First bachelor thesis

„Popular Cryptographic Hash Functions“

Completed with the aim of graduating with a
Bachelor (FH) in Telecommunications and Media
from the St. Pölten University of Applied Sciences Telecommunications and Media degree course

under the supervision of

Bakk.rer.soc.oec. Bernhard Fischer

Completed By

Marcus Nutzinger
tm0420261102

St. Pölten, on June 18, 2006 Signature:

Declaration

I declare that

- this thesis is all my own work, that the list of sources and aids is complete and that I have received no other unfair assistance of any kind,

- this thesis has not been assessed either in Austria or abroad before and has not been submitted for any other examination paper.

This piece of work corresponds to the work assessed by the appraiser.

St. Pölten, June 18, 2006
Place, Date

Signature

Abstract

This paper faces popular cryptographic hash functions, their design and their mode of operation. It will give an overview of used implementations and an overview of criteria for algorithms to be secure.

To give a good basis for this paper, some basic terms (like *hashing in general*, *hashing and cryptography*) are defined at the beginning. Then, the general aspects of hash functions in cryptography are faced. The main properities will be described and some hashing examples are given. Also the application areas of cryptographic hash functions will be discussed. At the end of this chapter, some possible points for attacks on hash functions are mentioned.

After covering these common basics of cryptographic hash functions, this paper will go into the depth of some of the most popular algorithms (like MD5 and SHA-1). For each algorithm the design and the mode of operation are shown. It will also be pointed out, if there has been any attacks on algorithms and how these attacks looked like.

The final part will try to point out some criteria an algorithm has to meet to be considered as secure. It will also give a brief overview of the cryptographers point of view on what algorithms should be used nowadays.

Contents

Chapter 1

Introduction

Cryptography has a very big domain of primitives. The most versatile ones are hash functions. Imagine that one stands for the problem of storing data not in plain text but just a representation of the data as a stream of digits. For this (and other applications), a hash function is consulted, which also has the benefit that no key is needed to convert plain text into senseless character streams. Furthermore are hash functions one-way functions, so once converted into a hash stream, no one can ever recompute the original input again.

The world needs such algorithms, because security gets more and more important nowadays, mainly when using internet protocols. So this paper will explain the way, how hash functions operate in general and what's their use in the security sector. Because of the fact, that there exist many different implementations of algorithms in the real world of cryptographic hash functions, this paper will describe some of the most popular algorithms and how they work.

The following algorithms will be faced:

- MD4
- MD5
- SHA
- RIPEMD
- Whirlpool
- Tiger

This paper faces the following research hypotheses:

- How are hash function basically designed and how does they work?
- What hash functions can be rated as secure today?

To understand the anwers to these questions, some basic knowledge is needed. So this paper will give some fundamentals to cryptographic hash functions in the beginning.

To answer the first hypothese, it's important to understand the design of the popular hash functions which will be described. These informations are provided in chapters 3 and 4. Furthermore it has to be specified, which hash functions are rated as secure to answer the second hypothese. So some reflections of the security from described algorithms and criteria for the design of secure hash algorithms are given in chapter 5.

Chapter 2

Definitions

2.1 Hashing

Hashing is the process of converting an input value in a different output value (which is in most cases of fixed length). This is done for different purposes, e.g. to generate an index in a data structure, to produce a fixed length output stream and hide the *real* message (input), to generate the checksum in the TCP header, etc.

2.2 Hash function

A hash function is a function, whose algorithm provides the transformation of variable length data into streams of fixed length and domain. In general, the ouput is represented as a random string of letters and numbers. This can be seen in figure 2.1. Here, three different output streams were generated, independent of the length or contents of the input values. Hash functions can be used for differnt applications. [14] [15]

Some examples are:

hash tables In computer sience, a hash table is seen as a data structure to associate keys with values. So the main challenge is to look up for a value represented trough a given key. A hash function is used to *hash* the key, so the application accessing the hash table can locate the desired value. [16]

error correction A hash function can be used to detect errors in transmissions. The algorithm is computed for the data at the sender, and the hashvalue is sent with the data. Then the alogrithm is performed again at the receiver, and if the hash values don't match, an error has occurred at some point during the transmission. This is called a redundancy check. Examples for redundancy check implementations are a checksum or the cyclic redundancy check (CRC). [15] [17]

verifying file integrity Hash functions are often used to compute the hashcode of a file. Anyone who gets the file (e.g. downloads it from an FTP-Server) can now compute the file's hashcode and compare it with the hashcode which was provided by the sender of the file (or the operator of an FTP-Server). If the values differ, then the downloader knows that his file got corrupted. On the other hand, if the two hashcodes are the same, he exactly knows that he downloaded a correct file. [18] [19]

cryptography This is the application area of hash function, this paper will face. See the next section for definitions.

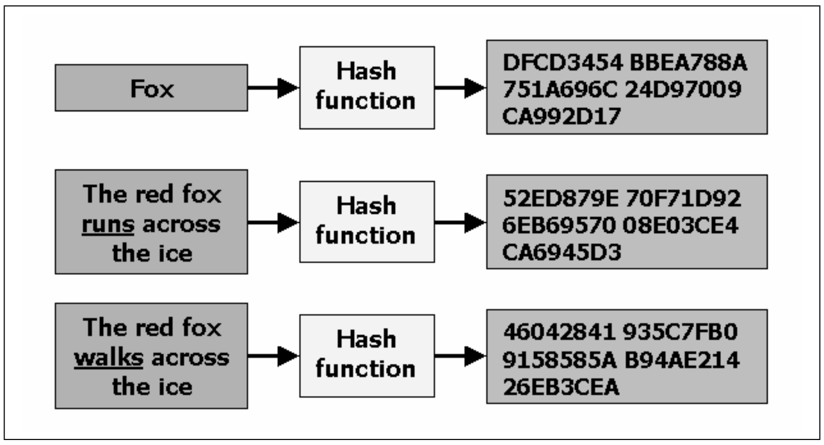

Fox
Hash function
DFCD3454 BBEA788A 751A696C 24D97009 CA992D17
The red fox runs across the ice
Hash function
52ED879E 70F71D92 6EB69570 08E03CE4 CA6945D3
The red fox walks across the ice
Hash function
46042841 935C7FB0 9158585A B94AE214 26EB3CEA

Figure 2.1: A hash function at work [20]

2.2.1 One-way function *H*

In mathematical theory, a hash function is often called function H with its input M. This function returns a fixed length ouput, called h, where m represents this fixed length. This paper will also use this nomenclature.

2.3 Hashing and cryptography

In cryptography, hash functions are also used to generate a fixed length output stream from a given (random) input. The output is - in this case - often called *message digest* or *digital fingerprint*. For cryptographic purposes, the requirements on hash functions are much stronger than in other applications. [15]

There are, for example, two main characterizations for cryptographic hash functions: [20]

- they have to be one-way functions
- there shouldn't happen any collisions

The meaning of these points will be discussed later with all the other requirements in chapter 3.

2.3.1 Encryption vs. digest

In this case it's very important to establish the border between these two terms. It's clear that the domain of hash functions also extends into the big field of cryptography. But this doesn't mean, that these algorithms are encryption algorithms like *DES* or *AES*. The big difference is, that encryption algorithms cipher the input value with a given key to generate a ciphertext. This text can easily be decrypted at any time, when someone knows the secret key. So encryption is called a *two-way operation*.

On the other hand are standing hash algorithms. When hashing input values, the length of the output will always be the same (see figure 2.1 on page 3 again). Also, it isn't possible to calculate the underlying message from a given hashcode (or at least it shouldn't be possible - Remember the two main requirements for cryptographic hash functions, section 2.3.). So hashing can be called a *one-way operation*.

2.4 The attacks

Real attacks and the weakness of hash functions will be described later in the chapters 4 and 5. But at this point it could be important to define what it means to attack a hash function.

Attacking a hash function can't only be done after knowing the algorithm. It has very much to do with cryptoanalysis. The function must be exactly analyzed and the attack has to meet one well defined point inside the algorithm. So to do this, it is nessecary to know all the mathematic sequences to get a good outcome. Because of all the complexity, this paper will just show some examples of attacks on algorithms (chapters 4 and 5) and will not go into the deep of attack-design.

Chapter 3

Cryptographic hash functions in general

A hashcode can be seen as a *digital fingerprint* for a random input, that represents this value. Mathematically, it's denoted by:

$$h = H(M)$$

where H is the hash function, M the input value and h the hashcode. Now it's important to define, how to come to an output value. One widespread algorithm is the Merkle-Damgård construction.

3.1 Merkle-Damgård construction

Chapter 2 said, that hash functions produce a fixed length output. Now it has to be clarified, how this process is done. One construction for this problem was found by Ralph Merkle and Ivan Damgård and is used by most of the common hash functions (like MD5 and SHA-1). As displayed in figure 3.1, every input value is broken up into l equal blocks (x_1 to x_l). Then, every block serves as input for a compression function (called f) to get a smaller output. The second input for the compression function is the output of this compression function from the block before the actual one (x_{l-1}). So in the first round, the compression function needs another second input value, which is called *initialization vector* (IV) and represents a fixed value. This means, calculating the hashcode of block x_i runs as follows: (see [1], 1996, p. 431)

$$h_i = f(x_i, h_{i-1})$$

The last block is padded with additional bits to get the same block size as all other blocks have. This process is called *length padding.* At the end, the ouput of the round actually represents the hashcode of the whole message. To harden the hash function, this value additional serves as input for another function g, the *finalisation function.* This function can perform many different operations to its input and is often built of the compression functions f. [21]

Thus, the singularity of each hash function lies in the design of compression functions f and finalization functions g. And, furthermore, the block size an input value gets fragmented in, is determining. Typical block sizes are 64 or 128 bytes.

3.2 Background

Now its time to face the requirements of hash functions. First, the possible problems are pointed out, to show the hash functions properties for cryptographic usage afterwards.

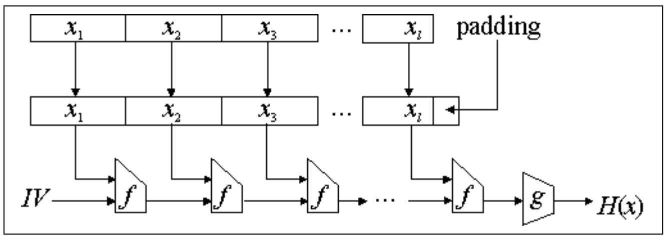


Figure 3.1: Schematic of the Merkle-Damgård construction [21]

3.2.1 Main challenges

A cryptographic hash function is feasible, if it acts nearly like a random function while still being deterministic. This is, of course, the point of view for a *ideal hash function*. But however, a cryptographic hash function is seen as insecure, if one of the following points are practicable: [20]

- finding a message which produces a given hashcode
- finding two different messages which compute to the same hashcode

This is of course no exhaustive listing of scenarios, which would make a hash function insecure. But these two points are the most (or at least very) important ones. So it has to be ensured, that a hash function avoids such happenings and to guarantee this, some requirements are nessecary.

3.2.2 Cryptographic properties

There can't be found any general description of properties for cryptographic hash functions. This has its origin in the big domain of these algorithms. Because if the prerequesits are somehow different for different environments, no standard is needed.

But one main characteristic is equal in all systems:

- Given M, it is easy to compute h

In addition, there have been found three demands which are generally considered: [20] (see [1], 1996, p. 429)

Preimage resistant Given h, it is hard to compute M such that $H(M) = h$

Second preimage resistant Given M, it is hard to find another message, M', such that $H(M) = H(M')$

Collision resistant It is hard to find two random messages, M and M', such that $H(M) = H(M')$

Preimage and second preimage resistant can be combined to the *one-way* property of hash functions. The term *preimage* means, that one (or even two) parameters are known. This differs from a collision, because there are no fixed parameters. So, when talking about attacks, a collision attack is easier than finding matches to given preimages (see section 3.5).

These main attributes can also be taken to measure the security of a hash function. Because a hash algorithm is more and more useless, if there can easy be found collisions. So these three terms will be described in more detail now, together with their impact on the weakness.

Preimage resistant measures, how difficult it is to find a message M which hashes to a given hashcode h. Breaking this would mean, that the underlying algorithm doesn't really provide a one-way function anymore. This is, of course, not 100% correct, because if one can find a message to a given hashcode, it isn't said, that the hashcode was computed with the found message. But however, a weak preimage resistance would be a big problem, because this knowledge could be used to attack password databases. The attacker may never find out the real passwords, but if he finds a message which hashes to the same value as the password, he can enter the system. [18]

Second preimage resistant measures the difficulty of finding a message M' which hashes to the same value as a given message M. This property reminds of the preimage resistance. But here, the attacker knows almost more, because a message *and* its hashcode are given. So to break this, it has to be found a message, which hashes to the same hashcode as the other message (this is the same as before), but the other message is familiar (this is new compared to the preimage resistance). A weak second preimage resistance would mean, that an attacker could e.g. distribute corrupted software where the packet produces the same hashcode as the „good“ software packet. Because when downloading from an FTP-server, the only method to check the originality of the packet is to check its hashcode. [18]

Collision resistant measures, how hard it is to find two messages which produce the same hashcode. An attacker here would like to find any two messages, but the result hashcode isn't very important, just that it's the same for both messages. With a weak collision resistance, e.g. digital signed documents would become suspect. The attacker could infiltrate another message (which computes the same hashcode as the signed document) and simulate that it's the signed one. [18]

Now the general demands are defined. Another factor, which should be discussed, is the bit length of the output stream. In most algorithms, this number is fixed for every hashcode, independent of the input value. The „right“ number of bits to choose is very much connected with possible attacks, so this topic will be discussed later in this chapter in section 3.5. Also an important property of cryptographic hash function is the randomness of the output value. Examples and a description is subject of the next section.

3.3 Hashing examples and the avalanche effect

It's time to demonstrate a hash function at work. For the following examples, the *md5sum* script is used, which is installed on most Unix and Linux environments. Three hashcodes of different files will be calculated first.

```
marcus@nuuz ~/test $ ls -l
insgesamt 103757
-rw-r--r--  1 marcus users        47 18. Apr 14:26 file1
-rw-r--r--  1 marcus users       130 18. Apr 14:27 file2
-rw-r--r--  1 marcus users 106135552 18. Apr 14:27 file3
```

The output shows these files, one is very big and the others have a standard textfile-size. The next step is to compute the hashcode out of these files.

```
marcus@nuuz ~/test $ md5sum *
ef814a3989fb2d6f25f90efc38db87a6  file1
94d43802bc7b21c45cbf0e7417d70c0e  file2
b3d4c6676f0bdff1e3bfca5e8e44ff27  file3
```

As it's displayed in this output, all hashcodes are different and every hashcode has the same length, independent of the different file lengths. The next example will first look at the output of two files f1 and f2.

```
marcus@nuuz ~ $ cat f1
popular cryptographic hash functions
marcus@nuuz ~ $ cat f2
qopular cryptographic hash functions
```

The content of these two files is nearly the same, but in file f2, the first word begins with a „q“. So the file f2 differs just in one bit, because „p“ conforms to `0x70` and „q“ to `0x71`. The next ouput displays their hashcodes.

```
marcus@nuuz ~ $ md5sum f1 f2
1e6ebf94e61a6d9cd0cf6fab4d9b80dd  f1
3efc15d5a929c49e005ec91d0386b9e9  f2
```

From the definitions above it's clear, that different input values generate different output streams. In this example, the two input values just differed in one bit, but the given hashcodes are although very different. This shows an important fact - the avalanche effect. [18]

Avalanche effect This effect is pointed out with the last example. It appears when two slightly different input values generate two significantly different output values. This property is very important for cryptographic hash functions. If the avalanche effect doesn't occur to a certain degree, then it may be easy for cryptoanalysts to give prophecies about the input value just because of the ouput stream. So another main design demand for cryptographic hash functions is to find an algorithm follows this effect. One attribute of hash functions, which has impact on this effect, is the block size the input value gets splitted in. [22]

3.4 Application areas

A cryptographic hash function is used in many different environments. But before speaking about what cryptographic hash functions can do, it has to be specified, which hash algorithms can't be counted to the cryptography.

3.4.1 Dissociation: non-cryptographic algorithms

As said in chapter 2, hash functions can be used for many different operations. Also, the output of various algorithms is called *hash* or *hashcode*. Well known algorithms are *checksums* or *cyclic redundancy check* codes (CRC). These algorithms produce an output of fixed length, from any input. With this definition, they can be added to the family of hash functions. But they have to be well distanced from the cryptography. Checksum algorithms are very different from algorithms used in cryptographic hash function. They use a simple mathematical structure, so it's e.g. easy to change the message without chaning the resulting checksum. So CRCs are just used to ensure the detection of errors in a message, whereas cryptographic hash functions can be used to ensure message integrity. [23]

3.4.2 Message authentication codes (MAC)

A MAC generates a tag out of an input value and a secret key. So it gets difficult for attackers to calculate a real message-key pair. Hash functions can be used to create MAC functions, which are called *keyed-hash MAC* (HMAC). HMACs can be used to verify the integrity *and* the authencity of a message. So opposite to hash algorithms, they provide the authenticity using a secret key. This key is added to the message before hashing it. So the receiver can verify, whether the message really comes from the sender or not. HMACs are e.g. termed HMAC-MD5, depending on the underlying hash function. So the security of a HMAC is also based upon the security of the used hash function and the MAC value has the same length as the hash output. [20] [19] [24]

The MAC value is calculated as follows:

$$h = H(key1, H(key2, m))$$

where H is the underlying hash function and *key1*, *key2* are two specially calculated variants of the secret key. [25]

3.4.3 Password storing

In most applications, passwords aren't stored in plain text because of security advisements. The text is hashed before e.g. storing it into a database, so an attacker can't find out the clear text passwords because of the one-way property. To check, whether a user is allowed to log into the system, the given password is hashed and compared to the hashcode from the password-database.

It's important to say, that there exist additional features when storing passwords. It's e.g. possible to *salt* the ouput value. This happens as follows: [26]

1. A user enters his password
2. The plain text password gets hashed, using any hash function (e.g. MD5).
3. After this, an *unique identifier* (UID) is generated, e.g. using some hardware information, time and date.
4. Then, this UID is added to the hash output value.
5. The resulting value is hashed again with any hash function.

So when storing this value in a database, any dictionary attack gets very difficult. Because when storing only the hashed password, it's generated an explicit connection between the user and his hashed password (because of the one-way property - every password hashes to another value). But when hashing the password-hash together with an UID, an attacker first has to create a dictionary with possible UIDs to attach them to his dictionary of password-hashes.

3.4.4 Digital signature

Signing a document is like placing the autograph under it. This process is often used when sending e-mails and has actual more to do with public-key cryptography than with hash functions. But the following example will make clear, in which way hash functions are used when signing documents.

Bob (any person) wants to send an e-mail to Alice (any other person). This e-mail has a big attachment, some files, pictures, etc. One way, to ensure, that the e-mail really comes from Bob, is to encrypt the whole mail with Bob's private key. So Alice can decrypt it only with Bob's public key and can be sure, that the mail comes from Bob. But the process of encryption and decryption is very time-consuming when having big messages. So Bob has another possibility. He can compute the hashcode of the whole e-mail. This e.g. 128 bits (when using MD5) can easily be encrypted with Bob's private key. Now, Bob sends the e-mail *and* the encrypted hashcode (which is called *signature*) to Alice and she can compute her own hashcode out of the message. Then she compares the hashcode from the decrypted signature with the calculated one. So it can be ensured again, that the message wasn't modified since Bob signed it. [27]

3.4.5 Other uses

The last three applications are very common. But there are, of course, still other ones, which are used widely.

Message integrity check This employment is similar to the signing of documents. Hash functions are used to ensure message integrity, because one can calculate a hashcode from a message. This hashcode can be transferred or kept together with the message (e.g. a file). So when receiving or re-reading the message, it's easy to check, whether the content has been modified or not. The only operation to do is to calculate the hashcode from the message once more and compare it to the existing one.

The intrusion detection software *Tripwire* uses this property for example to avert malware and attacks. [20]

Commitment scheme This term means, that it should be ensured for one person to commit to something without telling it anyone. The example will show, how this works: [20]

1. Alice wants to make sure that Bob gets a proof of any message, e.g. a mathematical solution (called b), without the chance to read b.
2. So Alice computes a random number (called nonce, n) and appends it to b. Then she computes the hashcode out of this value.
3. This hashcode gets Bob, as proof, that Alice really solved the problem.
4. When Bob also knows the solution, he can ask Alice about the nonce and calculte the hashcode for himself. So he knows, whether he and Alice had the same result and that Alice didn't cheat.

Shorten input values This feature is often used for any cryptographic operation, because these operations are much faster using a short input than a large one. So it's practical to hash a message and just use this e.g. 128 bit hash as input for any cryptographic algorithm. This property is used for example when signing a digital document (see 3.4.4).

3.5 Ways to attack

The last section pointed out, that cryptographic hash functions have their use in many different applications. They are widly used because of their two security properties: they are *one-way* and *collision-free*. This section will discuss generic, implementation-independent, attacks on hash functions.

Generally, the two properties can get attacked: [28] [6]

Against the *one-way* property These two variants are very similar, because attacks, that can find one type of preimage can often find the other one as well: [28]

- A *first-preimage attack* means, that an attacks wants to find a message that hashes to the same value as a given hash value h in fewer than 2^L trials.
- A *second-preimage attack* means, to find a message M', which hashes to the same value as a given message M in fewer than 2^L trials.

Against the *collision-free* property Here, the attacker wants to find two messages M and M', which hash to the same value in fewer than $2^{L/2}$ trials (L is the output length of the used hash function).

There are still other points to attack. It's e.g. possible to analyse the underlying compression function f and build an attack out of these considerations. Some mechanisms are shown afterwards (see 3.5.3). But before, a look at the current, real world situation is done.

3.5.1 Current situation

Most of the currently known attacks are against the *collision-free* property, because the few preimage attacks aren't practical nowadays. Here it's important to say, that finding two messages, that hash to the same value, is much easier than finding two message that hash to the same value *and* are both useful for real world attacks. But it's to note, that many protocol designers, which use hash functions in their operations, made the protocls immune against collision attacks. So it's much more harder to succeed an attack on a real world protocol which uses hash functions. [28]

Bruce Schneier and Paul Hoffman summarize the current situation in the following way:

> „Both MD5 and SHA-1 have newly found attacks against them, the attacks against MD5 being much more severe than the attacks against SHA-1.
>
> The attacks against MD5 are practical on any modern computer.
>
> The attacks against SHA-1 are not feasible with today's computers, but will be if the attacks are improved or Moore's Law continues to make computing power cheaper.
>
> Many common Internet protocols use hashes in ways that are unaffected by these attacks.
>
> Most of the affected protocols use digital signatures.
>
> Better hash algorithms will reduce the susceptibility of these attacks to an acceptable level for all users.“ [28]

3.5.2 Security of hash constructions

The three main requirements on cryptographic hash functions were defined in 3.2.2. But however, a hash function which has these characteristics may still be insecure, which can be seen here. It was defined in section 3.1, that most of today used hash functions are using the Merkle-Damgård construction. This is because of the fact that Merkle and Damgård could prove, that if the compression function (f) is collision resistant, then the whole hash function is collision resistant. But also this construction has its security weaknesses: [21]

- An attacker just knows the hashcode (h) and the length of the input message (*len(m)*). Now he can choose any message M' and compute $H(M||M')$ to find an appropriate h (called *length-extension*).
- Also with this construction, second preimage attacks are always much more efficient than brute force attacks.
- When found a collision, it's easy to generate a *multicollsion.* This indicates a situation, where one has a hashcode which matches to more than two messages.

After focusing the characteristics of this construction, it has to be said, that the designer of a hash function has also very much influence of its security. In section 3.1 was defined, that the last input block gets length padded. This is done with a *padding rule.* It was also defined another element, the *inizialization vector* (IV), which serves as input for the first pass of the *compression function* (f). These two already called elements have much influence on the security of the hash function. It's important to have a distinct padding rule. This means, that there can't exist two messages which produce padded the same message. So it's recommended to use a padding, which encodes the length of the original input message. This mechanism is called *„MD-strengthening“* (after its inventors Merkle and Damgård). Furthermore, the IV should be defined when designing the hash function, because it shouldn't be possible for an attacker to choose any self-defined values for the IV. [21] [6]

3.5.3 Generic attacks

It's already said, that attacks are possible on different properties of hash functions. Here, two classes are described: [6]

- Attacks, which depend only on the size of the output bits of a hash function
- Attacks, which depend on the specifications of the compression (or *round*) function f

Attacks depending on the number of bits

This class of attacks just want to find a right hashcode out of the domain of possible output values. When assuming a random function as hash function, the attacker has to try out every possible value to find a valid hashcode. Hash functions are nearly random function, but not at all. So these attacks get more and more successful when using a weak hash function, e.g. with a little number of output bits. These attacks are also called *brute force attacks*. Here, an attacker probably doesn't have a clue. The chance to find a valid output value lies somewhere about $1/2^{n/2}$, where n indicates the number of output bits. So statistically, one has to try half of the possible values to find a valid one. This class combines the attacks against the two properties: *one-way* and *collision free*. [6]

Preimage and second preimage attack These are attacks against the *one-way* property. An opponent selects any random chosen message M and hopes that $H(M) = h$, where h is a given hashcode. With a real random hash function, an attacker has to try 2^n possible values. This is, also with a low number of bits, very time-consuming. But with parallel attacks and other improvements, this attacks can get feasible with just a few bits for the output value. So this means, that the output length should be at least 64 bits. [6]

Birthday attack This is built on the birthday paradox and is a typical attack against the *collision free* property. Here, any two messages M and M' have to be found, such that $H(M) = H(M')$. Describing the *birthday paradox* will also make clear, what it takes to do such an attack.

Birthday paradox The *birthday paradox* states that if there are 23 or more people in a room then there is a chance of more than 50% that at least two of them will have the same birthday. [29] This is a standard statistics problem, it's called paradox because many people can't believe at first sight, that the chance is so big with this little number of people.

When attacking a hash function, this can be applied as follows: It's much more easier to find a collision (= two random messages that hash to any same hashcode) than finding a message M that hashes to a given hashcode. Because of this, the collision attack is called *birthday attack*. Here, an attacker would just need to try $2^{n/2}$ possible values which is much fewer work than attacking the *one-way* property. One can see, that the number of bits get diveded by two against the one-way attacks when talking about values to try. So to be secure, it's recommended to choose a bit length which is twice as much as it would possible be needed. E.g. when one wants to let the chance to find a collision just be $1/2^{80}$, then the output of the hash function should have a length of 160 bits. (see [1], 1996, p. 166)

Some thoughts about the number of bits

The above described class just focusses the number of bits within their ways of attack. Generally it can be said, that a 64 bit output value would be too small, because of the weakness against a *birthday attack* (with only 2^{32} trials needed, which can be done in a short amount of time). Actual used values are located between 128 bits and 256 bits, few algorithms generate an output value of 512 bits. Considering *Moore's law*, which says that computing power available for a given cost doubles every 18 months, makes clear, that 128 bits aren't idal for the security of the alogrithm. It can be said, when noticing computing power and *Moore's law*, that a hash function should have an output value of 160 bits to be collision-resistant. So 2^{80} operations aren't operable with a finite budget and time. (see [4], 2004, p. 30) (see [1], 1996, p. 430)

The next class which will be pointed out doesn't focus on the number of output-bits, but on the properties of the compression function *f*.

Attacks depending on the compression function properties

These attacks are also called *chaining attacks*, because they look at intermediate chaining variables (state of variables like H_i, x_i during the operation, see figure 3.1) and not at the whole hash output. The idea of these attack variants is to construct a preimage or second preimage.

Fixed point attack In a compression function *f*, a fixed point is a point where it applies that $f(H_i, x_i) = H_{i+1}$ and $H_{i+1} = H_i$. So an attacker who finds this fixed point, can insert an arbitrary number of blocks with the value x_i. When this can be done, a (second) preimage is found, because after inserting blocks, the hash result won't differ. But this attack is only possible, if the chaining variable (H) can be set to this specific value H_i. If the algorithm allows it, this can be done very easy by modifying the inizialisation vector *IV*. So a fixed IV and the *MD-strenghtening* (see 3.5.2) make this attack much more complex. [6] (see [4], 2004, p. 34)

Meet-in-the-middle attack This attack is a variation on the birthday attack, but it compares intermediate chaining variables instead of the whole hash result and it enables an attacker to construct (second) preimages. For example, a second preimage for message M is searched. The IV (H_0) and the hash value (*H(M)*) are fixed. Now, the attacker has to search an attack point between two blocks in the chain, for another input M'. When found a point where $H(M') = H(M)$, he can start with the IV and go back in the chain.
That this attack can work, one must be able to go backwards in the chain. The compression function f has to be inverted, this means a pair H_i and x_i has to be found, such that $f(H_i, x_i) = H_{i+1}$ for a given value of H_{i+1} (this is also called a pseudo preimage). (see [4], 2004, p. 35)

Correcting block attack When searching a (second) preimage for a message *M*, an alternative message M' has to be found, such that $H(M) = H(M')$. When using this attack, the attacker chooses one input block x_i and replaces it with another block ${x'}_i$, such that $f(H_i, x_i) = f(H_i, {x'}_i)$. In the case that all other blocks of M' have the same value as the corresponding blocks of *M*, the two hash results are the same.
This attack is applicable, when, given H_i and H_{i+1} (any values from the chain), a block ${x'}_i$ can efficiently be found such that $f(H_i, {x'}_i) = H_{i+1}$. (see [4], 2004, pp. 34,35)

Chapter 4

Real world implementations

This chapter will point out some of the most relevant hash alogrithms today. Table 4.1 shows algorithms which will be discussed.

Name	Output bits	Block size	Author(s), year
MD4	128	512	Rivest, 1990
MD5	128	512	Rivest, 1992
RIPEMD	128	512	RIPE Consortium, 1992
RIPEMD-128	128	512	Dobbertin, Bosselaers, Preneel, 1996
RIPEMD-160	160	512	Dobbertin, Bosselaers, Preneel, 1996
SHA-0	160	512	NIST/NSA, 1993
SHA-1	160	512	NIST/NSA, 1995
SHA-256	256	512	NIST/NSA, 2000
SHA-512	512	1024	NIST/NSA, 2000
Tiger	192	512	Anderson, Biham, 1996
Whirlpool	512	512	Barreto, Rijmen, 2000

Table 4.1: Lengths and designers of popular algorithms [30]

First, the MD4 and MD5 algorithms are described. Both are very much in common, MD4 more as basis for other algorithms, whereas MD5 is still used in many applications. After this the SHA algorithm variations are pointed out, which were invented from the NSA. Tiger and Whirlpool are also discribed, like RIPEMD variants, which were developed in Europe.

4.1 MD4

The *Message Digest 4* algorithm was designed by Ron Rivest. It was first proposed in 1990. MD4 generates a 128 bit hash code out of any input value. Rivests design goals were: [31] (see [1], 1996, pp. 435,436)

Security It's infeasible to find two messages hashing to the same value. Hence the most efficient attack is brute force. Also, the security of this alogritm isn't based on any mathematical assumptions like the difficulty of factoring (*direct security*).

Speed, simplicity and compactness The algorithm is based on a simple set of bit operations on 32-bit operands and so it is quite fast. It's also coded as simple as possible without large data structures (like substitution tables).

Favor little-endian architectures MD4 is fast on microprocessor archtectures, it's optimized specifically for Intel microprocessors.

The MD4 algorithm is placed in public domain. This means, that it's accessible for everybody for extensions and improvements. That's also the reason, why this algorithm served as basis for the design of some other cryptographic hash alogrithms. For example, MD5, the SHA-algorithms and RIPEMD are all variants of MD4.

4.1.1 Algorithm description

The algorithm gets any input message *M*, represented by b bits - m_0 to m_{b-1}. There are five steps performed to compute the hash result *H(M)* out of the input. [31]

Step 1: Append padding bits

Every message has to be a multiple of 448 (in bits). To get this, one „1“ bit and various „0“ bits are added to the end of the message. When having the message as a multiple of 448, it's exactly 64 bits away to be a multiple of 512, what's important for the next step. [31]

This step is always performed, even if the number of bits is already a multiple of 448. So one to 512 bits may be added.

Step 2: Append length

Now, a 64-bit representation of the length from the original message is added to the padded message. So the result is, that the message is now a multiple of 512. It's not very likely, but it could be that the original message is greater than 2^{64}. In this case, only the 64 low-order bits are added. The whole message can now be seen as a multiple of 16 32-bit words. This is called M_0 to M_{N-1}, where N is the number of 32-bit words, which represent the message. [31]

Step 3: Initialize buffer

MD4 uses a fixed inizialisation vector of four words. The hexadecimal values are as follows: [31]

word A	`67 45 23 01`
word B	`EF CD AB 89`
word C	`98 BA DC FE`
word D	`10 32 54 76`

So C and D are just the reversed A and B.

Step 4: Process message in 16-word blocks

The algorithm works in three rounds, where every round consists of 16 steps. So there are 48 steps which have to be done, and in each step, one of the four buffer values (A, B, C, D) is updated. For this, some nonlinear functions are used, depending on the round. These functions are: [31]

$$\begin{aligned} F(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\ G(X,Y,Z) &= (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \\ H(X,Y,Z) &= X \oplus Y \oplus Z \end{aligned}$$

These functions take three words as input and compute one word. F is used for all step operations in round 1, G in round 2 and H in round 3.

The compression function works on 512-bit blocks of the message, which are represented by 16 32-bit words. So in each round, every word is included, but in different order, depending on the round (see table 4.2).

Round 1	W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7
	W_8	W_9	W_{10}	W_{11}	W_{12}	W_{13}	W_{14}	W_{15}
Round 2	W_0	W_4	W_8	W_{12}	W_1	W_5	W_9	W_{13}
	W_2	W_6	W_{10}	W_{14}	W_3	W_7	W_{11}	W_{15}
Round 3	W_0	W_8	W_4	W_{12}	W_2	W_{10}	W_6	W_{14}
	W_1	W_9	W_5	W_{13}	W_3	W_{11}	W_7	W_{15}

Table 4.2: Rounds and order of words in MD4 ([4], 2004, p. 49)

Figure 4.1 shows, what happens in one step. The four buffer words are serving as input, but only three words are the input for f_r (either function F, G or H) and the forth word gets updated. In case of figure 4.1, the value of word A is added to the ouput of f_r. After this, one message word (W_j) and U_r is added. U_r represents a constant, which depends on the round. It's `0` for round 1, `5A827999` for round 2 and `6ED9EBA1` for round 3. The whole value gets shifted in the end. The rotation constant v_s varies in every step. In the next step, another buffer value (in figure 4.1, it's D) gets updated. [31] (see [4], 2004, pp. 49,50)

It's important to note, that the step operation of MD4 is reversible, because only a right shift and some subtractions have to be done. But after all 48 steps (three rounds), the old buffer values are added to the output values of round three.So the compression function can't be inverted. Figure 4.2 shows the MD4 compression function. (see [4], 2004, pp. 49-51)

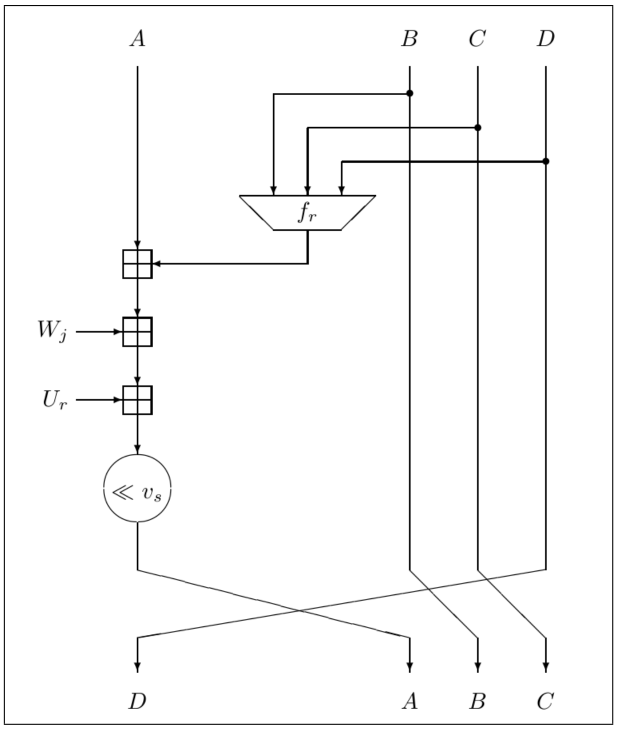
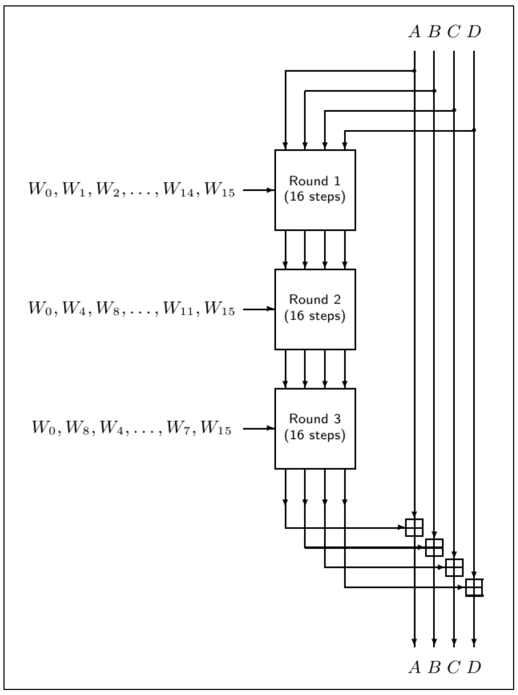


Figure 4.1: One step in MD4 ([4], 2004, p. 50)



$A\ B\ C\ D$
$W_0, W_1, W_2, \ldots, W_{14}, W_{15}$
Round 1 (16 steps)
$W_0, W_4, W_8, \ldots, W_{11}, W_{15}$
Round 2 (16 steps)
$W_0, W_8, W_4, \ldots, W_7, W_{15}$
Round 3 (16 steps)
$A\ B\ C\ D$

Figure 4.2: The MD4 compression function ([4], 2004, p. 51)

Step 5: Output

The hash value representing the message is the concatination of the last buffer values *A, B, C* and *D*. Four 32-bit words add up to excatly 128 bit.

4.1.2 Security

After introducing the algorithm, it was soon discovered, that the security level is much lower than expected. This was demonstrated by some attacks on reduced versions of the algorithm. For example, the first two rounds were successfully cryptoanalyzed and it was shown, that just 4 to 64 random selected messages are needed in order to find a collision of MD4. [7] [8] (see [4], 2004, p. 48)

So it can be said that MD4 was broken, even more than once. Hence, Rivest strengthened his algorithm and the result was MD5.

4.2 MD5

After some attacks on MD4, Rivest realized, that the security level of his algorithm isn't as strong as he thought. It's also said, that MD4 was implemented too fast in applications. So in 1991 to 1992, Rivest designed the *Message Digest 5* algorithm, a strenghtened version of MD4. His new algorithm is slightly slower than the predecessor, but it's better to give up a little in speed to be more secure. MD5 is also placed in public domain, like MD4. [32] [33] (see [5], 2003, p. 192)

4.2.1 Differences and improvements to MD4

The design of MD5 is very similar to MD4, but Rivest made some important changes, which make MD5 to the more secure algorithm: [32] (see [4], 2004, p. 69)

- A fourth round has been added, so the compression function now consists of 64 steps.

- The additive constant (U_s) now depends on the step (in MD4, U_r depends on the round), so there are 64 different values for U_s.
- The function G from round two was changed slightly to make it less symmetric.
- In each step, the result of the previous stop is added in the end. This procduces a faster avalanche effect.
- The order of message words (W_j) has changed in rounds two and three, which makes the patterns less alike.
- The per-step rotation constants (v_s) were changed, that different rounds never use the same value. Also, these constants were optimized to get a faster avalanche effect.

4.2.2 Algorithm description

MD5 generates a 128-bit output value, the same length like MD4. The algorithm gets any input message *M*, represented by b bits - m_0 to m_{b-1}. To compute the hash value *H(M)*, five steps are performed.

Step 1: Append padding bits
Step 2: Append length
Step 3: Initialize buffer

In the first three steps, the same things are performed as in MD4 (see 4.1.1). Also the four buffer words (*A, B, C, D*) which represent the inizialisation vector are the same.

Step 4: Process message in 16-word blocks

The MD5 algorithm works in four rounds to compute the output value. So with 16 steps in each round, 64 steps are done on each 512-bit (16-word) message block. Every round takes the four buffer words (*A, B, C, D*) as input and every step updates one of these four words. For this updating, one nonlinear function is used, depending on the round. So there are four functions defined: [32]

$$\begin{aligned} F(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\ G(X,Y,Z) &= (X \wedge Y) \vee (Y \wedge \neg Z) \\ H(X,Y,Z) &= X \oplus Y \oplus Z \\ I(X,Y,Z) &= Y \oplus (X \vee \neg Z) \end{aligned}$$

Like in MD4, every word of the 16-word message block is included in every round, but in different order, depending on the round (see table 4.3).

Round	W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7
1	W_8	W_9	W_{10}	W_{11}	W_{12}	W_{13}	W_{14}	W_{15}
Round	W_1	W_6	W_{11}	W_0	W_5	W_{10}	W_{15}	W_4
2	W_9	W_{14}	W_3	W_8	W_{13}	W_2	W_7	W_{12}
Round	W_5	W_8	W_{11}	W_{14}	W_1	W_4	W_7	W_{10}
3	W_{13}	W_0	W_3	W_6	W_9	W_{12}	W_{15}	W_2
Round	W_0	W_7	W_{14}	W_5	W_{12}	W_3	W_{10}	W_1
4	W_8	W_{15}	W_6	W_{13}	W_4	W_{11}	W_2	W_9

Table 4.3: Rounds and order of words in MD5 [32]

Figure 4.3 shows, how one step operates in MD5. The buffer words (*A, B, C, D*) serve as input. Three of these words are taken to compute one 32-bit value using one of the before defined nonlinear functions (f_r). So as shown in figure 4.3, buffer word A gets updated in the following way: [32] (see [4], 2004, p. 69)

$$A = (A + f_r(B, C, D) + W_j + U_s)^{<<v_s} + B$$

W_j represents one word of the message block, where j depends on the round and on the step (see table 4.3). The added constant U_s depends on the step. U_s is computed so that in step i, U_s is the integer part of $2^{32} \times abs(\sin i)$. The output gets shifted left v_s times, where v_s is also a constant which depends on the step. In the end, another buffer value gets added to the whole output (in case of figure 4.3, it's B) and then the step starts over again. (see [4], 2004, pp. 69,70) So it can be seen, that one step operation is revisible, as it's the case in MD4, just by doing some subtractions and a right shift. But the whole compression function (four rounds - 64 steps) can't be inverted, because the old buffer values (A, B, C, D) are added to the new ones before taking them as input for the next compression function. Figure 4.4 shows the MD5 compression function. [32] (see [4], 2004, p. 70)

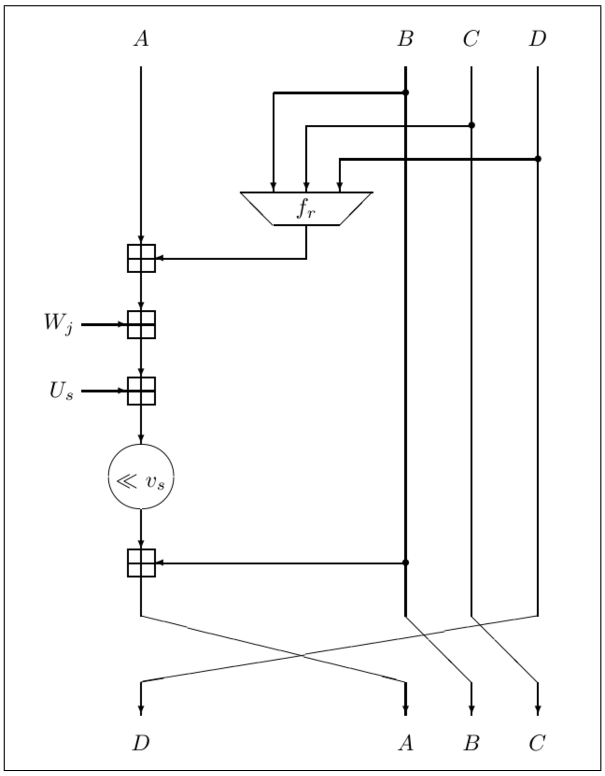


Figure 4.3: One step in MD5 ([4], 2004, p. 71)

Step 5: Output

The output is computed in the same way as in MD4, by just concatenating the final buffer values to one 128-bit stream.

4.2.3 Applications

MD5 is very popular. It's used for file integrity checks, with digital signature techniques and to store password hashes. Because of serious flaws it may not be very useful anymore to rely on MD5 hashes in secure applications.

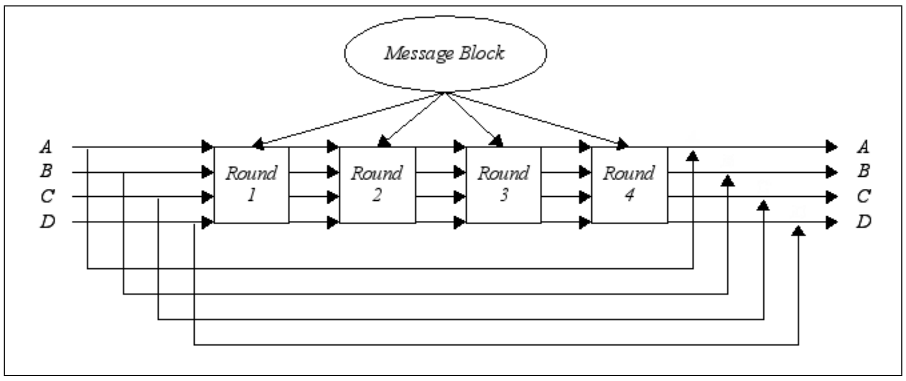


Figure 4.4: The MD5 compression function (see [1], 1996, p. 437)

4.2.4 Security

As said before, MD5 isn't free of attacks anymore. In 1996, a first attack on the compression function of MD5 was introduced to find collisions. After attacks in 2004 it can be said, that the compression function of MD5 isn't collision resistant, which makes the whole algorithm insecure. Furthermore, there are recognized possibilities to generate collisions for the whole algorithm. Also the small hash size (128 bits) isn't very applicable nowadays, because it makes MD5 vulnerable against a birthday attack. [33] (see [4], 2004, pp. 70-73)

MD5 is still very much in use nowadays, although weaknesses are known. But it's generally recommended to switch to another hash function, which at least produces a longer bit length for the output value to avoid possible birthday attacks. One alternative may be the SHA-1 algorithm.

4.3 SHA family

The term *Secure Hash Algorithm* describes a set of cryptographic hash functions. All of these algorithms were designed by the NSA and published by the NIST (an US government standards agency). These functions are also examples for algorithms, which where inspired by MD4. The most common member of the SHA family is called SHA-1. However, the first standard was called SHA (or SHA-0) and was published in 1993. It's design principles were never made public, but in 1995, the NSA released SHA-1. This was done because of flaws in SHA-0, which reduced it's security. Details were never been published, but further cryptoanalysis on SHA-0 have really shown weaknesses. So SHA-1 is the preferred algorithm. [34] [35] (see [4], 2004, p. 96)

Between 2002 and 2004, the NIST published some new variants of the SHA algorithm which are collectively called SHA-2. The standard describes four new functions, SHA-224, SHA-256, SHA-384 and SHA-512. The main difference to SHA-1 is, that these algorithms produce a longer output value (described by the number in the name of the standards). [34] (see [4], 2004, pp. 96,97)

Table 4.4 shows all SHA variants with their bitlength and the used message block size.

This section will first describe, how SHA-1 works. Also the little difference to SHA-0 is pointed out. After talking about the security of SHA-0 and SHA-1 implementations, SHA-2 variants are faced, their design and security.

Algorithm name	Block size (bit)	Ouput length (bit)
SHA-0	512	160
SHA-1	512	160
SHA-224	512	224
SHA-256	512	256
SHA-384	1024	384
SHA-512	1024	512

Table 4.4: SHA variants [34]

4.3.1 SHA-1 algorithm description

Both, SHA-0 and SHA-1 produce a 160-bit hash value as output. Here, only the operating mode of the SHA-1 algorithm will be described, but, of course, the difference to SHA-0 will also be shown.

As SHA-1 derives from MD4 (and thus also from MD5 in some case), the design is almost the same. First, the message is padded and the original length is appended, to get the message a multiple of 512 bits long. This happens in the same way like it was described in MD4 (see 4.1.1). The inizialisation vector of SHA-1 consists of five 32-bit words, *A, B, C, D* and *E*. These are five values, not four like in MD4 or MD5, because the output here has to be 160 bits long. The buffer is inizialised to the following values: [35] (see [1], 1996, p. 442) (see [4], 2004, p. 97)

word A `67 45 23 01`

word B `EF CD AB 89`

word C `98 BA DC FE`

word D `10 32 54 76`

word E `C3 D2 E1 F0`

The algorithm works on a 512-bit message block which is divided into 16 32-bit words (W_j). The compression function consists of 80 steps which can be divided into four rounds. Every step updates two of the five buffer values. Figure 4.5 shows one step operation. The shown step updates the words B and E in the following way: (see [4], 2004, pp. 97,98)

$$\begin{aligned} E &= E + f_r(B, C, D) + A^{<<5} + W_j + U_r \\ B &= B^{<<30} \end{aligned}$$

So the value of buffer B is just shifted to the left. To compute the new value of *E*, *B*, *C* and *D* are used as input for a nonlinear function, whose structure depends on the round. So four functions are defined: [35] (see [1], 1996, p. 443) (see [4], 2004, p. 97)

$$\begin{aligned} f_1(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\ f_2(X,Y,Z) &= X \oplus Y \oplus Z \\ f_3(X,Y,Z) &= (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \\ f_4(X,Y,Z) &= X \oplus Y \oplus Z \end{aligned}$$

These functions are the same as in MD4, but here, a fourth one was added which is just a copy of the second function. The output of these functions is a 32-bit word, which is added to the initial value of E (in case of figure 4.5). After this, A gets shifted five bit positions to the left and this result is also added to E. Then, a word from the message block (W_j) and a constant U_r, which depends on the round, are added. For U_r, four values are defined: [35] (see [1], 1996, p. 443) (see [4], 2004, pp. 97,98)

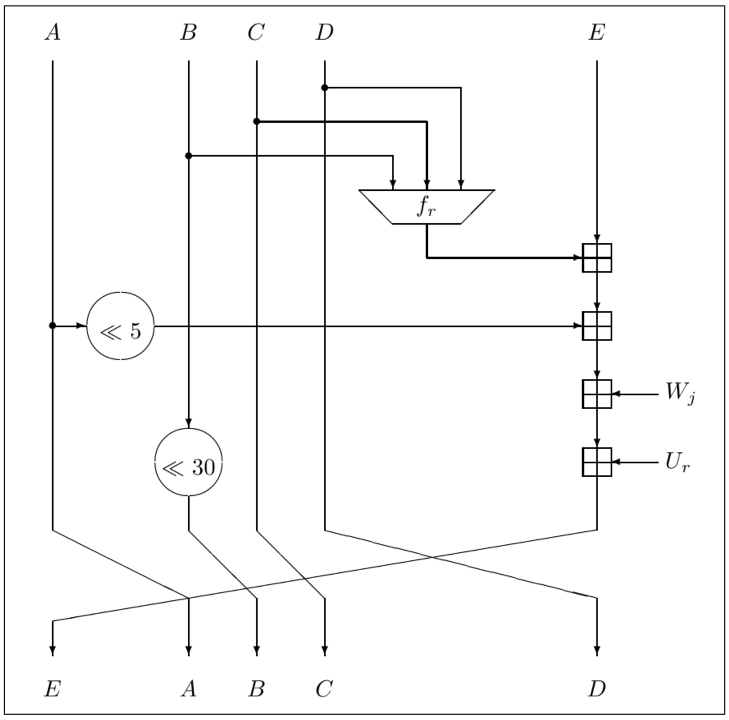


Figure 4.5: One step in SHA-1 ([4], 2004, p. 98)

U_1	`5A 82 79 99`
U_2	`6E D9 EB A1`
U_3	`8F 1B BC DC`
U_4	`CA 62 C1 D6`

The values for W_j are a bit special for SHA-1. The compression function consists of 80 steps, but because of the 512-bit message block as input, only 16 different message words are available. MD4 and MD5 just use the same 16 values three or four times in one compression function. But SHA-1 uses the 16 „real“ message words for the first 16 steps and computes new words for the other 64 steps afterwards. This procedure is called *message expansion* and the values of W_j for steps 17 to 80 are computed in the following way: [35] (see [1], 1996, p. 443) (see [4], 2004, p. 97)

$$W_j = (W_{j-3} \oplus W_{j-8} \oplus W_{j-14} \oplus W_{j-16})^{<<1}$$

In this function lies the only difference between SHA-0 and SHA-1, because in SHA-0, the shift by one bit position to the left isn't included. (see [4], 2004, p. 98)

The step operation of SHA-1 can easily be inverted, as it's the same with MD4 and MD5. But after all 80 steps in one compression function, the initial buffer values (A, B, C, D, E) are added to their final values, which makes the whole compression function irrevisible. (see Van Rompay, 2004, p. 99)

4.3.2 Security of SHA-0 and SHA-1

The NSA replaced SHA-0 with SHA-1 because of its insecurity. Although the real leaks of SHA-0 were never make public, first weaknesses were found by cryptoanalysts in 1998. By 2005, the complexity of finding collisions could be reduced to about 2^{39} operations. [34] (see [4], 2004, pp. 99,100)

SHA-1 is very similar to SHA-0, as mentioned before. But the little difference (rotation by one bit in the message expansion procedure) makes attacks more complex and attack strategies for SHA-0 can't be used to break SHA-1. But, of course, cryptoanalysts can rely on the knowledge of attacks on SHA-0 and MD5, so collision attacks on the full SHA-1 version were found e.g. in 2005, which require about 2^{69} operations. This number could be reduced by some researchers to 2^{63}, which leads to the thinking, that SHA-1 isn't secure enough anymore. But these attacks are all against the collision resistant property of hash functions, so it's no as dangerous as possible preimage attacks would be. So, compared to MD5, SHA-1 is secure, the only problem is the number of output bits. Nowadays, 2^{80} attempts for a birthday attack ($2^{n/2}$ trials are needed for this attack and $n = 160$ when using SHA-1) can't be seen as very secure anymore. [34] (see [1], 1996, pp. 444,445)

Schneier and Ferguson defined a „design security level“ of 128 bits, which means, that an output size (in bits) of at least 256 bits should be used to be considered secure. (see [2], 2004, pp. 84-89) This means that one should switch from SHA-1 to SHA-2 variants. Officially, the NIST meant that SHA-1 will be used until 2010 before they adopt some SHA-2 alogrithms instead (and SHA-2 variants will also be more sophisticated in 2010). [34]

4.3.3 SHA-2

Design and structure of all SHA-2 variants is very similar to SHA-1. They are relatively new and haven't been analyzed by the public very much. But they were developed by the NSA and one can assume, that they know what they're doing. If more security than SHA-1 is needed, a larger output value and hence a SHA-2 variant should be used. But it's also important to note, that SHA-2 variants are slower than SHA-1 (and MD5, which is also faster than SHA-1). (see [2], 2004, p. 89)

However, SHA-2 variants will form the future of hash algorithms. In fact that these variants are all more or less similar, only the design of the SHA-256 algorithm will be pointed out and after this, some distinctions between SHA-2 variants are described.

4.3.4 SHA-256 algorithm description

Here, a 256-bit value is computed out of any input. The compression function works with 512-bit message blocks, so the input message has to be padded in the beginning, which is done in the same way as in SHA-1, MD4 or MD5. The inizialisation vector consists of eight 32-bit words (*A, B, C, D, E, F, G, H*), to get an output value of 256 (8×32)-bit length. Table 4.5 shows the initial values. (see [4], 2004, pp. 101,102)

word A	`6A 09 E6 67`
word B	`BB 67 AE 85`
word C	`3C 6E F3 72`
word D	`A5 4F F5 3A`
word E	`51 0E 52 7F`
word F	`9B 05 68 8C`
word G	`1F 83 D9 AB`
word H	`5B E0 CD 19`

Table 4.5: SHA-256 inizialisation vector [9]

Internally, the message block in one compression function is divided into 16 32-bit words and the whole compression function consists of 64 steps. Different from SHA-1 is, that no clear distinction into rounds can be made here. One step in SHA-256 updates two of the eight buffer values, the others impact the outcome. Figure 4.6 shows one step in this algorithm. In the figure, words D and H are updated. This happens in the following way: (see [4], 2004, pp. 102,103)

$$\begin{aligned} D &= D + H + f_1(E,F,G) + \sum\nolimits_1(E) + W_j + U_s \\ H &= H + f_1(E,F,G) + \sum\nolimits_1(E) + f_2(A,B,C) + \sum\nolimits_2(A) + W_j + U_s \end{aligned}$$

It can be seen, that the calculations in every step of SHA-256 are very complex. There are four

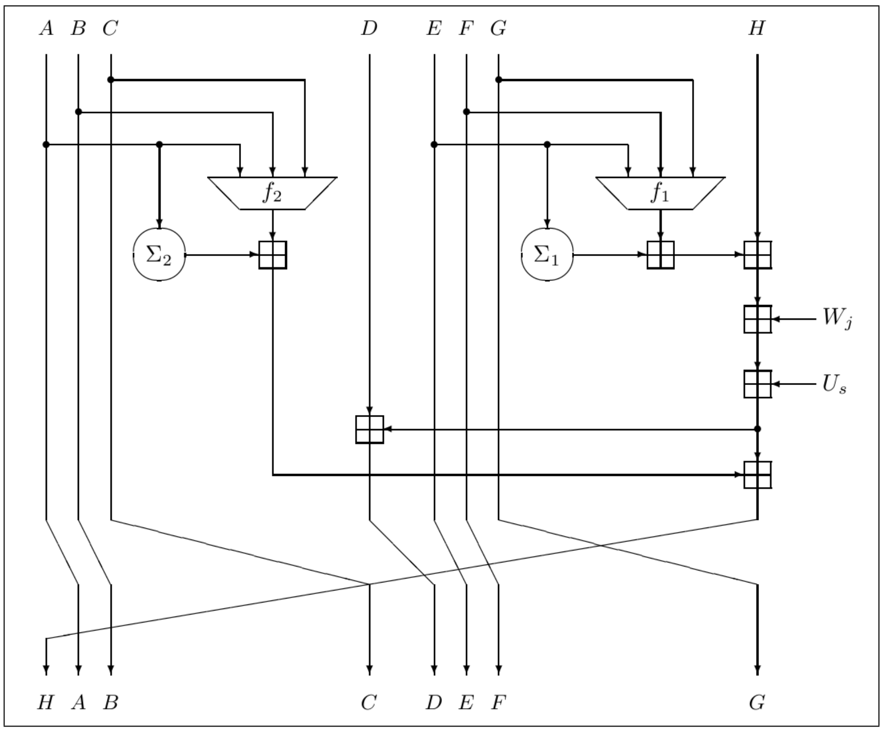


Figure 4.6: One step in SHA-256 ([4], 2004, p. 103)

functions defined, which are used in the step operation. f_1 and f_2 are boolean functions, similar to functions f_1 and f_3 in SHA-1 (see 4.3.1). These functions get three buffer values as input and compute one word out of them. $\sum_1$ and $\sum_2$ take one word and compute another word out of this input. They are defined as follows: (see [4], 2004, p. 102)

$$\begin{aligned} \sum\nolimits_1(Z) &= Z^{>>6} \oplus Z^{>>11} \oplus Z^{>>25} \\ \sum\nolimits_2(Z) &= Z^{>>2} \oplus Z^{>>13} \oplus Z^{>>22} \end{aligned}$$

Beside of these functions, the new values are influenced by W_j and U_s. U_s is just another additive constant which depends on the step. W_j is a 32-bit word from the message block. As it's in SHA-1, SHA-256 uses the 16 32-bit words for the first 16 steps and, to compute the other 48 values for W_j, message expansion is used. Here, message expansion is done in the following form (slightly different from SHA-1): (see [4], 2004, pp. 102,103)

$$W_j = \sigma_1(W_{j-2}) + W_{j-7} + \sigma_2(W_{j-15}) + W_{j-16}$$

where σ_1 and σ_2 are defined as follows: (see [4], 2004, p. 102)

$$\begin{aligned} \sigma_1(Z) &= Z^{>>17} \oplus Z^{>>19} \oplus Z^{\to 10} \\ \sigma_2(Z) &= Z^{>>7} \oplus Z^{>>18} \oplus Z^{\to 3} \end{aligned}$$

Like it's in SHA-1, one step in SHA-256 is revisible. But when all 64 steps of the compression function are done, the beginning buffer values (A to H) are added to the final ones. This makes the whole compression function irrevisible. (see [4], 2004, pp. 103,104)

4.3.5 Other SHA-2 variants

SHA-256 was described before, but the NIST published three more algorithms. All four variants are very similar, just the internal block size and the ouput size varies. SHA-224 works exactly in the same way as SHA-256, it just uses a different inizialisation vector and outputs only 224 of 256 bits as hash value. SHA-384 and SHA-512 are using 64-bit instead of 32-bit words and work on 1024-bit message blocks. But the structure of SHA-512 is again the same as in SHA-256, just the number of steps has been increased to 80. (see [4], 2004, p. 105)

Schneier and Ferguson mean, that the SHA-224 and SHA-384 standards are senseless. It's good enough (or even better) to use SHA-256 or SHA-512 instead, because internally these algorithms are used anyway and only the ouput values are truncated. (see [2], 2004, p. 89)

4.3.6 Security of SHA-2

Design principles for the SHA-2 variants haven't been made public. But it can be said, that even if the design is similar to SHA-1, important improvements have been made by the NSA. Also, all variants of the SHA-2 family use bitlength of 256 and above, which makes them more secure and not so vulnerable against birthday attacks. After some analysis it's also known, that these algorithms have very much security against known attack schemes on other hash algorithms. [34] (see [4], 2004, pp. 104-106)

But, however, these standards are new and the process of cryptoanalysis isn't in the stadium to tell whether these algorithms are really as secure as it seems.

4.4 RIPEMD

The *RACE Integrity Primitives Evaluation Message Digest* algorithm was designed in Europe by the RIPE Consortium and was first published in 1992. Its design is based on MD4 with the improvement of having two MD4 compression functions working in parallel. In 1996, Dobbertin found a collision attack on a reduced version of RIPEMD. Although only a reduced version was attacked, this meant an important security leak. So a strenghtened version of RIPEMD was designed in 1996, called RIPEMD-128. But RIPEMD-128 was just seen as a short time upgrade for RIPEMD users, because, also in 1996, a real successor was developed, called RIPEMD-160. [36] [11] [37] (see [4], 2004, p. 89)

This section will first describe the original RIPEMD algorithm and afterwards, the design of its successor, RIPEMD-160, will be faced.

4.4.1 RIPEMD algorithm description

The original algorithm produces a 128-bit hash value. So the buffer, which also serves as inizialisation vector, is divided into four 32-bit words (A, B, C, D). The compression function works on 512-bit message blocks and, as a special for this algorithm, consists of two trails that are executed in parallel. Every trail presents a own compression function, looking more or less like that in MD4. So 48 steps in three rounds are done two times in parallel to modify the buffer values. One step works similar as in MD4 (see 4.1.1). The whole compression function works as follows: (see [4], 2004, p. 90)

- The buffer values (A, B, C, D) serve as input for every trail.

- The two trails work in parallel and every trail updates the buffer values through three rounds.
- In the end, the output of the two trails and the initial buffer values are combined to compute new buffer values.

When defining the left trail output as A_L, B_L, C_L, D_L and the right trail output as A_R, B_R, C_R, D_R, then the resulting values for *A, B, C* and *D* (after one compression function) are computed in the following way: (see [4], 2004, p. 90)

$$\begin{aligned} A &= B + C_L + D_R \\ B &= C + D_L + A_R \\ C &= D + A_L + B_R \\ D &= A + B_L + C_R \end{aligned}$$

4.4.2 Security of RIPEMD

RIPEMD uses a slightly modified version of the compression function used in MD4. Although it was modified and some parameters (like constants) were changed, collisions for RIPEMD can still be found with a complexity similar to MD4 attacks. So the changes on MD4, which were made for RIPEMD, doesn't result in a much stronger algorithm. As mentioned before, Dobbertin found a collision on a reduced version of RIPEMD with only two rounds. But this attack could be extended to find collisions in the whole hash function. Also, the complexity of finding collisions for the reduced version is very low and lies at about 2^{31} computations. (see [4], 2004, pp. 91-93)

4.4.3 Successors

Because of the security leaks described before, the original RIPEMD algorithm had to be replaced. The weaknesses also influenced to design of new hash functions, called RIPEMD-128 and RIPEMD-160. As it was said before, RIPEMD-128 was seen as plug-in for RIPEMD and should only be used as temporary solution until RIPEMD-160 was released. This was also because 128 bit output values doesn't support any long time security (even not in 1996) whereas for RIPEMD-160 it had been expected that it should be secure for ten years or more. Nowadays, of course, even a 160-bit output value may offer no long term security anymore. Because of this, two extensions of RIPEMD-128 and RIPEMD-160 were introduced. These algorithms are called RIPEMD-256 and RIPEMD-320 respectively. They may be used in applications, where a hash value longer than 160 bits is needed. But it must be noted, that these extensions don't offer a higher security level. They use more or less the same structure as their forerunner and in fact they are only as secure as the 128-bit or 160-bit versions. On the other hand it has to be said, that the complexity of an ordinary birthday attack still depends on the number of bits. So against these attacks, RIPEMD-256 and RIPEMD-320 offer more security. [36] [37] (see [4], 2004, pp. 95,96)

4.4.4 RIPEMD-160 algorithm description

RIPEMD-160 produces a 160-bit hash value out of any input message. The buffer consists of five 32-bit words, *A, B, C, D* and *E*. These words serve as inizialisation vector and their first values are the same as in SHA-1 (see 4.3.1). The message first gets padded, as e.g. in MD4. The compression function words on 512-bit blocks of the input message. Figure 4.7 shows one step in RIPEMD-160, which is computed as follows: [11] [10] (see [4], 2004, p. 93)

$$\begin{aligned} A &= (A + f_r(B, C, D) + W_j + U_r)^{<<v_s} + E \\ C &= C^{<<10} \end{aligned}$$

As it can be seen in figure 4.7, every step updates two of the five buffer values. The value of one buffer word (in case of figure 4.7, it's *C*) is shifted ten times to the left. The new value of *A* is influenced by all other buffer words. First, the output of a boolean function is added to the value of A. Five functions were defined, one is used for every step in one round, but two different functions are used for the same round in the left and right trail.

These functions are: [11] (see [4], 2004, p. 94)

$$\begin{aligned}
f_1(X,Y,Z) &= X \oplus Y \oplus Z \\
f_2(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\
f_3(X,Y,Z) &= (X \vee \neg Y) \oplus Z \\
f_4(X,Y,Z) &= (X \wedge Y) \vee (Y \wedge \neg Z) \\
f_5(X,Y,Z) &= X \oplus (Y \vee \neg Z)
\end{aligned}$$

W_j is a word from the message block, consisting of 16 words. The order of the message words is

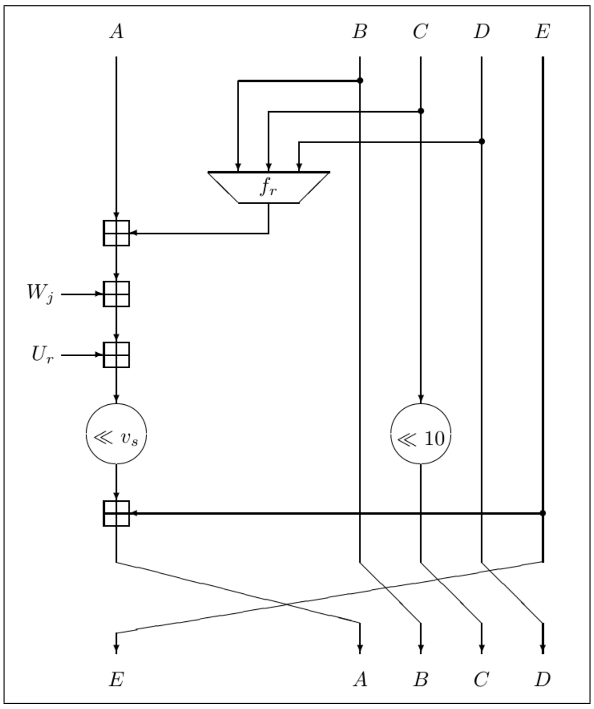


Figure 4.7: One step in RIPEMD-160 ([4], 2004, p. 95)

defined by the permutations p and π, which also guarantee, that the order differs in every round of every trail. The additive constant U_r depends on the round and also on the trail, so ten values for U_r were defined. Then, this value gets shifted v_s times, where v_s depends on the step. In the end, the input value of E gets added to this value to compute the final value of A (after one step). [11] (see [4], 2004, p. 94)

It's also important to note, that always two of these steps are done in parallel, one in every trail. So the total number of steps is 160, because five rounds with 16 steps each are done two times in one compression function. Like in all other described algorithms, the step operation is revisible. But at the end of one compression function, the output values of the two trails and the original input values are combined to compute the final output. This makes the whole compression function irrevisible. Figure 4.8 shows one RIPEMD-160 compression function. h_0 to h_4 denote the five buffer values A to E. It can also be seen, that every round uses another boolean function (f) and a different additive constant (K). And, as mentioned before, the input of the message words (X) depends on p and π. So the final values of A, B, C, D and E are computed as follows (A_L to

E_L are the output words of the left trail, A_R to E_R are the output words of the right trail): [11] [10] (see [4], 2004, p. 94)

$$\begin{aligned} A &= B + C_L + D_R \\ B &= C + D_L + E_R \\ C &= D + E_L + A_R \\ D &= E + A_L + B_R \\ E &= A + B_L + C_R \end{aligned}$$

So far, RIPEMD-160 is seen as secure. But it must be noted, that it's not as popular and hence not as well studied as other algorithms. At least the output value of 160 bits places a weakness on this algorithm, because this may make it insecure against birthday attacks in a short period of time. [36]

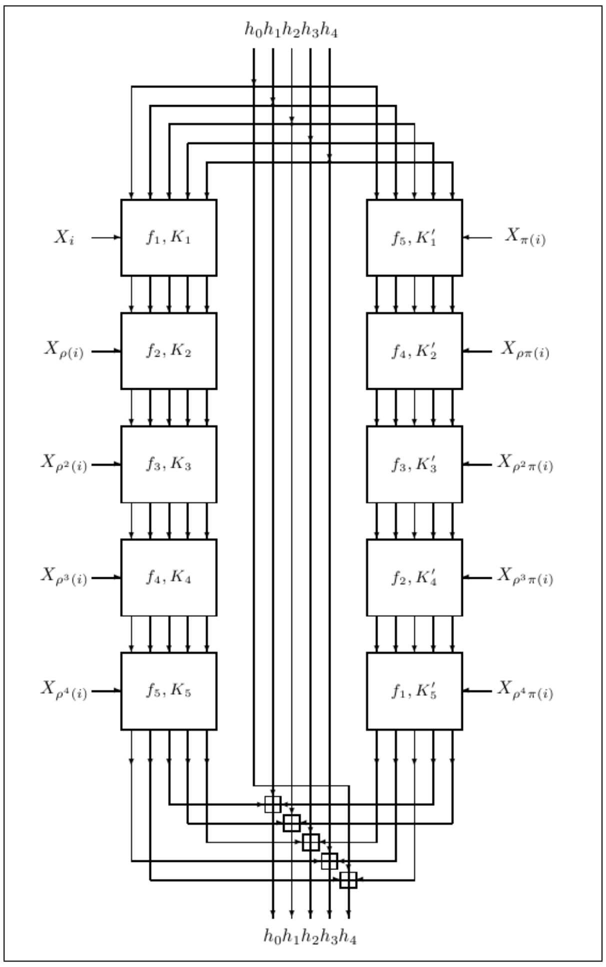


Figure 4.8: The RIPEMD-160 compression function [11]

4.5 Whirlpool

This algorithm was designed by Rijmen and Barreto. The first version (sometimes called *Whirlpool-0*) was released in 2000 and since then, two revisions have been made. Whirlpool has another design than all the algorithms described before in this chapter. It's compression function isn't based on the Merkle-Damgård construction (see section 3.1), but on the *Miyaguchi-Preneel scheme*. This means, that Whirlpool is a hash function which is based on a block cipher. [38] [39] [12]

4.5.1 Algorithm description

Whirlpool takes any input message less than 2^{256} bits in length and computes a 512-bit hash result. As mentioned before, the algorithm is based on a block cipher. This is a 512-bit cipher which uses a 512-bit key. Before starting the computations, the input value has to be padded to be a multiple of 512 (in bits). This happens in the same way as in other algorithms, with adding one „1" bit, „0" bits and a 256-bit representation of the original input length. The resulting message is divided into 512-bit blocks, m_0 to m_t. As inizialisation vector serves a 512-bit string, which is filled up with „0" bits. [39] [12]

The compression function follows the Miyaguchi-Preneel scheme and iterates over all 512-bit blocks of the padded message. Internally, the compression function works with a 8×8 matrix, so the message blocks and the inizialisation vector have to be converted into the matrix-format. This is done by function μ, which takes a bit stream and returns a 8×8 matrix, in which every entry represents one byte. Another important element of Whirlpool is the internal block cipher W, which is a modified version of the AES cipher. W encrypts a given message, using a given key. [39] [12]

$$W[key](message) = ciphertext$$

The whole algorithm works as follows: [12]

$$\begin{aligned} \eta_i &= \mu(m_i) \\ H_0 &= \mu(IV) \\ H_i &= W[H_{i-1}](\eta_i) \oplus H_{i-1} \oplus \eta_i \end{aligned}$$

As hash representation of the input message, the last value of H (H_t) is taken. [39] [12]

4.5.2 Security

Whirlpool got two revisions and is also adopted as an ISO/IEC standard [10]. It's of course not as widly used as other hash algorithms and haven't been as well studied, but it can be said that Whirlpool is secure, because no weaknesses have been found so far. Also using a hash result with 512-bit length makes the algorithm pretty resistant against brute force attacks (like birthday attacks).

4.6 Tiger

Tiger was first developed in 1996 by Anderson and Biham. The design motivation was to get an algorithm which works efficient on 64-bit machines and also on existing 32-bit machines. Because other algorithm were developed for 32-bit architectures, which made them slow for 64-bit users. Also, Tiger should replace current, but already attacked hash functions (like MD4). [40] [13]

4.6.1 Algorithm description

This algorithm produces a 192-bit hash value out of any input message. In Tiger, all words consist of 64 bits, hence the inizialisation vector is divided into three words with the following values: [13]

word A `01 23 45 67 89 AB CD EF`

word B `FE DC BA 98 76 54 32 10`

word C `F0 96 A5 B4 C3 B2 E1 87`

As it's in all other algorithms, the input message gets padded first and the original length is added. After this, the whole input is a multiple of 512 (in bits). The compression function works on 512-bit message blocks, which are divided into eight 64-bit words (m_0 to m_7). One flow of the compression function works in three passes, where every pass updates all buffer values (A, B, C) several times. Each pass is divided into eight rounds and every round gets all three buffer values and one (of eight) message words (m_i) as input. In every round, new values are computed for the buffer words, using XOR, addition, subtraction and multiplication operations. After the first and second pass, a *key schedule* operation is done, where all message words are updated two times. In the end, the original buffer values are added to the final ones to make the whole compression function irrevisible. [13]

4.6.2 Security

In the time, when Tiger was designed, it was considered as secure. Even the complexity of birthday attacks ($2^{192/2} = 2^{96}$) was much too large for efficient attacks. But, of course, times have changed and there were found collision attacks for the original version of Tiger with a complexity of about 2^{44} operations. [40] [13]

4.7 Conclusion

In this chapter, an overview of popular cryptographic hash functions was given. All algorithms, except Whirlpool, follow the Merkle-Damgård construction and were more or less derived from MD4. So it can be said, that MD4 influenced the design of some very widly used algorithms. It's interesting to note, that there's a lot of interaction between cryptography and cryptoanalysis, when designing hash functions. Because nearly every author took known weaknesses of other algorithms into account for the design of new ones.

There can't be said, which hash function one should choose. MD4 won't be the first choice, but e.g. MD5 and SHA variants are widly used. Between these two, SHA may be the better one, at least because of the longer output value. But also RIPEMD offers a 160-bit result, like SHA-1, so RIPEMD-160 may also be a candidate for an application.

In fact it must be clear that, mainly in cryptography, no real prognosis can be done. It could always happen that more dangerous weaknesses are found in a popular algorithm, which causes that applications switch to another, or new, hash function which isn't focussed in this chapter.

Chapter 5

Secure algorithms

As one can see on the number of different flavours of cryptographic hash functions, it's obvious that there doesn't exist a strict rule for the design of a secure algorithm. So this chapter will look at the design principles of algorithms described in chapter 4 (and maybe some others) and will try to select some criteria for an algorithm to be secure.

5.1 Main properties

The two main requirements on cryptographic hash functions were described in 3.2.2. A designer has to decide, whether an algorithm should be both, collision resistant and one-way, or not. This leads to the definiton of a *ideal hash function.*

Ideal hash function *The ideal hash function is a random mapping from all possible input values to the set of possible output values.* ([2], 2004, p. 85)

It's clear that a hash function can never met these demands. But a good algorithm acts very much like this ideal hash function. It's very important to proofe, whether the required prerequisits are fulfilled. For example some of the algorithms in chapter 4 are still one-way functions, but attacks have shown, that collisions can be found within a certain number of operations.

For a secure algorithm it's essential to define excatly, what parts accomplish these requirements. Because of the importance of these two properties (and the influence of weaknesses here on the whole function), also a lot of cryptoanalytic work has to be done, before an algorithm can be published as secure.

5.2 One-wayness vs. bijectivity

The meaning of one-wayness was defined in 3.2.2. Bijective means, that one input element is assigned to exactly one output element, which makes a function reversible. So a function f is bijective, if there's a one-to-one correspondence between two sets X and Y, such that $f(x) = y$. [41] Of course, this stands in disagreement with the one-wayness, but here, just rounds of the compression function are focussed. This means, that it has to be decided, at which point the bijectivity gets lost in order to get a one-way function.

Bijectivity can be good for a part of an internal round function. But this also means a one-way mapping of input values to output values, what eases meet-in-the-middle attacks, because an attacker can go backwards value for value. In most cases, the two techniques will be combined. So at some point in the compression function, a feed-forward operation is done. This operation may add the input buffer values to their final one, to make the whole compression function irrevisible. (see [5], 2003, p. 214) For example MD4 and its successors use a feed-forward at the end of their compression functions.

When designing an alogrithm, it must clearly be defined, if the algorithm is bijective and at which point (e.g. after one round or after one run of the compression function) and because of what operations it becomes a one-way function. It's also important to analyse, whether the one-way property is really given at the end of the algorithm.

5.3 Inizialisation vector and message padding

It's recommended to use a single, fixed IV and a padding rule, which e.g. adds the length of the original message to the input value. Because of this length adding, it may be useful for some algorithms to define a maximal length for an input message. (see [5], 2003, p. 215) The exact value for IV isn't critical, which can be seen when looking at the algorithms of chapter 4. All IV's are almost similar, except the on used in Whirlpool, which is set to 512 zero bits. This shows, that it's just important to use a fixed IV, because it complicates fixed point attacks.

Using a padding rule is also called *MD-strenthening*, which was explained in 3.5.2.

5.4 Procssing message words

MD4 and variants follow the technique of using every word of the message block several times. Generally, this can be seen as a good design principle. The same words are used on different positions and serve as input for different functions, which means, that one word of the message block impacts the output in several ways. (see [5], 2003, p. 216) Using this technique makes it harder for an attacker to change a message words at a random position in the compression function, because if the algorithm was good designed, the original word may have influenced the buffer values earlier in the whole hash function. SHA and its variants use a different technique to ensure that all message words are used several times. Here, words are computed out of others, to extend the number of useable words (*message expansion* - see section 4.3). It's a bit another way than MD4 goes, but in the end, this also guarantees, that every message words influences the buffer values more than once.

A designer should make sure, that a kind of this techniques will be used, because this means a secure way. The more often a word influences the whole output, the harder it is for an attacker to insert his own words inside the algorithm. This also makes a correcting block attack more difficult.

5.5 Output value

When designing hash functions, the number of bits of the output value is an important factor. It influences many pointsw in the whole algorithm and should be chosen wise, because of different criteria.

5.5.1 Attacks

One should look at the security and also on today's standards. This is maybe the most important point, because if someone releases an algorithm with a short number of bits (e.g. less than 160 bits), then the whole algorithm will be insecure, independent of all secure designed elements inside the algorithm. With the computer power, which can be used today, brute force attacks with a complexity of 2^{64} to 2^{80} (which means output length of 128 to 160 bits) are in a possible field. So output length with 160 bits or more should be used, where nowadays, 160-bit outputs are seen as the lowest limit.

5.5.2 Performance

It also has to be said, that the number of bits shouldn't be increased senseless. Here, the factor performance must be respected. Increasing the output length generally means, that the whole algorithm speed decreases (because the number of bits have to be generated in some way). For example MD5 (128-bit output) is much faster than SHA-1 (160-bit output). The SHA-2 variants (e.g. SHA-512) are very slow algorithms. So there have to be found a good balance between performance and the number of bits. Of course, hash functions have very complex operations which take time, but if someone wants to hash a big amount of data, then he'll give up some bits of the output for a better performance.

5.5.3 Function design

When defining an output size, the result will directly impact the size of the buffer values, the internal functions, round functions and the compression function. Tiger, SHA-384 and SHA-512, for example, use a 64-bit word length. This influences the inizialisation vector values (at least in the number of bits) and the process of splitting the whole input value in message words. Also, such algorithms perform better on 64-bit platforms (e.g. Tiger was specially designed to work fast on 64-bit machines). This fact has something to do with *big-* and *little-endian* architechtures and is more a topic of the previous point.

The construction of the mathematical (boolean) functions is also important for the security of a hash algorithm. For example, there shouldn't be too much linearity between functions used in different rounds.

5.6 Other criterias

It's clear, that there can't be given an exhaustive enumeration of all criterias, which have to be considered when designing a secure hash function. The five points mentioned before in this chapter are real big and important points, which mustn't be forgotten in the design process. There will be other ones, but as said at the beginning of this chapter, the variety and importance of criterias is different for different algorithms, so no distinct list can be given, covering all supposable algorithms.

In this section, this paper will face one last but maybe very important point in the design process, the cryptoanalysis.

In section 3.5, some attack mechanisms on hash functions where shown. So when designing a hash algorithm, these attacks have to be taken into account. Also, some evaluations have to be done, for statistical aims. The following things should be checked: (see [5], 2003, p. 216)

output value Are there any irregularities in the distribution or can the process of building the output be reconstructed?

input value How much does a little difference in the input affect the output (alavanche effect)?

linearities Do some linearities exist between input and output, between functions, outputs of internal rounds, in the message word processing, etc.?

In the end it should be said, that after designing an algorithm, all of its elements should be cryptoanalyzed before publishing the new hash function. Only a totally evaluated and cryptoanalyzed algorithm can be rated as really secure, when no weaknesses were found with neither of the methods.

5.7 Today's point of view

Latterly collisions have been found for the widly used MD5 algorithm and also SHA-1, which counted as more secure than MD5, experienced some attacks. This leads to the question, which algorithm should be used now and whether these attacks are very critical or not. It's not easy to answer such a question, for the beginning, table 5.1 will show weaknesses which were found in the algorithms described in chapter 4.

Algorithm	Known attacks
MD4	Collisions can be found using just 4 to 64 random selected messages.
MD5	Collisions were found for the compression function, which makes it also possible to find collisions for the whole algorithm.
SHA-0	Collision can be found with the complexity of about 2^{39} operations.
SHA-1	Theoretical collision can be found in about 2^{63} attempts. More dangerous is the output size of 160 bits, which will make a birthday attack possible in the near future.
SHA-2	No weaknesses were found. But these algorithms are very new, so the process of cryptoanalysis isn't so advanced to tell whether these algorithms are really secure. The output length is secure against birthday attacks, using 256 bits and more.
RIPEMD	Collisions for a two-round version were found, but these could be extended to the whole algorithm. Also the complexity for the reduced version is very low with about 2^{31} computations.
Whirlpool	No weaknesses were found. But this algorithm is also not very well studied. The output size is secure (512 bits).
Tiger	Collisions were found with a complexity of about 2^{44} attempts.

Table 5.1: Known attacks in popular algorithms

It can be seen, that the choices aren't very widly spread when searching a secure algorithm. But, of course, also attacked algorithms are still in use. For example MD5 is used by a lot of different applications, although real weaknesses were found. A better alternative would be SHA-1, which is another widly used algorithm. It has to be said, that the weaknesses found for SHA-1 are not as precarious as the one for MD5, so this algorithm may be good enough for the next few years. It's also mentioned, that it's output size is too short, but also this is only for theoretical brute force attacks, because with the computer power used nowadays, a 160-bit hash value is still secure enough. But to be on the safe side, one should come away from these two generally used algorithms MD5 and SHA-1.

Also the security community isn't sure, which way to follow. There is one part, who thinks, that everyone should migrate to SHA-256 as soon as possible. This is also because of the reason, that attacks will always get better. So it'll be better to migrate to a new hash standard, before there's a big panic because of broken algorithms. It's also said, that SHA-256 is at least more secure because of its 256-bit output value, which makes the algorithm pretty resistant against brute force attacks. But these people also represent the opinion, that further research should be done to find more stronger algorithms. [28]

Other parties think, that people can stay with their current algorithms (like SHA-1 or MD5), because the shown attacks are not primarily effecting real world applications. Also the opinion is, it's not totally evaluated, which algorithm will be totally secure for the future (e.g. from the SHA-2 variants). [28]

(These two opinions derive from Bruce Schneier's and Paul Hoffman's points of view, who wrote this into their RFC4270 *Attacks on Cryptographic Hashes in Internet Protocols* in November 2005.)

Chapter 6

Summary and prospect

This paper faced a very important sort of cryptographic algorithms, the hash functions. These versatile cryptographic primitives have many application areas. When the security plays a role, hash functions are used very often, of course also together with other cryptographic practices, like block ciphers. For example when working with internet protocols, hash functions are used to ensure security e.g. through privacy, when passwords or other data which is worth protecting is stored as hash representation and not as plain text.

Like it's with many other scientific fields, there have been endless designs and developments of different hash functions, which could all be used nowadays. But, of course, it's a fact that only some of all the released hash algorithms have been analyzed accurately enough that one can offer them for usage. So this paper pointed out most popular algorithms and attacks against them, to tell whether they are secure or not.

Two research hypotheses were faced in this thesis. The first question asked after the design of cryptographic hash functions. Most of the described functions in chapter 4 follow the iterated scheme which was designed by Merkle and Damgård (see section 3.1). But there exist also other constructions for the compression function of hash algorithms, like the Miyaguchi-Preneel scheme. This is used for hash functions which are based on block ciphers (like Whirlpool) but this sort of construction wasn't specified in this paper.

To give an understanding of the operation mode of hash functions, popular algorithms were accurately specified in chapter 4. The following hash functions were pointed out (particular sections are written in parentheses):

- Message Digest 4 - 128-bit output (4.1)
- Message Digest 5 - 128-bit output (4.2)
- Secure Hash Algorithms - 160 to 512-bit output (4.3)
- RACE Integrity Primitives Evaluation Message Digest - 160-bit output (4.4)
- Whirlpool - 512-bit output (4.5)
- Tiger - 192-bit output (4.6)

All algorithms from chapter 4, except Whirlpool, are influenced by the MD4 hash function. So this algorithm served as predecessor for two of the most popular and widly used hash functions, MD5 and SHA-1. Algorithms, which follow the MD4 hash function, always work in five consecutive steps.

Message padding A „1“ bit and some „0“ bits are added to the message.

Append length The bit representation of the input message length is added to the end of the message. After these two steps, the message is a multiple of 512 bits long.

Buffer inizialisation To compute a hash value of e.g. 128 bits, some 32-bit buffer values are needed (in case of 128 bits, four buffer words are initialized).

Iteration of the compression function Here, the function works through 512-bit blocks of the input message. Most compression functions are splitted into some rounds, in which every buffer value is updated several times.

Arrange the output The output is the concatenation of the last buffer word values.

The answer to the second question wanted to point out, what criteria are important for a secure algorithm. Some of the most common criteria were given in chapter 5. Furthermore the security of actual used (and in chapter 4 described) algorithms was summed up to give an overview, what popular hash functions could be called the most secure ones. The last chapter also gave a forecast in using algorithms from cryptographers points of view.

A very important field for good hash functions is the cryptoanalysis. It's, among others, used to proove the security of an algorithm, e.g. to verify whether there exist no other attacks which are simpler than standard brute force attacks. This evaluation process takes a long time and every algorithm should be evaluated in detail before it's released to the public. Facing the algorithms from chapter 4, some attacks were described. Also section 3.5 showed some generic attacks, which are independent of real implementations. Furthermore it would have been interesting to go more into the detail of the cryptoanalysis and the design of cryptoanalytic methods for hash functions. But this topic is very complex and would go beyond the scope of this thesis. But Bart Preneel [5] and Bart Van Rompay [4] did with their works a good job on the field of cryptoanalysis of hash function and block ciphers.

Bibliography

[1] B. Schneier. *Applied Cryptography Protocols, Algorithms and Source Code in C.* John Wiley & Sons, 1996.

[2] B. Schneier and N. Ferguson. *Practical Cryptography.* John Wiley & Sons, 2003.

[3] A. Menezes. *Handbook of Applied Cryptography.* CRC, 1996.

[4] B. Van Rompay. *Analysis and Design of Cryptographic Hash Functions, MAC Algorithms and Block Ciphers.* PhD thesis, Katholieke Universiteit Leuven, 2004. (`http://www.cosic.esat.kuleuven.be/publications/thesis-16.pdf`).

[5] B. Preneel. *Analysis and Design of Cryptographic Hash Functions.* PhD thesis, Katholieke Universiteit Leuven, 2003. (`http://www.esat.kuleuven.ac.be/~preneel/phd_preneel_feb1993.pdf`).

[6] B. Preneel. The state of cryptographic hash functions. *Lectures on Data Security*, pages 158–182, 1999. (`http://www.cosic.esat.kuleuven.ac.be/publications/article-374.pdf`).

[7] B. Den Boer and A. Bosselaers. *An Attack on the Last Two Rounds of MD4*, 1991. (`http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9100.pdf`).

[8] X. Wang et al. *Cryptanalysis of the Hash Functions MD4 and RIPEMD.* (`http://www.infosec.sdu.edu.cn/paper/md4-ripemd-attck.pdf`).

[9] NIST/ITL. *FIPS PUB 180-2 - Secure Hash Signature Standard (SHS)*, 2002. (`http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangeno%tice.pdf`).

[10] ISO/IEC. *Dedicated Hash Functions*, 2003. (`http://www.ncits.org/ref-docs/FDIS_10118-3.pdf`).

[11] B. Preneel, H. Dobbertin, and A. Bosselaers. The cryptographic hash function ripemd-160. *CryptoBytes*, pages 9–14, 1997. (`http://www.cosic.esat.kuleuven.ac.be/publications/article-317.pdf`).

[12] P.S.L.M Barreto and V. Rijmen. *The WHIRLPOOL Hashing Function*, 2003. (`http://planeta.terra.com.br/informatica/paulobarreto/whirlpool.zip`).

[13] R. Anderson and E. Biham. *Tiger: A Fast New Hash Function.* (`http://www.cl.cam.ac.uk/~rja14/Papers/tiger.pdf`).

[14] Hash function. `http://www.absoluteastronomy.com/ref/hash_function`. 14.4.2006.

[15] Hashing. `http://en.wikipedia.org/wiki/Hashing`. 14.4.2006.

[16] Hash table. `http://en.wikipedia.org/wiki/Hash_table`. 14.4.2006.

[17] Redundancy check. `http://en.wikipedia.org/wiki/Redundancy_check`. 14.4.2006.

[18] An illustrated guide to cryptographic hashes. http://unixwiz.net/techtips/iguide-crypto-hashes.html. 18.4.2006.

[19] Cryptographic hashes. http://www.vpnc.org/hash.html. 18.4.2006.

[20] Cryptographic hash function. http://en.wikipedia.org/wiki/Cryptographic_hash_function. 19.4.2006.

[21] Merkle-damgård hash function. http://en.wikipedia.org/wiki/Merkle-Damg%C3%A5rd_construction. 21.4.2006.

[22] Avalanche effect. http://en.wikipedia.org/wiki/Avalanche_effect. 18.4.2006.

[23] Cyclic redundancy check. http://en.wikipedia.org/wiki/Cyclic_redundancy_check. 18.4.2006.

[24] Hmac. http://en.wikipedia.org/wiki/HMAC. 18.4.2006.

[25] Hmac algorithm in detail. http://www.cryptostuff.com/crypto/index.php?title=hmac. 18.4.2006.

[26] Salted hash. http://de.wikipedia.org/wiki/Salted_Hash. 19.4.2006.

[27] What is a digital signature? http://www.youdzone.com/signature.html. 19.4.2006.

[28] Rfc4270 - attacks on cryptographic hashes in internet protocols. http://www.ietf.org/rfc/rfc4270.txt. 21.4.2006.

[29] Birthday paradox. http://en.wikipedia.org/wiki/Birthday_paradox. 23.4.2006.

[30] The hashing function lounge. http://paginas.terra.com.br/informatica/paulobarreto/hflounge.html. 26.4.2006.

[31] Rfc1320 - the md4 message-digest algorithm. http://www.ietf.org/rfc/rfc1320.txt. 1.5.2006.

[32] Rfc1321 - the md5 message-digest algorithm. http://www.ietf.org/rfc/rfc1321.txt. 10.5.2006.

[33] Md5. http://en.wikipedia.org/wiki/MD5. 10.5.2006.

[34] Sha hash functions. http://en.wikipedia.org/wiki/SHA. 14.5.2006.

[35] Rfc3174 - us secure hash algorithm 1 (sha1). http://www.ietf.org/rfc/rfc3174.txt. 14.5.2006.

[36] Ripemd. http://en.wikipedia.org/wiki/RIPEMD. 16.5.2006.

[37] The hash function ripemd-160. http://homes.esat.kuleuven.be/~bosselae/ripemd160.html. 16.5.2006.

[38] Whirlpool. http://en.wikipedia.org/wiki/WHIRLPOOL. 20.5.2006.

[39] The whirlpool hash function. http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html%. 20.5.2006.

[40] Tiger (hash). http://en.wikipedia.org/wiki/Tiger_%28hash%29. 20.5.2006.

[41] Bijection. http://en.wikipedia.org/wiki/Bijective. 25.5.2006.