Fabian Golser

Framework for Automated Cryptanalysis

Master's Thesis

Graz University of Technology

Institute of Applied Information Processing and Communications

Advisor: Eichlseder, Maria
Advisor: Dobraunig, Christoph Erwin
Advisor: Mendel, Florian

Graz, February 27, 2019

Statutory Declaration

Abstract

Cryptography in general is an important field. With the rise of computers it is more and more determining our lives. Cryptography is about protecting information, but with its manifold features also other meaningful goals, such as electronic commerce, digital currencies and secure authentication, can be achieved.

Research in this topic is indispensable to be able to provide state-of-the-art cryptographic algorithms, which ensure its defined attributes. Therefore, comprehensive competitions around the world are organized to find the best new cryptographic algorithms.

Every single new algorithm has to be evaluated and reviewed carefully to guarantee its quality. This is a demanding task, which needs various tools to fulfil it satisfactorily. Every tool needs as an input a specific version of the algorithm to be tested.

In this work, an existing framework is extended to analyse and translate a variety of cryptographic algorithms for cryptanalysis tools. The generated translations can be used as input for further analysis tools. Cryptanalysis deals with the analysis of cryptographic systems. Its goal is to get insight into their security, find weaknesses, or even break them.

This approach aims to accelerate the analysis of new algorithms. Especially the error susceptibility of human translation will be reduced because the whole translation process, which was done manually before, is automated.

Results show that it is possible to automatically translate reference-implementations of various cryptographic algorithms for further use with cryptanalysis tools. Although the resulting translations are not as efficient as manual translations could be, they serve as a good starting point for cryptologists.

Keywords: cryptography, differential cryptanalysis, analysis framework, hash algorithm, SAT solver

Table of Contents

List of Figures

List of Tables

List of Code Examples

List of Symbols

$\oplus$	Bitwise XOR operation
$\ll$	Left shift operation
$\gg$	Right shift operation
$\lll$	Left rotation operation
$\ggg$	Right rotation operation
$\wedge$	Logical AND operation
$\vee$	Logical OR operation
$\neg$	Logical NOT operation (*negation*)

Chapter 1

Introduction

1.1 Motivation

In this more and more computer-dependent society it is an important task to keep information protected. Cryptography is the field of study, which is concerned to achieve that goal. There are several ways to keep digital information secret. Since cryptography originated about 4000 years ago, many new cryptographic methods evolved [28]. Some methods experienced substantial adoption and are widely used nowadays.

It is not satisfactory for the reliable and useful exchange of information, to only protect the transmission by encryption. There are also other possibilities to corrupt information or its usefulness, such as an attacker who can modify the transmitted information without the senders' and receivers' knowledge. Encryption alone does not prevent this scenario.

Several properties need to hold to make the exchange of information reliable. The following four properties of digital information [38, p. 4], which can be achieved with cryptography, are the most important ones:

- **Confidentiality:** The information needs to be confidential, so that no third party can read it during transmission over an insecure channel. This property can be enforced by *encryption*, which will be explained in Section 2.6.

- **Authenticity:** The origin of information needs to be assured. Information can be worthless if the composer is not trustworthy or even verified. What is the value of information when there is no trust in its origin? This property can be enforced by *electronic signatures*, which will be explained in Section 2.10.

- **Integrity:** The information cannot be altered during its transmission. At first view, encrypting information before sending it is enough, but on further investigation it becomes obvious that this is not sufficient. Encrypted information can still be modified by an attacker, although it depends on the technique and effort taken, to make it look realistic. Therefore, it requires the strong need to make every modification of the transmitted information at least detectable for the receiver. This property can be enforced by *authenticated encryption*, which will be explained in Section 2.9.

- **Non-repudiation:** Non-repudiation prevents an entity from denying previous cryptographic actions or commitments. It is a property which can not be implemented with symmetric cryptography, since more entities would share the same key. Therefore asymmetric cryptography (Section 2.7) is needed. This property can be enforced by *electronic signatures* (Section 2.10) in combination with a Public Key Infrastructure (PKI), because a trusted third party is needed to resolve any dispute.

Specific *cryptographic primitives* need to be developed to achieve these properties. A cryptographic primitive is an algorithm implementing basic cryptographic properties such as one-way functions or pseudo-random functions. These primitives provide fundamental cryptographic properties to build up more complex constructs referred to as *cryptosystems* or *cryptographic protocols*. Cryptographic algorithms considered as cryptosystems provide more elaborate functionality such as key agreement or key exchange. The distinction between primitive and cryptosystem is not definitive. Nevertheless, the terms are used throughout this thesis as they have been defined above.

Various strategies have been examined over the years for the development process of these cryptographic primitives. One of them was to keep the algorithm secret and assure with this approach its security. It is called *security-by-obscurity.*

It turned out that this process is not a very good one in terms of security, because there are no possibilities to review the algorithm [26]. For example, if there is a weakness or in the worst case an error in the algorithm, no one has the possibility to identify it, because it is kept secret.

Security should rely exclusively on *Kerkckhoffs' principles* [29], stated by Auguste Kerckhoffs already in 1883. He proposed, among other things that "the system must be substantially, if not mathematically, undecipherable" and "the system must not require secrecy and can be stolen by the enemy without causing trouble".

Therefore, another process got popular, which is still used today to find new algorithms or cryptographic schemes. This process involves a public competition [42, 12] to find the best new algorithm for a specific purpose. The competition defines the rules and everyone can submit an own proposal of an algorithm. All participating algorithms are made available publicly and therefore, the possibility of reviewing them is given to everyone, no matter if participating in the competition or not. This is an enormous benefit, coming from the security-by-obscurity approach to the public competition-approach, in terms of security and efficiency for several reasons:

- The algorithm is better examined for errors when more people investigate it. Therefore, less errors or weaknesses remain in the chosen algorithm.
- There is limited possibility for an "evil" entity to knowingly design a weak algorithm, or to infiltrate weaknesses. Independent observers control each other.
- It can be faster to review the algorithms, because more helping hands are available. A company or institution cannot hire as much working capacities as the public can provide.
- The quality of the algorithm increases in general, because of the boost of available new ideas and different concepts from which the best can be chosen.

On the other hand, having more algorithms leads to some new challenges. One of these challenges is the problem that all the submitted algorithms have to be investigated carefully. Without a proper reviewing process there is no real gain in providing the possibility to do it.

This reviewing process takes some effort for sure and needs a lot of time for cryptanalysts to perform. The more new algorithm-proposals get submitted, the more analysis-work has to be done. It would be a beneficial issue to automate this process. On one side, it would speed up the reviewing process and on the other side the quality would remain equally distributed over all algorithms.

Unfortunately, it is not possible to automate the whole process. An algorithm can always be designed to fulfil the automated reviewing process requirements, but still be weak, no matter if intended or not. It seems to be infeasible to design a reviewing algorithm which can recognise every weakness reliably. Fortunately, at least some aspects of the reviewing process can be automated very well, which this work focuses on.

Since most cryptographic protocols rely on cryptographic primitives, it is important to keep them strong and secure. One of the primitives which is widely used in cryptographic protocols are cryptographic hash functions portrayed in Chapter 3.

A hash function is a special version of a one-way function designed to map input data of arbitrary size to output data of fixed size. The resulting output is called *hash*. Another important property of a hash function is its infeasibility to find two different input strings which map to the same output string. This is called collision resistance.

Although the mentioned framework is developed with a general approach to analyse a broad range of round-based algorithms, its main focus still refers on hash algorithms. This work focuses on hash algorithms and their automated review.

1.2 Goals

The goal of this project is to develop a framework for automated cryptanalysis like shown in Figure 1.1. There exists already a framework developed by Hechenblaikner [25] designed as a proof of concept, which is used as a basis for the current project. The framework uses an input file, processes it and generates an output file.

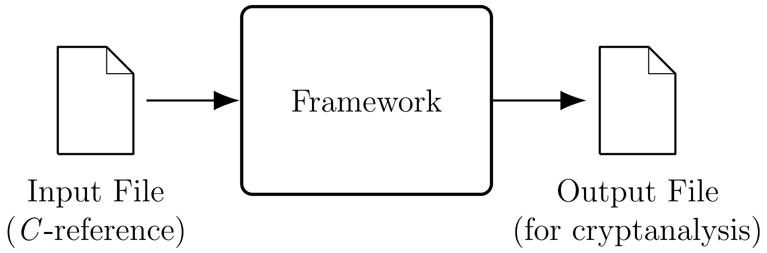

Figure 1.1: *Schematic of the automated analysis framework:* The input file is read by the framework. The framework analyses it and computes an output file for a specific analysis tool from it.

A special representation of a cryptographic algorithm has to be used as an input file. It is called C-reference implementation. A reference implementation has the intention to show the main concepts of the algorithm rather than being optimized for some criteria, like for example hardware optimizations. The typical procedure to submit an algorithm to a cryptographic competition is a C-reference implementation. Therefore, it is also used by this framework.

The reference implementation has to be annotated before it can be analysed by the framework. The annotation is done in standard *C*-comments, seen in Code Example 1.1.

```c
// @roundfunction
// @statevariable = state
// @statesize = 4
// @messagevariable = x
// @messagesize = 16
// @statewidth = 32
void roundFct(int state[4], int x[16])
{
    /*
    ...
    this is an example round function
    ...
    */
}
```

Code Example 1.1: *Example of a round function with annotations:* The annotations have to be made within standard *C*-comments, starting with a @-character. Some possible annotations are able to be seen in the code example. Some of them are obligatory, like @$roundfunction$ and some of them are optional, like @$messagevariable$.

The input file, and if required also the differential characteristic-file (Section 4.4.5), can be processed by the framework. It reads both files and translates the cryptographic algorithm into an intermediate language. This language is the ROSE Intermediate Representation (ROSE IR) internally based upon *SAGE III*, a language defined by the *ROSE-compiler framework* [45, 34]. This framework, developed at Lawrence Livermore National Laboratory (LLNL), is presented in Section 5.2. In this representation several transformations, described in Section 5.5, are performed.

With the final processing step the algorithm gets translated into language for a desired output format, such as a boolean logic language called *CVC* to be processed by a Satisfiability Modulo Theories (SMT) solver like the Simple Theorem Prover (STP) [22]. The algorithm can be translated into many different languages or formats serving different tools for further processing.

1.3 Current Work

In this Section the current state of research is summarized. At the moment, every single algorithm which needs to be tested, has to be translated manually for all the cryptanalysis tools. Therefore, this project tries to improve the current situation and replace the error-prone manual translation. Also, a speed-up of the cryptographic analysis process could be achieved.

There exists a proof of concept to design such a framework [25]. This project is a follow-up to the existing project, with the goal to expand its functionality and make it more general. Furthermore, it should be possible to process some real-world problems like the final round candidate algorithms from the last hash-competition [42]. Additionally, it should allow the processing of their differential descriptions to also have the possibility to perform differential cryptanalysis.

1.4 Contributions and Results

The framework developed along with this thesis can be used to automatically read and parse cryptographic algorithms for further analysis tools. As concrete example the *MD4* hash algorithm reference implementation form the Request for Comments (RFC[1]) was parsed into the *CVC* language. The generated output file can be used with a *SMT* solver like *STP* to look for differential characteristics. The results from this example and also some other results can be found in Chapter 6. In the following a list of the most important contributions of this thesis is shown:

- A new translator plug-in for the *CryptoSMT Tool* was developed.
- The framework was extended to read characteristic files and create template characteristics for different algorithms.
- A differential description can be created for selected operations.
- The framework can now handle some real world cryptographic examples, not only dummy code.
- The cryptographic algorithms *MD4* and *Ascon* can be translated successfully for two different cryptanalysis tools, the *NL Tool* and the *CryptoSMT Tool*.

1.5 Outline

All relevant topics will be described in more detail in the following Chapters. This current part gives an outline for the main part of the thesis. In the beginning the basics will be discussed and the reader will be introduced into the terminology of the topic. Gradually, the reader can go more into specifics,

[1]`https://www.rfc-editor.org/rfc-index.html`

regarding selected parts until the main Chapter of the thesis is reached. In this Chapter, the reader will get familiar with the framework built along with this thesis.

Chapter 2 is about **cryptography** in general, so relevant cryptographic concepts, some examples and important aspects will be explained with the intention that more detailed concepts in further Chapters can be understood.

The main aspect of this work focusses on **cryptographic hash functions**. Therefore, cryptographic hash functions will be defined in more detail in Chapter 3. It includes the questions what they are and for what they are needed. Some cryptographic hash functions are selected as examples to show the reader their need.

In Chapter 4 the topic of **cryptanalysis** will be discussed. It is about how cryptographic algorithms can be analysed, manually and automatically. There, the basic methods used for further analysis within external tools will be explained. Especially differential cryptanalysis is discussed, which is one of the most powerful analysis-methods for cryptographic algorithms, so it is a major topic of this thesis. It will be explained in general and also with a specific example, so the reader can follow its principles.

In Chapter 5 the **automated cryptanalysis framework** will be introduced. Among other things, implementation-specific details about it will be discussed. First of all an overview of the whole framework will be given and afterwards, special parts will be described in more detail.

Some **results** of the project are shown in Chapter 6. This includes differential characteristics generated by the framework, found hash-collisions and output files for further computations. These output files can be used as input files for the subsequent analysis programs. Some of the problems arising during the project are elaborated and their solutions discussed. For the progressing development of the framework this part is of special relevance, because the main problems occurring during the implementation of the framework should be a lesson for further development.

Chapter 7 will draw a **conclusion** about the project. Possibilities and limitations of the framework and the used processing method in general are shown. We provide some proposals for future work. In supplementation, ideas for additional features and improvements for the implementation are collected here.

Finally, the **references** list all the papers and projects which influenced this project, so it is a good starting point for further reading.

Chapter 2

Cryptography

2.1 Overview

In this chapter, the reader will be introduced to the topic of cryptography. First, important terms relating to cryptography are defined in Section 2.2. The history of cryptography will be reviewed shortly in Section 2.3, where some conclusions will be drawn out of the historic evolution. Examples on how to plot cryptographic schemes and interactions within cryptographic protocols will be given in Section 2.4, with the entities *Alice* and *Bob* typically used in cryptographic contexts, shown in Figure 2.1.

Afterwards, different encryption types and other cryptography related concepts will be explained, which are needed in the following to understand this work. Therefore, examples of specific algorithms and their encryption schemes in the Sections 2.6, 2.7 and 2.9 are given.

Hash algorithms (Section 2.8) are shortly introduced but explained in more detail later in a separate Chapter 3. In the end, some special topics are treated, like cryptographic competitions in Section 2.9.3 and electronic signatures in Section 2.10.

2.2 Terminology

Some terminology has to be defined, before we can move on to further sections, to have a common language:

Cryptography is about hiding information. The word *cryptography* comes from the greek words κρυπτός kryptós *hidden, secret*, and γράφειν graphein, *to write.* When we talk about cryptography, the term *encryption* can not be left aside. Encryption means the process of encoding information, so that it is not possible to read it without permission. Usually, an algorithm is provided with the message to be encrypted and a secret *key*, which is used to protect the information. This secret key, which is normally talked about in a cryptographic context, is a secret word or phrase protecting the information. An encryption algorithm encrypts information with a secret key and only the entities possessing the key can access the information.

Encrypted information can be *decrypted* to gain access to the secret information again. Usually the same algorithm and sometimes also the same key is used to recover the information. Nevertheless, there also exist circumstances where a different key is used to encrypt and to decrypt information. The same applies for the algorithm involved in the encryption and decryption process.

Information, which is not encrypted yet, is called *plaintext*. After the encryption process, when it can not be read any more, it is called *ciphertext*. A cryptographic algorithm can also be called *cipher* in some settings.

2.3 History of Cryptography

Information for this section was taken from [28]. Cryptography has a long history. Records go back to around 1900 B.C. in ancient Egypt, where cryptography was used first, as far as we know. It can not quite be compared with modern cryptography as it is used nowadays, but it incorporated a deliberate transformation of the writing, which is one of the essential elements of cryptography. It is the oldest text known to do so [28, p. 64ff].

Also the Greeks and Romans have used cryptography, but mostly for military purposes. The Spartans for example developed the first system of military cryptography as early as 500 B.C. It included a cryptographic device called *skytale*, which is a cylinder with a strip of parchment wound around (transposition cipher).

The Romans developed the *Caesar cipher*, which is a *shift cipher*, a type of *substitution cipher*, where all letters are shifted by three letters. Also many other peoples and cultures developed variations of these cryptographic concepts.

For thousands of years cryptography was more like what might be called *classic cryptography*, which is based on simple mathematical constructions. Only in the last decades, since the 20th century, more sophisticated mechanical and electronic machines were developed which made it possible to increase the cryptographic complexity. The increase of computational power with the raise of computers has continuously allowed schemes of still greater complexity, which are not suited to pen and paper any more.

Both world wars were under a big influence of cryptography and cryptanalysis. Battles were decided in favour of the code-breaking party and history was written depending on it.

In modern times cryptography is based on mathematical theory and computer science. In theory ciphers are breakable, but designed around the *computational hardness assumption* and therefore still infeasible to break in reasonable time.

2.4 Alice and Bob

If it comes to cryptography, we usually talk about two parties that try to communicate with each other and an adversary. *Alice* and *Bob* are the two communicating parties and *Eve* tries to intercept the communication. An exemplary situation can be seen in Figure 2.1. In the following *Alice* is replaced by A, *Bob* is replaced by B and *Eve* is replaced by E for convenience.

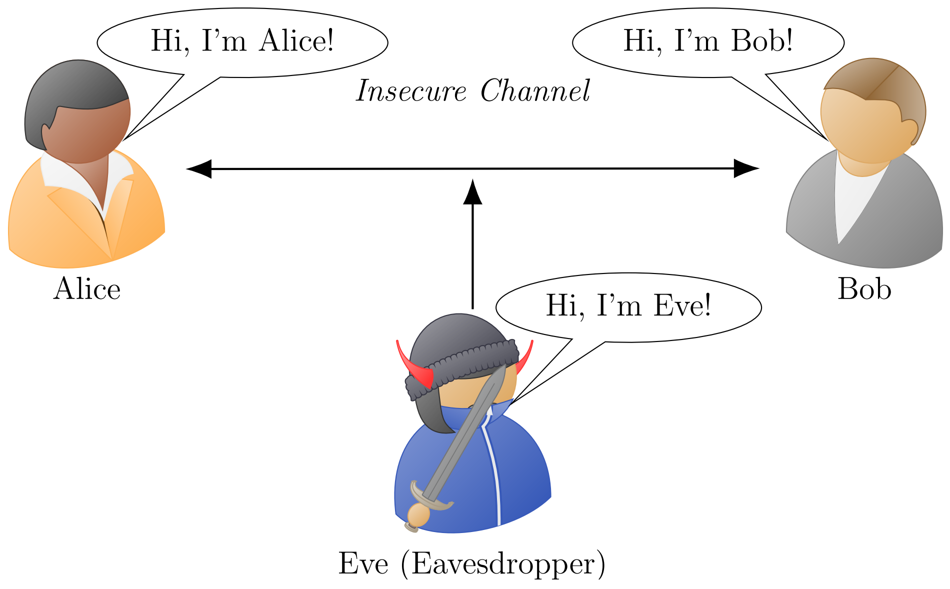

Figure 2.1: *Alice, Bob and Eve:* The typical diagram explaining entities and their connection in a cryptographic setting. In this example *Eve* is intercepting the private communication between *Alice* and *Bob* without them knowing.

If A and B communicate over an *insecure channel*, E can listen on the whole communication between the two without them knowing. E can also for example capture a message from A, modify it and forward the modified message to B. So E would be able to control the communication between them completely.

Encryption is one measure to prevent other entities listening on an insecure channel. If all messages A sends to B are encrypted, E is not able to listen to their communication. Still, E is able to intercept messages and delete or modify them, but without the possibility to read the information inside the messages. To protect the communication between A and B from other attacks than just reading the communication, more advanced cryptographic methods must be applied. Some of these are treated in this chapter.

In Figure 2.2 an example of a cryptographic protocol is shown. A and B are the two communicating entities. Their communication is shown over time in a chronological order. The two arrows from their names down respectively represent the time axis. The arrows which are drawn horizontally represent the messages sent from A to B and back.

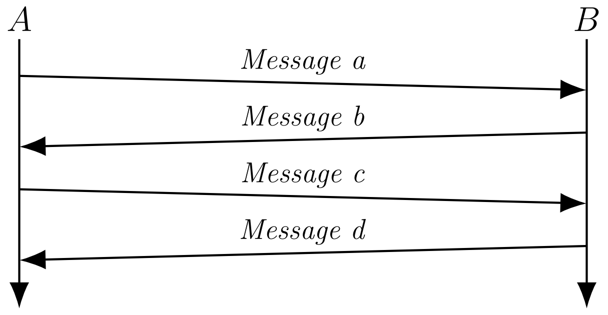

Figure 2.2: *Example of a cryptographic protocol:* A and B communicate with each other, the messages they send are shown in chronological order.

2.5 Encryption in General

Modern cryptography relies on computational hardness of publicly available algorithms instead of secret algorithms like in past times. Some algorithms also include provable security, meaning it is shown that an attacker has to solve the underlying hard problem in order to break its security. This aspect makes the algorithms trustworthy to the public. No single organisation is able to tamper with the encryption if the algorithm is applied correctly.

2.6 Symmetric Cryptography

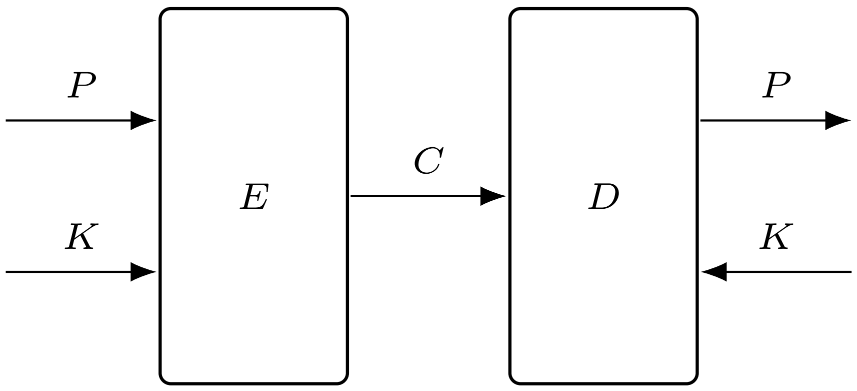

Figure 2.3: Symmetric encryption.

Symmetric encryption (Figure 2.3), also called secret key encryption, is one of the basic forms of encryption. The word symmetric is related to the encryption key K. It means that for encryption and decryption the same key K is used. Therefore, the key has to remain secret in order to keep the encrypted plaintext P protected.

Encryption: The plaintext P gets encrypted with the encryption operation $E_K(\cdot)$ using the key K. As a result, it returns the encrypted plaintext as ciphertext

$$C = E_K(P). \tag{2.1}$$

Decryption: The decryption operation $D_K(\cdot)$ with the same key K is the inverse operation to the encryption operation. It takes the ciphertext C as input and returns the original plaintext

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

2.7 Asymmetric Cryptography

Asymmetric encryption (Figure 2.4), also called public key encryption, is one of the basic forms of encryption. The word asymmetric is related to the asymmetry of the encryption and decryption operation. It means that for encryption and decryption, different keys K_{pu} and K_{pr} are used. The public key K_{pu}, which is available to the public, is used for the encryption operation and the private key K_{pr}, which is kept secret, is used for the decryption operation. This setting enables every entity to encrypt messages for a desired entity, but only the desired entity can decrypt and read them. This is an advantage in comparison to symmetric encryption, but it comes with the drawback that its cryptographic operations are slower.

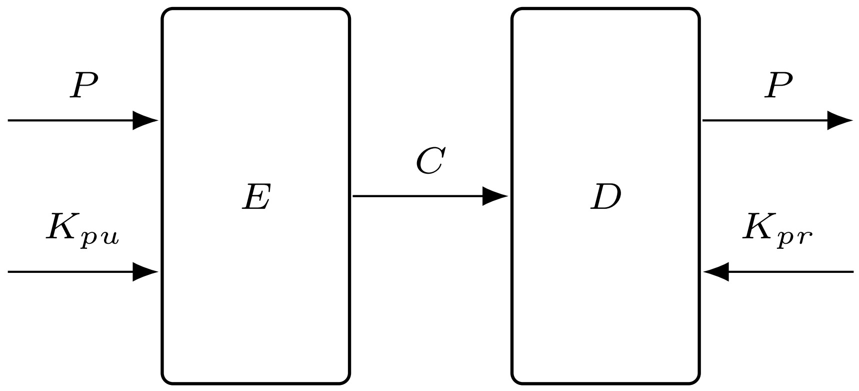

Figure 2.4: Asymmetric encryption.

Encryption: The plaintext P gets encrypted by the encryption operation $E_{K_{pu}}(\cdot)$ using the public key K_{pu}. As a result, it returns the encrypted plaintext as ciphertext

$$C = E_{K_{pu}}(P). \tag{2.3}$$

Decryption: The decryption operation $D_{K_{pr}}(\cdot)$ with the private key K_{pr} is the inverse operation to the encryption operation. It takes the ciphertext C as input and returns the original plaintext

$$P = D_{K_{pr}}(C) = D_{K_{pr}}(E_{K_{pu}}(P)). \tag{2.4}$$

2.8 Hash Functions

2.8.1 Overview

A hash function is a function that maps input data of arbitrary size to output data of fixed size. It takes an input message M of arbitrary length and scrambles it. A fix sized output value y, called *hash*, is generated. Internally it works like a “one-way” function, because it is easy to compute the hash but hard to invert it. A schematic picture of a hash function is shown in Figure 2.5.

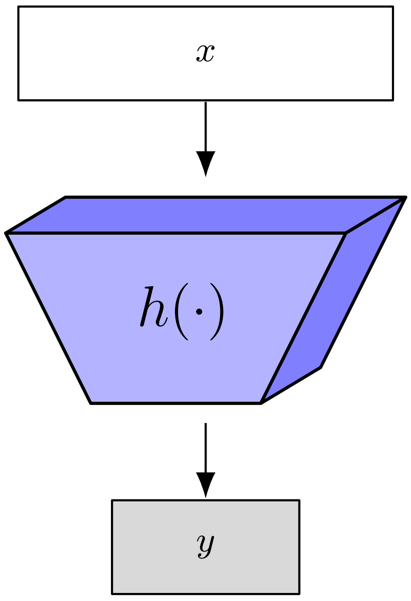

Figure 2.5: *Hash function:* The input message $x \in X$ can be of arbitrary length. The hash function $h(\cdot)$ generates the hash value $y \in Y$ of fixed length out of x.

The focus of this thesis will be on cryptographic hash functions, which are a subclass of general hash functions. They differ from each other in their properties. Therefore, hash functions will only be introduced briefly here. More details on cryptographic hash functions will be given in Chapter 3.

2.8.2 Usage of a Hash Function

Hash functions have many different use-cases. Some of them are in cryptographic applications of course, but there are also use cases outside of the cryptographic field. Only a few are described here for illustrative purposes.

- **Checksums:** Hash values can be used as a checksum for digital information. For example, when downloading a file from the Internet, sometimes a checksum can be found on the web-page to verify if the download was correct.
- **Hash Map:** A hash map is a data-structure, which is used in several programming languages such as in C++. This structure allows the programmer to store data in the format $HashMap < Key, Value >$ with an average access time complexity of $\mathcal{O}(1)$.
- **Git commit IDs:** A *GIT* commit id is a *SHA-1* hash over all the important data of a commit. It is used to make the verification process

of a GIT repository faster by just comparing the commit ids instead of its corresponding data.

2.9 Authenticated Encryption

2.9.1 Overview

Authenticated encryption, shown in Figure 2.6, is a special form of symmetric encryption. The word authenticated is related to the fact that the encrypted message can be verified after the decryption operation. An authentication tag is generated e.g. with a hash function along with the encryption operation. This tag is used within the decryption operation to verify the authenticity of the message. Not like electronic signatures, which provide *integrity* and *non-repudiation*, authenticated encryption implements *confidentiality* and *integrity.*

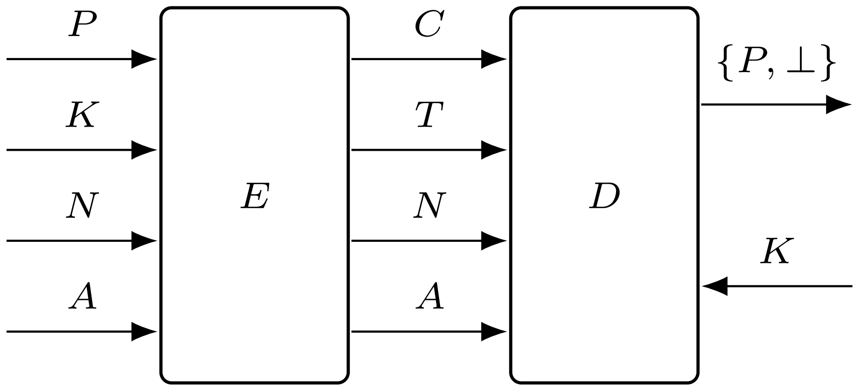

Figure 2.6: Authenticated encryption.

Encryption: The plaintext P gets encrypted by the encryption operation $E_K(\cdot,\cdot,\cdot)$ using the key K, a nonce N and some associated data A. As a result, it returns the encrypted plaintext as ciphertext C and an authentication tag T

$$C, T = E_K(N, A, P). \tag{2.5}$$

Decryption: The decryption operation $D_K(\cdot,\cdot,\cdot,\cdot)$ with the key K is the inverse operation to the encryption operation. It takes a nonce N, some associated data A, the ciphertext C and the authentication tag T as input and returns the original plaintext

$$\{P, \perp\} \ni D_K(N, A, C, T) = D_K(N, A, E_K(N, A, P)) \tag{2.6}$$

if the verification process of the tag is correct or $\perp$ if the verification fails.

2.9.2 Generic Composition

There are three commonly used approaches to authenticated encryption. Encrypt-then-MAC (EtM), Encrypt-and-MAC ($E\&M$), and MAC-then-Encrypt (MtE). They will be described in the following.

Encrypt-then-MAC (*EtM*)

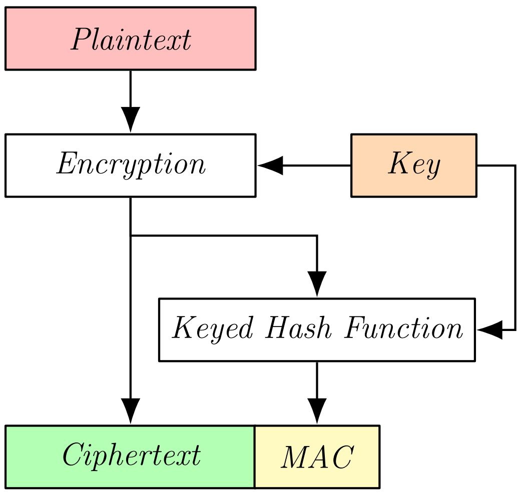

Figure 2.7: Encrypt-then-MAC (*EtM*).

This approach, shown in Figure 2.7, is the standard method according to *ISO/IEC 19772:2009*[1]. It is the only method which can reach the highest definition of security in authenticated encryption, but this can only be achieved when the MAC used is "strongly unforgeable" [4]. Therefore, it is used e.g. in *Internet Protocol Security* (IPsec), a secure network protocol to encrypt and authenticate packages, or in *Transport Layer Security* (TLS), a cryptographic protocol to provide communications security.

[1]`https://www.iso.org/standard/46345.html`

Encrypt-and-MAC (*E*&*M*)

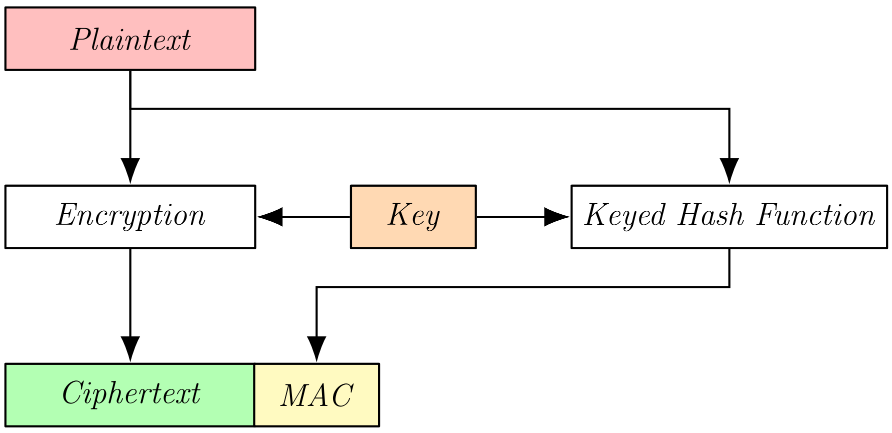

Figure 2.8: Encrypt-and-MAC (*E*&*M*).

This approach, shown in Figure 2.8, is used e.g. in *Secure Shell* (SSH[2]), although the scheme has not been proved to be strongly unforgeable in itself [4].

[2]`https://tools.ietf.org/html/rfc4251`

MAC-then-Encrypt (*MtE*)

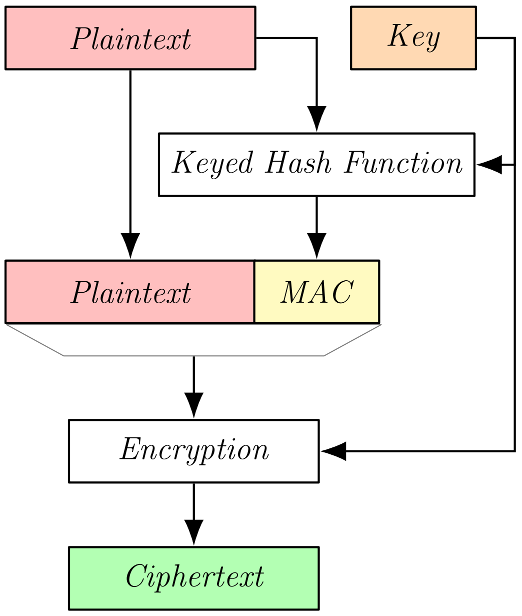

Figure 2.9: MAC-then-Encrypt (*MtE*).

Also this approach, shown in Figure 2.9, has not been proved to be strongly unforgeable. It is used in *Secure Sockets Layer* (SSL/TLS)[3] where in the recent past a padding error led to a successful padding oracle attack, called *Lucky Thirteen* [1] against it.

2.9.3 CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness

The *CAESAR-competition* [12] is a cryptographic competition trying to find a portfolio of authenticated ciphers. On February 20, 2019 the final portfolio was announced by the *CAESAR selection committee*. One of the candidates is *Ascon*, which is described in Section 2.9.4. *Ascon* was selected as first choice for use case 1: "Lightweight applications (resource constrained environments) ".

[3]deprecated, predecessor version of *TLS*

2.9.4 Ascon

Ascon [17] is an authenticated encryption algorithm. Its encryption operation can be seen in Figure 2.10 and the bitsliced implementation of the *S-Box* can be seen in Figure 2.11. There are many more algorithms, but here only one is listed exemplary for the whole class of authenticated encryption algorithms. The reason why this one is listed here, is because it is the only authenticated encryption algorithm which C-reference implementation is processed by the framework within this thesis. All the other processed algorithms are hash algorithms.

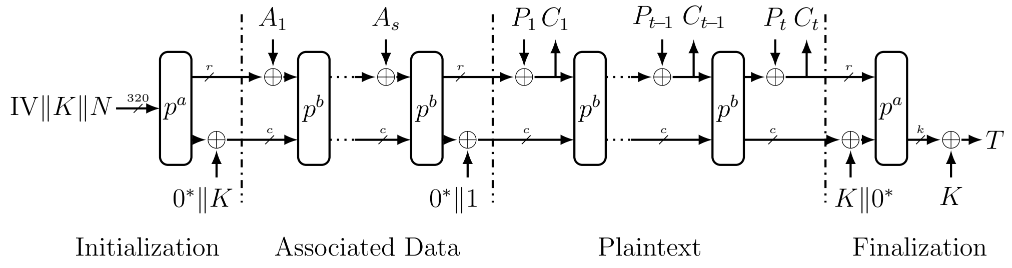

Figure 2.10: *Ascon* encryption. Figure from [27].

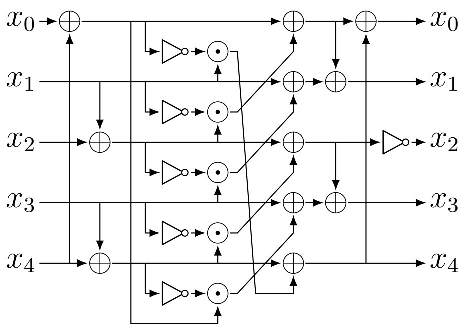

Figure 2.11: Bitsliced implementation of the 5-bit *Ascon S-Box*. Figure from [27].

2.10 Electronic Signatures

Electronic signatures are a useful tool for modern society. According to [52] they are an important component for various e-government solutions.

Since the European Parliament defined the Directive 1999/93/EC, which defines the properties and requirements of electronic signatures, their importance increased significantly [24]. With this directive, electronic signatures gained the legal equivalence to handwritten signatures. To ensure the same properties of handwritten signatures in the digital world, electronic signatures combine many cryptographic primitives to a bigger cryptosystem.

The properties *authenticity*, *integrity* and *non-repudiation of origin* have to be fulfilled to compete with handwritten signatures. These properties are already described in Section 1.1.

First, a *hash* algorithm is used to compute a cryptographic checksum over a document (information/data). Second, the hash value is signed (*authenticated with a public key algorithm*) with the private key belonging to the signer's certificate.

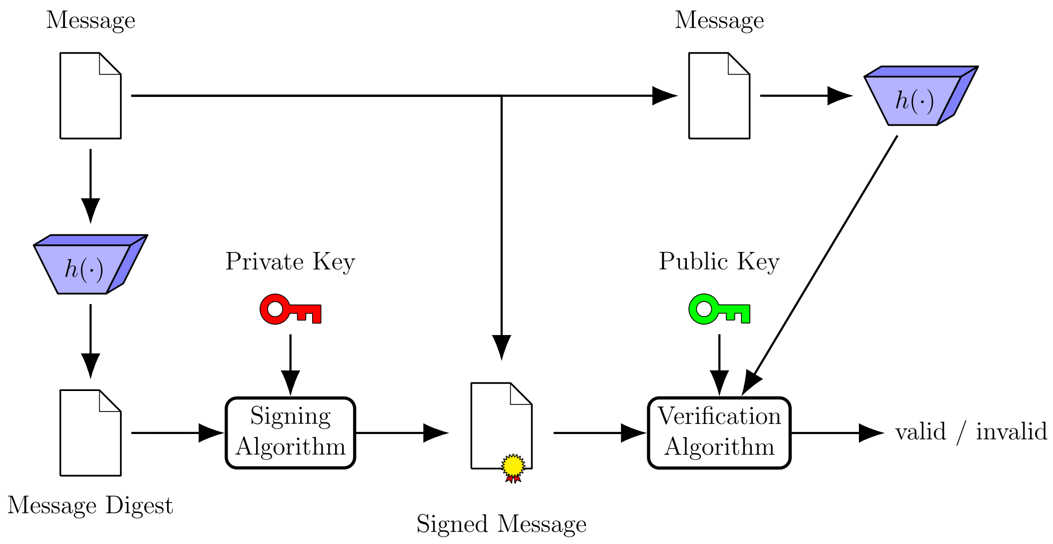

Figure 2.12: *Electronic signature scheme*: The message gets signed with the sender's private key, transmitted and verified with the sender's public key. If the original message digest and the new message digest are equal, the message's signature is valid. Otherwise the signature is invalid.

In Figure 2.12 the process of signing a message, transmitting it and verifying it on the receiver's side can be seen.

Chapter 3

Cryptographic Hash Functions

3.1 Overview

This chapter is about cryptographic hash functions, their composition and application. At first, an overview of the chapter will be given. Thereafter, the general functionality will be explained. As soon as it is made clear how cryptographic hash functions work, their usage and application in practice will be examined. For this purpose, some examples with common cryptographic hash function use-cases will be given. The security properties are elaborated carefully, because this is one of the main contents of this thesis. At last, some chosen algorithms are reviewed in more detail. This information is needed in the following to understand the assumptions made in Chapter 4 about cryptanalysis.

There are many different cryptographic hash functions in the field. Some of them are widely used, others not. The most important ones are discussed in Section 3.6. Starting from this point, only cryptographic hash functions are discussed. If at some point only the word "hash" or "hash function" occurs, it is always about cryptographic hash functions if not stated differently.

3.2 General Functionality of a Cryptographic Hash Function

A cryptographic hash function is a function that maps input data of arbitrary size to output data of fixed size. In this regard, they work the same as general hash functions described in Section 2.8. The main difference is that a cryptographic hash function has additional properties related to its security, which are described in Section 3.4.

A cryptographic hash function is designed to be a "one-way" function, this means it should be "infeasible" to invert the function. More accurately, it is fast to compute a hash of input data and "infeasible" to compute the original data from the hash value. What exactly "infeasible" means is described in Section 3.4. An ideal hash algorithm should not be able to be inverted faster than to brute-force it. Therefore, the computational effort should be as high as about 2^n, where n is the length of the hash value in bits.

Another property hash functions implement is the *avalanche effect*. This property states that if the input value changes slightly, the output value changes significantly. Even a single bit-flip in the input should generate a significantly different output.

3.3 Usage of a Cryptographic Hash Function

Cryptographic hash functions have many different use-cases. Some of them are described here for illustrative purposes. Use-cases of general hash functions are described in Section 2.8.2.

- **Digital Signatures:** Hash algorithms are a critical part in the process of creating digital signatures. A hash value over the data to be signed is computed. This value is signed with the key hold by the owner of the corresponding certificate.

- **Bitcoin Blocks:** Bitcoin ₿[1] is a cryptographic currency, or cryptocurrency, which is built upon the one-way property of cryptographic hash functions [40]. A peer-to-peer network timestamps transactions by hashing them into an ongoing chain of blocks.

- **Protecting data:** A hash value can be used to uniquely identify secret information. This requires that the hash function is collision-resistant, which means that it is very hard to find data that will result in the same hash value. Collision resistance is accomplished in part by generating very large hash values. For example, *SHA-1*, still one of the most widely used cryptographic hash functions, generates 160-bit values.

3.4 Security Properties

A cryptographic hash algorithms' security can be measured along with its three important security properties *pre-image resistance* 3.4.1, *second pre-image resistance* 3.4.2 and *collision resistance* 3.4.3. If all three properties

[1]`https://bitcoin.org/`

are fulfilled, the algorithm is treated as secure. Nevertheless, also other properties, even stronger ones which can be seen in Section 3.4.4 can be defined for an algorithm. To understand the explained security properties, hash algorithms have to be described first. The mathematical description of a hash algorithm is defined by [41] as followed. A hash function is a mapping

$$h : X \to Y, \tag{3.1}$$

with $X = \{0,1\}^*$ and $Y = \{0,1\}^n$ for some fixed $n \in \mathbb{N}^+$. Let $x \in X$ and $y \in Y$, then it is defined as $h(x) = y$. x is called *preimage* of y, where x corresponds to the input message and y to the output value called *hash value* or *digest* of the length n.

The security properties described in the following are taken from [38, p. 323f].

3.4.1 Preimage Resistance

For a given output value y of a hash function $h(\cdot)$ it is infeasible to find any message x with $h(x) = y$. Infeasible means, the complexity is as high as applying a brute-force attack, which is $\mathcal{O}(2^n)$.

3.4.2 Second Preimage Resistance

For a given message x_1 it is infeasible to find any second message $x_2 \neq x_1$ with $h(x_1) = h(x_2)$. Infeasible means, the complexity is as high as applying a brute-force attack, which is $\mathcal{O}(2^n)$.

3.4.3 Collision Resistance

It is infeasible to find any pair of messages $x_1 \neq x_2$ with $h(x_1) = h(x_2)$. Infeasible means, the complexity is as high as applying a brute-force attack with on average $2^{n/2}$ possible tries, corresponding to $\mathcal{O}(2^{n/2})$. The attack targeting collision resistance is also called *birthday attack*.

3.4.4 Stronger Security Properties

Some hash functions, which rely on a sponge construction, try to state their security not through the previous three properties, but are referring to a different property to avoid the need to fulfil these. The approach states that the hash function should behave like a *random oracle*, which is a theoretical black-box with ideal behaviour. This random oracle responds to each unique query with a *true random* answer. Because of its ideality no conclusion can be drawn, which input message was used. Therefore, it is suited for cryptographic applications.

3.5 Basic Building Blocks

Hash algorithms, like most other algorithms are built from *basic building blocks.* Some of these basic blocks, which are used later when writing about specific algorithms in Section 3.6 are explained here. The most important building blocks from which hash functions are built are *permutations* 3.5.1 and *compression functions* 3.5.2. Additionally, for hash functions there are different modes of operation described in Section 3.5.3.

3.5.1 Permutations

A permutation is a function $\sigma\colon X \to X$ that maps a finite set X onto itself, in such way that for each $y \in X$ there exists exactly one $x \in X$ such that $\sigma(x) = y$. Permutations are operations like *substitution-boxes* 3.5.1.1 or a *linear mixing layer* 3.5.1.2.

3.5.1.1 Substitution-Box

A substitution-box, short *S-Box*, is used as a basic building block in cryptography. It takes a number of input bits m and transforms them into a number of output bits n. In general it is a non-linear substitution operation. There are static *S-Boxes* like in *DES* and *AES*, which can be efficiently implemented with a lookup table, or dynamically generated *S-Boxes* derived from the secret key, which, in return, make cryptanalysis more difficult. Also for modern hash algorithms *S-Boxes* are often used, like e.g. in *Grøstl* (Section 3.6.3.5).

3.5.1.2 Linear Mixing Layer

A linear mixing layer can be a part of a hash function or an other cryptographic primitive with the purpose of scrambling bits. It works similar to an *S-Box* with the main difference that an *S-Box* is a non linear operation and a linear mixing layer is a linear operation. The linear mixing layer has a larger input space and consists only of linear bit-operations like $\oplus$, $\ll$, $\gg$ and $\lll$, $\ggg$.

3.5.2 Compression Functions

The compression function is the part of a hash function which reduces the number of bits. Because of the integral property that a hash function maps an arbitrary number of bits to a defined number of bits, every hash function needs to compress.

3.5.3 Modes of Operation

There are different modes of operation like the *Merkle-Damgård design* (Section 3.5.3.1) or the *sponge construction* (Section 3.5.3.2). Also others exist, but are not treated within this work e.g. the *HAIFA* construction [9].

3.5.3.1 Merkle-Damgård Design

The *Merkle-Damgård design* shown in Figure 3.1 is a specific design concept for hash functions which provides *collision resistance inheritance.* This means a hash function in *Merkle-Damgård design* is collision resistant if its compression function is collision resistant [39]. With this approach the security analysis can be focused on the smaller compression function, where the main focus lies on. The *Merkle-Damgård structure* was therefore applied to many popular hash algorithms like *MD4*, *MD5* and the *SHA-2* family.

In general it works in three steps. First, the input is padded and gets split into blocks m_i of equal length. Second, the compression function f is applied iteratively with the output h_{i-1} of the previous iteration and a new block m_i as inputs to produce the new intermediate state h_i as shown in Figure 3.1. And third, an optional post-processing step is applied.

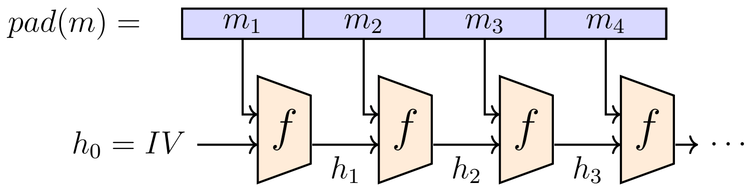

Figure 3.1: *Schematic of the Merkle-Damgård design:* The blocks get combined iteratively. Figure from [27].

3.5.3.2 Sponge Construction

The sponge construction is one of the modes of operation used for hash algorithms, but also for other cryptographic constructions like the stream cipher *Ketje* [7] or *Keyak* [8].

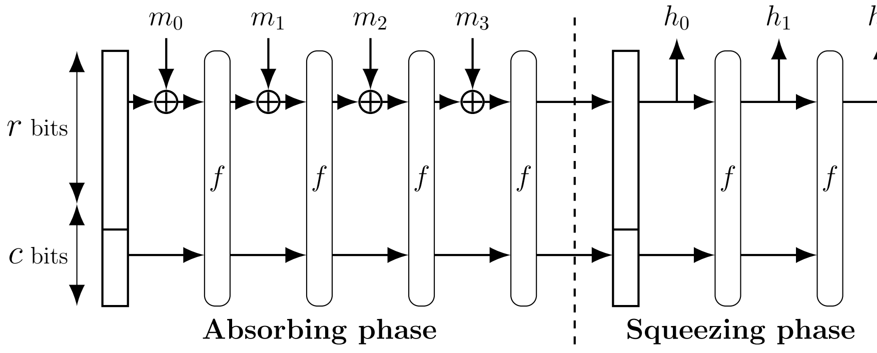

Figure 3.2: *Schematic of a sponge construction.* Figure from [27].

A sponge construction, also called sponge function, seen in Figure 3.2, uses its internal transformation f to map input of arbitrary length to output of arbitrary length. It is similar to the *Merkle-Damgård design* with the difference that its transformation function f does not compress but the bit-operation combining internal state and message block does.

The sponge function consists of an absorbing phase and a squeezing phase. During the whole processing the input and output state of f is divided into the c-bit inner state and the r-bit outer part. The absorbing phase reads the input message m divided into equally sized blocks $m_0 \dots m_i$ and *XOR*es them into the outer state r. In the squeezing phase the outer state is appended to the hash output $h_0 \dots h_n$ until the desired length is reached.

3.6 Algorithms

There are many different hash algorithms in the field. Over the years several new ones evolved. This chapter tries just to name some of the most important ones, like *MD4* 3.6.1, *MD5* [48], *SHA-1* [44], *SHA-2* 3.6.2, and the *SHA-3*-finalists 3.6.3. The *MD4* algorithm is explained in more detail in this work. It stands also as an example for the *MD5* and the *SHA-1* algorithm, because they are constructed in a similar way. Also the *SHA-2* and the *SHA-3* algorithms are explained more precisely in the following

3.6.1 MD4

MD4 [46] is one of the oldest widely used cryptographic hash algorithms. Ronald Rivest developed it in 1990 and it got standardized by RFC later in 1992 [47].

The prefix *MD* stands for “Message Digest” and the following number is increased with every new algorithm of its category. They are similar, but increasingly complicated. It has a digest length of 128 bits.

MD4 is already broken [13, 14], nevertheless it still has its applications. It is used for example as a fast checksum generating algorithm.

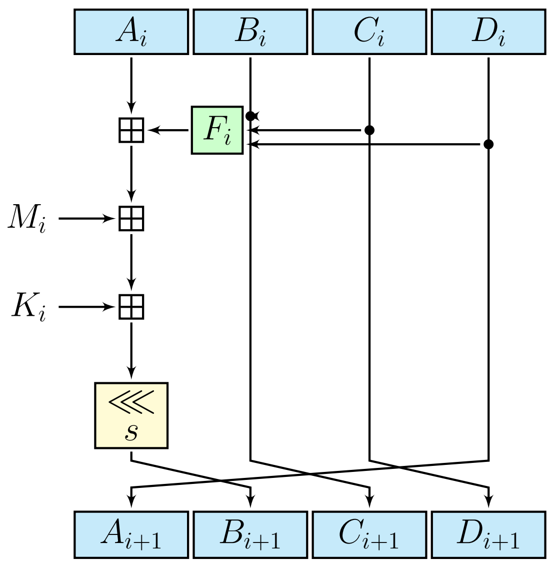

Figure 3.3: *Schematic of the MD4 slgorithm:* One of the algorithm’s 48 operations. The four internal states A, B, C, and D are initialized and processed in every operation. Figure from [27].

In Figure 3.3 one operation of its 48 is shown. The operations are grouped in three rounds with 16 operations each. M_i is a 32-bit block of the input message and K_i is a 32-bit constant, which differs for each of the 48 operations. F_i is the non linear function of the hash function.

3.6.2 SHA-2

The *SHA-2* family consists of a set of hash algorithms. The most widely used ones are the 256-bit *SHA256* and the 512-bit *SHA512* algorithms.

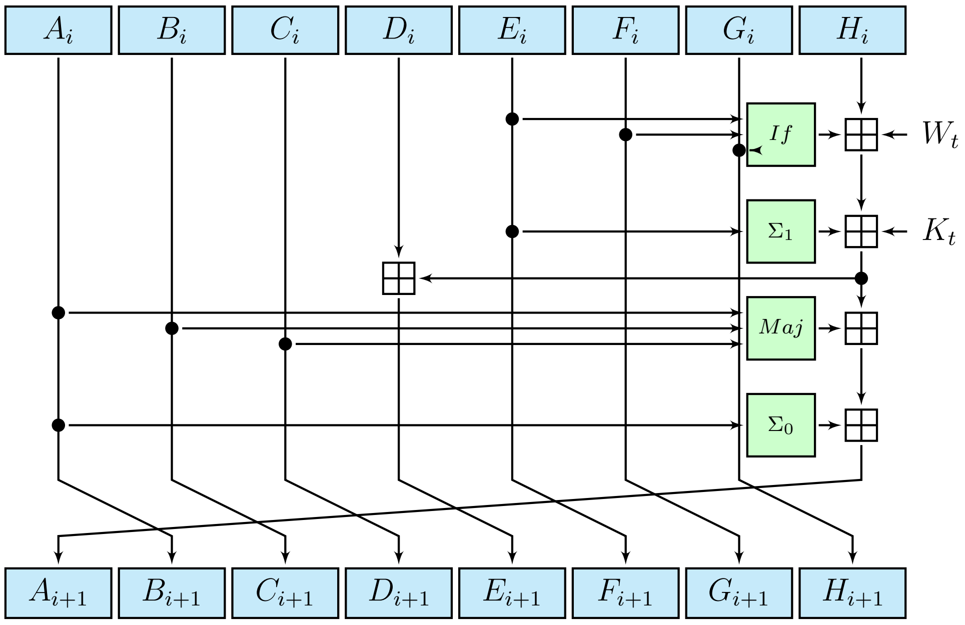

Figure 3.4: *Schematic of the SHA-2 algorithm:* One of the algorithm's 64 rounds. The 8 internal states A, B, C, D, E, F, G, and H are initialized and processed in every round. The green boxes correspond to different operations. Figure from [27].

In Figure 3.4 one of the 64 rounds of the *SHA256* algorithm is shown. The addition operations are performed $\mod 2^{32}$ for *SHA256* and $\mod 2^{64}$ for *SHA512*. The $If(\cdot,\cdot,\cdot)$ operation denotes to

$$If(E,F,G) = (E \wedge F) \oplus (\neg E \wedge G), \tag{3.2}$$

the $\Sigma_1(\cdot)$ operation denotes to

$$\Sigma_1(E) = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25), \tag{3.3}$$

the $Maj(\cdot,\cdot,\cdot)$ operation denotes to

$$Maj(A,B,C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C) \tag{3.4}$$

and the $\Sigma_0(\cdots)$ operation denotes to

$$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22). \tag{3.5}$$

3.6.3 SHA-3 Final Round Candidates

For the *SHA-3* algorithm there was made a cryptographic competition called the *SHA-3 Competition*. It was held during the time from February 11, 2007 to February 10, 2012. In March 2012 the final round candidates were announced [43]. This candidate algorithms are described in the next Sections. The winner of the competition, the algorithm which got standardized by *National Institute of Standards and Technology* (NIST)[2] in the end, was *Keccak*, described in Section 3.6.3.1.

Nevertheless, also the other four “final round candidates” are important and some have established their own field of application. Therefore, they are described in the following sections.

3.6.3.1 Keccak (SHA-3 Winner)

Keccak is the winner of the *SHA-3* competition. It is released under *Creative Commons “No Rights Reserved”* (CC0) license[3]. Although the old *SHA-2* algorithms are not broken and therefore not obsolete yet, *NIST* started a new competition with the goal of constructing a new hash algorithm.

Keccak follows a novel approach with its main functionality relying on the *sponge construction*, based on a wide random permutation [6]. It got standardized by *NIST* in 2015 [18].

3.6.3.2 BLAKE

The *BLAKE* algorithm [2] is based on the *ChaCha* stream cipher [5]. Several versions exist: *BLAKE-256* and *BLAKE-224*, which use 32-bit words and *BLAKE-512* and *BLAKE-384* which use 64-bit words. They produce a 256-bit, 224-bit, 512-bit and 384-bit digest respectively. It consists of 14 rounds for the 32-bit version and 16 rounds for the 64-bit version. There exists also an improved version called *BLAKE2* [3], which was released under CC0 in 2012.

3.6.3.3 Skein

The *Skein* algorithm [21] is based on the *Threefish tweakable block cipher*. It consists of 72 rounds for the 256-bit and 512-bit versions and of 80 rounds for the 1024-bit version. The algorithm and a reference implementation are given to public domain.

An interesting property about *Skein* is that it does not use any *S-Boxes*. Only addition $+$, rotation $\lll, \ggg$ and *XOR* $\oplus$ operations are used. According to [21] it is also possible to use *Skein* as randomized hashing, parallelizable tree hashing, a stream cipher, personalization, and a key derivation function.

[2]`https://www.nist.gov`

[3]`https://creativecommons.org/share-your-work/public-domain/cc0/`

An attack on *Threefish-256* and *Threefish-512* was published in 2010 [30], which successfully breaks 39 and 42 rounds respectively. The *Skein* algorithm is also affected by this attack, although the *Skein*-team claims that their algorithm is still secure[4].

3.6.3.4 JH

The *JH* hash algorithm [51] processes 512-bit input blocks with an internal state of 1024 bits in three steps: First, the input gets *XOR*ed into the left halve of the state, than, a 42 round permutation is applied and in the end it is *XOR*ed into the right halve of the state. The 1024-bit final state can be truncated to 224 bits, 256 bits, 384 bits or 512 bits depending on the desired digest length.

3.6.3.5 Grøstl

The *Grøstl* hash algorithm [23] is constructed with the same *S-Boxes* as used in *Advanced Encryption Standard* (*AES*)[5]. It divides the input into blocks and iteratively computes $h_i = f(h_{i-1}, m_i)$. The internal state kept by *Grøstl* is at least two times the size of its final output, which is truncated in the last round. The compression function

$$f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h \tag{3.6}$$

consists of two permutation functions $P(\cdot)$ and $Q(\cdot)$, which themselves are based on the *AES* algorithm but operate on more bytes. The algorithm consists of 10 rounds, each of them contains the same four operations:

- **AddRoundConstant:** The round constants are fixed, but differ between $P(\cdot)$ and $Q(\cdot)$. “Add” means $XOR \oplus$ in this context.
- **SubBytes:** Each byte in the state matrix is replaced by another one. This operation uses the same *S-Box* as the *AES* implementations do.
- **ShiftBytes:** Each byte from the state matrix gets rotated to the left $\lll$ similar to *AES*. The rotation differs between $P(\cdot)$ and $Q(\cdot)$, and 512-bit and 1024-bit versions.
- **MixBytes:** Multiplies each column of the state matrix by a constant 8×8 matrix in the finite field $\mathbb{F}_{256}$, which is the same as in *AES*.

[4]`http://www.skein-hash.info/sites/default/files/skein1.3.pdf`
[5]`https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`

Chapter 4

Cryptanalysis

4.1 Overview

Cryptanalysis is concerned with the study of analysing information systems regarding their hidden aspects. The goal is to breach their security by targeting weaknesses in the cryptographic algorithms, but also exploit weaknesses in their implementation. Some approaches for performing cryptanalysis are:

- **Ciphertext-only analysis:** Only the ciphertext is known and not the plaintext, nor any other information.

- **Known-plaintext analysis:** In this scenario the attacker has the knowledge of a plaintext-ciphertext pair. With this information it is tried to deduce the key used to encrypt the plaintext to get the corresponding ciphertext.

- **Chosen-plaintext analysis:** The attacker has the possibility to have any desired plaintext encrypted with a key and obtain the resulting ciphertext. The secret key is not known to the attacker. In this setting, *differential cryptanalysis* can be used.

- **Timing/differential power analysis:** It measures the time or power needed for specific cryptographic operations and tries to reconstruct the secret key from it. With this approach the implementation of an algorithm is attacked, rather than its implementation.

Different cryptanalytic attack-methods exist and are explained in the next sections. In this work we use *differential cryptanalysis*, described in Section 4.4.

4.2 Brute-Force Attack

The *brute-force attack* is a “trial and error” method of cryptanalysis. The attacker tries all possibilities for the secret key, until the correct one is found. Some variations try more frequent possibilities first, other use random combinations, and other use a systematic approach.

In general the probability p of guessing the correct key is indirect proportional to the number of possibilities for the secret key, like in

$$p = \frac{1}{2^n}. \tag{4.1}$$

This means the probability p decreases exponentially with the number of bits n of the secret key. For example an *AES256* encryption has 2^{256} possible keys. This is about $1.157920892 \times 10^{77}$, which is an incredibly large number. The probability to guess the key is therefore $p = \frac{1}{2^{256}}$ per try. In practice a key with such a large number of possibilities, or such a small probability, is infeasible to guess within reasonable time.

4.3 Linear Cryptanalysis

In *linear cryptanalysis* first linear equations are constructed. These equations should have a probability of holding, as close to either 0 or 1 as possible. Afterwards the linear equations are used in combination with known plaintext-ciphertext pairs. It should be possible to derive key bits with this. In

$$P_1 \oplus P_3 \oplus C_1 = K_2, \tag{4.2}$$

an example for a linear equation can be seen. The first plaintext bit P_1 is XORed $\oplus$ with the third plaintext bit P_3 and afterwards with the first ciphertext bit C_1. This equals the second key bit K_2. More of these equations combined together form conditions from which the secret key K can be derived.

Any linear equation connecting plaintext, ciphertext, and key bits can be used. In an ideal cipher any equation holds with a probability close to $\frac{1}{2}$. Therefore equations are needed which differ from this probability. The closer the probability is to 0 or to 1, the more information about the secret key can be gained from it. More accurately the equations are called *linear approximations.*

After deriving a linear approximation of the form

$$P_{i_1} \oplus P_{i_2} \oplus \cdots \oplus C_{j_1} \oplus C_{j_2} \oplus \cdots = K_{k_1} \oplus K_{k_2} \oplus \cdots , \tag{4.3}$$

the known plaintext-ciphertext pairs can be used to guess the values of the key bits involved in the approximations. For that, an algorithm called *Matsui's Algorithm 2* [35], can be used.

4.4 Differential Cryptanalysis

Differential cryptanalysis [10] is concerned with the propagation of differential properties within a cipher (Section 4.4.2) or hash function (Section 4.4.3). It is a special type of "chosen plaintext attack".

To test an algorithms' differential properties, an attacker has to choose two different input messages x and x' and computing their difference δ. This difference is then tracked through the entire algorithm.

4.4.1 Difference

In cryptography a difference is an inequality of specific bits within two messages x and x'. There exist several kinds of *differences* which can be used for cryptanalysis purposes.

In this work the focus lies on *generalized differences* [11]. Similar to *XOR*-differences

$$\Delta x_i = x_i \oplus x'_i \tag{4.4}$$

they describe the difference between two messages in a bitwise manner, but more fine-grained with 16 different cases. In Table 4.1 the differential notations for such inequalities are shown.

(x, x')	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
`?`	✓	✓	✓	✓
`-`	✓			✓
`x`		✓	✓	
`0`	✓			
`u`		✓		
`n`			✓	
`1`				✓
`#`				

Table 4.1: Differential notation for differences between two messages x and x'.

The used symbols within this notation are explained more accurately in the following:

- : The bit in characteristic A is equal to the bit in characteristic B. This means there is no difference between A and B at this position, although it is not defined if the bit is 0 or 1.

x : The bit in characteristic A is not equal to the bit in characteristic B. This means there is a difference between A and B at this position, although it is not defined if the bit is 0 or 1.

u : The bit in characteristic A is set to 0 and the bit in characteristic B is set to 1. This means there is a difference between A and B at this position.

n : The bit in characteristic A is set to 1 and the bit in characteristic B is set to 0. This means there is a difference between A and B at this position.

1 : The bit in characteristic A and the bit in characteristic B are both set to 1. This means there is no difference between A and B at this position.

0 : The bit in characteristic A and the bit in characteristic B are both set to 0. This means there is no difference between A and B at this position.

? : The bit in characteristic A nor the bit in characteristic B is known. This means it is also not known if there is a difference between A and B at this position.

: There is a contradiction in the characteristic at this position. It is not used when defining a characteristic, but during collision-finding with a characteristic.

4.4.2 Differential Cryptanalysis of Block Ciphers

Differential cryptanalysis of block ciphers has mainly the goal to recover the cipher's key. The attacker has to have the possibility to encrypt diverse chosen plaintexts with the same key and also retrieve the corresponding ciphertexts. Without this possibility the cryptanalysis task can not be performed.

4.4.3 Differential Cryptanalysis of Hash Functions

Differential cryptanalysis of hash functions is very similar to its application on block ciphers. The main contrast is the opposed goal, namely to find collisions in the algorithm. A collision means two diverse input messages x and x' result in the same hash value h.

4.4.4 Impossible Differential Cryptanalysis

Impossible differential cryptanalysis is a special form of *differential cryptanalysis*. Both forms distinguish in the expected probability with which differences propagate through a cipher. Differential cryptanalysis tracks low-probability differences [31] and impossible differential cryptanalysis high-probability differences.

4.4.5 Differential Characteristic

An example characteristic can be seen in Table 5.1 on page 44. This one is a template characteristic which was automatically generated by this framework. In Table 6.4 on page 69 another characteristic is shown, which can be used in practice.

Chapter 5

Automated Cryptanalysis Framework

We first describe the general structure of the framework in Section 5.1. The compiler framework and the used libraries on which this project is built are described in Section 5.2.

In Section 5.3 the plug-ins available in this framework are described, where the *NL Tool* plug-in (Section 5.3.1) was improved and its functionality extended to be able to process real-world cryptographic algorithms. The *CryptoSMT* plug-in described in Section 5.3.2 was introduced and implemented within this thesis.

In Section 5.4 needed preparations to be able to use this framework are explained. Some of the transformations described in Section 5.5 already existed and some of them are new.

Finally in Section 5.6 the translations applied by the framework are explained in more detail. This part existed partly for the *NL Tool* but not for the *CryptoSMT Tool*.

5.1 Overview

The automated cryptanalysis framework developed in this project is written in C++. Despite the aim to reduce it to a minimum, still some third party projects were used in order to develop the underlying framework. The most important one is the *ROSE Compiler Framework*, which is treated in Section 5.2. It is the base of the whole project. The framework is designed like a plug-in system, describing the defined plug-ins in Section 5.3.

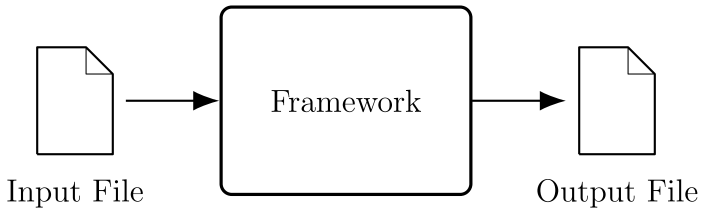

Figure 5.1: *Schematic of the automated analysis framework:* The input file is read by the framework. The framework analyses it and computes an output file for a specific analysis tool from it.

Figure 5.1 shows a schematic of the framework. It can read an input files, process them and write output files. Depending on the algorithm and purpose, several input and output files are needed. The transformations and other processing, done within the framework, are described in the following.

5.2 ROSE Compiler Framework and Libraries

In order to develop this automated cryptanalysis framework, another, already existing framework is used. The used framework is called *ROSE* [45] compiler framework. It is a cross-compiler framework, written in C++.

Additionally, the *Boost* library is used. *Boost* is a collection of C++ libraries for many tasks and structures. In this work it is used for text processing and file operations, because these tasks are needed many times.

5.3 Plug-ins

The framework is designed like a plug-in system. This means, functionality can be extended if needed. There is a common base class for the plug-in design called `ToolTranslator`, which can be subclassed for each new plug-in.

Two plug-ins are already developed, the *NL Tool* plug-in (Section 5.3.1) and the *CryptoSMT Tool* plug-in (Section 5.3.2). The possibility to add functionality for other tools is given.

5.3.1 NL Tool

The *NL Tool* is an analysis tool developed by the *Institute of Applied Information Processing and Communications* (IAIK). This tool can be used to search differential characteristics for cryptographic algorithms. It was not released to the public so far, but is described in a series of parers [36, 11, 37, 19, 20, 15].

The *NL Tool* plug-in was the first plug-in of the framework. It already existed and was adapted to further changes within the current work.

5.3.2 CryptoSMT Tool

The *CryptoSMT Tool* [32] uses a *SAT solver* to validate boolean logic expressions. This *SAT solver* is a specific version to verify *satisfiability modulo theories* (SMT). It can be used to check cryptographic algorithms with specific input and output conditions if they are defined in boolean logic.

The *CryptoSMT Tool* plug-in was developed as a new plug-in. Its goal is to generate an output file in a valid format for the tool in *CVC* language[1].

5.4 Preparations

There are some preparations which need to be done before the actual transformation of the *C*-reference implementation of the algorithm. One of these preparations is to *annotate* the algorithm, which can be seen in Section 5.4.1. Another preparation is *generating a template characteristic*, shown in Section 5.4.2.

5.4.1 Annotations

Before the actual transformation, the algorithm has to be annotated. This means, some hints have to be given to the translator by hand. Otherwise it is not possible to transform it correctly. Like in Code Example 5.1, the annotations are comments, starting with an `@` character:

`@roundfunction`
Annotates the roundfunction, which is the C++ method associated with the *round function* of the cryptographic algorithm, regardless of its purpose (hashing, encryption, *RNG*, etc.).

`@statevariable`
Annotates the name of the variable used as parameter for the initialization vector or starting values of the algorithm.

[1]`https://stp.readthedocs.io/en/latest/cvc-input-language.html`

`@state_variables`
Annotates the comma-separated list of names of the variables used as internal state variables, which are defined by the used characteristic (if a characteristic is used in the first place).

`@state_variables_mapping`
Annotates the comma-separated list of names of the variables used as a mapping between the internal state variables and the order of the assingnment-variable used by the *NL Tool*.

`@iv_state_offset`
Annotates the comma-separated list of offsets of the IV regarding the state variable. Can only be used for a *Feistel network* and is used exclusively by the *NL Tool*.

`@statesize`
Annotates the number of internal states that are predefined by the characteristic (or input parameter of the algorithm).

`@messagevariable`
Annotates the name of the variable used as parameter for the initialization vector or starting values of the algorithm.

`@messagesize`
Annotates the number of variables of the input message predefined by the characteristic (or input parameter of the algorithm).

`@statewidth`
Annotates the bitsize of all the variables used in the algorithm (usually 32 bits or 64 bits).

`@roundsvariable`
Annotates the number of rounds. If the framework is used to generate a template characteristic this number defines the number of internal template-states generated for the template characteristic. If the framework is used to transform the algorithm it defines the number of rounds

processed by the framework.

```
// @roundfunction
// @statevariable = state
// @state_variables = a
// @state_variables_mapping = , a
// @iv_state_offset = 1
// @statesize = 4
// @messagevariable = x
// @messagesize = 16
// @statewidth = 32
// @roundsvariable = round
void roundFct(UINT4 state[4], UINT4 x[16])
{

}
```

Code Example 5.1: Annotations example of the *MD4* algorithm.

In Code Example 5.1, annotations of an example *round function* are shown. This example was taken from the *round function* of the *MD4* algorithm with the additional, but for this algorithm irrelevant `@roundsvariable` annotation.

The framework reads the code and tries to retrieve all the available annotations before doing any transformations. Depending on the available annotations and their values, different transformations are performed. For example the `@statewidth` annotation defines the bitsize of all the variables in the transformation, which influences e.g. the rotation operations $\lll$ and $\ggg$ dramatically.

5.4.2 Generate a Template Characteristic

The framework can also be used to automatically generate a template characteristic out of the *C*-reference implementation of an algorithm and its annotations. Therefore it has to be started with the corresponding command line parameter. In Table 5.1, an example of an automatically generated template characteristic can be seen. This template characteristic was generated from the *MD4* algorithm with the annotations seen in Section 5.4.1.

There exists a specific LaTeX-export method to write a LaTeX-table out of the generated template characteristic. This method was used to generate the output for all the characteristic tables within this work.

```
-4   a:   --------------------------------
-3   a:   --------------------------------
-2   a:   --------------------------------
-1   a:   --------------------------------
 0   a:   --------------------------------   x:   --------------------------------
 1   a:   --------------------------------   x:   --------------------------------
 2   a:   --------------------------------   x:   --------------------------------
 3   a:   --------------------------------   x:   --------------------------------
 4   a:   --------------------------------   x:   --------------------------------
 5   a:   --------------------------------   x:   --------------------------------
 6   a:   --------------------------------   x:   --------------------------------
 7   a:   --------------------------------   x:   --------------------------------
 8   a:   --------------------------------   x:   --------------------------------
 9   a:   --------------------------------   x:   --------------------------------
10   a:   --------------------------------   x:   --------------------------------
11   a:   --------------------------------   x:   --------------------------------
12   a:   --------------------------------   x:   --------------------------------
13   a:   --------------------------------   x:   --------------------------------
14   a:   --------------------------------   x:   --------------------------------
15   a:   --------------------------------   x:   --------------------------------
16   a:   --------------------------------
17   a:   --------------------------------
18   a:   --------------------------------
19   a:   --------------------------------
20   a:   --------------------------------
21   a:   --------------------------------
22   a:   --------------------------------
23   a:   --------------------------------
24   a:   --------------------------------
25   a:   --------------------------------
26   a:   --------------------------------
27   a:   --------------------------------
28   a:   --------------------------------
29   a:   --------------------------------
30   a:   --------------------------------
31   a:   --------------------------------
32   a:   --------------------------------
33   a:   --------------------------------
34   a:   --------------------------------
35   a:   --------------------------------
36   a:   --------------------------------
37   a:   --------------------------------
38   a:   --------------------------------
39   a:   --------------------------------
40   a:   --------------------------------
41   a:   --------------------------------
42   a:   --------------------------------
43   a:   --------------------------------
44   a:   --------------------------------
45   a:   --------------------------------
46   a:   --------------------------------
47   a:   --------------------------------
```

Table 5.1: Template characteristic of the *MD4* algorithm: $a_{-4} \dots a_{-1}$ is the IV, $a_0 \dots a_{47}$ are the internal states and $x_0 \dots x_{15}$ is the input message.

The *MD4* characteristic from Table 5.1 has to be modified in order to use it as a useful input characteristic for the tool. Therefore the *Initialization Vector* (IV) of the algorithm has to be inserted. To help the collision-searching tool finding a collision, several additional values can be inserted into the differential characteristic. Typical additional value would be differential descriptions of some parts of the message or the internal states like in Table 6.4 on page 69.

5.5 Transformations

The cryptanalysis tools described in this work do not handle control logic. Therefore, this parts of the cryptographic algorithms have to be removed and transformed into basic operations, which can be handled with the analysis tools.

An algorithm of this form is called *straight-line program* (SLP) [50]. It is a sequence of basic operations without branches, loops, conditional statements and comparisons.

The input code has to be transformed several times before it gets translated into its final representation. The transformation steps can be seen in Figure 5.2. One of this transformations for example is the transformation to the standard form described in Section 5.5.11.

5.5.1 Inlining

Inlining means removing function calls within the *round function*. Every function call gets replaced by its function's underlying code. In Code Example 5.2 the function to be inlined can be seen. After the transformation step the resulting code can be seen in Code Example 5.3.
For example the code:

```
int addition(int a, int b)
{
  int r;
  r = a + b;
  return r;
}

int main()
{
  int x = 5, y = 3, z;
  z = addition(x,y);
}
```

Code Example 5.2: Inlining example: raw.

gets translated to:

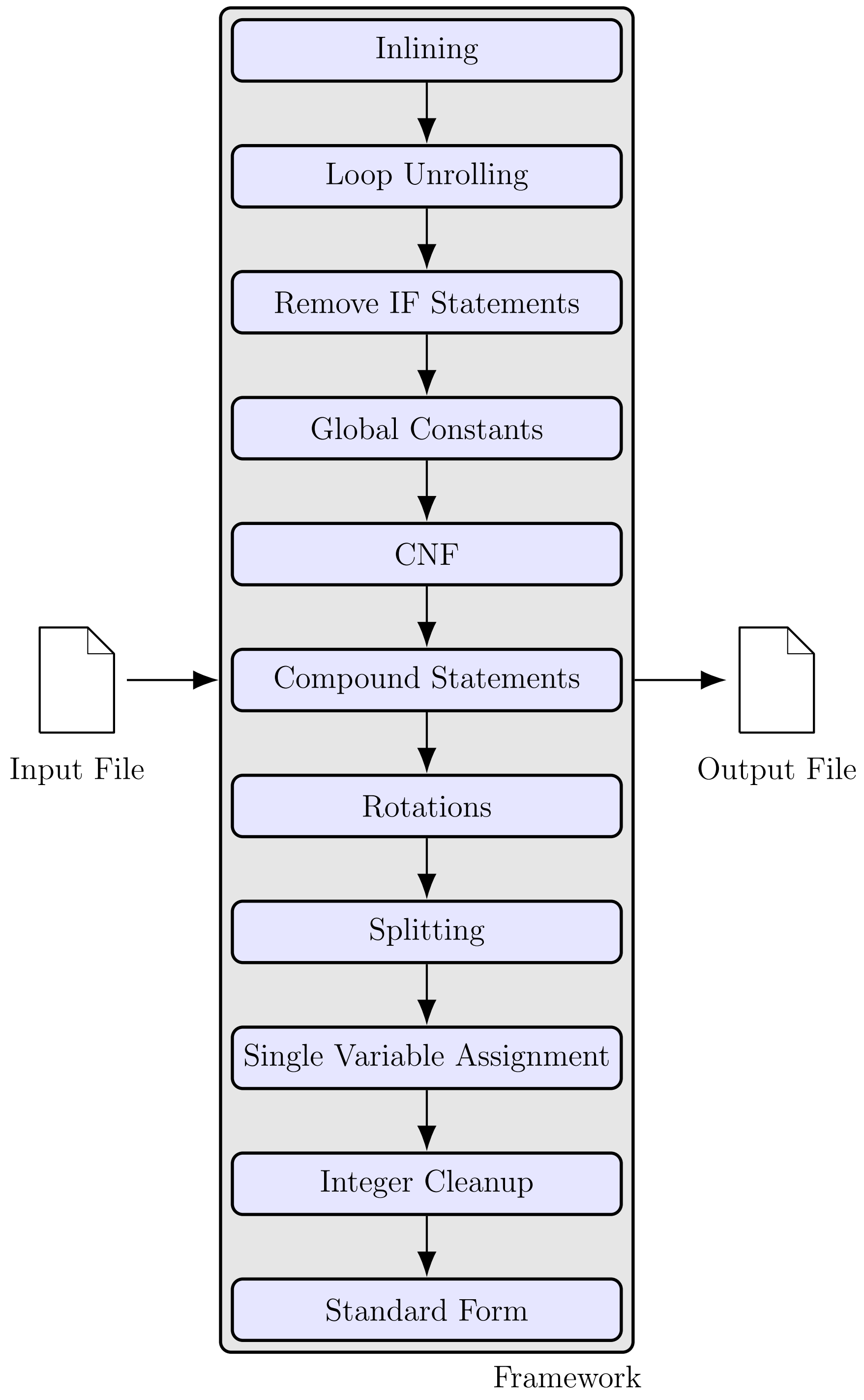

Figure 5.2: Transformations the framework applies to the input file.

```
int main()
{
  int x=5, y=3, z;

  int a = x;
  int b = y;
  int r;
  r = a + b;

  z = r;
}
```

Code Example 5.3: Inlining example: inlined.

5.5.2 Loop Unrolling

Loop unrolling means that all the loops in a method get unrolled. For example if a loop iterates three time over a piece of code (Code Example 5.4), the loop gets removed and the code inside the loop gets repeated three times instead (Code Example 5.5).
For example the code:

```
for(int i = 0; i < 3; i++)
{
    printf(i);
}
```

Code Example 5.4: Loop unrolling example: loop.

gets translated to:

```
printf(0);
printf(1);
printf(2);
```

Code Example 5.5: Loop unrolling example: unrolled loop.

5.5.3 Remove IF Statements

IF statements can not be handled by some analysis tools. Therefore it is tried to resolve the *IF* statements during translation. Most can be removed automatically just by evaluating the statement and remove either one or the other remaining branch, which gets not executed anyway. If this is not possible the algorithm has to be transformed manually into a version without the need of an *IF* statement. This can be quite cumbersome.

5.5.4 Global Constants

The framework tries to find all globally defined variables, regardless if they are *const* or not. All found constants are translated in *HEX*-values and defined for the output representation.

5.5.5 Generate a CNF out of a Truth Table

In some cases it is not possible to use the *const* array extracted like described in Section 5.5.4. One example where it is not possible would be if the indices of the array are generated dynamically during the execution of the algorithm dependent on the input. This means the index of the array could not be translated to a value, but would remain a variable until the execution of the SAT solver. It is not possible to use dynamic indices because the *CVC*-code is code generated for parallel execution.

In such a case, one possible solution is to compute a truth table out of the array. This truth table can be used to generate a *Conjunctive Normal Form* (*CNF*). The precomputed *CNF*-generation function can be inserted into the *CVC*-code. During the execution of the *SAT* solver, this precomputed *CNF* defines a condition for the array index.

The *CNF* generation is important when it comes to transforming global constants described in Section 5.5.4, which are accessed by indices dependent on input data.

5.5.6 Compound Statements

Compound statements are not allowed in many languages, therefore they are split. In some special cases they can be translated into a single operation in the output language and are annotated therefore. They get annotated starting with `/* @compoundOperation */` and ending with `/* @compoundOperation_end */`. One of this cases is the rotation operation in *CVC*-language.

5.5.7 Rotations

Rotations are for example annotated with `/* @ROTL(a, 3) */`, defining a rotation of the variable *a* 3 bits to the *left*. This *rotation*-annotation resides within a *compound*-annotation, to protect it from splitting, described in Section 5.5.6.

5.5.8 Splitting

All compound operations which do not get annotated so far, are split in single operations. *Splitting* continues until no further *splitting* can be performed.

The purpose is to obtain code which consists of *single variable assignments* (Section 5.5.9) only. This means, every variable is only assigned once during the entire runtime of the program. It serves the purpose to make the generated code invariant to its order.

5.5.9 Single Variable Assignment

Single variable assignment means that every variable is only assigned once within its lifetime. If another assignment would be needed, a new variable is declared and all further occurrences of the old one get replaced by the new one. The new one gets assigned once before use and never again.

5.5.10 Integer Cleanup

Integer cleanup denotes the simplification of integer operations. Simple arithmetic operations like *additions*, *subtractions*, *multiplication*, *divisions*, *modulo operations* and also some binary operations like *AND*, *OR*, *XOR* and *shift*-operations get solved and replaced by their result instead. This reduces complexity and code size.

5.5.11 Standard Form

Transforming into standard form defines the combination of all the previous transformations and rewriting *C*-code from them. The output is an executable version of the algorithm (*round function*) which returns the same output as the original *C*-reference implementation.

5.6 Translations

The framework aims to translate the *C*-reference implementation into different output formats. One of these formats is the *CVC*-format used by the *Simple Theorem Prover* (STP[2]) described in Section 5.6.1. Another output format is for the *NL Tool* described in Section 5.6.2.

[2]`https://stp.github.io`

5.6.1 SAT Solver

All operations are split and defined as a single statement. Each statement is further surrounded by its own `ASSERT` as seen in Code Example 5.6 which is omitted in the following operation-explanations.

```
ASSERT(...);
```

Code Example 5.6: `ASSERT`.

For boolean operations discussed in Section 5.6.1.1, we introduce the space of binary numbers which is described by the set

$$\Sigma := \{0, 1\}.$$

Additionally, the space

$$\Sigma^* = \{x : x \in \Sigma^n, \forall n \in \mathbb{N}^+\}$$

is introduced for binary information of arbitrary length.

5.6.1.1 Boolean Operations

Definition 1 (AND-Operation) *We denote the mapping* $\wedge : \Sigma \times \Sigma \to \Sigma$ *as* AND*-operation, where* $x, y, z \in \Sigma$ *are the variables declared before their usage. Since every variable can only be assigned once,* z *cannot be used before.*

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

Table 5.2: Truth Table *AND*-operation $\wedge$.

In *CVC*-language this operation reads as in Code Example 5.7. The corresponding Truth Table can be seen in Table 5.2.

```
(x & y) = z
```

Code Example 5.7: *AND*-operation.

Definition 2 (OR-Operation) *We denote the mapping* $\vee : \Sigma \times \Sigma \to \Sigma$ *as* OR*-operation, where* $x, y, z \in \Sigma$ *are the variables declared before their usage. Since every variable can only be assigned once,* z *cannot be used before.*

x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

Table 5.3: Truth Table *OR*-operation $\vee$.

In *CVC*-language this operation reads as in Code Example 5.8. The corresponding Truth Table can be seen in Table 5.3.

```
(x | y) = z
```

Code Example 5.8: *OR*-operation.

Definition 3 (XOR-Operation) *We denote the mapping $\oplus : \Sigma \times \Sigma \rightarrow \Sigma$ as* XOR*-operation, where $x, y, z \in \Sigma$ are the variables declared before their usage. Since every variable can only be assigned once, z cannot be used before.*

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

Table 5.4: Truth Table *XOR*-operation $\oplus$.

In *CVC*-language this operation reads as in Code Example 5.9. The corresponding Truth Table can be seen in Table 5.4.

```
BVXOR(x, y) = z
```

Code Example 5.9: *XOR*-operation.

Definition 4 (Shift-Operation) *We denote $\ll : \Sigma^n \times \Sigma^n \rightarrow \Sigma^n$ as Left-*Shift*-operation and $\gg : \Sigma^n \times \Sigma^n \rightarrow \Sigma^n$ as Right-*Shift*-operation, where $x, y, z \in \Sigma$ are the variables declared before their usage. Since every variable can only assigned once, z cannot be used before.*

In *CVC*-language this operations read as in Code Example 5.10 and 5.11.

```
(x << y) = z
```

Code Example 5.10: Left-*Shift*-operation.

```
(x >> y) = z
```

Code Example 5.11: Right-*Shift*-operation.

Definition 5 (Rotation-Operation) *We denote* $\lll : \Sigma^n \times \Sigma^n \to \Sigma^n$ *as Left-*Rotation*-operation and* $\ggg : \Sigma^n \times \Sigma^n \to \Sigma^n$ *as Right-*Rotation*-operation, where* $x, y, z \in \Sigma$ *are the variables declared before their usage. Since every variable can only assigned once,* z *cannot be used before.*

The `@` character is used to concatenate two binary represented numbers in *CVC*-language, where this operations read as in Code Example 5.12 and 5.13.

```
(x[25:0]@x[31:26]) = z
```

Code Example 5.12: Left-*Rotation*-operation: rotation by 5 bits: $x \lll 5 = z$, where $n = 32$.

```
(x[4:0]@x[31:5]) = z
```

Code Example 5.13: Right-*Rotation*-operation: rotation by 5 bits: $x \ggg 5 = z$, where $n = 32$.

5.6.1.2 Arithmetic Operations

Definition 6 (ADD-Operation) *We denote the mapping* $+ : \Sigma \times \Sigma \to \Sigma$ *as* $+$*-operation (integer addition* $\mod 2^n$*), where* $x, y, z \in \Sigma$ *are the variables declared before their usage. Since every variable can only be assigned once,* z *cannot be used before.*

x	y	z
0	0	00
0	1	01
1	0	01
1	1	10

Table 5.5: Truth Table *ADD*-operation $+$.

In *CVC*-language this operation reads as in Code Example 5.14. The corresponding Truth Table can be seen in Table 5.5.

```
BVPLUS(n, x, y) = z
```

Code Example 5.14: *ADD*-operation.

Definition 7 (differential ADD-Operation) *We define the* XOR*-operation from Definition 3 for three input values* $xor_3(\cdot,\cdot,\cdot) : \Sigma \times \Sigma \times \Sigma \to \Sigma$ *as*

$$xor_3(x, y, z) := x \oplus y \oplus z, \tag{5.1}$$

for $x, y, z \in \Sigma$*. Furthermore, we define the operation* $eq(\cdot,\cdot,\cdot) : \Sigma \times \Sigma \times \Sigma \to \Sigma$ *for three input values as*

$$eq(x, y, z) := (\neg x \oplus y) \wedge (\neg x \oplus z), \tag{5.2}$$

for $x, y, z \in \Sigma$. *Then, the* ADD-*operation* $\delta : \Sigma^n \times \Sigma^n \to \Sigma^{n+1}$ *is defined as*

$$\delta : (x, y) \mapsto z, \tag{5.3}$$

for $x, y \in \Sigma^n, z \in \Sigma^{n+1}$. *The operation* δ *is called* "good" *if the property*

$$eq(x \ll 1, y \ll 1, z \ll 1) \wedge (xor_3(x, y, z) \oplus (x \ll 1)) = 0, \tag{5.4}$$

holds for all $x, y \in \Sigma^n$ *and* $z \in \delta(\Sigma^n \times \Sigma^n)$. *Expression* (5.1), (5.2), (5.3) *and* (5.4) *are taken from [33].*

5.6.2 NL Tool

For the *NL Tool* all operations are split and defined as a single statement. The code examples vary from the examples provided for the *CryptoSMT Tool*. Because of its similarity to the *CryptoSMT Tool* examples and the lacking relevance for this work, the detailed explanations of the corresponding *NL Tool*-operations are not shown within this thesis. Still, an example of a final translation including some explanation of the *NL Tool* can be found in the results Section 6.4.

5.7 Command Line Parameters

The framework can be started over the command line with various parameters. Where `cipherTranslator` is the name of the program and `-h` can be used to display the help message shown in Code Example 5.15.

Other important parameters are `-f` to define the input file containing the *C*-reference implementation of a cryptographic algorithm and `-c` to define the differential characteristic file used to translate it. If no characteristic is available yet, the framework can be started with the `-f` and the `-dc` parameter defining the *C*-reference implementation of the algorithm and creating a template characteristic for it.

```
Usage:
    cipherTranslator -f <input file> -dc <characteristic> [-r <# of
    rounds>]
    cipherTranslator -f <input file> [-c <characteristic> [-t <(0|1)>][-r
     <# of rounds>]]
    cipherTranslator -h

Options:
    -h --help  Show help message.
    -f         Input file: C reference implementation of algorithm.
    -c         Characterisitc file.
    -dc        Create template differential characteristic.
    -t         Target tool (0|1) [default: 0] 0:CryptoSMT Tool 1:NL Tool.
    -r         Number of rounds [default: 0].
```
Code Example 5.15: Command line parameters of the framework.

Chapter 6

Results

6.1 Overview

In this chapter the results of the thesis are discussed. A typical work flow analysing the *MD4* algorithm, described in Section 3.6.1, is shown. First, in Section 6.2 the preparations needed for both tools are made. Then, in Section 6.3 the differential characteristic is shown. In Section 6.4, the specific input and output files and the processing steps for the *NL Tool* are shown. Next, in Section 6.3 the same is done for the *CryptoSMT Tool*. At last, the problems and limitations are discussed in Section 6.8.

6.2 Preparations

The first step is to annotate the *C*-reference implementation of the algorithm's *round function* like done in Code Example 6.1. The used annotations are explained in Section 5.4.1.

An alternative description with just one updated state word per round instead of all four states is used here, because of *MD4* using a *Feistel network* in its *round function*. Therefore, the `@state_variables = a` annotation has only one state variable assigned, but with the corresponding `@statesize = 4` annotation it matches the four states.

The `@state_variable_mapping = , a` annotation tells the framework to map every second assignment statement within the *round function* to the next output state. With this mapping the start of a new round is marked.

```
// @roundfunction
// @statevariable = state
// @state_variables = a
// @state_variables_mapping = , a
// @iv_state_offset = 1
// @statesize = 4
// @messagevariable = x
// @messagesize = 16
// @statewidth = 32
void roundFct(UINT4 state[4], UINT4 x[16])
{
    UINT4 a = state[0], b = state[1], c = state[2], d = state[3];

/* Round 1 */
FF (a, b, c, d, x[ 0], S11);            /* 1 */
FF (d, a, b, c, x[ 1], S12);            /* 2 */
FF (c, d, a, b, x[ 2], S13);            /* 3 */
FF (b, c, d, a, x[ 3], S14);            /* 4 */
FF (a, b, c, d, x[ 4], S11);            /* 5 */
FF (d, a, b, c, x[ 5], S12);            /* 6 */
FF (c, d, a, b, x[ 6], S13);            /* 7 */
FF (b, c, d, a, x[ 7], S14);            /* 8 */
FF (a, b, c, d, x[ 8], S11);            /* 9 */
FF (d, a, b, c, x[ 9], S12);            /* 10 */
FF (c, d, a, b, x[10], S13);            /* 11 */
FF (b, c, d, a, x[11], S14);            /* 12 */
FF (a, b, c, d, x[12], S11);            /* 13 */
FF (d, a, b, c, x[13], S12);            /* 14 */
FF (c, d, a, b, x[14], S13);            /* 15 */
FF (b, c, d, a, x[15], S14);            /* 16 */

/* Round 2 */
GG (a, b, c, d, x[ 0], S21);            /* 17 */
GG (d, a, b, c, x[ 4], S22);            /* 18 */
GG (c, d, a, b, x[ 8], S23);            /* 19 */
GG (b, c, d, a, x[12], S24);            /* 20 */
GG (a, b, c, d, x[ 1], S21);            /* 21 */
GG (d, a, b, c, x[ 5], S22);            /* 22 */
GG (c, d, a, b, x[ 9], S23);            /* 23 */
GG (b, c, d, a, x[13], S24);            /* 24 */
GG (a, b, c, d, x[ 2], S21);            /* 25 */
GG (d, a, b, c, x[ 6], S22);            /* 26 */
GG (c, d, a, b, x[10], S23);            /* 27 */
GG (b, c, d, a, x[14], S24);            /* 28 */
GG (a, b, c, d, x[ 3], S21);            /* 29 */
GG (d, a, b, c, x[ 7], S22);            /* 30 */
GG (c, d, a, b, x[11], S23);            /* 31 */
GG (b, c, d, a, x[15], S24);            /* 32 */

/* Round 3 */
HH (a, b, c, d, x[ 0], S31);            /* 33 */
HH (d, a, b, c, x[ 8], S32);            /* 34 */
HH (c, d, a, b, x[ 4], S33);            /* 35 */
HH (b, c, d, a, x[12], S34);            /* 36 */
HH (a, b, c, d, x[ 2], S31);            /* 37 */
HH (d, a, b, c, x[10], S32);            /* 38 */
HH (c, d, a, b, x[ 6], S33);            /* 39 */
HH (b, c, d, a, x[14], S34);            /* 40 */
HH (a, b, c, d, x[ 1], S31);            /* 41 */
HH (d, a, b, c, x[ 9], S32);            /* 42 */
HH (c, d, a, b, x[ 5], S33);            /* 43 */
```

```c
HH (b, c, d, a, x[13], S34);             /* 44 */
HH (a, b, c, d, x[ 3], S31);             /* 45 */
HH (d, a, b, c, x[11], S32);             /* 46 */
HH (c, d, a, b, x[ 7], S33);             /* 47 */
HH (b, c, d, a, x[15], S34);             /* 48 */

state[0] += a;
state[1] += b;
state[2] += c;
state[3] += d;

}
```

Code Example 6.1: *MD4* reference implementation: *round function* with annotations.

In Code Example 6.2 the corresponding preprocessor defines and global variables of the algorithm can be seen.

```c
/* Constants for MD4Transform routine.  */
#define S11 3
#define S12 7
#define S13 11
#define S14 19
#define S21 3
#define S22 5
#define S23 9
#define S24 13
#define S31 3
#define S32 9
#define S33 11
#define S34 15

/* F, G and H are basic MD4 functions. */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (y)) | ((x) & (z)) | ((y) & (z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))

/* ROTATE_LEFT rotates x left n bits. */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))

/* FF, GG and HH are transformations for rounds 1, 2 and 3 */
/* Rotation is separate from addition to prevent recomputation */
#define FF(a, b, c, d, x, s) {(a) += F ((b), (c), (d)) + (x); (a) =
    ROTATE_LEFT ((a), (s));}

#define GG(a, b, c, d, x, s) {(a) += G ((b), (c), (d)) + (x) + (UINT4)0
    x5a827999; (a) = ROTATE_LEFT ((a), (s));}

#define HH(a, b, c, d, x, s) {(a) += H ((b), (c), (d)) + (x) + (UINT4)0
    x6ed9eba1; (a) = ROTATE_LEFT ((a), (s));}
```

Code Example 6.2: *MD4* reference implementation: preprocessor defines for the *round function*.

6.3 Characteristic

With the *bash* command from Code Example 6.3, a template characteristic is created. The command line parameters of the framework are explained in Section 5.7. This characteristic can be filled manually and afterwards used again along with the *C*-reference implementation to create output files for the *NL Tool* or the *CryptoSMT Tool*.

```
$ ./cipherTranslator -f md4.c -dc template_characteristic.xml -r 48
```

Code Example 6.3: *Bash* command to create template characterisitc.

This command reads the *C*-reference implementation from the file *"md4.c"* and produces an *xml*-file called "*template_characteristic.xml*" and a corresponding *tex*-file. The *xml*-file can be used later and with other tools and the *tex*-file is used as a table within LaTeX. The generated template characteristic can be seen in Table 6.1. Information needed to create the template characteristic is coming from the annotations of the *C*-reference implementation and the command line parameters of the framework.

The annotations `@state_variables = a` and `@statesize = 4` define the 4 *IV*s corresponding `-4: a` to `-1: a` in the characteristic file in Table 6.1. The command line parameter *-r* 48 defines that the algorithm has 48 rounds resulting in lines `0: a` to `47: a`. The annotations `@messagevariable = x` and `@messagesize = 16` define the 16 words corresponding `0: x` to `16: x` in the characteristic file. The template characteristic consists of dashes (`-`), so there is no difference and it is up to the user to fill it with useful values.

In Table 6.2 the manually modified characteristic inspired by [49] is shown. For a successful search, the cryptologist has to fill in some data. For the *MD4* algorithm the *IV* a, ranging from -4 to -1, has to be set like defined for this algorithm. At least a difference has to be set in the input x, ranging from 0 to 16, or in the intermediate states a, ranging from 0 to 47. Otherwise the characteristic is not differential.

In Code Example 6.4 the prefix of the real input file for the *NL Tool* is shown and in Code Example 6.5 the suffix of the real input file for the *NL Tool* is shown.

```
<config>
<parameters>
  <parameter name="f"  value="md4"/>
  <parameter name="s"  value="48"/>
  <parameter name="w"  value="32"/>
  <parameter name="z" value="main"/>
</parameters>
<char value="
```

Code Example 6.4: Manually modified characteristic of the *MD4* algorithm: search configuration prefix.

```
-4  a:  --------------------------------
-3  a:  --------------------------------
-2  a:  --------------------------------
-1  a:  --------------------------------
 0  a:  --------------------------------  x:  --------------------------------
 1  a:  --------------------------------  x:  --------------------------------
 2  a:  --------------------------------  x:  --------------------------------
 3  a:  --------------------------------  x:  --------------------------------
 4  a:  --------------------------------  x:  --------------------------------
 5  a:  --------------------------------  x:  --------------------------------
 6  a:  --------------------------------  x:  --------------------------------
 7  a:  --------------------------------  x:  --------------------------------
 8  a:  --------------------------------  x:  --------------------------------
 9  a:  --------------------------------  x:  --------------------------------
10  a:  --------------------------------  x:  --------------------------------
11  a:  --------------------------------  x:  --------------------------------
12  a:  --------------------------------  x:  --------------------------------
13  a:  --------------------------------  x:  --------------------------------
14  a:  --------------------------------  x:  --------------------------------
15  a:  --------------------------------  x:  --------------------------------
16  a:  --------------------------------
17  a:  --------------------------------
18  a:  --------------------------------
19  a:  --------------------------------
20  a:  --------------------------------
21  a:  --------------------------------
22  a:  --------------------------------
23  a:  --------------------------------
24  a:  --------------------------------
25  a:  --------------------------------
26  a:  --------------------------------
27  a:  --------------------------------
28  a:  --------------------------------
29  a:  --------------------------------
30  a:  --------------------------------
31  a:  --------------------------------
32  a:  --------------------------------
33  a:  --------------------------------
34  a:  --------------------------------
35  a:  --------------------------------
36  a:  --------------------------------
37  a:  --------------------------------
38  a:  --------------------------------
39  a:  --------------------------------
40  a:  --------------------------------
41  a:  --------------------------------
42  a:  --------------------------------
43  a:  --------------------------------
44  a:  --------------------------------
45  a:  --------------------------------
46  a:  --------------------------------
47  a:  --------------------------------
```

Table 6.1: Differential template characteristic of the *MD4* algorithm: automatically generated.

```
-4   a:   01100111010001010010001100000001
-3   a:   00010000001100100101010001110110
-2   a:   10011000101110101101110011111110
-1   a:   11101111110011011010101110001001
 0   a:   ????????????????????????????????   x:   ---x-----10001000001001010000101
 1   a:   ????????????????????????????????   x:   00101011101010100101001110011001
 2   a:   ????????????????????????????????   x:   x1011000010001010110011010001000
 3   a:   ????????????????????????????????   x:   10110110001110011111000100010100
 4   a:   ????????????????????????????????   x:   x1101111100011000101101110001000
 5   a:   ????????????????????????????????   x:   10111001111001001001100101100011
 6   a:   ????????????????????????????????   x:   00101001100110101001011000000110
 7   a:   ????????????????????????????????   x:   11101010100001001111111111000000
 8   a:   ????????????????????????????????   x:   x0111001010010010111010101100101
 9   a:   ????????????????????????????????   x:   01010110100000111001110000001000
10   a:   ????????????????????????????????   x:   01011010011111100101100001011011
11   a:   ????????????????????????????????   x:   01111111011001111111101111100011
12   a:   ????????????????????????????????   x:   x1110110111010001001101100011100
13   a:   ????????????????????????????????   x:   01000110010100111001101101101100
14   a:   ????????????????????????????????   x:   01011101001100110110010110011110
15   a:   ????????????????????????????????   x:   11100110110111001101001100000001
16   a:   ????????????????????????????????
17   a:   ????????????????????????????????
18   a:   ????????????????????????????????
19   a:   ????????????????????????????????
20   a:   ????????????????????????????????
21   a:   --------------------------------
22   a:   --------------------------------
23   a:   --------------------------------
24   a:   --------------------------------
25   a:   --------------------------------
26   a:   --------------------------------
27   a:   --------------------------------
28   a:   --------------------------------
29   a:   --------------------------------
30   a:   --------------------------------
31   a:   --------------------------------
32   a:   x???????????????????????????????
33   a:   --------------------------------
34   a:   --------------------------------
35   a:   --------------------------------
36   a:   --------------------------------
37   a:   --------------------------------
38   a:   --------------------------------
39   a:   --------------------------------
40   a:   --------------------------------
41   a:   --------------------------------
42   a:   --------------------------------
43   a:   --------------------------------
44   a:   --------------------------------
45   a:   --------------------------------
46   a:   --------------------------------
47   a:   --------------------------------
```

Table 6.2: Manually modified characteristic of the *MD4* algorithm.

```
 -4   a:   01100111010001010010001100000001
 -3   a:   00010000001100100101010001110110
 -2   a:   10011000101110101101110011111110
 -1   a:   11101111110011011010101110001001
  0   a:   ????????????????????????????????   x:   100x10000100010000001001010000101
  1   a:   ????????????????????????????????   x:   00101011101010100101001110011001
  2   a:   ????????????????????????????????   x:   x1011000010001010110011010001000
  3   a:   ????????????????????????????????   x:   10110110001110011111000100010100
  4   a:   ????????????????????????????????   x:   x1101111100011000101101110001000
  5   a:   ????????????????????????????????   x:   10111001111001001001100101100011
  6   a:   ????????????????????????????????   x:   00101001100110101001011000000110
  7   a:   ????????????????????????????????   x:   11101010100001001111111111000000
  8   a:   ????????????????????????????????   x:   x0111001010010010111010101100101
  9   a:   ????????????????????????????????   x:   01010110100000111001110000001000
 10   a:   ????????????????????????????????   x:   01011010011111100101100001011011
 11   a:   ????????????????????????????????   x:   01111111011001111111101111100011
 12   a:   ????????????????????????????????   x:   x1110110111010001001101100011100
 13   a:   ????????????????????????????????   x:   ------10010100111001101101101100
 14   a:   ????????????????????????????????   x:   --------------------------------
 15   a:   ????????????????????????????????   x:   --------------------------------
 16   a:   ????????????????????????????????
 17   a:   ????????????????????????????????
 18   a:   ????????????????????????????????
 19   a:   ????????????????????????????????
 20   a:   ????????????????????????????????
 21   a:   --------------------------------
 22   a:   --------------------------------
 23   a:   --------------------------------
 24   a:   --------------------------------
 25   a:   --------------------------------
 26   a:   --------------------------------
 27   a:   --------------------------------
 28   a:   --------------------------------
 29   a:   --------------------------------
 30   a:   --------------------------------
 31   a:   --------------------------------
 32   a:   x???????????????????????????????
 33   a:   --------------------------------
 34   a:   --------------------------------
 35   a:   --------------------------------
 36   a:   --------------------------------
 37   a:   --------------------------------
 38   a:   --------------------------------
 39   a:   --------------------------------
 40   a:   --------------------------------
 41   a:   --------------------------------
 42   a:   --------------------------------
 43   a:   --------------------------------
 44   a:   --------------------------------
 45   a:   --------------------------------
 46   a:   --------------------------------
 47   a:   --------------------------------
```

Table 6.3: Another manually modified characteristic of the *MD4* algorithm.

```
    "/>
<search reseed="-1" credits="1000">
  <phase twobit_complete="1">
    <setting prob="1">
      <mask word="a" steps="32-47"/>
      <guess condition="?" choice_prob="1"/>
      <guess condition="x" choice_prob="0.000001"/>
    </setting>
  </phase>
  <phase twobit_complete="1">
    <setting prob="1">
      <mask word="a"/>
      <guess condition="?" choice_prob="1"/>
      <guess condition="x" choice_prob="0.000001"/>
    </setting>
  </phase>
  <phase twobit_complete="1">
    <setting prob="1" ordered_guesses="1">
      <mask word="a"/>
      <mask word="x"/>
      <guess condition="-" choice_prob="0.5"/>
    </setting>
  </phase>
</search>
</config>
```

Code Example 6.5: Manually modified characteristic of the *MD4* algorithm: search configuration suffix.

6.4 NL Tool

6.4.1 Original Implementation

The *NL Tool* already comes with an implementation of the *MD4* algorithm. This original implementation can be seen in the following to give an example of the *NL Tool* syntax. The header file (*.h*) is shown in Code Example 6.6 and the implementation file (*.cpp*) in Code Example 6.7.

```
#ifndef MD4_H_
#define MD4_H_

#include "hash.h"

/*! \class Md4
 *  \brief Implementation of the MD4 hash function.
 *
 *  https://tools.ietf.org/html/rfc1320
 */
class Md4: public Hash
{
public:
  static const uint32 K[3];
  static const int S[12];
  static const int P[48];

  Md4(int steps, int N = 32);

protected:
  int md4_steps_;
  ConditionWord W[16];
  ConditionWord tA[48 + 4];
  ConditionWord* A = &tA[4];
  ConditionWord F[48];
};

#endif // MD4_H_
```

Code Example 6.6: *MD4* header file for the *NL Tool*.

```
#include "md4/includes/md4.h"

#include "functions.h"
#include "bitslicestep.h"

const uint32 Md4::K[3] = {
    0x00000000,
    0x5a827999,
    0x6ed9eba1
};

const int Md4::S[12] = {
    3,7,11,19,
    3,5,9,13,
    3,9,11,15
};

const int Md4::P[48] = {
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15,
    0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15,
};

Md4::Md4(int steps, int N) :
    Hash(N), md4_steps_(steps) {

  Step* step = 0;

  for (int i = -4; i < 0; i++)
    A[i] = AddConditionWord("A", i, 4 + i, 0);
  for (int i = 0; i < std::min(16, md4_steps_); i++)
    W[i] = AddConditionWord("W", i, 4 + i * 2 + 1, 1);

  for (int i = 0; i < md4_steps_; i++) {
    F[i] = AddConditionWord("F", i, 4 + i * 2 + 0, 1, SUBWORD);
    A[i] = AddConditionWord("A", i, 4 + i * 2 + 1, 0);

    if (i < 16)
      step = Add(new BitsliceStep<IF>(N, A[i - 1], A[i - 2], A[i - 3], F[
    i]));
    else if (i < 32)
      step = Add(new BitsliceStep<MAJ>(N, A[i - 1], A[i - 2], A[i - 3], F
    [i]));
    else
      step = Add(new BitsliceStep<XOR3>(N, A[i - 1], A[i - 2], A[i - 3],
    F[i]));
    F[i]->SetStepToComputeProbability(step);

    int m = P[i];
    ConditionWord k(new ConditionWordImpl(K[i / 16]));
    step = Add(new CarryStep<ADD4>(N, A[i - 4], F[i], W[m], k, A[i]->Rotr
    (S[(i / 16) * 4 + i % 4])));
    step->SetProbabilityMethod(CYCLICGRAPH);
    A[i]->SetStepToComputeProbability(step);
  }
}
```

Code Example 6.7: *MD4* implementation for the *NL Tool*.

The *NL Tool* is a C++ program which has to be compiled with dedicated code for each cryptographic algorithm. This requires a predefined structure of the code explained in the following:

A new class derived from the *Hash* class has to be implemented. This class has to have a public constructor of the form `MD4(int steps, int N)`, where *steps* are the number of steps processed and *N* is the bitsize of the algorithm. Each word used in the algorithm has to be defined as `AddConditionWord()` with 4 or 5 parameters like shown in Code Example 6.8. The first parameter defines the name of the variable read from the characteristic file and the second defines the step number. The words for *IV*, *state* and *message* are defined as an array because there are many of the same type. The class does not contain any functions but the constructor, where the whole algorithm is implemented.

Every mathematical operation performed on a variable is defined as `BitsliceStep<operation>()`, where *operation* marks the performed operation. New operations can be added in the *"functions.h"* file shipped with the *NL Tool*. Constants are defined as `ConditionWordImpl()` like shown in Code Example 6.9.

```
ConditionWord AddConditionWord(std::string name, int step_number, int row
    , int col, int type = MAINWORD, int num_bits = 1);
```

Code Example 6.8: *NL Tool* `ConditionWord()` signature.

```
ConditionWordImpl(uint64 constant, int word_size = 32, std::string name =
     "");
```

Code Example 6.9: *NL Tool* `ConditionWordImpl()` signature.

6.4.2 Template Files

For the automatic code generation template files are used to match a given structure. The positions marked with %#% where # is a number between 1 and 10 are place holders for the automatically generated code pieces. In

Code Example 6.10 the template for the header file can be seen. Here, for example %1% is the place holder for the class name.

In Code Example 6.11 the implementation file can be seen. Here, for example %10% is the place holder for the translated *round function*.

```
/*
 * auto generated cipher representation
 */
#ifndef %1%_H_
#define %1%_H_

#include "hash.h"

class %1% : public Hash
{
public:
  %1%(int steps, int N = %4%);
  %2%

protected:
%3%
};

#endif // %1%_H_
```

Code Example 6.10: Header template for the automatic translation process.

```
#include "%1%/includes/%1%.h"
#include "linearstep.h"
#include "bitslicestep.h"
#include "functions.h"

////////////////////////////// class %2% //////////////////////////////

// constants
%9%

%2%::%2%(int steps, int N) :
        Hash(N)
{
  for(int i = -%6%; i < 0; i++)
  {
%3%
  }
  for(int i = 0; i < std::min(%7%, %8%); i++)
  {
%4%
  }
  for(int i = 0; i < %8%; i++)
  {
%5%
  }

// algorithm
%10%
}
```

Code Example 6.11: Implementation template for the automatic translation process.

6.4.3 Translation

With the *bash* command from Code Example 6.12, an implementation of the *MD4* algorithm for the *NL Tool* is created automatically. The output files called "*md4.h*" and "*md4.cpp*" can be seen in Code Example 6.13 and 6.14 respectively. Only the first two rounds of 48 are shown here as an example. These two files need to replace the original files in the *hash* directory of the *NL Tool*, which afterwards needs to be compiled again.

```
$ ./cipherTranslator -f md4.c -c
    template_characteristic_modified_manually.xml -r 48 -t 1
```

Code Example 6.12: *Bash* command to create *NL Tool* output.

```
/*
 * auto generated cipher representation
 */
#ifndef Md4_H_
#define Md4_H_

#include "hash.h"

class Md4 : public Hash
{
public:
  Md4(int steps, int N = 32);
  static const uint32_t K[2];

protected:
  int steps;
  ConditionWord x[16];
  ConditionWord ta[52];
  ConditionWord* a = &ta[4];

};

#endif // Md4_H_
```

Code Example 6.13: *MD4* header file for the *NL Tool*: generated automatically. 2 out of 48 rounds.

```
#include "md4/includes/md4.h"
#include "linearstep.h"
#include "bitslicestep.h"
#include "functions.h"

////////////////////////////// class Md4 //////////////////////////////

// constants
const uint32 Md4::K[2] = {
  0x5a827999,
  0x6ed9eba1
};

Md4::Md4(int steps, int N) :
        Hash(N)
{
  for (int i = -4; i < 0; i++)
  {
    a[i] = AddConditionWord("a", i, 4 + i, 0);
  }
  for (int i = 0; i < std::min(16, 48); i++)
  {
    x[i] = AddConditionWord("x", i, 4 + i * 2 + 1, 1);
  }
  for (int i = 0; i < 48; i++)
  {
    a[i] = AddConditionWord("a", i, 4 + i * 2 + 1, 0);
  }

// algorithm
ConditionWord k0(new ConditionWordImpl(Md4::K[0]));
ConditionWord k1(new ConditionWordImpl(Md4::K[1]));

//% ANCHOR PREFIX
ConditionWord binaryOpTemp01 = AddConditionWord("binaryOpTemp01", 0, 4,
    2, SUBWORD);
Add(new BitsliceStep<AND2>(N, a[-1], a[-2], binaryOpTemp01));
ConditionWord unaryOpTemp01 = AddConditionWord("unaryOpTemp01", 0, 4, 3,
    SUBWORD);
Add(new BitsliceStep<COMPL>(N, a[-1], unaryOpTemp01));
ConditionWord binaryOpTemp11 = AddConditionWord("binaryOpTemp11", 0, 4,
    4, SUBWORD);
Add(new BitsliceStep<AND2>(N, unaryOpTemp01, a[-3], binaryOpTemp11));
ConditionWord binaryOpTemp22 = AddConditionWord("binaryOpTemp22", 0, 4,
    5, SUBWORD);
Add(new BitsliceStep<OR>(N, binaryOpTemp01, binaryOpTemp11,
    binaryOpTemp22));
ConditionWord binaryOpTemp21 = AddConditionWord("binaryOpTemp21", 0, 4,
    6, SUBWORD);
Add(new CarryStep<ADD2>(N, binaryOpTemp22, x[0], binaryOpTemp21));
ConditionWord singleAssiTemp1 = AddConditionWord("singleAssiTemp1", 0, 4,
     7, SUBWORD);
Add(new CarryStep<ADD2>(N, a[-4], binaryOpTemp21, singleAssiTemp1));
ConditionWord singleAssiTemp2 = AddConditionWord("singleAssiTemp2", 0, 4,
     8, SUBWORD);
Add(new BitsliceStep<ID>(N, singleAssiTemp1->Rotl(3), a[0]));

ConditionWord binaryOpTemp02 = AddConditionWord("binaryOpTemp02", 1, 6,
    2, SUBWORD);
Add(new BitsliceStep<AND2>(N, a[0], b, binaryOpTemp02));
ConditionWord unaryOpTemp02 = AddConditionWord("unaryOpTemp02", 1, 6, 3,
    SUBWORD);
Add(new BitsliceStep<COMPL>(N, a[0], unaryOpTemp02));
ConditionWord binaryOpTemp12 = AddConditionWord("binaryOpTemp12", 1, 6,
    4, SUBWORD);
Add(new BitsliceStep<AND2>(N, unaryOpTemp02, c, binaryOpTemp12));
ConditionWord binaryOpTemp24 = AddConditionWord("binaryOpTemp24", 1, 6,
    5, SUBWORD);
Add(new BitsliceStep<OR>(N, binaryOpTemp02, binaryOpTemp12,
    binaryOpTemp24));
ConditionWord binaryOpTemp23 = AddConditionWord("binaryOpTemp23", 1, 6,
    6, SUBWORD);
Add(new CarryStep<ADD2>(N, binaryOpTemp24, x[1], binaryOpTemp23));
ConditionWord singleAssiTemp3 = AddConditionWord("singleAssiTemp3", 1, 6,
     7, SUBWORD);
Add(new CarryStep<ADD2>(N, d, binaryOpTemp23, singleAssiTemp3));
ConditionWord singleAssiTemp4 = AddConditionWord("singleAssiTemp4", 1, 6,
     8, SUBWORD);
```

```
Add(new BitsliceStep<ID>(N, singleAssiTemp3->Rotl(7), a[1]));
// ...
  int anchor;
}
```

Code Example 6.14: *MD4* implementation for the *NL Tool*: generated automatically. 2 out of 48 rounds.

Line 8 to 12 of Code Example 6.14 define the needed constants. In line 14 the implementation of the constructor starts. Line 17 to 20 initialises the *IV*, line 21 to 24 initialises the *message* and line 25 to 28 initialises the *state* variables. In line 31 and 32 the constants are initialised and in line 35 the *round function* starts.

For readability reasons only 2 out of 48 rounds are shown in the code example. The first round is located form line 35 to 48 and the second round from line 50 to line 63. In total the file has 747 lines of code.

6.4.4 Search

With the command from Code Example 6.15, the *NL Tool* can be started. First, it is compiled with the original *MD4* algorithm from Code Example 6.7 and started afterwards. Then, it is compiled with the new *MD4* algorithm from Code Example 6.14 and started again.

```
$ ./nltool hash/md4/chars/template_characteristic_modified_manually.xml
```

Code Example 6.15: *Bash* command to run the *NL Tool*.

6.4.5 Solution

The resulting characteristic file is shown in Table 6.4.

6.5 CryptoSMT Tool

The *CryptoSMT Tool* does not come with implementations of any cryptographic algorithms. Therefore, no original implementation can be used as a template to automatically generate a new implementation of the *MD4* algorithm.

6.5.1 Template Files

In Code Example 6.16 the template file for generating the algorithm is shown. The %2% label marks the place holder for the *round function*. The algorithm

```
-4   a:   01100111010001010010001100000001
-3   a:   00010000001100100101010001110110
-2   a:   10011000101110101101110011111110
-1   a:   11101111110011011010101110001001
 0   a:   x-------------------------------   x:   ---x----------------------------
 1   a:   --------------------------------   x:   --------------------------------
 2   a:   ---------------------x----------   x:   x-------------------------------
 3   a:   -------------x------------------   x:   --------------------------------
 4   a:   ---------xx---------------------   x:   x-------------------------------
 5   a:   ---x---------xx-----------------   x:   --------------------------------
 6   a:   xx-x------x---------------------   x:   --------------------------------
 7   a:   ---------------------xx---------   x:   --------------------------------
 8   a:   x----------------xxx------------   x:   x-------------------------------
 9   a:   --xx-xx-xxx---------------------   x:   --------------------------------
10   a:   --------------------------------   x:   --------------------------------
11   a:   ---x---------------x------------   x:   --------------------------------
12   a:   ------x-------------------------   x:   x-------------------------------
13   a:   x--x----------------------------   x:   --------------------------------
14   a:   --------------------------------   x:   --------------------------------
15   a:   x-------------------------------   x:   --------------------------------
16   a:   ---x----------------------------
17   a:   --------------------------------
18   a:   --------------------------------
19   a:   --------------------------------
20   a:   x-------------------------------
21   a:   --------------------------------
22   a:   --------------------------------
23   a:   --------------------------------
24   a:   --------------------------------
25   a:   --------------------------------
26   a:   --------------------------------
27   a:   --------------------------------
28   a:   --------------------------------
29   a:   --------------------------------
30   a:   --------------------------------
31   a:   --------------------------------
32   a:   x-------------------------------
33   a:   --------------------------------
34   a:   --------------------------------
35   a:   --------------------------------
36   a:   --------------------------------
37   a:   --------------------------------
38   a:   --------------------------------
39   a:   --------------------------------
40   a:   --------------------------------
41   a:   --------------------------------
42   a:   --------------------------------
43   a:   --------------------------------
44   a:   --------------------------------
45   a:   --------------------------------
46   a:   --------------------------------
47   a:   --------------------------------
```

Table 6.4: Differential characteristic of the *MD4* algorithm: *MD4* collision called "*test.xml*".

in *CVC* language finishes with `QUERY FALSE;` and `COUNTEREXAMPLE;` to force the *CryptoSMT Tool* to generate a concrete example for its decision.

```
%2%
QUERY FALSE;
COUNTEREXAMPLE;
```

Code Example 6.16: Implementation template for the automatic translation process.

6.5.2 Translation

With the *bash* command from Code Example 6.17, an implementation of the *MD4* algorithm for the *CryptoSMT Tool* is created automatically. The generated output file called *"md4.cvc"* is shown in Code Example 6.18. Only the first two rounds of 48 are shown here as an example. In total the file has 1788 lines of code.

```
$ ./cipherTranslator -f md4.c -c test.xml -t 0
```

Code Example 6.17: *Bash* command to create *CryptoSMT Tool* output.

The generated characteristic with the *NL Tool*, shown in Table 6.4, is used along with the *C*-reference implementation from Code Example 6.1 to create the code for the *CryptoSMT Tool*. This automatically generated code written in *CVC*-language is shown in Code Example 6.18.

```
% ...
state : ARRAY BITVECTOR(32) OF BITVECTOR(32);
state_b : ARRAY BITVECTOR(32) OF BITVECTOR(32);
state_out : ARRAY BITVECTOR(32) OF BITVECTOR(32);
state_out_b : ARRAY BITVECTOR(32) OF BITVECTOR(32);
x : ARRAY BITVECTOR(32) OF BITVECTOR(32);
x_b : ARRAY BITVECTOR(32) OF BITVECTOR(32);
ASSERT(state[0hex00000000] = 0hex67452301);
ASSERT(state[0hex00000001] = 0hexefcdab89);
ASSERT(state[0hex00000002] = 0hex98badcfe);
ASSERT(state[0hex00000003] = 0hex10325476);
ASSERT(state_b[0hex00000000] = 0hex67452301);
ASSERT(state_b[0hex00000001] = 0hexefcdab89);
ASSERT(state_b[0hex00000002] = 0hex98badcfe);
ASSERT(state_b[0hex00000003] = 0hex10325476);
ASSERT(x[0hex00000001] = 0hex2baa5399);
ASSERT(x[0hex00000003] = 0hexb639f114);
ASSERT(x[0hex00000005] = 0hexb9e49963);
ASSERT(x[0hex00000006] = 0hex299a9606);
ASSERT(x[0hex00000007] = 0hexea84ffc0);
ASSERT(x[0hex00000009] = 0hex56839c08);
ASSERT(x[0hex0000000a] = 0hex5a7e585b);
ASSERT(x[0hex0000000b] = 0hex7f67fbe3);
ASSERT(x[0hex0000000d] = 0hex46539b6c);
ASSERT(x[0hex0000000e] = 0hex5d33659e);
ASSERT(x[0hex0000000f] = 0hexe6dcd301);
ASSERT(x_b[0hex00000001] = 0hex2baa5399);
ASSERT(x_b[0hex00000003] = 0hexb639f114);
ASSERT(x_b[0hex00000005] = 0hexb9e49963);
ASSERT(x_b[0hex00000006] = 0hex299a9606);
ASSERT(x_b[0hex00000007] = 0hexea84ffc0);
ASSERT(x_b[0hex00000009] = 0hex56839c08);
ASSERT(x_b[0hex0000000a] = 0hex5a7e585b);
ASSERT(x_b[0hex0000000b] = 0hex7f67fbe3);
ASSERT(x_b[0hex0000000d] = 0hex46539b6c);
ASSERT(x_b[0hex0000000e] = 0hex5d33659e);
ASSERT(x_b[0hex0000000f] = 0hexe6dcd301);
ASSERT((x[0hex00000000] & 0hex88441285) = 0hex88441285);
ASSERT((x[0hex00000000] & 0hex67bbed7a) = 0hex00000000);
ASSERT((x[0hex00000002] & 0hex58456688) = 0hex58456688);
ASSERT((x[0hex00000002] & 0hex27ba9977) = 0hex00000000);
ASSERT((x[0hex00000004] & 0hex6f8c5b88) = 0hex6f8c5b88);
ASSERT((x[0hex00000004] & 0hex1073a477) = 0hex00000000);
ASSERT((x[0hex00000008] & 0hex39497565) = 0hex39497565);
ASSERT((x[0hex00000008] & 0hex46b68a9a) = 0hex00000000);
ASSERT((x[0hex0000000c] & 0hex76e89b1c) = 0hex76e89b1c);
ASSERT((x[0hex0000000c] & 0hex091764e3) = 0hex00000000);
ASSERT((x_b[0hex00000000] & 0hex88441285) = 0hex88441285);
ASSERT((x_b[0hex00000000] & 0hex67bbed7a) = 0hex00000000);
ASSERT((x_b[0hex00000002] & 0hex58456688) = 0hex58456688);
ASSERT((x_b[0hex00000002] & 0hex27ba9977) = 0hex00000000);
ASSERT((x_b[0hex00000004] & 0hex6f8c5b88) = 0hex6f8c5b88);
ASSERT((x_b[0hex00000004] & 0hex1073a477) = 0hex00000000);
ASSERT((x_b[0hex00000008] & 0hex39497565) = 0hex39497565);
ASSERT((x_b[0hex00000008] & 0hex46b68a9a) = 0hex00000000);
ASSERT((x_b[0hex0000000c] & 0hex76e89b1c) = 0hex76e89b1c);
ASSERT((x_b[0hex0000000c] & 0hex091764e3) = 0hex00000000);
ASSERT(BVXOR(x[0hex00000000],x_b[0hex00000000]) = 0hex10000000);
ASSERT(BVXOR(x[0hex00000002],x_b[0hex00000002]) = 0hex80000000);
ASSERT(BVXOR(x[0hex00000004],x_b[0hex00000004]) = 0hex80000000);
ASSERT(BVXOR(x[0hex00000008],x_b[0hex00000008]) = 0hex80000000);
ASSERT(BVXOR(x[0hex0000000c],x_b[0hex0000000c]) = 0hex80000000);
% ANCHOR PREFIX
a : BITVECTOR(32);
a_b : BITVECTOR(32);
ASSERT(a = state[0hex00000000]);
ASSERT(a_b = state_b[0hex00000000]);
b : BITVECTOR(32);
b_b : BITVECTOR(32);
ASSERT(b = state[0hex00000001]);
ASSERT(b_b = state_b[0hex00000001]);
c : BITVECTOR(32);
c_b : BITVECTOR(32);
ASSERT(c = state[0hex00000002]);
ASSERT(c_b = state_b[0hex00000002]);
d : BITVECTOR(32);
d_b : BITVECTOR(32);
ASSERT(d = state[0hex00000003]);
ASSERT(d_b = state_b[0hex00000003]);
binaryOpTemp01 : BITVECTOR(32);
binaryOpTemp01_b : BITVECTOR(32);
ASSERT((b & c) = binaryOpTemp01);
ASSERT((b_b & c_b) = binaryOpTemp01_b);
unaryOpTemp01 : BITVECTOR(32);
unaryOpTemp01_b : BITVECTOR(32);
ASSERT((~b) = unaryOpTemp01);
ASSERT((~b_b) = unaryOpTemp01_b);
binaryOpTemp11 : BITVECTOR(32);
binaryOpTemp11_b : BITVECTOR(32);
```

```
ASSERT((unaryOpTemp01 & d) = binaryOpTemp11);
ASSERT((unaryOpTemp01_b & d_b) = binaryOpTemp11_b);
binaryOpTemp22 : BITVECTOR(32);
binaryOpTemp22_b : BITVECTOR(32);
ASSERT((binaryOpTemp01 | binaryOpTemp11) = binaryOpTemp22);
ASSERT((binaryOpTemp01_b | binaryOpTemp11_b) = binaryOpTemp22_b);
binaryOpTemp21 : BITVECTOR(32);
binaryOpTemp21_b : BITVECTOR(32);
ASSERT(BVPLUS(32,binaryOpTemp22,x[0hex00000000]) = binaryOpTemp21);
ASSERT(BVPLUS(32,binaryOpTemp22_b,x_b[0hex00000000]) = binaryOpTemp21_b);
singleAssiTemp1 : BITVECTOR(32);
singleAssiTemp1_b : BITVECTOR(32);
ASSERT(BVPLUS(32,a,binaryOpTemp21) = singleAssiTemp1);
ASSERT(BVPLUS(32,a_b,binaryOpTemp21_b) = singleAssiTemp1_b);
singleAssiTemp2 : BITVECTOR(32);
singleAssiTemp2_b : BITVECTOR(32);
ASSERT((singleAssiTemp1[28:0]@singleAssiTemp1[31:29]) = singleAssiTemp2);
ASSERT((singleAssiTemp1_b[28:0]@singleAssiTemp1_b[31:29]) =
    singleAssiTemp2_b);
binaryOpTemp02 : BITVECTOR(32);
binaryOpTemp02_b : BITVECTOR(32);
ASSERT((singleAssiTemp2 & b) = binaryOpTemp02);
ASSERT((singleAssiTemp2_b & b_b) = binaryOpTemp02_b);
unaryOpTemp02 : BITVECTOR(32);
unaryOpTemp02_b : BITVECTOR(32);
ASSERT((~singleAssiTemp2) = unaryOpTemp02);
ASSERT((~singleAssiTemp2_b) = unaryOpTemp02_b);
binaryOpTemp12 : BITVECTOR(32);
binaryOpTemp12_b : BITVECTOR(32);
ASSERT((unaryOpTemp02 & c) = binaryOpTemp12);
ASSERT((unaryOpTemp02_b & c_b) = binaryOpTemp12_b);
binaryOpTemp24 : BITVECTOR(32);
binaryOpTemp24_b : BITVECTOR(32);
ASSERT((binaryOpTemp02 | binaryOpTemp12) = binaryOpTemp24);
ASSERT((binaryOpTemp02_b | binaryOpTemp12_b) = binaryOpTemp24_b);
binaryOpTemp23 : BITVECTOR(32);
binaryOpTemp23_b : BITVECTOR(32);
ASSERT(BVPLUS(32,binaryOpTemp24,x[0hex00000001]) = binaryOpTemp23);
ASSERT(BVPLUS(32,binaryOpTemp24_b,x_b[0hex00000001]) = binaryOpTemp23_b);
singleAssiTemp3 : BITVECTOR(32);
singleAssiTemp3_b : BITVECTOR(32);
ASSERT(BVPLUS(32,d,binaryOpTemp23) = singleAssiTemp3);
ASSERT(BVPLUS(32,d_b,binaryOpTemp23_b) = singleAssiTemp3_b);
singleAssiTemp4 : BITVECTOR(32);
singleAssiTemp4_b : BITVECTOR(32);
ASSERT((singleAssiTemp3[24:0]@singleAssiTemp3[31:25]) = singleAssiTemp4);
ASSERT((singleAssiTemp3_b[24:0]@singleAssiTemp3_b[31:25]) =
    singleAssiTemp4_b);
binaryOpTemp03 : BITVECTOR(32);
binaryOpTemp03_b : BITVECTOR(32);
ASSERT((singleAssiTemp4 & singleAssiTemp2) = binaryOpTemp03);
ASSERT((singleAssiTemp4_b & singleAssiTemp2_b) = binaryOpTemp03_b);
unaryOpTemp03 : BITVECTOR(32);
unaryOpTemp03_b : BITVECTOR(32);
ASSERT((~singleAssiTemp4) = unaryOpTemp03);
ASSERT((~singleAssiTemp4_b) = unaryOpTemp03_b);
% ...
ASSERT(singleAssiTemp4 = 0hex3bb8748b);
ASSERT(singleAssiTemp4_b = 0hex3bb8748b);
ASSERT((singleAssiTemp2 & 0hex42209424) = 0hex42209424);
ASSERT((singleAssiTemp2 & 0hex3ddf6bdb) = 0hex00000000);
ASSERT((singleAssiTemp2_b & 0hex42209424) = 0hex42209424);
ASSERT((singleAssiTemp2_b & 0hex3ddf6bdb) = 0hex00000000);
ASSERT(BVXOR(singleAssiTemp2,singleAssiTemp2_b) = 0hex80000000);

QUERY FALSE;
COUNTEREXAMPLE;
```

Code Example 6.18: Automatically generated code for the *CryptoSMT Tool*. 2 out of 48 rounds.

6.5.3 Search

The *CryptoSMT Tool* starts searching for a collision just by supplying the generated *CVC*-file from Code Example 6.18 With the *bash* command from Code Example 6.19 the tool starts searching.

```
$ stp_simple md4.cvc
```

Code Example 6.19: *Bash* command to search the collision.

6.5.4 Solution

With the *bash* command from Code Example 6.20 the differential input message to the algorithm can be seen in Code Example 6.21.

```
$ stp_simple md4.cvc | sort | grep ' x'
```

Code Example 6.20: *Bash* command to extract the collision.

```
ASSERT( x[0x00000000] = 0x88441285 );
ASSERT( x[0x00000001] = 0x2BAA5399 );
ASSERT( x[0x00000002] = 0x58456688 );
ASSERT( x[0x00000003] = 0xB639F114 );
ASSERT( x[0x00000004] = 0xEF8C5B88 );
ASSERT( x[0x00000005] = 0xB9E49963 );
ASSERT( x[0x00000006] = 0x299A9606 );
ASSERT( x[0x00000007] = 0xEA84FFC0 );
ASSERT( x[0x00000008] = 0xB9497565 );
ASSERT( x[0x00000009] = 0x56839C08 );
ASSERT( x[0x0000000A] = 0x5A7E585B );
ASSERT( x[0x0000000B] = 0x7F67FBE3 );
ASSERT( x[0x0000000C] = 0xF6E89B1C );
ASSERT( x[0x0000000D] = 0x46539B6C );
ASSERT( x[0x0000000E] = 0x5D33659E );
ASSERT( x[0x0000000F] = 0xE6DCD301 );
ASSERT( x_b[0x00000000] = 0x98441285 );
ASSERT( x_b[0x00000001] = 0x2BAA5399 );
ASSERT( x_b[0x00000002] = 0xD8456688 );
ASSERT( x_b[0x00000003] = 0xB639F114 );
ASSERT( x_b[0x00000004] = 0x6F8C5B88 );
ASSERT( x_b[0x00000005] = 0xB9E49963 );
ASSERT( x_b[0x00000006] = 0x299A9606 );
ASSERT( x_b[0x00000007] = 0xEA84FFC0 );
ASSERT( x_b[0x00000008] = 0x39497565 );
ASSERT( x_b[0x00000009] = 0x56839C08 );
ASSERT( x_b[0x0000000A] = 0x5A7E585B );
ASSERT( x_b[0x0000000B] = 0x7F67FBE3 );
ASSERT( x_b[0x0000000C] = 0x76E89B1C );
ASSERT( x_b[0x0000000D] = 0x46539B6C );
ASSERT( x_b[0x0000000E] = 0x5D33659E );
ASSERT( x_b[0x0000000F] = 0xE6DCD301 );
```

Code Example 6.21: Collision.

With the *bash* command from Code Example 6.22 the differential output message to the algorithm can be seen in Code Example 6.23.

```
$ stp_simple md4.cvc | sort | grep out
```

Code Example 6.22: *Bash* command to extract the state.

```
ASSERT( state_out[0x00000000] = 0x7E663C7E );
ASSERT( state_out[0x00000001] = 0xB78F3B6D );
ASSERT( state_out[0x00000002] = 0x114AE04A );
ASSERT( state_out[0x00000003] = 0x09B8AC68 );
ASSERT( state_out_b[0x00000000] = 0x7E663C7E );
ASSERT( state_out_b[0x00000001] = 0xB78F3B6D );
ASSERT( state_out_b[0x00000002] = 0x114AE04A );
ASSERT( state_out_b[0x00000003] = 0x09B8AC68 );
```

Code Example 6.23: State.

Since there are differences in the input message *x* and *x_b* from Code Example 6.21 and no differences in the output *state_out* and *state_out_b* from Code Example 6.23 any more, the collision found with the *NL Tool* is verified with the *CryptoSMT Tool* correctly.

6.6 Implemented Operations

Table 6.5 shows some functionality added to the framework. The table lists general functionality, operations on state words implemented for the *NL Tool* and the *CryptoSMT Tool* and integer operations performed during *integer cleanup*. Still this list is not exhaustive.

6.7 Performance Evaluation

Figure 6.1 shows a performance evaluation comparing the various tools in two different experiments. Both experiments compare the computation time of the *MD4* algorithm between the original *NL Tool* implementation, the automated translation for the *NL Tool* and the automated translation for the *CryptoSMT Tool*.

	working before		working now	
tool	*NL Tool*	*CryptoSMT*	*NL Tool*	*CryptoSMT*
general functionality				
single variable assignment	*	-	✓	✓
CNF	-	-	-	✓
Feistel ciphers	-	-	✓	✓
differential `ADD`	-	-	-	✓
loop unrolling	*	-	✓	✓
remove IF statements	-	-	✓	✓
global constants	-	-	✓	✓
integer cleanup	-	-	✓	✓
state word operation				
`OR`	✓	-	✓	✓
`XOR`	✓	-	✓	✓
`AND`	✓	-	✓	✓
$\neg$	✓	-	✓	✓
$\ll, \gg$	✓	-	✓	✓
$\lll, \ggg$	✓	-	✓	✓
integer operations				
$+$	-	-	✓	✓
$-$	-	-	✓	✓
$*$	-	-	✓	✓
$/$	-	-	✓	✓
mod	-	-	✓	✓
$\vee$	-	-	✓	✓
$\wedge$	-	-	✓	✓

Table 6.5: Implemented operations of the tools. (-) meaning it does not work, (*) meaning it works partly, (✓) meaning it works.

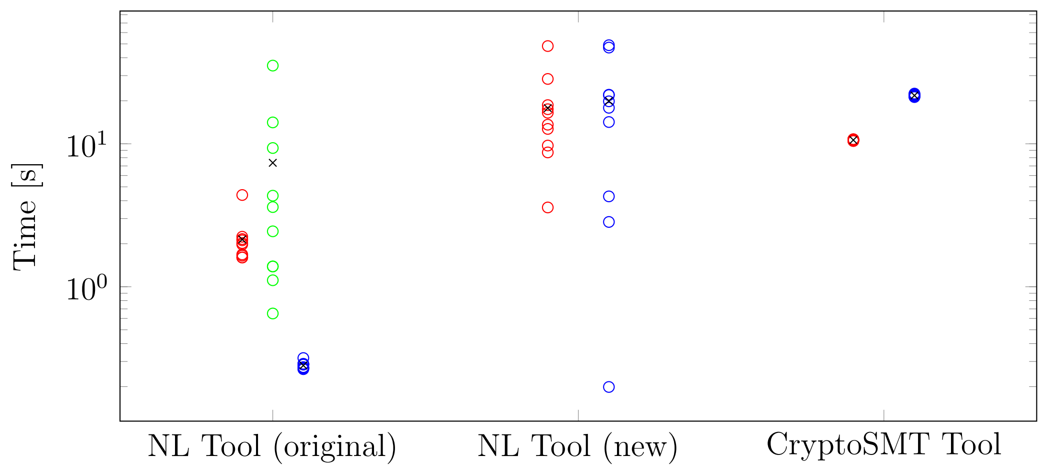

Figure 6.1: Computation time *NL Tool*: original vs. automatically generated version. The red circles (○) mark the experiment with the characteristic file from Table 6.2, the blue circles (○) mark the experiment described in Section 6.7 and the green circles (○) mark the experiment with the original characteristic file from [49]. The black cross (×) marks the corresponding mean value.

In the first experiment, marked with the red circles (○), the differential characteristic file from Table 6.2 was used. In the second experiment, marked with the blue circles (○), the differential characteristic file from Table 6.3 was used. In all experiments the mean value is marked with a black cross (×).

The green circles (○) mark another experiment, only performed with the original *NL Tool* implementation and with the characteristic file from [49]. It was not possible to perform this experiment with the automatically generated translations due to performance reasons.

The original implementation of the *NL Tool* is the fastest one, because it is optimised in terms of speed.

One reason for the reduced performance of the automatically translated algorithms is the temporary variables created during the translation process. Maintain the automated translation process as general as possible, naturally implies unpacking and simplifying of compound or specialized statements. Disassemble these statements results in creating temporary variables which further results in less efficient code.

There are countermeasures against this kind of performance decrease. One, which was implemented is the handling of rotations. They are not disassembled and split into their building blocks, but automatically annotated and translated directly into a rotation operation for the desired output language.

6.8 Challenges and Limitations

6.8.1 Overview

Several challenges occurred during the work on this project. Some of them are listed here followed by their solution, if there was found any. For a couple of them no solution was found. Therefore, they are treated as *limitations* and will be discussed further as *future work* in Section 7.2.

The challenges are grouped in three categories. General challenges regarding the processing of the C-reference implementations are discussed in Section 6.8.2. Challenges regarding the translation process are discussed in Section 6.8.3 for the *NL Tool* and Section 6.8.4 for the *CryptoSMT Tool*.

6.8.2 General

Single Variable Assignment

Single variable assignment did not work for arrays. A new method to remember array entries and reassign them to new variables had to be developed. Since an array has the same name no matter which index is accessed, the new method had to keep track of the corresponding index.

Runtime Performance of the Code Transformations

Previously, a list with all appearing variables and their name to be replaced was maintained during the translation process. With every variable in the algorithm, the framework iterated over the whole list and replaced all occurring variables in the remaining part of the algorithm with the corresponding temporary variable name. With this approach, the same variable could be renamed many times, before it got its final name.

Therefore, a new internal data structure to represent the algorithms' statements before the final translation was developed. This data structure allows data access of $\mathcal{O}(log(n))$ compared to the old data structure with an $\mathcal{O}(n^2)$ performance, where n is the number of statements describing the entire algorithm. This affects mainly the performance of the *single variable assignment* procedure, which was the most time consuming transformation before.

Needed Adaptations for new Algorithms

The framework has to be adapted for almost every new algorithm, because they vary in implementation details very much. Many features are already covered by the framework, but not everything. For example loops with multiple conditions are not handled yet. Therefore, the probability is high that a new algorithm contains some by now not handled code pieces. The framework was built with this knowledge and has many points to hook in easily. For most of the missing features it should be little effort to add them to the framework. Hard is to still preserve its general approach.

Coding Standard

The biggest problem is the variability of different implementation styles. Different developers use different *C* code constructs, where some of them are hard to handle. In general the great diversity of possible implementations is hard to cover e.g. defines, arrays of arrays, method-inlining, pointer usage, global arrays as truth tables, if, ...

One solution to the problem could be to force user of the framework to adapt the *C*-reference implementations. Some code could be replaced by parts which are easier to process automatically. For example, references to structs could be replaced by arrays.

6.8.3 NL Tool

The framework was used to translate the algorithms *MD4* and *Ascon* for the *NL Tool*. Additionally we experimented with *MD5* and *SHA256*.

Performance

For the *NL Tool* the biggest problem is the bad performance. Unfortunately this issue could not be resolved so far. Therefore, it is one of the limitations of the framework. It is possible to generate correctly working code for the tool, although compared to the original implementations it has too many temporary variables needed for the automatic translation process. Especially the *single variable assignment* causes this problem.

In Figure 6.1 the runtime of the automatically generated code compared to the original code for the *MD4* algorithm is shown.

Characteristic Processing

The characteristic file provided to the framework for mapping the input values of the characteristic to the *NL Tool* output is sometimes hard to interpret. One example of such a case are ciphers consisting of a *Feistel network*. They allow a compressed notation of the characteristic file, where not all states of the algorithm have to be fully expressed. For these specific ciphers, the additional annotation `@iv_state_offset` was introduced. This annotation holds a comma separated list of numbers indicating an offset of the presented state from the first state for each represented state in the characteristic file. This approach allows correct mapping of the characteristics *IV* to the states of the *NL Tool*.

6.8.4 CryptoSMT Tool

The framework was used to translate several algorithms like *MD4*, *MD5*, *SHA256*, *Ascon*, *Skein*, and *JH* for the *CryptoSMT Tool*. Additionally we experimented with *SHA-1*, *Grøstl*, *BLAKE* and *Keccak*.

Array Indices Computation

If an index of an array is dependent from the input of the algorithm it is hard to represent this structure within the *CVC* language. All statements have to be defined on compile time, also the indices of the arrays. Considering that the input can be a difference, or unknown at compile time, because it is searched by the tool, this can be a problem.

Therefore, a truth table is generated, which computes the index of an array and stores it in a variable on compile time. A *CNF* is generated to represent the index. This approach is working, but takes a lot of processing time and code size.

Runtime Performance of SAT-Solver

For the *CryptoSMT Tool* it is hard to represent differential propagations, therefore additional constraints can be set to help the tool. For the *ADD*-operation explained in Definition 7 it is implemented already. Constraints for other operations mentioned in Section 5.6.1 could be added.

Chapter 7

Conclusion and Future Work

In this thesis an extended framework for automated cryptanalysis was presented. It can be used to perform a preliminary analysis of arbitrary C-code, although it is best suited for cryptographic algorithms containing a *round function*.

In Section 7.1 a final conclusion is drawn and in Section 7.2 future work is discussed.

7.1 Conclusion

The framework can successfully read C-reference implementations of various cryptographic algorithms, analyse them and translate them into diverse output formats for distinct analysis tools. The C-reference implementations used as input files for the framework are the same typically submitted to cryptographic competitions. Therefore, a wide application range is given.

This work focuses specifically on the *NL Tool* and the *CryptoSMT Tool*. The framework leaves the possibility to be easily extended by another plug-in for a different cryptanalytic tool.

The transformations needed to convert the C-reference implementation into a standard form can be automated for most C-code constructs since the *ROSE* framework used for that purpose is a cross-compiler framework, which understands the whole C-language. Although, it is a lot of work to implement every single feature to generate a standard form which can be generally used for the various translator plug-ins. Therefore, the focus was on the most commonly used once, especially the compositions needed to process the *MD4* algorithm. More exotic compositions still have to be implemented and are therefore not available.

For the *NL Tool*, the automatic code generation process works, but remains impracticable because of its poor runtime.

Automatic code generation for the *CryptoSMT Tool* on the other hand works as expected. Once a differential characteristic is available, the framework can be used to generate a boolean description of the algorithm including the characteristic and test against it.

In conclusion it can be said, that automatic code generation of cryptographic algorithms from their *C*-reference implementation is possible, although more complex as expected in the first place. Depending on the original implementation of an algorithm it can be a fairly straight forward process. The two plug-ins currently available show that automatic code generation works in general, but depends strongly on the tool if it is usable in practice.

7.2 Future Work

Of course, there are many improvements and additional features which can be implemented to enhance the analysis process. Some of them which are considered as promising are listed here:

More efficient implementation of CNF generation

At the moment, the *CNF* generation, described in Section 5.5.5, for input-dependent global const arrays works on a "bit-by-bit basis". This approach results in an extreme amount of generated code, which grows exponentially with the bit size of the constants.

One possible solution to that problem would be the modification of the *CNF* generation algorithm to use a method based on a binary tree. This approach would result in a logarithmic growth according to the bit size of the constants.

Write additional Tool Plug-ins

The framework consists of plug-ins for two tools. Namely the *NL Tool* plug-in and the *CryptoSMT Tool* plug-in. In general, a plug-in for any desired tool can be developed. The more tools this framework supports, the wider its application reaches. For example a plug-in for the *lineartrails Tool* [16], a heuristic tool for linear cryptanalysis, could be added in the future.

To develop a new plug-in, the `ToolTranslator` class of the framework has to be subclassed, which defines the tool's translations. There, translation instructions for each statement can be defined.

Save/load intermediate representation

At the current development state, for each tool, the framework needs to start a separate translation process. This includes all the transformations described in Section 5.5, where one pass for all tools should suffice. This behaviour needs a lot of unnecessary processing time and resources.

One solution to the problem would be to implement a method to store the *ROSE IR* of the processed input file right after the transformation process described in Section 5.5. This configuration could than be loaded for each new translation.

References

[1] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy*, pages 526–540. IEEE Computer Society, 2013.

[2] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C-W Phan. SHA-3 proposal BLAKE. *Submission to NIST*, 2008.

[3] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *ACNS*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.

[4] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000.

[5] Daniel J Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*, volume 8, pages 3–5, 2008.

[6] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (round 2)*, 3(30):320–337, 2009.

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. CAESAR submission: Ketje v2. `http://competitions.cr.yp.to/round3/ketjev2.pdf`, 9 2016.

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. CAESAR submission: Keyak v2. `http://competitions.cr.yp.to/round3/keyakv22.pdf`, 9 2016.

[9] Eli Biham and Orr Dunkelman. A framework for iterative hash functions - HAIFA. *IACR Cryptology ePrint Archive*, 2007:278, 2007.

[10] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993.

[11] Christophe De Cannière and Christian Rechberger. Finding SHA-1 characteristics: General results and applications. In *ASIACRYPT*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[12] CAESAR committee. CAESAR - Competition for Authenticated Encryption: Security, Applicability, and Robustness. `https://competitions.cr.yp.to/caesar.html`. Accessed February 27, 2019.

[13] Bert den Boer and Antoon Bosselaers. An attack on the last two rounds of MD4. In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 194–203. Springer, 1991.

[14] Hans Dobbertin. Cryptanalysis of MD4. In *FSE*, volume 1039 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 1996.

[15] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. Analysis of SHA-512/224 and SHA-512/256. In *ASIACRYPT (2)*, volume 9453 of *Lecture Notes in Computer Science*, pages 612–630. Springer, 2015.

[16] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. Heuristic tool for linear cryptanalysis with applications to CAESAR candidates. In *ASIACRYPT (2)*, volume 9453 of *Lecture Notes in Computer Science*, pages 490–509. Springer, 2015.

[17] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round3/asconv12.pdf`, 2016.

[18] Morris J Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

[19] Maria Eichlseder, Florian Mendel, Tomislav Nad, Vincent Rijmen, and Martin Schläffer. Linear propagation in efficient guess-and-determine attacks. In *WCC*, pages 142–149, 2013.

[20] Maria Eichlseder, Florian Mendel, and Martin Schläffer. Branching heuristics in differential collision search with applications to SHA-512. In *FSE*, volume 8540 of *Lecture Notes in Computer Science*, pages 473–488. Springer, 2014.

[21] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein hash function family. *Submission to NIST (round 3)*, 7(7.5):3, 2010.

[22] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[23] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl - a SHA-3 candidate. *Submission to NIST (round 3)*, 2011.

[24] Fabian Golser. Verification of electronic signatures on Android. Bachelor's thesis, University of Technology, Graz, 2014.

[25] Christoph Hechenblaikner. Automated cryptanalysis of new authenticated ciphers. Master's thesis, University of Technology, Graz, 2014. `http://diglib.tugraz.at/automated-cryptanalysis-of-new-authenticated-ciphers-2014`.

[26] Jaap-Henk Hoepman and Bart Jacobs. Increased security through open source. *Commun. ACM*, 50(1):79–83, 2007.

[27] Jérémy Jean. TikZ for cryptographers. `https://www.iacr.org/authors/tikz`, 2016. A collection of tikz figures for important cryptographic algorithms.

[28] David Kahn. *The Codebreakers: The comprehensive history of secret communication from ancient times to the Internet.* Simon and Schuster, 1996.

[29] Auguste Kerckhoffs. La cryptographic militaire. *Journal des sciences militaires*, 9:5–38, 1883. Kerckhoffs' principles.

[30] Dmitry Khovratovich and Ivica Nikolic. Rotational cryptanalysis of ARX. In *FSE*, volume 6147 of *Lecture Notes in Computer Science*, pages 333–346. Springer, 2010.

[31] Lars Knudsen. DEAL - a 128-bit block cipher. *complexity*, 258(2):216, 1998.

[32] Stefan Kölbl. CryptoSMT: An easy to use tool for cryptanalysis of symmetric primitives. `https://github.com/kste/cryptosmt`.

[33] Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. In *FSE*, volume 2355 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2001.

[34] Lawrence Livermore National Laboratory (LLNL). ROSE compiler framework, open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale applications. `http://rosecompiler.org`. Accessed February 27, 2019.

[35] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *EUROCRYPT*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 1993.

[36] Florian Mendel, Tomislav Nad, and Martin Schläffer. Finding SHA-2 characteristics: Searching through a minefield of contradictions. In *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 288–307. Springer, 2011.

[37] Florian Mendel, Tomislav Nad, and Martin Schläffer. Improving local collisions: New attacks on reduced SHA-256. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 262–278. Springer, 2013.

[38] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[39] Ralph Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford University, Stanford, 1979.

[40] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[41] Koji Nuida, Takuro Abe, Shizuo Kaji, Toshiaki Maeno, and Yasuhide Numata. A mathematical problem for security analysis of hash functions and pseudorandom generators. *Int. J. Found. Comput. Sci.*, 26(2):169–194, 2015.

[42] National Institute of Standards and Technology (NIST). Cryptographic hash algorithm (SHA-3) competition. `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`, 2007. Accessed February 27, 2019.

[43] National Institute of Standards and Technology (NIST). Cryptographic hash algorithm (SHA-3) competition on cr.yp.to. `https://competitions.cr.yp.to/sha3.html`, 2012. Accessed February 27, 2019.

[44] National Institute of Standards and Technology. The SHA-1 algorithm. `https://archive.org/stream/federalinformati1801nati`, 4 1995. Federal Information Processing Standards Publication FIPS 180-1.

[45] Daniel J. Quinlan. ROSE: compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[46] Ronald L. Rivest. The MD4 message digest algorithm. In *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1990.

[47] Ronald L. Rivest. The MD4 message-digest algorithm. *RFC*, 1320:1–20, 1992.

[48] Ronald L. Rivest. The MD5 message-digest algorithm. *RFC*, 1321:1–21, 1992.

[49] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. New message difference for MD4. In *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 329–348. Springer, 2007.

[50] Ákos Seress. *Permutation group algorithms*, volume 152. Cambridge University Press, 2003.

[51] Hongjun Wu. The hash function JH. *Submission to NIST (round 3)*, 6, 2011.

[52] Thomas Zefferer, Fabian Golser, and Thomas Lenz. Towards mobile government: Verification of electronic signatures on smartphones. In *EGOVIS/EDEM*, volume 8061 of *Lecture Notes in Computer Science*, pages 140–151. Springer, 2013.