Master's Thesis

at

Graz University of Technology

submitted by

Milan Milinković

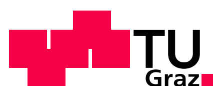
Cycle Finding Techniques for Hash Functions

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology
A-8010 Graz, Austria

February 28, 2011

© Copyright 2011 by Milan Milinković

Assessor: Univ.-Prof. Dr. Ir. Vincent Rijmen
Advisor: Dipl.-Ing. Dr.techn. Florian Mendel



Abstract

Finding collision in a hash function is not a trivial task. Even if a serious leak is found in the hash function, leading to the reduction of iterations before the detection of the collision, enormous number of values need to be computed. Some of the major problems are storing, indexing and searching such amount of values. Introducing cycle finding techniques for finding the collision can be the solution for this problem. They reduce needs for the storage size. The penalty for this approach is an additional evaluation of the function and an increment of the time complexity. These techniques use a property of the function defined over the finite set. If the output value of the hash function deterministic evaluates the input value of the next one, the trial must start cycling eventually. Various characteristics and comparisons were investigated and proved by practical experiments and performed on 1 bit version of RadioGatún hash function designed by Guido Bertoni et al. [6]. All theoretical and practical observations are mostly focused on the distinguished point as well as Nivasch, Floyd and Brent methods.

Statutory Declaration

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

______________	______________	______________
Place	Date	Signature

Contents

List of symbols **iii**

1 Introduction **1**

2 Cryptographic Hash Function **3**

2.1 Introduction 3
2.2 Classification of Hash Functions 4
2.2.1 Manipulation Detection Code (MDC) 6
2.2.2 Message Authentication Code (MAC) 8
2.3 Iterated Hash Functions 9
2.3.1 Weakness in Iterated Hash Functions 10
2.4 Types of Hash Functions 12
2.4.1 Hash Functions Based on Block Cipher 12
2.4.2 A Custom Design Hash Function 15
2.4.3 Hash Functions Based on Algebraic Structures 15
2.5 Attacks on Hash Functions 17
2.5.1 Attacks Independent of the Algorithm 18
2.5.2 Attacks Dependent on the Chaining 21
2.5.3 Attacks Dependent on the Underlying Block Cipher 22
2.6 Summary 23

3 RadioGatún **25**

3.1 Introduction 25
3.2 Basic Description 25
3.3 Structure of RadioGatún 26
3.4 Performance 28
3.5 Security of RadioGatún 29
3.6 Summary 29

4 Cycle finding algorithms **31**

4.1 Introduction 31

4.2 Objectives . . . 32
4.3 Characteristics of Random Functions in Cycle Finding Algorithms . 32
4.3.1 Directed Parameters . . . 34
4.3.2 Cumulative Parameters . . . 34
4.3.3 Extremal Parameters . . . 35
4.4 Pollards Rho Integer Factorization . . . 35
4.5 Cycle Detection Using Meet In the Middle Attack . . . 36
4.6 Cycle Detection Methods . . . 39
4.6.1 Floyd's cycle finding algorithm . . . 39
4.6.2 Brent's cycle finding algorithm . . . 41
4.6.3 Sedgewick, et al.'s algorithm . . . 44
4.6.4 Gosper's algorithm . . . 44
4.6.5 Time Memory Trade-Off Cycle Algorithms . . . 44
4.7 Summary . . . 51

5 Collisions in Cycle Algorithms 53
5.1 Introduction . . . 53
5.2 Objectives . . . 54
5.3 Finding Collisions Using Memoryless Algorithms . . . 55
5.3.1 Collision in Brent's algorithm . . . 57
5.3.2 Finding Collision using Nivasch's Stack-Based algorithm . . 58
5.3.3 Finding Collision Using Distinguished Points . . . 60
5.4 Summary . . . 64

6 Analysis 67
6.1 Introduction . . . 67
6.2 General Property of Random Mapping in Cycle Algorithms . . . 68
6.2.1 Performance of Different Input Size . . . 68
6.2.2 Applying Meet in the Middle Method on Cycle Finding Algorithm . . . 69
6.2.3 Collisions in Cycle Algorithms . . . 70
6.3 Analysis of Cycle Finding Algorithms . . . 72
6.3.1 Distinguished Point Parameters . . . 72
6.3.2 Efficiency of Using Stack in Nivasch's algorithm . . . 78
6.3.3 Performance of Cycle Methods on Different Platforms . . . 81
6.4 Summary . . . 83

7 Conclusion 85

References 89

List of symbols

c	the internal state of a function
m	a number of processors used
S	a table for storing data
$S[r]$	the location at position r in the table S
R	a reduction function
F	a step function
h	a compression function
g	a map function
gcd	the greatest common divisor
l	an average length of trial without distinguished points
Θ	the probability that the value has a distinguished property
f_s	a function whose an evaluation depends on some factor s
SP	a start point of the chain
EP	an end point of the chain
λ	the length of the tail
μ	the length of the cycle
ρ	the length from the random start point to the first repetition
$\langle x_i \rangle$	a sequence corresponds to the values x_0, x_1, x_2...
log	the logarithm of a number to a base 2
lg	the logarithm of a number to a base 10
ln	the logarithm of a number to a balse e
$\lceil x \rceil$	the smallest integer not less than x
$\lfloor x \rfloor$	the largest integer not greater than x

Chapter 1

Introduction

Hash function compresses an input string of arbitrary size (usually very large) to the output of fixed size which can be used in different applications, for example, to ensure the authenticity and data integrity of the message during the transmission over an insecure channel. To be used in cryptographic applications, a function needs to meet certain requirements. A hash function, as any other function, provides an output value which corresponds to the input value. There are, of course, many inputs that correspond to a single output but this is not easy to be determined. For a hash function to be secured, it is necessary for it to be collision resistant and one-way. If it is hard to find two different values that have the same hash value than this function is considered collision resistant. In one-way function, it is easy to generate a hash value from some arbitrary string, but it is hard to generate any string from the given hash value.

Not getting into the structure, finding collision is usually related to the birthday paradox. That is, for a possible n outputs an expectation that collision will be found is after producing about $\sqrt{n}$ distinct inputs. In its rough form, this means that after applying a hash function, the output value will be stored in a table if it is not already there. This kind of an attack is very hard to be produced in practice. The biggest problem is the enormous need for the storage. Another problem is that the time requirement for searching a certain element in the table is unaccepted due to the large number of elements.

During the last couple of decades a number of algorithms, which don't need a large number of storage in order find the collision, was produced. In the same way, the time looking into the table for the certain element is drastically reduced. One class of algorithms which deal with this problem are cycle finding algorithms. They exploit the fact that, if the function h is defined over some finite set then the sequence $x_0, x_1 = h(x_0), x_2 = h(x_1), \ldots$ for a given initial value x_0 must eventually start cycling. Pollard rho integer factorization uses this structure of pseudo-random walk in the group to find a factor of some integer. The same idea was used in

cycle finding algorithms but they do not exploit the group structure in which a random walk of function h is defined. In fact, methods can be applied to any set where iterated function h makes a random walk. Examples are Floyd's [27] and later, its improvement Brent's algorithm [10]. They used fixed and small amount of memory with a penalty of additional evaluation of function h. A slightly different approach of the same problem was described by Hellman [30] and two years latter improved by Rivest [20]. They introduced the distinguished point method that drastically reduced the number of lookups in the table. Since then, this method has been studied intensively and has experienced several variants. Introducing the parallelization in some of these methods can be another field where these methods can be accelerated and surely given a special treatment.

Detecting a cycle is just one part of the problem. The second part implies that, after determining that the element is in the cycle, collision must be found. There is no universal method that can be applied and it depends on the cycle finding algorithm used and the additional information extracted. Just analyzing each of these methods separately can give us the whole picture.

In order to ensure a random walk through the set, a random oracle in a form of a hash function is introduced. The decision fell on RadioGatún hash function. With the straightforward structure, changeable word size, easy implementation, possibility of modification of the function for certain purpose RadioGatún hash function can give analyzing and testing clear and simple.

Chapter 2

Cryptographic Hash Function

2.1 Introduction

A cryptographic hash function takes the message of arbitrary length as an input and maps to the fixed length output string. Moreover, cryptographic hash function protects against some attacks. The idea is to shift the protection of information authenticity of arbitrary data length to the protection of authenticity of fixed data length. The difference will be made depending on whether the protection of authentication relays on secrecy or not.

Hash functions are mostly based on randomness. The randomness ensures onewayness for the hash function. Most designs use block ciphers. However, in [41] Lai and Massey showed that the security of the hash function depends on the security of compression function. This as well as other properties described in this chapter leads to the fact that constructing iterated hash function requires careful design and implementation.

The best method to evaluate the hash function is to see which attacks are applicable. It is also assumed that it is possible to perform an adaptive chosen plaintext attack, where an adversary can freely choose the plaintext, ask for its hash value and try to evaluate the plaintext with the same hash value. There are several methods that can be used. In this chapter the attack applicable to the particular hash scheme will not be covered and the compression function is observed as the black box.

In this chapter, the basic concept of cryptographic hash function is explained. It also discusses about security issues related to hash functions, such as establishing an authenticity and/or an integrity over an unsecure channel. It provides a set of definitions, algorithms and schemes necessary for better understanding and gives the wide picture of its functionality.

The remainder of this chapter is organized as follows: the chapter starts with

the basic classification of hash functions. After that iterated hash function as the most common implementation is presented. Also different types of hash functions are observed. At the end of this chapter the most common general attacks are analyzed.

2.2 Classification of Hash Functions

A cryptographic hash function plays a central role in ensuring security properties in applications. A hash function is usually related to the digital signature, message integrity and authentication. It can also be used to produce pseudorandom numbers. First time a hash function was introduced in [22] by Whitfield Diffie and Martin E. Hellman in 1976. They used the hash function to ensure data integrity, signing a "digest" of the message rather than the whole message. They presented the protocol (aka Diffie-Hellman protocol) that generates a symmetric secret key. Until now, many ideas have been proposed in order to meet different application requirements. Nowadays, developers demand that the hash function in the application is fast enough. Also, the design of hash function should be publicly known [58].

Hash function must have at least two following properties:

1. *compression* - for any message x of arbitrary length, function h maps to fixed output length
2. *easiness of computation* - computation of function h is easy

This means that any function that has at least these two properties can be called a hash function. Because of different requirements in applications, a hash function usually has to have some additional properties. Hash functions are classified on the highest level on two disjoint classes: *unkeyed hash functions* where output depends only on the input message (Manipulation Detection Code or **MDC**) and *keyed hash functions* where the output depends on the input message and the secret key (Message Authentication Code or **MAC**). Depending on the requirements the first one can be further divided into *one-way hash function* (OWHF) and *collision resistant hash function* (CRHF) (see Figure 2.2).

In generall, applications that use hash functions require some of following three properties:

first preimage resistance for a given hash value $y = h(x)$, it is computationally infeasible to find a value x' such that $h(x') = y$

second preimage resistance for a given value x it is computationally infeasible to find a value $x' \neq x$ such that $h(x) = h(x')$

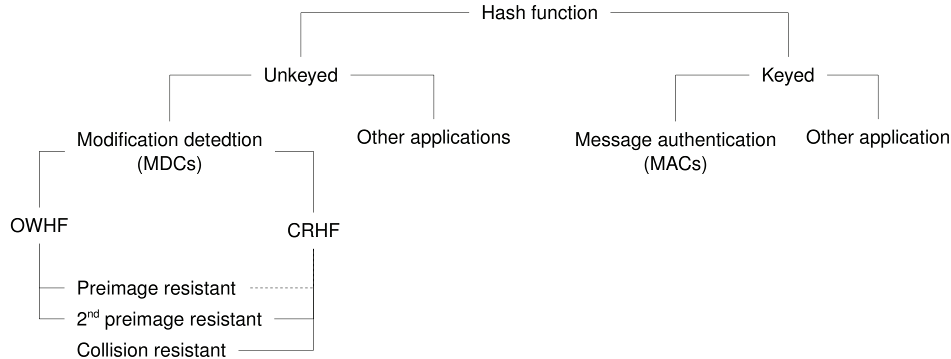


Figure 2.1: Classification of hash functions

Property	Security
preimage resistance	2^n
second preimage resistance	2^n
collision resistance	$1.2 \cdot 2^{n/2}$

Table 2.1: Ideal security for different properties of n-bit hash functions

collision resistance it is computationally infeasible to find any two values x and x', $x \neq x'$ such that $h(x) = h(x')$

Manipulation Detection Code is also known as *modification detection codes* or, not so often used, *message integrity codes* (MICs) [44]. The main goal of MDC is to ensure data integrity.

The hash function has an *ideal security* if the best known attack is a generic attack. If someone can find the attack that has the complexity higher than the ideal security for a given property, this hash function is considered broken. Still, it can be used in some applications. The complexity of generic attacks on the hash function is given in the Table 2.1.

one-way hash function (OWHF) is an unkeyed hash function that has the following properties: preimage resistance, second peimage resistance. It is also known as *weak one-way hash function*

collision resistance hash function (CRHF) is an unkeyed hash function that has the following properties: second preimage resistance, collision resistance. It is also known as *strong one-way hash function*

If the function is one-way, it doesn't guarantee that it is hard to find two inputs with the same output. This is the reason why a collision resistance as a separated property needs to be introduced.

Also, CRHF does not guarantee the preimage resistance by itself but in practice, CRHR almost always has this additional property. Here, the term collision resistance is not quite suitable because the collisions exist but it is hard to find them. An alternative name *collision intractable hash functions* was proposed in [79] and [78]. Another common used name is *collision free hash function*.

2.2.1 Manipulation Detection Code (MDC)

Manipulation Detection Code (MDC) (sometimes also called fingerprints, cryptographic secure checksums, hash function and others) was originaly defined by Merkle [45]. He described one-way hash function using a random block cipher.

One-Way Hash Function (OWHF)

Originally, OWHF was introduced by Diffie and Hellman in [22]. Before detailed explanation of OWHF, one-way function in general should be defined. The function h is a *one-way* if:

1. The design of the function h is public
2. For a given x, the computation of $h(x)$ is easy
3. For a given y in the range of h, finding x such that $h(x) = y$ is hard.

This definition is only informal. To make this definition formal one should specify the distribution to select x and y. One should also specified the meaning of "hard" and "easy". It must be taken into account that computing inverse function is possible but only with a small range (e.g. exhaustive search). Many other definitions are needed to make this definition formal and it is not a trivial task.

The first property does not need to be met if we want to call the function a one-way hash function but, by avoiding this statement, we have a situation known as security by obscurity which is against Kirchhoff's principle[1]. The existence of the one-way function is not proven but it is believed to be infeasible. It is still an open question: do one-way functions exist? The following functions are just some that are believed to be one-way:

- Factoring problem: $f(p, q) = pq$, where p and q are randomly chosen primes
- Discrete logarithm problem: $f(p, q, x) = \langle p,\ q, g^x (mod\ p)\rangle$, where g is a generator of Z_p^* and p is a prime number.

[1]Kirchhoff's principle says that even if everything, except the key of the cryptosystem is publicly known the system should be secure

- Discrete root extraction problem: $f(p, q, e, y) = \langle pq, e, y^e (mod\ pq)\rangle$, where p and q are primes, y is in Z^*_{pq} and e is in Z_{pq} and co-prime with $(p-1)(q-1)$
- Subset sum problem: $f(a, b) = \langle \sum_{i=1}^{n} a_i b_i, b\rangle$, where $a_i \in \{0, 1\}$ and n-bit integer b_i

Probably the best known usage of one-way function is for storing passwords. Namely, instead of storing the password p in plain form, the value of one-way function $h(p)$ is usually used. Correctness of the password can be easily checked and even the one who has access to the storage location cannot deduce the user's password. The definition of OWHF can be given in the following way:

Definition A function h that turns arbitrary input message length x into fixed length output message digest is called *one-way hash function* (OWHF) if:

- Function h is a one-way function
- For a given x and $h(x)$ it is hard to find $x' \neq x$ such that $h(x) = x'$

Collision Resistant Hash Function (CRHF)

A collision resistance hash function was mentioned for the first time in [16]. A construction of such a function was proposed and proven that the collision is hard to be found. The main motive why CRHF was introduced is to improve the security of signature scheme and channel authentication. It is also suggested in [44] that MDC should be CRHF if an adversary has a full control over an input of the hash function. Because of the collision resistance property, CRHFs are harder to construct than OWHFs and require an additional effort during its design.

Definition A *collision resistance hash function* is a function h that satisfied following properties:

- The design of the function h is public
- An input value x can be of arbitrary length and an output value $h(x)$ has a fixed length
- For a given x, it is easy to compute $h(x)$
- It is computationally infeasible to find any pair x and x' such that $x \neq x'$ and $h(x) = h(x')$

CRHFs are easier and much preferable to be used in a system than OWHF. Collision resistance is required for digital signature to preclude the repudiation to the sender. The reason is that there is no pre-conditions in selecting a message x and it provides higher level of security. Without satisfying a collision resistance

property, an adversary can produce a message pair (x, x') and let another party sign a message x. Later on, an adversary can use another message x' and clams that another party signed it. Namely, a constraint in a system with OWHF is that a message x cannot be chosen by a party who has a motive to break the system. Because of that, many authors exclusivelly recommend usage of CRHF.

2.2.2 Message Authentication Code (MAC)

MAC algorithms ware introduced in the late seventies after the start of open research in the field. In the beginning, a MAC was used in banking systems. MACs have deployed an application that operates with electronic purses (such as Proton, CEPS) and credit/debit (the EMV-standard). Also, MACs have been used to secure Internet (e.g., IPSec, TLS) [59]. The first reference was given in 1972 by Simmons at al. in [68].

Definition A *Message Authenticate Code (MAC)* is a function h with the following properties:

- The design of the function h is public and the only secret information is a key k
- An input value x can be of arbitrary length and an output value $h_k(x)$ has a fixed length
- For a given x and k it is easy to compute $h_k(x)$
- Even if an arbitrary number of pairs is known $(x_i, h_k(x_i))$ it is computationally infeasible to compute any pair $x, h_k(x)$ for any input $x \neq x_i$

The goal is that an eavesdropper who has an ability to change a message can't predict the MAC value. The security is based on the fact that it is computationally infeasible to predict the MAC value for a given message without knowing the key. MAC algorithm can be established only between mutually trusting parties. It requires less storage and it is much faster than digital signature[58]. The disadvantage is the fact that the management of symmetric keys is expensive.

CBC-MAC

The most popular MAC algorithm is CBC-MAC [56]. It is widely used in financial applications and smart cards. It is based on block cipher derived from CBC mode of operation. MAC key is used as a cipher key in each step of iteration (see section 2.3 for iterated hash function). CBC-MAC construction can be defined as iterated hash function with the following compression function:

$$H_i = E_k(H_{i-1} \oplus x_i), \;\; 1 \leq i \leq n$$

Where $E_k(x)$ is encryption of x using a key k. It is widely used with DES. Usually, the initial value is zero ($IV = H_0 = 0$). The final result is given using output transformation $g(H_n) = MAC_k(x)$. The output transformation g should avoid the following forgeries [56]:

- For given $MAC(x)$, follows that $MAC_k(x||(x \oplus MAC_k(x))) = MAC_k(x)$ using one block x
- For given $MAC(x)$ and $MAC(x')$ follows that $MAC_k(x||(x' \oplus MAC_k(x))) = MAC_k(x')$ using one block x
- For given $MAC(x)$, $MAC(x||y)$, and $MAC(x')$ follows that $MAC(x'||y') = MAC(x||y)$ if $y' = y \oplus MAC(x) \oplus MAC(x')$

The possible solution is to perform processing on the last block using two keys.

$$g(H_n) = E_{k1}(D_{k2}(H_n))$$

where D denotes decryption. This approach requires extra computations and needs an additional key but it is resistant on exhaustive search against DES. This solution is known as *ANSI retail MAC*.

2.3 Iterated Hash Functions

Most hash functions are based on the compression function with a fixed size input. An iterated hash function takes arbitrary input length that is successively processed to fixed-length blocks. An arbitrary length input $x = (x_1 x_2 ... x_n)$ is divided into l-bit blocks x_i. Input message usually needs extra bits to attain a length that is multiple of a block length. Each compression function takes two inputs of lengths m and l and produces an intermediate fixed-length outputs of length m. Denoting H_i as intermediate result after i-th iteration, the general process of computing a hash value with an input $x = (x_1 x_2 ... x_n)$ is:

$$H_0 = IV, \quad H_i = f(H_{i-1}, x_i), 1 \leq i \leq n, \quad h(x) = g(H_n)$$

where H_0 is a pre-defined *initial value*. H_i is usually called the *chaining variable*. Sometimes, after the last output is given, an *output transformation* g is applied so that it maps the l-bit value into m-bit result. Mostly, g is identity function.

An important condition in creating iterated CRHF was given by Damgård [17] and Merkle [45].

Theorem (Damgård-Merkle) Let f be collision resistance function that maps l-bit input to m-bit output ($l > m$). If unambiguous padding scheme is used the following construction yields a CRHF:

$$H_1 = f(0^{m+1}||x_1), \;\; H_i = f(H_{i-1}||1||x_i) \;\; (i = 2, ..., n)$$

They proved that collision resistance property of compression function f leads to collision resistance property of iterated hash function h.

One of the main facts in creating a secure iterated hash function h is how "good" the compression function f is. In other words, how should a function f be created so that h satisfies certain properties? One of the answer to this question was given in [41] by Lai and Massey. They gave necessary and sufficient conditions for compression function f in order to obtain second preimage resistance for h.

Theorem (Lai-Massay) Suppose that a message contains at least two blocks. Iterated hash function h with unambiguous padding scheme has ideal computational security against second preimage attack with a fixed initial vector IV if and only if its compression function f has ideal computational security against second preimage attack for chosen H_{i-1}.

From security aspect, two important elements are involved: the padding scheme and the initial vector. The padding scheme must be deterministic and unambiguous. That is, there are no two different messages that can be padded to the same one. At the end of padding part, the length of original message should be embedded. The IV must be pre-defined and it is a part of hash function's description. After Damgård [17] and Merkle[45] it is called MD-strengthening

2.3.1 Weakness in Iterated Hash Functions

These attacks concern the security of iterated hash function against generic attacks. It turns out that the iterated hash function has a number of generic attacks whose weakness does not lie in the poor design of compression function.

As it is already mentioned, a hash function takes the arbitrary amount of data. One of the most natural ways to solve this problem is to divide the input value into blocks and to proceed all blocks into some kind of iterated structure. This solution was initially proposed by Merkle in [46]. Constructing the good design is not an easy task. Schemes that have been used most often in hash functions are Davies-Meyer and Miyaguchi-Preneel [47]. These two schemes usually give enough security in most applications even though some weaknesses have been found.

Multicollisions in Iterated Hash Functions

Finding multicollisions in iterated hash functions can be done more efficiently than it was believed. Joux [31] estimated that finding 2^t collisions is only t times slower than constructing 2 collisions.

Finding r-mulsticollisions in cryptographic hash function means finding the r-tuple $(m_1, m_2, ..., m_r), r \geq 2$ for $m_i \neq m_j, i \neq j$, and $i, j \in \{1, 2, ..., r\}$ where

$h(m_1) = h(m_2) = ... = h(m_r)$. For a perfect random hash function finding r-multicollisions requires $r!^{1/r}2^{n(r-1)/r}$ operations [1]. Joux found a much faster way to find multicollisions in an iterated hash function. After finding a t pairs $(b_1, b_1'), (b_2, b_2'), ..., (b_t, b_t')$ where values in each pair produce the collision for the compression function f, one can easily generate 2^t massages of t blocks that generate the same hash value. It can be done by using the input $B = (B_1, B_2, ..., B_t)$ where B_i is whether b_i or b_i'. That is, finding 2^t multicollisions requires $t \cdot 2^{n/2}$ operations.

Using the advantage of r-multicollisions Joux found the way to improve the attack on cascaded hash functions whose component functions have an ideal security. Suppose that two iterated hash functions are given: h and g with n_h and n_g-bits outputs respectively and that there is no better attack on these hash functions than generic birthday attack. The large hash value can be construct concatenating two output values $(h(x)||g(x))$. Joux claims that better generic attack exists on $h||g$ than $2^{(n_h+n_g)/2}$. Moreover, he found the attack on $(h(x)||g(x))$ that found a collision with complexity of the order $n_g 2^{n_f/2} + 2^{n_g/2}$ if $n_f \leq n_g$ (respectively $n_f 2^{n_g/2} + 2^{n_f/2}$ if $n_g \leq n_f$).

A similar situation is when one tries to construct preimages. The complexity was believed to be $2^{(n_f+n_g)}$. However, there is a much better attack that can find a preimage of $f||g$ hash function with complexity of the order of $n_g 2^{n_f/2} + 2^{n_f} + 2^{n_g}$ if $n_f \leq n_g$ (respectively $n_f 2^{n_g/2} + 2^{n_g} + 2^{n_f}$ if $n_g \leq n_f$).

Attack on the Second Preimage

An ideal security of the hash function requires about 2^n operations to brake second preimage resistance. Dean showed [18] that in the iterated hash function, where it is easy to construct fixed points for its compression functions, the second preimage can be found much easier. He used the so called expandable messages to apply this attack. An idea is to use sets of pairs of messages. They are used to construct messages with the same hash values but different lengths. Instead of one block someone can use more blocks, as long as the hash value is the same.

To find a second preimage someone can construct an expendable message and connect it to the output hash value of the expendable message with the chaining values of the message. When the desired hash value is found one can adjust the expandable message and change the length to the length of the first part of original message. This way, the new message will have the same length as the original one. Using an expandable message one can produce messages length form k to $2^k + k + 1$ blocks to find second preimage from the message $m = (m_0, m_1, ...m_{2^k+k})$. The total complexity is $k2^{n/2+1} + 2^{n-k+1}$. In the case of SHA-1, for a very long message of 2^{60} bytes one can find a second preimage in 2^{106} rather than 2^{160} [43].

J. Kelsey et al. in [33] had a very similar approach for this problem. They showed that even without finding fixed points it is possible to find the second preimage with

the less than 2^n operations. For a long message of 2^k blocks they managed to find second a preimage with the work factor $k2^{n/2} + 1 + 2^{n-k+1}$.

Length-Extension Attack

This attack is applied on Merkle-Damgård scheme of iterated hash functions [24]. For a given hash function h, assume that an adversary knows the hash value $h(x)$ and the length $|x|$. An adversary can select the suffix s and compute $h(x||s)$ without knowing x. This can be done by padding the length of the message $x||s$ and the end. It is possible since an adversary already knows the length of x and he is free to chose the content of s.

2.4 Types of Hash Functions

This section discusses about three types of MDC hash functions: *hash function based on block cipher*, *custom design hash function*, *hash function based on algebraic structures* (modular arithmetic, knapsack, lattice problems).

2.4.1 Hash Functions Based on Block Cipher

The usage of hash functions based on block cipher has historical and practical reasons. The first reason is because DES was the first cryptographic primitive that was widely available. It seems pretty natural for designers to use the potential of this block cipher to construct a hash function. Another reason is the minimization of the design and an implementation effort. A good hash function is hard to design and designers can use security property of block cipher to construct it. Also existing software and hardware implementation can be reused. One of the main advantages is trust that can be transfered to the hash function. On the other hand, hash functions based on a block cipher are not fast and efficient as these that have a custom design. It must be also taken into account that using block ciphers additional problems in hash functions may be created. They can expose some weaknesses that only occur in hashing mode [59]. The usage of block ciphers in hash functions requires stronger properties of block cipher. That is, certain properties of a block cipher that do not affect the process of the encryption can cause some problems in hash functions. For example, semi, weak and quasi weak keys can cause some problem [50, 38]. DES can be exploited in differential cryptoanalysis of the hash function [61].

An encryption of the block cipher is defined as $c = E_k(p)$ where p is a plaintext, c is a ciphertext and k is a key. The size (in bits) of a plaintext is denoted by r and of key by k. A *hash rate* of the hash function based on block cipher is defined as the number of block inputs necessary for a single encryption. DES has 1 hash rate of $r = 64$ bits and a key size $k = 56$ bits.

A difference will be made regarding the size of the result of the hash function. Thus, three cases will be considered: when the size of the result is equal to the block size ($n = r$), when the size of the result is equal to twice the block size ($n = 2r$) and when the size of the result is greater than twice the block size ($n > 2r$). A motive for a greater size of the result lays in the fact that the result of one block size became too small to obtain resistance on collisions. An alternative approach is to obtain a greater key size [57].

Result of Hash Function Equal to the Block Size

All schemes of this type have a hash rate 1. The first secure construction of this type was given by Matyas et al. in [69]. It is known as Matyas-Meyer-Oseas scheme and it has the form:

$$H_i = E_{s(H_{i-1})}(x_i) \oplus x_i$$

where s represent a compression function that maps a cipher space to the key space. Another one, very similar to the previous one is Davies-Meyer scheme: It fits much better in the case when the key size and block size are different [57]. The compression function in the Davies-Meyer scheme compresses the $n + k$ bits into b bits and has the form:

$$H_i = E_{x_i}(H_{i-1}) \oplus H_{i-1}$$

By making iterations using this scheme and combining them with MD-strengthening one can construct a hash function that is known as Davies-Meyer hash function. It was shown that using the block cipher whose properties have an ideal security and if $k \geq n$ finding (second) preimage requires about 2^n and collision $2^{n/2}$ encryption [8].

The third variant is a Miyaguchi-Preneel scheme and it was proposed independently by Preneel et al. [2] and Miyaguchi et al. [48].

$$H_i = E_{s(H_{i-1})}(x_i) \oplus x_i \oplus H_{i-1}$$

Preneel et al. in [55] identified 12 different security constructions and briefly discussed about their security. Black et al. in [8] went a little further and proved the security of these schemes.

Result of Hash Function Equal to Twice the Block Size

The goal of a double block size hash function is to increase security against collision attacks. A collision attack in an ideal case needs approximately $2^{n/2}$ encryptions and a second preimage attack 2^n [57]. Probably the most famous examples are MDC-2 and MDC-4. They required 2 and 4 block cipher operations per block for hash input [44] or in other words MDC-2 and MDC-4 have 1/2 and 1/4 hash rate respectively. They combine 2 or 4 Matyas-Meyer-Oseas schemes and produce a double block length result. In original specification, when DES was used as a block cipher, 128-bit output hash value is produced but any other block cipher can be used. An MDC-2 scheme has a form:

$$
\begin{aligned}
T_i^1 &= E_{u(H_{i-1}^1)}(x_i) \oplus x_i = LT_i^1 || RT_i^1 \\
T_i^2 &= E_{u(H_{i-1}^2)}(x_i) \oplus x_i = LT_i^2 || RT_i^2 \\
H_i^1 &= LT_i^1 || RT_i^2 \\
H_i^2 &= LT_i^2 || RT_i^1
\end{aligned}
$$

where H_0^1 and H_0^2 are initial vectors IV_1 and IV_2 respectively. A hash value is calculated concatenating the values H_n^1 and H_n^2. It is also required that $u(IV_1) \neq v(IV_2)$.

The construction of MDC-4 consist of joining together two MDC-2 where outputs of the first step ($H2_{i-1}$ and $H1_{i-1}$) are used as the inputs for the second step:

$$
\begin{aligned}
T1_i &= E_{u(H1_{i-1})}(x_i) \oplus (x_i) = LT1_i || RT1_i \\
T2_i &= E_{v(H2_{i-1})}(x_i) \oplus (x_i) = LT2_i || RT2_i \\
U1_i &= LT1_i || RT2_i \\
U2_i &= LT2_i || RT1_i \\
V1_i &= E_{u(U1_i)}(H2_{i-1}) = LV1_i || RV1_i \\
V2_i &= E_{u(U2_i)}(H1_{i-1}) = LV2_i || RV2_i \\
H1_i &= LV1_i || RV2_i \\
H2_i &= LV2_i || RV1_i.
\end{aligned}
$$

The best known preimage attack on MDC-2 requires $2^{63.3}$ compression function evaluations [39]. In [65] an efficient collision attack on MD-4 with complexity less than 3 MD-4 hash operations is presented.

Result of Hash Function Larger Than Twice the Block Size

Knutsen and Preneel proposed a scheme for a collision resistance compression function [37]. They proved the collision resistant security level equal to $2^{n/2}$, $2^{3n/4}$ or more and all this with rates larger than 1/2. The internal memory in this case requires more than 2 or 3 blocks and an output transformation must be made [57].

2.4.2 A Custom Design Hash Function

This class of hash functions is designed to perform hashing operations. What is common for all custom designed hash functions, is that designers knew the purpose of the hash algorithm in advance with performance efficiency in mind. Most of them are based on Davies-Meyer construction. Rivest suggested in [64] MD4. It has optimal logic operations and integer arithmetic on 32-bit processors [57]. Design principles of MD4 were subsequently used in the construction of other hash functions such as SHA-1 and RIPEMD-160. After a serious leak was found in MD4, Rivest designed MD5 [63] to replace it. There is no proof for security of custom design hash functions and a chance for attacks always exists.

Boer and Bosselaers showed in [19] that compression function of MD5 is not collision resistant. Namely, they have found a collision with distinct initial values but the same input messages. Because of different initial values they can not be used in Merkle-Damgård method and do not have direct impact on applications. Sometimes are also called pseudocollisions. Dobbertin has found the collision with the same initial values but distinct input messages for the compression function [23]. Initial value that Dobbertin has found is different than initial value in Merkle-Damgård specified by MD5 and this is the reason why it can implied a collision of MD5.

X. Wang, D. Feng, X. Lai and H. Yu showed that MD5 can not be considered a collision resistant function. Moreover, in August 2004. they published collisions for MD4, MD5, HAVAL-128 and RIPEMD-128 [75]. Nowadays, MD5 is considered broken. In [36] Klima demonstrated how a collision for any initial value on an average PC can be found in less than a minute. The best known attack was found in 2009. that uses only $2^{20.96}$ to beak collision resistance of MD5 [76].

In February 2005. X. Wang, Y. L. Yin and H. Yu were presented an algorithm that finds a collision in SHA1 with 2^{69} hash operations [74]. This is significantly less than 2^{80} hash operations of the brute force birthday attack. De Canni'ere et al. in [11] gave the best known example of 70-step collision for SHA1.

2.4.3 Hash Functions Based on Algebraic Structures

This section shortly discusses about hash factions based on modular arithmetic. It also shortly covers the hash functions based on knapsack problems. The last part of this section is reserved for incremental hash functions. Some of these hash functions are vulnerable to the insertion of a trapdoor. A trapdoor one-way function is a such a function where it is hard to invert unless some secret information, also called a *trapdoor*, is known. This allows someone to define parameters that can lead to the collision and other vulnerabilities. An example for a trapdoor one-way function is a factorization of a product of two large primes. Computing the product of two

large prime numbers is relatively easy but factoring them is (believed to be) hard unless some additional information is known. This is the basis of an RSA algorithm. Because of that, these parameters must be generated very carefully.

Hash Functions Based on Modular Arithmetic

An idea is to use a modular arithmetic in an iterated hash function as a basis for a compression function. The security is based on the difficulty to solve some well-know number theoretic problems. Some of these problems are discrete logarithm problem and factorization problem. The advantage of hash functions designed in this way is that a digest length is scalable, depending on the size of modulus. The disadvantage is that homomorphic structure can be exploited as well as the fixed point of modular arithmetic.

Hash functions based on modular arithmetic can be divided into two classes; some that have provable reduction scheme and some that do not have provable reduction to the underlying hard problem. Schemes without provable reduction usually use factorization problem. It can be very practical in combination with RSA as the digital scheme. Here, the following problem can be exposed: the party that generates the modulus knows the factorization and has an advantage over other parties. One of the solutions is to use *trusted third party* (TTP) to generate the modulus.

Hash Functions Based on Knapsack Problems

In cryptography, the knapsack problem of dimension n and $l(n)$ can be defined as follows: For a given set $A = \{a_i | i = 1, ..., n\}$ where a_i is a b-bit integer, and an s-bit integer S find a subset $A' \subseteq A$ such that $\sum_{a_i \in A'} a_i x_i = S \bmod 2^{l(n)}$. A knapsack problem belongs to the well known NP-complete problems. The hardware and software implementation are much faster than schemes based on number theoretic problems. Nevertheless, the hard mathematical basis of almost all hash functions based on knapsack problem have been broken. That brought a bad reputation of the knapsack problem in cryptographic community.

Incremental Hash Functions

This type of hash function is relatively new. It was introduced 1994. by Bellare at al. in [4] and has the following property: if the hash massage x is slightly changed into the new message x' then the computation of $h(x')$ should be proportional to the amount of modification between x and x'. This hash function has found its purpose in applications such as virus protection, broadcast networks, video surveillance broadcasting [4] and memory checkers [25].

2.5 Attacks on Hash Functions

An attack on a cryptographic hash function means creating an algorithm capable of violating one of the security properties of the hash function. For instance, if a hash function is claimed to be collision resistant, a successful attack will be to find any two distinct messages that have the same hash value. It is assumed that the algorithm of the hash function is a public knowledge. It is also assumed that it is possible to perform the, so called, adaptive chosen message attack, where an adversary may choose messages and calculate a hash value and then try to find a message with the same hash value. Several approaches for violating the security hash functions will be presented in the rest of this section.

Preneel in [55] gave the classification of the known methods of attack on hash functions. He proposed the following classification:

1. attacks independent of the algorithm,
2. attacks dependent on the chaining,
3. attacks dependent on an interaction with the signature scheme,
4. attacks dependent on the underlying block cipher,
5. high level attacks.

In the case of MDC an attack consists of finding preimage or two different values with the same image. A special treatment will be taken for IV depending whether the value of IV is different from the specified value. In [55] a distinguish was made between:

Preimage an adversary tries to find a preimage for the given hash value

Second peimage an adversary tries to find a second preimage for a given hash value

Pseudo-preimage an adversary tries to find a preimage for a given hash value where $IV \neq IV'$

Second pseudo-preimage an adversary tries to find a second preimage for a given hash value where $IV \neq IV'$

Collision an adversary tries to find two distinct values that have the same hash value

Collision for differetn IV an adversary tries to find two distinct values that have the same hash value where $IV \neq IV'$

Pseudo collision an adversary tries to find a pair x and x' for same IV and IV' such that $h_{IV}(x) = h_{IV'}(x')$

2.5.1 Attacks Independent of the Algorithm

Attacks that belong to this class do not depend on the structure of the hash function. These attacks do not analyze the algorithm of the hash function. They only depend on the size of the hash value and on the size of the key in the case of MAC. All calculations are assumed to have their output values uniformly distributed. Examples of these attacks are: exhaustive key search, birthday attack, random attack and pseudo attack.

Exhaustive Key Search

The key in the MAC function is used to make the algorithm secure. It is presumed that an adversary has one or more pairs $(x, h_k(x))$. An adversary then tries to find a key that corresponds to a given pair. The key can be found examining the key space. The relation between the plaintext and the hash value does not need to be one-to-one. Sometimes more than one key, fitting the given pair $(x, h_k(x))$, can be found. However, by finding all possibilities for more than one pair $(x, h_k(x))$, the key can be determined uniquely.

As it shown in [55], the expected number of trials is given by

$$(1 - \frac{1}{2^r}) \sum_{i=1}^{m} \frac{1}{2^{r(i-1)}} < \frac{1}{1-2^{-r}}$$

where r is the length of hash value in bits and m is the number of pairs $(x, h_k(x))$. The upper bound of trials to identify the key is:

$$m + \frac{2^k - 1}{1-2^{-r}}$$

where k is the key length in bits. Expected number of resulting key is given by

$$1 + \frac{2^k - 1}{2^{mr}}$$

This means that expected number of pairs $(x, h_k(x))$ to determine the secret key is about $\frac{k}{r}$.

The time required for finishing exhaustive key search depends of the number of possible keys (k), the time for performing the hashing using one candidate for the key (t) and the number of processors used for finding the key (p). Each processor will find about k/p keys and would take kt/p time to perform the hashing. On average it is expected to find the key on the half way. This make the expectation time to find the key $\frac{kt}{2p}$.

The big part of the history of exhaustive key search is related to Data Encryption Standard (DES) [52]. When DES has been announced it was considered controversial mostly because of its short key. Deffie and Hellman in 1977. [21] made an estimation that DES can be broken in a day using a machine of 20 million dollars. In 1993., Wiener was presented a design of 1 million dollars machine for DES search key. This machine consisted of $57,600$ chips, each capable to test a key every $10ns$. This lead to the fact that the expectation for finding a secret key was $\frac{kt}{2p} = 3.5$ hours.

In 1998. Electronic Frontier Foundation built a machine whose main purpose was a DES key search. The machine was made with the budget of $250,000$$ using $1,856$ custom chips each capable of testing 60 million keys per second. The machine found a key in a little more than 2 days.

There are some well known suggestions for how to prevent exhaustive key search. One of the methods that is usually suggested is to use sufficient long key. Increasing the key length, the searching time $\frac{kt}{2p}$ becomes larger.

Birthday Attack

This attack is based on the birthday paradox. Namely, the probability that among of groups of randomly chosen people at least two have birthday the same day is more than 50%. It follows from

$$\frac{365}{365} \cdot \frac{365-1}{365} \cdots \frac{365-22}{365} \approx 0.49 < 0.5$$

or, in general

$$\prod_{0 \leq i \leq 1.18\sqrt{p}} \frac{p-1}{p} \approx 0.5$$

In other words, it is reasonable to expect a duplicate after about $\sqrt{p}$ randomly chosen elements. It is called a paradox because the number is significantly smaller than we would expect. Translating this into cryptology, this means that if the output of the hash function has p distinctive values, the expectation for two equal hash values is about after $\sqrt{p}$ randomly chosen inputs.

Some applications have problems related to this. They can be defined as follows: if there are two groups and in each group are 17 randomly chosen persons, then the probability that two people in two different groups have birthday on the same day is more than 50%. General approach can be as follows: if there are two sets r_1 and r_2 of randomly chosen values and if it is assumed that the hash value has r bits output, the probability of finding a match between two sets r_1 and r_2 is about

$$P \approx 1 - e^{-\frac{r_1 r_2}{2^r}}$$

where r_1, r_2 are big values [53]. In special cases, when $r_1 = r_2 = 2^{r/2}$ and $r_1 r_2 = O(\sqrt{r})$ the probability of a match is

$$P \approx 1 - 1/e \approx 0.63.$$

One very important characteristic is that by slightly increasing either r_1 or r_2 the success probability will be increased drastically. If $r_1 + r_2$ has to be minimized, it can be easily calculated that $r_1 = r_2 = \sqrt{n}$. This is the reason why this attack is sometimes called "square root attack".

Yuval in [77] exploited the birthday paradox to attack digital signature scheme of Rabin. He showed that it is easier to find a collision for a one-way hash function than to find a preimage of a second preimage of a specific value. The result is that a signature scheme can be vulnerable to Yuval's attack. This attack is applicable on all unkeyed hash functions.

Yuval's algorithm works the following way. Suppose that is given n-bit one-way hash function. First, an adversary generates r_1 hash values from the randomly chosen plaintexts and stores them in a searching way. Usually, $2^{n/2}$ such values are provided. After that, an adversary computes hash values from some another randomly chosen plaintexts, which are different from the previous set of plaintexts. After each computation an adversary checks if that value matches some of the values in the store. This continues until a match is found. A match is expected after $2^{n/2}$ candidates.

Random Attack

In this attack an adversary choses a random plaintext and hopes that it will produce the hash value equally to the genuine one. For an ideally secure hash function, the probability that an adversary remains undetected is $1/2^n$ where n is the number of bits of the hash value. The efficiency of this attack depends on the strategy taken after detecting the wrong result, the expected value of the successful attack and the number of trials that can be performed [2].

The important characteristic of MDC is that it can be used off-line. This implies that the number of hash values can be produced sufficient enough. On the other hand, MAC can be used on-line and parallel and depends on several elements described in [55]. First of all, an adversary is limited by the time he or she can spend using the system with the same key as well as the speed needed to calculate the hash value. The number of trials can be limited by the number of faulty results. After this time has exceed an adversary has some kind of penalty that can be in the form of waiting some period of time until the next trial. Also, an adversary can be motivated by the benefit of a successful attack.

Pseudo Attack

Bakhtiari at al. have called this attack pseudo attack in [3] since the cryptanalyst tries to find a pseudo key $\hat{k}$ such that $h_{\hat{k}}(x) = h_k(x)$ where k is a real secret key. For an adversary that knows the pseudo key in this way of impersonating another user is allowed. Figuring out the pseudo key doesn't automatically mean that an adversary can generate a correct hash value for another plaintext. Even more, if an adversary finds a pseudo key that generates correct hash values for t plaintexts it does not necessary mean that this pseudo key will produce the correct hash value for another plaintext.

2.5.2 Attacks Dependent on the Chaining

Chaining attacks are those which exploit the nature of iterated hash functions. These attacks depend on some properties of the compression function f and the focus of these attacks is more on the compression function f than on the whole hash function.

Correcting-Block Chaining Attacks

Here, an adversary creates a new plaintext x', correcting only one block in x, such that $h(x') = h(x)$. Let us suppose that the compression function $f : \{0,1\}^{n+m} \rightarrow \{0,1\}^n$ is given and the iterated hash function is defined as $H_{i+1} = f(H_i, x_i)$, $H_0 = IV$. An adversary picks one of the input blocks x_i and replaces it with another block x_i' without changing the output. Usually this means that an adversary tries to find an input block value x_i' such that $f(H_i, x_i) = f(H_i, x_i') = H_{i+1}$. Sometimes it is necessary for more than one block to be corrected but it doesn't change the essence of the problem.

Having the first/last block in the chain attacked is sometimes called *correcting first/last block attack*. Hash functions based on modular arithmetic are extremely vulnerable to this attack [55]. This attack can be used to find preimages, second preimages as well as collisions. It can be avoided using per-block redundancy, but it decreases the performance.

Meet-in-the-middle Chaining Attacks

Meet-in-the-middle is a known plaintext attack. It is a variation to the birthday attack that uses space-time tradeoff. It tries to find a collision on the intermediate result rather than the overall hash result. An adversary creates a r_1 variations on the first part of the plaintext and r_2 variations on the second part of the plaintext. Later on, by using values from the first variation going forward ($H_i = f(H_{i-1}, x_i)$) and values from the second variations going backwards ($H_i = f^{-1}(H_{i+1}, x_{i+1})$) an

adversary tries to match the same intermediate result. To be able to find a match, the chaining mode must be invertible and allow the attacker to go backward.

Fixed Point Chaining Attack

The goal of this attack is to find a pair (H_{i-1}, x_i) such that $f(H_{i-1}, x_i) = H_{i-1}$. In that case it is possible to insert an arbitrary number of blocks x_i without changing the hash value. This attack is only possible if the chaining variable can be forced to be equal to H_{i-1}, which is called a fixed point. It can be done in the following cases:

- large number of fixed point can be constructed. That is, for almost all values of chaining variables H_{i-1}, one can find x_i such that $f(H_{i-1}, x_i) = H_{i-1}$.
- it can somehow be arranged that the chaining variable takes the fixed point value.
- IV can be freely chosen. Then, one can chose IV for the fixed point.

This attack is also concerned in the case when an adversary can find k pairs (H_{i+l-1}, x_{i+l}), $l = 0 \ldots k-1$ such that $f(H_{i+l-1}, x_{i+l}) = H_{i+l}, 0 \leq l < k-1$ and $f(H_{i+k-2}, x_{i+k-1}) = H_{i-1}$. The attack can be prevented by making redundancy to the input block and embedding the plaintext length to the input, before performing the last block. One of the scheme vulnerable to the fixed-point attack is Davies-Meyer scheme pointed out in [49].

Differential Chaining Attacks

Primary, differential cryptanalysis was applied to block ciphers but it can also be used on hash functions as well as on stream ciphers. It analyzes how differences in the input affect the differences at the output. A chosen plaintext attack is usually when an adversary choses a set of plaintexts trying to obtain desirable set of outputs. Usually, an adversary searches for some statistical anomalies on an output. In case of collisions the output difference should be zero. If the iterated hash function uses the block cipher, depending on the mode of operations, an adversary can require that the output difference is zero or equal to the input difference.

2.5.3 Attacks Dependent on the Underlying Block Cipher

The hash function based on block ciphers was discussed in the section 2.4.1. Sometimes even a well designed encryption algorithm can cause problems in the iterated hash function. The problem lays in the fact that the algorithm is designed only

with encryption/decryption in mind. However, an underlying problem can arise when such block cipher is used in the round function of a hashing algorithm. The block ciphers in a hash function require much stronger properties. Examples of such problems are weak keys, key collisions, fixed points and complementation properties.

- *complementation property*: $y = E_k(x) \Leftrightarrow \bar{y} = E_{\bar{y}}(\bar{x})$ where $\bar{x}$ denotes the bitwise complement. It is one of the well-known properties of DES that reduce the exhaustive key search by factor 2. It also allows making a trivial collision. A linear transformation of the Matyas-Mayer-Oseas function that has the compression function $f(H_{i-1}, x_i) = E_{H_{i-1} \oplus x_i}(x_i) \oplus x_i$ produces the same result for x_i and for $\bar{x}_i$.

- *weak keys*: $E_k(E_k(x)) = x$ for all x. For example, DES has 4 weak keys. This allows an adversary to easily create fixed point of compression function f in only two steps. It is enough to use the same block input x_i that contains the weak key. The iterated block cipher with the *semi-weak keys* (DES has 6 semi-weak keys) has a similar property.

- *fixed points*: $E_k(x) = x$. Already explained in section 2.5.2. If a block cipher behaves like a random mapping then it probably has a fixed point. Nevertheless, finding the fixed point should be hard even if the adversary has control over the plaintext or the key. For a poor design iterated hash function producting fixed points is easy under some conditions.

- *key collisions*: $E_k(x) = E_{k'}(x)$. If the block cipher behaves like a random mapping, the key can be found using the birthday attack. To avoid such an attack, the key should be sufficiently large. Even if such a threat exists the good design of a hash function can make such an attack useless.

2.6 Summary

In this chapter, the most important security issues that are relevant to cryptographic hash function were described. The basic design and requirements as well as the attacks against hash functions were explained. The potential risk and countermeasures for some of these were analyzed. The focus was on generic attacks where the whole hash function or just the part of them is observed as the black-box. That is, attacks that analyze the internal structure of functions were not discussed here.

Chapter 3

RadioGatún

3.1 Introduction

In this section RadioGatún hash function will be explained. It is a family of hash functions proposed by Bertoni et al. in [6]. It compresses variables using iterated applications of round functions. In general, it is divided in two pairs: *belt* and *mill* are sometimes also called *belt-and-mill* structures. RadioGatún has been proven as a hash function with good performance and it is extremely fast in hardware.

The rest of the chapter is organized as follows: after the basic description and presentation of the structure of RadioGatún the performance and security overview was given.

3.2 Basic Description

RadioGatún is derived from Panama, intended to improve design and correct problems in Panama [15]. It is based on alternating-input of Iterative Mangling Function (IMF) with belt-and-mill structure. The concept of IMF structure is presented in Figure 3.1. It uses the iterative model, processing the iterative fixed-size input block, followed by the fixed number of rounds without inputs or outputs. The input mapping maps the input block bits to the bits updating the internal state at every round. The output is the fixed-size hash value. The length of the output is adjusted consecutively applying the round function where the output block is taken from mapping the internal state.

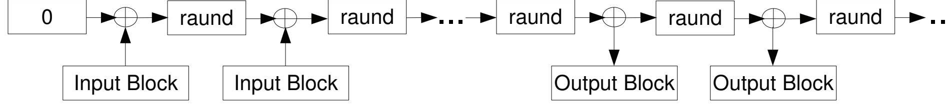


Figure 3.1: The structure of IMF

3.3 Structure of RadioGatún

The round function is the crucial part of any alternating-input IMF. The functionality of RadioGatún can be divided into two parts: the belt and the mill, and the round function. The mill function is applied to the belt. It is an invertible, nonlinear function. Bell function is an invertible, linear function and it is applied to the belt.

The mill has 19 words denoted by $a[i]$. The belt has 13 stages each consisting of 3 words denoted by $b[i, j]$. The input block consist of 3 words $p[i]$. The output block consists of 2 words $z[i]$. The diagram of the round function R is given in the Figure 3.2. An Algorithm 1 [6] defines the round function:

Algorithm 1 The Round Function R

1: $(A, B) = R(a, b)$
2: **for all** i **do**
3: $\quad B[i] = b[i + 1 \; mod \; 13]$
4: **end for**{Belt function: simple rotation}
5: **for** $i = 0$ to 11 **do**
6: $\quad B[i + 1, i \; mod \; 3] = B[i + 1, i \; mod \; 3] \oplus a[i + 1]$
7: **end for**{Mill to belt feedforward}
8: $A = \text{Mill}(a)$ {Mill function}
9: **for** $i = 0$ to 2 **do**
10: $\quad A[i + 13] = A[i + 13] \oplus b[12, i]$
11: **end for**{Belt to mill feedforward}

The round function R uses the mill function described in the Algorithm 2:

The input mapping that uses the input blocks to alter the internal state is specified by the Algorithm 3:

The output mapping that takes the value from the internal state is specified by the Algorithm 4:

RadioGatún has an internal state of 58 words of size l_w bits. Using a different value for l_w RadioGatún is treated as another function. The round function treats

Algorithm 2 The mill function

1: $A = \text{Mill}(a)$
  {all indices should be taken modulo 19}
  {$x \ggg y$ denotes rotation of bits within x over y positions}
2: **for all** i **do**
3: $A[i] = a[i] \oplus \overline{\overline{a[i+1]}a[i+2]}$
4: **end for**{γ: non-linearity}
5: **for all** i **do**
6: $a[i] = A[7i] \ggg i(i+1)/2$
7: **end for**{π: intra-word and inter-word dispersion}
8: **for all** i **do**
9: $A[i] = a[i] \oplus a[i+1] \oplus a[i+4]$
10: **end for**{θ: diffusion}
11: $A[0] = A[0] \oplus 1$ {ι: asymmetry}

Algorithm 3 The mill function

1: $(a, b) \leftarrow 0$
2: **for** $i = 0$ to 2 **do**
3: $b[0, i] = p[i]$
4: $a[i + 16] = p[i]$
5: **end for**
6: **return** (a, b)

belt and mill parts differently. The mill is fed with 3 words from the input block and 3 words from the belt in a linear way. Afterward, it is updated with a nonlinear function. The belt uses the linear transformation and it is fed with 3 words from the message block and 12 blocks from the mill in a linear way. The function defined this way is invertible.

The input message is padded in an appropriate way and divided by 3-word blocks. The initial state is 0 for all internal words. The 3-word input block is injected and then the round function is applied. This is repeated until the whole message is injected and then 16 blank round is processed. In the last phase the infinite number of output blocks is produced by consecutively applying the round function and taking 2-words block from the mill.

Algorithm 4 The output mapping F_0

1: $z[0] = a[1]$
2: $z[1] = a[2]$
3: **return** z

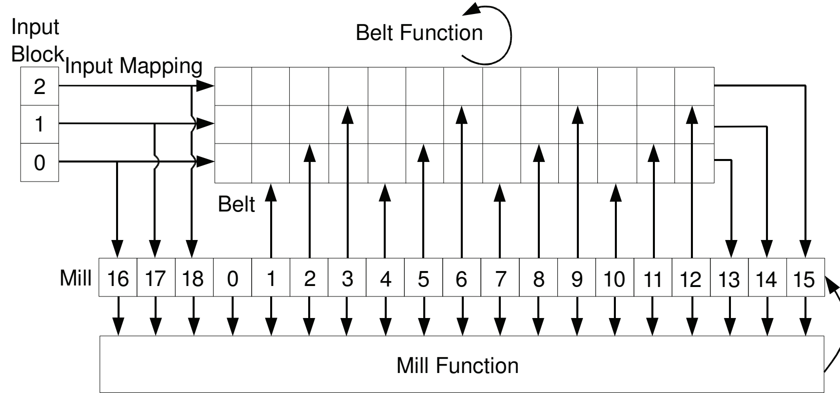


Figure 3.2: The RadioGatún round function

	Windows (32 bits) VS 2005	Linux (x86 64) GCC 3.3.5.
SHA-1	90	91
SHA-256	65	80
Panama	480	288
RadioGatún[32]	120	175
RadioGatún[64]	55	270

Table 3.1: Software performance (MByte/sec)

3.4 Performance

A basic performance overview comparing RadioGatún to Panama, SHA-1 and SHA-256 was given in [6].

The results from the software implementation taken from Dell Precision 670 with Intel Xeon 3GHz, comparing the speed for hashing the long inputs, was given in Table 3.1. It can been seen from the table that for $l_w = 32$, which is supposed to be secure as SHA-256, RadioGatún is two times faster. Worse results can be expected in the case when short inputs are used, due to the fact that fixed length of blank rounds and padding scheme become noticeable. Implementation in hardware is very simple and shows extremely good performance. A more detailed explanation about the optimization of hardware implementations, along with a comparison of some experimental results with other hash functions, is given in [6].

3.5 Security of RadioGatún

The design of RadioGatún was built as the improvement of Panama hash function. Panama was designed in 1998. and had a big influence on many other designs of hash functions. After breaking the Panama hash function in 2002. [62] the serious problem in design was revealed. In 2007. a practical attack on Panama was presented [14] which had unaccepted complexity of collisions and it is considered broken. In 2006. RadioGatún that relays on underlaying Panama hash function was introduced. RadioGatún did not have the weaknesses which Panama had. Up to now RadioGatún is considered secure.

In the original paper the authors claim that RadioGatún has a security level indicated by a capacity $l_c = 19l_w$ where l_w is a word length and it can be applied on both collision and second-preimage attack. For the word length of 64 bits it gives a capacity of 1216 bits. That is, taking the first l_h bits of output stream it can be used as a hash function with a l_h-bits digest. Here, it must be noticed that taking the hash value $l_h > l_c$ the collision resistance depends on l_c rather than l_h. They also claim that security level for both attacks is $2^{9.5l_w}$. Thus, the best attack is not generic [7].

In [34] Khovratovich described two attacks but neither of them broke the security claim of RadioGatún. They found the collisions and second preimage using meet-in-the-middle attack with complexity 2^{18l_w} and $2^{23.1l_w}$ respectively. Dmitry Khovratovich also described another attack in [35] with complexity of 2^{18l_w}.

Another attack was described in [9] applied on a 1-bit version of RadioGatún and it managed to find a collision in $2^{24.5}$ operations. However, it can not be applied on other bit versions of RadioGatún. This attack still does not break security claim but has the complexity less than the birthday paradox.

The best attack so far was presented in [28]. The attack presented in this paper has a complexity of 2^{11l_w} but as the other attacks it does not break the security claim of RadioGatún.

3.6 Summary

The structure of RadioGatún is straightforward, enabling clear and simple analysis. The simplicity does not imply that the hash function is not secure. What is more, all well known attacks are not even close to break the security claim of RadioGatún. The invertible characteristic ensures that meet-in-the middle attack can be implemented and tested. Taking into account that the performance in the software as well as the hardware implementation achieves enviable results, RadioGatún gives us a good choice for hash function.

It has slightly weaker performance if short input message is used but in some

situations it can be overcome, observing the internal state but not the output value. Namely, the collision of two distinct inputs in RadioGatún hash function can be found right after the input blocks. In other words, blank rounds and output blocks that drastically reduce the search for the collision can be skipped. With this approach the problem with small inputs can be overcame.

Chapter 4

Cycle finding algorithms

4.1 Introduction

In this chapter several algorithmic techniques which apply the birthday paradox on randomly chosen object are introduced. All algorithms are focused on detecting periodicity in a sequence generated by iterating a fixed function. Most of the techniques are possible to adjust in order to meet certain requirements. This is very important since this class of functions has many applications where it can be exploited. Some of these applications are: determining the cycle length of a pseudo-random number generator, detecting an infinite loop and specially some algorithms based on Pollard's Rho method for integer factorization.

In general, all cycle finding algorithms can be defined in the following way. Supposed that a function h from some finite set S to itself is given. Then, defining the starting point x_0 from S we generate the sequence $x_{i+1} = h(x_i)$ for $i \geq 1$. Because the set S is a finite, there must exist some i and $j(\neq i)$ such that $x_i = x_j$. Also, $x_{i+1} = h(x_i) = h(x_j) = x_{j+1}$. It is obvious that $x_{i+2} = x_{j+2}$, $x_{i+3} = x_{j+3}$ and so on. If such a pair (i, j) can be found where $i \neq j$ and $x_i = x_j$ then it can be said that the cycle is detected and that the object x_i is in the cycle.

All methods were based on an assumption that the function h behaves more or less randomly. Without that assumption the birthday paradox can not be applied. For such defined function a collision is expected to be found after $2^{n/2}$ elements. It can be simply assumed that the sequence of elements x_i makes a loop and comes back to some of the previous values. A special case is when this loop comes back to the start value x_0 and does not produce the collision which is also known as "Robin Hood" [72].

The remainder of this chapter is organized as follows: at the beginning Pollard's rho factorization algorithm is given as the main idea of cycle detection explanation. Next, the chapter observes behaviors of random functions. The rest of the chapter

explains other well known cycle detection algorithms.

4.2 Objectives

A set of methods that do not require a large amount of memory to detect the cycle will now be presented. Here, detecting the cycle (what this chapter is about) should not be mixed with finding the collisions (that is described in details in Chapter 5). Some of them require minimal and constant amount of memory like Floyd's and Brent's algorithm. For finding the cycle more memory can be used but detecting the cycle can be more faster. In general, these methods store more previously computed values and compare the new value with them. In this case, authors very often refer on the time-memory tradeoff. All these methods exploit the behavior of random functions. Analyzing the behavior the random function reveals interesting and sometimes unexpected structures.

A short review is given on mathematical and historical aspects of the problem. They exploited the group structure in which the function defines the random walk (comparing to cycle finding algorithm which is applied on any set on which the function defines the random walk). Even though they are not related to cycle finding algorithms described in this chapter in all cases, they give a good theoretical base on cycle detection.

4.3 Characteristics of Random Functions in Cycle Finding Algorithms

The good behavior of any cycle algorithm depends of the function used in producing the sequence of values. For a better overview, the behavior of random functions, sometimes also called random mapping, must be presented. A good survey of the random function is given in [26]. In the following, we will summarize the facts from this paper. It gives a nice survey, statistic and definitions for basic terms related to random functions.

Let h be a function from set S of n elements to itself. It can be observed as a directed graph G_f whose elements are from $\{1, ...n\}$ and edges are pairs $(x, f(x))$ for all $x \in \{1, ..., n\}$. As it was described earlier, starting from the node x_0 and applying the iteration a sequence $x_1 = h(x_0), x_2 = h(x_1), ...$ will be given. Then the node x_j which has the value equal to one of $x_0, x_1, x_2, ..., x_{j-1}$ must be reached. In the graphical representation, the path will be connected to the cycle. The length of the path is called a *tail length* and is usually denoted by $\lambda(x_0)$ or simply λ. The *cycle length* is then denoted by $\mu(x_0)$ or just simply μ. The rho-length denotes the length of the tail and the cycle together, i.e. $\rho(x_0) = \rho = \lambda + \mu$.

A functional graph is set of disjoined connected components. Each connected component is a set of trees connected to the same cycle. For instance, if there is a function $h(x) = x^2 + 3 \pmod{30}$ then the graphical representation is given as in the Figure 4.1. This functional graph consists of four connected components. Each of these connected components has a cycle of length 2.

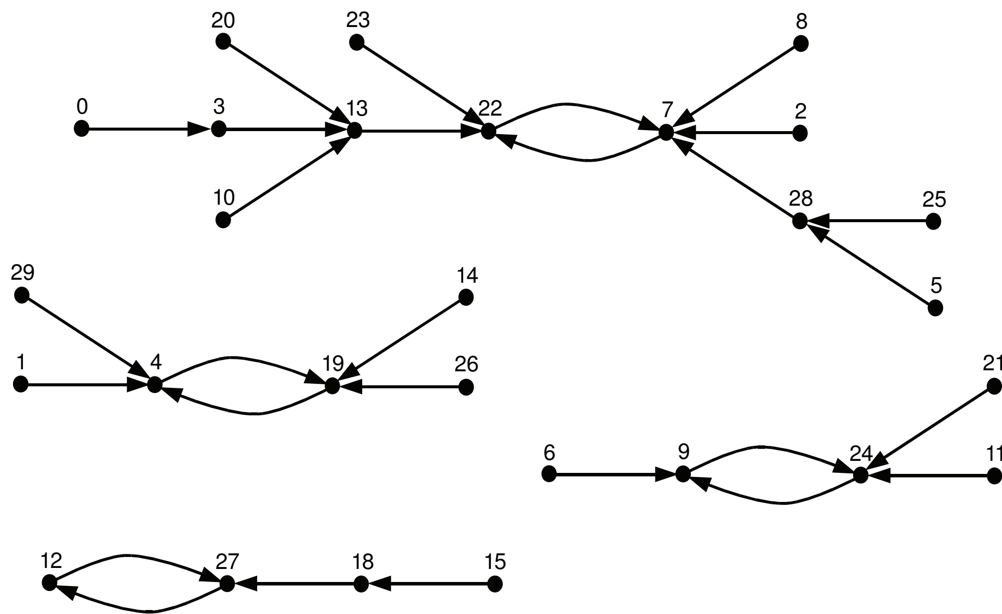


Figure 4.1: A functional graph associated to the function $h(x) = x^2 + 3 \pmod{30}$

For a such defined graph expected values of several parameters which are in interest of the random function are derived, i.e. the expectation of parameters of random function, when n has the asymptotic form $n \to \infty$. In general, parameters can be divided in two classes:

- *direct parameters* refer to properties depending on the graph itself (e.g. the number of connected components)
- *cumulative parameters* refer to properties depending on some random point in the graph (e.g. the expected tail length)

4.3.1 Directed Parameters

connected component is obtained by grouping the points that can be reached from one starting point and traveling through the graph in any direction. Asymptotically, the number of connected components in functional graph is $\frac{1}{2}\log(n)$.

terminal point is a node which does not have a preimage. That is, for any terminal node y a node x, such that $h(x) = y$, can not be found, or in other words, a node which has a property that $h^{-1}(y) = \emptyset$. Asymptotically, the number of terminal nodes in the graph is $\frac{n}{e}$.

image point is a node which is in the image of function h, i.e. nodes which have an image. Asymptotically, the number of image points in the graph is $(1 - e^{-1})n$. The image point is the opposite of the terminal point.

k-th iterated image point is a point y, if and only if, there exists a point x in the graph such that $y = h^k(x)$ where h^k is a k consecutive application of function h. It should be noted that if y is k-th iterated image point then (k-1)-th is also iterated image point. Image point is just a special case of k-th iterated image point. Asymptotically, the number of k-th iterated image points is $(1 - \tau_k)n$ where τ_k is defined recursively by $\tau_0 = 0$, $\tau_{k+1} = e^{-1+\tau_k}$.

4.3.2 Cumulative Parameters

tail length as it was defined earlier is a length of the path from the start point to the cycle obtained by iterating the function h. Asymptotically, for a random start point the average tail length is given by $\sqrt{n\pi/8}$.

cycle length is the number of nodes (or edges) in the cycle. An average cycle length is equal to the average tail length and it is $\sqrt{n\pi/8}$.

rho length is the length from the random start point to the first repetition. Thus, the average rho length is $\sqrt{n\pi/2}$.

tree size is a size of the tree, rooted on a cycle, which contains the start point. The average size of this tree is $n/3$.

connected component size is the size of a connected component that contains the start point. The average size of such connected component is $2n/3$. This implies that the large number of nodes in the functional graph is grouped in one single component size, usually called giant component.

predecessor's size is a number of nodes that are iterated preimage of the start point. That is, the tree rooted at the start point. The number of such nodes are $\sqrt{\pi n/8}$.

4.3.3 Extremal Parameters

longest cycle in the random function is the expectation of the maximum cycle length and it is $c_1\sqrt{n}$ where $c_1 \approx 0.78248$.

longest tail is the expectation of the maximum tail length in the random function and it is $c_2\sqrt{n}$ where $c_2 \approx 1.73746$.

longest path is the expectation of maximum rho length in the random function and it is $c_3\sqrt{n}$ where $c_3 \approx 2.4149$. It is interesting that $c_3 < c_2 + c_1 \approx 2.5199$. This inequality tells that, with non zero asymptotic probability, the longest cycle doesn't correspond to the longest tail.

giant component as it was said earlier, for a random start point, an average component size is $2n/3$ and the average tree size is $n/3$. The expected size of the longest tree is $d_1 n$ where $d_1 \approx 0.49$ and the expected size of the longest component is $d_2 n$ where $d_2 \approx 0.75782$.

4.4 Pollards Rho Integer Factorization

This method was originally presented in [54] by Pollard. This method aims at finding the factor of some integer n. The same idea was used in Floyd's and Brent's algorithm in order to find a cycle in the sequence. It is a probabilistic method and it does not guarantee success where the running time is not rigorous but it is very effective in practice. One of the biggest advantage is that it uses only a small and constant amount of memory.

Pollard's algorithm generates the sequence through a group or semigroup until the match is found. This sequence needs some additional properties. For any prime factor p of n, the sequence modulo p must be also defined by a recursion formula. This means that the function h must be selected carefully. It is a good practice to choose polynomial function for h. Most often, it is defined as:

$$h(x) = x^2 + c$$

where c is a constant.

For such a function the sequence h is defined by:

$$x_{i+1} = h(x_i) \ (mod\ n)$$

and let p be the nontrivial factor of n such that $gcd(p\ ,n/p) = 1$. Then, the sequence $\langle x_i \rangle$ corresponds to the sequence $\langle x'_i \rangle$ modulo p, where $x'_i = x_i\ (mod\ p)$. If the function h has only arithmetic operations (in our case, addition and squaring), it can be computed x'_{i+1} directly from x'_i. In this case, the sequence with modulo p behaves as the bigger sequence of modulo n. For this reason, the sequence $\langle x'_i \rangle$ return the same value as the sequence $\langle x_i \rangle$.

The number of steps before the sequence $\langle x'_i \rangle$ starts cycling is $O(\sqrt{p})$. Exceptionally good results are given if p is very small in comparison to n because the sequence $\langle x'_i \rangle$ in that case starts cycling much faster than $\langle x_i \rangle$. In other words, the sequence $\langle x'_i \rangle$ starts cycling when two elements from the sequence $\langle x_i \rangle$ have the same modulo p. In Figure 4.2 an example of factorizing the number 1387 is shown so that it has factors 19 and 73. A part (a) gives a sequence by producing the values $x_{i+1} = (x_i^2 - 1) \pmod{1387}$ starting from 2. Solid arrows indicated the sequence before the factor 19 was found. Dotted arrows indicate points whose values are not reached. Values in the gray cycle are values that are stored. The solution was found reaching the value 177 since $gcd(1387, 63 - 177) = 19$. A part (b) gives the same sequence modulo 19. The cycle was detected since both 63 and 177 from (a) are equivalent to 6 modulo 19. A part (c) gives the same sequence modulo 73.

Let λ' be the index of the first repeated value in the sequence $\langle x'_i \rangle$ and let μ' be the length of the cycle of the same sequence. Then λ' and μ' are the smallest number such that $x'_{\lambda'+i} = x'_{\lambda'+\mu'+i}$ where $i \geq 1$. Thus, $p|(x_{\lambda'+\mu'+i} - x_{\lambda'+i})$ and $gcd(x_{\lambda'+\mu'+i} - x_{\lambda'+i},\ n) > 1$. In Algorithm 5 is a pseudocode of Pollard's factoring algorithm.

In the code the parameters x with the index i were used but the algorithm doesn't need to remember the previous values and it works only with the most recent value of x. For that reason, the algorithm also works correctly if x_i is substituted with x. Once when y takes the value from x_k for $k \geq \mu'$, $y\ (mod\ p)$ stays in the cycle all the time. At some point, k will be big enough so that the whole loop of the cycle modulo p is made without changing the value of y. In the cycle modulo n will be detected x_i that takes the previous stored value of y modulo p with checking if $gcd(y - x_i,\ n)$ has a nontriviral solution.

4.5 Cycle Detection Using Meet In the Middle Attack

With minor changes, this attack can also be successfully applied on different cycle finding techniques. Before the explanation of how it can be employed, the short overview of the meet-in-the-middle attack in general needs to be given.

Meet-in-the-middle attack was firstly introduced in [21] by W. Diffie and M.

Algorithm 5 Pollard rho method

Input: n, c
Output: d

1: $i = 1$;
2: x_i = Random(0, $n - 1$);
3: $y = x_i$;
4: $k = 2$;
5: **while** true **do**
6: $i = i + 1$;
7: $x_i = (x_i + c)\ (mod\ n)$;
8: $d = gcd(y - x_i,\ n))$;
9: **if** $d \neq 1$ and $d \neq n$ **then**
10: **return** d;
11: **else if** $i = k$ **then**
12: $y = x_i$;
13: $k = 2k$;
14: **end if**
15: **end while**

Hellman. It is a special kind of an attack that can be applied on the function which has its inverse. With the birthday attack an adversary tries to find two distinct values that map to the same one. The meet-in-the-middle attack tries to find two values so that forward mapping of the function applied on the first value gives the same result as the inverse mapping of the function applied on the second value. That is, two values going forward and backward are trying to meet each other in the middle.

A very similar approach was explained in the Section 2.5.2. Unlike the birthday attack that tries to violate the collision resistance, the meet-it-the-middle attack that will be described here is mostly focused on the first preimage resistance property of the function. The evaluation of the function h is given by $y = h_s(x)$ and the function h is supposed to be invertible. The value s denotes an initial internal state or some other value involved in the process of evaluation. If it is supposed that the function h has a predefined initial state then the evaluation can be denoted by $y = h(x)$. After applying the function h on the parameter x, the state of the function is changed. This internal state of the function h uniquely defines the output value y and vice versa. That is, there is some bijective relationship between the output internal state and the value y. For the meter of simplicity, the output value y can be treated as the output state of the function h. For calculating the function h^{-1} the initial state has a very important role and it can not be predefined. So, the

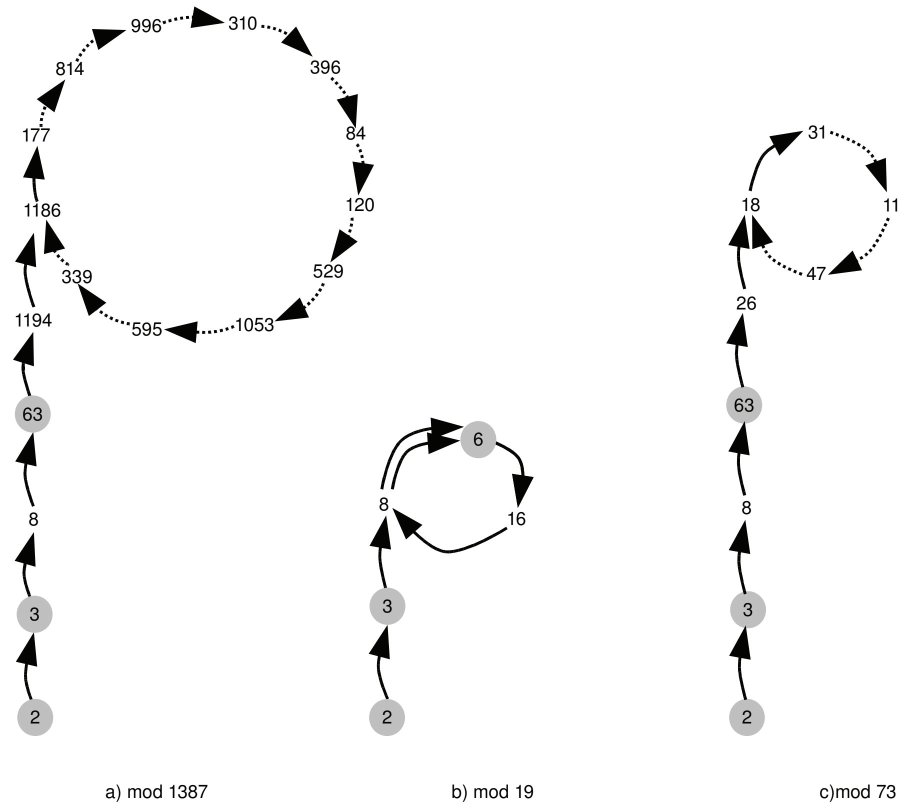


Figure 4.2: Pollard rho factorization of 1387

inverse of the function h is usually denoted as $x = h_s^{-1}(y)$. As for the function h, the function h^{-1} has its own output internal state that is in a one to one relationship with the value x and, also, for the matter of simplicity can be treated as the same value.

Employing a cycle finding algorithm to find a value x' such that $y = h(x')$ can be given in the following way: for a given value y and predefined initial state the iteration starts by choosing the random value x_0. A certain criteria must exist, dividing values into two disjoint sets, i.e. checks whether the first bit is zero or one. This criteria can be used to decide whether the function h or h^{-1} will be performed. For instance, if x_i starts with 0 then $x_{i+1} = h(x_i)$ will be calculated. If x_i starts with 1 then an inverse function $x_{i+1} = h_{x_i}^{-1}(y)$ will be performed. This way, values $x_0, x_1, x_2, ..., x_{i-1}, x_i, ..., x_{j-1}, x_j$ will be given where x_j is the first occurrence of the

value that has already been evaluated, for instance $x_i = x_j$. Without losing any generality, let suppose that the value x_i was gained by $x_i = h(x_{i-1})$ and the value x_j by $x_j = h^{-1}_{x_{j-1}}(y)$. That is, they are both obtained from the same function but from "different directions", h and h^{-1}. In that case the value $x' = x_{i-1}||x_{j-1}$ meets the equation $y = h(x')$. On the other hand, if x_i and x_j are both obtained from the same function h and both are from the "same direction" (i.e. both obtained from the function h or both obtained from the function h^{-1}), it will be impossible to conclude any predecessors of the value y. That is, if there is an equal chance in the sequence that the function h and h^{-1} will be performed then the probability that the value x' such that $y = h(x')$ will be found using this method is 1/2.

The explanation by itself has only a rough form. It includes that there is an enough space in the memory where all iterated values can be stored. The time and the memory complexity are the same as in the exhaustive search. Hopefully, this method can be integrated into any cycle finding algorithm in order to find preimage of the function h. The main difference is that this method does not guarantee that the preimage will be found. The pseudo code for the rough form is given by in an Algorithm 6:

4.6 Cycle Detection Methods

All methods described here are applied to any final set on which the iterated function is used to perform a random walk. They do not exploit the structure of the set where the iterated function is defined and can be used in more general case. Some methods go a little bit further and exploit some additional properties of the element. Such methods are value-dependent where the value characteristic of a single element is taken into account during the cycle detection. The price for that is using more memory. In the following part of this section some cycle detection methods with their basic characteristics will be presented.

4.6.1 Floyd's cycle finding algorithm

According to [40] for a periodic sequence $x_0, x_1, x_2, ...$ there exists an $i > 0$ such that $x_i = x_{2i}$. The smallest such i is in the interval $[\mu, \mu + \lambda]$. This fact is exploited in Floyd's cycle finding algorithm. This algorithm uses only a small and constant amount of memory with the penalty in the running time. It creates two sequences, x_i and x_{2i} for $i = 0, 1, 2, ...$ until the match $x_i = x_{2i}$ is found. It avoids to store the whole sequence x_i and remembers only two most recently evaluated values. This algorithm is described in 7:

If $\lambda \geq \mu$ then the collision will happen after λ iterations. If $\lambda < \mu$ the match will be found after $\lambda \left\lceil \frac{\mu}{\lambda} \right\rceil$ iterations which is somewhere between μ and 2μ [13].

Algorithm 6 Meet in the middle attack

Input: x_0, iterated function $h : G \to G$, for some finite set G, value y such that $y = h(x)$ for some x
Output: x' such that $h(x') = y$

```
1: create memory storage S
2: i = 0
3: x_0 = RandomValue()
4: while true do
5:    if found x_j in S such that x_j = x_i then
6:       if first bit of x_{i-1} ≠ first bit of x_{j-1} then
7:          if first bit of x_{i-1} = 0 then
8:             x' = x_{i-1}||x_{j-1}
9:          else
10:            x' = x_{j-1}||x_{i-1}
11:         end if
12:         break;
13:      else
14:         return null
15:      end if
16:   else
17:      put x_i in S
18:      if first bit of x_i = 0 then
19:         x_{i+1} = h(x_i)
20:      else
21:         x_{i+1} = h^{-1}_{x_i}(y)
22:      end if
23:      i = i + 1
24:   end if
25: end while
26: return x';
```

Algorithm 7 Floyd's algorithm

Input: x_0, iterated function $h : G \to G$, for some finite set G
Output: i such that $h^i(x) = h^{2i}(x)$

1: $x = h(x_0)$;
2: $y = h(x) = h^2(x_0)$;
3: $i = 1$;
4: **while** $x \neq y$ **do**
5: $\quad i = i + 1$;
6: $\quad x = h(x)$;
7: $\quad y = h^2(y)$;
8: **end while**

For a random function h, the length of μ and λ is the same and it is $\sqrt{n\pi/8}$. In each iteration there is one comparison and three evaluations. This means that the expected number of iterations is $\frac{3}{2}\sqrt{n\pi/8}$ and the expected number of evaluations is $\frac{9}{2}\sqrt{n\pi/8}$.

4.6.2 Brent's cycle finding algorithm

One disadvantage of Floyd's algorithm is that it uses three evaluations in each iteration of the function h. Brent's algorithm [10] modifies Floyd's algorithm which solves this problem. It requires only the copy of the original sequence that is used as the mark to detect the cycle. The principle is quite similar to Pollards rho factoring algorithm described in Section 4.4. It also uses two pointers and it tries to find the smallest power of two 2^k bigger than μ and λ. It uses a mark point, witch holds the value x_{2^k-1}. In each new iteration it is checked if the mark point is equal to x_i. If the index i equals a power of two minus one, then the mark point is associated to this value. This process is repeated until the match is found. The whole algorithm is defined in Algorithm 8:

Expected number of iterations after the cycle is detected using Brent's algorithm is $\approx 1.9828\sqrt{n}$. This number is two times higher than the number of iterations using the Floyd's algorithm, but uses less number of evaluations. Floyd's algorithm evaluates the function h three times per iteration and Brent's uses only once. With Brent's algorithm runtime is reduced to about one third in evaluations. The price for this reduction is the much higher number of comparisons. In some cases it can be a serious drawback. There is a variant of Brent's algorithm which gives a better performance and has less number of comparisons. It is proven in [12] that if i is the smallest index such that $x_i = x_{l(i)-1}$ then i satisfies $\frac{3}{2}l(i) \leq i \leq 2l(i)$, where $l(i)$

Algorithm 8 Brents's algorithm

Input: x_0, iterated function $h : G \rightarrow G$, for some finite set G
Output: i and j such that $h^i(x_0) = h^j(x_0)$

```
1: m = x_0;
2: x = x_0;
3: i = 0;
4: l = 1;
5: while true do
6:    i = i + 1;
7:    x = h(x);
8:    if x = m then
9:       break;
10:   end if
11:   if i ≥ (2l − 1) then
12:      m = x;
13:      l = 2l;
14:   end if
15: end while
16: return (i, j = l − 1);
```

is the larger power of two contained in the current index i. For instance, it can be defined as $l(i) = 2^{\lfloor \lg i \rfloor}$. Using this fact an improved version of Brent's algorithms can be constructed and it is described in an Algorithm 9.

There are also other variants of Brent's algorithm. These variants are described in [66] and [71]. The idea is to make p different units, $U_1, U_2, ..., U_p$. At the beginning, put x_0 into each of them. After x_i is computed it is checked whether this value is already in one of p units. If it is already in one of them, a match is found. Suppose that x_j is stored in U_1. This process is repeated until $i \geq \alpha j$ for some fixed $\alpha \geq 1$. Then the content of units is shifted from $U_k + 1$ to U_k, for $k = 1, ..., p-1$ and the value of x_i is stored in U_p. In [71] Taske experimentally showed that the best result is for α between 3 and 4, and for $p = 8$. For $\alpha = 3$ and $p = 8$ he showed that expected number of iterations until the match is found of about 1.13 times worse than the best possible solution. That is, it must be about $1.412\sqrt{n}$ iterations until a match is detected.

Algorithm 9 Improved Brents's algorithm

Input: x_0, iterated function $h : G \to G$, for some finite set G
Output: i and j such that $h^i(x_0) = h^j(x_0)$

```
1:  m = x_0;
2:  x = x_0;
3:  i = 0;
4:  l = 1;
5:  while true do
6:     i = i + 1;
7:     x = h(x);
8:     if x = m then
9:        break;
10:    end if
11:    if i ≥ (2l − 1) then
12:       m = x;
13:       l = 2l;
14:       while i < (3/2 l − 1) do
15:          x = h(x);
16:          i = i + 1;
17:       end while
18:    end if
19: end while
20: return (i, j = l − 1);
```

4.6.3 Sedgewick, et al.'s algorithm

This algorithm is described in [67] and it attempts to optimize the worst case scenario, using the fixed amount of memory. It uses a table of predefined size m. Another input parameter is g. At the beginning the parameter d takes the value 1. In each iteration i, if $i \ (mod \ dg) < d$, we try to locate the value x_i in the table. If i is multiple of d, the pair (x_i, i) is stored in the table. If the table is full, the parameter d is doubled, and all values (x_j, j) are removed where j is not a multiple of d anymore. The assumption is that the table asymptotically reduce the worst case search time, for instance, a hashing table or balanced tree.

Let t_s denote the time needed to perform the search in the table and let t_h denote the time needed to evaluate the iterated function h. Then the parameter g can be defined as the function of t_s, t_h and m such that algorithm running time is $t_h(\mu + \lambda)(1 + O(\sqrt{t_s/mt_h}))$.

4.6.4 Gosper's algorithm

This algorithm was described in [42] and it is very useful, especially when a former argument in the function h is not easy (or not possible) to substitute. For instance, if the iterated function h is a pseudo random number generator that takes parameter from the some iterative "black box". This algorithm guarantees that the repetition will be detected before the third occurrence of any element.

Let x_0 be the initial value and L be the maximum length of a cycle. Let also $m = \lceil \lg L \rceil$ be the size of the table S where last m iteration values will be stored and i will be the number of iterations of function h. In each step i it is checked if the value $h^i(x_0)$ is equal with some of the first k values in the table where $s = \lceil \ln i \rceil$ (s also can be observed as the number of bits necessary to represent i). If none of the values matches, increase i for 1 and then store $h^i(x_0)$ into $S[r]$ where r is a number of zero bits at the end of the binary representation of i. If the match is found at position e then the loop length is 1 more than $e + 2$ bits of $i - 2^{e+1}$. We should be very carefully here due to the fact that if L is too small the algorithm will not detect the loop. This algorithm has advantages in comparison to some other algorithms because, without additional computation it gives us a correct length of the loop.

4.6.5 Time Memory Trade-Off Cycle Algorithms

All algorithms described above do not use the full potential of the output value property. They create values of the sequence in a restrictive way, performing only the equality test or using as the input parameter for the function h. The question is whether the property of values can be used to improve the cycle finding algorithm.

Algorithm 10 Gosper's algorithm

Input: x_0, iterated function $h : G \rightarrow G$, for some finite set G, L - the largest possible period of the sequence

Output: i and j such that $h^i(x_0) = h^j(x_0)$

```
1: m = ⌈lg L⌉;
2: create table S of size m
3: S[0] = x_0;
4: i = 1;
5: z = h(x_0);
6: while (i ≤ |G|) do
7:    s = ⌈ln i⌉
8:    for k = 0 to s − 1 do
9:       if z = S[k] then
10:         t = 1 + ((i − 2^{e+1}) mod 2^{e+2})
11:         return (i, j = i + t)
12:      else
13:         i = i + 1;
14:         z = h(z);
15:         r is number of last binary zeros of i
16:         S[r] = z;
17:      end if
18:   end for
19: end while
20: return false;
```

The answer is 'yes'. Additional properties can be very helpful and can be used to establish a completely new approach of the problem.

Algorithms that will be presented in addition are distinguished point and Nivasch's method.

Nivasch's Cycle Algorithm

Nivasch proposed an algorithm in [51] witch is one of the most efficient known cycle detection algorithm on a single machine. Worst-case scenario of most other algorithms for detecting the cycle in the path has much better performance [13]. It ensures that this process will be terminated somewhere between x_λ and $x_{\lambda+2\mu-1}$, i.e., during the second cycle sequence iteration. This can be very useful when the length of the cycle is small.

The idea is to construct a stack of already computed values in the way which ensures cycle detection. The algorithm requires some ordering relation to be defined over the whole set. Number of elements in the stack should be small in order to ensure a good performance. The stack must guarantee that the smallest element in the cycle will always be stored for detection during the second loop. In order to achieve that, the stack is configured in the following way: initially, the stack is empty. At iteration i, remove all elements (x_j, j) from the stack where $x_i < x_j$. If x_j is found such that $x_i = x_j$ then the process is terminated. From here it can be directly calculated that the length of the cycle is $i - j$. Otherwise, put (x_i, i) on the top of the stack and continue calculating the next iteration, $x_{i+1} = h(x_i)$.

The stack configured in this way will detect the cycle in the second loop. During the first loop, a minimal element of the cycle will be stored on the stack. This element will not be removed anymore since this element is the minimal. During the second loop when the path comes to the minimal element of the cycle, an algorithm will recognize this minimal element and the cycle will be detected.

Flajolet in [26] explained that the expectation tail length (λ) is $\sqrt{\pi n/8}$. Also expected length for the cycle (μ) is $\sqrt{\pi n/8}$ and for rho length ($\lambda + \mu$) is $\sqrt{\pi n/2}$. If we assume that the function h is a random mapping, the expected number of evaluations until the minimum in the cycle occurs is $\mu + \lambda(1 + \frac{1}{2}) = \frac{5}{2}\sqrt{\pi n/8}$. In [51] it was shown that the expected size of the stack in the step n is $\ln(n)$ and stays below $O(e \ln(n))$. In other words, the stack size stays small and the memory requirement is not too expensive unless in very rare cases. This means that the size of the stack can be estimated before the search starts and therefore can be implemented as an array with modification of resizing in very special cases, when number of entries overflows the size of the stack. Therefore, there is no need to create or delete memory allocations of the stack; the new entry will only overwrite the old value and only the stack size will be kept. Note that since elements in the stack have relation ordering, the search can be realized with binary search

algorithm. All algorithms above have a running time proportional to μ and λ. When the iteration of the function comes into the cycle, detecting the cycle does not depend on the λ anymore. Thus, this algorithm can be useful when the cycle is small comparing to tail [51]. Also, it can directly output the cycle length without additional computation. For some applications where this information is important it can save us any additional efforts. In Algorithm 11 is given a pseudocode for a Nivasch's algorithm.

Algorithm 11 Nivasch's algorithm

Input: x_0, iterated function $h : G \to G$, for some finite set G
Output: i and j such that $h^i(x_0) = h^j(x_0)$

```
1: create stacks s which contains pairs (element, index)
2: set index j to −1
3: x = x0;
4: while true do
5:    if j ≥ 0 then
6:       find a smallest t ≤ j such that element(s, t) ≥ x
7:    else
8:       t = −1;
9:    end if
10:   if (t ≠ −1) and element(s, t) = x then
11:      break;
12:   end if
13:   i = i + 1;
14:   j = t + 1;
15:   if j > size(s) then
16:      resize s;
17:   end if
18:   put (x, i) into s;
19:   x = h(x);
20: end while
21: return (i, j = index(s, t));
```

Multi-stack algorithm In the second loop, the algorithm stops at a uniformly random point in the cycle. Thus, it will be very important to reduce the number of steps in the second loop, especially if the cycle is extremely large. Multiple stacks in Nivasch's algorithm give us a solution for this problem. A method increases a

probability that algorithm stops closer to the beginning of the second loop. The memory is increased but only by a constant factor and it has no effect on time penalty per step.

In [51], the following procedure was proposed: for some integer k, the whole set is divided into k disjoint classes. It is preferred that the cardinality of classes be the same. For instance, the set can be divided according to the values of some bits for a certain internal representation of elements. For each class a different stack is created. Each new element is classified according to internal representation and put into related stack, i.e., for each step i we assign the class the element x_i belongs to and then (x_i, i) goes into a corresponding stack. This algorithm is also known as *multi-stack* algorithm. After determining which class particular element belongs to, the algorithm works on a single stack and this is the reason why the run time per step does not change drastically. This algorithm works, since each stack has its own minimal element after the first loop. The algorithm stops when, during the second loop, one of the stacks detects a match. A pseudocode for Nivasch's multi-stack algorithm is shown in Algorithm 12.

If an assumption that the function h is random is taken into consideration, i.e., that all elements are distributed uniformly and independently, then the average running time until the cycle is detected is $\lambda + \mu(1 + 1/(k+1))$. Also, for $k \ll \mu + \lambda$, the expected amount of memory is $O(k \ln(\mu + \lambda))$ which is k times more than when only one stack is used. A multi-stack algorithm can be a good solution when this algorithm is performed on a single processor.

Distinguished Point Algorithm

Before explaining how distinguished point algorithm can be employed in detecting the cycle, we introduce Hellman's method [30] (sometimes also called Hellman's time-memory tradeoff). It was originally applied on a block cipher for a given plaintext, $h(x) = E_x(P)$. The method consists of two parts: *precomputation phase* (also called *off-line phase*) and *on-line phase*. The idea is to precompute as much pairs $(x, h(x))$ for a given plaintext P as possible. To reduce the memory storage this is organized in the chain of fixed length.

In the off-line phase, m different keys are chosen. They are used as *start points* (SP) of the chains. Each chain uses the start point as a key to compute the cipertext C. This cipertext is then used to generate the new key in order to compute the new cipertext and so on. Sometimes, it is necessary to use a *reduction function* R to fit the length of the key. Then, the process of creating the chain looks like the following: the first resulting key is given using the SP and the reduction function R, $k_1 = Rh(SP)$. Next resulting key is given by $k_2 = Rh(k_1)$ and so on. The composition of h and R can be denoted as the new function F and it is called *step-*

Algorithm 12 Nivasch's multi-stack algorithm

Input: x_0, iterated function $h : G \to G$, for some finite set G, number of stacks K, function $P : G \to [0, ..., K-1]$

Output: i and j such that $h^i(x_0) = h^j(x_0)$

```
1: create stacks s[0], ..., s[K − 1] which contains pairs (element, index)
2: set index i_0, ..., i_{K−1} to −1
3: x = x_0;
4: while true do
5:    k = P(x);
6:    if i_k ≥ 0 then
7:       find a smallest t ≤ i_k such that element(s[k], t) ≥ x
8:    else
9:       t = −1;
10:   end if
11:   if (t ≠ −1) and element(s[k], t) = x then
12:      break;
13:   end if
14:   i = i + 1;
15:   i_k = t + 1;
16:   if i_k > size(s[k])  then
17:      resize s[k];
18:   end if
19:   put (x, i) into s[k];
20:   x = h(x);
21: end while
22: return  (i, j =index(s[k], t));
```

function F [29]. After t computations of step-function, the chain stops and the *end point* (EP) is taken. For such computed end point, the pair (SP, EP) that uniquely identify the chain is stored. The problem can be if two or more chains collide or if the chain falls in a loop. The biggest reason for the first problem is the surjective nature of a reduction function R that compresses the output values. The reduction function can have two or more different values which are compressed to the same output. As much as the number of chains arise, the probability that the collide may occur is greater. Hellman solved this problem by using multiple tables where each of them has a different reduction function.

In the on-line phase for the given cipertext C, one tries to figure out the corresponding key. First, the reduction function on the cipertext, $R(C)$, is applied and checked if this value is the same as a certain end point of the chain. If not, the next value is computed, $F(R(C))$, checked if it has the same value as a certain end point of the chain, and so on. This is repeated at most t times. If the match is found after i times then the key from the corresponding SP is reconstructed. Then, the candidate for the key is $k = F^{t-i-1}(SP)$. Note that it does not necessarily mean that this key is the right one. Therefore, the equation $h(k) = C$ must be verified. If the verification is positive, the key is found. Otherwise, another table must be used and the whole procedure repeated.

Distinguished point is a variant of the Hellman's method. It was introduced in 1982 by Rivest [20]. In the on-line phase of Hellman's method the majority of time is spent on searching the corresponding value in the table. Rivest was able to reduce the table access. He introduced the so called distinguished point that has very simply criteria for recognition (e.g. first n bits are zeros). He put this values at the end of the chain. This means that in the off-line phase the chain is computed until some distinguished point or the chain reaches the maximum length of t. If the chain reaches the maximum length, this chain is discarded. In the on-line phase, the table does not need to be accessed after each step. Only when some distinguished point is occurred we must check whether some stored end points match in the table.

The idea of distinguished point can also be used in detecting the cycle. Lets now assume again that our iterated function is given by $x_{i+1} = h(x_i)$. First, the distinguished criteria must be defined to be able to differ just particular elements from the path. These elements are distinguished points. Secondly, enough memory storage must be ensured. The amount of the memory storage can be precomputed if the expected number of iterations is known before a collision occurs and probability that a distinguished point is in the path. If it is necessary, m memory units are used to store value of one element. Let n be the number of iterations until collision occurs and Θ the probability that the value has a distinguished property. Then the expected size of the memory storage is $\frac{mn}{\Theta}$. The algorithm works as follows: at iteration i is checked if x_i is distinguished point. If not, proceed with the calculation of the new iterated value. On the other hand, if it is a distinguished point then an

algorithm tries to find this value in the memory storage. If it is found, the cycle is detected and the algorithm stops. If it is not there, put this value in the memory storage and continue calculating the new iterated value. The pseudo-code for cycle detection using distinguished point is given in an Algorithm 13:

Algorithm 13 Cycle Detection Using Distinguished Point

Input: x_0, iterated function $h : G \rightarrow G$, for some finite set G
Output: i and j such that $h^i(x_0) = h^j(x_0)$

```
1: create memory storage S
2: x = x0;
3: i = 0;
4: while true do
5:    if x is distinguished point then
6:       if found (x, j) in S for some arbitrary j then
7:          break;
8:       else
9:          put (x, i) in S
10:      end if
11:   end if
12:   i = i + 1;
13: end while
14: return (i, j);
```

4.7 Summary

In this chapter different cycle finding algorithms were described. In general, each of them gives us some advantage in comparision to another algorithm in some aspects. In other words, there is no universal algorithm that fits the best in any situation. It all depends of the environment in which it runs. For instance, if we expect that the length of the cycle can be small (iterated function doesn't behave as as random mapping), then it is not a good idea to use the distinguished point method. On the other hand, if an iterated function behaves like a random mapping then Nivasch's stack based method as well as distinguished point method can be considered. It must also be taken into account that these algorithms are value-dependent and can not be used in every situation. All these facts give us conclusion that before applying any cycle finding algorithms all advantages and disadvantages they provide must be considered.

Time memory trade-off algorithms require an additional amount of memory but provide faster methods for cycle detection. In Nivasch's algorithm the amount of required memory stays below $O(e \ln(n))$ but the expected number is $\ln(n)$. It also gives the opportunity to estimate the size of the stack before the start of the algorithm. Even if the size of this stack is relatively small it has to interact every time when the new value is produced. On the other hand, the distinguished point method must be able to allocate $\frac{n}{\Theta}$ values where n is the number of iterations until collision occurs and Θ the probability that the value has a distinguished property. This is much more than Nivash's method but the distinguished point method for detecting the cycle avoids memory access after each new value and stores only with the particular properties.

Detecting the cycle has a central role in the process of finding a collision. It is not just important how fast the cycle is detected. What kind of information particular method can provide must also be taken into consideration and this is the issue that next chapter is dealing with. In this sense, detecting the cycle is just the first step in the process of finding the collision.

Chapter 5

Collisions in Cycle Algorithms

5.1 Introduction

Cycle finding algorithm can be employed for finding a cycle but not for finding a collision. Once, when the point in the cycle has been found another algorithm must be used to find a collision. Which algorithm will be used depends on the nature of initial method for finding the point in the cycle. This chapter considers the problem of finding the collision in the cycle graph.

In the previous chapter several algorithms show how the cycle in the graph can be detected. All these algorithms reveal the cycle in the graph but none of these tell anything about the collision. In other words, once the cycle is detected two points in the graph e_1 and e_2 must be found such that $h(e_1) = h(e_2) = c$. If the start point is not in the cycle, then there must exist two points e_1 and $e_2(\neq e_1)$ making it possible to find a collision. Otherwise, when the start point is a part of the cycle then it is not possible to find a collision directly but it is likely to find a predecessor. Using different mask functions more predecessors for one initial point can be found. If it can be ensured that by using several mask functions the path in the graph stays in the cycle all the time, it is very likely that all predecessors of the initial points are different.

The goal of collision search is to take a function h and find two distinct input that produce the same output. The assumption is that the function h behaves like a random mapping. The obvious method for finding a collision is to produce $x_i, i = 1, 2, \ldots$ and to check $h(x_i)$ for collision. Let n be the range of output for function h. Then the probability that the collision is not found after k iterations is

$$(1 - 1/n)(1 - 2/n)\ldots(1 - k/n) \approx e^{-k^2/(2n)}$$

for a large n and k. The expected number of iterations before the collision occurs is $\sqrt{\pi n/2}$ [26]. If an assumption was taken that the new value of $h(x_i)$ is stored

in constant time then this method finds a collision in $O(\sqrt{n})$ time and has $O(\sqrt{n})$ storage access [72]. From such an algorithm the length of the tail and the cycle can be read directly and also detecting the collision is instantly.

Due to the large complexity of storage access this method is not accessible. There is a imbalance between the cost of storage and computations. While the computation of 2^{40} iterations is relatively cheap, the storage of at least 2^{40} bytes = 1TB is a still expensive. It is important to be aware that birthday attack can be run memoryless with only a slight increase in the number of evaluations of hash functions. From such an approach to the problem the length of the cycle and the tail sometimes can not be read directly. The same goes for finding the collision and thus an additional effort must be spend to get these information.

Another important characteristic of an attack is whether it can be run parallel or whether it requires a sequential iteration. By sequential iteration, the speed is limited by the frequency of CPU, FPGA or an integrated circuit [47]. Using parallelization the speed of finding a collision can be sufficiently increased and the only limitation is the attacker's budget.

The remainder of this chapter is organized as follows: it describes the basis idea of the finding collision in the cycle. It also gets more details about Nivasch's stack-based and distinguished point algorithms.

5.2 Objectives

Finding the cycle in the graph does not mean that the collision is found. After the cycle is detected additional effort must be done to find such a collision. All cycle-finding algorithms only reveals the point located in the cycle. For instance, Nivasch stack base algorithm described in Section 4.6.5 finds the minimum element in the cycle. This element by itself tells us nothing about the collision since the cycle minimum x_{min} appears in the random position in the cycle. When one such point is found in the cycle a way to find point e_1 and $e_2(\neq e_1)$ must be obtained such that $h(e_1) = h(e_2)$. Different cycle finding algorithms have different properties and carry different information. Depending on the nature of these information we can define the strategy how to find the collision for the particular cycle finding algorithm. In this chapter a set of solutions to reach a collision using cycle finding algorithms is presented.

5.3 Finding Collisions Using Memoryless Algorithms

There is a general concept for finding the collision. If there are 2^n different outputs (e.g. n bits outputs) of a hash function we expect to find the collision until approximately 2^{n-1} comparations between distinct pairs. Distinct pairs can be chosen in different ways. One can compute a hash value and then produce a lot of different inputs for a hash function and compare all results with the chosen one. In this case 2^{n-1} hash values are expected to be produced until the collision is found (preimage resistance). Birthday attack is nothing more than producing 2^{n-1} distinct pairs. The only difference is that these pairs are chosen in another way. Namely, for a chosen different k, if each element is compared to all other elements then it has to be $k(k-1)/2 \approx k^2/2$ distinct pairs which are compared. In our case for $2^{n/2}$ different elements it has $\frac{(2^{n/2})^2}{2} = 2^{n-1}$ comparations. So, it can be said that if the collision has to be found, someone needs to find 2^{n-1} distinct pairs but how these pairs will be chosen is up to implementation.

The problem of collision search can arise when domain has less elements than codomain. That is, the number of distinct input values is smaller than number of distinct output values. Here, map function g which maps output value can be used. Suppose that we have a hash function with the output of n bits. The output of the hash function depends on the internal state c and input message that has n_k bits. This means $y = h(c, x)$, where y is the output message, x is the input message and c is the current state.

Let the internal state c be fixed for the first moment. An iterated function can be created:

$$x_{i+1} = g(h(c, x_i))$$

with the fixed initial value x_0, where the function g maps n bits message to n_k bits message in deterministic way. Eventually two distinct messages x_l and $x_m (\neq x_l)$ must be found such that $g(h(c, x_l)) = g(h(c, x_m))$. This is called the *simple collision*. If x_l and x_m are such that they also satisfy the equation $h(c, x_l) = h(c, x_m)$ it is called the *real collision*[1]. It is obvious that simple collision does not imply real collision. If we found only the simple collision and not the real collision we change the initial state c randomly and repeat the whole process again. We do this until the real collision is found. This process is described using pseudo-language in an Algorithm 14:

All algorithms can be used to find the length of the cycle μ but neither of them tells us for sure the length of the tail λ. If the goal is to find the collision, this

[1] It should not be mixed with pseudo collision or pseudo key explained in sections 2.5 and 2.5.1. To avoid this ambiguity new terms simple collision and real collision are introduced

Algorithm 14 Finding Collisions Using Memoryless Algorithms

Input: x_0

Output: x_m , x_l and c such that $h(c, x_m) = h(c, x_l)$

```
1: repeat
2:    c=random_value();
3:    m=0;
4:    empty(table);
5:    repeat
6:       put x_m into table;
7:       m = m + 1;
8:       x_m = g(h(c, x_{m-1}))
9:    until found x_l in table such x_l = x_m; {found simple collision}
10: until h(c, x_{l_1}) = h(c, x_{m_1}) {found real collision}
11: return x_{m_1}, x_{l_1}, c
```

information is very important. When the length of the tail and the length of the cycle is known then

$$h(x_{\lambda-1}) = x_\lambda = x_{\lambda+\mu} = h(x_{\lambda+\mu-1}) \text{ where } x_{\lambda-1} \neq x_{\lambda+\mu-1}$$

and that is the only place where a collision can be found.

Using one of the algorithms (e.g. Brent's or Floyd's) the point x_t that lies in the cycle can be detected. From here, it can be concluded that $\lambda < t$. The length of the tail can be precisely determined in several different ways. In the following part two approaches also described in [32] will be discussed.

Dichotomy search is a method where collision is guessed dividing each time in two separated classes. If μ is known it is possible to see whether for some i x_i belongs to tail testing $x_i = x_{i+\mu}$. If this equation is satisfied then x_i belongs to the cycle and $i \geq \lambda$. In that case less than i times must be searched for the index. Otherwise, $i > \lambda$ and the search continues for some index greater than i. Computing the sequence up to $i + \lambda$ requires $O(\lambda + \mu)$ computation. Thus, finding the collision using dichotomy search has complexity $O((\lambda + \mu) \log(\lambda + \mu))$. This complexity is nothing better than a generic birthday method based on quick sort algorithm.

Direct search solves one of the major problems in dichotomy search where some overlapping parts is needed to compute several times. To avoid this drawback, the following approach can be used: first computes x_μ and then computes in parallel two sequences. One sequence starts from x_0 and another from x_μ. In each step the equation $x_i = x_{\mu+i}$ is checked for $i = 0, 1, 2, ...$ When the

equation is satisfied for the first time then $i = \lambda$. Unless $\lambda = 0$, the collision is recovered in $O(\lambda + \mu)$ steps.

A direct search has better characteristics than a dichotomy search. It can be even speed up if the algorithm uses some additional information that can be exploited. An example for this is the distinguished point algorithm (see subsection 4.6.5) that gives the sequence from x_0 a possibility to avoid overlapping using the first distinguished point before the collision. This algorithm stores distinguished points in order to detect the cycle. After some period distinguished points are reached for the second time and the cycle is detected. All stored distinguished points give the possibility to allocate two distinguished points, one in the cycle and one in the tail, that precede the collision point. This additional information drastically reduces the search for the collision.

5.3.1 Collision in Brent's Algorithm

The main purpose of Brent's algorithm as any other cycle finding algorithm is to detect the cycle. That is, during the design of the algorithm finding the collision was not in mind. Some algorithms give additional information like the length of the tail or the cycle. Brent's algorithm does not give much additional information by itself. So, the algorithm must be adjusted in order to find a collision. In Section 4.6.2 the basic concept of Brent's cycle finding algorithm was described and in the following will be presented how the collision can be found.

Once the trial comes into the cycle it stays in it all the time. Iterated function creates points $x_0, x_1, x_2, \ldots$. Among all points that Brent's algorithm creates $(x_0, x_1, x_2, \ldots)$ mark points $x_0, x_1, x_3, x_7, \ldots, x_{2^k-1}, \ldots$ will be distinguished. At one point, the new mark point x_{2^k-1} will be created. If the size of the cycle is not bigger than 2^{k+1} then in the next loop the mark point will be recognized and the cycle will be detected. If the cycle is bigger than 2^{k+1} then the new point $x_{2^{k+1}-1}$ will be marked and the trial continues. This will be repeated until the cycle becomes larger than 2^{n+1} for some $n > k$. For such n the trial will make the whole loop without creating a new mark point. In this case a cycle is detected.

The estimation of the cycle length is important. To achieve this we must track the number of iterations. Let us denote the number of iterations by i when the point x_{2^k-1} is reached for the second time. Since the value 2^k is known then the cycle length is

$$\mu = i - 2^k - 1.$$

Suppose that the cycle is detected and the last mark point is x_{2^k-1}. In order to increase the performance in the process of finding the collision we distinguish two situations: $\mu \leq 2^{k-1} - 1$ and $\mu > 2^{k-1} - 1$. In practice, the second case is the most

common. The process of finding the collision when $\mu \leq x_{2^{k-1}-1}$ can be seen as the special case of the second one when the performance of the searching algorithm can be increased.

In the first case an algorithm will mark only one point in the cycle and it is x_{2^k-1}. This means that the previous mark point $x_{2^{k-1}-1}$ is out of the cycle and lies somewhere on the tail. Two points must be found in order to allocate, one from the tail and one from the cycle, that are equally distant from the collision. One point is $x_{2^{k-1}-1}$ that belongs to the tail. To allocate the point in the cycle, starting from the mark point, it must be stepped forward $\mu - (2^{k-1} \bmod \mu)$ times. Stepping together from these two points they must reach the same point and we can say that the collision is found. Note that this scenario is rare and it has a small influence in the general complexity of collision searching algorithm.

In another case when $\mu > 2^{k-1} - 1$ it can not be said for sure that some of previous mark points are not in the cycle. Thus, the search for the collision can not begin from $x_{2^{k-1}-1}$ but from the start point. Note that the start point can be in the cycle and finding the collision is not possible. Similar as in the previous case two points must be recognized in order to find a collision. The first one is x_0 and the second one will be found from the last mark point. To allocate the second point $\mu - (2^k \bmod \mu)$ times from the mark point must be stepped. Stepping together from these two points they must eventually meet the collision if it exists. If we take that the expected value for $2^k \bmod \mu$ is $\mu/2$ then by using one processor the expected work for finding the collision is

$$\tfrac{5}{2}\sqrt{\pi n/2}.$$

Using two processors the whole process can be made faster. The expected work required in this case is:

$$\sqrt{2\pi n}.$$

5.3.2 Finding Collision Using Nivasch's Stack-Based Algorithm

In the Section 4.6.5 Niasch's algorithm was described. Expected number of iterations until the collision has occurred is $2^{k/2}$ where k is cardinality of the range of the function h. Here μ - the length of the cycle and λ - the length tail must be distinguished. It is well known that $2^{k/2} = \mu + \lambda$. Assume that the value of $h(x_i)$ is stored in the sorted list. When the new value of (x_i) is evaluated it is compared with the elements in sorted list starting from the maximum. The new value is put on the stack and all elements from the stack that have the value greater than the new one are deleted.

Lets denote x_0 the start point and let $x_i = h(x_{i-1})$. To be able to find a collision after the minimum in the cycle is found the number of iterations must be tracked. That is, the pair (x_i, i) should be put on the stack where i is the number of iterations from the start point x_0 to x_i. The iteration of the function h will be repeated j times when it turns out that $x_i = x_j$. Then the calculation of the cycle length $\mu = j - i$ is easy. Also, it is already known that i is the number of steps until the first time the minimum in the cycle is detected. If $i > \mu$ we start stepping from the beginning (x_0) otherwise we start from the x_i. This process will be repeated $|i - \mu|$ times. This part must be done sequentially. Then stepping two trials forward together until they both hit the same point. This part can be done with parallel processors. The goal is to find the points a and b such that $h(a) = h(b)$, but $a \neq b$ is required. The work required for a single processor to find a collision is

$$2(\lambda + \mu) = \sqrt{2\pi n}$$

which give us complexity $O(\sqrt{n})$. The speed of finding the collision can be slightly increased using two parallel processors. It can be used in the last phase when two trials go together until they reach the same point. In this case the expected work for finding the collision is

$$\lambda + \mu(1 + \tfrac{1}{2}) + \lambda = 2\lambda + \tfrac{3}{2}\mu = \tfrac{7}{4}\sqrt{\pi n/2}$$

steps. This algorithm also guarantees that the minimum in the cycle will be found somewhere between $\mu + \lambda$ and $\lambda + 2\mu - 1$ iterations, i.e. during the second iteration through the cycle. This can be very useful if the length of the cycle (μ) is small comparing to the tail (λ).

Using parallel processors is not always a good idea. Most of the work must be done sequentially and only the last part when two trials go together can be accelerated by using two processors. Also two processes must be synchronized after each step. This means that in average $\frac{1}{4}\sqrt{\pi n/2}$ iterations of the function h can be saved. The main problem here is that one processor will wait until the another one finds the minimum of the cycle. Two processors save 12.5% time comparing to case when only one processor is used. This also means that 85.71% of time the second processor will not do anything.

In the Section 4.6.5 partitioning technique was described as well as how the cycle can be detected earlier if more stacks are used. But what does it really mean for finding a collision? Suppose that there are k stacks in which the values of iterations are stored. This leads to the fact that there is a great chance that the cycle is detected much earlier. After the cycle is detected the process continues until the collision point. That is exactly the same number of iterations as it was used in the case with one stack and this is in average $2(\lambda + \mu) = \sqrt{2\pi n}$ iterations. Using two processors the process of finding the collision can be accelerated. For a fixed λ and μ, the average running time for detecting the cycle is

$$\lambda + \mu(1 + \tfrac{1}{k+1}) = (2 + \tfrac{1}{k+1})\sqrt{\pi n/8}.$$

After detecting the cycle the process continues in order to find a collision. It starts with iterating the function $\frac{1}{k+1}\sqrt{\pi n/8}$ times in average until it reaches two points on tail and cycle equally distant from the collision point. It follows by $\frac{k}{k+1}\sqrt{\pi n/8}$ parallel steps in average after which the collision is detected. That is, an additional number of iterations after the cycle is detected in order to find a collision is:

$$\tfrac{1}{k+1}\sqrt{\pi n/8} + \tfrac{k}{k+1}\sqrt{\pi n/8} = \sqrt{\pi n/8}$$

and it doesn't depend on the number of stacks. Summing up all together (the number of steps necessary for detecting the cycle and an additional number of steps for detecting the collision) we come to the conclusion that an average number of steps to detect the collision using Nivasch's method with k stacks is:

$$(2 + \tfrac{1}{k+1})\sqrt{\pi n/8} + \sqrt{\pi n/8} = (3 + \tfrac{1}{k+1})\sqrt{\pi n/8}.$$

Using two processors can save the time for finding the collision in the manner that

$$\tfrac{k}{k+1}\mu = \tfrac{k}{k+1}\sqrt{\pi n/8}$$

steps will be done in parallel manner. This is $\frac{1}{4}(1 - \frac{1}{k+1})$ less time than using one processor. For a $k = 2, 3, 4...$ it is respectively 16.67%, 18.75%, 20%... in average less time for finding a collsion using two processors. As it can be seen, there is no drastic difference between two and three processors in the process of finding the collision. This difference is even smaller between three and four processors. It must be said here that it does not matter how much stacks we define and how early we detect the cycle, for a fixed $\lambda = \sqrt{\pi n/8}$ and $\mu = \sqrt{\pi n/8}$, comparing with one processor, time for finding the collsion using two or more processors cannot be saved more than 25%. Still, most of the time the second processor does not work. More pecisely, for a given k stacks the average waiting time for the second processor is $\frac{2k+4}{k+1}\sqrt{\pi n/8}$.

5.3.3 Finding Collision Using Distinguished Points

The original Hellman attack is not considered to be a treat since the offline phase takes the same time as an exhaustive search. The basic idea of using distinguished point in the cycle method is to store a small number of values of the sequence so such that these values can be easily checked afterward. In this way the repetition can be simply detected. A distinguished point, as it was said earlier, is the value that has easily checked property such as the first k bits are zeros. Going though the pseudo-random walk, points with the distinguished property will be stored. When

pseudo-random walk starts cycling, the distinguished point will be stored for the second time and the cycle will be detected.

Let x_0 be the start point and define iterated function h as $x_i = h(x_{i-1})$. An expected number of iterations until the collision is found is $\sqrt{\pi n/2}$. After this point the process must continue until the next distinguished point. Then the distinguished point for the second time will be detected. This means that the pseudo-random walk started cycling and it must "go backward" in order to reach the collision point. Let us denote the distinguished point that is reached for the second time with x_p, distinguished point on the tail that preceded x_p with x_λ and distinguished point on the cycle that preceded x_p with x_μ. If points on the tail and the cycle are denoted with a and b respectively such that $h(a) = h(b)$ then a lies somewhere between x_λ and x_p, and b lies somewhere between x_μ and x_p.

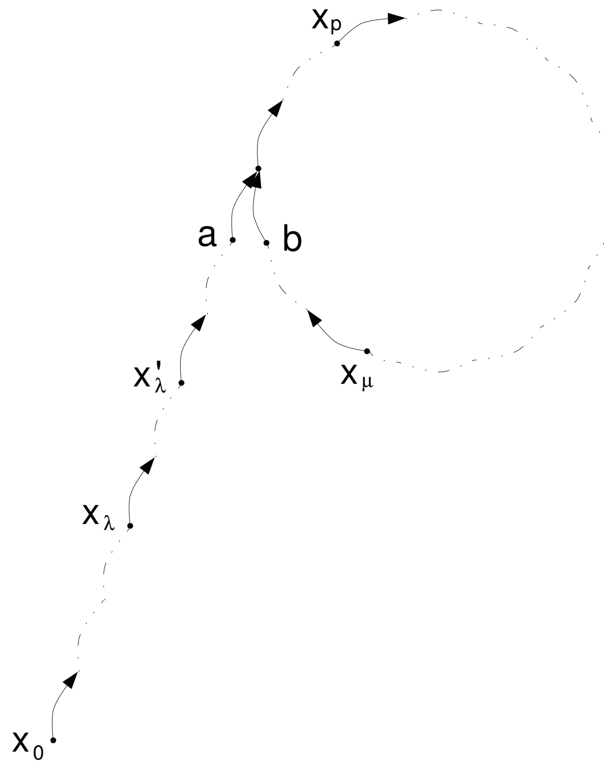


Figure 5.1: Finding a collision using a distinguished point method

Let l be an expected number of iterations between two distinguished points. For instance, if the distinguished property is first k zeros then $l = 2^k$. And let l_λ be a number of iterations from x_λ to x_p and l_μ a number of iterations from x_μ to x_p. Suppose that $l_\lambda > l_\mu$ without any loss of generality. Then starting from x_λ and stepping $l_\lambda - l_\mu$ times it will reach the point $x_{\lambda'}$ that is equally far from x_p as x_μ. What is more important, it is equally far from collision point. This means that starting from x_μ and $x_{\lambda'}$ two trials must step forward together until they both hit the same point. If only one processor is used then expected work to find a collision is

$$\sqrt{\pi n/2} + 3l.$$

Usually it is considered that when $l \ll n$ and then the expected work required for a single processor to find a collision is $\sqrt{\pi n/2}$.

The problem with the cycle method is exposed when the distinguished point method does not have the mechanism to detect an infinite loop. That is, the distinguished point in the cycle is not found. Leave the cycle undetected, the process will continue infinitely without detecting any distinguished points. Therefore, a mechanism must be introduced for detecting the cycle that does not have any distinguished points. This problem is real if the distinguished point is chosen poorly or the function h is not completely random. After some number of iterations without success in finding distinguished point it starts to be suspicious and it can be assumed that the process falls into a loop. Thus, the main difficulty is to choose the length of easily distinguished property in order to get the right probability of occurrence. If the length is too long, then the cycle may not be detected and the algorithm fails. On the other hand, if the length is too short, then we need a huge amount of values that have to be stored.

In [73] the maximum trail length without exceeds the distinguished point was suggested. $1/l$ is the probability that the value has a distinguished property. That is, the average length of a trail without distinguished point is l. Of course l can be adjusted using a differently sized distinguished point. Nevertheless, no matter how l is chosen it is always possible that a trail will fall into the infinite loop. The suggested maximum trail length without a distinguished point is $20l$ after it is supposed that the trail falls into an infinite loop. The probability of trails that reach the length of $20l$ without a distinguished point is

$$(1 - 1/l)^{20l} = e^{-20}.$$

The most obvious solution when the maximum trail length is reached is to start from the beginning with a new start point. Another solution is to continue using some other cycle detection method e.g. Nivasch's or Floyd's method. It guarantees that the cycle will eventually be found. If the value with a distinguished property

is reached, the alternative method is discarded and continues with a distinguished point method as before. Of course this can not be called a pure distinguished point method because in special cases it employs an additional method to find the cycle.

One of possible solutions for detecting the cycle without leaving the original distinguished point method is to redefine the distinguished property and start from the last detected distinguished point. This redefining must be so that the points with the new property are superset of the points with previously defined distinguished point property. For instance, if the distinguished property is last k zeros, then the new distinguished property can be defined as the last $k-m_0$ zeros where $0 < m_0 < k$. In this case, it is more likely to find a distinguished point with the new property. Or, more precisely, there are in average 2^{m_0} more distinguished points and the probability that some of them are in the cycle is 2^{m_0} times higher. Let suppose that after changing the property of distinguished point to $k - m_0$ zeros, the trial is still into an infinitely loop. In this case the distinguished property can be redefined as $k - m_1$ last zeros where $m_0 < m_1 < k$. Eventually, the distinguished point must be found and it guarantees that the cycle is detected. Since this solution is only probabilistic in the sense that there is always a chance that the trail is not in the infinite loop but the distinguished point is still not reached. Nevertheless, if the point with original distinguished property with the k zeros is detected, the method can be switched to an original setting again. However, the scenario that random walk comes into infinite loop is very rare if the distinguished point method is configured properly and has only a theoretical interest.

In Section 4.6.5 a distinguished point method for efficient parallelization of collision search is described. It is distinguished in two cases: one when only small number of collisions is required and the other, when a large number of collision is necessary. An idea is to use more processors which share a single common list. Each processor adds distinguished points with additional information and starts producing a new trial from a new starting point. It can be also considered that the trial continues from the distinguished point but it has a drawback when a large number of collisions is required. Namely, when a trial reaches a collision it falls into the path when same distinguished points are detected over and over again. In the single common list a starting point x_0 and the length of the trial d must also be stored in order to efficiently locate a collision. A collision is detected when a distinguished point appears twice in a list or more precisely when one trial touches another trial. From this point two trials will have the same path up to a distinguished point. When this happens a collision is detected. If more collisions is required one must continue until desired number of collisions is reached. A special case is when one trial touches another trial in a starting point x_0. This yields to the "Robin Hood" which does not produce the collision point [72].

Let m denote the number of processors used to find the collision. We know that $\sqrt{n\pi/2}$ is the expected number of points produced until the collision is found.

Using m processors means that each of them should produce

$$\frac{\sqrt{n\pi/2}}{m}$$

points in average. After the collision is found, it is expected to produce additional l points until the next distinguished point. As in the case with one processor, collision point must be located on both trials. The only difference is that one part can be done in a parallel manner. Namely, it starts with one processor, beginning by stepping the longer trial until it reaches the size of the shorter one. This part must be done serially. After that, both parts step together until they reach a point a and $b(\neq a)$ such that $h(a) = h(b)$. This part can be done in parallel. In [73] the computation was given in details where expected work for locating the collision is

$$\frac{\sqrt{\pi n/2}}{m} + 2.5l$$

Another issue is the case when a large number of collisions is required. This is especially useful when not every solution is desired. For instance, when an input value is smaller than the output value. Then two different output values can use compression function to produce two inputs with the same values which will eventually lead to the same distinguished point. That is, if two trials reach the same distinguished point it does not imply that the collision is found. On the other hand, the goal is to find two different inputs which have the same output. For this reason, it is required to produce a large number of "pseudo-collisions" until the real collision is located. The algorithm is the same as when one collision is required except that it continues until the real collision is found. Of course, there must be some kind of a test that checks if the collision is desired or not. The similar problem occurs in [60] by Quisquater and Delescaille where authors aimed to find a key collision for DES. They used a compression function that maps 64-bit DES output text to the 56-bit DES key. This means that only one of 2^8 DES collisions lead to the true collision rather than pseudo-collision.

5.4 Summary

In this chapter methods for detecting the collisions using different cycle finding algorithms were presented. More precisely, we analyzed the second part of this method, the part after the cycle is detected. The focus was on the time memory trade-off algorithms that are more suitable for the hash functions. For the slight increase of memory they give a faster way of finding a collision.

In the analysis we also considered possibilities of parallelization. Unfortunately, the parallelization can be done only for the limited number of steps, depending on the method, and it has a sense to be executed only by a maximum of two processors.

For some methods they don't give enough acceleration for collision detection. This is the case for distinguished point method and in some situations for Brent's method. In other methods like Nivash's or Floyd's after the cycle is detected the operation continues from the start point. It guarantees that an expected parallel computation will be done for $\frac{1}{2}\sqrt{\frac{n\pi}{8}}$ steps.

In the distinguished point method the case when the point with distinguished property is not found in the cycle or it is not found for a long period was investigated. In [73] it was suggested that this period is $20l$ where l is an average length of trial without distinguished point. Two approaches for overcoming this problem were presented. In Nivasch's multi-stack algorithm is also considered how it affects on detecting the collision. It turned out that multi-stack algorithms does not accelerate the process of collision searching. Only with parallelization, when two processors are included can some time for finding the collision be saved.

Chapter 6

Analysis

6.1 Introduction

Comparing practical results with theoretical expectation is always useful. In practice, creating any cryptographic component tends to have results very close to theoretical, but it is very hard or even impossible to set the sign of equality between them. For that reason, we can just talk about how practical results are close to theoretical. One of the main reasons is the fact that the pure random function is very hard to be constructed. Hash function is a very common tool in the case where the randomness is necessary. Very often, the value of the hash function is considered to have property of randomness. Any deviation from the pure random function leads to situation where generic attacks have better results than expected [5]. In that sense, the pure random function in only a theory.

As in the exhaustive search, finding collision using cycle algorithms depends in general on randomness of the hash function onto which it is applied. On the other hand, the performance of particular algorithm highly depends on the choice of its parameters. There are no strict regulations how these parameters should be used. However, with adjusting mask function, input size as well as distinguished point size in distinguished point algorithm or the number of stacks in Nivasch's algorithm someone can manipulate with a memory complexity, a processing complexity of a success rate.

In this chapter, we present the analysis of cycle finding algorithms. They are applied on a 1-bit version of RadioGatún hash function with a 58-bits output. It ensures fast collision detection. Analysis with different parameters is confirmed by experimental results. It can give us a much realistic picture of how some attacks are treated.

The remainder of the chapter is organized as follows: after an environment in which tests are performed is defined a general property of the random function in

cycle algorithm is explained. Then, with more detail for specified algorithms a set of tests is ran in order to observe the behavior for each of them.

6.2 General Property of Random Mapping in Cycle Algorithms

This section generalizes the concept of random mapping in the cycle algorithm without getting deeper in any of the particular methods. It explains the functionality of the cycle algorithm as the method independent of external factors such as memory or time complexity. Also, different approaches and important parameters have been observed such as meet-in-the-middle attack applied on cycle algorithm or how different size of the parameter in the random function affects finding a collision. The common characteristic is that they can be applied on a cycle algorithm independently from the chosen method used for this purpose.

6.2.1 Performance of Different Input Size

Different size of the input affects on the time and the memory expectation. Applying this on iterated function can be treated as the mask function that has the smaller input than the output. More details about different input size were explained in Section 5.3. Applied as the iterated function in the cycle finding algorithm, in some cases, it can decrease the needs for the storage but at the same time increases the computation time. Depending on the input size, the memory requirements as well as the computation time can vary. Since this experiment compares performance, all results depend on the hash function implementation.

An experimental comparison shows that long inputs are processed 16 times faster than short inputs. The size of one block is considered one input, that is, 39 words size. The padding scheme that fits the input of multiple block size is not processed. Padding scheme has a drastic influence in the hashing of small inputs. This difference will be even bigger if padding scheme is taken into account. The reason why this difference is so big lies in the fact that input blocks follows 16 blank iterations. The Figure 6.1 shows how the hashing speed depends on the input size. The input is a multiply of block size and the padding process is not considered. Both figures show how blank iterations are influenced by the speed of hashing. Tested on 64 bit Inter Pentium(R) Dual-Core E5300 2.60 GHz with 2M cache and 4GB RAM under Debian Linux Kernel 2.6.30 an average speed of RadioGatún [32] is about 290 MB/s and 550 MB/s for RadioGatún[64] for a big input. The speed of one block is 17.5 MB/s for 32 bit version and 35 MB/s for 64 bit version using the same environment which is about 16 times slower than for the big input.

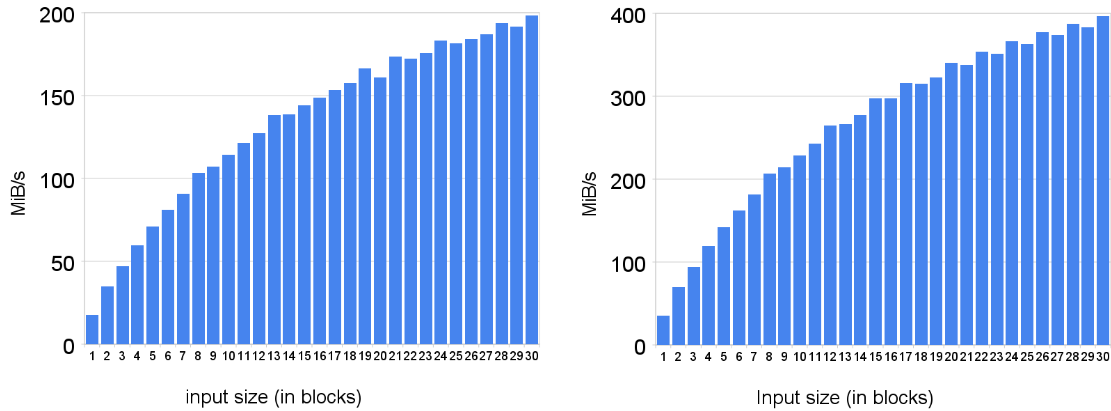


Figure 6.1: The speed of RadioGatún[32] (left) and RadioGatún[64] (right) for small inputs

Another drawback is the padding scheme which takes significant processor time when the input is small. The Figure 6.2 shows how the speed of hashing using RadioGatún is changing by taking different size of inputs. The small inputs is taken and all irregularity behavior of the graph are shown depending on the input size. Two big things can be seen from the graph: 13 small "cliffs" on each 3 word size input and the big "hills" on each 39 word size input. These clearly describe the internal structure and the hashing process. Namely, in order to increase the performance two different functions for hashing data are possible. Input data must be separated by three words. The padding scheme is done in the way that, at the end, it is multiplied of three and then the hashing function takes three by three words. This is the smallest unit that can be hashed at once. Hashing data this way is slow. For that reason, it takes the whole block of 39 words and hashes it "at once". In other words, input data is divided in blocks of 39 words. The last one is multiply of three. Since the hashing block is much faster of hashing three by three words, the significant increase when the input reaches the size that is multiple of 39 can be noticed in the graph .

6.2.2 Applying Meet in the Middle Method on Cycle Finding Algorithm

In order to be able to perform meet-in-the-middle attacks on a certain function, it should be possible to be computed in a backward direction from the given value. It is believed that reaching all zeros initial state value going in a backward direction is not less difficult than reaching the target state from the starting initial point [6]. The invertible nature of the round function in RadioGatún gives the possibility of

application in meet-in-the-middle strategies.

In the Section 4.5 the basic idea of the meet-in-the-middle attack was given as well as the explanation of how it can be applied in the cycle finding algorithm in general. Using different time-memory trade off techniques will not change the expectation for finding the collision. The only difference is that instead of an usual iterated function the inverse function with the criteria that decides in which "direction" the iterated function should be used can be applied as well.

The meet-in-the-middle strategy in the cycle finding algorithm has one very important advantage. It can be used not just for creating one preimage of the given value but for finding multiple preimages too. From the same image value y and starting each time from different values different collisions can be found. Namely, suppose that using the explanation from the Section 4.5 starting from the random value x_0^1 the collision such that $x^1 = x_{i-1}^1 || x_{j-1}^1$ and $y = h(x^1)$ was reached. Suppose further that a different started random value x_0^2 leads to the collision that gives the solution $x^2 = x_{i-1}^2 || x_{j-1}^2$ such that $y = h(x^2)$ and so on. This can be continued depending on how much preimages are needed. This means that at the end different values $x^1, x^2, ..., x^n$ will be calculated that after applying the function h on them the same output value y is produced. From the experimental results of the preimage value using one bit version of RadioGatún in the Table 6.1 two preimage results are presented. More results continuing the execution of meet-in-the-middle method or changing the internal state of RadioGatún can be found.

image
1011001011110000011000110101100101000110110100000111100101

preimages
110000100000011010100010101001100010101011111101001010111111111111 11111111110110111110010101011110110110000010111011001001101001101100
11111111100011100111111010011000100110100101001000010001001111111111 11111111110110101100111010001110100011111010111011110100011101010010

Table 6.1: Preimage values given using meet-in-the-middle method

6.2.3 Collisions in Cycle Algorithms

Before we start to perform different tests it is necessary to see how RadioGatún corresponds to basic expectations. The internal state of RadioGatún is directly controlled by the input block. In the last input, three words are injected into Mill and Belt parts which is then transformed regarding functions defined in standard RadioGatún specification. Since in internal state the Mill as well as the Belt part are included and are both updated from the same input block, consequently, the

generic attack has the security that corresponds to a 58 word size. This means that the expectation to find the collision is $1.17 \cdot 2^{29} \approx 2^{29.23}$. Experimental results shows that RadioGatún can be broken after $2^{29.20}$ operations in average, corresponding very closely to the given specification.

Similar to the subsection 6.3.1 the behavior of the expectation for the collision can be tested. Even if the expected collision length is confirmed practically, this does not necessary mean that the whole distribution corresponds unambiguously. Different segments of the distribution was taken and, for each of them, the test result is compared with the expected values. Table 6.2 shows results obtained by the samples of 100 collisions.

Collision property	Region (log_2)	Theoretical (log_2)	Experimental (log_2)
29.23	26-27	26.5764	26.5683
29.23	27-28	27.5677	27.5773
29.23	28-29	28.5504	28.5356
29.23	29-30	29.5156	29.4992
29.23	30-31	30.4479	30.4482
29.23	31-32	31.3119	31.2838

Table 6.2: Expectation of collision compared with experimental results

The role of different input size in finding collisions in the cycle finding algorithm was already discussed in the Section 5.3. Using a smaller input size for the iterated random function causes a decrease in memory requirements. Since the simple collision, by itself, is not a satisfying solution, the process must be repeated until the real collision is found. On the other hand, it increases the number of iterations. This causes that an additional time is needed for the real collision.

Suppose that the output size of the random function has a length n and the input value takes the size of $k < n$. Approximately $1.17 \cdot 2^{k/2}$ iterations are needed before the trial starts cycling. This will produce a simple collision but not necessarily a real collision. If this collision is not the desired one, the process should be repeated using a different starting point each time until the real collision is reached. As it was discussed earlier, to find the real collision 2^{n-1} comparisons was needed to be created. In the process of finding one simple collision about 2^{k-1} comparisons are involved. If l is the number of simple collisions that are found before at least one real collision is found, then

$$l2^{k-1} \geq 2^{n-1}.$$

This means that the expectation to find the first real collision is after

$$l = 2^{n-k}$$

tries. This leads to the next observation. The number of iterations until the real collision is found is

$$2^{n-k}2^{k/2} = 2^{n-k/2}$$

6.3 Analysis of Cycle Finding Algorithms

Theoretical expectations do not coincide always with the experimental results. It is necessary beside the theoretical results to be observed how the cycle finding algorithms behave in practice. It gives us a much wider picture of how a particular algorithm uses its memory and processor resources. By comparing different algorithms and their characteristics all advantages and disadvantages must be considered.

Not every algorithm is suitable for each situation and for that reason each one must be treated differently. They also use the resources in different way and, because of that, there is no universal analysis for all algorithms. In the most cases, they must be analyzed separately and a different approach must be constructed for each of them. The correct prediction of the result is crucial in the process of analyzing. It gives us the starting facts which need to be confirmed by the experimental results.

6.3.1 Distinguished Point Parameters

The effectiveness of the distinguished point method mostly depends on how parameters are chosen. Choosing appropriate parameters has a central role in the memory/processing efficiency. There is no strict rule how they should be chosen. There are only suggestions. The distinguished point method is probabilistic method and it does not guarantee cycle detection. That is, no matter how good distinguished point parameters are chosen, there is always a theoretical chance that this method, in its rough form, does not give us a collision. Analyzing the behavior of the distinguished point method can give us a more realistic expectation. The great impact on the success of the distinguished point method has the function used in the process of iterations. Here, all experiments are applied on RadioGatún hash function which is supposed to behave as random. In practice, not all functions are random. They behave as quasi-random or have some special characteristics such as extremely small cycles. In this case, these results can not be considered accurate and, what is more, the distinguished point method is not the best choice.

The Number of Distinguished Points

Changing parameters in the distinguished point method affects the performance of the method. One of the most important parameters is how to choose the distinguished point and how long it should be. Making the distinguished point shorter

takes the penalty in the number of the memory accesses, the storage for the data is getting bigger and the searching for data is getting slower, especially in the worst case scenario. Longer distinguished points cause an extra computation time and higher possibility that the trial comes into an infinitely loop. Consequently, calculating the right prediction is important. Thus, here, the number of distinguished point in the cycle will be computed. In the following computation, various expectation regarding the number of distinguished points in the cycle will be calculated and compared with the practical experiments.

In order to achieve some of these expectations, a theoretical background must be set up. The number of distinguished points depends on the path length. That is, it is related to the expectation of collision occurence. The probability that the collision occurres in time k is:

$$P(col = k) = 1 \cdot (1 - \frac{1}{n})(1 - \frac{2}{n}) \dots (1 - \frac{k-1}{n})\frac{k}{n}$$

$$P(col = k) = \frac{n!}{(n-k)!n^k}\frac{k}{n}$$

where n is the number of all possible outputs. Using Stirling's approximation for $n!$ it follows that

$$P(col = k) = \sqrt{\frac{n}{n-k}}\left(\frac{n}{n-k}\right)^{n-k} e^{-k}\frac{k}{n}$$

In the Figure 6.3 the probability distribution of the path length until the collision is reached is given. If we know the path length, we can give the expectation for this particular case. This is given by the formula

$$E(X^{\mu}_{DP}) = (\mu + \lambda)/2^d$$

where 2^d is the number of distinguished points. The same can be applied on a certain arbitrary path length. The only difference is that instead of $\mu + \lambda$ will be the new symbol that represents this arbitrary path length. The expected number of distinguished points is easy when the length is known. A more complicated evaluation takes place when the length is not known. Then in the equation the random variable that represent the path length must be included.

Estimation of the Length Between Two Distinguished Points

There is also one behavior that can be changed using the different parameters in distinguished point method. This is the length between two distinguished points sometimes also known as a chain length. First the probability that the distinguished

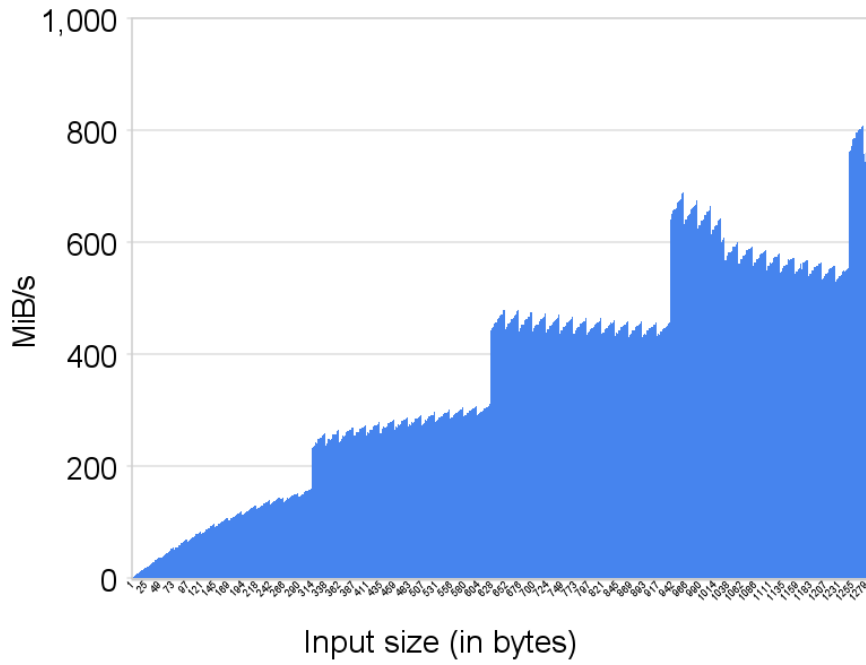


Figure 6.2: Hashing speed for different input size

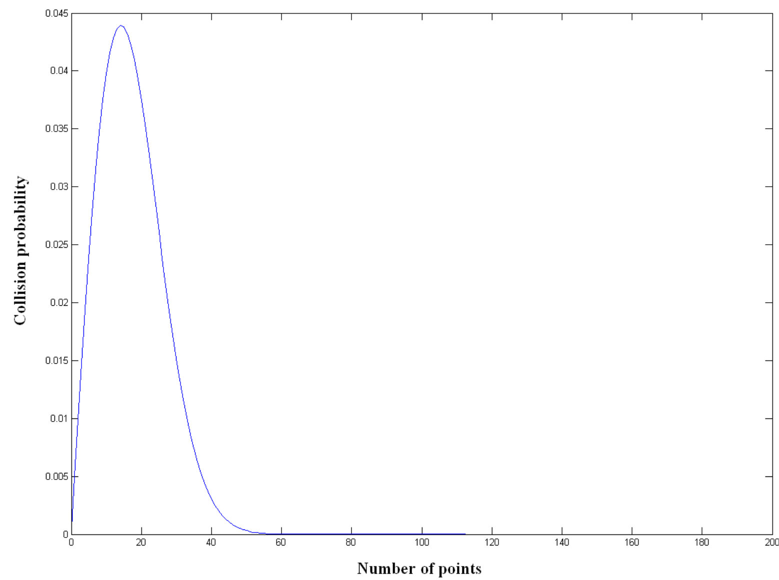


Figure 6.3: Probability distribution of the first collision appearance

point is reached after k steps must be estimated. It must be also taken an assumption that inside this k steps the cycle will not be detected, otherwise a loop will be infinite. In [70] some interesting evaluations of the distinguished point method applied on DES were given. To be able to define the distribution the probability that the distinguished point is reached for the first time in exactly l steps must be computed first. The probability that the distinguished point does not occur in less than l steps is:

$$P(d \geq l) = \left(1 - \frac{2^{n-d}}{2^n}\right)\left(1 - \frac{2^{n-d}}{2^n - 1}\right)\left(1 - \frac{2^{n-d}}{2^n - 2}\right)\cdots\left(1 - \frac{2^{n-d}}{2^n - (l-1)}\right)$$

$$= \prod_{i=0}^{l-2}\left(1 - \frac{2^{n-d}}{2^n - i}\right)$$

The denominator is decreased each time since the probability of making the cycle in the chain is excluded. Since we have in practice that $i \ll n$, an approximation $i = \frac{l-1}{2}$ can be taken which gives the equation:

$$P(d \geq l) \approx \left(1 - \frac{2^{n-d}}{2^n - \frac{l-2}{2}}\right)^{l-1}$$

To compute the probability that the collision occurred in exactly l steps $P(d \geq l) - P(d \geq l+1)$ needs to be calculated or it can be simply defined as:

$$P(d = l) \approx \left(1 - \frac{2^{n-d}}{2^n - \frac{l-2}{2}}\right)^{l-1} \frac{2^{n-d}}{2^n - l + 1}$$

The given equation describes the distribution of the chain length. In order to compare the practical results with the theoretical expectation some results from [70] were taken. The interval in the graph is divided in m parts $[t_0, t_1 - 1], [t_1, t_2 - 1], [t_2, t_3 - 1] ... [t_{m-1}, t_m - 1]$. For some parts the expectation of the chain length is then calculated and compared to practical results. The average chain length between t_i and t_{i+1} is given by:

$$\overline{l_{t_i, t_{i+1}}} = \frac{\sum_{l=t_i}^{t_{i+1}-1} lP(\rho = l)}{\sum_{l=t_i}^{t_{i+1}-1} P(\rho = l)}$$

This is not very accurate for analyzing and it must be transformed in a differnt form. So,

$$\sum_{l=t_i}^{t_{i+1}-1} P(d = l) = P(d \geq t_i) - P(d \geq t_{i+1})$$

A nominator is a little bit harder to transform in an acceptable form. The following formula can be used:

$$\sum_{l=t_i}^{t_{i+1}-1} lP(d=l) = \sum_{l=t_i}^{t_{i+1}-1} l(P(d \geq l) - P(d \geq l+1))$$

$$= \sum_{l=t_i}^{t_{i+1}-1} l \left(\prod_{i=0}^{l-2} \left(1 - \frac{2^{n-d}}{2^n - i}\right) - \prod_{i=0}^{l-1} \left(1 - \frac{2^{n-d}}{2^n - i}\right) \right)$$

$$\approx \sum_{l=t_i}^{t_{i+1}-1} l \left(\left(1 - \frac{2^{n-d}}{2^n - \frac{t_{i+1}+t_i-1}{2}}\right)^{l-2} - \left(1 - \frac{2^{n-d}}{2^n - \frac{t_{i+1}+t_i-1}{2}}\right)^{l-1} \right)$$

$$= \sum_{l=t_i}^{t_{i+1}-1} l \left((1-x)^{l-2} - (1-x)^{l-1} \right)$$

where $x = \frac{2^{n-d}}{2^n - \frac{t_{i+1}+t_i-1}{2}}$. This gives us the following

$$\sum_{l=t_i}^{t_{i+1}-1} l \left((1-x)^{l-2} - (1-x)^{l-1} \right)$$

$$= (1-x)^{t_i-2}(t_i + \frac{1-x}{x}) - (1-x)^{t_{i+1}-2}(t_{i+1} - 1 + \frac{1}{x})$$

This means that the average chain length for the given interval is

$$\overline{l_{t_i,t_{i+1}}} = \frac{(1-x)^{t_i-2}(t_i + \frac{1-x}{x}) - (1-x)^{t_{i+1}-2}(t_{i+1} - 1 + \frac{1}{x})}{P(d \geq t_i) - P(d \geq t_{i+1})}$$

Using this equation can be practically tested the chain length.

First the probability of the chain length for the fixed distinguished point property will be tested. The setting is adjusted such that the distinguished point has the length 20 bits and the input value has the length 58. For each observer region 100 examples was taken. Table 6.3 contains the results of the observation.

The expectation of the chain length for various distinguished point property is considered in the next test. A wider region was taken to be seen if theoretical results correspond to experimental results. Table 6.4 can be used for comparing the given results.

This practical results confirms the theoretical expectations in the sense that the chain length doesn't have just the expected length in general but also the expected length for the certain region. In this way we confirmed that results coincides in the whole region of the probability distribution.

DP property	Observer region (log_2)	Theoretical (log_2)	Experimental (log_2)
20	16-17	16.5799	16.6129
20	17-18	17.5749	17.5512
20	18-19	18.5648	18.5478
20	19-20	19.5445	19.5377
20	20-21	20.5039	20.5037
20	21-22	21.4260	21.4183
20	22-23	22.3001	22.3202

Table 6.3: The chain length for the fixed distinguished point property

DP property	Observer region (log_2)	Theoretical (log_2)	Experimental (log_2)
14	12-16	14.2318	14.1883
16	14-18	16.2317	16.2342
18	15-20	18.2317	18.2328
20	18-22	20.2317	20.2241
22	20-24	22.2317	22.2213
24	22-26	24.2317	24.2147

Table 6.4: Observation of the chain length for the various distinguished point properties

Influence of Parameters in Time/Memory Complexity

Calculating time/memory complexity in the distinguished point method can be done only in the conjunction with different parameters. First of all, these parameters consider the length of the output and the distinguished point property. So, time/memory complexity will be observed when different parameters of distinguished point methods vary. Here, all calculations were taken under the assumption that the iterated function has a random mapping. Of course, all expectations will be experimentally confirmed by a set of tests.

A memory requirements, is directly related with the expectation number of the distinguished points. This means that the size of the memory grows linearly, depending on the number of distinguished points. The expected number of distinguished points for the given path length was explained earlier. This gives us the expected memory requirements for the given path

$$(n-d)\sqrt{\pi n/2}$$

since it is not necessary to store data which describes a distinguished property.

6.3.2 Efficiency of Using Stack in Nivasch's algorithm

Unlike the distinguished point method, the amount of the memory in the Nivasch's method depends on the value order in the path. In this part the behavior of the stack occupation using the Nivasch's method will be investigated. The subsection 5.3.2 already discussed about the various expectations. In [51] an expectation of the stack size for a random function was defined. First, we will test how RadioGatún responds as the function with the random mapping on the following expectation:

- the stack size at the time n has an expectation $\ln(n) + O(1)$
- the stack size at the time n is almost surely $> \delta \ln(n)$ for any constant $\delta < 1$
- the maximum stack size up to time n is almost surely $< \delta \ln(n)$ for any constant $\delta > e$

A test applied on RadioGatún hash function gives us the results represented in the Table 6.4. The test ran 100 executions of Nivash's algorithm. Upper line can be seen as a bound of the memory requirements. If the memory is static it needs to be estimated before the execution of iterated function. The experiment that made on 100 examples shows that an expected memory occupation at the time n is $\ln(n) + 0.5148$.

In the same subsection (5.3.2) the case when more than one stack is used to store data was given. The question is what benefits can give us this approach of using the Nivasch's method? As discussed earlier, the number of iterations stays the same in the process of finding the collision since, after the minimum value in the cycle is allocated, iteration must be continued until the collision. That gives that in any case $2(\lambda + \mu)$ iterations need to be performed. So, even if the number of iterations stays unchanged, the situation with the memory units, that need to support multiple stacks, becomes more complicated.

Let us suppose that there are k stacks used to accomplish Nivasch's multiple stack algorithm. At the time n each stack will process in average n/k values. In other words, using the function with the random mapping the Nivasch's multiple stack algorithm will have the following expectation:

- the stack size at the time n has an expectation $\ln(n/k) + O(1)$. The size of all stacks at time n has an expectation $k(\ln(n/k) + O(1))$
- the stack size at the time n is almost surely $> \delta \ln(n/k)$ for any constant $\delta < 1$.
- the maximum stack size up to time n is almost surely $< \delta \ln(n/k)$ for any constant $\delta > e$.

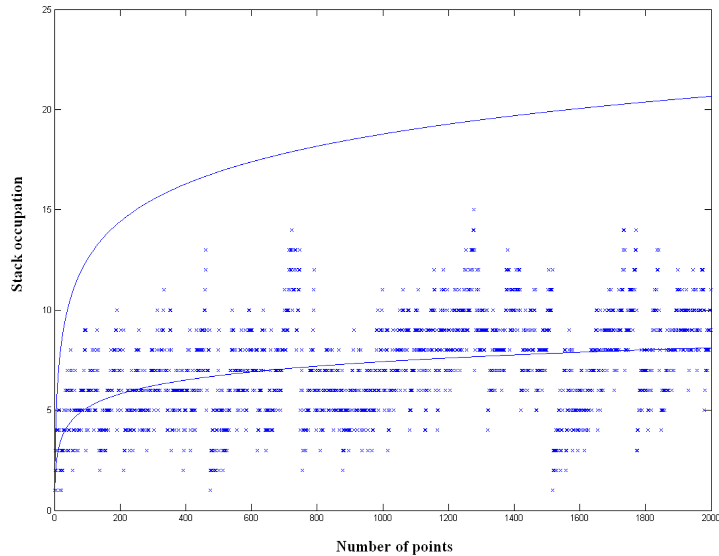


Figure 6.4: Stack occupation in Nivasch's method. The behavior of the stack size is compared with en expected stack size (lower line) and the upper bound of stack size (upper line).

From here it can be seen that even the cycle detection using multiple stacks is much more effective, the memory requirements for a such approach are much more demanding. Figure 6.5 gives results from the Nivasch multiple stacks method compared to their expectations. As for the single stack the sample was taken from 100 executions of Nivash's multiple stack algorithm. The smooth line represents the expected stack size in time n. Taking the sample an experimental result shows that the expected stack size at time n when $k = 4$ distinct stacks are used is $k \ln(n/k) + 2.15$. Taking the results from the single stack algorithm and comparing them to the results of the multi-stack algorithm the conclusion that the expectation stays into the appropriate theoretical frame results can be given.

Very often the performance of the distinguished point and Nivasch's method is compared. Choosing the method that fits the best depends mostly on the environment. Distinguished point method in general requires detecting the cycle which is more faster than the Nivasch's method but in the case when the function is not random, i.e. where the function makes small cycles, the Nivasch's method can be a good choice.

If it is necessary to decrease the memory storage in the Nivasch's method, it can be done by taking only the point with certain property into account. It is similar as to the distinguished point method but, instead of comparing to all previous

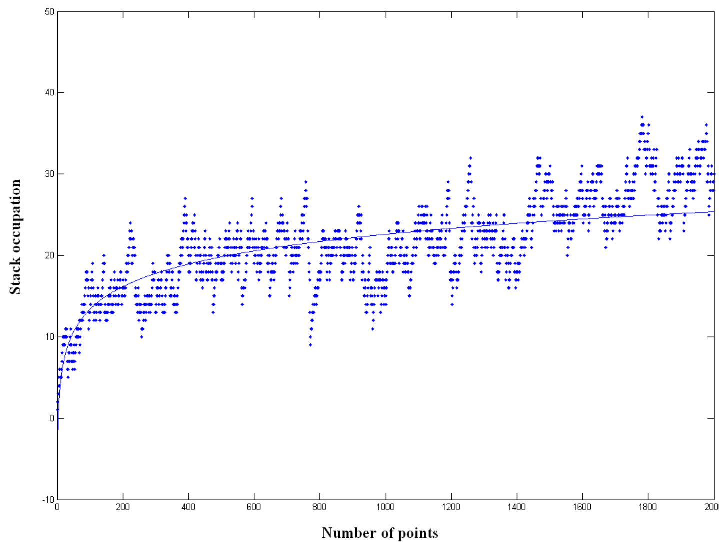


Figure 6.5: The average and the real memory occupation of the Nivasch's multi-stack algorithm

stored values, a stack is used as in the Nivasch's method. Changing the Nivasch's method this way makes it lose a very important characteristic; this method is no longer deterministic in the way that it does not guarantee the cycle detection in any possible situation and it becomes probabilistic. However, if it is dealing with the random function and if parameters are chosen correctly then the chance that the cycle stays undetected is almost zero. An interesting thing is that using this approach does not change an average number of iterations until the cycle is detected and stays unchanged as in the original Nivasch's method. In both cases it is in average $\frac{5}{2}\sqrt{\pi n/8}$ iterations. Consequently, the expectation for the collision is also unchanged and it is $\frac{7}{4}\sqrt{\pi n/2}$ iterations in average.

The amount of saved memory can be adjusted. Suppose that the property is taken so that it is expected that the appropriate value is taken after each k steps in average. It can be observed as one of the stack in the multiple stack variant of Nivasch's method. Then in the time n it is expected that the size of the stack is $\ln(n/k) + O(1)$. If it is needed that the stack size is decreased for p then $k = e^p$ should be chosen. Interestingly, it does not depend on the time n. It gives the conclusion that for any value of n, the stack size will be, in average, smaller for k in comparison to the usual stack of Nivasch's method. However, the cost for decreasing the memory is high since k depends on p exponentially. The reason for this must be very convincing and justified, for example in the device with very limited memory size. One of the major advantages of using this approach is that

the need for memory access is drastically decreased. It reduces the memory access in average k times.

6.3.3 Performance of Cycle Methods on Different Platforms

Comparing the structure of the Microsoft Windows and Linux operating system family is a common topic in the personal computer industry. They both went through various versions and advanced in order to satisfy the needs of their users. While Microsoft Windows operating system is focused on the sales market, the Linux has the status of being the most prominent free software operating system. They all expand their market on a variety of other devices such as embedded systems. The most interesting fact here is that they are constructed on different philosophies and they very often have different solutions and algorithms for solving the same problems. This leads to the clue that they may have different execution time for the same task.

Cycle finding methods including the structure of the hash function consist of primitive functions and simple algorithms. They are executed very fast on both platforms but even the slightest difference in the performance can have very important meaning when the process of the whole method is considered. The memory management is another issue, but it is only actual when the method that is used the memory efficiently is observed, such as Nivasch of distinguished point method. However, storing and finding data depends on the sorting algorithm. Thus, to be able to better explain possible differences, before the method is compared on different platforms, sorting solution chosen for some a particular method must be explained shortly. In the distinguished point method the hash table with chaining list for storing data is used. It establishes a relatively fast algorithm for finding data and constant time for adding a new element. The balanced tree instead of the chaining list can be also considered as a possible solution since the load factor is large. In the Nivasch's method the sorted list is used. New element is always compared with the top one. If the new element is smaller it is compared with the next one and so on until the element with the biggest value is reached. At he first glance, it seems very trivial and inefficient but this algorithm makes sense since the smallest elements are at the bottom of the list and they are accessed very rarely. On the other hand the top of the list is changed very often and it is reasonable that the search starts from the above.

For the comparison test environment Windows 7 and Debian Linux Kernel 2.6.30. distribution are taken. These both run on 64 bit Intel Pentium(R) Dual-Core CPU E5300 2.60 GHz with 2M Cache and 4GB RAM. Both opereting systems use Intel C++ Compiler 11.1 which performance is optimized for Intel processors. Figure 6.6 shows test results applied on different algorithms. The test applied after 50 collisions for each method. The settings, such as the start point and mask func-

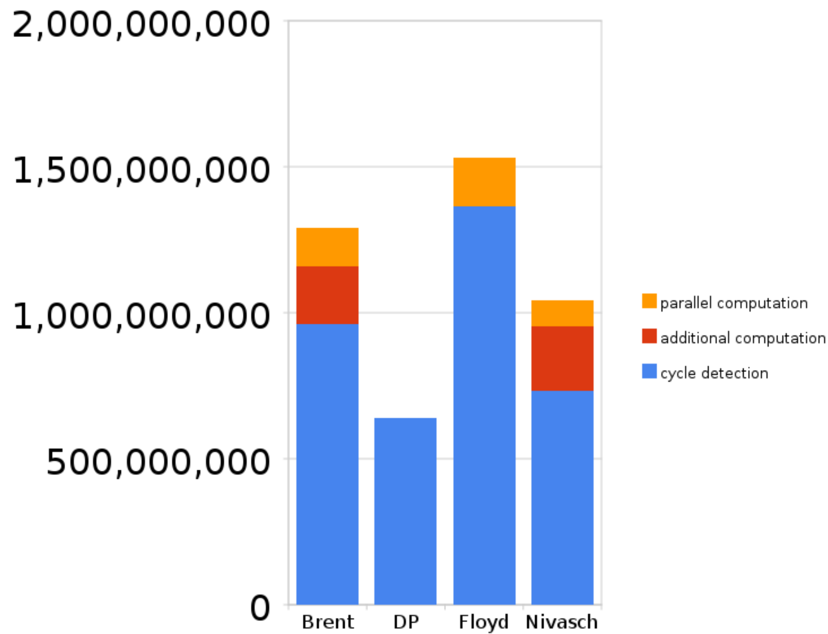


Figure 6.6: Graphical representation of results taken from Table 6.5

tion, is also the same for each method. The number of iterations and computations in each of observed algorithms were tested. Note that the number of iterations does not need to be the same as the number of computations. For instance, in the Floyd's algorithm in one iteration three computations of the function are included. The process of detecting the collision using cycle finding techniques can be divided in three phases: the first phase is detecting the cycle; the second phase is the part after the cycle is detected and where the process must be done sequentially; the third phase is after which the collision is found and it can be done in parallel. In the Table 6.5 the test results are given. The last two columns are durations. They shows how much time is needed for particular method to find a collision. Methods are tested on both operating systems. Note that tests are performed on software implementation and that results depend on it. This means that tests relay on another implementation can lead to different results.

Simple testing of the RadioGatún hash function, so that the output of one execution is an input of the next one, shows that, under the Linux environment it is approximately $5-6\%$ faster than under Windows. Note that 1 bit version without blank and output round function was using and that the result is taken directly as the internal state after the input round functions. It turns out that the distinguished point method and Nivasch's method are about 5.44% and 6.08% slower on Windows environment. It also shows that the Floyd and Brent methods

are about 5% slower in Windows than in Linux environments.

Algorithm	Cycle detection (log_2)	Additional computation (log_2)	Parallel computation (log_2)	Time Linux (minute)	Time Win (minute)
Brent	29.837	27.601	26.959	61.186	64.25
DP	29.252	18.468	17.822	26.24	27.67
Floyd	30.345		27.315	62.71	65.84
Nivasch	29.450	27.710	26.445	42.75	45.35

Table 6.5: Collision test results using different cycle finding algorithms on 1 bit version of RadioGatún

6.4 Summary

All tests in this chapter used RadioGatún hash function. In general, all of them can be also applied on any other function. The aim of this analysis is to demonstrate different cycle method techniques. Different theoretical analysis was presented for various cycle finding techniques. Thus, the results are not so important and they are here only to prove our expectation. They are used to confirm the property of RadioGatún hash function observed from different aspects. What is more important, here are used methods to find collisions and their performances in order to be able to choose the appropriate one in certain situations. Various models allow predictions of some abnormalities in the function and need corrections in the early stage. So, the situation where the theoretical prediction induces the practical expectation is always suggested.

In this chapter a parallelization is presented and were experimentally shown its affect on each methods. It can be applied only on the last stage of the collision search and for different methods it has a different influence. Floyd's algorithm has the longest parallel computation, mostly since it does not require any additional computation after the cycle is detected. The shortest parallel computation is in a distinguished point method which, in average, is not bigger than the distance between two distinguished points.

The Nivasch's multiple stack algorithm does not reduce the number of computations drastically. What is more, the only sensible way of using multiple stack algorithm is when two processors are involved, since this allows the final part of computation to be done in parallel. Also, this chapter also experimentally shows, that the stack in Nivasch's multi stack algorithm has an expectation of $k(\ln(n/k)+O(1))$ where k is the number of stacks. It gives us a possibility to define the number of

stacks in order to reduce the memory requirement but, since it grows exponentially, it does not give a nice performance.

Chapter 7

Conclusion

Representing and analyzing different cycle finding techniques arises one question. Which one is the best? This primarily depends on circumstances and environments. If it is assumed that an iterated function behaves like a random function then distinguished point method can be a good choice. If the environment is set so that the memory has a very limited capacity then Nivasch's method with controlled stack can solve this problem. On other hand, if the iterated function does not behave like a random function and has some abnormal characteristics (i.e. often creates short cycles) then, for sure, the distinguished point is not a good choice since it is a probabilistic method and does not guarantee that the cycle (and therefore the collision) will be detected. In short, a universal answer on this question does not exist and it depends on many factors.

To be able to make a good choice, the time complexity for finding the collision needs to be calculated. It requires additional effort. Namely, when the cycle is detected the process must continue until it reaches the collision point. This task is not always trivial. How fast the collision will be found after detecting the cycle depends mostly on the information the particular method provides. If the method keeps track of previously reached points (such as the distinguished method point and sometimes Brent's method) then finding the collision after the cycle is detected can be relatively fast. On other hand, methods such as Nivasch's, Floyd's or in most cases Brent's do not carry any useful information which lead to significant reduction of iterations after the cycle is detected. Further research can go in this direction, in the modification of these methods so that the part after the cycle is detected is reduced. Producing some information can be used in the iteration process that does not need to start from the initial value but from some points close to the collision.

Here, an implementation of the most important cycle finding techniques was performed and experimental results, that confirmed possibilities and effectiveness, were presented for the purpose of finding the collision. All results and expectations are used to show generic attacks on 1 bit version of RadioGatún hash function

but it can be also applied on any other hash function or just function in general without changing algorithms. Although all methods can not be used in any situation and each of them has certain advantages when compared to other algorithms, in the case when a random function with a sufficient amount of memory is involved distinguished point method has the overall performance. The distinguished point method finds collisions with 38% iterations less than when Nivasch's method is used. Also it has 57% and 58% iterations less then when Brent's or Floyd's method is used respectively. The main power of the distinguished point method lies in the fact that after the cycle is detected the number of remaining steps, before the collision is located, is small, unlike in other methods where the iteration must continue mainly from the beginning. The second big advantage (especially when it is compared to Nivasch's method) is that it does not need to search for data in a look-up table all the time.

Before any analysis was set, we had calculated different expectations for any cycle finding algorithm we observed. We went a little more further and gave some collision expectations in cycle algorithms. In Brent's algorithm, which is an advanced variant of Floyd's algorithm, we differentiate two cases $\mu \leq 2^{k-1} - 1$ and $\mu > 2^{k-1} - 1$ where in the first case, the collision can be found faster. In Nivasch's algorithm with k stack using one processor, an average number of iterations to find a collision is $2(\lambda + \mu) = \sqrt{2\pi n}$ and it does not depend of parameter k. Only by using two processors the time can be saved in the sense that $\frac{k}{k+1}\mu = \frac{k}{k+1}\sqrt{\pi n/8}$ steps can be done in parallel. Nevertheless, using parallel computation more than approximately 25% of time can not be saved. This lead to thinking whether introducing the parallelization in this method makes sense. Another doubt is introducing the parallel computation into cycle finding technique in the distinguished point method. The length of parallel computations in average is not more than the chain length and it depends on the distinguished point property. This is significantly less than the number or steps required for the cycle detection. Thus, introducing parallelization in this method must have a very good reason.

Choosing parameters in distinguished point method (like different properties for distinguished points) were used for testing the behavior of the hash function. In this case, RadioGatún hash function was used. More precisely, the distance from two distinguished points was used to test the randomness of the hash function. The probability distribution was established based on different distinguished point properties and observed regions. The average chain length for different regions was tested and compared with theoretical expectations. This comparison can tell us more about randomness of a hash function as well as the prediction of memory requirements before the collision is found. The size of memory in Nivasch's method was also analyzed. Since the size of the memory is not continuous and varies after each step a good prediction was necessary. For this purpose the size of the memory was analyzed and it has been showed at the time n the memory occupation is

$\ln(n) + 0.5148$ in average but not more than $e \ln(n)$. This is useful when we need to predict the size of memory before the process for detecting the collision starts.

Implementations and tests for all algorithms are done as the software solution. Further applications and analysis can go in the direction where these methods are realized as hardware solution. Then, working with a small memory size can be a very important factor. Also detecting the cycle and finding the collision in it does not need to be strictly related to random functions. Thus, investigating cycle finding techniques on non-random functions can also be an interesting challenge.

Bibliography

[1] Jean-Philippe Aumasson. Faster multicollisions. In *INDOCRYPT*, pages 67–77, 2008.

[2] J. Vandewalle B. Preneel, R. Govaerts. Cryptographically secure hash functions: an overview. *ESAT Internal Report*, 1989.

[3] S. Bakhtiari, R. Safavi-naini, and J. Pieprzyk. Practical and secure message authentication. In *In Series of Annual Workshop on Selected Areas in Cryptography (SAC)*, pages 55–68, 1995.

[4] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In *CRYPTO*, pages 216–233, 1994.

[5] Mihir Bellare and Tadayoshi Kohno. Hash function balance and its impact on birthday attacks. In *EUROCRYPT*, pages 401–418, 2004.

[6] Guido Bertoni, Joan Daemen, Gilles Van Assche, and Michaël Peeters. Radiogatún, a belt-and-mill hash function. NIST - Second Cryptographic Hash Workshop, August 24-25, 2006.

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The road from panama to keccak via radiogatún. In Helena Handschuh, Stefan Lucks, Bart Preneel, and Phillip Rogaway, editors, *Symmetric Cryptography*, number 09031 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[8] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from pgv. In *CRYPTO*, pages 320–335, 2002.

[9] Charles Bouillaguet and Pierre-Alain Fouque. Analysis of the collision resistance of radiogatúnusing algebraic techniques. In *Selected Areas in Cryptography*, pages 245–261, 2008.

[10] Richard P. Brent. An improved monte carlo factorization algorithm. *BIT*, 20:176–184, 1980.

[11] Christophe De Cannière, Florian Mendel, and Christian Rechberger. Collisions for 70-step sha-1: On the full cost of collision search. In *Selected Areas in Cryptography*, pages 56–73, 2007.

[12] Henri Cohen. *A course in computational algebraic number theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[13] Henri Cohen and Gerhard Frey, editors. *Handbook of elliptic and hyperelliptic curve cryptography*. CRC Press, 2005.

[14] Joan Daemen and Gilles Van Assche. Producing collisions for panama, instantaneously. In Alex Biryukov, editor, *FSE*, volume 4593 of *LNCS*, pages 1–18. Springer, 2007.

[15] Joan Daemen and Craig S. K. Clapp. Fast hashing and stream encryption with panama. In *FSE*, pages 60–74, 1998.

[16] Ivan Damgård. Collision free hash functions and public key signature schemes. In *EUROCRYPT*, pages 203–216, 1987.

[17] Ivan Damgård. A design principle for hash functions. In *CRYPTO*, pages 416–427, 1989.

[18] Drew Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

[19] Bert den Boer and Antoon Bosselaers. Collisions for the compression function of md5. In *EUROCRYPT '93: Workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 293–304, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[20] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[21] W. Diffie and M. Hellman. Exhaustive cryptanalysis of the nbs data encryption standard. *Computer*, 10(6):74–84, June 1977.

[22] Whitfield Diffie and Martin E. Hellman. New directions in cryptography invited paper, 1976.

[23] Hans Dobbertin. Cryptanalysis of md5 compress. Technical report, In Rump Session of EuroCrypt '96, 1996.

[24] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[25] Marc Fischlin. Incremental cryptography and memory checkers. In *EUROCRYPT*, pages 293–408, 1997.

[26] Philippe Flajolet and Andrew M. Odlyzko. Random mapping statistics. In *EUROCRYPT*, pages 329–354, 1989.

[27] Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14(4):636–644, 1967.

[28] Thomas Fuhr and Thomas Peyrin. Cryptanalysis of radiogatún. In *FSE*, pages 122–138, 2009.

[29] Tim Güneysu, Andy Rupp, and Stefan Spitz. Cryptanalytic time-memory tradeoffs on copacobana. In *GI Jahrestagung (2)*, pages 205–209, 2007.

[30] Martin E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

[31] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *CRYPTO*, pages 306–316, 2004.

[32] Antoine Joux. *Algorithmic Cryptanalysis*. Chapman & Hall/CRC, 2009.

[33] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In *EUROCRYPT*, pages 474–490, 2005.

[34] Dmitry Khovratovich. Two attacks on radiogatún. In *INDOCRYPT*, pages 53–66, 2008.

[35] Dmitry Khovratovich. Cryptanalysis of hash functions with structures. In *Selected Areas in Cryptography*, pages 108–125, 2009.

[36] Vlastimil Klima. Tunnels in hash functions: Md5 collisions within a minute. Cryptology ePrint Archive, Report 2006/105, 2006.

[37] L. Knudsen and B. Preneel. Enhancing the security of hash functions using non-binary error correcting codes. *IEEE Transactions on Information Theory*, 48(4):2524–2539, September 2002.

[38] Lars R. Knudsen. New potentially 'weak' keys for des and loki (extended abstract). In *EUROCRYPT*, pages 419–424, 1994.

[39] Lars R. Knudsen, John Erik Mathiassen, Frédéric Muller, and Søren S. Thomsen. Cryptanalysis of md2. *J. Cryptol.*, 23(2):72–90, 2010.

[40] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[41] Xuejia Lai and James L. Massey. Hash functions based on block ciphers. In *Advances in Cryptology - EUROCRYPT'92 Proceedings*, pages 55–70. Springer-Verlag, 1993.

[42] H. Rich Schroeppel M. Beeler, R. William Gosper. In *Hakmen, Al Memo 239*, page 64. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1972.

[43] Krystian Matusiewicz. *Analysis of Modern Dedicated Cryptographic Hash Functions.* PhD thesis, Macquarie University, 2007.

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

[45] Ralph C. Merkle. One way hash functions and des. In *CRYPTO*, pages 428–446, 1989.

[46] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford University, Stanford, CA, USA, 1979.

[47] Ilya Mironov. Hash functions: Theory, attacks, and applications. Technical report, Microsoft Research, Silicon Valley Campus, november 2005.

[48] S. Miyaguchi, M. Iwata, and K. Ohta. New 128-bit hash function. In *Proc. 4th International Joint Workshop on Computer Communications*, pages 279–288, 1989.

[49] Shoji Miyaguchi, Kazuo Ohta, and Masahiko Iwata. Confirmation that some hash functions are not collision free. In *EUROCRYPT*, pages 326–343, 1990.

[50] Judy H. Moore and Gustavus J. Simmons. Cycle structure of the des for keys having palindromic (or antipalindromic) sequences of round keys. *IEEE Trans. Software Eng.*, 13(2):262–273, 1987.

[51] Gabriel Nivasch. Cycle detection using a stack. *Inf. Process. Lett.*, 90(3):135–140, 2004.

[52] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES).* U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, October 1999.

[53] Kazuo Ohta and Kenji Koyama. Meet-in-the-middle attack on digital signature schemes. In *AUSCRYPT*, pages 140–154, 1990.

[54] J. M. Pollard. A monte carlo method for factorization. *Bit*, 15:331–334, 1975.

[55] B. Preneel. *Analysis and Design of Cryptographic Hash Functions.* Thesis (ph.d.), K. U. Leuven, Leuven, Belgium, January 1993.

[56] Bart Preneel. Cryptanalysis of message authentication codes. In *ISW*, pages 55–65, 1997.

[57] Bart Preneel. Cryptographic primitives for information authentication - state of the art. In *State of the Art in Applied Cryptography*, pages 49–104, 1997.

[58] Bart Preneel. Hash functions and mac algorithms based on block ciphers. In *IMA Int. Conf.*, pages 270–282, 1997.

[59] Bart Preneel. The state of cryptographic hash functions. In *Lectures on Data Security*, pages 158–182, 1998.

[60] Jean-Jacques Quisquater and Jean-Paul Delescaille. How easy is collision search? application to des (extended summary). In *EUROCRYPT*, pages 429–434, 1989.

[61] Vincent Rijmen and Bart Preneel. Improved characteristics for differential cryptanalysis of hash functions based on block ciphers. In *FSE*, pages 242–248, 1994.

[62] Vincent Rijmen, Bart Van Rompay, Bart Preneel, and Joos Vandewalle. Producing collisions for panama. In *FSE*, pages 37–51, 2001.

[63] R. Rivest. The md5 message-digest algorithm, 1992.

[64] Ronald L. Rivest. The md4 message digest algorithm. In *CRYPTO*, pages 303–311, 1990.

[65] Yu Sasaki, Yusuke Naito, Noboru Kunihiro, and Kazuo Ohta. Improved collision attacks on md4 and md5. *IEICE Transactions*, 90-A(1):36–47, 2007.

[66] Claus P. Schnorr and Jr. Hendrik W. Lenstra. A monte carlo factoring algorithm with linear storage. *Mathematics of Computation*, 43:289–311, 1984.

[67] Robert Sedgewick, Thomas G. Szymanski, and Andrew Chi-Chih Yao. The complexity of finding cycles in periodic functions. *SIAM J. Comput.*, 11(2):376–390, 1982.

[68] G.J. Simmons. How to insure that data acquired to verify treaty compliance are trustworthy. *Proceedings of the IEEE*, 76(5):621–627, May 1988.

[69] C.H. Mayer. J. Oseas S.M. Matyas. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, 27, 1985.

[70] François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. A time-memory tradeoff using distinguished points: New analysis & fpga results. In *CHES*, pages 593–609, 2002.

[71] Edlyn Teske. A space efficient algorithm for group structure computation. *Math. Comput.*, 67(224):1637–1663, 1998.

[72] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with application to hash functions and discrete logarithms. In *ACM Conference on Computer and Communications Security*, pages 210–218, 1994.

[73] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.

[74] X. Wang, Y. Yin, and H. Yu. Collision search attacks on sha1, February 2005.

[75] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions md4, md5, haval-128 and ripemd, 2004.

[76] Tao Xie and Dengguo Feng. How to find weak input differences for md5 collision attacks. Cryptology ePrint Archive, Report 2009/223, 2009. `http://eprint.iacr.org/`.

[77] Gideon Yuval. How to swindle rabin. *Cryptologia*, 3:187–189, 1979.

[78] Y. Zheng, T. Matsumoto, and H. Imai. Connections among several versions of one-way hash functions, 1990.

[79] Yuliang Zheng, Tsutomu Matsumoto, and Hideki Imai. Duality between two cryptographic primitives. In *AAECC*, pages 379–390, 1990.