$$C' = E_{K_E}(M) \tag{2.13}$$
$$T = h_{K_A}(C') \tag{2.14}$$
$$C = C'||T \tag{2.15}$$

Security: This combination provides confidentiality, authenticity and ciphertext integrity and can therefore be considered secure [5]. Possible combination of secure encryption schemes and MACs combined in this way are AES-CBC (confidentiality) and AES-CMAC or HMAC-SHA1 (authenticity) [5].

2.2.2 Dedicated Authenticated Encryption Ciphers

Besides the above stated combination of encryption and MACs, dedicated authenticated encryption schemes have been developed over the years. ISO/IEC 19772:2009 standardises the following designs (mostly block cipher modes-of-operations) [14] [15]:

- GCM
- CCM
- OCB
- EAX
- KeyWrap / SIV

2.2.3 Drawbacks and Limitations

The before discussed solutions for adding authenticity to symmetric encryption systems show major drawbacks causing the need for new authenticated encryption schemes. Methods combining symmetric primitives and MACs have to use multiple primitives to ensure authenticity and confidentiality. Therefore, the security of the system relies on the security of two different components. Further, this makes such schemes relatively slow and demands for two different symmetric keys to not show potential vulnerabilities.

The described dedicated designs of authenticated ciphers show good properties in terms of performance, single-pass design, only rely on a single underlying primitive and only use one key. Single-pass designs name primitives, that do not imply a second path branching of the input of the cipher to calculate the authenticity tag separately. Although these designs state a major improvement compared to previously described solutions, they still show some drawbacks.

In particular for the block cipher mode *GCM*, a relatively high number of weak keys [23] and problematic cyclic properties [27] justify the need for new schemes improving these properties.

2.3 Goals of Dedicated Authenticated Encryption Schemes

Authenticated encryption has the goal of providing one cryptographic primitives that ensures confidentiality, integrity as well as data authenticity in a symmetric cryptographic system. As shown in previous chapters, symmetric encryption is not capable of providing this using standard symmetric primitives. The disadvantages of methods combining

symmetric ciphers with MACs and weaknesses of existing dedicated schemes (as discussed in Section 2.2.3) raise new needs for dedicated authenticated ciphers.

As discussed before in detail, primary goals of dedicated authenticated encryption schemes are to provide confidentiality as well as data authenticity and integrity, when only using one key and primitive.

Besides these basic goals, practical applications demand for an additional requirement: *Associated Data*. Authenticated ciphers shall be able to ensure data authenticity on a part of the sent message (the associated data) that is actually not encrypted. Figure 2.2 illustrates the concept of such a cipher, named *AEAD - Authentic Encryption with Associated Data*. Such a cryptographic system takes as input the plaintext message M and an encryption key K as well as authenticated data AD and outputs the ciphertext C and an authentication tag T. Even though T authenticates P and AD, C does not contain the encrypted representation of AD.

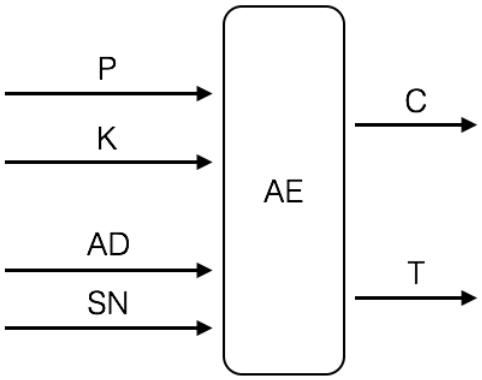

Figure 2.2: Basic structure of an authenticated encryption cipher. P...plaintext, C...ciphertext, K...secret key used for encryption and authentication, AD...associated data, SN...secret message number (optional), T...authentication tag

A very popular example to justify the need for the support of authenticated data is encrypted network traffic. In computer networking, data packets are exchanged between clients on different networks. These packets contain, besides the actual data exchanged, header information that is needed by different protocols to ensure a reliable transportation of the packets. This header information is read by network devices (routers, switches, etc.) to make sure the packet arrives at it's intended destination. If this network transfer now needs to be secured, the encryption can only be applied to the packet data and not it's header information, since this needs to be read by devices in transit to deliver the packet. Even though confidentiality is not a requirement for this header information, the receiver might want to ensure integrity and authenticity on this header information. Therefore, it might be treated as associated data resulting in the desired authenticity and integrity.

This sums up to form the four main requirements of a *AEAD* schemes:

- **Confidentiality** of the plaintext encrypted using the cipher. Nobody should be able to extract information from the encrypted message without the corresponding secret key.
- **Integrity** of sent data. It must not be possible for an attacker to alter transmitted messages without the receiver detecting it.
- **Authenticity** of transmitted ciphertext. It has to be ensured, that the data received was sent by the communication partner sharing a secret key.
- **Authenticity** of appended **associated data**. It must not be possible for an attacher to alter attached associated data event though this data is not encrypted.

Further, there exist certain requirements regarding the implementation of these primitives such as:

- Only one key shall be required to ensure both, confidentiality and authenticity.
- The design shall be single-pass.
- The cipher shall show good properties for implementing it in hardware.
- The performance of the cipher has to be considered.

2.4 CAESAR Challenge

In order to find, analyse and select new dedicated authenticated cipher designs fulfilling the before mentioned requirements for the next decade, the international cryptologic research community initiated a new cryptographic competition for authenticated ciphers called *CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness)*. The goal of this competition is to find a new portfolio of AEAD encryption schemes showing advantages over *AES-GCM*. Even though standards committees might show interest in this portfolio and standardize parts of it, the CAESAR challenge itself does not drive standardisation of any cipher [1].

The requirements in the CAESAR - call for submission requests authenticated cipher designs in the form of a cipher specification, security goals and security analysis parts. Further, a C reference implementation for a specified interface has to be provided. As in every cryptographic competition, the cryptographic community examines the different submission with the goal of breaking weak designs and establishing deep analysis on robust designs. The CAESAR committee will then select a portfolio of winner designs and provide justification for their decisions [1].

There were 55 ciphers submitted to the competition in the first round. At the time of writing, only 7 of those ciphers have been broken or withdrawn in about 6 months

of analysis time. This generally indicates the high quality of the submitted ciphers. The general trend of increased quality of submitted ciphers and a higher number of submissions to cryptographic competitions in general, dramatically increases the need for automated analysis methods and tools. We refer to Section 1.3 for more general information about cryptographic competitions and the problem addressed by automatic analysis methods.

The submissions of the competition are generally categorized by [18]:

- Maintype ∈ {Block Cipher, Permutation, Other}
- Subtype ∈ {Unmodified AES, N-round AES, AES-like, Other Block Cipher, Sponge Construction Based, FSR, ARX, LRX}
- Parallelizable ∈ {Fully, Partly, No}
- Online ∈ {Fully, Partly, No, Nonce MR}
- Nonce Misuse Resistance ∈ {MAX, MAX Online, LCP+X, A+N, None}
- Inverse Free ∈ {Yes, No, N/A}

2.5 Design Principles and Approaches

Many different designs have been submitted to the CAESAR challenge. The variety of design principles ranges from new block cipher modes of operation over sponge constructions and ARX / LRX ciphers to feedback shift registers or variants of AES.

Since this thesis focuses on three submissions based on the sponge construction, the basic design principles are described here.

2.5.1 The Sponge Construction

In general, the *sponge constructions* is a cryptographic structure that can be used to construct various cryptographic primitives such as MACs, hash functions, block ciphers, stream ciphers, pseudo random number generators and authentic encryption ciphers [10]. It is a simple iterated construction for forming a variable input-length, variable output-length function F based on a fixed input/output length *transformation function* f. f operates on a fixed size *state* S consisting of two parts, c the so called *capacity*, and r the *rate*. The size of S is called *width* and defined by $b = c + r$ [6].

The sponge construction performs it's operation in three phases [6]:

1. **Initialization Phase**: In this phase, the state S is initialized to zero and the input I is padded and cut into blocks of size r.

2. **Absorbing Phase**: In this phase the construction *absorbs* the input message. This is done by XORing the first r bits of the current state S_i with the current padded input block I_i (size of r due to padding in step 1). After this, the transformation function f is applied to S_i, resulting in the state S_{i+1}. This procedure is repeated until all the padded input blocks have been absorbed into the state S.
3. **Squeezing Phase**: In this phase, parts of S are *squeezed* out of the state S_i and form parts of the output message O. After r bits have been extracted, the transformation function f is applied to the state. This procedure is repeated until an output message O of desired length has been extracted.

Figure 2.3 illustrates the basic structure of the sponge construction.

Note that the capacity part of the state is never directly manipulated through the absorbing or squeezing phase. So no parts of the input message I are XORed to the capacity of the state and no parts of the capacity are extracted in the squeezing phase. Although the capacity is part of the in- and output of the transformation function f. Further, the steps of this basic construction might vary depending on the application (and cryptographic primitive formed using the sponge construction). The basic design described here was initially intended to be used as a basic structure for a hash function which corresponds to the above described phases [6]. A sponge construction is dependent on the following parameters:

- r: the rate of the state S
- c: the capacity of the state S
- f: the transformation function
- *pad*: the applied padding operation

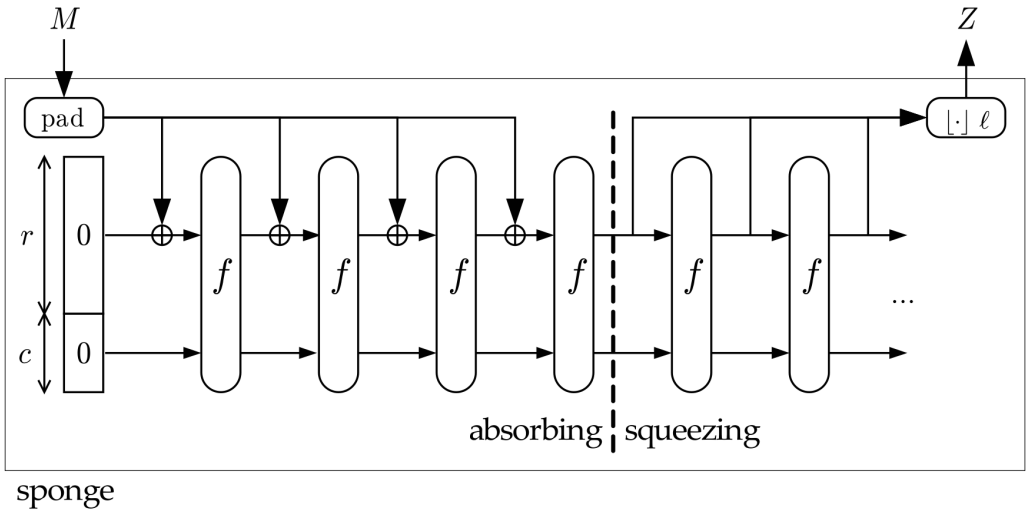

Figure 2.3: The basic sponge construction. [10]

2.5.2 The Duplex Construction

The duplex construction states a very similar structure to the sponge construction described before. The main difference between the two principles is that the sponge construction absorbs the full input length in the *absorption* phase and extracts the whole output message in the *squeezing* phase. In contrast to that, the duplex construction features an operation, in which the injection of an input block and extraction of an output block is performed in one step. Thus, a duplex construction supports the operation $duplexing(\sigma_i, l)$, where σ_i donates the ith input block and l represents the number of requested output bits. This operation returns a bit string Z_i with length $|Z_i| = l$ [10]. As explained before, σ_i is injected into and Z_i is extracted from the rate r of the internal state S. Between the injection and extraction, the transformation function f is applied to S. Figure 2.4 illustrates this process [6]. The duplex construction is dependent on the same parameters as the sponge construction, while only the phases of execution (injection / extraction) are different.

Note that the maximum value of l is defined by the size of the rate r. Further, the duplex construction has no defined phases such as *absorption* or *squeezing* and the value of the output Z_i depends on all messages $\sigma_0 \dots \sigma_{i-1}$ previously injected into the rate. The construction therefore can be seen as obtaining an internal state dependent on message parts injected [10].

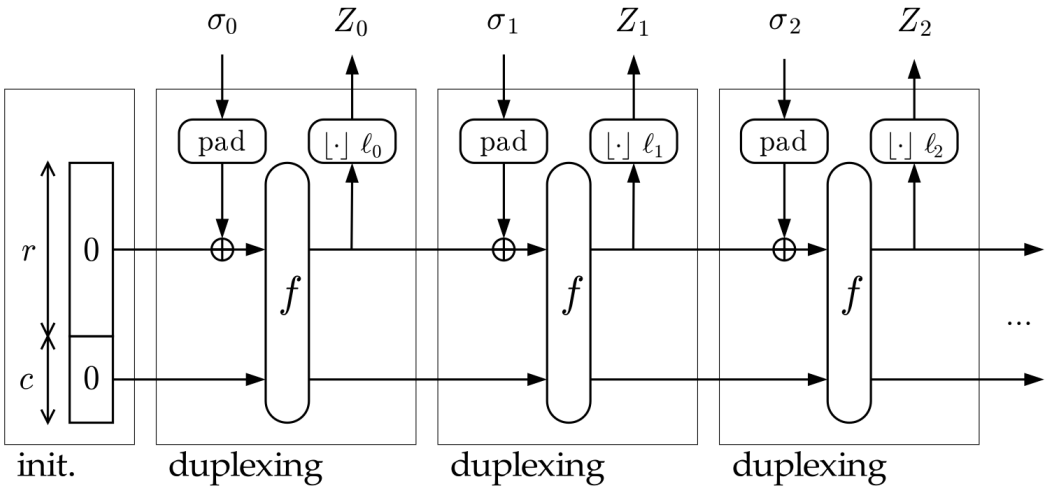

Figure 2.4: The basic duplex construction [6].

2.5.3 The Monkey Duplex Construction

The monkey duplex construction now provides a way of how a duplex construction can be used to construct an authenticated cipher. In the previous structures discussed,

the initialisation of the state was performed setting it to a fixed value (zero). Since in authenticated encryption, the values need to be dependent on the secret key, the monkey-duplex construction defines a way of initializing the state to be key dependent. Therefore, the operation $initialize(K, N)$ is defined and initializes the state, where K names the secret key, and N is a nonce unique to this encryption process. As in every nonce based construction, the uniqueness of the pair (K, N) is required to ensure security of the cipher. Unlike the duplex construction, the monkey duplex construction supports two types of round transformation calls for initialisation and regular state transformation. These two calls vary in number of iterations of f applied to the state. The structure of the monkey duplex construction is illustrated in Figure 2.5. [7] The monkey duplex construction is dependent on the parameters:

- r: the rate of the state S
- c: the capacity of the state S
- f: the transformation function
- pad: the applied padding operation
- l_{key}: the length of the key
- l_{nonce}: the length of the unique nonce
- n_{init}: the iteration number of f after the initialization. Thus, the function f is applied n_{init} time after the state has been initialized.
- n_{duplex}: the iteration number between duplex operations.

This structure can be used to construct authenticated cipher schemes. Therefore, the ciphertext produced during encryption is donated by the output Z_i, whereas the authentication tag is represented by the state after the last transformation.

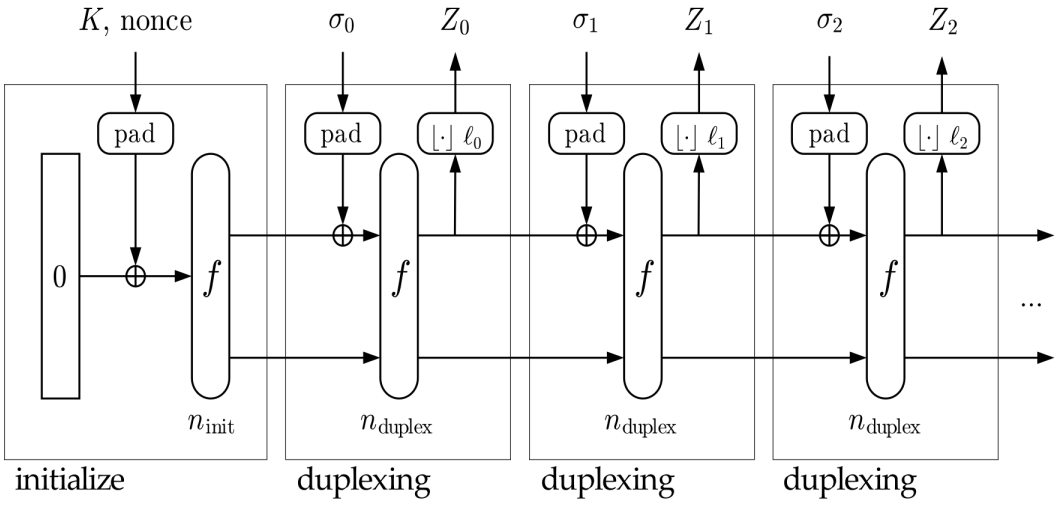

Figure 2.5: The monkey duplex construction. [7]

2.6 Attacks

When analysing or attacking authenticated ciphers, one has to distinguish between two basic types of attacks / goals of the analysis:

- **key recovery**: In this kind of attack, the goal is to reveal the secret key used for encryption and authentication (or parts of it).
- **forgery**: In a forgery attack, an attacker tries to break authenticity of the scheme. Therefore, this type of attack does not aim for revealing the secret key, but tries to forge authenticity of a message. Assume a message M is encrypted using an authenticated encryption scheme with the key K resulting in the ciphertext C and the authentication tag T. The goal of this attack now is to efficiently find and / or construct a message M' or ciphertext C' resulting in the same authentication tag T. In this case an attacker can break authenticity since the same authentication tag is delivered for different messages or ciphertexts.

The analysis methods in the scope of this thesis all aim towards forgery of messages. This is based on the assumption that the examined schemes (based on their design principles) provide a high level of security against key recovery attacks. Further, analysis with the goal of being able to forge messages is much more specific to this special form of encryption schemes and differs from standard cryptanalysis.

2.7 Analysed Ciphers

This thesis performs cryptanalysis using an automated analysis framework for three submissions to the CAESAR competition. This section describes these ciphers and their structure. Note that all further explanation and details about the performed analysis and the created automated analysis framework will relate to NORX. Hence, NORX is described in detail and we mainly refer to the submission documents of the other ciphers MORUS and Ketje.

2.7.1 NORX

NORX is a design based on the previously described MonkeyDuplex construction with arbitrary degree of parallelism and authentication tag length. NORX uses operations like shifts, AND, XOR and rotations. The addition operation is avoided due to hardware implementation benefits and easy application of cryptanalysis methods. Figure 2.6 illustrates the basic structure of NORX. We refer to [4] for a detailed specification.

The authors propose five different instances of NORX, which differ in word size (W), number of rounds (R) for the transformation function F^R and parallelism degree (D). Table 2.1 defines the proposed combinations.

W	64	32	64	32	64
R	4	4	6	6	4
D	1	1	1	1	4

Table 2.1: Different proposed instances of NORX based on word size (W), round number (R) and parallelism degree (D).

The authenticated cipher can be run in two different modes:

a) **Encryption Mode**: Input to this mode is a $4W$ bit key K, a $2W$ bit nonce N and an input message $M = H||P||T$ consisting of:

- a header H
- a plaintext payload P
- a trailer T

It outputs a ciphertext C (only the payload P is encrypted) and an authentication tag A authenticating the whole message M.

b) **Decryption Mode**: Input to this mode is a $4W$ bit key K, a $2W$ bit nonce N and an input message $M = H||C||T$ consisting of:

- a header H
- a ciphertext payload C
- a trailer T
- an authentication tag A authenticating M

It outputs P (the plaintext payload) if the verification $verify(H, T, C, A)$ is successful and an error symbol otherwise.

Even though NORX can be configured to process data in parallel, the following description is based on the sequential case (parallelism parameter $D = 1$).

Basic Representation and Principles

In NORX, the state is represented as a concatenation of 16 statewords $S = S_0||S_1||S_2||...||S_{15}$, which are arranged as a 4x4 matrix. The NORX state consists of a rate of ten statewords $S_0...S_9$, where the capacity accounts for the remaining six statewords $S_{10}...S_{15}$. The resulting values of rate and capacity of the state in the different versions of NORX are listed in Table 2.2.

The transformation function (round function) of the MonkeyDuplex construction, as described in Section 2.5.3, is donated by the function F and F^R donates R applications of F to its input state S.

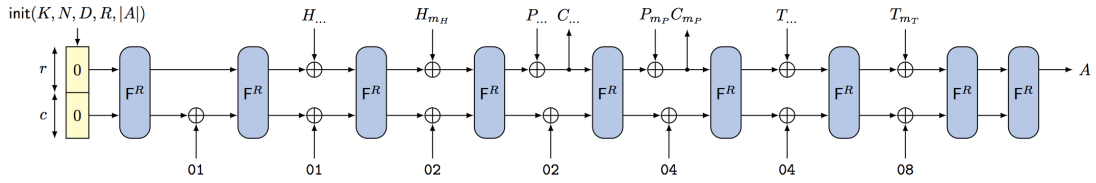

Figure 2.6: Overview over the NORX authenticated cipher.

	NORX64	NORX32
b	1024	512
r	640	320
c	384	192

Table 2.2: NORX capacities and rates for different word sizes of it.

The round function

The round-function F is defined by applying the transformation $G(a, b, c, d)$ eight times to different words of the state. These eight applications of G are graphically illustrated in Figure 2.7. It is split up into four column steps as well as four diagonal steps.

$$G(S_0, S_4, S_8, S_{12}) \ , \ G(S_1, S_5, S_9, S_{13}) \tag{2.16}$$
$$G(S_2, S_6, S_{10}, S_{14}) \ , \ G(S_3, S_7, S_{11}, S_{15}) \tag{2.17}$$
$$G(S_0, S_5, S_{10}, S_{15}) \ , \ G(S_1, S_6, S_{11}, S_{12}) \tag{2.18}$$
$$G(S_2, S_7, S_8, S_{13}) \ , \ G(S_3, S_4, S_9, S_{14}) \tag{2.19}$$

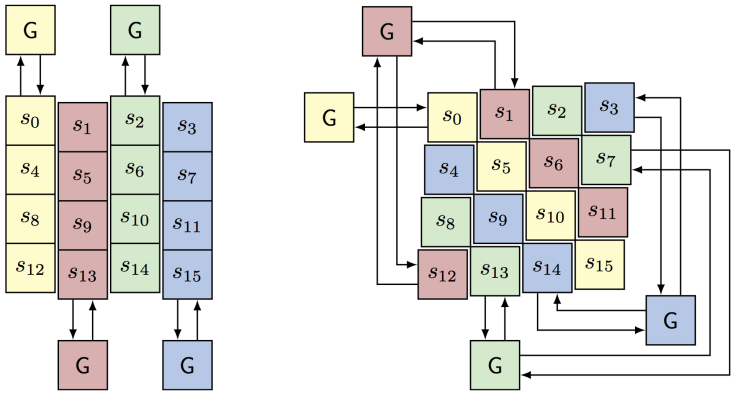

Figure 2.7: Application of $G(a, b, c, d)$ to different parts of the state S. [4]

The function $G(a, b, c, d)$ is defined in Equations 2.20 to 2.27. As mentioned before, NORX avoids additions. Instead, an addition $A + B$ is approximated by $A \oplus B \oplus ((A \wedge B) \ll 1)$ (based on the identity $A + B = (A \oplus B) + ((A \wedge B) \ll 1)$).

$$a \longleftarrow (a \oplus b) \oplus (a \wedge b) \ll 1 \tag{2.20}$$
$$d \longleftarrow (a \oplus d) \ggg r_0 \tag{2.21}$$
$$c \longleftarrow (c \oplus d) \oplus (c \wedge d) \ll 1 \tag{2.22}$$
$$b \longleftarrow (b \oplus c) \ggg r_1 \tag{2.23}$$
$$a \longleftarrow (a \oplus b) \oplus (a \wedge b) \ll 1 \tag{2.24}$$
$$d \longleftarrow (a \oplus d) \ggg r_2 \tag{2.25}$$
$$c \longleftarrow (c \oplus d) \oplus (c \wedge d) \ll 1 \tag{2.26}$$
$$b \longleftarrow (b \oplus c) \ggg r_3 \tag{2.27}$$

Basic Operations

This describes the process on encrypting and decrypting/verifying a message as well as initializing the state. The basic operations are:

- **Padding**: NORX generally uses *multi-rate* padding to pad input data and messages to a multiple of the rate r. This is necessary since data is only injected (extracted) into (from) the rate of the state.
- **Domain-Separation**: To omit symmetric properties and prevent the application of sliding attacks, NORX facilitates domain separation by injecting domain specific constants before certain operations are conducted. This constant is injected into S_{15} of the state (capacity).
- **Initialization**: The initialisation is basically the process of establishing an initial state S dependent on the nonce N and the key K. The NORX-Initialisation takes place in three steps:
  1. Basic Setup: The state is initialized to a set of constant values dependent on parts of the key K, the nonce N and defined constants. We refer to [4] for a detailed description of the process.
  2. Parameter Integration: In this step, the previously described parameters D, R, W and $|A|$ are injected into one word of S. After the integration the round function is applied R times.

$$S_{14} \longleftarrow S_{14} \oplus ((R \ll 26) \oplus (D \ll 18) \oplus (W \ll 10) \oplus |A|) \tag{2.28}$$
$$S \longleftarrow F^R(S) \tag{2.29}$$

3. Finalisation: Finally, a domain constant v, which depends on the operation performed after the initialisation (see [4] for the domain separation constants of different operations) is integrated into the state and the round function is applied R times.

$$S_{15} \longleftarrow S_{15} \oplus v \tag{2.30}$$
$$S \longleftarrow F^R(S) \tag{2.31}$$

- **Message Processing**:

1. Header Processing: The header data H is padded to a multiple of r resulting in $pad_r(H) = H_0||H_1||H_2||...||H_{n_H-1}$ with $|H_i| = r \ \forall i \in \{0, ..., n_H - 1\}$. Each of theses blocks H_i is split up into $r/W = 10$ words $H_l = h_{l,0}||...||h_{l,9}$ with $|h_i| = W \ \forall i \in \{0, ..., 9\}$ and injected into the state in one step.

$$S_j \longleftarrow S_j \oplus h_{l,j} \ , \ for \ 0 \leq j \leq 9 \tag{2.32}$$
$$S_{15} \longleftarrow S_{15} \oplus v \tag{2.33}$$
$$S \longleftarrow F^R(S) \tag{2.34}$$

2. Payload Processing: The payload data P is padded to a multiple of r resulting in $pad_r(P) = P_0||P_1||P_2||...||P_{n_P-1}$ with $|P_i| = r \ \forall i \in \{0, ..., n_P - 1\}$. Each of theses blocks P_i is split up into $r/W = 10$ words $P_l = p_{l,0}||...||p_{l,9}$ with $|p_i| = W \ \forall i \in \{0, ..., 9\}$. The payloadblock P_l is then encrypted to a ciphertextblock $C_l = c_{l,0}||...||c_{l,9}$ by injecting plaintext blocks and extracting ciphertext blocks.

$$S_j \longleftarrow S_j \oplus p_{l,j} \ , \ for \ 0 \leq j \leq 9 \tag{2.35}$$
$$c_{l,j} \longleftarrow S_j \tag{2.36}$$
$$S_{15} \longleftarrow S_{15} \oplus v \tag{2.37}$$
$$S \longleftarrow F^R(S) \tag{2.38}$$

3. Ciphertext Processing: The ciphertext data C is padded to a multiple of r resulting in $pad_r(C) = C_0||C_1||C_2||...||C_{n_C-1}$ with $|C_i| = r \ \forall i \in \{0, ..., n_C-1\}$. Each of theses blocks C_i is split up into $r/W = 10$ words $C_l = c_{l,0}||...||c_{l,9}$ with $|c_i| = W \ \forall i \in \{0, ..., 9\}$. The ciphertextblock C_l is then decrypted to a plaintextblock $P_l = p_{l,0}||...||p_{l,9}$ by extracting plaintextblocks and injecting ciphertext blocks.

$$p_{l,j} \longleftarrow S_j \oplus c_{l,j} \ , \ for \ 0 \leq j \leq 9 \tag{2.39}$$
$$S_j \longleftarrow c_{l,j} \tag{2.40}$$
$$S_{15} \longleftarrow S_{15} \oplus v \tag{2.41}$$
$$S \longleftarrow F^R(S) \tag{2.42}$$

Note that this operation is substantially different from the payload processing operation since now, state words are extracted and used for decryption while the ciphertext received overwrites the rate of the state in order to produce and equal state as at the same point during the encryption.

4. Trailer Processing: The trailer data T is padded to a multiple of r resulting in $pad_r(T) = T_0||T_1||T_2||...||T_{n_T-1}$ with $|T_i| = r \ \forall i \in \{0, ..., n_T - 1\}$. Each of theses blocks T_i is split up into $r/W = 10$ words $T_l = t_{l,0}||...||t_{l,9}$ with $|t_i| = W \ \forall i \in \{0, ..., 9\}$ and injected into the state in one step.

$$S_j \longleftarrow S_j \oplus h_{l,j} \ , \ for \ 0 \leq j \leq 9 \tag{2.43}$$

$$S_{15} \longleftarrow S_{15} \oplus v \tag{2.44}$$

$$S \longleftarrow F^R(S) \tag{2.45}$$

- **Tag Handling**:

1. Tag Generation: To generate the authentication tag A after H, P and T have been processed, the round function F is again applied R times. The $|A|$ least significant bits of the rate of the resulting state S ($S_0||...||S_9$) is set as authentication tag A.

2. Tag Verification: To verify a received authentication tag A', an authentication tag A based on the received data (H, C, T) is generated during the decryption process using the same tag generation process as stated before. If $A' = A$ holds the tag A' can be verified. It is important that while performing verification, no information is leaked to a potential attacker.

2.7.2 MORUS

MORUS names an authenticated cipher based on sponge constructions operating on a relatively large state of 640 or 1280 bits. It supports keys of length 128 and 256 bits when using a round transformation only based on XOR, Shifts and AND operations. For detailed information about MORUS, we refer to the submission document in [29].

2.7.3 KETJE

Ketje is a sponge based authenticated cipher design facilitating the *MonkeyDuplex* construction. It supports relatively small states sizes of 200 and 400 bits with a rate of 8% of the state. The basic permutation is derived from *KECCAK* [8], where a tunable number of rounds are supported. For detailed information about KETJE, we refer to the submission document in [9].

Chapter 3

Cryptanalysis

This chapter is going to introduce the basic ideas and principles of different kinds of cryptanalysis with a focus on differential cryptanalysis. In general, cryptanalysis names the study of mathematical techniques with the goal of breaking cryptographic systems or finding weaknesses that reduce their security level [21].

So when cryptography, as defined in Chapter 1, names the study of creating systems that solve cryptologic problems, cryptanalysis names the discipline of breaking such systems. There exist various different ways and techniques of how to conduct cryptanalysis to different cryptographic primitives. This chapter gives a short overview of common approaches and provide a more detailed understanding of the technique applied in the scope of this thesis: differential cryptanalysis.

3.1 Goals of Cryptanalysis

As mentioned before, the overall goal of cryptanalysis is to break a cryptographic primitive or find a way to lower it's security level. Although, cryptanalysis can have various different and diverse goals dependent on the component or system attacked.

In general one can distinguish between two basic types of attacks on a cryptographic system:

- **Passive**: An attacker observes the system, but does not take any actions of influencing the cipher or device executing the cryptographic primitive.
- **Active**: An attacker actively influences a system in it's execution in order to attack it.

When performing an attack, we further have to distinguish between various attack models based on the degree of information an attacker has about inputs / outputs to the cipher under attack [21] [28]:

- **Ciphertext-Only Attack**: The attacker tries to recover the key when only having access to the ciphertext generated by the cipher attacked.
- **Known-Plaintext Attack**: The attacker has a set of plaintexts and corresponding ciphertexts (C_i, P_i) for the cipher under attack.
- **Chosen-Plaintext Attack**: The attacker can define plaintexts and is able to obtain the corresponding ciphertexts.
- **Adaptive-Chosen-Plaintext Attack**: Is a chosen-plaintext attack where the chosen plaintexts might depend on ciphertexts previously obtained from the cipher under attack.
- **Chose-Ciphertext Attack**: The attacker chooses the ciphertext and is able to obtain the corresponding plaintext to it.
- **Adaptive-Chosen-Ciphertext Attack**: Is a chosen-ciphertext attack where the chosen ciphertexts might depend on plaintexts previously obtained from the cipher under attack.

Note that even though an attacker can gain different information in different attack models, all considerations happen under the basic assumption of *Kerckhoff's Principle* [21]. Besides other assumptions, this states that the whole cryptographic system is assumed to be known to an attacker. A cryptographic systems must be secure, if everything but the used secret key is public. Systems hiding their specification must be considered insecure.

To explain the basic idea and goals of cryptanalysis, let's assume a cipher TOY (definition publicly available), which encrypts messages blocks M_i of size W using a key K of size $2W$ resulting in a ciphertext C of size W. This simple cipher is illustrated in Figure 3.1. In the following, the operation $C_{K_i} = E_{K_i}(P)$ donates the encryption of a message P with a certain key K_i.

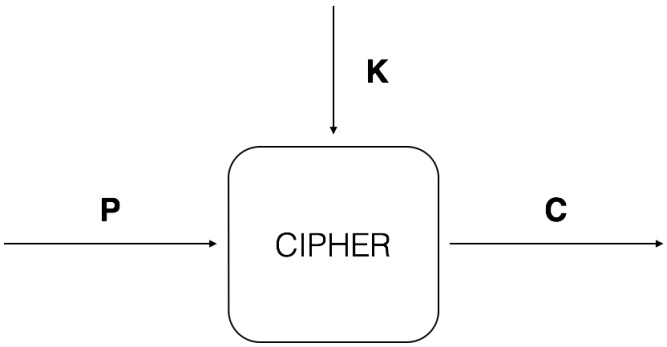

Figure 3.1: Basic setting when attacking the cipher TOY. $|C| = W, |P| = W, |K| = 2W$.

An attacker (or cryptanalyst) now tries to break this system by revealing it's secret key

in any of the above mentioned attack models. To keep it simple, let's assume a known plaintext attack. So the attacker knows at least one pair of ciphertext and corresponding plaintext (C_0, P_0).

The simplest way to attack the cipher is to perform a so called *brute-force* attack. Here the attacker would try all 2^{2W} different values of the secret key K (K_i where $i \in \{0, ..., 2^W - 1\}$ and perform an encryption operation on the known plaintext P_0 resulting in $C_{K_i} = E_{K_i}(P_0)$. If $C_{K_i} = C_0$ holds, then K_i is a candidate for the key K. This candidate can be verified using an additional plaintext/ciphertext pair. Therefore, two plaintext / ciphertext pairs are required in this attack to recover the right secret key.

This form of attack is of course practically infeasible, since the attacker needs to perform 2^{2W} encryption operations (worst case) which should take an extremely long time (dependent on the key length $2W$). As an example assume a cipher with a key length of $2W = 128$ bits. An attacker would need to calculate $2^{128} \approx 3.4 \cdot 10^{38}$ encryption operations. When assuming an attacker conducts this attack in hardware that can calculate $2 \cdot 10^7$ encryption operations per second, the attack would take (worst case)

$$\frac{3.4 \cdot 10^{38}}{3.15569 \cdot 10^7 \cdot 2 \cdot 10^7} = 5,387 \cdot 10^{37} \tag{3.1}$$

years.

The goal of cryptanalysis now is to find a method of breaking TOY that requires less (and as few as possible) operations than brute-force $n_a < 2^{2W}$.

3.2 Analysis Methods

Different approaches for reaching this goal have been established in the history of cryptography and cryptanalysis. Methods reach from pure statistical analysis of ciphertext, usage of certain symmetric properties of ciphers to methods that evaluate inner states of ciphers and therefore narrow down the potential key-space.

In this thesis, three of the most important analysis methods - differential, linear and impossible differential cryptanalysis - are described. All these methods exploit statistical properties of the cipher under attack in different forms. The focus in this case lies on differential cryptanalysis, since this forms the method used to demonstrate the automated analysis framework in this thesis (see Chapter 4 and 5).

3.2.1 Differential Cryptanalysis

The basic idea of differential cryptanalysis is to analyse a cipher not using concrete values known x(such as for example a value for P or C), but using differential values of such known parameters. A difference is a relational measure between two values of the same

variable or parameter with respect to a certain operation. For example if two plaintext values P_1 and P_2 in the above described cipher are given, the difference with respect to the XOR operation can be calculated as $\triangle P = P_1 \oplus P_2$. The operation used for the construction of such differences can be chosen according to the requirements. Different common types of operations are explained below.

Differential cryptanalysis is, in general, a chosen plaintext attack. So the attacker is able to encrypt arbitrary plaintexts P_x and obtain the corresponding ciphertext C_x.

Types of Differentials

The term *differential* can almost be applied to any operation Ξ applied to two different values of the same parameter within a cipher. The type of differential in the cipher is chosen according to the characteristics and properties that can be exploited or used. The following forms of differentials are very common:

- XOR Differences: Here the difference is stated as applying the bit-wise XOR operation (for each bit i) on the input values as illustrated below.

$$\triangle x = x_1 \oplus x_2 \Leftrightarrow \triangle x_i = x_{1_i} \oplus x_{2_i} \tag{3.2}$$

- Modular Differences: Here the difference is calculated by applying the modular subtraction operation on two words x_1 and x_2 as illustrated below.

$$\triangle x = x_1 - x_2 \bmod n \mid n = 2^W - 1 \tag{3.3}$$

- Signed Differences: In this type of difference, three different states (-1, $+1$, 0) are distinguished for each bit instead of two (as with XOR differences). Therefore, it is defined on a bit level as described below.

$$\triangle x_i = \begin{cases} 0 & , x_{1_i} = x_{2_i} \\ -1 & , x_{1_i} > x_{2_i} \Leftrightarrow x_{2_i} - x_{1_i} \\ +1 & , x_{1_i} < x_{2_i} \end{cases} \tag{3.4}$$

- Multi-Bit Conditions: Here, differences in a single bit are calculated using conditions based on other bits in the words. This dependency on up to n bits from a condition function f is illustrated below.

$$\triangle x_i = f(x_{1_0}, \ldots, x_{1_{n-1}}, x_{2_0}, \ldots, x_{2_{n-1}}) \tag{3.5}$$

Behaviour of Differentials

This section briefly gives an overview of how differentials behave when being applied to different operations often found in the construction of ciphers. This section covers the behaviour of XOR differences, since they are used in analysis methods further discussed in this thesis. In all of the following examples, x_1 and x_2 donate two different values of the same parameter where $\triangle x = x_1 \oplus x_2$ is their difference (in this case according to the XOR operation). Further, $\triangle x_o$ states the difference after the operation has been applied.

- **Shifts, Rotations and Bit Reordering**: When applying change to the bit-order differentials propagate with a probability of 1 as well, since only the same bits in the two components are changes, their relation is left unchanged, since the same permutation is applied to both components the same way. So the permutation is applied to the difference. Equation 3.6 illustrates this in a generic operation *permute* that can be replaced with any bit permutation.

$$\triangle x_o = permute(x_1) \oplus permute(x_2) = permute(x_1 \oplus x_2) = permute(\triangle x) \quad (3.6)$$

- **Linear Maps**: The same applies for linear maps as illustrated in 3.7 (L state the application of a linear map).

$$\triangle x_o = L(x_1) \oplus L(x_2) = L(x_1 \oplus x_2) = L(\triangle x) \quad (3.7)$$

  A very important property of differentials in the context of linear maps is that they are invariant to the application of the XOR operation with constants. This means that differences propagate through this type of operation with a probability of 1 according to Equation 3.8.

$$\triangle x_o = (x_1 \oplus K) \oplus (x_2 \oplus K) = x_1 \oplus x_2 = \triangle x \quad (3.8)$$

  This property is especially important since keys in ciphers are often added this way. This property therefore allows the investigation of cipher properties after such a key addition.

- **Non-Linear Maps**: For non-linear maps, the situation is quite disparate. Here an input difference to this non-linear map is transformed in a way, so that the output of the map can not be determined only dependent on the differential. This is the case, since the same difference can be produced using varying pairs of values and the map might behave unequal for each of those values. Therefore, an input difference can propagate to various output differences with a certain probability. This is expressed using the *Difference Distribution Table*, as shown in an example later.

Example and Procedure

To explain the basic idea of differential cryptanalysis on a fairly easy example (adapted from [17]), let's refine the specification of TOY and look at it's internal structure. Fig-

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
S[x]	6	4	c	5	0	7	2	e	1	f	3	d	8	a	9	b

Table 3.1: Substitution S[x]

ure 3.2 shows the internal structure of the cipher. It is a Substitution-Permutation-Network (SP-Network). This ciphers generally consist of non-linear substitution layers (e.g. S-boxes) and linear permutation (of diffusion) layers. In this simple example, no permutation layer is present and the cipher consists of the XOR operation of messages with a key, as well as an S-Box performing the substitution. As it can be seen in Figure 3.2, the key is split into two sub-keys $K = K_1||K_2$. Those sub-keys are XORed to the message encrypted before and after the substitution using the S-Box. This substitution is simply implemented as a lookup table as illustrated in Table 3.1.

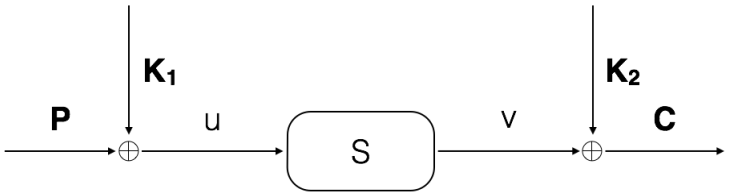

Figure 3.2: Internal structure of the example cipher.

Lets assume a bit-length of $W = 4$ in this scenario resulting in a key length of $|K| = 2W = 8$ bit and a block length of $|P| = |C| = W = 4$ bit. In this example a known-plaintext attack is conducted to 3.10) with the goal of revealing the secret key K in less operations (and in a smarter way) than brute-force using differential cryptanalysis. Further, in this example XOR-differences and the following plaintext / ciphertext pairs were used:

$$P_1 = \mathtt{5} \longrightarrow C_1 = \mathtt{F} \tag{3.9}$$

$$P_2 = \mathtt{A} \longrightarrow C_2 = \mathtt{0} \tag{3.10}$$

The cipher design sets a pretty bad situation for a cryptanalyst, since even though two pairs of P (P_1 and P_2) and C (C_1 and C_2) are known we cannot determine the value of u or v internal to the cipher, since these values are fully dependent on the key. No statement about K_1, K_2 and the internal variables v and u can be made.

So the goal at this stage has to be to gain further information about v and u in order to reveal information about the secret key K. If we were able to determine specific values for v (v_1 or v_2) and u (u_1 or u_2) the key could immediately be revealed by calculating $K_1 = u_1 \oplus P_1 = u_2 \oplus P_2$ and $K_2 = v_1 \oplus C_1 = v_2 \oplus C_2$.

This is where differential cryptanalysis states a huge advantage. A very important observation can be made when taking a look at the differences of the two plaintext pairs after the key has been added to them.

$$u_1 = P_1 \oplus K_1 \tag{3.11}$$
$$u_2 = P_2 \oplus K_2 \tag{3.12}$$
$$\triangle u = u_1 \oplus u_2 = (P_1 \oplus K_1) \oplus (P_2 \oplus K_1) = P_1 \oplus P_2 = \triangle P \tag{3.13}$$

As it can be seen, the sub-key K_1 has no influence on the value of $\triangle u$ (the differential of u) anymore. Even though we can not determine the concrete values of u_1 and u_2, we have significant information about it's differential behaviour.

The same procedure can be applied to the given ciphertext difference $\triangle C$ and the internal variable $\triangle v$ as shown below.

$$C_1 = v_1 \oplus K_2 \Leftrightarrow v_1 = C_1 \oplus K_2 \tag{3.14}$$
$$C_2 = v_2 \oplus K_2 \Leftrightarrow v_2 = C_2 \oplus K_2 \tag{3.15}$$
$$\triangle v = v_1 \oplus v_2 = (C_1 \oplus K_2) \oplus (C_2 \oplus K_2) = C_1 \oplus C_2 = \triangle C \tag{3.16}$$

At this point, we know the differences $\triangle u$ and $\triangle v$ before and after the substitution $S[x]$. Since u and v themselves cannot be calculated, the attack proceeds in the direction of narrowing down the set of possible values for v and u based on the information known through the differences of plaintext-and ciphertext-pairs. Since information before and after $S[x]$ is present, it seems reasonable to further investigate properties of this substitution.

Differential Distribution Table (DDT)

One important property of such substitutions used in differential cryptanalysis is expresses in the differential distribution table. It is based on the observation, that input differences to a substitution can result in a set of output differences with a distinct probability. In the context of differential cryptanalysis, this property is used to obtain additional information about variables within a cipher when differential properties are given.

In a substitution every possible input value is mapped to another fixed output value as illustrated in Table 3.1. Therefore, without knowing the concrete value of input / output no information about the output / input is available to an attacker. But the situation changes, when observing which input differences map to which output differences.

First, we have to note that every input difference $\triangle i = i_1 \oplus i_2$ to such an S-Box can be constructed in different ways using different values for i_1 and i_2. For example, the difference $\triangle i = 0x01$ can be constructed by either $i_1 = 0x01$, $i_2 = 0x00$ or $i_1 = 0x11$, $i_2 = 0x01$.

Secondly it has to be stated, that an input difference $\triangle i$ can map to various output differences dependent on which values i_1 and i_2 were used to construct it. Therefore, $\triangle i$ only maps to a specific $\triangle out$ with a certain probability $p_{i,o}$. Note that all input differences need to map to some output difference, where some combinations might be impossible ($p = 0$). Thus, the sum over the probabilities of all output differences for a specific input difference has to be 1.

What is now specific about each substitution is which input differences $\triangle i$ can possibly map to a specific output difference $\triangle o$ with a certain probability. Among those possible combinations, some will occur with a significantly high or low probability p leaking very significant and important information. More specific, the constraints on which input differences can map to which output differences limit the possible concrete values for input and output occurring within the cipher. So if a specific $\triangle u$ and $\triangle v$ are given for an S-Box, the constraints on the differential propagation through the S-Box narrow down the set of possible values for u and v and are therefore very useful for an attacker.

To this point, we have only been able to establish statements regarding difference for some points within the cipher. Our goal was to reveal specific values for u and v in order to discover the key K. The information leaked through the DDT of the S-Box now establishes a connection between differential relations / behaviour and the set of possible values for u and v. This information will be used to break the cipher in the following.

Note that it is impossible to create an S-Box not showing the discussed properties. If an attacker is able to construct a certain input and output difference at an S-Box in a cipher, it leaks information which can be used in an attack. Thus, the goal of cipher designs is to make it difficult to determine the differences for input and output at a certain S-Box with sufficient high probability.

The *IN-Set* of a substitution names the set of input variables, in which a given input difference can be injected and which causes a given difference at the output of the substitution. We can define the IN-Set of an input difference $\triangle X$ and an output difference $\triangle Y$ as illustrated in Equation 3.17.

$$IN(\triangle X, \triangle Y) = \{X : S[X] \oplus S[X + \triangle X] = \triangle Y\} \tag{3.17}$$

The information about differential behaviour regarding input- and output-differences of a substitution / non-linear map is summarized in a so called *Difference-Distribution-Table*. The values at each position X/Y of the table can be calculated as the size of the IN-Set at this specific position as shown below.

$$DDT(X, Y) = ||IN(X, Y)|| \tag{3.18}$$

Table 3.2 shows the DDT of the substitution S, where Code Example 3.1 shows a simple Java code of calculating this DDT for arbitrary non-linear maps, where the map can be passed as parameter to the function. Further, Code Example 3.2 shows how the DDT was calculated for the given substitution S in TOY.

Code Example 3.1: Function calculating the DDT

```
public static int[][] calculateDDTofSBOX(HashMap<Integer, Integer> sbox)
{
//defining result
int[][] result = new int[sbox.size()][sbox.size()];

//iterating over all inputs
for (int in = 0; in < sbox.size(); in++)
{
  //iterating over all outputs
  for (int out = 0; out < sbox.size(); out++)
  {
    //iterating over all states of the SBOX
    int currentCount = 0;
    for (int x = 0; x < sbox.size(); x++)
    {
      //calculating reference value
      int sboxInput = x^in;
      int sboxOutput = sbox.get(sboxInput).intValue() ^ sbox.get(x).intValue();

      //if result matches: incrementing counter
      if(sboxOutput == out)
      {
        currentCount++;
      }
    }

    //inserting result into code
    result[out][in] = currentCount;
  }
}

return result;
}
```

Code Example 3.2: Function used to define the S-Box and calculate the DDT

```
public static void main(String [] args)
{
  HashMap<Integer, Integer> sBox = new HashMap<Integer, Integer>();
  sBox.put(0, 6);
  sBox.put(1, 4);
  sBox.put(2, 12);
  sBox.put(3, 5);
  sBox.put(4, 0);
  sBox.put(5, 7);
  sBox.put(6, 2);
  sBox.put(7, 14);
  sBox.put(8, 1);
  sBox.put(9, 15);
  sBox.put(10, 3);
  sBox.put(11, 13);
  sBox.put(12, 8);
  sBox.put(13, 10);
  sBox.put(14, 9);
  sBox.put(15, 11);

  int[][] ddt = calculateDDTofSBOX(sBox);

  for (int i = 0; i < ddt.length; i++)
  {
    for (int j = 0; j < ddt.length; j++)
    {
      System.out.print(ddt[j][i] + " & ");
    }
    System.out.println();
  }
}
```

in / out	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	6	-	-	-	-	2	-	2	-	-	2	-	4	-
2	-	6	6	-	-	-	-	-	-	2	2	-	-	-	-	-
3	-	-	-	6	-	2	-	-	2	-	-	-	4	-	2	-
4	-	-	-	2	-	2	4	-	-	2	2	2	-	-	2	-
5	-	2	2	-	4	-	-	4	2	-	-	2	-	-	-	-
6	-	-	2	-	4	-	-	2	2	-	2	2	2	-	-	-
7	-	-	-	-	-	4	4	-	2	2	2	2	-	-	-	-
8	-	-	-	-	-	2	-	2	4	-	-	4	-	2	-	2
9	-	2	-	-	-	2	2	2	-	4	2	-	-	-	-	2
a	-	-	-	-	2	2	-	-	-	4	4	-	2	2	-	-
b	-	-	-	2	2	-	2	2	2	-	-	4	-	-	2	-
c	-	4	-	2	-	2	-	-	2	-	-	-	-	-	6	-
d	-	-	-	-	-	-	2	2	-	-	-	-	6	2	-	4
e	-	2	-	4	2	-	-	-	-	-	2	-	-	-	-	6
f	-	-	-	-	2	-	2	-	-	-	-	-	-	10	-	2

Table 3.2: The calculated Difference Distribution Table of the Substitution S.

Using DDT Characteristics

By facilitating this DDT, we know which input differences can map to which output differences with what probability. This will now be used to break the cipher *TOY*.

In a previous step, we discovered the relations for input difference $\triangle u$ and output difference $\triangle v$ to the S-Box S illustrated in Equation 3.13 and 3.16.

Since we have two different plaintext / ciphertext pairs given, we can calculate the values of this differences as shown below.

$$\triangle u = P_1 \oplus P_2 = \texttt{0x5} \oplus \texttt{0xA} = \texttt{0xF} \tag{3.19}$$

$$\triangle v = C_1 \oplus C_2 = \texttt{0xF} \oplus \texttt{0x0} = \texttt{0xF} \tag{3.20}$$

If we now take a look at the value in the DDT for this input difference $\triangle u$ and output difference $\triangle v$ we can see, that only 2 possible values of u exist. These concrete values can be calculated using the IN-Set of Equation 3.17. Thus, the internal state $u \in \{7, 8\}$. Revisiting the relation $u_1 = P_1 \oplus K_1 \Leftrightarrow K_1 = P_1 \oplus u_1$ we can obtain 2 different values for K_1. Therefore, the number of possible sub-keys for K_1 (4 bit) just reduced from 2^4 to 2.

The cipher is broken, since we are efficiently able to reduce the possible keys for this cipher (independent from which keys are used). Note that the same procedure can be applied for the sub-key K_2.

Real world ciphers typically have more rounds to prevent attacks like this. To perform an attack similar to the one described before, an attacker would need to identify both input and output differential at a specific S-Box in the cipher. Therefore, differential characteristics with a high probability over several rounds of a cipher need to be found. The term differential characteristic is explained in the following section.

In general differential cryptanalysis is a chosen-plaintext attack, so an attacker can encrypt arbitrary plaintext messages to construct internal differences. The goal of the attacker is to construct differences in a way, so that the input and output differences of a certain substitution (the last S-Box involved most of the time) can be determined to conduct an attack. The goal of the cipher designer is to construct the cipher in a way, so that these input / output differences can only be established with low probability. This comes down to the problem of mitigating differential characteristics with high probabilities as discussed and explained later in this chapter.

Further, note that all calculations of IN-Sets and DDT, are invariant to the used key $K = K_1||K_2$, and therefore only has to be performed once in order to break all future keys used for this cipher.

Differential Characteristics

In general, a differential characteristic names the propagation of a difference - as defined in Section 3.2.1 - over certain rounds of transformations. To illustrate this concept, consider a function f transforming an input into an output as illustrated in Figure 3.3 and Equation 3.21 to 3.22. Now assume the input difference $\triangle a$ at the input of the function as well as the output difference of it as $\triangle b$. Since the input difference $\triangle a$ results in the output difference $\triangle b$, we call the pair $\triangle a \xrightarrow{f} \triangle b$ a differential characteristic. Please note that, as discussed in Section 3.2.1, this characteristic might only hold with a certain probability $0 \leq p \leq 1$.

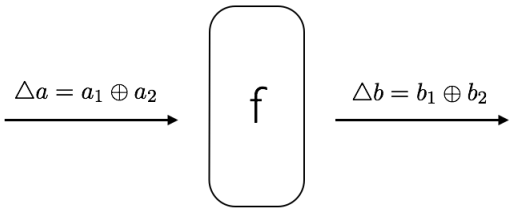

Figure 3.3: Idea of the concept of differential characteristics.

$$f : \{0,1\}^m \longrightarrow \{0,1\}^m \tag{3.21}$$

$$f(a) = b \tag{3.22}$$

In practice, characteristics over more than one round of such a (potentially non-linear) function f are interesting. Such a differential over R rounds of the function f would be donated by $\triangle x_0 \overset{f^R}{\underset{p}{\rightarrow}} \triangle x_R$. This multi-round-characteristic is illustrated in Figure 3.4.

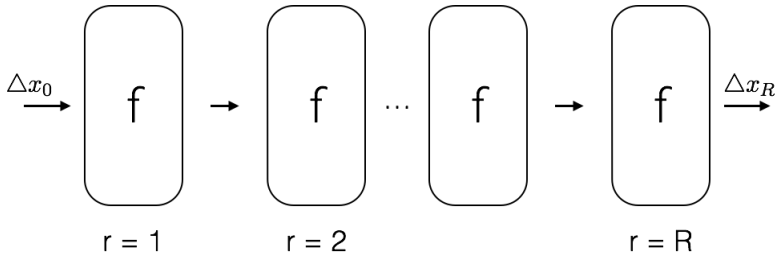

Figure 3.4: A differential characteristic over multiple rounds of a function f.

3.2.2 Impossible Differential Cryptanalysis

Impossible Cryptanalysis states a sub-form of differential cryptanalysis following a slightly different approach. Rather than trying to narrow down the key-space in the demonstrated way by finding differential trails with a high probability throughout ciphers, this type of attack aims towards finding impossible differential characteristics in a cipher. This means the goal is to find characteristics that propagate through the cipher with probability $p = 0$ and derive statements and information about the cipher and it's secret components (keys) from this.

The basic idea is based on finding a differential trail $\triangle X \longrightarrow \triangle Y$ over several rounds of the cipher with probability $p = 0$. This means one has to calculate all possible ways of how $\triangle X$ can propagate through the rounds and find a point where all branches show a probability of $p = 0$ to propagate to $\triangle Y$. A common way to attack a cipher using impossible differential is to find impossible characteristics $\triangle X \longrightarrow \triangle Y$ to a certain internal point in the cipher and propagate values of C_1 and C_2 (ciphertexts) under a certain key assumption K_i to that point starting from the other side resulting in C_{1_k} and C_{2_k}. If at the defined point, the impossible differential proofs that $\triangle C_k = C_{1_k} \oplus C_{2_k}$ is impossible at this point, the attacker knowns that the key assumption K_i was wrong and can cross this candidate off the list. This way, the last sub-key can be attacked separately and the key-space can be narrowed down relatively efficient.

3.2.3 Linear Cryptanalysis

Besides differential cryptanalysis, linear cryptanalysis is the analysis most widely used method to analyse ciphers and cryptographic primitives. The basic goal of this analysis is to find a set of linear equations describing or approximating a cipher and then solving

this system of equations in order to reveal the secret key K. Linear cryptanalysis takes place in two main steps:

1. Find a description / approximation of the cipher through a set of linear equations (with a high probability)
2. Solve this set of equations to recover (parts of) the secret key K using known plaintext / ciphertexts pairs (P_i, C_i)

In contrast to differential cryptanalysis, linear cryptanalysis is a known-plaintext attack rather than a chosen-plaintext attack.

To conduct the attack, a set of equations in the general form, illustrated in Equation 3.23, is required and directly reveals the secret key. It relates bits of the W sized words P, C and K.

$$P[0] \oplus \ldots \oplus P[W-1] \oplus C[0] \oplus \ldots \oplus C[W-1] = K[0] \oplus \ldots \oplus K[W-1] \tag{3.23}$$

As it can be seen in Equation 3.23, the equation relates selected bits of the plaintext, ciphertext and used key. Therefore, the equation can be written as a *selection* of bits of each of these parameters, related to each other. This selection of bits can be represented by the multiplication of a column vector x containing the variables bits and a mask m filtering / selecting bits to be part of the equation (as illustrated in Equation 3.24). Equations 3.26 to 3.29 show an example of this notation.

$$x = \begin{pmatrix} x[0] \\ x[1] \\ \vdots \\ x[W-1] \end{pmatrix} \quad m = \begin{pmatrix} m[0] \\ m[1] \\ \vdots \\ m[W-1] \end{pmatrix}^{\mathsf{T}} \tag{3.24}$$

$$\tag{3.25}$$

$$\texttt{e.g.:} \quad P[0] \oplus P[1] \oplus P[3] \oplus C[2] \oplus C[3] = K[0] \oplus K[3] \tag{3.26}$$

$$\Leftrightarrow \alpha \cdot P \oplus \beta \cdot C = \gamma \cdot K \tag{3.27}$$

$$\tag{3.28}$$

$$\Leftrightarrow \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix}^{\mathsf{T}} \cdot \begin{pmatrix} P[0] \\ P[1] \\ P[2] \\ P[3] \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}^{\mathsf{T}} \cdot \begin{pmatrix} C[0] \\ C[1] \\ C[2] \\ C[3] \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}^{\mathsf{T}} \cdot \begin{pmatrix} K[0] \\ K[1] \\ K[2] \\ K[3] \end{pmatrix} \tag{3.29}$$

Of course, non-linear parts of ciphers (such as S-Boxes) cannot be described in linear equations. Thus, such components have to be linearised or approximated so that they can be described in linear equations. This means that the goal of the cryptanalyst is, to find an equation $\alpha \cdot x = \beta \cdot S[x]$ that matches the behaviour of the real S-Box with a significant probability.

When applying linear cryptanalysis to the previously introduced example, the relations stated in Equations 3.30 to 3.32 can be derived (adapted from [17]).

$$\alpha \cdot P = (\alpha \cdot u) \oplus (\alpha \cdot K_1) \text{ with probability } p_1 \tag{3.30}$$

$$\alpha \cdot u = (\beta \cdot v) \text{ with probability } p_2 \tag{3.31}$$

$$\beta \cdot v = (\beta \cdot C) \oplus (\beta \cdot K_2) \text{ with probability } p_3 \tag{3.32}$$

Equations 3.30 and 3.32 result of the XOR operations $u = P \oplus K_1$ and $C = v \oplus K_2$. The probability of these equations to match is actually 1, since they describe a linear operation. Therefore, $p_1 = p_3 = 1$. Equation 3.31 represents an approximation of the used S-Box S. So this is a linear expression describing a non-linear operation and will not be valid for all possible values, and thus $0 < p_2 < 1$. The usage of variables and masks is illustrated in Figure 3.5.

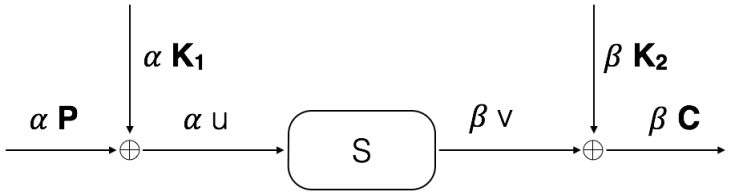

Figure 3.5: Cipher with masks to select bits involved in the linear equations.

The goal of the cryptanalyst is, to create an approximation for the non-linear operation that has a significant probability. The worst case for an attacker (and the best for a designer) is a probability of 1/2. In this case, the approximation matches half of the time, which results in the least degree of information for an attack. Thus, the goal of the approximation of any non-linear operation is to find masks α and β, that produce a probability of the approximation $p = \frac{1}{2} \pm \epsilon$ where $|\epsilon|$ is maximised. We call ϵ the *bias*. The probability of the approximation can be easily determined by simply counting the matches between $\alpha \cdot u$ and $\beta \cdot v = \beta \cdot S[u]$ over all possible input values u.

Note that significantly small probabilities are also good from a cryptanalytic point of view, since if $\alpha \cdot u = \beta \cdot v$ has a low probability p, the relation $\alpha \cdot u \oplus 1 = \beta \cdot c$ has the probability $p* = 1 - p$, which is therefore high. So both forms of values showing a high bias ϵ resulting in an advantageous situation for an attacker.

If we combine Equation 3.30 to 3.32 and order the equations (all key-dependent parts to the right hand side) and cancelling terms we can obtain.

$$(\alpha \cdot P) \oplus (\beta \cdot C) = (\alpha \cdot K_1) \oplus (\beta \cdot K_2) \tag{3.33}$$

This fully describes the approximated cipher with a probability p_2 resulting from the approximation of the non-linear operation. Therefore, the first phase of the attack has been completed.

As a next step, we use this description to conduct an actual attack and reveal the secret key. Please note that the right hand side of Equation 3.33 is unknown due to the unknown sub-keys K_1 and K_2. Nevertheless, the left hand side expression is known since it's a known-plaintext attack.

The goal now is to determine the value of the right hand side of the equation by calculating the left hand side for a huge number N of plaintext/ciphertext pairs. Since the left hand side can be calculated (for N pairs) and we know the probability of the equation to hold, we can establish the value of the right hand side ($\in \{0, 1\}$) with a probability of p_2. The better the approximation is (larger bias ϵ), the more likely the equation will hold and we can calculate the value of the right hand side using a large enough set of plaintext/ciphertext pairs. To proceed, we simply evaluate the left hand side for all N known pairs and count the occurrences of 1 and 0 as result. Either one of these values occurring more often (assuming N is large enough) is taken as assumption for the right had side value with probability p_2.

Therefore, one bit of the key can be determined. This procedure can be repeated in a similar way to attack the remaining bits of the key.

3.3 Analysis Methods for AE

This section discusses some basic type of analysis used to perform cryptanalysis on authenticated ciphers. This will focus on ciphers such as discussed in Chapter 2 and concepts based on a sponge construction as defined in the scope of this thesis.

3.3.1 Goals

As mentioned in earlier chapters, there are mainly two types of attacks on authenticated ciphers:

a) **Key Recovery Attacks**: Goal of this attack is to break confidentiality of ciphers and reveal the secret key K.

b) **Forgery Attacks**: Goal of this attack is to break authenticity of a cipher by being able to force a message - i.e. produce two ciphertext messages C and C', resulting in the same authentication tag A. So authenticity is not fulfilled and the cipher is broken.

Since all considered ciphers are based on a design, that initializes the state based on the key and a nonce, recovering the key is dependent on finding good differential character-

istics over the - typically high amount of - initialization rounds of the round transformations. Based on the assumptions of all designs providing good protection against finding such characteristics the analysis methods described here mainly focus on forgery attacks.

In a forgery attack on the ciphers, an attacker is able to manipulate a ciphertext C, transmitted over an insecure channel. So the attacker has no influence on the key K but only on the ciphertext C and the used nonce N. The goal of authenticity fulfilled by the cipher is to detect any changes that the attacker would apply to C by detecting a wrong authentication tag A. The goal of the attacker is to change C into $C' \neq C$, that produces the same A when decrypted. Therefore, the receiver of the message would not detect the tampering of the data. Figure 3.6 illustrates this attack setting.

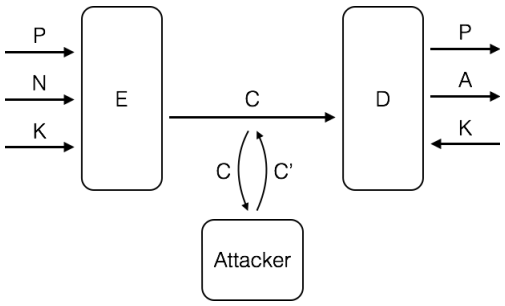

Figure 3.6: Attack setting in a forgery attack.

Conducting a forgery attack is mainly done by searching for internal differential characteristics, that cause the internal state to collide at certain positions within the cipher. The following describes certain techniques and tools of how to find such characteristics.

The goal of a forgery attack is to construct or find ciphertexts C and C' resulting in the same authentication tag A. Considering the sponge construction, this comes down to finding collisions in the internal state of the cipher. This means finding a differential characteristic over R rounds that produces differences of zero in the state at the end. So, a non-zero difference $\triangle S_0 = S_0 \oplus S'_0$ at the beginning resulting in a zero difference $\triangle S_R = S_R \oplus S'_R$ after R rounds. This is illustrated in Figure 3.7.

This case represents a collision of the state solely based on properties of the round function f. The differences $\triangle S_0$ could be injected into the state S_0 by the given ciphertexts C and C' used for the attack. The above described scenario requires the whole state to collide after certain rounds, which is very unlikely to happen in practice, since the ciphers are designed to prevent this. But given the fact that after some round, the state can be manipulated (at parts) using the injected ciphertext, only certain parts of the state need to collide.

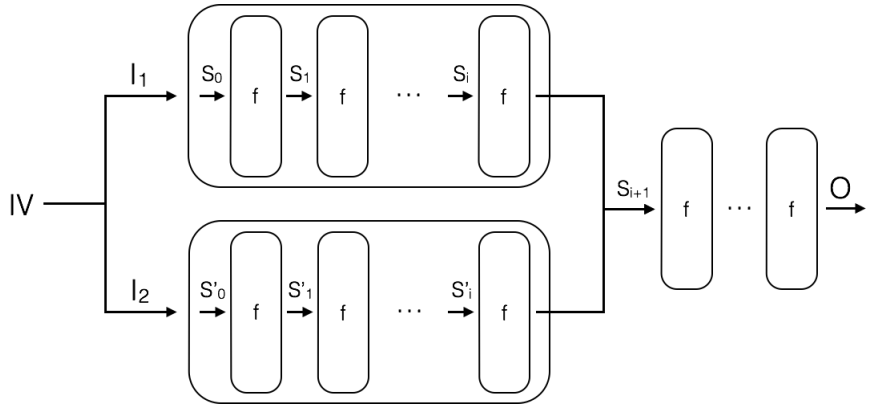

Figure 3.7: Development of a state collision.

3.3.2 Concepts and Tools

This section gives an overview of different methods and concepts of analysing authenticated encryption schemes as well as tools to perform such an analysis.

Coding Theory

One way of finding such collisions in internal states is to leverage known algorithms for finding low-weight codewords in linear codes. To do this, the problem of finding colliding states, or in general, differences matching certain conditions has to be mapped to finding codewords of a linear code. This is based on the assumption that finding codewords showing certain properties in a linear code is generally easy. Further, is uses a linearised model of the cipher in order to be able to apply these techniques.

A linear code C is defined by its generator matrix G. The rows of this generator matrix form a basis and therefore span a vector space of that specific code. So the linear code is formed as the *row space* of the generator matrix G. All linear combination of this base vectors form elements in that vector space and therefore are valid codewords.

This matrix can be used to create codewords c_i from arbitrary input vectors v as shown below.

$$c_i = v \cdot G \Leftrightarrow \begin{pmatrix} c_{i_0} & \dots & c_{i_{n-1}} \end{pmatrix} = \begin{pmatrix} v_{i_0} & \dots & v_{i_{n-1}} \end{pmatrix} \cdot \begin{pmatrix} G_{0,0} & \cdots & G_{0,n-1} \\ \vdots & \ddots & \vdots \\ G_{n-1,0} & \cdots & G_{n-1,n-1} \end{pmatrix} \quad (3.34)$$

The generator matrix can be described in a so called *standard* form as shown in Equation 3.35. Here the matrix consists of a part that has the form of an identity matrix I_k and an arbitrary part P. Note that every generator matrix can be brought into this

standard format.

$$G = \big(I_k | P\big) \tag{3.35}$$

Algorithms such as [12], perform an efficient search for low-weight codewords in such linear codes. The problem of finding differential characteristics showing defined properties can be shifted towards finding codewords with certain properties. It has to be noted, that the problem of finding minimum weight codewords is NP-hard. Although probabilistic algorithms, such as [12], find codewords with low Hamming weight relatively efficient in practise.

A generator matrix G (or a vector space) has to be defined representing differential behaviour of the cipher.

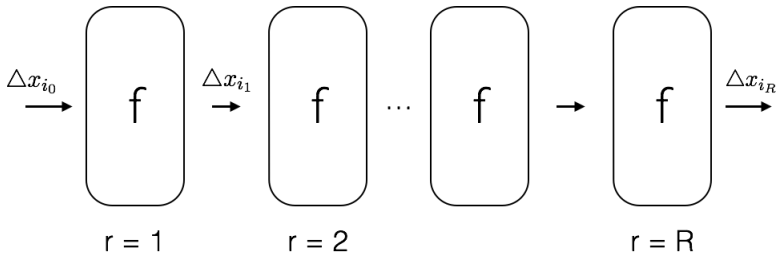

Figure 3.8: The construction of a row vector of the generator matrix.

This process can be summarized in four steps:

1. **Linearisation**: Since a cipher is non-linear by design, it has to be linearised to be represented by a linear code. Using the example of *NORX*, the only non-linear operation in the round-function is the approximation of the word-wise addition as $x = (a \oplus b) \oplus ((a \wedge b) \ll 1)$. This operation can be linearised in various ways, showing varying differential behaviours. In the analysis performed in the scope of this thesis, this operation is linearised as $x = a \oplus b$.

2. **Creation of Linear Code**: In this step the generator matrix G of a linear code is constructed. Here, a row of the generator matrix is equivalent to the propagation of a difference (of size n) through round transformation f. These differences are concatenated to form a row vector $g_i = g_{i_0} || \ldots || g_{i_R} = \triangle x_{i_0} || \ldots || \triangle x_{i_R}$ of G with size $|g_i| = n \cdot (R+1)$., where R donates the considered rounds of the analysis. This process is illustrated in Figure 3.8. To form a basis for a vector space and therefore form a linear code, the matrix needs to have n linearly independent row-vectors (rows) forming this basis. The values for the input differences $\triangle x_{i_0}$ can be chosen as forming an identity matrix I_n as the first part of G. This guarantees linear independence of the input differences and force G to be in standard form. So for each row g_{i_0} of G, two n-sized codewords X_1 and X_2 with $\triangle x_{i_0} = X_1 \oplus X_2 = 1 || 0 \ldots 0 \gg i \mid 0 \leq i \leq n-1$ are created. The propagation of $\triangle x_{i_0}$ through the rounds of f forms the remaining elements of g_i (characteristics).

3. **Enforce Collision**: In the third step, the existences of collisions is enforced. Therefore, certain parts of the found codewords are required to be 0. Since the codewords equal differential characteristics in the cipher state, parts of the state show difference 0, which is a partial collision. Therefore, a number of bits n of the state can be forced to 0. If too many bits are forced to 0, no code word might be found. In order to apply a forgery attack on NORX, the whole capacity c of the state needs to be forced to zero ($c \leq n \leq |S|$). This is needed, since no ciphertext word is ever injected into the capacity (see Section 2.5.2). Therefore, a collision in these parts of the state cannot be enforced by injecting appropriate messages / ciphertexts. At least the capacity has to collide to find an internal collision. Of course, the more state bits collide in the characteristic the better, but forcing to many state bits to zero in this type of search might result in invalid characteristics or none at all.
4. **Search for Codewords**: The final step of the process is to apply the mentioned search algorithms and find codewords of the defined code. The codewords of this cipher are equivalent to characteristics over multiple rounds R of the cipher. Of all possible codewords in the space defined by G, these algorithm finds those with low Hamming weight $hw(c)$. In this context, low Hamming weight is treated as rough criteria for a high probability of the characteristic to hold in practice, since this indicates a low probability of the linearisation distorting the characteristic. If the codewords are sparse, ones in the characteristic will less likely propagate through a part of the cipher that was changed due to linearisation which results in a lower probability of the linearisation generating wrong characteristics.

Note that this method relies on the linearisation of a cipher. This means that the search space for potentially found codewords is narrowed down and characteristics found using that model might not be valid in reality. Although it allows the application of advanced algorithms and general purpose tools for coding theory (non-specific to cryptography).

The tasks of the described process can be executed using various different tools providing support for the above mentioned algorithms. In this thesis - as it will be explained in later chapters - the *IAIK Coding Tool* [22] was used to perform this tasks. It offers great support for search algorithms, forcing states bits to zero and generator matrix representation and generation.

SAT Solvers

Another very popular approach to find such characteristics is using SAT-Solvers. In general, SAT solvers are tools that aim to solve the *Boolean Satisfiability Problem*. The problem is determining if a Boolean equation $f(x_1, x_2, \ldots, x_n) \mid x_1, \ldots x_n \in \{0, 1\}$ is satisfiable and can therefore evaluate to 1 as well as finding the set of variables that satisfies it. This problem has many applications and is known to be a *NP-complete*

problem. As an example, one could consider the Boolean functions:

$$f_1(x_1, x_2, x_3) = (x_1 \vee x_3) \wedge (x_1 \vee x_2) \tag{3.36}$$
$$f_2(x_1, x_2, x_3) = (x_1) \wedge (\overline{x_1} \wedge x_2) \tag{3.37}$$

While it can be easily seen, that f_1 is satisfiable by using the value set $(x_1, x_2, x_3) = (1, 1, 1)$, f_2 cannot be satisfied, since it would require x_1 to be 1 in the first term and 0 in the second. So no set of values (x_1, x_2, x_3) can be found so that f_2 evaluates to 1. SAT solvers implement heuristic algorithms for find solutions to this hard problem.

In cryptography, ciphers can be expressed as a system of Boolean equations, where the plaintext-, ciphertext- and key-bits are represented as Boolean variables. A SAT solver can now verify or disprove the satisfiability of this equation system. Besides the cipher description itself, various other constraints can be expressed in Boolean expressions as well and added to the system. These constraints will be considered by the SAT solver. The input to the SAT solver has to be a set of Boolean equations (mostly in CNF).

Besides plain SAT solvers certain constraint solvers exist to make these processes easier. A constraint solver (such as *STP* [3]), do not require the user to enter a Boolean description of the equations to solve, but equations can be expressed using higher level operations such as additions, subtractions, XORs or bit-shifts. The constraint solver then transforms this input into Boolean equations and applies a SAT solver to it.

In the context of differential cryptanalysis, SAT solvers can be used in the following ways:

- **Search for Characteristics**: Given a differential description of a cipher in Boolean equations (differential behaviour of subcomponents is modelled), a SAT solver can search for variable-values satisfying the equations and can therefore also find differential characteristics for a cipher. In this case, constraints to the differential characteristics are simply modelled as additional equations in the system. In [16] this method was applied to find differential characteristics for NORX using SAT solvers. Note that this provides a much larger search space compared to the coding theory approach described earlier, since not only solutions matching the linearised model are considered.
- **Proof Security Bounds**: In this application, a SAT solver is used to prove certain security bounds of a cipher. If a cryptanalysist has found certain relations to express the security bound of cipher, they can be formulated as Boolean functions and passed to the SAT-solver in order to proof / validate it.
- **Check Characteristics**: As explained in earlier chapters, differential characteristics only hold with a certain probability. Further, characteristics might have been discovered using linearised models of the cipher and therefore have to be checked against its non-linear description. SAT solvers can do just that by checking if a given or found characteristic can satisfy the description as Boolean functions of a cipher. To check such a characteristic $\triangle S = \triangle S_0 || \triangle S_1 || \ldots || \triangle S_R$, one defines two

separate but identical descriptions of the R round cipher as illustrated in Figure 3.9 with the inputs I_1 and I_2 and the outputs O_1 and O_2. These inputs and outputs are treated as separate variables in the Boolean equation system passed to the SAT-Solver. Note that these two descriptions are already given in the form of Boolean equations. Next, a list of constraints enforcing the relations between the states of the two described ciphers as the differentials have to be defined. Each of these constraints defines that the differences of the states of the two described ciphers must be equal to the corresponding part of the differential characteristic. So the constraint c_o would define that $S'_0 \oplus S''_0$ has to be equal to $\triangle S_0$. To describe the presence of the whole characteristic, R different constraints are defined:

$$c_0 : \ S_0 \oplus S'_0 = \triangle S_0 \tag{3.38}$$
$$c_1 : \ S_1 \oplus S'_1 = \triangle S_1 \tag{3.39}$$
$$\vdots \tag{3.40}$$
$$c_R : \ S_R \oplus S'_R = \triangle S_R \tag{3.41}$$

After all equations for both descriptions and the constraints have been added, the SAT solver can be applied to the resulting system of equations. If the equations system is satisfiable, the characteristic is correct.

Dedicated Tools

Besides the mentioned general purpose tools (SAT, CodingTool etc.), there also exist a variety of dedicated tools to perform cryptanalysis. Most of these tools support various different features including calculating probabilities of differentials in ciphers, searching for characteristics as well as many other analysis features. Three of those tools are:

- IAIK NLTool [20]
- ARXTools [19]
- YAARX [11]

3.4 Classic Analysis Workflow

This section gives a short overview of the *classical* (with respect to the automated concept introduced in this thesis) workflow of how to use tools previously described to analyse new designs. As mentioned in earlier chapters the main goal of analysis dealt with in this thesis is to forge the authentication tag A of an authenticated cipher. Further, as described in the methodology section of this chapter, using differential cryptanalysis to create such forgeries is based on finding differential characteristics over various rounds of these ciphers causing collisions of the internal state.

A cryptanalysis would therefore conduct such an analysis in the following steps:

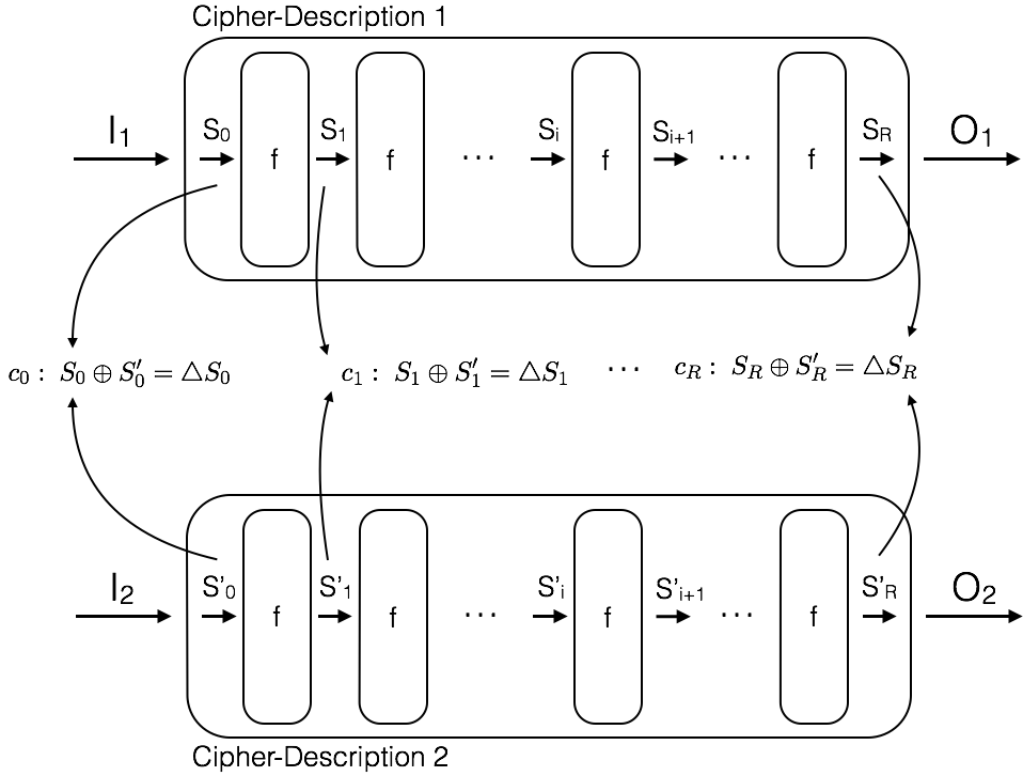

Figure 3.9: Verification process of a characteristic in a cipher using SAT solvers.

1. **Preliminary Analysis**: Apply different tools and techniques to find colliding (partially colliding) differences over the targeted number of rounds. An analyst would try various different ways and different tools with varying parameters to get a preliminary analysis and a feeling or intention for where potential weaknesses of a cipher might be. As an example, a coding theory based tool as well as a SAT-solver search could be started with varying parameters such as state bits forced to collide or number of rounds. All found characteristics, even though they might prove invalid later on, contribute to provide an understanding of towards where to conduct further - manual and dedicated - analysis.

2. **Validation**: The characteristics found in the previous step have to be validated using other tools. This might be based on assumptions or simplifications (linearisation) that took place in order to make the search for characteristics easier or feasible in the first place. For example, a cryptanalyst might use a dedicated tool or a SAT-solver to describe the cipher and prove the validity of found characteristics. The result of this tells whether the found trail is valid or invalid.

3. **Dedicated Analysis**: Based on the found characteristics and the gained understanding of the characteristics of the cipher under investigation, the cryptanalysis

now tries to target weaknesses of the cipher in order to create good characteristics and force collisions in an efficient way. This turns out to be very complex and requires dedicated and manual effort as well as a high cryptanalytic expertise to find vulnerabilities. This might be done using various tools with adapted cipher descriptions and own, targeted analysis code. For example, the description of the cipher as Boolean equations - for use in a SAT solver - can be adapted or special conditions can be added etc.

4. **Attack**: In this step, based on the results from the previous step, an actual attack is performed on the cipher, limitations and potential of the attack as well as it's impact is evaluated.

3.4.1 Toolchains

As seen before, different tools are used in the workflow of performing an attack on a proposed cipher design. Hereby, we can generally distinguish two kinds of tools:

a) **Analysis Tools**: In this operational mode, tools search for characteristics in a certain form. Based on different techniques, the goal is to find a candidate characteristic showing specific properties defined on beforehand. It takes as input a description of the cipher in the required format as well as a search configuration defining basic parameters. It outputs a candidate characteristic found according to the requirements.

b) **Validation / Verification Tools**: In this mode, tools try to validate found characteristics. Therefore, they take the characteristic as well as a cipher description in an arbitrary format as an input, and output a verification result $\in \{valid, invalid\}$.

Note that most tools can potentially be operated in both modes and therefore strictly separating between analysis and verification tools is not possible. For example, a SAT-solver can be used both to search for characteristics and verify them using different cipher descriptions and additional conditions.

3.4.2 Problems

As described in this chapter, the cryptanalysis process of ciphers is mainly done following an analysis process that still requires a lot of effort from the cryptanalyst to establish a preliminary analysis using various different tools. Nevertheless this part of the process ensures a broad understanding and indication for where potential weaknesses of a cipher might be. The main work in performing this preliminary analysis goes into establishing different forms of cipher representations used by different tools. Especially in an environment like cryptographic competitions, where a large number of ciphers needs to be analysed, this effort increases dramatically. This motivates the need for automation of this process in any form.

Chapter 4

Automated AE Analysis

This chapter describes what an automated preliminary analysis framework can look like, which problems it addresses, how it is used and what challenges the implementation of such a framework implies. Further, it explains what design principle it follows and illustrates its potential extensions and applications. In Chapter 5, the implemented framework and it's components and processes are explained in detail.

4.1 Problem Description

As discussed in Chapter 3, tools - such as analysis and verification tools - are commonly chained together in order to reach the analysis goal. Further, it can be observed, that all of those available tools operate on very different representation of the cipher analysed. For example, a coding theory based tool takes the generator matrix of the cipher (or at least it's round function) as a description, while a SAT-solver requires the cipher specified as a system of Boolean equations. Other tools have specific formats of how algorithms and functions are defined. Representations of results and parameters widely vary among tools. Figure 4.1 illustrates an example workflow showing how many different formats and representations are needed in order to use one tool to find characteristics (`NLTool` in this case) and validate the result (using a SAT-solver here). Note that this represents one of many different analysis paths running in parallel to conduct preliminary analysis and gain an overview of the cipher's characteristic. It might be necessary to have multiple such trails running in parallel to gain some insights in the security of a cipher.

Cryptanalysts face the huge problem of having to put a large amount of work into implementing different cipher representations for every tool in every mode of operation, defining parameters and input/output formats and writing analysis code in many different formats and languages. Figure 4.1 shows an example where a cryptanalysis need to define:

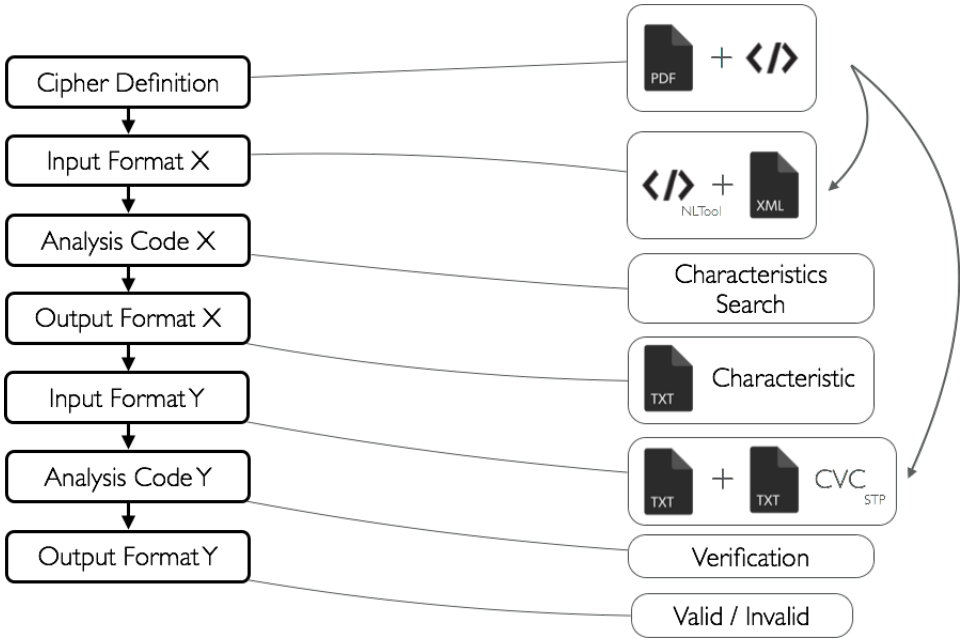

Figure 4.1: Example of the tools used in a standard analysis workflow.

- Two cipher representations: NLTool (C-code), CVC (input language SAT solver) representing Boolean equations
- An adapter for the input / output formats of the different tools (or manual translation)
- Configuration files for search (NLTool) and verification (SAT)
- Analysis Code executing search and verification: terminal commands or Script for NLTool, C-code or terminal commands (scripts) for SAT-solver

This common scenario might have to be extended to n different such tool chains resulting in significant overhead to perform preliminary analysis. This scenario is illustrated in Figure 4.2.

Cryptographic Competitions

Since in cryptographic competitions a large number of ciphers require external analysis through the analysis process, the increasing number of submissions as well as the increased quality of submissions, as illustrated in Chapter 1, magnify the effort going into the preliminary analysis-step. Therefore, especially in the context of such a competition, this raises the need for an automated preliminary analysis framework that handles step one of the before mentioned workflow and lets cryptanalysis focus on designing and performing dedicated analysis techniques and attacks. Such a framework, that can facilitate

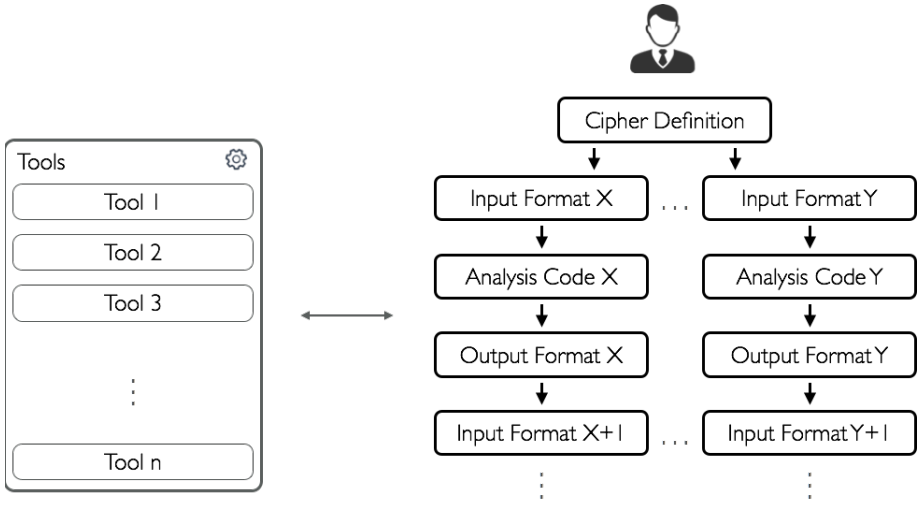

Figure 4.2: Applying n tool chains to establish a preliminary cipher analysis.

various different tools, makes it very easy to integrate tools and works on arbitrary ciphers without adaptation due to automated cipher specification parsing as introduced in this thesis.

4.2 The Idea

The idea of the automated analysis framework is based on being able to automatically perform the stated preliminary analysis of a variety of different submissions to a cryptographic competition. The framework analyses general properties of the cipher and tries to find indicators for potential weaknesses as well as candidates for characteristics, while operating on a low cryptanalytic level. As mentioned before, this can only be done by applying and combining various different tools and toolchains. The framework must allow easy integration of different tools to be used for both analysis and verification following the definition in Chapter 3. This basic idea of taking various cipher definitions as input and applying different tools to perform preliminary analysis is illustrated in Figure 4.3.

At it's core, the idea is based on being able to automatically create an abstract representation Γ of ciphers and algorithms submitted to cryptographic competitions as well as being able to transform Γ to the different representation formats used by different tools. This is needed as a basis for performing automated analysis, since different representation formats are required to run different tools without adapting their implementation. A possible solution to this problem would be defining an abstract cipher definition language and implementing transformations to the different input formats used during analysis. However, this thesis explores a solution of achieving this goal without any additional description or definition of the cipher.

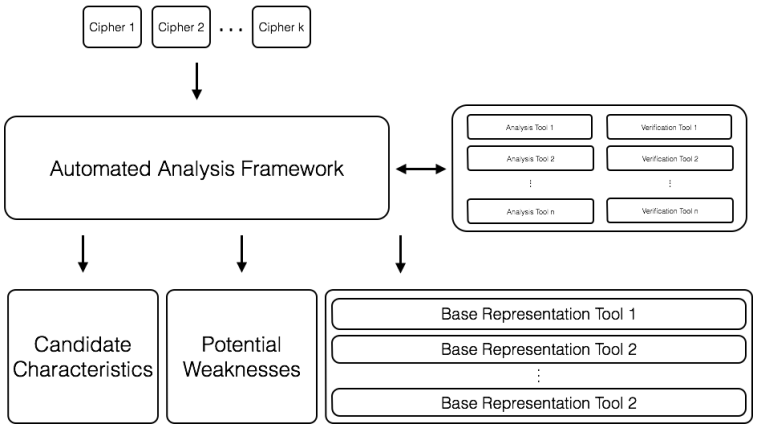

Figure 4.3: Schematic illustration of the preliminary analysis framework.

To implement an automated framework performing preliminary analysis, an abstract representation of the analysed cipher needs to be created. This forms the basis for translation to the representations used by other tools throughout the process. In order to get to such a representation when only using elements given in the submission, it is important to identify a submission component that can be used as a basis for its construction. The submission consists of two main components:

a) **Specification**: The design specification is represented as a paper describing basic design principles, the cipher design itself as well as security claims and preliminary cryptanalysis on the scheme. It is usually submitted in the form of a pdf document in a non-standardized form. Therefore, this component, even though containing the cipher definition, is not suitable for the task of creating an abstract cipher representation.

b) **Reference Implementation**: Submissions to competitions usually also contain a reference implementation of the proposed cipher in a defined format and language. This mainly serves two goals:

   - A reference implementation allows the cryptographic community to test the cipher and gain practical understanding of how it operates. This overcomes potential ambiguities in the textual definition of the cipher and helps to immediately understand the cipher design.
   - Further, in many cryptographic competitions (especially for symmetric cryptography) properties like performance or memory consumption of the proposed primitives play a major role in the evaluation and ranking of the submissions. Therefore, the evaluators have to be provided with an implementation that allows the application of performance tests for benchmarking on all candidates. Commonly, researches can submit optimised implementations of their ciphers - for different platforms - sometime after the submission deadline for that purpose.

   This implementation is commonly delivered in the language C, while defined interfaces

need to be implemented so that performance tests can be run automatically. In recent cryptographic competitions, the *eBACS* system has been used as a benchmarking tool for submissions using distributed environment in order to reach coverage of many architectures and systems [13]. This submission component forms a good basis for creating an abstract cipher representation.

Therefore, the framework takes the definition of a cipher given by it's reference implementation in C-code as an input, transforms it to an abstract cipher representation and implements adapters to easily transform it into representations needed by different tools used during analysis. This basic process is illustrated in Figure 4.4.

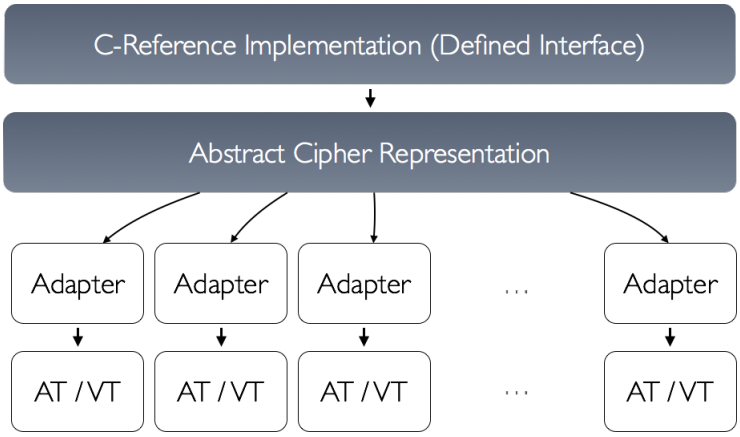

Figure 4.4: Basic transformation of cipher representations within the framework using adapters to transform the abstract representation into formats needed by different tools.

4.3 The Automated Analysis Workflow

The framework introduced in this thesis mainly addresses the first step in the *classical* analysis process as introduced in Chapter 3. So preliminary analysis of the different ciphers as well as the creation of the basic cipher representation used for further dedicated analysis is performed by the framework. It can be applied to various different cipher submissions at the same time. Therefore, cryptanalysts can focus on conducting dedicated analysis on ciphers showing exploitable properties such as characteristics with high probability. The *scanning* for such properties as well as applying standard tools using various different cipher representations is automatically handled by the framework .The following illustrates the usage of it from a cryptanalyst's perspective.

4.3.1 Using the Framework

In order to start an automated analysis process, the user of the framework would provide the following items as input as illustrated in Figure 4.3:

- **Reference Implementation**: This names the reference implementation in C, contained in the submission. The user might have to make short annotations in the code to enable automated generation of the abstract representation Γ. This is discussed in detail in Chapter 5.
- **AT/VT**: A set of n analysis and verification tools to be used by the framework. As mentioned earlier, some tools can perform both operations (analyse / verify) in different modes.
- **Adapters**: This names a set of n adapters, transforming the abstract cipher representation Γ into the representations used by the specific tool. So for every tool used in combination with the framework, an adapter has to be implemented once.
- **Analysis Code**: The code defining the steps of the analysis. Note that it uses the framework to perform the analysis and defines the toolchain. This code can be reused for the same analysis procedure independent from which cipher is analysed.

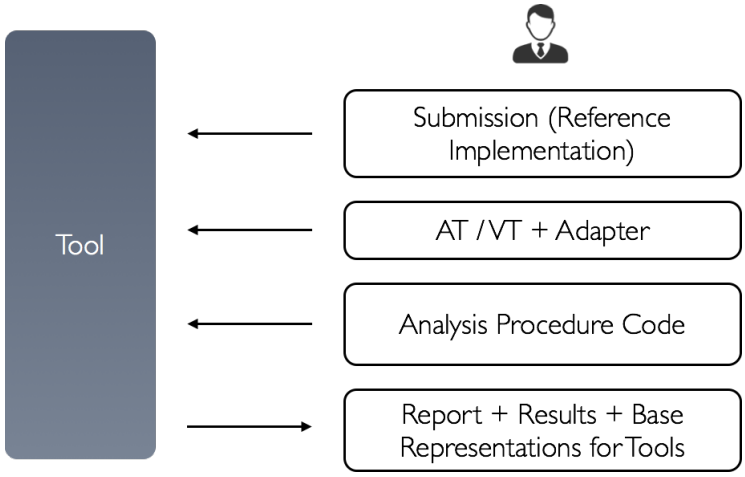

Figure 4.5: Overview over input and output items to the framework.

The output of the system are the following as illustrated in Figure 4.5:

- **Report**: A report describing the search and verification process. This illustrates which characteristics were found, what properties they have and if they could be verified. Further, it gives information about the covered part of the search space and used parameters.
- **Results**: The results mainly consist of candidate characteristics, found during the

analysis. They can be used to point out directions of further dedicated analysis and attacks.

- **Base Representations for different tools**: As mentioned earlier, a base representation of the cipher for each of the used tools is generated. Besides being used for the preliminary analysis, these form the basis for further dedicated analysis and provide the cryptanalyst with a basic description that can be adapted to specific needs. For example, a cryptanalyst might take this base representation and conduct an attack based on properties delivered by the preliminary analysis.

4.3.2 Application Setting

A very likely application scenario of the framework is not running the analysis procedure on local machines, but on a cluster. After the cryptanalyst submits x ciphers to the software running on the cluster, it would run different search and verification steps according to the analysis code and return the above mentioned output. The idea here is that the software would continuously run on the server, perform it's procedure with a setting that does not terminate and report results back to the cryptanalyst on a regular basis. Since algorithms used for searching characteristics mostly rely on heuristic and randomized approaches, this would ensure a coverage of a great portion of the search space with continuous feedback to the cryptanalyst. This setting is illustrated in Figure 4.6.

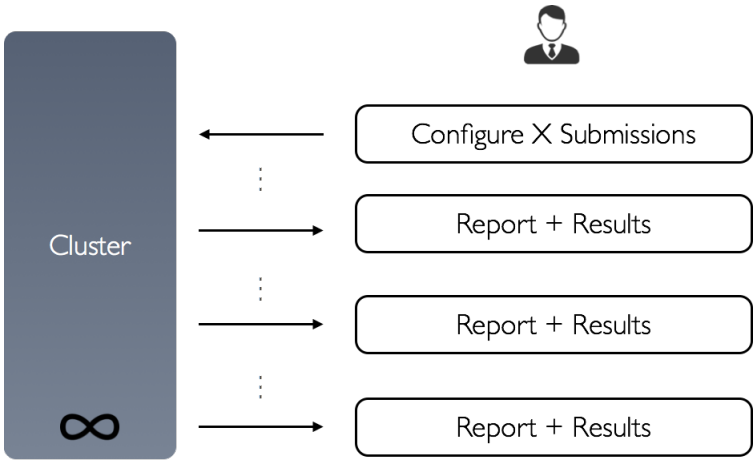

Figure 4.6: Principle of continuous preliminary analysis of ciphers using a cluster server.

4.4 Goals

When the automated framework was first designed, the following basic goals were identified to ensure high benefit to the cryptanalyst using the framework as well as a high level of flexibility for future task and use cases:

a) **Abstract Cipher Representation**: Finding an abstract representation format that can be used to represent different kinds of analysed ciphers as well as to perform operations on it and allow the implementation of adapter is essential to the design of the framework. It should be easily possible for a cryptanalyst, to implement adapters for newly integrated tools without deep knowledge of the abstract representation and independent of the cipher analysed.

b) **Automated Parsing**: The tool should be able to automatically parse reference implementation code given in submissions to competitions into the abstract representation Γ. Although, it should not be dependent on a specific structure of the code itself. The cryptanalyst should be able to annotate the code and therefore configure the automated parsing and interpretation.

c) **Extendibility**: The framework should be very easy to extend in terms of adding new tools and implementing the adapters necessary to include them into an analysis. Therefore, well defined interfaces and a modular structure of the framework are required. It should be easy to integrate different kinds of tools, even beyond the described types of analysis and verification tools.

d) **Combination of Tools**: Cryptanalysts shall be able to easily combine different kinds of tools in analysis code. Therefore, besides the representation being transformed to different formats, parameters to different tools, input and results must be easily combinable.

e) **Demonstration**: A basic version of the framework shall be implemented in this thesis to prove the viability of such a framework and demonstrates it's benefits and potential.

We refer to Chapter 5 more more detail on how these goals are met in the implementation of the automated framework.

4.5 Supported Tools

In the following section, the tools supported in the framework developed in the thesis are presented and shortly explained. The tools themselves implement analysis principles described in Chapter 3. We will describe the basic formats for input / output and cipher representations they use and how the analysis is conducted.

4.5.1 Extended IAIK CodingTool

As mentioned in earlier chapters, the IAIK Coding Tool [22] was used to perform a preliminary analysis part using the coding theory approach described in Chapter 3. The tool itself supports the creation of the required generator matrix G as well as for searching for low-weight codewords and the data structures needed to perform this task and hold

data. In this thesis, the framework was extended in order to support automated creation of this generator matrix based on an arbitrary C++11 closure representing the round function. Therefore, a closure or function pointer to the round function can be passed to the extended tool which then performs an automated preliminary analysis following the previously described approach. The workflow of this procedure is as follows:

1. The round function of a reference implementation of the cipher analysed is passed to the function as a closure or function pointer. The tool input forms a vector of such closures or function pointers. This way, several different sub-rounds can be explicitly added. The state differences between each of the passed sub-rounds will be included in G and therefore considered in the search. So if, for example, sub-rounds of the original round transformations $S_{i+1} = f(S_i)$ are interesting, the round can simply be split in two parts as closures for $S_{i+1} = f_2 \circ f_1(S_i)$. The tool will consider differences for the intermediate state value in the search and results.

2. The tool constructs a generator matrix G in standard form. This is done by constructing the i-th row g_i by forming differentials of the form $(1||0 \dots 0) \gg i \mid 0 \leq i \leq n-1$ at the beginning of the cipher and propagating the differential through the specified number of rounds R, when each intermediate value is appended to the row.

3. In the code-shortening step, the tool forces a defined number of state bits to be zero by transforming G to a corresponding form. This is done using simple Gaussian elimination.

4. As a next step, the tool searches for low-weight codewords in the linear code defined by G using probabilistic algorithms from coding theory applications, as discussed in Chapter 3.

5. Once such a codeword is found, it calls a method of a defined callback passing the found characteristic as well as it's Hamming weight. The callback can then try to validate / verify the characteristic using another tool or simply check if the Hamming weight of the found word is interesting or relevant for the analysis. If the callback returns `true`, the word is considered verified. The tool then calls one of two other callback methods dependent on if the word could be verified. These methods tell the tool whether to store the found word or not.

6. In it's final step, the tool returns a collection class holding all found codewords / characteristics as well a value indicating if they could be verified. This collection class further allows the generation of a LaTeX- report containing the results.

Code Example 4.1 shows an analysis code performing the coding analysis on $0 < R \leq 4$ rounds with various different parts of the state forced to collide. First, it generates the closures for two sub-rounds (`columnG` and `diagonalG`) based on the functions `F1` and `F2` from the NORX implementation as described in Chapter 2. Then it iterates over the target rounds as well as the number of state bits forced to zero and calls the method

`runCodingTheoryAnalysis` to perform the analysis. The object `resultSeries` holds the results and is used to generate the LaTeXreport.

Code Example 4.1: Application example of the adapted / extended CodingTool to perform coding analysis on NORX.

```
#include <iostream>
#include "LowWeightSearch.h"
#include "CodeMatrix.h"
#include "Logger.h"
#include "CipherAnalyzer.h"
#include "StateAdapter.h"
#include <tuple>
#include <stddef.h>

//norx includes
#include "src/ciphers_linearized/norx/norx6441/ref/norx.c"

int main(int argc, const char* argv[])
{
    //NORX-------------------------------------------------------
    //1-4 rounds of round function, 64 bit state words, 16 word state size
    std::cout << "Starting analysis of NORX" << std::endl;

    //creating vector of sub round functions
    std::vector<function<void(uint64_t*)>> norxRoundSubFunctions;

    //adding parts of the round function as closures
    auto columnG = [](uint64_t* state){F1(state);};
    auto diagonalG = [](uint64_t* state){F2(state);};
    norxRoundSubFunctions.push_back(columnG);
    norxRoundSubFunctions.push_back(diagonalG);

    CodingTheoryAnalysisResultSeries resultSeries("norx_analysis");

    //performing coding theroy analysis for 1 to 4 rounds
    for (int roundoption = 1; roundoption <= 4; ++roundoption)
    {
        //creating 64 bit norx cipheranalyzer
        CipherAnalyzer<uint64_t> norx("norx", 16, roundoption, norxRoundSubFunctions);

        //performing coding theory analysis and forcing 7-16 state words (6 words = c->always forced to 0)
        for (int tozero = 7; tozero <= 16; ++tozero)
        {
            //creating callbacks
            std::function<bool(std::vector<bool>, uint64_t)> verificationCallback = [](std::vector<bool>
  characteristic, uint64_t weight)
            {
                if(weight < 300)
                {
                    return true;
                }
                else
                {
                    return false;
                }
            };

            std::function<bool(std::vector<bool>, uint64_t)> shouldStoreVerifiedCallback = [](std::vector<bool>
  characteristic, uint64_t weight)
            {
                return true;
            };

            std::function<bool(std::vector<bool>, uint64_t)> shouldStoreNotVerfiedCallback = [](std::vector<bool>
  characteristic, uint64_t weight)
            {
                return false;
            };

            //running codingtheory analysis
            norx.runCodingTheoryAnalysis(tozero, verificationCallback,shouldStoreVerifiedCallback,
  shouldStoreNotVerfiedCallback, &resultSeries);
        }
        Logger::logMessage("done with NORX...");
        resultSeries.writeToLatexFile();
    }
    //-----------------------------------------------------------------
}
```

The code internally creating G and performing the analysis (`CipherAnalyzer.h`) is publicly available and the steps are documented in code. Note that the original version of the IAIK Coding Tool was adapted for this purpose.

4.5.2 NLTool

As briefly described in Chapter 3, the IAIK NLTool is a dedicated tool for performing cryptanalysis on symmetric primitives. It supports various functionalities including the search for characteristics, validation of characteristics as well as characteristic probability calculation.

It operates on an own cipher representation format written in C. This format model is constructed from different forms of steps performed in the cipher. There are different types of steps such as `LinearStep`, `BitsliceStep` or `CarryStep` for operations involving carry bits such as modular additions. Each of these steps describes a certain function modelled as template parameter within the step. Functions can be implemented as special methods in a defined class. Different steps are added to the cipher description using the `Add()` method to form the cipher representation. Code Example 4.2 gives an example of such a description. In can be noted that used variables must be added beforehand. This examples illustrated the approximation of the addition used in NORX as described in Chapter 2.

$$a = (a \oplus b) \oplus ((a \wedge b) \ll 1) \tag{4.1}$$

Code Example 4.2: NLTool representation of the NORX approximation of the addition process.

```
#include "functions.h"

test::test(int steps, int N)
{
  //defining codewords
  CodeWord a = AddConditionWord("a", 1, 0, 1);
  CodeWord b = AddConditionWord("b", 1, 0, 2);
  CodeWord temp1 = AddConditionWord("temp1", 1, 0, 3);
  CodeWord temp2 = AddConditionWord("temp2", 1, 0, 4);
  CodeWord temp3 = AddConditionWord("temp3", 1, 0, 5);

  //a XOR b
  Add(new BitsliceStep<XOR2>(N, a, b, temp1));

  //a AND b
  Add(new BitsliceStep<AND2>(N, a, b, temp2));

  //a AND b << 1
  temp3 = temp2->Shl(1);

  //a = (a XOR b) XOR ((a AND b) << 1)
  Add(new BitsliceStep<XOR2>(N, temp1, temp3, a));
}
```

This representation can be adapted in order to perform dedicated cryptanalysis. The goal of the automated framework in this case is to provide a basic representation which adoptions can be based upon.

The tool is then started using the command line with specific parameters defining the mode of operations and input files. The input file is structured in a way so that added `CodeWords` correspond to a variable declared in a grid model. The 3rd and 4th parameter in the `Add()` call represent the row and column in that input / output format grid.

4.5.3 STP

STP (Simple Theory Prover) is a constraint solver over the space of bit-vectors and single dimension arrays. It takes as input, equations and expressions describing a certain theorem and uses SAT solvers internally to figure out if this theorem is satisfiable. Therefore, the theorem can be defined in a high level language supporting operations such as concatenation, extraction, shifting, addition, multiplication signed modulo / division as well as bitwise Boolean operations and many more. So it provides a convenient, high level format to express such input equations, translates them to a low-level CNF representation needed by SAT solvers and proves their satisfiability or constructs counterexamples. The convenience in the input language lies in the rich set of available high level operations as well as rich data structures which make it very easy to express complex theorems.

Applied to cryptographic problems, STP can be used just like a SAT-solver (explained in Chapter 3) with a high level input language.

STP supports the different input languages CVC, SMT-LIB1 and SMT-LIB2. These are file format inputs that define a syntax of how to declare variables and operations on them. Further, it provides a C-interface that allows an easy way to express theorems and prove them. In this thesis, this interface is used to define cipher descriptions and obtain a theorem / set of equations that allows the verification of characteristics. However, we want to note that SAT or constraint solvers can also be used to search for characteristics for the cipher definition.

The C-interface works on the basic type `Expr`, which names every possible expression within STP. Further, for every supported operation it provides a function taking input expressions and returning the resulting expressions. After the whole theorem has been expressed in a system of equations (most likely equalities of `Expr` statements), it can be queried for validation returning if the expression could be validated. Queries can be, of course, applied to various components of a cipher description (or the corresponding expressions) separately (e.g. for each state word). If all expressions are valid, the whole system of expressions / equations is valid.

Code Example 4.3 gives an example of a STP description of the approximation of the addition operation used in NORX, as illustrated in Chapter 2 using the C-interface.

$$a = (a \oplus b) \oplus ((a \wedge b) \ll 1) \tag{4.2}$$

Code Example 4.3: STP representation of the NORX approximation of the addition operation

```
Expr test(int N)
{
    //creating handle
    VC handle = vc_createValidityChecker();

  //defining expression for input
  Expr a = vc_varExpr(handle, "a", vc_bvType(handle, N));
  Expr b = vc_varExpr(handle, "b", vc_bvType(handle, N));

  //a XOR b
  Expr temp1 = vc_bvXorExpr(handle, a, b);

  //a AND b
  Expr temp2 = vc_bvAndExpr(handle, a, b);

  //(a AND b) << 1
  Expr temp3 = vc_bvLeftShiftExpr(handle, 1, temp2);

  //(a XOR b) XOR ((a AND b) << 1)
  Expr result = vc_bvXorExpr(handle, temp1, temp3);

  return result;
}
```

4.6 Used Tools and Libraries

This section shortly describes and explains some important components and libraries used to establish the framework.

4.6.1 Transcompilers

A transcompiler, or also called source-to-source compiler or source-to-source translator, is a compiler that transforms code of one programming language into code of another programming language or format. In contrast to a conventional compiler, which transforms code into a language with a reduced abstraction level, both source and destination language have about the same abstraction level when using a transcompiler. Transcompilers usually consist of a *frontend* used to read and parse code from the source language, and a *backend* transforming it into the target language. So called source-to-source analysers, as used in this thesis, feature a software component that allows analysis and manipulation of the parsed code representation before being compiled to the target language. Figure 4.7 illustrates this principle. Between the frontend and the backend, the code is represented in a so called *intermediate representation* (IR). The design and capabilities of this IR are heavily dependent on the used transcompiler.

Transcompilers themselves are mostly used to port algorithms to other languages due to implementation constraints. Source-to-source translators on the other hand (among others) have two main applications:

- **Code analysis**: The structure of source-to-source translators offers great capabilities for program analysis that can be performed on the IR. Although this is

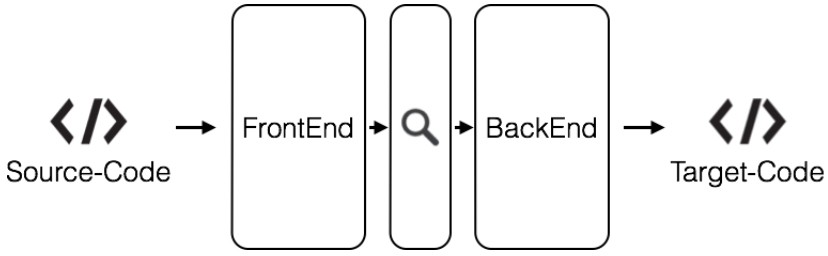

Figure 4.7: Basic structure of a source-to-source or analyser.

very dependent on the capabilities and complexity of the used IR, possible analysis types involve cycle detection, software verification, software security analysis, loop analysis, performance analysis as well as data flow analysis and many more. As an example, a source-to-source translator could be used to parse C-code and detect unwanted cycles in the program flow [24].

- **Code manipulation**: In this application scenario, existing code can be manipulated or new code can be selectively injected into the IR before the application of the backend. Therefore, arbitrary modification can be applied to the source code (in it's intermediate representation) even automatically. This involves performance optimization, automated unit test generation or interface generation. Statements (such as log messages) can be, for example, injected at the end of each method matching certain criteria or statements and function calls can be easily manipulated or interchanged [24]. In this mode the source-to-source translator acts a lot like a weaver known from modern aspect-oriented programming languages.

In the framework introduced in this thesis, such a source-to-source translator is used to automatically parse cipher representations from source code into an abstract (yet powerful) IR, perform modifications on the source code to bring it to an simple standard form and implement adapters to transform the modified IR to input formats used by different tools. For detailed information on this process, we refer to Chapter 5.

4.6.2 The ROSE Compiler Framework

The *ROSE* compiler infrastructure is a tool developed by the *Lawrence Livermore National Laboratory* with the goal of providing a framework for standard compiler features to non-compiler experts. It supports frontends and backends for various different programming languages, a very powerful IR as well as various features for manipulating, analysing and constructing parts of the IR. Among many other features, it provides high level, mid level as well as low level interfaces for construction of code in the IR, program and data flow analysis features as well great tools for automated optimization of source code [24, 25].

Rose itself is written in C++, so the translator code performing the IR analysis or

manipulation has to be written in C++ as well. Even though ROSE consists of an extremely large number of different components, the following were used during the tasks performed in this thesis as illustrated in Figure 4.8.

- The **intermediate representation** SAGE III states the core of the ROSE framework. It is parsed from the input code by the fronted and preserves and contains a great variety of different informations such as positions of elements in the original code, comments or performance relevant information. The intermediate presentation itself and it's features are described later in this section.
- The **SageInterface** states an interface for manipulating and transforming parts and nodes of the IR. It is used for all transformations and optimizations on it. For example, an existing loop in the source code can be located and unrolled or optimized according to the users needs with this interface.
- The **SageBuilder** interface is used to create elements of the IR from scratch. If not only existing structures need to be transformed, but a completely new one needs to be injected, this interface is used. It can, for example, be used to construct a new method body including return type and parameter list.
- The **QueryEngine** is an interface providing rich query operations to the IR. It allows to query for certain structures in the IR and therefore prevents expensive traversal of the IR data structures in order to find relevant elements. For example, this is used to locate all assign operations in the source code (or parts of it) matching certain criteria (such as e.g. left hand side is an integer variable).
- The **Translation Code** guides the procedure of tranlating the code. It invokes the frontend to create the IR, calls the components (such as SageInterface, SageBuilder or QueryEngine) to work on the code represented by the IR, and calls the backend to transform it to the target code.

The function and interaction of these components is shown in a very simple example in Code Example 4.4. The shown code uses the frontend to obtain an IR representation and invokes a query for all `for` statements in the intermediate representation using the QueryEngine. It then iterates over all found `for` statements and prints their line of occurrence in the original source code. Further, it uses the SageInterface to attach a comment to each `for` statement. In the last part, it calls the backend to invoke translation back to source code.

Code Example 4.5 shows a simple input program (source code) to that translation code. It can be seen that two for statements are present in that code.

Code Example 4.5 illustrates the output of this translation code. It can be seen that it correctly detected the for statements and printed their line in the source code.

Code Example 4.5 shows the resulting target code with the comments attached to the for statements. This code was generated from the IR using the backend.

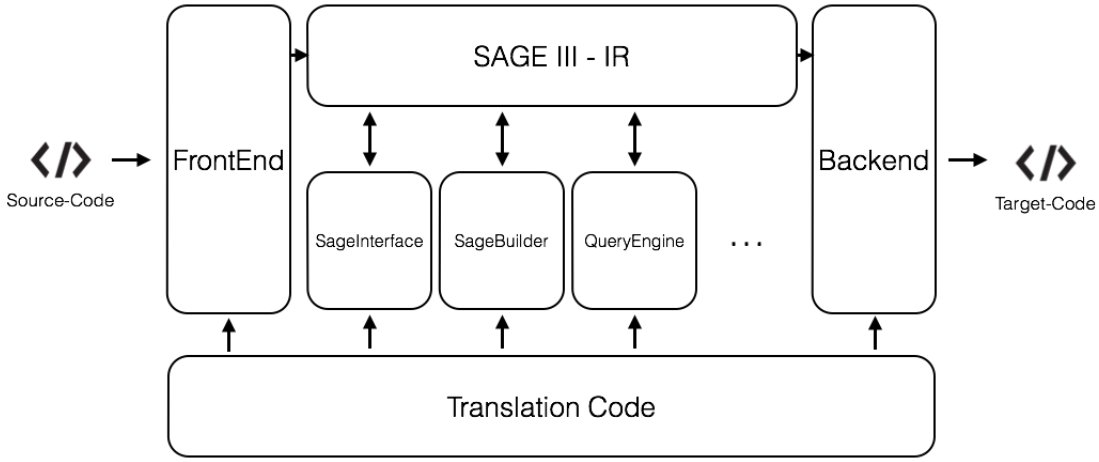

Figure 4.8: The basic components of the ROSE framework.

Code Example 4.4: Simple translation code using the described components to find for loops in an input source code

```
#include "rose.h"

int main (int argc, char** argv)
{
    // 1) invoke frontend to get IR
    SgProject* project = frontend(argc, argv);
    ROSE_ASSERT (project != null);

  //2) invoking a query for loops and perform operations
    //getting all for loops in project
    Rose_STL_Container<SgNode*> loopsInFunction = NodeQuery::querySubTree(project, V_SgForStatement);

    //iterating over for loops
    Rose_STL_Container<SgNode*>::iterator it = loopsInFunction.begin();
    while(it != loopsInFunction.end())
    {
        SgForStatement* forStatement = isSgForStatement(*it);
        std::cout << "found for loop at: " << forStatement->get_file_info()->get_line() << std::endl;

        //attaching comment to for statement
        SageInterface::attachComment(forStatement, "this comment was attached");
        it++;
    }

    //3) call the backend to transform back to source code
    return backend(project);
}
```

Code Example 4.5: Input code to the simple example. Two for loops are present.

```

int main ()
{
    for(int i = 0; i < 10; ++i)
    {
        i = i + 100;
        for(int j = 0; j < i; ++j)
        {
            j = j + 5;
            if(j > 100)
                return i+j;
        }
```

```
13      }

15      return 0;
}
```

Code Example 4.6: Output of the simple example

```
[100%] Built target cipherTranslator
found for loop at: 4
found for loop at: 7
demo@ubuntu:~/cipherTranslator/_build$
```

Code Example 4.7: Target code generated with the backend from the manipulated IR

```

int main()
{
// this comment was attached
  for (int i = 0; i < 10; ++i) {
    i = (i + 100);
// this comment was attached
    for (int j = 0; j < i; ++j) {
      j = (j + 5);
      if (j > 100)
        return i + j;
    }
  }
  return 0;
}
```

This states a very simple example of how the different components can be applied to analyse and transform input code. In this case, the translation code as well as the source and target code is written in C++.

The SAGE III Intermediate Representation

The SAGE III intermediate representation is the one used by ROSE to represent source code parsed. It allow transformations and modifications in an abstract format as well as providing the basis for translating into other languages. The intermediate language is based on SAGE++ developed at the University of Indiana. It is an IR with full support for object oriented languages including features like templates, AST (abstract syntax tree) visualizations, full support for C, C99, UPC, C++, Fortran 66, Fortran 77, Fortran 90/96, Fortran 2003 and many more. It preserves information about the location of nodes in the source code, comments and preprocessor directives, allows easy copying of parts of the IR as well as automated correctness checks on parts of the AST, easy manipulation and an AST to string interface allowing easy generation of strings based on parts of the IR [24].

The IR is AST-based and enhanced with a lot of meta-information about the source code the AST was parsed from.

Basic Interfaces and Transformations

This section provides an overview of some important operations and transformations supported by the ROSE framework. It shows the variety of different features that the framework in combination with the IR provides. We refer to [24] and [25] for detailed information and tutorials on the available features:

- **AST Traversal**: ROSE supports various different methods for traversing the IR / AST. In addition to the standard tree traversal methods (post / pre / simple) it provides a structure to inherit AST attributes during a traversal as well as nested traversals. This allows to perform manual transformations on the AST (which don't use any of the other transformation interfaces) very efficiently.
- **AST Query**: ROSE provides an own query engine, that can be used to efficiently query the IR or parts of it using an easy query language. It can query for specific types of nodes or lets users implement custom callback methods to filter query results. Predefined queries make it easier to query for complex IR-node types (e.g. query for methods / functions with certain numbers of arguments etc.).
- **Loop Optimization**: ROSE contains a powerful and automated transformation interface (within SageInterface) dedicated to loops. It supports automated loop interchange, loop blocking, loop fusion, loop unrolling as well as an automated optimization part to transform loop-IR representations. These methods operate automatically, detect boundaries and iterations in loops and perform the transformation accordingly.
- **AST Transformation**: The SageInterface components allow easy and high level manipulation of the AST. It operates on found (traversal, query etc.) nodes of the IR and allows to manipulate parts of the IR in various different ways. For example, it allows adding comments, splitting expressions statements, in-lining of function- or method-calls as well as outlining parts of the code in own methods or classes. Even though lower level interfaces for manipulating the IR exist for complex tasks, the SageInterface allows easy and straightforward implementation of commonly used transformations. In Chapter 5, this interface is heavily used to transform source code from reference implementation in order to bring it into a standardized form to be able to translate it to arbitrary tool representations.
- **AST Generation**: The SageBuilder interface provides functionality to build own parts of the IR from scratch and inject it into the parsed source code (using the SageInterface class). It offers methods for creating virtually any programming language component present in the supported source languages. Therefore, it allows users to create whole functions, methods or classes including implementation code in the IR and inject it into the parsed program at arbitrary positions. Chapter 5 shows examples of how this was used to create code blocks used in different tool input formats.

- **Program Analysis**: Besides the capabilities to manipulate parts of the existing IR or create new ones, ROSE supports a huge variety of analysis methods. This ranges from call-graph analysis, class hierarchy analysis, control flow analysis, data-flow analysis or dependence analysis. For all these types, ROSE provides various methods to perform subtasks commonly used and partially automatic and predefined analysis tools (such as an automated creation of a call and class hierarchy graph etc.).

In the next chapter, we show how an automated analysis framework based on the principles and techniques described in this chapter was implemented. It gives an overview of the implemented components, processes and transformations and illustrates the different steps using the implementation of NORX.

Chapter 5

Automated Analysis Framework

This chapter finally introduces the architecture, design and implementation of the automated preliminary analysis framework. It describes the structure of the software, it's components and the process of conducting an analysis inside the framework. All of the transformations and processes are illustrated using NORX. Further, it describes the challenges, limitations and future work and shows how it can be extended to customize its behaviour and include other tools.

As mentioned in earlier chapters, the framework implemented in the scope of this thesis consists of two main parts. Firstly, the IAIK CodingTool was extended in order to be able to automatically search for potential characteristics. Secondly, and automated transformation framework was implemented using ROSE with the goal of being able to perform automated transformation of cipher representations.

In the following, the two components are explained in detail. Note that in this chapter, the term *source function* refers to the function passed as an input to the transformation, whereas *target function* names the transformed function.

5.1 CipherAnalyzer

CipherAnalyzer names the component performing a coding theory based search for codewords using the *IAIK Coding Tool* as briefly discussed in Chapter 4. In this chapter, the basic procedure and components of this extended tool are briefly discussed.

5.1.1 Components

The extended tool has two main components:

1. **CipherAnalyzer**: Builds up the generator matrix for the linearised model of the analysed cipher, performs code-shortening and searches for codewords.
2. **CodingTheoryAnalyzerDelegate**: A delegate interface (abstract class) that implements delegate methods used by the `CipherAnalyzer` to verify found codewords and determine which codewords to store in the results.

The two objects interact through `CipherAnalyzer` calling the delegate methods of an instance implementing the `CodingTheoryAnalyzerDelegate` methods and processing found codewords based on the return type. The delegate never calls any `CipherAnalyzer` methods and will in practise most likely be implemented by the class running the analysis code.

5.1.2 Procedure

The procedure of conducting a coding theory based search follows the exact procedure described in Section 4.5.1. The basic external analysis code of this tool has already been explained, so this part will focus on the internals of the `CipherAnalyzer::runCodingTheoryAnalysis` method. The method firstly defines a callback closure used to build up a generator matrix row. It then calls `CodeMatrix::Build` of the CodingTool, which then invokes the callback for each row of the generator matrix. The callback method builds the generator matrix row as discussed in Section 4.5.1. After code shortening has been applied to a defined number of state bits (passed to the method) the tool starts the search for characteristics by calling the `lowWeightSearch.ChanteatChabaud` method. After the found word has been verified by the callback, it is added to the external results object. For detailed information, we refer to the in-code documentation as the code is publicly available.

5.1.3 CipherTranslator

CipherTranslator names the part of the software handling transformations of cipher representations. It takes as an input a reference implementation in C, parses it into an IR (as described in Chapter 4), applies certain transformations on it to bring it to a standard form and finally allows the implementation of adapters to transform it into other representations used by the analysis tools.

5.2 Process

In general the implementation facilitates the ROSE compiler framework for parsing code into an IR and uses the functions within the IR to bring it to a standard form. Instead of the backend, it uses a component that allows users to implement adapters in order to parse the IR into other cipher representation formats. Therefore, the ROSE backend is

not used but might be facilitated in future extensions of the framework. Figure 5.1 illustrates the basic process using terms related to transcompilers as introduced in Chapter 4.

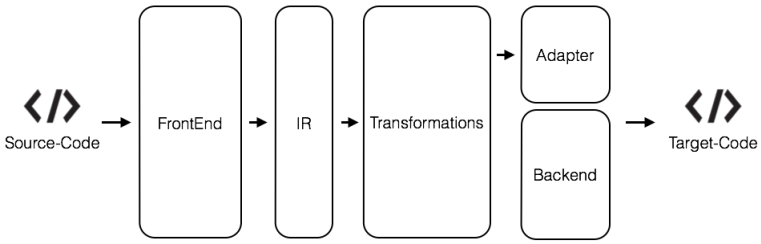

Figure 5.1: Overall picture of the process implemented by the framework.

5.3 Components

The translation framework mainly consists of three components:

- `TranslationUtils`: This static components implements all tools and transformations used by the framework. It features all transformations applied by the framework. Besides transformation methods, it implements methods for querying the IR-AST for certain specific elements, inserting needed elements into it as well as many other features that might be required in an analysis process. The method `transformFunctionToStandardForm` can be used to transform any function into the desired standard form, which will be described later in Section 5.4.
- `CodeGenerationUtils`: This class implements utility methods used to generate template code in order to perform a translation.
- `CodeTransformer`: This class is responsible for leading the transformation to another format. It traverses the function in standard form and uses the implementation of a specific `ToolTranslator` to transform statements. Further, it injects the transformed statements into a target function to create the transformed code.
- `ToolTranslator`: This component is responsible for setting the target environment and providing the translation of certain statements to the target language. For example, it would setup a special function representing a round function in the target format and translate occurrences of XOR operations in the source code to the string representing an XOR in the target format.
- `AnalysisCode`: This component is individually implemented by a user facilitating the other components. This analysis code is used to transform cipher representations, search for characteristics (e.g. using the tool introduced in Section 5.1) and verify found characteristics.

5.4 Transformations

The *CipherTranslator* part of the framework performs several transformations on the input code after parsing (IR) in order to transform it into a so called standard form. This standard form is defined by two criteria:

a) **Operation isolation**: Only one of the supported operations is allowed in an assignment per line of code. For example, the expression

$$a = (a \oplus b) \ll 7 \tag{5.1}$$

(not in standard form) would equal

$$temp1 = a \oplus b \tag{5.2}$$
$$a = temp1 << 7 \tag{5.3}$$

in standard form. So only one assignment to a variable resulting from one binary operation (taking two operands) is allowed per line. This is required since the framework tries to transform as small portions of the input code as possible at a time, resulting in easier implementations of the adapters performing this task.

b) **Function isolation**: No function calls are allowed within the source function in standard form. So all operations must be performed within this function, not relying on external code. Since the framework transforms a finite set of operations in the cipher, it must be able to find boundaries for what to transform, which justifies this property of the standard form.

c) **Statement singularity**: No operators applying multiple operations at once in the background shall be present. This applies to elements such as compound or overloaded operators and enforces a kind of *canonical* representation of operations. This is required in order to limit the complexity of the framework. Considering overloading or complex background definitions of operations would exceed the scope of this tool.

The transformation into such a format is performed in several steps internally as illustrated in Figure 5.2. This section explains the different transformation steps and illustrates them using NORX as an example.

5.4.1 Inlining

The first operation performed aims towards function isolation. In functions to be transformed, various calls to other functions or macros are made. This does not align with the desired standard form, since it makes it very difficult to parse all operations / expressions in the function in order to transform it into another representation format required by other tools.

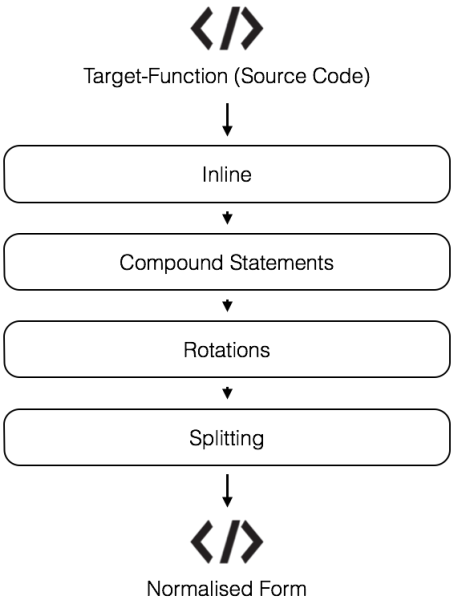

Figure 5.2: Steps of transformation to standard form

The inlining detects calls to functions or macros and integrates their definition in the function. Therefore, the call to a function is substituted with the real operations performed in the function. This ensures function isolation, since all external operations are integrated into one function.

As a simple example, take the function `test` illustrated in Code Example 5.1. It calls the function `int op(int x, int y)` to perform a certain calculation.

Code Example 5.1: Input code for the inlining transformation

```
1  int test(int a, int b)
   {
3    a = a ^ b;

5    //calling external function
     b = op(a, b);
7
     return a + b;
9  }

11 int op(int a, int b)
   {
13   int temp = a ^ b;
     return temp ^ 16;
15 }
```

After applying the inlining operation, the steps performed in the call of `op` are integrated into the original function an not dependent on any external code any more as illustrated in Code Example 5.2.

Code Example 5.2: Code after inlining was applied to *function*

```
int function(int a, int b)
{
  a = a ^ b;

  //inlined code
  int temp = a ^ b;
  b =  temp ^ 16;

  return a + b;
}
```

In ROSE, inlining of a function call is supported using the `SageInterface::doInline` method. This is used in the method `inlineStatementsInFunction`, implemented in `TranslationUtils` and is illustrated in Code Example 5.3. It issues a query to obtain all function calls within the function. Then, it iterates over the list of function calls and invokes the inlining method to inline them recursively. If the process could be completed successfully, it returns `true` and the method quits the inner loop, rescans for function calls and starts the inlining process again. This construct of two loops prevents errors / assertions due to trying to inline a function call already processed recursively and therefore not valid anymore.

After all function calls have been inlined, some ROSE methods are called that cleanup the IR nodes manipulated and check the IR.

Code Example 5.3: Method for inlining all function calls in a source function using the ROSE framework.

```
void TranslationUtils::inlineStatementsInFunction(SgFunctionDeclaration* function, SgProject* project)
{
    bool astWasModified = false;
    int inlineCount = 0;

    //inlining function calls
    do
    {
        //resetting indicator variable
        astWasModified = false;

        //querying for all assings
        Rose_STL_Container<SgNode*> functionCalls = NodeQuery::querySubTree(function,V_SgFunctionCallExp);

        //getting iterator
        Rose_STL_Container<SgNode*>::iterator it = functionCalls.begin();

        //iterating over function calls
        while(astWasModified == false && it != functionCalls.end())
        {
            //getting current function call and make sure it is one
            SgFunctionCallExp* currentFunctionCall = isSgFunctionCallExp(*it);
            ROSE_ASSERT(currentFunctionCall != null);

            std::cout << "found function call " << currentFunctionCall->getAssociatedFunctionSymbol()->get_name()
      << std::endl;

            //inlining function call recursively
            astWasModified = doInline(currentFunctionCall, true);

            if(astWasModified == true)
                std::cout << "Sucessfully inlined function and modified the ast" << std::endl;

            //incrementing iterator
            it++;
        }
    }
    while(astWasModified == true && inlineCount < 100);

    //cleaning up
    // Call function to postprocess the AST and fixup symbol tables
```

```
    FixSgProject(*project);

    // Rename each variable declaration
    //renameVariables(project);

    // Fold up blocks
    flattenBlocks(project);

    // Clean up inliner-generated code
    cleanupInlinedCode(project);
}
```

5.4.2 Fixing Compound Statements

Compound statements, in this case, name operators of programming languages combining two operations in one operator. An example for this would be common operators like `a-=b`, `a+=b` or `a&=b`. These statements do not violate operation isolation or function isolation, but do not align with operation singularity. Such operations make it very hard to parse and transform code, since the complexity of the algorithm and the number of different cases requiring consideration rises significantly. Therefore, these statements are not allowed in the standardised form and have to be substituted with their *canonical* representations. For example, the line `int i &= 4;` would be replaced with `int i = i & 4;`. The method handling compound statements in this framework is illustrated in Code Example 5.4.

The method forms a query to obtain all compound assignment IR-nodes within the target function, iterates over these statements and extracts their left hand side and right hand side expression. Afterwards it distinguishes between the different possible types of compound assignments and constructs a new (and equal) expression in canonical form using the left and right hand side expressions. Finally, it uses the `SageBuilder` interface to build the new operation and inserts it into the IR using the `SageInterface` component.

Code Example 5.4: Code fixing compound statements using the ROSE interface.

```
void TranslationUtils::fixCompountAssignStatementInFunction(SgFunctionDeclaration* function)
{
    std::cout << "Searching and fixing compound assign statements" << std::endl;

    //getting all compound statements
    Rose_STL_Container<SgNode*> compoundAssignStatements = NodeQuery::querySubTree(function, V_SgCompoundAssignOp);

    Rose_STL_Container<SgNode*>::iterator iter;
    for(iter = compoundAssignStatements.begin(); iter != compoundAssignStatements.end(); iter++)
    {
        SgCompoundAssignOp* compoundOp = isSgCompoundAssignOp(*iter);
        ROSE_ASSERT(compoundOp != null);

        //getting copy of node for later replacement
        SgCompoundAssignOp* compoundOpCopy = SageInterface::deepCopy(compoundOp);

        //std::cout << "Fixing expression: " << compoundOpCopy->unparseToCompleteString() << std::endl;

        //getting left and right component
        SgExpression* lhs = compoundOpCopy->get_lhs_operand();
        SgExpression* rhs = compoundOpCopy->get_rhs_operand();

        // std::cout << "Detected sub expressions - lhs: " << lhs->unparseToCompleteString() << " rhs: " << rhs->
    unparseToCompleteString() << std::endl;

        //defining variable to write expression to
        SgExpression* newExpression;

```

```
        //handling different cases of compound element
29      if(isSgAndAssignOp(compoundOp))
        {
31          newExpression = SageBuilder::buildAndOp(lhs, rhs);
        }
33      else if(isSgDivAssignOp(compoundOp))
        {
35          newExpression = SageBuilder::buildDivideOp(lhs, rhs);
        }
37      else if(isSgExponentiationAssignOp(compoundOp))
        {
39          newExpression = SageBuilder::buildExponentiationOp(lhs, rhs);
        }
41      else if(isSgIntegerDivideAssignOp(compoundOp))
        {
43          newExpression = SageBuilder::buildIntegerDivideOp(lhs, rhs);
        }
45      else if(isSgLshiftAssignOp(compoundOp))
        {
47          newExpression = SageBuilder::buildLshiftOp(lhs, rhs);
        }
49      else if(isSgRshiftAssignOp(compoundOp))
        {
51          newExpression = SageBuilder::buildRshiftOp(lhs, rhs);
        }
53      else if(isSgMinusAssignOp(compoundOp))
        {
55          newExpression = SageBuilder::buildSubtractOp(lhs, rhs);
        }
57      else if(isSgModAssignOp(compoundOp))
        {
59          newExpression = SageBuilder::buildModOp(lhs, rhs);
        }
61      else if(isSgMultAssignOp(compoundOp))
        {
63          newExpression = SageBuilder::buildMultiplyOp(lhs, rhs);
        }
65      else if(isSgPlusAssignOp(compoundOp))
        {
67          newExpression = SageBuilder::buildAddOp(lhs, rhs);
        }
69      else if(isSgXorAssignOp(compoundOp))
        {
71          newExpression = SageBuilder::buildBitXorOp(lhs, rhs);
        }
73
        //creating new assign statement
75      SgAssignOp* newOperation = SageBuilder::buildAssignOp(lhs, newExpression);

77      std::cout << "Fixing Compounds: " << compoundOp->unparseToCompleteString() << " was replaced with " <<
    newOperation->unparseToCompleteString() << std::endl;

79      //replacing old expression with newly created one
        SageInterface::replaceExpression(compoundOp, newOperation);
81  }
}
```

5.4.3 Handle Rotations

A major drawback of the approach of parsing reference implementations into an abstract cipher representation is the fact, that certain operations used in the ciphers might not be natively supported by the programming language of the reference implementation. Therefore, the designer needs to construct it using other operations, and more importantly the framework needs to pay attention to it and detect and transform these compound operations back. This is based on the assumption, that tool input formats might support the operations and they can therefore be directly translated. This problem results from the different operation sets of different programming languages or formats and can not be prevented following the given approach.

The only compound operation needed in the work done in this thesis - due to the ciphers considered - is the rotation operation. Since C (reference implementation) does not support rotations, they are constructed using two shift operations. The following expressions are used when a donates the variable shifted, x states the number of bits shifted, N donates the word size (so $|x| = N$) and $\ggg$ and $\lll$ donate the left and right rotation. The operation simply shifts x in the corresponding direction (all shifted in bits will be zero) and adds the circular behaviour of the rotation by calculating the remaining (the shifted in) bits by simply shifting a by $N - x$ into the other direction.

$$x = a \ggg x \quad \Leftrightarrow x = (a \gg x) \oplus (a \ll (N - x)) \Leftrightarrow x = (a \gg x) \vee (a \ll (N - x)) \quad (5.4)$$

$$x = a \lll x \quad \Leftrightarrow x = (a \ll x) \oplus (a \gg (N - x)) \Leftrightarrow x = (a \ll x) \vee (a \gg (N - x)) \quad (5.5)$$

Either the `OR` or `XOR` operation can be used to combine the two shift operations.

These four constructs need to be detected by the operation and treated as rotations in all following processes. The framework handles this by not changing the code itself, but annotating all compound operations with a comment defining its real operation. Therefore, any rotation would not be changed in the code, but annotated in an appropriate way. All further processes check for this annotation and treat the code line accordingly. The method detecting and annotating rotations is shown in Code Example 5.5.

Code Example 5.5: Code detecting rotations using a query calling a specified callback method and handling them accordingly by annotation.

```
void TranslationUtils::handleRotationStatementsInFunction(SgFunctionDeclaration* function)
{
    Rose_STL_Container<SgNode*> expressions = NodeQuery::querySubTree(function, V_SgExpression);
    Rose_STL_Container<SgNode*> rotationExpressions = NodeQuery::queryNodeList(expressions, &
      queryRotationOperationSolver);

    std::cout << "Found " << rotationExpressions.size() << " rotation expressions" << std::endl;

    Rose_STL_Container<SgNode*>::iterator iter;
    for(iter = rotationExpressions.begin(); iter != rotationExpressions.end(); iter++)
    {
        SgExpression* exp = isSgExpression(*iter);
        std::cout << "handling rotation " << exp->unparseToCompleteString() << std::endl;

        //checking if rotation top level element is binary operation
        SgBinaryOp* rotation = isSgBinaryOp(*iter);
        ROSE_ASSERT(rotation != null);

        //creating indicator comments
        std::string indicator = COMPOUNT_STATEMENT_ANNOTATION;
        std::string comment = "@ROT";

        //determining direction of shift / rotation
        SgBinaryOp* shiftExpression = isSgBinaryOp(rotation->get_lhs_operand());
        if(isSgRshiftOp(shiftExpression))
        {
            comment = comment + "R(";
        }
        else if(isSgLshiftOp(shiftExpression))
        {
            comment = comment + "L(";
        }

        ROSE_ASSERT(shiftExpression != null);

        //std::cout << "The type of the shifted thing is " << shiftExpression->get_lhs_operand()->sage_class_name()
      << std::endl;
        //std::cout << "The type of the shift thing is " << shiftExpression->get_rhs_operand()->sage_class_name()
     << std::endl;
```

```
38          SgExpression* expr = shiftExpression->get_lhs_operand();
            //checking argument
40          ROSE_ASSERT(expr != null);

42          //shift expression can be variable or integer, determining and adapting comment accordingly
            SgIntVal* shiftIndex = isSgIntVal(shiftExpression->get_rhs_operand());
44          if(shiftIndex != null)
            {
46              comment = comment + expr->unparseToCompleteString() + ", " + boost::lexical_cast<std::string>(
      shiftIndex->get_value()) + ")";
            }
48          else if(isSgVarRefExp(shiftExpression->get_rhs_operand()))
            {
50              comment = comment + expr->unparseToCompleteString() + ", " + isSgVarRefExp(shiftExpression->
      get_rhs_operand())->get_symbol()->get_name()  + ")";
            }
52          else
            {
54              std::cout << "Could not identify rotation variable, going to next one..." << std::endl;
                continue;
56          }

58          //attaching comment to rotation statement
            SageInterface::attachComment(rotation, indicator);
60          SageInterface::attachComment(rotation, comment);
        }
62  }
```

It uses a query with the provided callback function `queryRotationOperationSolver` to find rotation statements, iterates over the list of detected rotations, determines the type and operands of it and adds the annotations as comments to the statement.

The callback method detects rotations by specifically searching for AST-tree structures representing the operations stated before when covering multiple cases of elements involved in the rotation definition. If it detects that a node (passed to the method) represents a rotation, it returns the node itself (in a vector). This rotation detection method, as well as the callback function, are shown in Code Example 5.6.

Code Example 5.6: Callback method and rotation detection method.

```
NodeQuerySynthesizedAttributeType TranslationUtils::queryRotationOperationSolver(SgNode* astNode)
2   {
        //node must not be null
4       ROSE_ASSERT(astNode != 0);
        NodeQuerySynthesizedAttributeType returnType;
6
        if(detectRotation(astNode) == true)
8       {
            returnType.push_back(astNode);
10      }

12      return returnType;
    }
14
    bool TranslationUtils::detectRotation(SgNode* astNode)
16  {
        //checking top level operation
18      if(isSgBitOrOp(astNode) != null || isSgBitXorOp(astNode) != null)
        {
20          //std::cout << "Detected OR in expression " << orOp->unparseToCompleteString() << std::endl;

22          SgBinaryOp* orOp = isSgBinaryOp(astNode);
            ROSE_ASSERT(orOp != null);
24
            //getting operats
26          SgExpression* leftOperand = orOp->get_lhs_operand();
            SgExpression* rightOperand = orOp->get_rhs_operand();
28
            //checking if negated rotation directions are given
30          if((isSgRshiftOp(leftOperand) && isSgLshiftOp(rightOperand)) || (isSgLshiftOp(leftOperand) && isSgRshiftOp(
      rightOperand)))
            {
32              int test = astNode->depthOfSubtree();
```

```
34            //std::cout << "Structure of shifts fits" << std::endl;

36            SgBinaryOp* shortShift = null;
              SgBinaryOp* longShift = null;
38
              //the thing that is shifted
40            SgExpression* shiftedExpression = null;

42            //the value used for shifting
              SgExpression* shiftExpression = null;
44
              int leftDepth = leftOperand->depthOfSubtree();
46            int rightDepth = rightOperand->depthOfSubtree();

48            bool leftContainsArrayRef = NodeQuery::querySubTree(leftOperand, V_SgPntrArrRefExp).size() == 0 ? false
       : true;
              bool rightContainsArrayRef = NodeQuery::querySubTree(rightOperand, V_SgPntrArrRefExp).size() == 0 ?
      false : true;
50
              //std::cout << "left: " << leftOperand->unparseToCompleteString() << " depth: " << boost::lexical_cast<
      std::string>(leftDepth) << " arrayref in there: " << boost::lexical_cast<bool>(leftContainsArrayRef) << std::
      endl;
52            //std::cout << "right: " << rightOperand->unparseToCompleteString() << " depth: " << boost::
      lexical_cast<std::string>(rightDepth) << " arrayref in there: " << boost::lexical_cast<bool>(
      rightContainsArrayRef) << std::endl;

54            //determining short shift
              if((leftContainsArrayRef == false && leftDepth == 1) || (leftContainsArrayRef == true && leftDepth ==
      2))
56            {
                  shortShift = isSgBinaryOp(leftOperand);
58            }
              else if((rightContainsArrayRef == false && rightDepth == 1) || (rightContainsArrayRef == false &&
      rightDepth == 2))
60            {
                  shortShift = isSgBinaryOp(rightOperand);
62            }

64            //determining long shift
              if((leftContainsArrayRef == false && leftDepth == 4) || (leftContainsArrayRef == true && leftDepth ==
      5))
66            {
                  longShift = isSgBinaryOp(leftOperand);
68            }
              else if((rightContainsArrayRef == false && rightDepth == 4) || (rightContainsArrayRef == true &&
      rightDepth == 5))
70            {
                  longShift = isSgBinaryOp(rightOperand);
72            }

74            //checking long and short assignment result
              if(longShift == null || shortShift == null || longShift == shortShift)
76                return false;

78            //std::cout << "Detected short shift side as " << shortShift->unparseToCompleteString() << std::endl;
              //std::cout << "Detected long shift side as " << longShift->unparseToCompleteString() << std::endl;
80
              //checking short shift
82            shiftedExpression = shortShift->get_lhs_operand();
              shiftExpression = shortShift->get_rhs_operand();
84
              //saving type of shiftexpression
86            SgType* shiftExpressionType = shiftedExpression->get_type();

88            //std::cout << "shifted expression: " << shiftedExpression->unparseToCompleteString() << " type " <<
      shiftedExpression->sage_class_name() << " shift expression: " << shiftExpression->unparseToCompleteString() <<
       " type " << shiftExpression->sage_class_name() << std::endl;

90            //checking operands
              if(shiftedExpression == null || shiftExpression == null)
92                return false;
              if(isSgVarRefExp(shiftedExpression) == null && isSgValueExp(shiftedExpression) == null &&
      isSgPntrArrRefExp(shiftedExpression) == null)
94                return false;
              if(isSgVarRefExp(shiftExpression) == null && isSgValueExp(shiftExpression) == null && isSgPntrArrRefExp
      (shiftedExpression) == null)
96                return false;

98            //checking long expression
              //checking shifted item
100           if(longShift->get_lhs_operand() != shiftedExpression)
              {
```

```
102             if(longShift->get_lhs_operand()->unparseToCompleteString().compare(shiftedExpression->
    unparseToCompleteString()) != 0)
                {
104                 std::cout << longShift->get_lhs_operand()->unparseToCompleteString() << " type " << longShift->
    get_lhs_operand()->sage_class_name() << " NOT EQUAL "
                        << shiftedExpression->unparseToCompleteString() << " type " << shiftedExpression->
    sage_class_name() << std::endl;
106                 return false;
                }
108         }

110         //checking that child is minus operation
            SgSubtractOp* subOp = isSgSubtractOp(longShift->get_rhs_operand());
112         if(subOp == null)
                return false;
114

116         //std::cout << "Detected subtract op " << subOp->unparseToCompleteString() << std::endl;
            //            std::cout << "rhs " << subOp->get_rhs_operand()->unparseToCompleteString() << " of type "
     << subOp->get_rhs_operand()->sage_class_name() << " get type " <<
118         //                    subOp->get_rhs_operand()->get_type()->unparseToCompleteString() << "depth " <<
    boost::lexical_cast<std::string>(subOp->get_rhs_operand()->depthOfSubtree()) << std::endl;

120
            //std::cout << "Before detection of shift type " << std::endl;
122
            //performing queries on subtree to detect type of shifted expression
124         Rose_STL_Container<SgNode*> integerInSubtree = NodeQuery::querySubTree(subOp->get_rhs_operand(),
    V_SgIntVal);
            Rose_STL_Container<SgNode*> variableExpressionsInSubtree = NodeQuery::querySubTree(subOp->
    get_rhs_operand(), V_SgVarRefExp);
126         Rose_STL_Container<SgNode*> arrayRefExpressionsInSubtree = NodeQuery::querySubTree(subOp->
    get_rhs_operand(), V_SgPntrArrRefExp);
            //case 1 shifted value is integer
128         if(integerInSubtree.size() == 1)
            {
130             SgIntVal* subtractIntVal = isSgIntVal(integerInSubtree.at(0));
                SgIntVal* shiftIntVal = isSgIntVal(shiftExpression);
132
                if(shiftIntVal == null)
134                 return false;

136             if(subtractIntVal->get_value() != shiftIntVal->get_value())
                    return false;
138         }
            //case 2 shifted value is variable
140         else if(variableExpressionsInSubtree.size() == 1)
            {
142             //checking if it is a variable Reference
                SgVarRefExp* varRef = isSgVarRefExp(variableExpressionsInSubtree.at(0));
144             ROSE_ASSERT(varRef);

146             //converting shiftexpression, if it is not a varref, the expression is not a rotation
                SgVarRefExp* shiftVarRefExpression = isSgVarRefExp(shiftExpression);
148             if(shiftVarRefExpression == null)
                    return false;
150             //
                //                  std::cout << "Found varref: " << varRef->unparseToCompleteString() << std::
    endl;
152             //
                //                  std::cout << "Comparing variable " << varRef->unparseToCompleteString() << "
    of type " << varRef->sage_class_name() << " with variable " <<
154             //                          shiftExpression->unparseToCompleteString() << " of type " <<
    shiftExpression->sage_class_name() << std::endl;

156             //must be variable, comparing
                if(varRef->get_symbol() != shiftVarRefExpression->get_symbol())
158             {
                    //                      std::cout << "Not equal" << std::endl;
160                 return false;
                }
162         }
            else if(arrayRefExpressionsInSubtree.size() == 1)
164         {
                SgPntrArrRefExp* refExpr = isSgPntrArrRefExp(arrayRefExpressionsInSubtree.at(0));
166             ROSE_ASSERT(refExpr);

168             SgPntrArrRefExp* shiftPntrRefExpr = isSgPntrArrRefExp(shiftExpression);
                if(shiftPntrRefExpr == null)
170                 return false;

172             if(refExpr != shiftPntrRefExpr)
                    return false;
```

```
174             }
176             //searching for multiplies in subtree
                Rose_STL_Container<SgNode*> multiplies = NodeQuery::querySubTree(subOp->get_lhs_operand(),
    V_SgMultiplyOp);
178             if(multiplies.size() == 0)
                {
180                 //std::cout << "Detected const value shift " << shortShift->unparseToCompleteString() << std::endl;
182                 SgIntVal* bitSize = isSgIntVal(subOp->get_lhs_operand());
                    if(bitSize == null)
184                     return false;
186                 if(!isPowerOfTwo(bitSize->get_value()))
                        return false;
188             }
                else if(multiplies.size() == 1)
190             {
                    SgMultiplyOp* multOp = isSgMultiplyOp(multiplies.at(0));
192                 ROSE_ASSERT(multOp != null);
194                 //std::cout << "Detected multiply op " << multOp->unparseToCompleteString() << " right type " <<
    multOp->get_rhs_operand()->sage_class_name() << std::endl;
                    SgExpression* leftMultOp = multOp->get_lhs_operand();
196                 SgExpression* rightMultOp = multOp->get_rhs_operand();
198
200                 Rose_STL_Container<SgNode*> sizeOfs = NodeQuery::querySubTree(multOp, V_SgSizeOfOp);
                    Rose_STL_Container<SgNode*> ints = NodeQuery::querySubTree(multOp, V_SgIntVal);
202
                    if(sizeOfs.size() != 1 || ints.size() == 0)
204                 {
                        std::cout << "sizeofs: " << boost::lexical_cast<std::string>(sizeOfs.size()) << " ints: " <<
    boost::lexical_cast<std::string>(ints.size()) << std::endl;
206                     return false;
                    }
208
                    SgSizeOfOp* sizeOfOp = null;
210                 SgIntVal* intValOp = null;
212                 //getting size of operation used in rotation
                    sizeOfOp = isSgSizeOfOp(sizeOfs.at(0));
214
                    //getting integer multiplicator
216                 Rose_STL_Container<SgNode*>::iterator iter;
                    for(iter = ints.begin(); iter != ints.end(); iter++)
218                 {
                        SgIntVal* intNode = isSgIntVal(*iter);
220                     if(!isSgSizeOfOp(intNode->get_parent()))
                        {
222                         intValOp = intNode;
                        }
224                 }
226                 ROSE_ASSERT(sizeOfOp != null);
                    ROSE_ASSERT(intValOp != null);
228
                    if(intValOp->get_value() != 8)
230                     return false;
232                 //std::cout << "found sizeOfOp " << sizeOfOp->unparseToCompleteString() << " and intValOp " <<
    intValOp->unparseToCompleteString() << std::endl;
234                 if(sizeOfOp->get_operand_expr() != shiftedExpression)
                    {
236                     if(sizeOfOp->get_operand_expr()->unparseToCompleteString().compare(shiftedExpression->
    unparseToCompleteString()) != 0)
                        {
238                         return false;
                        }
240                 }
            }
242
            return true;
244       }
    }
246
    return false;
248 }
```

5.4.4 Splitting

In this step, operation isolation is ensured for the statements in the function. Therefore, statements, expressions and operations in the source code are split. Splitting names the process of extracting minimal parts of an expression into an own expression to generate multiple small statements from a potentially large one. So the statement

$$\texttt{declare } x = (a \oplus b \oplus c) \tag{5.6}$$

is split into small declarations with one operation per line resulting in

$$\texttt{declare x, delcare temp} \tag{5.7}$$

$$temp1 = a \oplus b \tag{5.8}$$

$$x = temp1 \oplus c \tag{5.9}$$

ROSE supports splitting of parts of expressions in its `SageInterface` component by calling the method `splitExpression`. It takes an expression and extracts it from the containing statements, determines its type and moves it to an own temporary variable. The framework searches for all expressions and binary operations at a certain level of the AST and splits them in order to establish the standard format required.

Code Example 5.7 shows the method used to detect elements to split and perform the splitting. It first obtains all expressions in the target function and detects if they are relevant to splitting by filtering it via the `querySplittableExpressionsSolver` method shown in Code Example 5.8. Further, it splits all those expressions by calling the `split` operation.

After the expressions have been split, the method queries for all binary operations relevant to splitting via the `querySplittableOperationSolver` callback method and splits those elements as well. Note that this is performed in a while loop, since the operation might have to be applied in several steps to ensure correct splitting of complex expression.

Code Example 5.7: Code detecting operations and expressions to split and performing the splitting using the `split` method.

```
void TranslationUtils::splitOperationsInFunction(SgFunctionDeclaration* function)
{
    //getting expressions to split (like functions etc.)
    Rose_STL_Container<SgNode*> expressions = NodeQuery::querySubTree(function, V_SgExpression);
    Rose_STL_Container<SgNode*> splitExpressions = NodeQuery::queryNodeList(expressions, &
      querySplittableExpressionsSolver);

    //splitting expressions
    split(splitExpressions, "exprTemp");

    //splitting operations until no slittable element can be found any more
    int splitContainerSize = 0;

    //needed for temp variable naming
    int transformationRound = 0;

    do
    {
        //getting operations to split (XOR, AND, etc.)
        Rose_STL_Container<SgNode*> operations = NodeQuery::querySubTree(function, V_SgBinaryOp);
```

```
20          Rose_STL_Container<SgNode*> splitOperations = NodeQuery::queryNodeList(operations, &
        querySplittableOperationSolver);

22          //getting size of operations to split as break condition variable
            splitContainerSize = splitOperations.size();
24
            //splitting operations
26          split(splitOperations, "binaryOpTemp" + boost::lexical_cast<std::string>(transformationRound));
            transformationRound++;
28
        } while (splitContainerSize != 0);
30  }

32
    void TranslationUtils::split(Rose_STL_Container<SgNode*> toSplit, std::string tempVarName)
34  {
        //checking size of stuff to split
36      if(toSplit.size() == 0)
            std::cout << "Nothing to split" << std::endl;
38
        //resetting counter for temp variables
40      int tempVariableCounter = 0;

42      //iterating over elements and splitting
        Rose_STL_Container<SgNode*>::iterator iter;
44      for (iter = toSplit.begin(); iter != toSplit.end(); iter++)
        {
46          //getting operation
            SgExpression* op = isSgExpression(*iter);
48          ROSE_ASSERT(op != null);

50          //creating unique name
            tempVariableCounter = tempVariableCounter + 1;
52          std::cout << "splitting: " << op->unparseToString() << std::endl;

54          //splitting expression
            std::string tempVariableName = tempVarName + boost::lexical_cast<std::string>(tempVariableCounter);
56          SageInterface::splitExpression(op, tempVariableName);
        }
58  }
```

Code Example 5.8 shows the callback methods used to detect expressions relevant for splitting. The method `querySplittableExpressionsSolver` is executed by a NodeQuery to identify expressions relevant for splitting. The method only classifies an expression as valid for splitting if :

- The nodes sub-AST has a depth greater than zero. This indicates that it's a non-trivial operation with operands.
- The node is no dereference operation (e.g. `S[4]`).
- The node is no binary operation. Binary operations will be split separately.
- There are no binary expressions except for pointer dereference operation in the sub-AST. These will be split later and automatically split the expression itself.

The method checks for these conditions and adds the node to the results (to be split).

The method `querySplittableOperationSolver` detects operations relevant for splitting. It ignores annotated operations (e.g. rotations) since they represent one operation and are not meant to be split. The method now searches for the binary operations on the lowest level of the node's sub-AST. For example, in the operation

$$x = (a \oplus b) \wedge (a \oplus c) \tag{5.10}$$

the method searches for the expressions $a \oplus b$ and $a \oplus c$ rather than for the $\wedge$ operation.

These operations show a small sub-tree depth, since their children are either values or pointer dereference operations (arrays etc.). Therefore, such expressions can't be assign operations and their parent must not be an assign operator or initializer. If their parent is an assign operator or initializer, then the operation is either not the one on the lowest AST level, or the line is already in standard form. To determine if the operation is on the lowest AST level, it checks the depth of the subtree in combination with information on if array dereferences occur in the sub tree . These dereference operations have a depth of one. Therefore, the target operation is allowed to have a subtree depth of one if no dereferences are present, and two if there are.

The method filters all nodes not matching this criteria and returns it to the calling component.

Code Example 5.8: Callback methods for detecting expressions and operations relevant to splitting.

```
NodeQuerySynthesizedAttributeType TranslationUtils::querySplittableExpressionsSolver(SgNode* astNode)
{
    //node must not be null
    ROSE_ASSERT(astNode != 0);
    NodeQuerySynthesizedAttributeType returnType;

    //iterating over parents to find annotation that would not allow splitting
    bool isAnnotated = isSystemAnnotatedInParents(astNode);
    if(isAnnotated)
    {
        //std::cout << "Skipping statement" << astNode->unparseToCompleteString() <<" due to annotation" << std::
      endl;
        return returnType;
    }
    else
    {
        //std::cout << "Found no annotation for node" << std::endl;
    }


    //node has to be operation, filter this in upper query
    SgExpression* exp = isSgExpression(astNode);
    ROSE_ASSERT(exp != null);

    //getting depth of subtree from that ast node
    int depth = exp->depthOfSubtree();

    if(depth >= 1 && !isSgPntrArrRefExp(astNode) && !isSgBinaryOp(astNode))
    {
        //std::cout << "Found relevant expression: " << astNode->unparseToCompleteString() << " with depth " <<
      boost::lexical_cast<std::string>(depth) << std::endl;

        Rose_STL_Container<SgNode*> subBinaryExpressions = NodeQuery::querySubTree(astNode, V_SgBinaryOp);
        Rose_STL_Container<SgNode*> subPntrRefExpressions = NodeQuery::querySubTree(astNode, V_SgPntrArrRefExp);
        //std::cout << "Found binary ops in subtree " << boost::lexical_cast<std::string>(subBinaryExpressions.size
      ()) << std::endl;
        if(subBinaryExpressions.size() == 0 || subBinaryExpressions.size() == subPntrRefExpressions.size())
        {
            std::cout << "No binary operations in subtree...splitting" << std::endl;
            returnType.push_back(astNode);
        }
    }

    return returnType;

}

NodeQuerySynthesizedAttributeType TranslationUtils::querySplittableOperationSolver(SgNode* astNode)
{
    //node must not be null
    ROSE_ASSERT(astNode != 0);
    NodeQuerySynthesizedAttributeType returnType;

    //iterating over parents to find annotation that would not allow splitting
    bool isAnnotated = isSystemAnnotatedInParents(astNode);
    if(isAnnotated)
```

```
54      {
            //std::cout << "Skipping statement" << astNode->unparseToCompleteString() <<" due to annotation" << std::
        endl;
56          return returnType;
        }
58      else
        {
60          //std::cout << "Found no annotation for node" << std::endl;
        }
62

64      //node has to be operation, filter this in upper query
        SgBinaryOp* op = isSgBinaryOp(astNode);
66      ROSE_ASSERT(op != null);
        SgPntrArrRefExp* ptnrArrayRefExpr = isSgPntrArrRefExp(astNode);
68
        if(!isSgAssignOp(astNode) && !isSgAssignOp(astNode->get_parent()) && !isSgAssignInitializer(astNode->get_parent
          ()))
70      {
            //getting depth of subtree from that ast node
72          int depth = op->depthOfSubtree();

74          //if binary operation, depth is one and not an arrayref: we have the canonical operation we wanna split
            if(depth == 1 && ptnrArrayRefExpr == null)
76          {
                //std::cout << "Found operation with depth 1: " << op->unparseToString() << std::endl;
78              returnType.push_back(astNode);
            }
80          //if depth is two, it could be that the second ast tree level is caused by array refs e.g. S[4], therefore
          filter those
            else if(depth == 2)
82          {
                //std::cout << "Found operation with depth 2: " << op->unparseToString() << " checking if second level
          caused by array ref" << std::endl;
84              Rose_STL_Container<SgNode*> arrayPointerRefs = NodeQuery::querySubTree(astNode, V_SgPntrArrRefExp);
                if(arrayPointerRefs.size() != 0)
86              {
                    //std::cout << "Second level caused by array ref, we can split this" << std::endl;
88                  returnType.push_back(astNode);
                }
90          }
            else
92          {
                std::cout << "Found operation " << astNode->unparseToCompleteString() << " with depth " <<
94                      boost::lexical_cast<std::string>(astNode->depthOfSubtree()) << " and type " << astNode->
          sage_class_name() << " not splitting this" << std::endl;
            }
96      }
        else
98      {
            //std::cout << "Found expression: " << astNode->unparseToCompleteString() << " as assign on child of assign
          " << std::endl;
100     }

102     return returnType;
    }
```

The `split` method, shown in Code Example 5.7, performs the actual splitting of expressions. It takes a list of elements to split as well as a string defining the name of the temporary variables needed to outline elements. Since various of those temporary variables are required, it holds an iteration count and appends the counter to the passed string for naming.

The final operation of splitting transforms the input code to the standard format needed for transformation. The IR of the source function has now been normalized such that the conversion to other representation formats can take place in the adapters, which was the goal of these transformations.

5.4.5 Example

This section shows the various transformation steps applied to the implementation of NORX. Code Example 5.9 shows the untransformed round function. It can be seen that it heavily uses macros to perform the core operations (G, U).

Code Example 5.10 shows the round function after the inlining, it can be seen that all macro calls have been inlined into the round function. It does therefore not depend on any external code anymore.

Code Example 5.11 shows the same round function after the compound statements have been fixed. It can be seen that they have been substituted by their *canonical* expression.

Code Example 5.12 shows the round function after the rotations have been detected and annotated. It can be seen that the rotations have not been altered, but their type and parameters have been detected and an annotation has been added to their comments. This annotation will be detected by further steps and indicates treatment as one operation defined by the annotation.

Code Example 5.13 shows the round function after splitting. All operations have been split. This represents the round function in standard form that can be used for further processing and transformation.

Code Example 5.9: The NORX round function before any transformation.

```
/* The nonlinear primitive */
#define U(A, B) ( ( (A) ^ (B) ) ^ ( ( (A) & (B) ) << 1) )

/* The quarter-round */
#define G(A, B, C, D) \
do \
{ \
  (A) = U(A, B); (D) ^= (A); (D) = ROTR((D), R0); \
  (C) = U(C, D); (B) ^= (C); (B) = ROTR((B), R1); \
  (A) = U(A, B); (D) ^= (A); (D) = ROTR((D), R2); \
  (C) = U(C, D); (B) ^= (C); (B) = ROTR((B), R3); \
} while (0)

//@roundfunction
/* The full round */
static NORX_INLINE void F(norx_word_t S[16])
{
    /* Column step */
    G(S[ 0], S[ 4], S[ 8], S[12]);
    G(S[ 1], S[ 5], S[ 9], S[13]);
    G(S[ 2], S[ 6], S[10], S[14]);
    G(S[ 3], S[ 7], S[11], S[15]);
    /* Diagonal step */
    G(S[ 0], S[ 5], S[10], S[15]);
    G(S[ 1], S[ 6], S[11], S[12]);
    G(S[ 2], S[ 7], S[ 8], S[13]);
    G(S[ 3], S[ 4], S[ 9], S[14]);
}
```

Code Example 5.10: The NORX round after the inlining transformation was applied.

```
//@roundfunction
/* The full round */

inline static void F(norx_word_t S[16UL])
{
/* Column step */
  do {
    S[0] = ((S[0] ^ S[4]) ^ ((S[0] & S[4]) << 1));
```

```
    S[12] ^= S[0];
    S[12] = ((S[12] >> 8) | (S[12] << sizeof(S[12]) * 8 - 8));
    S[8] = ((S[8] ^ S[12]) ^ ((S[8] & S[12]) << 1));
    S[4] ^= S[8];
    S[4] = ((S[4] >> 19) | (S[4] << sizeof(S[4]) * 8 - 19));
    S[0] = ((S[0] ^ S[4]) ^ ((S[0] & S[4]) << 1));
    S[12] ^= S[0];
    S[12] = ((S[12] >> 40) | (S[12] << sizeof(S[12]) * 8 - 40));
    S[8] = ((S[8] ^ S[12]) ^ ((S[8] & S[12]) << 1));
    S[4] ^= S[8];
    S[4] = ((S[4] >> 63) | (S[4] << sizeof(S[4]) * 8 - 63));
  }while (0);
  do {
    S[1] = ((S[1] ^ S[5]) ^ ((S[1] & S[5]) << 1));
    S[13] ^= S[1];
    S[13] = ((S[13] >> 8) | (S[13] << sizeof(S[13]) * 8 - 8));
    S[9] = ((S[9] ^ S[13]) ^ ((S[9] & S[13]) << 1));
    S[5] ^= S[9];
    S[5] = ((S[5] >> 19) | (S[5] << sizeof(S[5]) * 8 - 19));
    S[1] = ((S[1] ^ S[5]) ^ ((S[1] & S[5]) << 1));
    S[13] ^= S[1];
    S[13] = ((S[13] >> 40) | (S[13] << sizeof(S[13]) * 8 - 40));
    S[9] = ((S[9] ^ S[13]) ^ ((S[9] & S[13]) << 1));
    S[5] ^= S[9];
    S[5] = ((S[5] >> 63) | (S[5] << sizeof(S[5]) * 8 - 63));
  }while (0);
  do {
    S[2] = ((S[2] ^ S[6]) ^ ((S[2] & S[6]) << 1));
    S[14] ^= S[2];
    S[14] = ((S[14] >> 8) | (S[14] << sizeof(S[14]) * 8 - 8));
    S[10] = ((S[10] ^ S[14]) ^ ((S[10] & S[14]) << 1));
    S[6] ^= S[10];
    S[6] = ((S[6] >> 19) | (S[6] << sizeof(S[6]) * 8 - 19));
    S[2] = ((S[2] ^ S[6]) ^ ((S[2] & S[6]) << 1));
    S[14] ^= S[2];
    S[14] = ((S[14] >> 40) | (S[14] << sizeof(S[14]) * 8 - 40));
    S[10] = ((S[10] ^ S[14]) ^ ((S[10] & S[14]) << 1));
    S[6] ^= S[10];
    S[6] = ((S[6] >> 63) | (S[6] << sizeof(S[6]) * 8 - 63));
  }while (0);
  do {
    S[3] = ((S[3] ^ S[7]) ^ ((S[3] & S[7]) << 1));
    S[15] ^= S[3];
    S[15] = ((S[15] >> 8) | (S[15] << sizeof(S[15]) * 8 - 8));
    S[11] = ((S[11] ^ S[15]) ^ ((S[11] & S[15]) << 1));
    S[7] ^= S[11];
    S[7] = ((S[7] >> 19) | (S[7] << sizeof(S[7]) * 8 - 19));
    S[3] = ((S[3] ^ S[7]) ^ ((S[3] & S[7]) << 1));
    S[15] ^= S[3];
    S[15] = ((S[15] >> 40) | (S[15] << sizeof(S[15]) * 8 - 40));
    S[11] = ((S[11] ^ S[15]) ^ ((S[11] & S[15]) << 1));
    S[7] ^= S[11];
    S[7] = ((S[7] >> 63) | (S[7] << sizeof(S[7]) * 8 - 63));
  }while (0);
/* Diagonal step */
  do {
    S[0] = ((S[0] ^ S[5]) ^ ((S[0] & S[5]) << 1));
    S[15] ^= S[0];
    S[15] = ((S[15] >> 8) | (S[15] << sizeof(S[15]) * 8 - 8));
    S[10] = ((S[10] ^ S[15]) ^ ((S[10] & S[15]) << 1));
    S[5] ^= S[10];
    S[5] = ((S[5] >> 19) | (S[5] << sizeof(S[5]) * 8 - 19));
    S[0] = ((S[0] ^ S[5]) ^ ((S[0] & S[5]) << 1));
    S[15] ^= S[0];
    S[15] = ((S[15] >> 40) | (S[15] << sizeof(S[15]) * 8 - 40));
    S[10] = ((S[10] ^ S[15]) ^ ((S[10] & S[15]) << 1));
    S[5] ^= S[10];
    S[5] = ((S[5] >> 63) | (S[5] << sizeof(S[5]) * 8 - 63));
  }while (0);
  do {
    S[1] = ((S[1] ^ S[6]) ^ ((S[1] & S[6]) << 1));
    S[12] ^= S[1];
    S[12] = ((S[12] >> 8) | (S[12] << sizeof(S[12]) * 8 - 8));
    S[11] = ((S[11] ^ S[12]) ^ ((S[11] & S[12]) << 1));
    S[6] ^= S[11];
    S[6] = ((S[6] >> 19) | (S[6] << sizeof(S[6]) * 8 - 19));
    S[1] = ((S[1] ^ S[6]) ^ ((S[1] & S[6]) << 1));
    S[12] ^= S[1];
    S[12] = ((S[12] >> 40) | (S[12] << sizeof(S[12]) * 8 - 40));
    S[11] = ((S[11] ^ S[12]) ^ ((S[11] & S[12]) << 1));
    S[6] ^= S[11];
    S[6] = ((S[6] >> 63) | (S[6] << sizeof(S[6]) * 8 - 63));
  }while (0);
```

```
  do {
    S[2] = ((S[2] ^ S[7]) ^ ((S[2] & S[7]) << 1));
    S[13] ^= S[2];
    S[13] = ((S[13] >> 8) | (S[13] << sizeof(S[13]) * 8 - 8));
    S[8] = ((S[8] ^ S[13]) ^ ((S[8] & S[13]) << 1));
    S[7] ^= S[8];
    S[7] = ((S[7] >> 19) | (S[7] << sizeof(S[7]) * 8 - 19));
    S[2] = ((S[2] ^ S[7]) ^ ((S[2] & S[7]) << 1));
    S[13] ^= S[2];
    S[13] = ((S[13] >> 40) | (S[13] << sizeof(S[13]) * 8 - 40));
    S[8] = ((S[8] ^ S[13]) ^ ((S[8] & S[13]) << 1));
    S[7] ^= S[8];
    S[7] = ((S[7] >> 63) | (S[7] << sizeof(S[7]) * 8 - 63));
  }while (0);
  do {
    S[3] = ((S[3] ^ S[4]) ^ ((S[3] & S[4]) << 1));
    S[14] ^= S[3];
    S[14] = ((S[14] >> 8) | (S[14] << sizeof(S[14]) * 8 - 8));
    S[9] = ((S[9] ^ S[14]) ^ ((S[9] & S[14]) << 1));
    S[4] ^= S[9];
    S[4] = ((S[4] >> 19) | (S[4] << sizeof(S[4]) * 8 - 19));
    S[3] = ((S[3] ^ S[4]) ^ ((S[3] & S[4]) << 1));
    S[14] ^= S[3];
    S[14] = ((S[14] >> 40) | (S[14] << sizeof(S[14]) * 8 - 40));
    S[9] = ((S[9] ^ S[14]) ^ ((S[9] & S[14]) << 1));
    S[4] ^= S[9];
    S[4] = ((S[4] >> 63) | (S[4] << sizeof(S[4]) * 8 - 63));
  }while (0);
}
```

Code Example 5.11: The NORX round after inlining and the handling of compound expressions.

```
//@roundfunction
/* The full round */

inline static void F(norx_word_t S[16UL])
{
/* Column step */
  do {
    S[0] = ((S[0] ^ S[4]) ^ ((S[0] & S[4]) << 1));
    S[12] = S[12] ^ S[0];
    S[12] = ((S[12] >> 8) | (S[12] << sizeof(S[12]) * 8 - 8));
    S[8] = ((S[8] ^ S[12]) ^ ((S[8] & S[12]) << 1));
    S[4] = S[4] ^ S[8];
    S[4] = ((S[4] >> 19) | (S[4] << sizeof(S[4]) * 8 - 19));
    S[0] = ((S[0] ^ S[4]) ^ ((S[0] & S[4]) << 1));
    S[12] = S[12] ^ S[0];
    S[12] = ((S[12] >> 40) | (S[12] << sizeof(S[12]) * 8 - 40));
    S[8] = ((S[8] ^ S[12]) ^ ((S[8] & S[12]) << 1));
    S[4] = S[4] ^ S[8];
    S[4] = ((S[4] >> 63) | (S[4] << sizeof(S[4]) * 8 - 63));
  }while (0);
  do {
    S[1] = ((S[1] ^ S[5]) ^ ((S[1] & S[5]) << 1));
    S[13] = S[13] ^ S[1];
    S[13] = ((S[13] >> 8) | (S[13] << sizeof(S[13]) * 8 - 8));
    S[9] = ((S[9] ^ S[13]) ^ ((S[9] & S[13]) << 1));
    S[5] = S[5] ^ S[9];
    S[5] = ((S[5] >> 19) | (S[5] << sizeof(S[5]) * 8 - 19));
    S[1] = ((S[1] ^ S[5]) ^ ((S[1] & S[5]) << 1));
    S[13] = S[13] ^ S[1];
    S[13] = ((S[13] >> 40) | (S[13] << sizeof(S[13]) * 8 - 40));
    S[9] = ((S[9] ^ S[13]) ^ ((S[9] & S[13]) << 1));
    S[5] = S[5] ^ S[9];
    S[5] = ((S[5] >> 63) | (S[5] << sizeof(S[5]) * 8 - 63));
  }while (0);
  do {
    S[2] = ((S[2] ^ S[6]) ^ ((S[2] & S[6]) << 1));
    S[14] = S[14] ^ S[2];
    S[14] = ((S[14] >> 8) | (S[14] << sizeof(S[14]) * 8 - 8));
    S[10] = ((S[10] ^ S[14]) ^ ((S[10] & S[14]) << 1));
    S[6] = S[6] ^ S[10];
    S[6] = ((S[6] >> 19) | (S[6] << sizeof(S[6]) * 8 - 19));
    S[2] = ((S[2] ^ S[6]) ^ ((S[2] & S[6]) << 1));
    S[14] = S[14] ^ S[2];
    S[14] = ((S[14] >> 40) | (S[14] << sizeof(S[14]) * 8 - 40));
    S[10] = ((S[10] ^ S[14]) ^ ((S[10] & S[14]) << 1));
    S[6] = S[6] ^ S[10];
    S[6] = ((S[6] >> 63) | (S[6] << sizeof(S[6]) * 8 - 63));
  }while (0);
```

```
  do {
    S[3] = ((S[3] ^ S[7]) ^ ((S[3] & S[7]) << 1));
    S[15] = S[15] ^ S[3];
    S[15] = ((S[15] >> 8) | (S[15] << sizeof(S[15]) * 8 - 8));
    S[11] = ((S[11] ^ S[15]) ^ ((S[11] & S[15]) << 1));
    S[7] = S[7] ^ S[11];
    S[7] = ((S[7] >> 19) | (S[7] << sizeof(S[7]) * 8 - 19));
    S[3] = ((S[3] ^ S[7]) ^ ((S[3] & S[7]) << 1));
    S[15] = S[15] ^ S[3];
    S[15] = ((S[15] >> 40) | (S[15] << sizeof(S[15]) * 8 - 40));
    S[11] = ((S[11] ^ S[15]) ^ ((S[11] & S[15]) << 1));
    S[7] = S[7] ^ S[11];
    S[7] = ((S[7] >> 63) | (S[7] << sizeof(S[7]) * 8 - 63));
  }while (0);
/* Diagonal step */
  do {
    S[0] = ((S[0] ^ S[5]) ^ ((S[0] & S[5]) << 1));
    S[15] = S[15] ^ S[0];
    S[15] = ((S[15] >> 8) | (S[15] << sizeof(S[15]) * 8 - 8));
    S[10] = ((S[10] ^ S[15]) ^ ((S[10] & S[15]) << 1));
    S[5] = S[5] ^ S[10];
    S[5] = ((S[5] >> 19) | (S[5] << sizeof(S[5]) * 8 - 19));
    S[0] = ((S[0] ^ S[5]) ^ ((S[0] & S[5]) << 1));
    S[15] = S[15] ^ S[0];
    S[15] = ((S[15] >> 40) | (S[15] << sizeof(S[15]) * 8 - 40));
    S[10] = ((S[10] ^ S[15]) ^ ((S[10] & S[15]) << 1));
    S[5] = S[5] ^ S[10];
    S[5] = ((S[5] >> 63) | (S[5] << sizeof(S[5]) * 8 - 63));
  }while (0);
  do {
    S[1] = ((S[1] ^ S[6]) ^ ((S[1] & S[6]) << 1));
    S[12] = S[12] ^ S[1];
    S[12] = ((S[12] >> 8) | (S[12] << sizeof(S[12]) * 8 - 8));
    S[11] = ((S[11] ^ S[12]) ^ ((S[11] & S[12]) << 1));
    S[6] = S[6] ^ S[11];
    S[6] = ((S[6] >> 19) | (S[6] << sizeof(S[6]) * 8 - 19));
    S[1] = ((S[1] ^ S[6]) ^ ((S[1] & S[6]) << 1));
    S[12] = S[12] ^ S[1];
    S[12] = ((S[12] >> 40) | (S[12] << sizeof(S[12]) * 8 - 40));
    S[11] = ((S[11] ^ S[12]) ^ ((S[11] & S[12]) << 1));
    S[6] = S[6] ^ S[11];
    S[6] = ((S[6] >> 63) | (S[6] << sizeof(S[6]) * 8 - 63));
  }while (0);
  do {
    S[2] = ((S[2] ^ S[7]) ^ ((S[2] & S[7]) << 1));
    S[13] = S[13] ^ S[2];
    S[13] = ((S[13] >> 8) | (S[13] << sizeof(S[13]) * 8 - 8));
    S[8] = ((S[8] ^ S[13]) ^ ((S[8] & S[13]) << 1));
    S[7] = S[7] ^ S[8];
    S[7] = ((S[7] >> 19) | (S[7] << sizeof(S[7]) * 8 - 19));
    S[2] = ((S[2] ^ S[7]) ^ ((S[2] & S[7]) << 1));
    S[13] = S[13] ^ S[2];
    S[13] = ((S[13] >> 40) | (S[13] << sizeof(S[13]) * 8 - 40));
    S[8] = ((S[8] ^ S[13]) ^ ((S[8] & S[13]) << 1));
    S[7] = S[7] ^ S[8];
    S[7] = ((S[7] >> 63) | (S[7] << sizeof(S[7]) * 8 - 63));
  }while (0);
  do {
    S[3] = ((S[3] ^ S[4]) ^ ((S[3] & S[4]) << 1));
    S[14] = S[14] ^ S[3];
    S[14] = ((S[14] >> 8) | (S[14] << sizeof(S[14]) * 8 - 8));
    S[9] = ((S[9] ^ S[14]) ^ ((S[9] & S[14]) << 1));
    S[4] = S[4] ^ S[9];
    S[4] = ((S[4] >> 19) | (S[4] << sizeof(S[4]) * 8 - 19));
    S[3] = ((S[3] ^ S[4]) ^ ((S[3] & S[4]) << 1));
    S[14] = S[14] ^ S[3];
    S[14] = ((S[14] >> 40) | (S[14] << sizeof(S[14]) * 8 - 40));
    S[9] = ((S[9] ^ S[14]) ^ ((S[9] & S[14]) << 1));
    S[4] = S[4] ^ S[9];
    S[4] = ((S[4] >> 63) | (S[4] << sizeof(S[4]) * 8 - 63));
  }while (0);
}
```

Code Example 5.12: The NORX round after inlining, compound expression handling and rotation detection.

```
//@roundfunction
/* The full round */

inline static void F(norx_word_t S[16UL])
{
```

```
/* Column step */
  do {
    S[0] = ((S[0] ^ S[4]) ^ ((S[0] & S[4]) << 1));
    S[12] = S[12] ^ S[0];
    S[12] =
/* @compoundOperation */
/* @ROTR(S[12], 8) */
((S[12] >> 8) | (S[12] << sizeof(S[12]) * 8 - 8));
    S[8] = ((S[8] ^ S[12]) ^ ((S[8] & S[12]) << 1));
    S[4] = S[4] ^ S[8];
    S[4] =
/* @compoundOperation */
/* @ROTR(S[4], 19) */
((S[4] >> 19) | (S[4] << sizeof(S[4]) * 8 - 19));
    S[0] = ((S[0] ^ S[4]) ^ ((S[0] & S[4]) << 1));
    S[12] = S[12] ^ S[0];
    S[12] =
/* @compoundOperation */
/* @ROTR(S[12], 40) */
((S[12] >> 40) | (S[12] << sizeof(S[12]) * 8 - 40));
    S[8] = ((S[8] ^ S[12]) ^ ((S[8] & S[12]) << 1));
    S[4] = S[4] ^ S[8];
    S[4] =
/* @compoundOperation */
/* @ROTR(S[4], 63) */
((S[4] >> 63) | (S[4] << sizeof(S[4]) * 8 - 63));
  }while (0);
  do {
    S[1] = ((S[1] ^ S[5]) ^ ((S[1] & S[5]) << 1));
    S[13] = S[13] ^ S[1];
    S[13] =
/* @compoundOperation */
/* @ROTR(S[13], 8) */
((S[13] >> 8) | (S[13] << sizeof(S[13]) * 8 - 8));
    S[9] = ((S[9] ^ S[13]) ^ ((S[9] & S[13]) << 1));
    S[5] = S[5] ^ S[9];
    S[5] =
/* @compoundOperation */
/* @ROTR(S[5], 19) */
((S[5] >> 19) | (S[5] << sizeof(S[5]) * 8 - 19));
    S[1] = ((S[1] ^ S[5]) ^ ((S[1] & S[5]) << 1));
    S[13] = S[13] ^ S[1];
    S[13] =
/* @compoundOperation */
/* @ROTR(S[13], 40) */
((S[13] >> 40) | (S[13] << sizeof(S[13]) * 8 - 40));
    S[9] = ((S[9] ^ S[13]) ^ ((S[9] & S[13]) << 1));
    S[5] = S[5] ^ S[9];
    S[5] =
/* @compoundOperation */
/* @ROTR(S[5], 63) */
((S[5] >> 63) | (S[5] << sizeof(S[5]) * 8 - 63));
  }while (0);
  do {
    S[2] = ((S[2] ^ S[6]) ^ ((S[2] & S[6]) << 1));
    S[14] = S[14] ^ S[2];
    S[14] =
/* @compoundOperation */
/* @ROTR(S[14], 8) */
((S[14] >> 8) | (S[14] << sizeof(S[14]) * 8 - 8));
    S[10] = ((S[10] ^ S[14]) ^ ((S[10] & S[14]) << 1));
    S[6] = S[6] ^ S[10];
    S[6] =
/* @compoundOperation */
/* @ROTR(S[6], 19) */
((S[6] >> 19) | (S[6] << sizeof(S[6]) * 8 - 19));
    S[2] = ((S[2] ^ S[6]) ^ ((S[2] & S[6]) << 1));
    S[14] = S[14] ^ S[2];
    S[14] =
/* @compoundOperation */
/* @ROTR(S[14], 40) */
((S[14] >> 40) | (S[14] << sizeof(S[14]) * 8 - 40));
    S[10] = ((S[10] ^ S[14]) ^ ((S[10] & S[14]) << 1));
    S[6] = S[6] ^ S[10];
    S[6] =
/* @compoundOperation */
/* @ROTR(S[6], 63) */
((S[6] >> 63) | (S[6] << sizeof(S[6]) * 8 - 63));
  }while (0);
  do {
    S[3] = ((S[3] ^ S[7]) ^ ((S[3] & S[7]) << 1));
    S[15] = S[15] ^ S[3];
    S[15] =
```

```
/* @compoundOperation */
/* @ROTR(S[15], 8) */
((S[15] >> 8) | (S[15] << sizeof(S[15]) * 8 - 8));
    S[11] = ((S[11] ^ S[15]) ^ ((S[11] & S[15]) << 1));
    S[7] = S[7] ^ S[11];
    S[7] =
/* @compoundOperation */
/* @ROTR(S[7], 19) */
((S[7] >> 19) | (S[7] << sizeof(S[7]) * 8 - 19));
    S[3] = ((S[3] ^ S[7]) ^ ((S[3] & S[7]) << 1));
    S[15] = S[15] ^ S[3];
    S[15] =
/* @compoundOperation */
/* @ROTR(S[15], 40) */
((S[15] >> 40) | (S[15] << sizeof(S[15]) * 8 - 40));
    S[11] = ((S[11] ^ S[15]) ^ ((S[11] & S[15]) << 1));
    S[7] = S[7] ^ S[11];
    S[7] =
/* @compoundOperation */
/* @ROTR(S[7], 63) */
((S[7] >> 63) | (S[7] << sizeof(S[7]) * 8 - 63));
  }while (0);
/* Diagonal step */
  do {
    S[0] = ((S[0] ^ S[5]) ^ ((S[0] & S[5]) << 1));
    S[15] = S[15] ^ S[0];
    S[15] =
/* @compoundOperation */
/* @ROTR(S[15], 8) */
((S[15] >> 8) | (S[15] << sizeof(S[15]) * 8 - 8));
    S[10] = ((S[10] ^ S[15]) ^ ((S[10] & S[15]) << 1));
    S[5] = S[5] ^ S[10];
    S[5] =
/* @compoundOperation */
/* @ROTR(S[5], 19) */
((S[5] >> 19) | (S[5] << sizeof(S[5]) * 8 - 19));
    S[0] = ((S[0] ^ S[5]) ^ ((S[0] & S[5]) << 1));
    S[15] = S[15] ^ S[0];
    S[15] =
/* @compoundOperation */
/* @ROTR(S[15], 40) */
((S[15] >> 40) | (S[15] << sizeof(S[15]) * 8 - 40));
    S[10] = ((S[10] ^ S[15]) ^ ((S[10] & S[15]) << 1));
    S[5] = S[5] ^ S[10];
    S[5] =
/* @compoundOperation */
/* @ROTR(S[5], 63) */
((S[5] >> 63) | (S[5] << sizeof(S[5]) * 8 - 63));
  }while (0);
  do {
    S[1] = ((S[1] ^ S[6]) ^ ((S[1] & S[6]) << 1));
    S[12] = S[12] ^ S[1];
    S[12] =
/* @compoundOperation */
/* @ROTR(S[12], 8) */
((S[12] >> 8) | (S[12] << sizeof(S[12]) * 8 - 8));
    S[11] = ((S[11] ^ S[12]) ^ ((S[11] & S[12]) << 1));
    S[6] = S[6] ^ S[11];
    S[6] =
/* @compoundOperation */
/* @ROTR(S[6], 19) */
((S[6] >> 19) | (S[6] << sizeof(S[6]) * 8 - 19));
    S[1] = ((S[1] ^ S[6]) ^ ((S[1] & S[6]) << 1));
    S[12] = S[12] ^ S[1];
    S[12] =
/* @compoundOperation */
/* @ROTR(S[12], 40) */
((S[12] >> 40) | (S[12] << sizeof(S[12]) * 8 - 40));
    S[11] = ((S[11] ^ S[12]) ^ ((S[11] & S[12]) << 1));
    S[6] = S[6] ^ S[11];
    S[6] =
/* @compoundOperation */
/* @ROTR(S[6], 63) */
((S[6] >> 63) | (S[6] << sizeof(S[6]) * 8 - 63));
  }while (0);
  do {
    S[2] = ((S[2] ^ S[7]) ^ ((S[2] & S[7]) << 1));
    S[13] = S[13] ^ S[2];
    S[13] =
/* @compoundOperation */
/* @ROTR(S[13], 8) */
((S[13] >> 8) | (S[13] << sizeof(S[13]) * 8 - 8));
    S[8] = ((S[8] ^ S[13]) ^ ((S[8] & S[13]) << 1));
```

```
172     S[7] = S[7] ^ S[8];
        S[7] =
174 /* @compoundOperation */
    /* @ROTR(S[7], 19) */
176 ((S[7] >> 19) | (S[7] << sizeof(S[7]) * 8 - 19));
        S[2] = ((S[2] ^ S[7]) ^ ((S[2] & S[7]) << 1));
178     S[13] = S[13] ^ S[2];
        S[13] =
180 /* @compoundOperation */
    /* @ROTR(S[13], 40) */
182 ((S[13] >> 40) | (S[13] << sizeof(S[13]) * 8 - 40));
        S[8] = ((S[8] ^ S[13]) ^ ((S[8] & S[13]) << 1));
184     S[7] = S[7] ^ S[8];
        S[7] =
186 /* @compoundOperation */
    /* @ROTR(S[7], 63) */
188 ((S[7] >> 63) | (S[7] << sizeof(S[7]) * 8 - 63));
      }while (0);
190   do {
        S[3] = ((S[3] ^ S[4]) ^ ((S[3] & S[4]) << 1));
192     S[14] = S[14] ^ S[3];
        S[14] =
194 /* @compoundOperation */
    /* @ROTR(S[14], 8) */
196 ((S[14] >> 8) | (S[14] << sizeof(S[14]) * 8 - 8));
        S[9] = ((S[9] ^ S[14]) ^ ((S[9] & S[14]) << 1));
198     S[4] = S[4] ^ S[9];
        S[4] =
200 /* @compoundOperation */
    /* @ROTR(S[4], 19) */
202 ((S[4] >> 19) | (S[4] << sizeof(S[4]) * 8 - 19));
        S[3] = ((S[3] ^ S[4]) ^ ((S[3] & S[4]) << 1));
204     S[14] = S[14] ^ S[3];
        S[14] =
206 /* @compoundOperation */
    /* @ROTR(S[14], 40) */
208 ((S[14] >> 40) | (S[14] << sizeof(S[14]) * 8 - 40));
        S[9] = ((S[9] ^ S[14]) ^ ((S[9] & S[14]) << 1));
210     S[4] = S[4] ^ S[9];
        S[4] =
212 /* @compoundOperation */
    /* @ROTR(S[4], 63) */
214 ((S[4] >> 63) | (S[4] << sizeof(S[4]) * 8 - 63));
      }while (0);
216 }
```

Code Example 5.13: The NORX round in standard form.

```
    //@roundfunction
2   /* The full round */

4   inline static void F(norx_word_t S[16UL])
    {
6   /* Column step */
      do {
8       norx_word_t binaryOpTemp01 = (S[0] ^ S[4]);
        norx_word_t binaryOpTemp02 = (S[0] & S[4]);
10      norx_word_t binaryOpTemp11 = (binaryOpTemp02 << 1);
        S[0] = (binaryOpTemp01 ^ binaryOpTemp11);
12      S[12] = S[12] ^ S[0];
        S[12] =
14  /* @compoundOperation */
    /* @ROTR(S[12], 8) */
16  ((S[12] >> 8) | (S[12] << sizeof(S[12]) * 8 - 8));
        norx_word_t binaryOpTemp03 = (S[8] ^ S[12]);
18      norx_word_t binaryOpTemp04 = (S[8] & S[12]);
        norx_word_t binaryOpTemp12 = (binaryOpTemp04 << 1);
20      S[8] = (binaryOpTemp03 ^ binaryOpTemp12);
        S[4] = S[4] ^ S[8];
22      S[4] =
    /* @compoundOperation */
24  /* @ROTR(S[4], 19) */
    ((S[4] >> 19) | (S[4] << sizeof(S[4]) * 8 - 19));
26      norx_word_t binaryOpTemp05 = (S[0] ^ S[4]);
        norx_word_t binaryOpTemp06 = (S[0] & S[4]);
28      norx_word_t binaryOpTemp13 = (binaryOpTemp06 << 1);
        S[0] = (binaryOpTemp05 ^ binaryOpTemp13);
30      S[12] = S[12] ^ S[0];
        S[12] =
32  /* @compoundOperation */
    /* @ROTR(S[12], 40) */
34  ((S[12] >> 40) | (S[12] << sizeof(S[12]) * 8 - 40));
```

```
    norx_word_t binaryOpTemp07 = (S[8] ^ S[12]);
    norx_word_t binaryOpTemp08 = (S[8] & S[12]);
    norx_word_t binaryOpTemp14 = (binaryOpTemp08 << 1);
    S[8] = (binaryOpTemp07 ^ binaryOpTemp14);
    S[4] = S[4] ^ S[8];
    S[4] =
/* @compoundOperation */
/* @ROTR(S[4], 63) */
((S[4] >> 63) | (S[4] << sizeof(S[4]) * 8 - 63));
  }while (0);
  do {
    norx_word_t binaryOpTemp09 = (S[1] ^ S[5]);
    norx_word_t binaryOpTemp010 = (S[1] & S[5]);
    norx_word_t binaryOpTemp15 = (binaryOpTemp010 << 1);
    S[1] = (binaryOpTemp09 ^ binaryOpTemp15);
    S[13] = S[13] ^ S[1];
    S[13] =
/* @compoundOperation */
/* @ROTR(S[13], 8) */
((S[13] >> 8) | (S[13] << sizeof(S[13]) * 8 - 8));
    norx_word_t binaryOpTemp011 = (S[9] ^ S[13]);
    norx_word_t binaryOpTemp012 = (S[9] & S[13]);
    norx_word_t binaryOpTemp16 = (binaryOpTemp012 << 1);
    S[9] = (binaryOpTemp011 ^ binaryOpTemp16);
    S[5] = S[5] ^ S[9];
    S[5] =
/* @compoundOperation */
/* @ROTR(S[5], 19) */
((S[5] >> 19) | (S[5] << sizeof(S[5]) * 8 - 19));
    norx_word_t binaryOpTemp013 = (S[1] ^ S[5]);
    norx_word_t binaryOpTemp014 = (S[1] & S[5]);
    norx_word_t binaryOpTemp17 = (binaryOpTemp014 << 1);
    S[1] = (binaryOpTemp013 ^ binaryOpTemp17);
    S[13] = S[13] ^ S[1];
    S[13] =
/* @compoundOperation */
/* @ROTR(S[13], 40) */
((S[13] >> 40) | (S[13] << sizeof(S[13]) * 8 - 40));
    norx_word_t binaryOpTemp015 = (S[9] ^ S[13]);
    norx_word_t binaryOpTemp016 = (S[9] & S[13]);
    norx_word_t binaryOpTemp18 = (binaryOpTemp016 << 1);
    S[9] = (binaryOpTemp015 ^ binaryOpTemp18);
    S[5] = S[5] ^ S[9];
    S[5] =
/* @compoundOperation */
/* @ROTR(S[5], 63) */
((S[5] >> 63) | (S[5] << sizeof(S[5]) * 8 - 63));
  }while (0);
  do {
    norx_word_t binaryOpTemp017 = (S[2] ^ S[6]);
    norx_word_t binaryOpTemp018 = (S[2] & S[6]);
    norx_word_t binaryOpTemp19 = (binaryOpTemp018 << 1);
    S[2] = (binaryOpTemp017 ^ binaryOpTemp19);
    S[14] = S[14] ^ S[2];
    S[14] =
/* @compoundOperation */
/* @ROTR(S[14], 8) */
((S[14] >> 8) | (S[14] << sizeof(S[14]) * 8 - 8));
    norx_word_t binaryOpTemp019 = (S[10] ^ S[14]);
    norx_word_t binaryOpTemp020 = (S[10] & S[14]);
    norx_word_t binaryOpTemp110 = (binaryOpTemp020 << 1);
    S[10] = (binaryOpTemp019 ^ binaryOpTemp110);
    S[6] = S[6] ^ S[10];
    S[6] =
/* @compoundOperation */
/* @ROTR(S[6], 19) */
((S[6] >> 19) | (S[6] << sizeof(S[6]) * 8 - 19));
    norx_word_t binaryOpTemp021 = (S[2] ^ S[6]);
    norx_word_t binaryOpTemp022 = (S[2] & S[6]);
    norx_word_t binaryOpTemp111 = (binaryOpTemp022 << 1);
    S[2] = (binaryOpTemp021 ^ binaryOpTemp111);
    S[14] = S[14] ^ S[2];
    S[14] =
/* @compoundOperation */
/* @ROTR(S[14], 40) */
((S[14] >> 40) | (S[14] << sizeof(S[14]) * 8 - 40));
    norx_word_t binaryOpTemp023 = (S[10] ^ S[14]);
    norx_word_t binaryOpTemp024 = (S[10] & S[14]);
    norx_word_t binaryOpTemp112 = (binaryOpTemp024 << 1);
    S[10] = (binaryOpTemp023 ^ binaryOpTemp112);
    S[6] = S[6] ^ S[10];
    S[6] =
/* @compoundOperation */
```

```
/* @ROTR(S[6], 63) */
((S[6] >> 63) | (S[6] << sizeof(S[6]) * 8 - 63));
  }while (0);
  do {
    norx_word_t binaryOpTemp025 = (S[3] ^ S[7]);
    norx_word_t binaryOpTemp026 = (S[3] & S[7]);
    norx_word_t binaryOpTemp113 = (binaryOpTemp026 << 1);
    S[3] = (binaryOpTemp025 ^ binaryOpTemp113);
    S[15] = S[15] ^ S[3];
    S[15] =
/* @compoundOperation */
/* @ROTR(S[15], 8) */
((S[15] >> 8) | (S[15] << sizeof(S[15]) * 8 - 8));
    norx_word_t binaryOpTemp027 = (S[11] ^ S[15]);
    norx_word_t binaryOpTemp028 = (S[11] & S[15]);
    norx_word_t binaryOpTemp114 = (binaryOpTemp028 << 1);
    S[11] = (binaryOpTemp027 ^ binaryOpTemp114);
    S[7] = S[7] ^ S[11];
    S[7] =
/* @compoundOperation */
/* @ROTR(S[7], 19) */
((S[7] >> 19) | (S[7] << sizeof(S[7]) * 8 - 19));
    norx_word_t binaryOpTemp029 = (S[3] ^ S[7]);
    norx_word_t binaryOpTemp030 = (S[3] & S[7]);
    norx_word_t binaryOpTemp115 = (binaryOpTemp030 << 1);
    S[3] = (binaryOpTemp029 ^ binaryOpTemp115);
    S[15] = S[15] ^ S[3];
    S[15] =
/* @compoundOperation */
/* @ROTR(S[15], 40) */
((S[15] >> 40) | (S[15] << sizeof(S[15]) * 8 - 40));
    norx_word_t binaryOpTemp031 = (S[11] ^ S[15]);
    norx_word_t binaryOpTemp032 = (S[11] & S[15]);
    norx_word_t binaryOpTemp116 = (binaryOpTemp032 << 1);
    S[11] = (binaryOpTemp031 ^ binaryOpTemp116);
    S[7] = S[7] ^ S[11];
    S[7] =
/* @compoundOperation */
/* @ROTR(S[7], 63) */
((S[7] >> 63) | (S[7] << sizeof(S[7]) * 8 - 63));
  }while (0);
/* Diagonal step */
  do {
    norx_word_t binaryOpTemp033 = (S[0] ^ S[5]);
    norx_word_t binaryOpTemp034 = (S[0] & S[5]);
    norx_word_t binaryOpTemp117 = (binaryOpTemp034 << 1);
    S[0] = (binaryOpTemp033 ^ binaryOpTemp117);
    S[15] = S[15] ^ S[0];
    S[15] =
/* @compoundOperation */
/* @ROTR(S[15], 8) */
((S[15] >> 8) | (S[15] << sizeof(S[15]) * 8 - 8));
    norx_word_t binaryOpTemp035 = (S[10] ^ S[15]);
    norx_word_t binaryOpTemp036 = (S[10] & S[15]);
    norx_word_t binaryOpTemp118 = (binaryOpTemp036 << 1);
    S[10] = (binaryOpTemp035 ^ binaryOpTemp118);
    S[5] = S[5] ^ S[10];
    S[5] =
/* @compoundOperation */
/* @ROTR(S[5], 19) */
((S[5] >> 19) | (S[5] << sizeof(S[5]) * 8 - 19));
    norx_word_t binaryOpTemp037 = (S[0] ^ S[5]);
    norx_word_t binaryOpTemp038 = (S[0] & S[5]);
    norx_word_t binaryOpTemp119 = (binaryOpTemp038 << 1);
    S[0] = (binaryOpTemp037 ^ binaryOpTemp119);
    S[15] = S[15] ^ S[0];
    S[15] =
/* @compoundOperation */
/* @ROTR(S[15], 40) */
((S[15] >> 40) | (S[15] << sizeof(S[15]) * 8 - 40));
    norx_word_t binaryOpTemp039 = (S[10] ^ S[15]);
    norx_word_t binaryOpTemp040 = (S[10] & S[15]);
    norx_word_t binaryOpTemp120 = (binaryOpTemp040 << 1);
    S[10] = (binaryOpTemp039 ^ binaryOpTemp120);
    S[5] = S[5] ^ S[10];
    S[5] =
/* @compoundOperation */
/* @ROTR(S[5], 63) */
((S[5] >> 63) | (S[5] << sizeof(S[5]) * 8 - 63));
  }while (0);
  do {
    norx_word_t binaryOpTemp041 = (S[1] ^ S[6]);
    norx_word_t binaryOpTemp042 = (S[1] & S[6]);
```

```
    norx_word_t binaryOpTemp121 = (binaryOpTemp042 << 1);
    S[1] = (binaryOpTemp041 ^ binaryOpTemp121);
    S[12] = S[12] ^ S[1];
    S[12] =
/* @compoundOperation */
/* @ROTR(S[12], 8) */
((S[12] >> 8) | (S[12] << sizeof(S[12]) * 8 - 8));
    norx_word_t binaryOpTemp043 = (S[11] ^ S[12]);
    norx_word_t binaryOpTemp044 = (S[11] & S[12]);
    norx_word_t binaryOpTemp122 = (binaryOpTemp044 << 1);
    S[11] = (binaryOpTemp043 ^ binaryOpTemp122);
    S[6] = S[6] ^ S[11];
    S[6] =
/* @compoundOperation */
/* @ROTR(S[6], 19) */
((S[6] >> 19) | (S[6] << sizeof(S[6]) * 8 - 19));
    norx_word_t binaryOpTemp045 = (S[1] ^ S[6]);
    norx_word_t binaryOpTemp046 = (S[1] & S[6]);
    norx_word_t binaryOpTemp123 = (binaryOpTemp046 << 1);
    S[1] = (binaryOpTemp045 ^ binaryOpTemp123);
    S[12] = S[12] ^ S[1];
    S[12] =
/* @compoundOperation */
/* @ROTR(S[12], 40) */
((S[12] >> 40) | (S[12] << sizeof(S[12]) * 8 - 40));
    norx_word_t binaryOpTemp047 = (S[11] ^ S[12]);
    norx_word_t binaryOpTemp048 = (S[11] & S[12]);
    norx_word_t binaryOpTemp124 = (binaryOpTemp048 << 1);
    S[11] = (binaryOpTemp047 ^ binaryOpTemp124);
    S[6] = S[6] ^ S[11];
    S[6] =
/* @compoundOperation */
/* @ROTR(S[6], 63) */
((S[6] >> 63) | (S[6] << sizeof(S[6]) * 8 - 63));
  }while (0);
  do {
    norx_word_t binaryOpTemp049 = (S[2] ^ S[7]);
    norx_word_t binaryOpTemp050 = (S[2] & S[7]);
    norx_word_t binaryOpTemp125 = (binaryOpTemp050 << 1);
    S[2] = (binaryOpTemp049 ^ binaryOpTemp125);
    S[13] = S[13] ^ S[2];
    S[13] =
/* @compoundOperation */
/* @ROTR(S[13], 8) */
((S[13] >> 8) | (S[13] << sizeof(S[13]) * 8 - 8));
    norx_word_t binaryOpTemp051 = (S[8] ^ S[13]);
    norx_word_t binaryOpTemp052 = (S[8] & S[13]);
    norx_word_t binaryOpTemp126 = (binaryOpTemp052 << 1);
    S[8] = (binaryOpTemp051 ^ binaryOpTemp126);
    S[7] = S[7] ^ S[8];
    S[7] =
/* @compoundOperation */
/* @ROTR(S[7], 19) */
((S[7] >> 19) | (S[7] << sizeof(S[7]) * 8 - 19));
    norx_word_t binaryOpTemp053 = (S[2] ^ S[7]);
    norx_word_t binaryOpTemp054 = (S[2] & S[7]);
    norx_word_t binaryOpTemp127 = (binaryOpTemp054 << 1);
    S[2] = (binaryOpTemp053 ^ binaryOpTemp127);
    S[13] = S[13] ^ S[2];
    S[13] =
/* @compoundOperation */
/* @ROTR(S[13], 40) */
((S[13] >> 40) | (S[13] << sizeof(S[13]) * 8 - 40));
    norx_word_t binaryOpTemp055 = (S[8] ^ S[13]);
    norx_word_t binaryOpTemp056 = (S[8] & S[13]);
    norx_word_t binaryOpTemp128 = (binaryOpTemp056 << 1);
    S[8] = (binaryOpTemp055 ^ binaryOpTemp128);
    S[7] = S[7] ^ S[8];
    S[7] =
/* @compoundOperation */
/* @ROTR(S[7], 63) */
((S[7] >> 63) | (S[7] << sizeof(S[7]) * 8 - 63));
  }while (0);
  do {
    norx_word_t binaryOpTemp057 = (S[3] ^ S[4]);
    norx_word_t binaryOpTemp058 = (S[3] & S[4]);
    norx_word_t binaryOpTemp129 = (binaryOpTemp058 << 1);
    S[3] = (binaryOpTemp057 ^ binaryOpTemp129);
    S[14] = S[14] ^ S[3];
    S[14] =
/* @compoundOperation */
/* @ROTR(S[14], 8) */
((S[14] >> 8) | (S[14] << sizeof(S[14]) * 8 - 8));
```

```
284     norx_word_t binaryOpTemp059 = (S[9] ^ S[14]);
        norx_word_t binaryOpTemp060 = (S[9] & S[14]);
286     norx_word_t binaryOpTemp130 = (binaryOpTemp060 << 1);
        S[9] = (binaryOpTemp059 ^ binaryOpTemp130);
288     S[4] = S[4] ^ S[9];
        S[4] =
290 /* @compoundOperation */
    /* @ROTR(S[4], 19) */
292 ((S[4] >> 19) | (S[4] << sizeof(S[4]) * 8 - 19));
        norx_word_t binaryOpTemp061 = (S[3] ^ S[4]);
294     norx_word_t binaryOpTemp062 = (S[3] & S[4]);
        norx_word_t binaryOpTemp131 = (binaryOpTemp062 << 1);
296     S[3] = (binaryOpTemp061 ^ binaryOpTemp131);
        S[14] = S[14] ^ S[3];
298     S[14] =
    /* @compoundOperation */
300 /* @ROTR(S[14], 40) */
    ((S[14] >> 40) | (S[14] << sizeof(S[14]) * 8 - 40));
302     norx_word_t binaryOpTemp063 = (S[9] ^ S[14]);
        norx_word_t binaryOpTemp064 = (S[9] & S[14]);
304     norx_word_t binaryOpTemp132 = (binaryOpTemp064 << 1);
        S[9] = (binaryOpTemp063 ^ binaryOpTemp132);
306     S[4] = S[4] ^ S[9];
        S[4] =
308 /* @compoundOperation */
    /* @ROTR(S[4], 63) */
310 ((S[4] >> 63) | (S[4] << sizeof(S[4]) * 8 - 63));
      }while (0);
312 }
```

5.5 Translation to other Formats

The second part of the process fulfilled by the framework handles the translation of the standardised IR into other cipher representation formats. The input to the process is the IR resulting from the application of the transformation steps described in the previous section. The output forms the final representation of the cipher in the format of a specific tool.

Two classes are mainly responsible for handling the transformation of the IR to other tool representations as illustrated in Figure 5.3.

- `CodeTransformer`: This component traverses the target function in the IR line per line. It queries the responsible `ToolTranslator` for representations of different operations and expressions in the target language / representations. The `ToolTranslator` returns a string representation for each of the commands, which the `CodeTransformer` inserts as command in the correct line of the target code (output).
- `ToolTranslator`: This component is responsible for setting up the basic environment of the target representation (e.g. classes, files, etc.). Further, it is responsible for providing a string representation of different operations in the target language when the corresponding method is called. The interface `ToolTranslator` defining the different methods has two methods for setting up and completing the translation: `setupTranslation` and `completeTranslation`. All other methods handle different operations such as occurrences of XOR, AND or SHIFT operations and return a string representing the operation in the target language (e.g. NLTool description language).

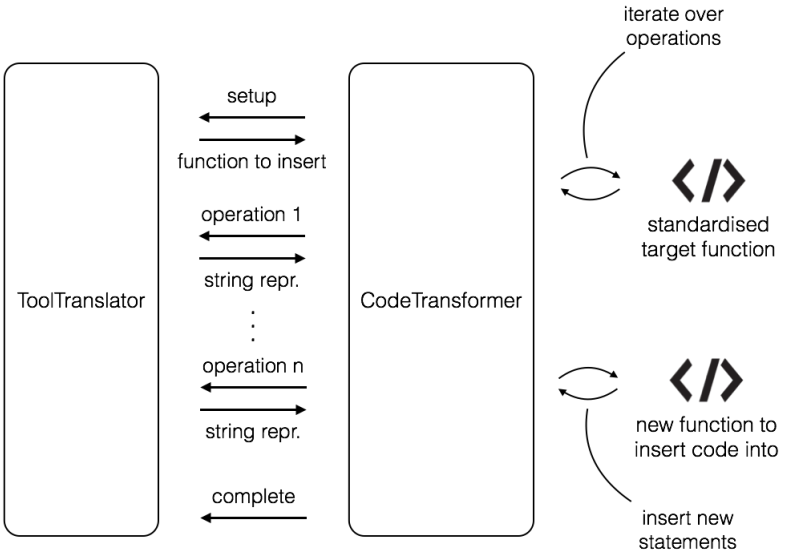

Figure 5.3: Interaction between the two main components of the framework.

To perform a transformation, the following steps are required:

1. A class implementing `ToolTranslator` is initialized and it's `setupTranslation` method is called to setup the target representation template / structure.
2. This method returns an IR element representing the function where the translated code of the target function shall be placed. This is needed in further steps to inject the new statements.
3. A `CodeTransformer` instance is created passing the previously created `ToolTranslator` instance. This sets up everything necessary to start the transformation.
4. Finally, the `CodeTransformer::transformFunction` method is called to start transforming. It traverses the IR representation of the source function, calls the corresponding transformation functions on the `ToolTranslator` object and injects the returned string representation at the right position in the target function.

5.5.1 CodeTransformer

Code Example 5.14 shows the code for the `CodeTransformer` implementation. The class inherits from the `ASTSimpleProcessing` class, which allows traversing the code by calling the `traverse` method. When a node is found during traversing the AST, the method `visit` is called on the object to handle it. The transformation process is started by calling the `transformFunction` method. It creates an *anchor statement* needed to keep

track of the position to insert new code in the function, and starts the traversal of the passed function.

The implementation of the `visit` method distinguishes three main types of nodes: operations, variable declarations and compound statements. It calls internal handle methods for all three types.

The handle methods determine the type of operation and call the corresponding method in the configured `ToolTransformer` object.

After the transformation has been performed by the translator, the statements are inserted into the target function.

Code Example 5.14: `CodeTransformer.cpp`

```
#include "CodeTransformer.h"
#include <boost/lexical_cast.hpp>
#include <boost/algorithm/string.hpp>
#include "TranslationUtils.h"


CodeTransformer::CodeTransformer(ToolTranslator* transformator)
{
    _nameCount = 0;
    _translator = transformator;
}

void CodeTransformer::transformFunction(SgFunctionDeclaration* function, SgFunctionDeclaration* destinationFunction
    )
{
    std::cout << "-------------------Starting Transformation ----------------------------------" << std::endl;

    //setting target function
    _targetFunction = destinationFunction;

    //creating anchor statement
    createAnchorStatement();

    //traversing roundfunction statements
    this->traverse(function, preorder);

    //removing anchor statement
    removeAnchorStatement();
}

void CodeTransformer::createAnchorStatement()
{
    //getting function body
    SgBasicBlock* functionBody = _targetFunction->get_definition()->get_body();

    //defining anchor dummy declaration
    _anchorDeclaration = SageBuilder::buildVariableDeclaration("anchor", SageBuilder::buildIntType());
    SageInterface::appendStatement(_anchorDeclaration, functionBody);
}

void CodeTransformer::removeAnchorStatement()
{
    //TODO: find way to delete anchor statement while keeping other instructions

    //removing anchor statement
    //functionBody->remove_statement(_anchorDeclaration);
    //SageInterface::removeStatement(_anchorDeclaration, true);
}

void CodeTransformer::visit(SgNode *n)
{
    SgAssignOp* assign = isSgAssignOp(n);
    SgVariableDeclaration* varDecl = isSgVariableDeclaration(n);
    if(assign || varDecl)
    {
        std::cout << "found node " << n->unparseToCompleteString() << " of type " << n->sage_class_name() << std::
    endl;

        SgNode* variable;
        SgNode* operationNode;
```

```
60          //if var is
            if(assign)
62          {
                variable = assign->get_lhs_operand();
64              operationNode = assign->get_rhs_operand();
            }
66          else
            {
68              //extracting name from subtree
                Rose_STL_Container<SgNode*> name = NodeQuery::querySubTree(varDecl,V_SgInitializedName);
70              ROSE_ASSERT(name.size() != 0);
                variable = name.at(0);
72
                //adding variable declaration
74              std::string variableDeclaration =  _translator->declareVariable(variable->unparseToString());
                addStatementToTargetFunction(variableDeclaration);
76
                //extracting AssignInitializer from subtree
78              Rose_STL_Container<SgNode*> assignInitializers = NodeQuery::querySubTree(varDecl, V_SgAssignInitializer
      );
80              //if no assign initializer present, it is just a declaration, returning
                if(assignInitializers.size() == 0)
82              {
                    return;
84              }

86              operationNode = assignInitializers.at(0);
            }
88
            //getting binary operation
90          Rose_STL_Container<SgNode*> opsInOperationSubtree = NodeQuery::querySubTree(operationNode, V_SgBinaryOp);
            ROSE_ASSERT(opsInOperationSubtree.size() != 0);
92
            //getting pointer ref expressions
94          Rose_STL_Container<SgNode*> pntrRefsInSubtree = NodeQuery::querySubTree(operationNode, V_SgPntrArrRefExp);

96          //calculating real binary ops
            int binaryOpsWithoutPntrRefs = opsInOperationSubtree.size() - pntrRefsInSubtree.size();
98
            //extracting (first if compound) binary expression
100         SgBinaryOp* operation  = isSgBinaryOp(opsInOperationSubtree.at(0));
            ROSE_ASSERT(operation != null);
102
            //defining transformed operation string to be filled
104         std::string transformedOperation = "";

106         //must be compound statement
            if(binaryOpsWithoutPntrRefs > 1)
108         {
                //handling compound statement
110             transformedOperation = handleCompoundOperation(variable, operation);
            }
112         else
            {
114             //handling binary operation
                transformedOperation = handleBinaryOperation(variable, operation);
116         }

118         //adding transformed statement if not empty
            if(transformedOperation.compare("") != 0)
120         {
                addStatementToTargetFunction(transformedOperation);
122         }
        }
124 }

126 void CodeTransformer::atTraversalEnd()
    {
128     std::cout << "Traversal end..." << std::endl;

130     //calling delegate method
        _translator->completeTranslation();
132 }

134 std::string CodeTransformer::handleCompoundOperation(SgNode* variable, SgNode* operation)
    {
136     std::string returnValue = "";

138     //this must be a rotation or compound operation
        //making sure it's annotated
140     if(!TranslationUtils::isSystemAnnotatedInParents(operation))
```

```
    {
142     std::cout << "rotation seems to be not annotated..." << operation->unparseToCompleteString() << std::endl;
        return "";
144 }

146 std::string operationDescriptionString = TranslationUtils::getCompoundOperationString(operation);

148 //rotation
    if(boost::starts_with(operationDescriptionString, "@ROT"))
150 {
        //determining direction of rotation
152     if(boost::starts_with(operationDescriptionString, "@ROTR"))
        {
154         //getting operands
            Rose_STL_Container<SgNode*> rightShifts = NodeQuery::querySubTree(operation, V_SgRshiftOp);
156         ROSE_ASSERT(rightShifts.size() != 0);

158         //casting element to right shift operation
            SgRshiftOp* rightShift = isSgRshiftOp(rightShifts.at(0));
160
            //calling translator to handle this
162         returnValue = _translator->handleRrotationOperation(variable, rightShift->get_lhs_operand(), rightShift
      ->get_rhs_operand());
        }
164     else if(boost::starts_with(operationDescriptionString, "@ROTL"))
        {
166         //getting operands
            Rose_STL_Container<SgNode*> leftshifts = NodeQuery::querySubTree(operation, V_SgLshiftOp);
168         ROSE_ASSERT(leftshifts.size() != 0);

170         //casting element to right left operation
            SgLshiftOp* leftShift = isSgLshiftOp(leftshifts.at(0));
172
            //calling translator to handle this
174         returnValue = _translator->handleLrotationOperation(variable, leftShift->get_lhs_operand(), leftShift->
      get_rhs_operand());
        }
176     else
        {
178         std::cout << "!!COULD NOT DETECT ROTATION DIRECTION!!" << std::endl;
            return "";
180     }

182 }

184 return returnValue;
}
186
std::string CodeTransformer::handleBinaryOperation(SgNode* variable, SgBinaryOp* operation)
188 {
    std::string returnValue = "";
190
    //this is a normalized operation
192 ROSE_ASSERT(operation != null);

194 if(SgBitXorOp* op = isSgBitXorOp(operation))
    {
196     returnValue = _translator->handleXorOperation(variable, op);
    }
198 else if(SgBitAndOp* op = isSgBitAndOp(operation))
    {
200     returnValue = _translator->handleAndOperation(variable, op);
    }
202 else if(SgLshiftOp* op = isSgLshiftOp(operation))
    {
204     returnValue = _translator->handleLshiftOperation(variable, op);
    }
206 else if(SgRshiftOp* op = isSgRshiftOp(operation))
    {
208     returnValue = _translator->handleRshiftOperation(variable, op);
    }
210 else if(SgAddOp* op = isSgAddOp(operation))
    {
212     returnValue = _translator->handleAddOperation(variable, op);
    }
214 else
    {
216     std::cout << "You need to implement handling for operation type " << operation->get_type()->
      unparseToCompleteString() << std::endl;
    }
218
    return returnValue;
220 }
```

```
222 void CodeTransformer::addStatementToTargetFunction(std::string statement)
    {
224     SageInterface::addTextForUnparser(_anchorDeclaration, "\n" + statement, AstUnparseAttribute::e_before);
    }
```

5.5.2 CodeTranslator

This section shows how the translation process was applied to NORX in order to transform it's code into an NLTool representation. This representation is given as a C++ class with the cipher description in it's constructor. The transformation will create the class and a method representing the round function. Further, it will create a constructor, which takes the number of rounds R as a parameter, creates the state words and calls the round method R times.

To perform this task, the class `NLToolTranslator` was implemented. This class inherits from `ToolTranslator` and implements the corresponding virtual methods. Code Example 5.15 shows the source file of this class.

The method `setupTranslation` takes a function (in IR standardised form) as an input and sets up the target representation structure accordingly. It adapts all types of parameters passed to the round function to the target type `ConditionWord` and builds a function definition with the same parameters plus the round number as integer. The function is given a name passed in the method. It then generates the header file of the class for the NLTool using the *boost* template engine and a predefined template. The templates for source and header file are illustrated in Code Example 5.16 and 5.17. The method then saves the created function declaration. This declaration is used to inject transformed statements. The final function code will be written to the target code file upon completion of the process.

The methods translating special operations or expressions are implemented facilitating template definitions for different types. For example, the `declareVariable` method is implemented as obtaining a template from `getFormatterForAddingConditionWord`, and completing the template using the passed variable name. It returns the statement as a string.

As seen in the previous section, the `CipherTransformer` takes the returned string and injects it into the target function. This takes place for each statement in the standard form of the source function.

The method `completeTranslation` is called upon completion of the process. It creates the target folder structure and writes the template code for the source file using the template shown in Code Example 5.17 and the string obtained from the transformed target function.

Code Example 5.18 and 5.19 show the output code of this procedure. It can be seen that the NLTool class was created and the code was properly transformed to the format.

Code Example 5.15: NLToolTranslator.cpp

```
#include "NLToolTranslator.h"
#include <boost/lexical_cast.hpp>
#include <CodeGenerationUtils.h>


#define null NULL

NLToolTranslator::NLToolTranslator()
{
    _tempVarCounter = 0;
}

//interface methods
SgFunctionDeclaration*  NLToolTranslator::setupTranslation(SgProject* project, std::string cipherName,
      SgFunctionDeclaration* sourceFunction)
{
    //resetting counter
    _tempVarCounter = 0;

    //setting name
    _cipherName = cipherName;

    //getting global scope of project
    SgGlobal* globalScope = SageInterface::getFirstGlobalScope(project);

    //getting existing parameter list of round function
    SgFunctionParameterList* parameterList = sourceFunction->get_parameterList();
    SgInitializedNamePtrList parameterArgsList = parameterList->get_args();

    //adapting types of all existing target function parameters parameters
    parameterList = CodeGenerationUtils::adaptAllTypesOfParameters(parameterList, "ConditionWord");

    //adding round number parameter
    SgInitializedName* roundNumberParameterArgument = SageBuilder::buildInitializedName(ToolTranslator::
      ROUND_NUMBER_PARAMETER_NAME, SageBuilder::buildIntType());
    SageInterface::appendArg(parameterList, roundNumberParameterArgument);

    //building function
    std::string roundFunctionName = _cipherName + "_round";
    _targetFunction = SageBuilder::buildDefiningFunctionDeclaration(SgName(roundFunctionName), SageBuilder::
      buildVoidType(), parameterList, globalScope);

    //creating folder
    std::string folderName = "NLTool";
    boost::filesystem::path folderPath = CodeGenerationUtils::createFolder(folderName);

    //creating header file
    std::string headerName = _cipherName + ".h";
    boost::filesystem::path headerPath = folderPath / boost::filesystem::path(headerName);

    //getting formatters from templates
    boost::format headerTemplateFormat = getFormatterForTemplateHeaderFile();

    //extracting signature of round function (without trailing void and curly brackets)
    std::string roundFunctionSignature = _targetFunction->unparseToString().erase(0,5);
    roundFunctionSignature = roundFunctionSignature.erase(roundFunctionSignature.length() - 2,
      roundFunctionSignature.length());

    //completing and writing header file
    std::string headerTemplate = (headerTemplateFormat % _cipherName % roundFunctionSignature).str();
    CodeGenerationUtils::writeStringToFile(headerTemplate ,headerPath);

    //prepending main function to global scope
    SageInterface::appendStatement(_targetFunction, globalScope);

    return _targetFunction;
}

void NLToolTranslator::completeTranslation()
{
    //creating folder
    std::string folderName = "NLTool";
    boost::filesystem::path folderPath = CodeGenerationUtils::createFolder(folderName);

    //creating source files
    std::string sourceName = _cipherName + ".cpp";
    boost::filesystem::path sourcePath = folderPath / boost::filesystem::path(sourceName);

    //getting formatter for source file
    boost::format sourceTemplateFormat = getFormatterForTemplateSourceFile();
```

```
78      //constructing function signature (without trailing void)
        std::string roundFunctionString = _targetFunction->unparseToCompleteString().erase(0,6);
80
        //completing and writing source file
82      std::string sourceTemplate = (sourceTemplateFormat % _cipherName % roundFunctionString).str();
        CodeGenerationUtils::writeStringToFile(sourceTemplate, sourcePath);
84
        //removing temporary function definition
86      SageInterface::removeStatement(_targetFunction, false);

88      std::cout << "-----completed translation ----" << std::endl;
    }
90
    std::string NLToolTranslator::declareVariable(std::string name)
92  {
        std::cout << "Declaring variable " << std::endl;
94
        boost::format formatter = getFormatterForAddingConditionWord(true);
96      std::string addCall = (formatter % name % name % 0 % _tempVarCounter).str();
        _tempVarCounter++;
98
        return addCall;
100 }
102
    std::string NLToolTranslator::handleXorOperation(SgNode* targetVariable, SgBitXorOp* xorOperation)
104 {
        std::cout << "Transforming XOR" << std::endl;
106
        boost::format formatter = getFormatterForOperationWithParameters(3);
108     std::string xorCall = (formatter % "XOR2" % xorOperation->get_lhs_operand()->unparseToCompleteString() %
                               xorOperation->get_rhs_operand()->unparseToCompleteString() % targetVariable->
          unparseToCompleteString()).str();
110
        return xorCall;
112 }
    std::string NLToolTranslator::handleAndOperation(SgNode* targetVariable, SgBitAndOp* andOperation)
114 {
        std::cout << "Transforming AND" << std::endl;
116
        boost::format formatter = getFormatterForOperationWithParameters(3);
118     std::string andCall = (formatter % "AND2" % andOperation->get_lhs_operand()->unparseToCompleteString() %
                               andOperation->get_rhs_operand()->unparseToCompleteString() % targetVariable->
          unparseToCompleteString()).str();
120
        return andCall;
122 }

124 std::string NLToolTranslator::handleAddOperation(SgNode* targetVariable, SgAddOp* addOperation)
    {
126     std::cout << "Transforming ADD" << std::endl;

128     boost::format formatter = getFormatterForOperationWithParameters(3);
        std::string andCall = (formatter % "ADD2" % addOperation->get_lhs_operand()->unparseToCompleteString() %
130                            addOperation->get_rhs_operand()->unparseToCompleteString() % targetVariable->
          unparseToCompleteString()).str();

132     return andCall;
    }
134
    std::string NLToolTranslator::handleLshiftOperation(SgNode* targetVariable, SgLshiftOp* leftShiftOperation)
136 {
        std::cout << "Transforming LShift" << std::endl;
138
        return targetVariable->unparseToCompleteString() + " = " + leftShiftOperation->get_lhs_operand()->
          unparseToCompleteString() + "->Shl("+
140           leftShiftOperation->get_rhs_operand()->unparseToCompleteString() + ");";
    }
142
    std::string NLToolTranslator::handleRshiftOperation(SgNode* targetVariable, SgRshiftOp* rightShiftOperation)
144 {
        std::cout << "Transforming RShift" << std::endl;
146
        return targetVariable->unparseToCompleteString() + " = " + rightShiftOperation->get_lhs_operand()->
          unparseToCompleteString() + "->Shr("+
148           rightShiftOperation->get_rhs_operand()->unparseToCompleteString() + ");";
    }
150
    std::string NLToolTranslator::handleRrotationOperation(SgNode* targetVariable, SgNode* shiftedExpression, SgNode*
          shiftIndexExpression)
152 {
        std::cout << "Transforming Rrotation" << std::endl;
154
```

```
    return targetVariable->unparseToCompleteString() + " = " + shiftedExpression->unparseToCompleteString() + "->Rotr("+
            shiftIndexExpression->unparseToCompleteString() + ");";
}

std::string NLToolTranslator::handleLrotationOperation(SgNode* targetVariable, SgNode* shiftedExpression, SgNode* shiftIndexExpression)
{
    std::cout << "Transforming Lrotation" << std::endl;

    return targetVariable->unparseToCompleteString() + " = " + shiftedExpression->unparseToCompleteString() + "->Rotl("+
            shiftIndexExpression->unparseToCompleteString() + ");";
}


boost::format NLToolTranslator::getFormatterForOperationWithParameters(int numberOfParameters)
{
    std::string formatString = "Add(new BitsliceStep<%1%>(";
    for(int i = 2; i <= numberOfParameters; ++i)
    {
        formatString = formatString + "%" + boost::lexical_cast<std::string>(i) + "%, ";
    }
    formatString = formatString + "%" + boost::lexical_cast<std::string>(numberOfParameters + 1) +"%));";

    return boost::format(formatString);
}

boost::format NLToolTranslator::getFormatterForAddingConditionWord(bool isSubword)
{
    //AddConditionWord("F", i, 5 + i * 3 + 0, 0, SUBWORD);
    std::string formatString = "ConditionWord %1% = AddConditionWord(\"%2%\"," + ToolTranslator::ROUND_NUMBER_PARAMETER_NAME + ", %3% , %4%";
    if(isSubword == true)
    {
        formatString = formatString + ", SUBWORD";
    }
    formatString = formatString + ");";

    return boost::format(formatString);
}

boost::format NLToolTranslator::getFormatterForTemplateHeaderFile()
{
    boost::filesystem::path headerTemplatePath("../inputs/templates/nltool/nltool_template_header.h");
    std::string headerFileTemplate = CodeGenerationUtils::readStringFromFile(headerTemplatePath);

    // std::cout << "Read template header " << headerFileTemplate << std::endl;

    return boost::format(headerFileTemplate);
}


boost::format NLToolTranslator::getFormatterForTemplateSourceFile()
{
    boost::filesystem::path sourceTemplatePath("../inputs/templates/nltool/nltool_template_source.cpp");
    std::string sourceFileTemplate = CodeGenerationUtils::readStringFromFile(sourceTemplatePath);

    //std::cout << "Read template source " << sourceFileTemplate << std::endl;

    return boost::format(sourceFileTemplate);
}
```

Code Example 5.16: The template for the NLTool header file.

```
/*
 * auto generated cipher representation
 */

#ifndef %1%_H_
#define %1%_H_

#include "hash.h"

class %1%: public Hash
{
public:
  %1%(int N);
  void %2%;
};

#endif // %1%_H_
```

Code Example 5.17: The template for the NLTool source file.

```
#include "%1%.h"

////////////////////////////// class %1% ////////////////////////////////

%1%::%1%(int N, int R) :
    Hash(N)
{

}

void %1%::%2%
```

Code Example 5.18: The header file of the resulting tool representation of NORX.

```
/*
 * auto generated cipher representation
 */

#ifndef Norx_H_
#define Norx_H_

#include "hash.h"

class Norx: public Hash
{
public:
  Norx(int N, int R);
  void Norx_round(class ConditionWord S[16UL],int roundNumber);
};

#endif // Norx_H_
```

Code Example 5.19: The source file of the resulting tool representation of NORX.

```
#include "Norx.h"

////////////////////////////// class Norx ////////////////////////////////

Norx::Norx(int N, int R) :
    Hash(N)
{

}

void Norx::Norx_round(class ConditionWord S[16UL],int roundNumber)
{
ConditionWord binaryOpTemp01 = AddConditionWord("binaryOpTemp01",roundNumber, 0 , 0, SUBWORD);
Add(new BitsliceStep<XOR2>(S[0], S[4], binaryOpTemp01));
ConditionWord binaryOpTemp02 = AddConditionWord("binaryOpTemp02",roundNumber, 0 , 1, SUBWORD);
Add(new BitsliceStep<AND2>(S[0], S[4], binaryOpTemp02));
ConditionWord binaryOpTemp11 = AddConditionWord("binaryOpTemp11",roundNumber, 0 , 2, SUBWORD);
binaryOpTemp11 = binaryOpTemp02->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp01, binaryOpTemp11, S[0]));
Add(new BitsliceStep<XOR2>(S[12], S[0], S[12]));
S[12] = S[12]->Rotr(8);
ConditionWord binaryOpTemp03 = AddConditionWord("binaryOpTemp03",roundNumber, 0 , 3, SUBWORD);
Add(new BitsliceStep<XOR2>(S[8], S[12], binaryOpTemp03));
ConditionWord binaryOpTemp04 = AddConditionWord("binaryOpTemp04",roundNumber, 0 , 4, SUBWORD);
Add(new BitsliceStep<AND2>(S[8], S[12], binaryOpTemp04));
ConditionWord binaryOpTemp12 = AddConditionWord("binaryOpTemp12",roundNumber, 0 , 5, SUBWORD);
binaryOpTemp12 = binaryOpTemp04->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp03, binaryOpTemp12, S[8]));
Add(new BitsliceStep<XOR2>(S[4], S[8], S[4]));
S[4] = S[4]->Rotr(19);
ConditionWord binaryOpTemp05 = AddConditionWord("binaryOpTemp05",roundNumber, 0 , 6, SUBWORD);
Add(new BitsliceStep<XOR2>(S[0], S[4], binaryOpTemp05));
ConditionWord binaryOpTemp06 = AddConditionWord("binaryOpTemp06",roundNumber, 0 , 7, SUBWORD);
Add(new BitsliceStep<AND2>(S[0], S[4], binaryOpTemp06));
ConditionWord binaryOpTemp13 = AddConditionWord("binaryOpTemp13",roundNumber, 0 , 8, SUBWORD);
binaryOpTemp13 = binaryOpTemp06->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp05, binaryOpTemp13, S[0]));
Add(new BitsliceStep<XOR2>(S[12], S[0], S[12]));
S[12] = S[12]->Rotr(40);
```

```
ConditionWord binaryOpTemp07 = AddConditionWord("binaryOpTemp07",roundNumber, 0 , 9, SUBWORD);
Add(new BitsliceStep<XOR2>(S[8], S[12], binaryOpTemp07));
ConditionWord binaryOpTemp08 = AddConditionWord("binaryOpTemp08",roundNumber, 0 , 10, SUBWORD);
Add(new BitsliceStep<AND2>(S[8], S[12], binaryOpTemp08));
ConditionWord binaryOpTemp14 = AddConditionWord("binaryOpTemp14",roundNumber, 0 , 11, SUBWORD);
binaryOpTemp14 = binaryOpTemp08->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp07, binaryOpTemp14, S[8]));
Add(new BitsliceStep<XOR2>(S[4], S[8], S[4]));
S[4] = S[4]->Rotr(63);
ConditionWord binaryOpTemp09 = AddConditionWord("binaryOpTemp09",roundNumber, 0 , 12, SUBWORD);
Add(new BitsliceStep<XOR2>(S[1], S[5], binaryOpTemp09));
ConditionWord binaryOpTemp010 = AddConditionWord("binaryOpTemp010",roundNumber, 0 , 13, SUBWORD);
Add(new BitsliceStep<AND2>(S[1], S[5], binaryOpTemp010));
ConditionWord binaryOpTemp15 = AddConditionWord("binaryOpTemp15",roundNumber, 0 , 14, SUBWORD);
binaryOpTemp15 = binaryOpTemp010->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp09, binaryOpTemp15, S[1]));
Add(new BitsliceStep<XOR2>(S[13], S[1], S[13]));
S[13] = S[13]->Rotr(8);
ConditionWord binaryOpTemp011 = AddConditionWord("binaryOpTemp011",roundNumber, 0 , 15, SUBWORD);
Add(new BitsliceStep<XOR2>(S[9], S[13], binaryOpTemp011));
ConditionWord binaryOpTemp012 = AddConditionWord("binaryOpTemp012",roundNumber, 0 , 16, SUBWORD);
Add(new BitsliceStep<AND2>(S[9], S[13], binaryOpTemp012));
ConditionWord binaryOpTemp16 = AddConditionWord("binaryOpTemp16",roundNumber, 0 , 17, SUBWORD);
binaryOpTemp16 = binaryOpTemp012->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp011, binaryOpTemp16, S[9]));
Add(new BitsliceStep<XOR2>(S[5], S[9], S[5]));
S[5] = S[5]->Rotr(19);
ConditionWord binaryOpTemp013 = AddConditionWord("binaryOpTemp013",roundNumber, 0 , 18, SUBWORD);
Add(new BitsliceStep<XOR2>(S[1], S[5], binaryOpTemp013));
ConditionWord binaryOpTemp014 = AddConditionWord("binaryOpTemp014",roundNumber, 0 , 19, SUBWORD);
Add(new BitsliceStep<AND2>(S[1], S[5], binaryOpTemp014));
ConditionWord binaryOpTemp17 = AddConditionWord("binaryOpTemp17",roundNumber, 0 , 20, SUBWORD);
binaryOpTemp17 = binaryOpTemp014->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp013, binaryOpTemp17, S[1]));
Add(new BitsliceStep<XOR2>(S[13], S[1], S[13]));
S[13] = S[13]->Rotr(40);
ConditionWord binaryOpTemp015 = AddConditionWord("binaryOpTemp015",roundNumber, 0 , 21, SUBWORD);
Add(new BitsliceStep<XOR2>(S[9], S[13], binaryOpTemp015));
ConditionWord binaryOpTemp016 = AddConditionWord("binaryOpTemp016",roundNumber, 0 , 22, SUBWORD);
Add(new BitsliceStep<AND2>(S[9], S[13], binaryOpTemp016));
ConditionWord binaryOpTemp18 = AddConditionWord("binaryOpTemp18",roundNumber, 0 , 23, SUBWORD);
binaryOpTemp18 = binaryOpTemp016->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp015, binaryOpTemp18, S[9]));
Add(new BitsliceStep<XOR2>(S[5], S[9], S[5]));
S[5] = S[5]->Rotr(63);
ConditionWord binaryOpTemp017 = AddConditionWord("binaryOpTemp017",roundNumber, 0 , 24, SUBWORD);
Add(new BitsliceStep<XOR2>(S[2], S[6], binaryOpTemp017));
ConditionWord binaryOpTemp018 = AddConditionWord("binaryOpTemp018",roundNumber, 0 , 25, SUBWORD);
Add(new BitsliceStep<AND2>(S[2], S[6], binaryOpTemp018));
ConditionWord binaryOpTemp19 = AddConditionWord("binaryOpTemp19",roundNumber, 0 , 26, SUBWORD);
binaryOpTemp19 = binaryOpTemp018->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp017, binaryOpTemp19, S[2]));
Add(new BitsliceStep<XOR2>(S[14], S[2], S[14]));
S[14] = S[14]->Rotr(8);
ConditionWord binaryOpTemp019 = AddConditionWord("binaryOpTemp019",roundNumber, 0 , 27, SUBWORD);
Add(new BitsliceStep<XOR2>(S[10], S[14], binaryOpTemp019));
ConditionWord binaryOpTemp020 = AddConditionWord("binaryOpTemp020",roundNumber, 0 , 28, SUBWORD);
Add(new BitsliceStep<AND2>(S[10], S[14], binaryOpTemp020));
ConditionWord binaryOpTemp110 = AddConditionWord("binaryOpTemp110",roundNumber, 0 , 29, SUBWORD);
binaryOpTemp110 = binaryOpTemp020->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp019, binaryOpTemp110, S[10]));
Add(new BitsliceStep<XOR2>(S[6], S[10], S[6]));
S[6] = S[6]->Rotr(19);
ConditionWord binaryOpTemp021 = AddConditionWord("binaryOpTemp021",roundNumber, 0 , 30, SUBWORD);
Add(new BitsliceStep<XOR2>(S[2], S[6], binaryOpTemp021));
ConditionWord binaryOpTemp022 = AddConditionWord("binaryOpTemp022",roundNumber, 0 , 31, SUBWORD);
Add(new BitsliceStep<AND2>(S[2], S[6], binaryOpTemp022));
ConditionWord binaryOpTemp111 = AddConditionWord("binaryOpTemp111",roundNumber, 0 , 32, SUBWORD);
binaryOpTemp111 = binaryOpTemp022->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp021, binaryOpTemp111, S[2]));
Add(new BitsliceStep<XOR2>(S[14], S[2], S[14]));
S[14] = S[14]->Rotr(40);
ConditionWord binaryOpTemp023 = AddConditionWord("binaryOpTemp023",roundNumber, 0 , 33, SUBWORD);
Add(new BitsliceStep<XOR2>(S[10], S[14], binaryOpTemp023));
ConditionWord binaryOpTemp024 = AddConditionWord("binaryOpTemp024",roundNumber, 0 , 34, SUBWORD);
Add(new BitsliceStep<AND2>(S[10], S[14], binaryOpTemp024));
ConditionWord binaryOpTemp112 = AddConditionWord("binaryOpTemp112",roundNumber, 0 , 35, SUBWORD);
binaryOpTemp112 = binaryOpTemp024->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp023, binaryOpTemp112, S[10]));
Add(new BitsliceStep<XOR2>(S[6], S[10], S[6]));
S[6] = S[6]->Rotr(63);
ConditionWord binaryOpTemp025 = AddConditionWord("binaryOpTemp025",roundNumber, 0 , 36, SUBWORD);
Add(new BitsliceStep<XOR2>(S[3], S[7], binaryOpTemp025));
```

```
ConditionWord binaryOpTemp026 = AddConditionWord("binaryOpTemp026",roundNumber, 0 , 37, SUBWORD);
Add(new BitsliceStep<AND2>(S[3], S[7], binaryOpTemp026));
ConditionWord binaryOpTemp113 = AddConditionWord("binaryOpTemp113",roundNumber, 0 , 38, SUBWORD);
binaryOpTemp113 = binaryOpTemp026->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp025, binaryOpTemp113, S[3]));
Add(new BitsliceStep<XOR2>(S[15], S[3], S[15]));
S[15] = S[15]->Rotr(8);
ConditionWord binaryOpTemp027 = AddConditionWord("binaryOpTemp027",roundNumber, 0 , 39, SUBWORD);
Add(new BitsliceStep<XOR2>(S[11], S[15], binaryOpTemp027));
ConditionWord binaryOpTemp028 = AddConditionWord("binaryOpTemp028",roundNumber, 0 , 40, SUBWORD);
Add(new BitsliceStep<AND2>(S[11], S[15], binaryOpTemp028));
ConditionWord binaryOpTemp114 = AddConditionWord("binaryOpTemp114",roundNumber, 0 , 41, SUBWORD);
binaryOpTemp114 = binaryOpTemp028->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp027, binaryOpTemp114, S[11]));
Add(new BitsliceStep<XOR2>(S[7], S[11], S[7]));
S[7] = S[7]->Rotr(19);
ConditionWord binaryOpTemp029 = AddConditionWord("binaryOpTemp029",roundNumber, 0 , 42, SUBWORD);
Add(new BitsliceStep<XOR2>(S[3], S[7], binaryOpTemp029));
ConditionWord binaryOpTemp030 = AddConditionWord("binaryOpTemp030",roundNumber, 0 , 43, SUBWORD);
Add(new BitsliceStep<AND2>(S[3], S[7], binaryOpTemp030));
ConditionWord binaryOpTemp115 = AddConditionWord("binaryOpTemp115",roundNumber, 0 , 44, SUBWORD);
binaryOpTemp115 = binaryOpTemp030->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp029, binaryOpTemp115, S[3]));
Add(new BitsliceStep<XOR2>(S[15], S[3], S[15]));
S[15] = S[15]->Rotr(40);
ConditionWord binaryOpTemp031 = AddConditionWord("binaryOpTemp031",roundNumber, 0 , 45, SUBWORD);
Add(new BitsliceStep<XOR2>(S[11], S[15], binaryOpTemp031));
ConditionWord binaryOpTemp032 = AddConditionWord("binaryOpTemp032",roundNumber, 0 , 46, SUBWORD);
Add(new BitsliceStep<AND2>(S[11], S[15], binaryOpTemp032));
ConditionWord binaryOpTemp116 = AddConditionWord("binaryOpTemp116",roundNumber, 0 , 47, SUBWORD);
binaryOpTemp116 = binaryOpTemp032->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp031, binaryOpTemp116, S[11]));
Add(new BitsliceStep<XOR2>(S[7], S[11], S[7]));
S[7] = S[7]->Rotr(63);
ConditionWord binaryOpTemp033 = AddConditionWord("binaryOpTemp033",roundNumber, 0 , 48, SUBWORD);
Add(new BitsliceStep<XOR2>(S[0], S[5], binaryOpTemp033));
ConditionWord binaryOpTemp034 = AddConditionWord("binaryOpTemp034",roundNumber, 0 , 49, SUBWORD);
Add(new BitsliceStep<AND2>(S[0], S[5], binaryOpTemp034));
ConditionWord binaryOpTemp117 = AddConditionWord("binaryOpTemp117",roundNumber, 0 , 50, SUBWORD);
binaryOpTemp117 = binaryOpTemp034->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp033, binaryOpTemp117, S[0]));
Add(new BitsliceStep<XOR2>(S[15], S[0], S[15]));
S[15] = S[15]->Rotr(8);
ConditionWord binaryOpTemp035 = AddConditionWord("binaryOpTemp035",roundNumber, 0 , 51, SUBWORD);
Add(new BitsliceStep<XOR2>(S[10], S[15], binaryOpTemp035));
ConditionWord binaryOpTemp036 = AddConditionWord("binaryOpTemp036",roundNumber, 0 , 52, SUBWORD);
Add(new BitsliceStep<AND2>(S[10], S[15], binaryOpTemp036));
ConditionWord binaryOpTemp118 = AddConditionWord("binaryOpTemp118",roundNumber, 0 , 53, SUBWORD);
binaryOpTemp118 = binaryOpTemp036->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp035, binaryOpTemp118, S[10]));
Add(new BitsliceStep<XOR2>(S[5], S[10], S[5]));
S[5] = S[5]->Rotr(19);
ConditionWord binaryOpTemp037 = AddConditionWord("binaryOpTemp037",roundNumber, 0 , 54, SUBWORD);
Add(new BitsliceStep<XOR2>(S[0], S[5], binaryOpTemp037));
ConditionWord binaryOpTemp038 = AddConditionWord("binaryOpTemp038",roundNumber, 0 , 55, SUBWORD);
Add(new BitsliceStep<AND2>(S[0], S[5], binaryOpTemp038));
ConditionWord binaryOpTemp119 = AddConditionWord("binaryOpTemp119",roundNumber, 0 , 56, SUBWORD);
binaryOpTemp119 = binaryOpTemp038->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp037, binaryOpTemp119, S[0]));
Add(new BitsliceStep<XOR2>(S[15], S[0], S[15]));
S[15] = S[15]->Rotr(40);
ConditionWord binaryOpTemp039 = AddConditionWord("binaryOpTemp039",roundNumber, 0 , 57, SUBWORD);
Add(new BitsliceStep<XOR2>(S[10], S[15], binaryOpTemp039));
ConditionWord binaryOpTemp040 = AddConditionWord("binaryOpTemp040",roundNumber, 0 , 58, SUBWORD);
Add(new BitsliceStep<AND2>(S[10], S[15], binaryOpTemp040));
ConditionWord binaryOpTemp120 = AddConditionWord("binaryOpTemp120",roundNumber, 0 , 59, SUBWORD);
binaryOpTemp120 = binaryOpTemp040->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp039, binaryOpTemp120, S[10]));
Add(new BitsliceStep<XOR2>(S[5], S[10], S[5]));
S[5] = S[5]->Rotr(63);
ConditionWord binaryOpTemp041 = AddConditionWord("binaryOpTemp041",roundNumber, 0 , 60, SUBWORD);
Add(new BitsliceStep<XOR2>(S[1], S[6], binaryOpTemp041));
ConditionWord binaryOpTemp042 = AddConditionWord("binaryOpTemp042",roundNumber, 0 , 61, SUBWORD);
Add(new BitsliceStep<AND2>(S[1], S[6], binaryOpTemp042));
ConditionWord binaryOpTemp121 = AddConditionWord("binaryOpTemp121",roundNumber, 0 , 62, SUBWORD);
binaryOpTemp121 = binaryOpTemp042->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp041, binaryOpTemp121, S[1]));
Add(new BitsliceStep<XOR2>(S[12], S[1], S[12]));
S[12] = S[12]->Rotr(8);
ConditionWord binaryOpTemp043 = AddConditionWord("binaryOpTemp043",roundNumber, 0 , 63, SUBWORD);
Add(new BitsliceStep<XOR2>(S[11], S[12], binaryOpTemp043));
ConditionWord binaryOpTemp044 = AddConditionWord("binaryOpTemp044",roundNumber, 0 , 64, SUBWORD);
Add(new BitsliceStep<AND2>(S[11], S[12], binaryOpTemp044));
```

```
206 ConditionWord binaryOpTemp122 = AddConditionWord("binaryOpTemp122",roundNumber, 0 , 65, SUBWORD);
    binaryOpTemp122 = binaryOpTemp044->Shl(1);
208 Add(new BitsliceStep<XOR2>(binaryOpTemp043, binaryOpTemp122, S[11]));
    Add(new BitsliceStep<XOR2>(S[6], S[11], S[6]));
210 S[6] = S[6]->Rotr(19);
    ConditionWord binaryOpTemp045 = AddConditionWord("binaryOpTemp045",roundNumber, 0 , 66, SUBWORD);
212 Add(new BitsliceStep<XOR2>(S[1], S[6], binaryOpTemp045));
    ConditionWord binaryOpTemp046 = AddConditionWord("binaryOpTemp046",roundNumber, 0 , 67, SUBWORD);
214 Add(new BitsliceStep<AND2>(S[1], S[6], binaryOpTemp046));
    ConditionWord binaryOpTemp123 = AddConditionWord("binaryOpTemp123",roundNumber, 0 , 68, SUBWORD);
216 binaryOpTemp123 = binaryOpTemp046->Shl(1);
    Add(new BitsliceStep<XOR2>(binaryOpTemp045, binaryOpTemp123, S[1]));
218 Add(new BitsliceStep<XOR2>(S[12], S[1], S[12]));
    S[12] = S[12]->Rotr(40);
220 ConditionWord binaryOpTemp047 = AddConditionWord("binaryOpTemp047",roundNumber, 0 , 69, SUBWORD);
    Add(new BitsliceStep<XOR2>(S[11], S[12], binaryOpTemp047));
222 ConditionWord binaryOpTemp048 = AddConditionWord("binaryOpTemp048",roundNumber, 0 , 70, SUBWORD);
    Add(new BitsliceStep<AND2>(S[11], S[12], binaryOpTemp048));
224 ConditionWord binaryOpTemp124 = AddConditionWord("binaryOpTemp124",roundNumber, 0 , 71, SUBWORD);
    binaryOpTemp124 = binaryOpTemp048->Shl(1);
226 Add(new BitsliceStep<XOR2>(binaryOpTemp047, binaryOpTemp124, S[11]));
    Add(new BitsliceStep<XOR2>(S[6], S[11], S[6]));
228 S[6] = S[6]->Rotr(63);
    ConditionWord binaryOpTemp049 = AddConditionWord("binaryOpTemp049",roundNumber, 0 , 72, SUBWORD);
230 Add(new BitsliceStep<XOR2>(S[2], S[7], binaryOpTemp049));
    ConditionWord binaryOpTemp050 = AddConditionWord("binaryOpTemp050",roundNumber, 0 , 73, SUBWORD);
232 Add(new BitsliceStep<AND2>(S[2], S[7], binaryOpTemp050));
    ConditionWord binaryOpTemp125 = AddConditionWord("binaryOpTemp125",roundNumber, 0 , 74, SUBWORD);
234 binaryOpTemp125 = binaryOpTemp050->Shl(1);
    Add(new BitsliceStep<XOR2>(binaryOpTemp049, binaryOpTemp125, S[2]));
236 Add(new BitsliceStep<XOR2>(S[13], S[2], S[13]));
    S[13] = S[13]->Rotr(8);
238 ConditionWord binaryOpTemp051 = AddConditionWord("binaryOpTemp051",roundNumber, 0 , 75, SUBWORD);
    Add(new BitsliceStep<XOR2>(S[8], S[13], binaryOpTemp051));
240 ConditionWord binaryOpTemp052 = AddConditionWord("binaryOpTemp052",roundNumber, 0 , 76, SUBWORD);
    Add(new BitsliceStep<AND2>(S[8], S[13], binaryOpTemp052));
242 ConditionWord binaryOpTemp126 = AddConditionWord("binaryOpTemp126",roundNumber, 0 , 77, SUBWORD);
    binaryOpTemp126 = binaryOpTemp052->Shl(1);
244 Add(new BitsliceStep<XOR2>(binaryOpTemp051, binaryOpTemp126, S[8]));
    Add(new BitsliceStep<XOR2>(S[7], S[8], S[7]));
246 S[7] = S[7]->Rotr(19);
    ConditionWord binaryOpTemp053 = AddConditionWord("binaryOpTemp053",roundNumber, 0 , 78, SUBWORD);
248 Add(new BitsliceStep<XOR2>(S[2], S[7], binaryOpTemp053));
    ConditionWord binaryOpTemp054 = AddConditionWord("binaryOpTemp054",roundNumber, 0 , 79, SUBWORD);
250 Add(new BitsliceStep<AND2>(S[2], S[7], binaryOpTemp054));
    ConditionWord binaryOpTemp127 = AddConditionWord("binaryOpTemp127",roundNumber, 0 , 80, SUBWORD);
252 binaryOpTemp127 = binaryOpTemp054->Shl(1);
    Add(new BitsliceStep<XOR2>(binaryOpTemp053, binaryOpTemp127, S[2]));
254 Add(new BitsliceStep<XOR2>(S[13], S[2], S[13]));
    S[13] = S[13]->Rotr(40);
256 ConditionWord binaryOpTemp055 = AddConditionWord("binaryOpTemp055",roundNumber, 0 , 81, SUBWORD);
    Add(new BitsliceStep<XOR2>(S[8], S[13], binaryOpTemp055));
258 ConditionWord binaryOpTemp056 = AddConditionWord("binaryOpTemp056",roundNumber, 0 , 82, SUBWORD);
    Add(new BitsliceStep<AND2>(S[8], S[13], binaryOpTemp056));
260 ConditionWord binaryOpTemp128 = AddConditionWord("binaryOpTemp128",roundNumber, 0 , 83, SUBWORD);
    binaryOpTemp128 = binaryOpTemp056->Shl(1);
262 Add(new BitsliceStep<XOR2>(binaryOpTemp055, binaryOpTemp128, S[8]));
    Add(new BitsliceStep<XOR2>(S[7], S[8], S[7]));
264 S[7] = S[7]->Rotr(63);
    ConditionWord binaryOpTemp057 = AddConditionWord("binaryOpTemp057",roundNumber, 0 , 84, SUBWORD);
266 Add(new BitsliceStep<XOR2>(S[3], S[4], binaryOpTemp057));
    ConditionWord binaryOpTemp058 = AddConditionWord("binaryOpTemp058",roundNumber, 0 , 85, SUBWORD);
268 Add(new BitsliceStep<AND2>(S[3], S[4], binaryOpTemp058));
    ConditionWord binaryOpTemp129 = AddConditionWord("binaryOpTemp129",roundNumber, 0 , 86, SUBWORD);
270 binaryOpTemp129 = binaryOpTemp058->Shl(1);
    Add(new BitsliceStep<XOR2>(binaryOpTemp057, binaryOpTemp129, S[3]));
272 Add(new BitsliceStep<XOR2>(S[14], S[3], S[14]));
    S[14] = S[14]->Rotr(8);
274 ConditionWord binaryOpTemp059 = AddConditionWord("binaryOpTemp059",roundNumber, 0 , 87, SUBWORD);
    Add(new BitsliceStep<XOR2>(S[9], S[14], binaryOpTemp059));
276 ConditionWord binaryOpTemp060 = AddConditionWord("binaryOpTemp060",roundNumber, 0 , 88, SUBWORD);
    Add(new BitsliceStep<AND2>(S[9], S[14], binaryOpTemp060));
278 ConditionWord binaryOpTemp130 = AddConditionWord("binaryOpTemp130",roundNumber, 0 , 89, SUBWORD);
    binaryOpTemp130 = binaryOpTemp060->Shl(1);
280 Add(new BitsliceStep<XOR2>(binaryOpTemp059, binaryOpTemp130, S[9]));
    Add(new BitsliceStep<XOR2>(S[4], S[9], S[4]));
282 S[4] = S[4]->Rotr(19);
    ConditionWord binaryOpTemp061 = AddConditionWord("binaryOpTemp061",roundNumber, 0 , 90, SUBWORD);
284 Add(new BitsliceStep<XOR2>(S[3], S[4], binaryOpTemp061));
    ConditionWord binaryOpTemp062 = AddConditionWord("binaryOpTemp062",roundNumber, 0 , 91, SUBWORD);
286 Add(new BitsliceStep<AND2>(S[3], S[4], binaryOpTemp062));
    ConditionWord binaryOpTemp131 = AddConditionWord("binaryOpTemp131",roundNumber, 0 , 92, SUBWORD);
288 binaryOpTemp131 = binaryOpTemp062->Shl(1);
```

```
Add(new BitsliceStep<XOR2>(binaryOpTemp061, binaryOpTemp131, S[3]));
Add(new BitsliceStep<XOR2>(S[14], S[3], S[14]));
S[14] = S[14]->Rotr(40);
ConditionWord binaryOpTemp063 = AddConditionWord("binaryOpTemp063",roundNumber, 0 , 93, SUBWORD);
Add(new BitsliceStep<XOR2>(S[9], S[14], binaryOpTemp063));
ConditionWord binaryOpTemp064 = AddConditionWord("binaryOpTemp064",roundNumber, 0 , 94, SUBWORD);
Add(new BitsliceStep<AND2>(S[9], S[14], binaryOpTemp064));
ConditionWord binaryOpTemp132 = AddConditionWord("binaryOpTemp132",roundNumber, 0 , 95, SUBWORD);
binaryOpTemp132 = binaryOpTemp064->Shl(1);
Add(new BitsliceStep<XOR2>(binaryOpTemp063, binaryOpTemp132, S[9]));
Add(new BitsliceStep<XOR2>(S[4], S[9], S[4]));
S[4] = S[4]->Rotr(63);
  int anchor;
}
```

5.5.3 Analysis Code

Code Example 5.20 shows the code used to take the input C reference implementation and transform it to a NLTool cipher description. The project uses the ROSE `frontend` method to parse the input code, tests the generated IS (AST) and fix preconditions assumed by the framework. Afterwards it uses the `findRoundFunction` method to find the annotated input method (annotation: *@roundfunction* as comment) in the input source. If the method can be found, it calls `transformFunctionToStandardForm` to apply the transformations discussed earlier in this chapter. It creates an `NLToolTranslator` and a `CodeTransformer` instance (starting at line 48) and calls the corresponding methods to start the code transformation. The method calls the ROSE `backend` method on line 60 to output the transformed version of the input code. This step is not required since the transformation is performed independent from the ROSE backend.

Code Example 5.20: `The code used to perform the transformation`

```
// ROSE translator example: identity translator.
//
// No AST manipulations, just a simple translation:
//
//    input_code > ROSE AST > output_code

#include "rose.h"
#include <string>
#include <AstInterface_ROSE.h>
#include "CommandOptions.h"
#include "LoopTransformInterface.h"
#include "TranslationUtils.h"
#include "NLToolTranslator.h"
#include "CodeTransformer.h"
#include "CodeGenerationUtils.h"
#include "boost/lexical_cast.hpp"

#define null NULL


int main (int argc, char** argv)
{
    //std::cout << "argc: " << boost::lexical_cast<std::string>(argc) << " argv: " << *argv << "argv+1: " << *(argv
       + 1) << std::endl;

    if(SgProject::get_verbose() > 0)
        printf("In main()");

    // Build the AST used by ROSE
    SgProject* project = frontend(argc, argv);
    ROSE_ASSERT (project != null);

    // Run internal consistency tests on AST
    AstTests::runAllTests(project);

    //fixing preconditions (main function present etc.)
```

```
    TranslationUtils::establishInputPreconditions(project);
37
    //finding round function
39  SgFunctionDeclaration* roundFunction = TranslationUtils::findRoundFunction(project);
    if(roundFunction != null)
41  {
        std::cout << "found round function " << roundFunction->get_name() << " at " << roundFunction->get_file_info
      ()->get_line() << " in " << roundFunction->get_file_info()->get_filename() << std::endl;
43
        //bringing function in standard form
45      TranslationUtils::transformFunctionToStandardForm(roundFunction, project);

47      //creating Transformator
        NLToolTranslator nlToolTranslator;
49      SgFunctionDeclaration* targetFunction =  nlToolTranslator.setupTranslation(project, "Norx", roundFunction);
        CodeTransformer transformer(&nlToolTranslator);
51      transformer.transformFunction(roundFunction, targetFunction);
    }
53  else
    {
55      std::cout << "could not find round function, aborting ..." << std::endl;
        return 0;
57  }

59  //transforming modified ast back to source code
    return backend(project);
61 }
```

5.6 Customization / Extension

The framework allows easy customization in terms of the integration of new tools as well as adoption of the transformation process. If additional transformation steps need to be applied, they can simply be performed before / after the standard format is established. These transformations can facilitate functions of the `TransformationUtils` class or use the ROSE framework methods directly. Although, it is important that the source code is in standard form before being transformed using the introduced mechanisms. To integrate new tools all we need to do is to implement a `ToolTranslator` class for that tool. Note that this is independent of the analysed cipher and will process any cipher reference implementation.

5.7 Application and Results

This section describes the application of the introduced framework on three submissions of the CAESAR competition in order to demonstrate it's function. It shortly describes the setting and goals of the analysis types and shows their results.

This example illustrates the usage of a toolchain consisting of a differential characteristic search using the extended CodingTool (described in Section 4.5.1), as well as a verification of the found characteristics using the IAIK NLTool (described in Section 4.5.2). Based on the steps and techniques described in Chapter 4 and Chapter 5, the framework parses the reference implementation of the ciphers into an abstract representation format and transforms it into the representations needed by the two tools in this toolchain. The analysis code then performs the characteristic search and verifies found code words using

the other tool. Note that this analysis code can be used to analyse arbitrary ciphers, since the representations required for the tools are generated automatically.

5.7.1 Analysis Types

The following briefly describes the different types of analysis, used for the demonstration of the automated analysis framework on the three cipher candidates:

a) **Forgery Analysis** The goal of this first forgery analysis is to find sparse and colliding state characteristics over several round that might allow conducting a forgery attack. Since the message words of the considered ciphers are injected into the rate of the state, the goal is to find colliding capacity parts before and after a certain amount of rounds R. To model this property, code shortening (in the context of a coding theory analysis) as described in Section 4.5.1, is applied on the generator matrix G. The capacity of the state is therefore forced to zero before and after the application of R round transformations.

b) **Round Analysis** In this basic analysis, the goal is to find sparse and valid characteristics over certain rounds of the cipher. Since the tool used for characteristic search in this toolchain is based on a coding theory analysis, random words of the state are forced to zero in order to narrow down the search space for the heuristic search algorithms. The information gained in such an analysis could, for example, later be used to construct attacks on the initialisation rounds.

5.7.2 Results

This section presents the results of the conducted analysis using the framework introduced in this thesis. It shows the hamming weight of the found characteristics over certain rounds R with a certain amount of state-bits s forced to zero. Note that these results only provide a demonstration of the introduced automated framework and has therefore only been performed with very small computational resources. The performance can be easily improved by conducting a more resource intense analysis. Further note that the introduced framework performs preliminary analysis and does therefore not claim high quality results as discussed earlier. Note that in the following tables, - donates weights of such a high value, not relevant to the analysis any more. Those values have been removed from the tables to improve the overall readability.

NORX

Since the focus of this thesis lies on the cipher NORX, both of the before mentioned analysis forms were performed to demonstrate the possibilities of the discussed toolchain. Since the rate of NORX is 6 state words, 768 ($= 2 \cdot 4 \cdot 64$) bits were forced to zero at

the correct positions. Table 5.1 shows the result of this analysis in form of the Hamming weights of found characteristics over different rounds of the round transformation.

R	1	2	3	4	5	6
$s = 768$	432	919	1417	1902	2398	2912

Table 5.1: NORX: The Hamming weight of the found (and verified) characteristics over different rounds R and s state bits forced to zero.

Table 5.2 shows the results of the round analysis performed on NORX.

s / R	1	2	3	4	5	6	7	8	9	10
0	12	893	1381	1880	-	-	-	-	-	-
64	12	888	1381	1868	-	-	-	-	-	-
128	12	900	1397	1868	-	-	-	-	-	-
192	12	901	1388	1880	-	-	-	-	-	-
256	15	906	1386	1869	-	-	-	-	-	-
320	15	892	1399	1872	-	-	-	-	-	-
384	15	908	1396	1894	-	-	-	-	-	-

Table 5.2: NORX: The Hamming weight of the found (and verified) characteristics over different rounds R and s state bits forced to zero.

MORUS and KETJE

For the two ciphers, MORUS and KETJE, only the round analysis was performed over several rounds R and with several state words s forced to zero. Table 5.3 illustrates the results of the round analysis performed on MORUS, while Table 5.4 shows the results for a similar analysis on KETJE.

s / R	1	2	3	4	5	6	7	8
0	3	6	11	19	32	53	3784	-
64	3	6	11	19	32	53	89	-
128	3	6	11	19	32	60	3784	-
192	3	6	11	19	32	71	3843	-
256	3	6	13	29	65	3092	3784	-
320	3	7	13	29	65	3150	3872	-
384	3	6	13	29	65	3184	3843	-

Table 5.3: MORUS: The Hamming weight of the found (and verified) characteristics over different rounds R and s state bits forced to zero.

s / R	1	2	3	4	5	6	7	8	9	10
0	21	84	640	823	1143	2984	-	-	-	-
16	21	115	593	823	1143	3013	-	-	-	-
32	24	183	692	864	1143	2934	-	-	-	-
48	37	85	652	842	1143	3149	-	-	-	-
64	23	85	680	823	1143	3149	-	-	-	-
80	22	85	891	813	1143	3149	-	-	-	-
96	31	96	891	719	1147	3149	-	-	-	-

Table 5.4: KETJE: The Hamming weight of the found (and verified) characteristics over different rounds R and s state bits forced to zero.

Chapter 6

Conclusion and Future Work

In this thesis, we introduced an automated framework for applying preliminary analysis on arbitrary authenticated ciphers. The framework takes a reference implementation - as typically delivered in the submission to a cryptographic competition - as input and performs an analysis using multiple tool chains fully automated. The framework parses the reference implementation, constructs an abstract cipher representation based on the code, brings it to a standard form and allows a user to implement adapters in order to transform it to any representation used by an analysis tool. The cryptanalyst using the framework implements the required adapters, writes code defining the analysis procedure, and can apply the analysis on an arbitrary number of different ciphers without any additional effort. This analysis can form the basis for further dedicated and more advanced analysis methods.

Since the focus of all analysis methods considered in this thesis is the analysis of authenticated ciphers, we introduced the basics of cryptography, described it's basic goals and introduced the reader to common techniques of how to ensure authenticity in symmetric cryptography. Further, we discussed the disadvantages of current solutions and showed the need for new dedicated authenticated cipher designs. Design approaches for constructing authenticated ciphers were discussed with a special focus on three submissions to the CAESAR competition. A detailed description was provided for the submission of the cipher NORX, since it was used to demonstrate the framework and it's components throughout the thesis.

We show the need for such an automated framework, especially in the context of recent and past cryptographic competitions, and gave scenarios of application as well as potential use cases. We described which steps cryptanalysis involve today, which effort is linked to the steps and how the automated framework can help to significantly reduce the effort of cryptanalysts going into preparing an automated framework for preliminary analysis.

The framework can perform this step of the analysis process fully automated on a po-

tentially large number of ciphers. This allows cryptanalysts to focus on designing and conducting dedicated cryptanalysis and attacks on primitives resulting in a generally higher quality of a large number of ciphers analysed.

We described the basics of cryptanalysis in general and different methods used to conduct it when focusing on methods used by tools implemented in the scope of this thesis. The focus of the methods here lies on the preliminary analysis of new authenticated cipher schemes using differential cryptanalysis. We further introduced how common analysis tools operate in order to find differential characteristics or verify found ones and explained how these tools can be combined in a toolchain to obtain useful results.

In addition, the integrated tools, the used framework and libraries as well as the design, architecture and procedures inside the tool were explained in detail. We showed how it is possible to construct such a framework solely based on a reference implementation, how the implemented algorithm is represented in an abstract format and which steps and transformations are required to allows it's translation to different cipher representations. We further show, how new tools can be easily integrated into the framework and how cryptanalyst can perform custom transformations and modifications on the abstract cipher representation based on the implemented functionality and the feature set provided by the framework facilitated underneath.

The results section of the thesis finally shows how the automated analysis framework was applied to three submissions of the CAESAR competition. We performed an automated preliminary analysis and facilitated a demonstration toolchain based on three tools used for the search and verification of differential characteristics. Therefore, we were able to demonstrate the viability of an easily extendible framework performing automated preliminary cryptanalysis on arbitrary authenticated encryption ciphers only operating on a C-reference implementation of the ciphers.

6.1 Scope and Limitations

It is essential to highlight that the goal of the implemented automated framework is to perform preliminary analysis. This means it applies different methods and tools to give a cryptanalyst hints into which direction potential dedicated analysis methods might be promising based on the results. So the results might, for example, show potential weaknesses in a certain components of the analysed cipher and a dedicated analysis could explore and possibly exploit these weaknesses. The framework or analysis performed does not raise the claim of providing usable differential characteristics to break ciphers. Rather it provides a tool that allows cryptanalysts to perform a preliminary analysis on a large set of ciphers to obtain information about the cipher and obtain indications for potential weaknesses. Therefore, the characteristics found by the framework will only act as a preliminary indicator rather than differentials useful for an attack.

6.2 Future Work

The goal of the framework implementation at this stage was to proof the feasibility of such a automated preliminary analysis framework. Therefore, this leaves space for future work and improvement potentially leading to great benefit for cryptanalysts. These improvements include:

- Application to other cryptographic primitives (this work focuses on authenticated encryption)
- Enable additional features for implemented tools (e.g. search instead of verify for SAT solvers)
- Extension to support other languages
- Add support for additional tools to search and verify characteristics.
- Actively support control flow structures in source functions
- Use the backend of the ROSE-framework to transcompile to other languages
- Enable the framework to search / verify characteristics in a distributed computing environment
- Improve program architecture for running on cluster structures
- and many more

The source code for all tools implemented during this thesis is available online and further development of the framework is highly anticipated and supported by the author.

Bibliography

[1] CAESAR call for submissions, final (2014.01.27). http://competitions.cr.yp.to/caesar-call.html. Accessed: 13.09.2014.

[2] Cryptographic competitions. http://competitions.cr.yp.to/. Accessed: 16.09.2014.

[3] Simple theorem prover. http://stp.github.io/stp/. Accessed: 19.10 .2014.

[4] J. Aumasson, P. Jovanovic, and S. Neves. NORX: parallel and scalable AEAD. In M. Kutylowski and J. Vaidya, editors, *ESORICS 2014*, volume 8713 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2014.

[5] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21:469–491, 2008.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. Assche. Cryptographic sponge functions. *Submission to NIST (Round 3)*, 2011.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. Assche. Permutation-based encryption, authentication and authenticated encryption. *Directions in Authenticated Ciphers*, 2012.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak specifications. http://keccak.noekeon.org/, 2009.

[9] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer. CAESAR submission: KETJEv1. 2014.

[10] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Duplexing the sponge: single-pass cryption and other applications. In *Selected Areas in Cryptography*, pages 320–337. Springer, 2012.

[11] A. Biryukov and V. Velichkov. Automatic search for differential trails in ARX ciphers. In J. Benaloh, editor, *CT-RSA 2014*, volume 8366 of *Lecture Notes in Computer Science*, pages 227–250. Springer, 2014.

[12] A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: Application to mceliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Transactions on Information Theory*, 44(1):367–378, 1998.

[13] Ecrypt. eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to/`. Accessed: 05.10 .2014.

[14] E. Fleischmann, C. Forler, and S. Lucks. Mcoe: A family of almost foolproof on-line authenticated encryption schemes. In *Fast Software Encryption*, pages 196–215. Springer, 2012.

[15] ISO/IEC. Information technology – security techniques – authenticated encryption. In *Information technology – Security techniques – Authenticated Encryption*, 2009.

[16] P. Jovanovic, S. Neves, and J. Aumasson. Analysis of NORX. *IACR Cryptology ePrint Archive*, 2014:317, 2014.

[17] L. R. Knudsen and M. Robshaw. *The block cipher companion*. Springer, 2011.

[18] S. Kölbl, M. M. Lauridsen, C. Rechberger, and T. Tiessen. Authenticated encryption zoo. `https://aezoo.compute.dtu.dk/doku.php`. Accessed: 05.10 .2014.

[19] G. Leurent. Arxtools: A toolkit for arx analysis. In *The Third SHA-3 Candidate Conference*, 2012.

[20] F. Mendel, T. Nad, and M. Schläffer. Finding SHA-2 characteristics: Searching through a minefield of contradictions. In D. H. Lee and X. Wang, editors, *ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 288–307. Springer, 2011.

[21] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[22] T. Nad. The codingtool library. workshop on tools for cryptanalysis 2010. `http://www.iaik.tugraz.at/content/research/krypto/codingtool/`. Accessed: 05.10 .2014.

[23] G. Procter and C. Cid. On weak keys and forgery attacks against polynomial-based MAC schemes. In S. Moriai, editor, *FSE 2013*, volume 8424 of *Lecture Notes in Computer Science*, pages 287–304. Springer, 2013.

[24] D. Quinlan, C. Liao, T. Panas, R. Matzke, M. Schordan, R. Vuduc, and Q. Yi. Rose user manual: A tool for building source-to-source translators draft user manual. `http://rosecompiler.org/ROSE_UserManual/ROSE-UserManual.pdf`. Accessed: 29.09.2014.

[25] D. Quinlan, M. Schordan, R. Vuduc, Q. Yi, T. Panas, C. Liao, and J. J. Willcock. Rose tutorial:a tool for building source-to-source translators. `http://rosecompiler.org/ROSE_Tutorial/ROSE-Tutorial.pdf`. Accessed: 29.09.2014.

[26] P. Rogaway and T. Shrimpton. A provable-security treatment of the key-wrap problem. In *Advances in Cryptology-EUROCRYPT*, pages 373–390. Springer, 2006.

[27] M. O. Saarinen. Cycling attacks on gcm, GHASH and other polynomial macs and hashes. In A. Canteaut, editor, *FSE 2012*, volume 7549 of *Lecture Notes in Computer Science*, pages 216–225. Springer, 2012.

[28] W. Trappe and L. C. Washington. *Introduction to Cryptography with Coding Theory (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[29] H. Wu and T. Huang. The Authenticated Cipher MORUS. `http://competitions.cr.yp.to/`, 2014.