INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

AUTHENTICATION SCHEMA

DEFYING CRACKERS' KNOWN TECHNIQUES

by

PASCAL J. ABADIE

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1997

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 9720070

Copyright 1997 by
Abadie, Pascal Jean

All rights reserved.

UMI Microform 9720070
Copyright 1997, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

© 1997

PASCAL J. ABADIE

All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This manuscript has been read an accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

12/15/96
Date

D.I.A. Cohen
Professor Daniel I.A. Cohen
Chair of Examining Committee

12/19/96
Date

Stanley Habib
Professor Stanley Habib
Executive Officer

Professor Christina Zamfirescu, Hunter College

Professor Michael Anshel, CCNY

Dr. David Cohen, National Security Administration, IDA.

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

AUTHENTICATION SCHEMA DEFYING KNOWN CRACKERS' TECHNIQUES

by

Pascal J. Abadie

Adviser: Professor Daniel I. A. Cohen

Thousands of computer users utilize systems requiring that they prove their identity in order to access specific services. This procedure, called authentication, usually requires the entering of a password unique to the user. Internet users are the fastest growing population of computer users required to authenticate themselves. Unfortunately, Internet security is very weak–any cracker, anywhere, can potentially access a computer user's Internet account by dialing a publicly available telephone number (to the provider) and then trying to figure out the user's password. As we shall see in the following paper, a large majority of users employ passwords that are easy to remember and therefore easy to guess. This provides a great opportunity for crackers to guess their passwords. In addition, it is very easy to tap a communication line (especially on the Internet), giving a cracker the opportunity to see passwords, as most Internet providers don't use secure authentication schema to grant access to their service.

Therefore, we will propose a method allowing a user to employ badly chosen passwords (passwords that are easy to remember), and still access a system securely. We are going to prove that this method defies all known crackers' techniques and uses algorithms

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that have logarithmic complexity. We will show that the password transformation has a linear complexity and that the key used for the password transformation can be created in n steps, where n is the length of the password. We will also give a secure solution for authenticating a user who employs a badly chosen password. We will compare this technique with other known authentication schema and we will show that this method is the only technique that is not weakened by the use of badly chosen keys or passwords. We will also prove that our technique resists most of the known statistical and cryptanalytical attacks (plaintext attack and chosen plaintext attack), and that it can be used in conjunction with other cryptographic methodologies to augment them with its special properties against cryptanalysis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGMENTS

The author wishes to thank his wife for her moral support.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

ACKNOWLEDGMENTS VI

TABLE OF CONTENTS VII

TABLE OF FIGURES X

CHAPTER 1 1

1. CRACKERS' TECHNIQUES, CONSIDERATIONS: 1

1.1 INTRODUCTION: 1
1.2 BAD PASSWORDS: 1
1.3 DO PEOPLE USE BAD PASSWORDS? 3
1.4 COMPUTER CRACKERS' TECHNIQUES: 4
1.5 USING THE FASTEST COMPUTER TO ENUMERATE ALL THE PASSWORDS: 5
1.6 HOW LONG SHOULD A PASSWORD BE? 6
1.6.1 Considering the fastest technology available *6*
1.6.2 Considering the average technology: *8*
1.6.3 Predicting the technology: *9*
1.7 GOOD PASSWORDS: 10

2. SOME DEFINITIONS 12

2.1 THE DIFFERENT TYPES OF CRYPTANALYTIC "ATTACKS" 14
2.1.1 The attacker has only the cyphertext: *14*
2.1.2 Known plaintext attack: *14*
2.1.3 Chosen plaintext attack: *15*
2.1.4 Chosen ciphertext attack: *15*
2.1.5 Adaptive chosen plaintext attack: *15*

3. PASSWORD TRANSFORMATION USING PERMUTATIONS 16

3.1 DEFINITION OF THE HOMOPHONIC CODE: 16
3.2 HOMOPHONIC LENGTH L IN FUNCTION OF CARDINALITY OF AN ALPHABET: 17
3.2.1 The Homophonic cardinality: *17*
3.2.2 *Finding a lower bound for H_C:* *17*
3.3 EXAMPLE OF A HOMOPHONIC CODE 18
3.4 PERMUTING HOMOPHONIC CODE 19
3.5 TRANSFORMING HOMOPHONIC WORDS INTO A PSEUDO-RANDOM BIT STRING: 19
3.6 HOW HOMOPHONIC CODES ELIMINATE SOME STATISTICAL ANALYSIS: 20
3.7 DEFINITIONS: 20
3.7.1 Permutation: *20*
3.7.2 Complexity: *21*
3.7.3 Surjection definition *22*
3.7.4 Bijection definition: *22*
3.8 GETTING A PERMUTATION ACCORDING TO A KEY: 22
3.9 GETTING A PERMUTATION USING A KEY STRING: 23
3.10 DEFINITION OF A SURJECTION F BETWEEN A SET OF KEYS AND A SET OF PERMUTATIONS: 24
3.11 DEFINITION OF THE FUNCTION W: 24
3.11.1 Example of W: *24*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.12 DEFINITION OF THE FUNCTION V: 25
3.12.1 Example: 25
3.13 DEFINITION OF F: 25
3.14 PROOF THAT F IS A SURJECTION: 27
3.15 PROOF THAT THE COMPLEXITY OF F IS LOGARITHMIC: 28
3.15.1 Complexity of V 28
3.15.2 Complexity of W 29
3.15.3 Complexity of F: 29
3.16 EXAMPLE USING F: 31
3.17 COMPLEXITY OF ENUMERATING ALL THE PERMUTATIONS CONSTRUCTED WITH THE FUNCTION F AND A SET OF KEY: 32
3.18 DEFINITION OF A BIJECTION B BETWEEN A SET OF KEYS AND A SET OF PERMUTATIONS: 33
3.18.1 Length of the permutations generated by the bijection B:: 33
3.18.2 Getting $\Theta(P_K)$: 34
3.18.3 Algorithm for finding $\Theta(P_K)$: 35
3.18.4 Algorithm of the function B:: 36
3.19 LEHMERCODE OF A PERMUTATION: 48
3.20 OBTAINING A PERMUTATION FROM LEHMERCODE IN O(N): 49
3.20.1 Algorithm and complexity: 49
3.20.2 Complexity of B: 53
3.20.3 Example of using bijection B: 53

4. HOW TO TRANSFORM A BAD PASSWORD INTO A GOOD ONE: 56

4.1 WHAT KIND OF PERMUTATIONS DEFY STATISTICAL ANALYSIS? 56
4.1.1 Statistical analysis definition: 56
4.2 NUMBER OF PERMUTATIONS OF A SET P SATISFYING PROPOSITION 6: 59
4.2.1 An upper bound of $|\mathbb{P}_n|$ 60
4.2.2 A lower bound of $|\mathbb{P}_n|$ 60
4.2.3 Characteristic function of $\mathbb{P}_n$: 65
4.2.4 Complexity of the characteristic function C: 65
4.3 EXTRACTING THE SET $\mathbb{P}$N FROM PN: 66
4.3.1 Example: 67
4.3.2 Proof that the algorithm always returns an element of $\mathbb{P}_n$. 68
4.3.3 Complexity of the GenerateDn: 68
4.4 REMARKS ON PRINCIPLE 6: 69
4.5 TRANSFORMING PASSWORDS : THE COMPLETE ALGORITHM: 70
4.5.1 Permute the letters of the password using the permutation p^+: 71
4.5.2 Permuting the bits of the password using permutation p^+: 72
4.5.3 Retrieving the original password: 72
4.5.4 The computational advantages of the password transformation: 73
4.5.5 Definitions of the key sets K_1 and K_2: 74
4.5.6 On the critical Cardinality of K_1 and K_3: 74
4.5.7 On the critical cardinality of K_2 76
4.5.8 How big should the permutation be that permutes bits to resist a plaintext attack? 76
4.5.9 Plaintext attack: 80
4.5.10 Chosen Plaintext Attack: 80
4.5.11 Chosen cyphertext attack: 81
4.5.12 Adaptive chosen plaintext attack: 82
4.5.13 Does the transformed password have to look random? 86
4.5.14 Transforming the password back to the original: 87

5. USING PASSWORD TRANSFORMATION FOR AUTHENTICATION: 88

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.1 USER DEFINITION: 88
5.2 VERIFIER DEFINITION: 88
5.3 PUBLIC KEY EXCHANGE: 89
5.4 KEEPING SECRET K_1, K_2 AND K_3: 91
5.4.1 Using Smart Cards or magnetic stripe card: 92
5.5 ALGORITHM LAYOUT: 92
5.6 AN AUTHENTICATION EXAMPLE: 94
5.6.1 Transforming the User's password: 96
5.7 RANDOM GENERATING FUNCTIONS: 99
5.8 CRYPTO: 100
5.8.1 Crypto's header: 100
5.8.2 Crypto's functions description: 102
5.9 AN EXAMPLE USING THE CRYPTO LIBRARY: 117

6. ADVANTAGE OF THE AUTHENTICATION SCHEMA OVER OTHER KNOWN ALGORITHMS: 119

6.1 SENDING PLAIN PASSWORDS: 119
6.2 SENDING PASSWORDS USING PUBLIC KEY ENCRYPTION: 119
6.3 KEY CRUNCHING: 120
6.4 WIDE-MOUTH FROG: 120
6.5 OTHER PROTOCOLS USING A TRUSTED PARTY: 121
6.6 PGP: 122
6.6.1 PGP authentication weaknesses: 123
6.7 USING ONE WAY HASHING FUNCTIONS FOR PASSWORD TRANSFORMATION: 123
6.8 KERBEROS: 124
6.8.1 The Kerberos protocol: 124
6.8.2 System weaknesses: 125
6.9 GENERAL ADVANTAGES OF OUR AUTHENTICATION SCHEME: 126
6.9.1 An apparent weakness: 126
6.9.2 Why use a personal password? 127
6.10 OTHER USES OF PASSWORD TRANSFORMATION: 128

BIBLIOGRAPHY 137

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF FIGURES

TABLE 1(BAD PASSWORDS LIST) 2
TABLE 2: PASSWORD CATEGORIES ON THE INTERNET 3
TABLE 3: ENUMERATION IN SECONDS OF WORDS OF DIFFERENT LENGTH 7
TABLE 4: ENUMERATION IN YEARS OF WORDS OF DIFFERENT LENGTH 7
TABLE 5: ENUMERATION IN YEARS OF WORDS OF VARIOUS LENGTH(AVERAGE TECHNOLOGY) 8
TABLE 6: LIST OF THE EVOLUTION OF COMPUTER SPEED 9
TABLE 7: A HOMOPHONIC ALPHABET 18
TABLE 8 : PLAIN TEXT PERMUTATION EXAMPLE 71
TABLE 9 : BIT PERMUTATION 72
TABLE 10: CRYPTO FUNCTION (FUNCTIONS DECLARED IN CRYPTO.H) 100
TABLE 11: HASH FUNCTIONS (SIZE OF THE INPUT AND THE OUTPUT) 124

FIGURE 1: JOE ACCOUNTS 4
FIGURE 2: PASSWORD LENGTH ENUMERATION (FASTEST TECHNOLOGY) 7
FIGURE 3: PASSWORD ENUMERATION (AVERAGE TECHNOLOGY) 8
FIGURE 4: EVOLUTION OF THE COMPUTER SPEED FROM 1992-1996 10

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 1

1. Crackers' techniques, considerations:

1.1 Introduction:

In order to restrict access to authorized users, computer systems and Internet providers usually require the user to login and enter a password. Unfortunately, it is very easy for a cracker to enter a computer system. After presenting computer cracker' techniques and the weak points of computer systems using passwords for security, we will propose solutions that render obsolete any technique currently employed by computer crackers.

1.2 Bad passwords:

A bad password is a password that can be easily guessed. A bad computer cracker will try an exhaustive list, starting by typing the password AAAAAA and continuing with AAAAAB, AAAAAC, and so on. This was done by Lazlo Hollyfeld in the movie *Real Genius*. Unfortunately, this technique is very bad, since trying all the possible combinations would be too time consuming for a human being.

Suppose that the minimum length of a password is 6 letters and that the alphabet used has 26 letters, then the number of passwords that can be formed with this alphabet is 26^6. If a human being can enter a password once every second (which is extremely fast) it would take him 26^6 seconds to try all the passwords of length 6. This is equivalent to $308915776 / 86400 = 3575.41$ days, in fact roughly 10 years of hard work without sleep.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

On the other hand, a computer equipped with a 120 MHz Pentium processor can perform an average of 50 MIPS. Consequently, if we suppose that such a computer can try 1 password per instruction, it would take such a computer 20 seconds to go through the exhaustive list of all the possible passwords of length 6 constructed with an alphabet of 26 letters.

Here is a list of bad passwords (extract of a list from Garfinkel, 1994):

TABLE 1: (BAD PASSWORDS LIST)

1. Your name.
2. Your spouse's name.
3. Your parents' names.
4. Your child's name.
5. Your pet's name.
6. Names of close friends or coworkers.
7. Names of your favorite fantasy characters.
8. Your boss's name.
9. Anybody's name.
10. The name of the operating system that you are using.
11. The host name of your computer.
12. Your phone number.
13. Your license plate.
14. Any part of your social security number.
15. Anybody's birth date.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

16. Other information that is easily obtained about you.
17. Words such as wizard, guru, Gandalf, and so on.
18. Any user name on the computer in any form.
19. A word in the English dictionary.
20. A word in a foreign dictionary.
21. A place.
22. A proper noun.
23. Passwords of all the same letters.
24. Simple patterns of letters such as qwerty.
25. Any of the above spelled backwards.
26. Any of the above followed or propended by a single digit.

1.3 Do people use bad passwords?

A large percentage of people use bad passwords, and many use extremely bad passwords. We can consider that an account where the user name or login and the password are the same, called a Joe account, has an extremely bad password. According to a study done on 13,797 accounts, 368 accounts (2.7 %) were Joe accounts and 7.4 % used words from a UNIX system dictionary that has 25,000 words. Therefore, it has been estimated that an attacker can expect to crack the first password in less than 2 minutes (Cooper et. al, 1995).

TABLE 2: PASSWORD CATEGORIES ON THE INTERNET

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Joe Accounts	Words from UNIX dictionary	Other
368	1020	12409

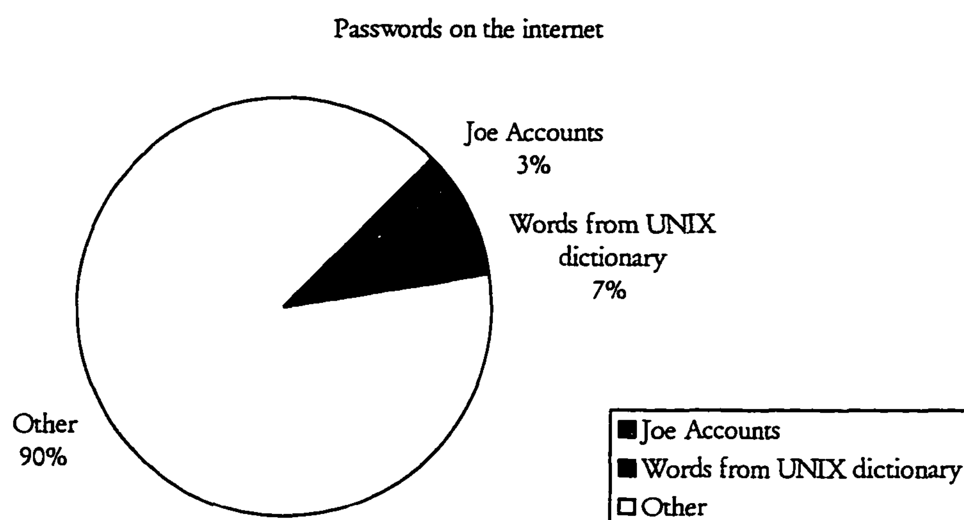

FIGURE 1: JOE ACCOUNTS

1.4 Computer crackers' techniques:

Instead of trying every combination of letters, crackers use hit lists of common passwords such as wizard or demo. Also, computer crackers believe that a significant percentage of all computers contain at least one Joe account. Most computer crackers can find an entry point into almost any system simply by checking every account to see if it is a Joe account (Garfinkel, 1994).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Also good computer crackers try all the possible bad passwords that are part of the preceding list. One of the common bad passwords is a word from the dictionary. If we take all the possible passwords of maximum length 8 formed with the English alphabet of 26 letters there are $26 * 26^2 * 26^3 * 26^4 * 26^5 * 26^6 * 26^7 * 26^8 = 26 * 676 * 17576 * 456976 * 11881376 * 308915776 * 8031810176 * 208827064576 = 217180147158$.

The largest dictionary ever published is the Oxford English Dictionary. In 1989, the second edition (OED 2) was published in 20 volumes. The OED 2 contains 22,000 pages and defines more than 500,000 words.

Thus the OED 2 contains only 0.00023% of all the possible passwords of maximum length 8. Therefore, a good cracker instead of trying all the possible password combinations would try only 0.00023% of all the possible passwords. If the computer cracker's speed is 1 password per second, it would take only $500000 / 86400 \approx 5.78$ days to try the entire

OED 2.

1.5 Using the fastest computer to enumerate all the passwords:

Intel's giant Paragon(tm) XP/S MP Supercomputers together claim the title of "World's Fastest Computer." The Paragon ran an industry-standard MP benchmark, at sustained speeds of 281 billion operations per second (281 gigaflops). This is more than 50 percent faster than the record of 170 Gflops set by Fujitsu's Numerical Wind Tunnel in August,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1994. On a second application, a double-precision complex LU factorization code, for which industry records are not routinely kept, the Intel machine reached 328 gigaflops.

How fast is 328 gigaflops?

In one second, the Paragon system accomplished what would take 11,240 years on a hand-held calculator–assuming one could do one operation per second and never take a break.

The record-breaking runs were performed by a team of scientists from Intel and Sandia National Laboratories. The Paragon system used for the record-breaking runs included 2256 compute nodes, each with three Intel i860(tm) XP microprocessors, for a total of 6,768 microprocessors. The computer nodes are a new multi-processing node architecture that the Scalable Systems Division of Intel introduced this fall.

Consequently, if we suppose that in one operation such a computer can enumerate 1 password, it can enumerate 281,000,000,000 passwords per second. Therefore, it can enumerate all the possible words of length 8 formed with an alphabet of 26 letters in 0.775 seconds.

1.6 How long should a password be?

1.6.1 Considering the fastest technology available

From a theoretical point of view, to calculate maximum possible speed, let us suppose that the Paragon computer communicates with another Paragon computer. One of the computers tries to crack the system by enumerating all the possible passwords of length n formed with an alphabet of cardinality 26. Suppose that the host computer can verify a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

password in one cycle. Therefore, how long should a password be such that the fastest computer could not enumerate/verify all the possible passwords in a reasonable time?

Length of word	8	9	10	11	12	13	14
Seconds	0.77	1 9	504	13108	340817	8861242	230392292

TABLE 3: ENUMERATION IN SECONDS OF WORDS OF DIFFERENT LENGTH

Length of word	8	9	10	1 1	12	13	14	15	16
year	0	0	0	0	0.01	0.28	7.3	190	4939

TABLE 4: ENUMERATION IN YEARS OF WORDS OF DIFFERENT LENGTH

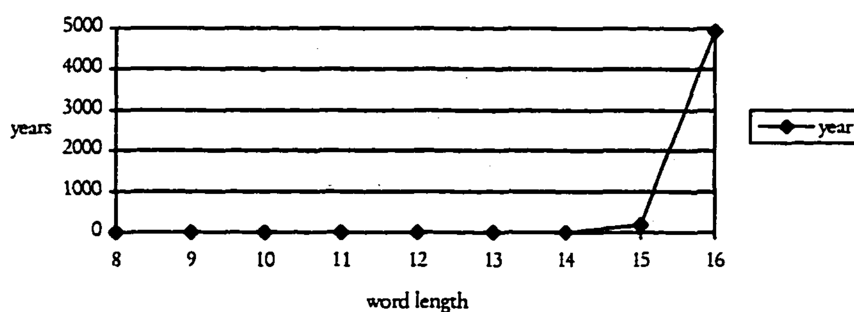

FIGURE 2: PASSWORD LENGTH ENUMERATION (FASTEST TECHNOLOGY)

The preceding tables have been computed the following way:

y = s / 31536000 where 31526000 is the approximate number of seconds during a year.

We took 365.25 as the maximum number of days that the year can have.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.6.2 Considering the average technology:

In 1996, an average personal computer is a 120 MHz Pentium that can deliver an average of 50 MIPS. Suppose that the host computer can verify a password in one cycle. Therefore, how long should a password be such that the average computer could not enumerate/verify all the possible passwords in reasonable time?

TABLE 5: ENUMERATION IN YEARS OF WORDS OF VARIOUS LENGTH(AVERAGE TECHNOLOGY)

Length of word	8	9	10	11	12
Years	0.09	2.3	60	1573	40911

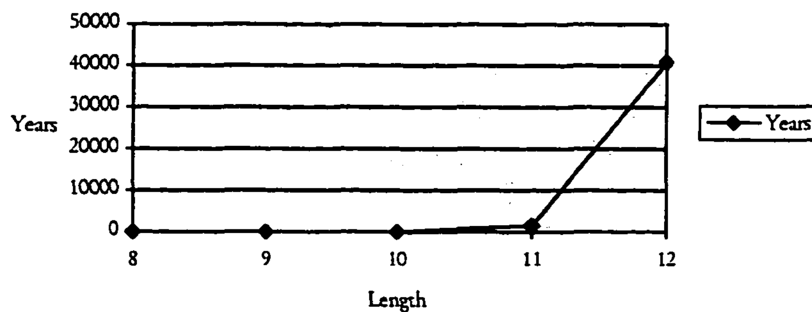

FIGURE 3: PASSWORD ENUMERATION (AVERAGE TECHNOLOGY)

It is not feasible for the fastest computer to enumerate all the passwords of length 16 in a reasonable time.

It is not feasible for the average computer to enumerate all the passwords of length 12 in a reasonable time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.6.3 Predicting the technology:

If we have an alphabet of n letters, and if we add an additional letter to the maximum number of letters of a password, the technology of the fastest computer that could enumerate all the passwords of length l constructed with an alphabet of length n, in reasonable time, would have to be multiplied by n.

According to the following table, we can see that since 1994, the speed of supercomputers has doubled each year. Consequently, if it has been decided that the length of a password is l, using an alphabet of cardinality n, such that all the possible passwords could not be enumerated in reasonable time by the fastest supercomputer, then to keep pace with the technology, one should choose a password of length l+1 every n/2 years. Suppose that we have an alphabet of 26 letters and that the length of the password has to be at least 16 bytes, then in 2008 we should use passwords of length 17. But, these are only loose predictions. If we look at Table 5 we can see that in 1996, no record has yet been established.

Year	1992	1993	1994	1995	1996
Gigaflops	30	65	140	280	280
Supercomputer	NEC SX/3	TMC CM-5	Fujitsu VPP 500	Intel Paragon	Intel Paragon
# processors	4	1024	100	6768	6768

TABLE 6: LIST OF THE EVOLUTION OF COMPUTER SPEED

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

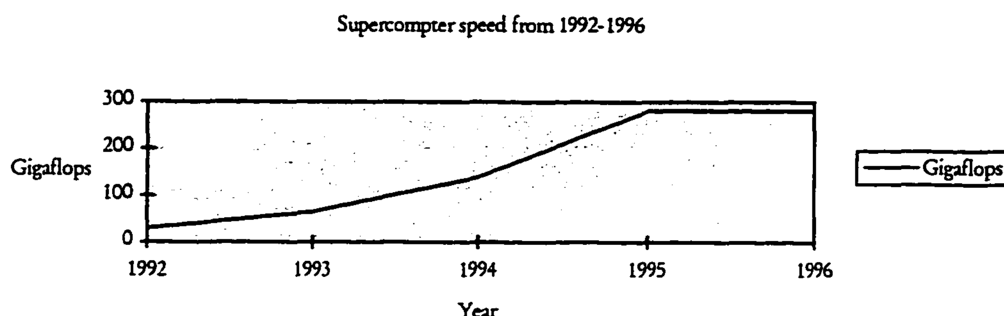

FIGURE 4: EVOLUTION OF THE COMPUTER SPEED FROM 1992-1996

1.7 Good Passwords:

In order to prevent a cracker from guessing a good password, one should use a password that is not part of Table 1. Also, in order to prevent automated programs that could use brute force to guess a password it would be safe to choose a password that has a length l such that $n^l/28*10^{10}*31471200 > 1000$ years therefore:

$l > \log_n(28*10^{10}*31471200)$

where n is the cardinality of the alphabet used to construct passwords of length l.

$28*10^{10}$ is the speed of the fastest supercomputer in floating points per second

31471200 is the number of seconds in a year (including leap year, considering that a year has 365.25 days).

Realistically, we can consider that if we include uppercase and lowercase, people will use an alphabet of cardinality 52. Therefore, the length l of the password should be 11 characters since $52^{11}/280*10^{10}*31471200 = 2306.6$ years.

Remark:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A computer keyboard has more than 100 keys, but it is safe to predict that most of the time, most people will use only the upper and lower case letters of the alphabet, which gives an alphabet of 52 letters.

Therefore, a good password should be at least 11 bytes (letters) long.

The following problem arises:

Most people choose bad passwords (see 1.3).

It is very difficult to ask people to choose a password that is not part of a hit list and that has the proper length because such a password is difficult to remember.

Now that we know how crackers proceed and what the characteristics of a good password should be, we are going to propose a method that allows users to continue to use badly chosen passwords. We are going to do so by disguising the passwords that are used. Most of the disguise techniques are part of cryptology. Therefore, the next chapter will be devoted to the definition and description of various cryptological terms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 2

2. Some definitions

In order to continue the discussion we will define some basic concepts of cryptology, cryptography and cryptanalysis in order to clarify all the terms that will be used throughout the discussion.

Cryptosystem:

A cryptosystem is a system that uses a methodology that transforms messages in such a way that only a certain number of persons can transform back them into their original version.

As stated at the following WEB site: http://www.math.uio.no/nett/faq/cryptography-faq

> The story begins: When Julius Caesar sent messages to his trusted acquaintances, he didn't trust the messengers. So he replaced every A by a D, every B by a E, and so on, through the alphabet. Only someone who knew the "shift by 3" rule could decipher his messages.
> A **cryptosystem** is usually a whole collection of algorithms. The algorithms are labeled, and these labels are called **keys**. For instance, Caesar probably used "shift by n" encryption for several different values of n. It is natural to say that n is the key here.
> The people who are supposed to be able to see through the disguise are called **recipients**. Other people are enemies, opponents, interlopers, eavesdroppers, or third parties.

Cryptography: It is the science of creating cryptosystems.

Cryptanalysis: It is the science of breaking cryptosystems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Cryptology: It is a science that takes into consideration both cryptography and cryptanalysis.

Plaintext: It is the text that has to be encrypted or enciphered.

Ciphertext: a plaintext that has to be encrypted is called a ciphertext or encrypted text.

Encryption: it is the method or algorithm used to transform a plaintext into a ciphertext.

Decryption: it is a method or algorithm used to transform a ciphertext into a plain text.

Brute force:

A a good cryptosystem would force a good cryptanalist to admit that the only way to decrypt a plaintext is to try all the possible keys in order to find the corresponding ciphertext or vice versa. In fact a good cryptosystem has to have a large key space meaning that enumerating all the possible keys cannot be done in reasonnable time.

Passwords can be considered has keys. A cracker knows that even if the theoretical password space may be large, a large percentage of people will use a predictable small password space (such as a word from the dictionary). This paper will therefore present a method that enlarges a predictable small password space into a large one where brute force cannot be used.

Basic properties of a strong cryptosystem:

The two majors securities of a strong cryptosystem are the secrecy of the key and the secrecy of the algorithm itself.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Nevertheless, a cryptosystem has to have a large key space and has to produce a cyphertext that appears random.

"Appearing random" produces large debates among cryptographers and others. Some will request that a strong cryptosystem has to appear random to all standard statistical tests (Knuth, 1981). Other will argue that the point in fact is that the encrypted text does not have to look random, as long as one cannot find the key when looking at the plaintext and the encrypted text.

2.1 The different types of cryptanalytic "attacks"

We are going to give the basic definition of cryptanalitic attack in order of difficulty.

2.1.1 The attacker has only the cyphertext:

This attack is called the cyphertext only attack.

The only information that the attacker has is an encrypted message from which to determine the plaintext. It is considered that any encryption method has to resist a cyphertext only attack since it is usually presumed possible that the attacker can have the ciphertext.

2.1.2 Known plaintext attack:

The attacker knows the plaintext and the ciphertext in order to find the encryption algorithm and the key(s) associated to it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.1.3 Chosen plaintext attack:

The attacker has the possibility of chosing a plaintext and get the corresponding ciphertext.

2.1.4 Chosen ciphertext attack:

The attacker has the possibility of chosing a ciphertext and get the corresponding plaintext.

2.1.5 Adaptive chosen plaintext attack:

According to FAQ cryptography at *http://38.250.63.3/tpatti/CRYFAQ03.HTM*

> In adaptive chosen plaintext, the attacker can determine the cyphertext of chosen plaintexts in an interactive or iterative process based on previous results. This is the general name for a method of attacking product ciphers called "differential cryptanalysis."

Now that we have defined the basic terminology and concepts of cryptology we will explain in the next chapters how it is possible to perform a secure authentication with badly chosen passwords.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 3

3. Password transformation using permutations

We will propose an algorithm that protects against badly chosen passwords. This method uses specific permutations and allows the use of any password (bad or good).

This methodology uses an old enciphering methodology updated with major changes called transposition cyphers (Sinkov, 1966). The idea is to transform a password such that it will not be part of the hit list of a cracker (see Table 1: (bad passwords list)). Before showing how the password transformation is done we must have specific tools. These tools will permit us to transform a password in logarithmic time, and we will prove that trying to retrieve the original password would take an algorithm of exponential complexity. In the following paragraphs we will define the tools that we need for password transformation.

3.1 Definition of the homophonic code:

We propose the homophonic code in order to simplify proofs and other definitions. Also a homophonic representation of plaintext is key to eliminating statistical analysis.

A code is homophonic when all the elements of a set represented by that code are represented by a bit string that has the same number of zeros and ones. The homophonic length L of a code is the minimum bit string length needed to represent all the elements of a set using a homophonic code. Note that the length L of a homophonic code must be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

an even number, that the number of zeros in a bit string are L/2, and that the number of ones in a bit string are also L/2.

3.2 Homophonic length L in function of cardinality of an alphabet:

3.2.1 The Homophonic cardinality:

Suppose that a homophonic code has a homophonic length of L, therefore the number of different bit strings of length L, called homophonic cardinality, will be $H_c = \binom{L}{L/2}$

Proof:

Counting the number of different bit strings of length L that have the same number of zeros and ones is the same as placing L/2 indistinguishable objects into L/2 + 1 boxes. Since there are $\binom{n+k-1}{k-1}$ ways to place n indistinguishable objects into k distinguishable boxes then $H_c = \binom{(L/2)+(L/2)+1-1}{(L/2)+1-1} = \binom{L}{L/2}$

Q.E.D.

Suppose that we have an alphabet of cardinality n then it will be safe to choose L = $2\lceil \mathrm{Log}_2 n \rceil$. We will show the equality by finding a lower bound for d.

3.2.2 *Finding a lower bound for H_c:*

Let us say that L = 2d, therefore d = number of zeros = number of blanks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since $\binom{2d}{d} = \binom{2d}{0}^2 + \binom{2d}{1}^2 + \binom{2d}{2}^2 + \ldots + \binom{2d}{0}^2$

and $2^d = \binom{d}{0} + \binom{d}{1} + \binom{d}{2} + \ldots + \binom{d}{d}$ therefore $\binom{2d}{d} \geq 2^d \Leftrightarrow H_c \geq 2^d$

consequently, we can choose $\text{Log}_2 n$ as a lower bound for d. Since $L/2 = d$ then we can choose $2\lceil \text{Log}_2 n \rceil$ as a lower bound for L.

3.3 Example of a homophonic code

Let's take the alphabet:

α = {A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,ƀ}. The cardinality of α is 27. Since we established that it would be enough to choose $L = 2\lceil \text{Log}_2 n \rceil$.then for the alphabet α, $L = 2\lceil \text{Log}_2 n \rceil = 2\lceil \frac{\ln 27}{\ln 2} \rceil = 2\lceil (3.3/0.7) \cong 2\lceil 4.71 \rceil = 10$. In that case $d = 5$.

Letter	Code	Letter	Code
A	0000011111	O	0001101110
B	0000101111	P	0001110011
C	0000110111	Q	0001110101
D	0000111011	R	0001110110
E	0000111101	S	0001111001
F	0000111110	T	0001111010
G	0001001111	U	0001111100
H	0001010111	V	0010001111
I	0001011011	W	0010010111
J	0001011101	X	0010011011
K	0001011110	Y	0010011101
L	0001100111	Z	0010011110
M	0001101011	ƀ	0010100111
N	0001101101		

TABLE 7: A HOMOPHONIC ALPHABET

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.4 Permuting homophonic code

Suppose that we want to permute homophonic code of length L at the bit level. Therefore we use a permutation of length L. We have L! different permutations. The number of homophonic codes of length L is $H_c = \binom{L}{L/2}$. Consequently, L! different permutations have to permute H_c.

We are going to compute the number of permutations that an element of the homophonic code will permute in the same way.

Trivially there are :

$$L!/H_c = \frac{L!}{\left(\frac{L!}{\left(L-\frac{L}{2}\right)!*\left(\frac{L}{2}\right)!}\right)} = \frac{L!}{\left(\frac{L!}{\left(\left(\frac{L}{2}\right)^2\right)!}\right)} = \left(\frac{L}{2}!\right)^2$$

3.5 Transforming Homophonic words into a pseudo-random bit string:

We call a homophonic word a word that is formed with elements of an homophonic alphabet. As an example if we use the alphabet represented in Table 5, the word WIZARD is represented by the string:

0010010111 0001011011 0010011110 0000011111 0001110110 0000111011

By definition any homophonic word is represented by a bit string that has the same number of zeros and ones.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.6 How homophonic codes eliminate some statistical analysis:

Most transposition cyphers have many weaknesses. First they don't support plaintext attacks. Another major weakness is that they hold up poorly against statistical analysis. Suppose that a user uses the password "JOHN". A cryptanalyst knowing that a transposition cipher has been used to shuffle the password "JOHN" would count the number of zero's and ones that represent the encrypted password "JOHN" since simple transposition will not change that number. Therefore, many potential passwords can be eliminated by the cryptoanalyst. The passwords that don't have the right number of zeros and ones can be easily eliminated. If we use a homophonic code to represent plaintext, the number of zeros and ones that represent any words of the same length would always be the same, making it impossible to eliminate any potential passwords.

3.7 Definitions:

3.7.1 Permutation:

A permutation of a non-finite empty set A is a bijection from X to X. We will often take X as $I_n = \{1, 2,.., n\}$ where I is the set of positive integers.

We will denote by S_n the set of all permutations from the set of integers I_n.

Therefore S_n being the set of all permutations of I_n the cardinality of S_n is n!

We shall denote permutations of a set in cycle notation(Biggs, 1990).

Example:

$I_3 = \{1,2,3\}$.

$S_3 = \{(1)(2)(3), (1)(23), (123), (12)(3), (132), (13)(2)\}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Other notation:

For simplification we will use the following notation for transpositions:

Let's take the set {1,2,3} and apply the permutation (12)(3) on it:

1 2 3

↓ ↓ ↓

2 1 3

therefore we will note the permutation (12)(3) $\Leftrightarrow$ 2,1,3

(1)(2)(3) $\Leftrightarrow$ 1,2,3

(1)(23) $\Leftrightarrow$ 1,3,2

(123) $\Leftrightarrow$ 2,3,1

(124) (3) $\Leftrightarrow$ 2,1,3

(125) $\Leftrightarrow$3,1,2

(13)(2) $\Leftrightarrow$ 3,2,1

3.7.2 Complexity:

A cipher's strength is determined by the computational power needed to break it (Garey, 1979). We will express the complexity of the algorithms using the "big O" notation. We will denote that an algorithm is linear O(n) if its complexity grows linearly with n.

We will denote that an algorithm is polynomial or logarithmic $O(n^t)$ if it needs at least n^t steps to be computed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We will denote that an algorithm is polynomial or logarithmic $O(t^{f(n)})$ if it needs at least $t^{f(n)}$ steps to be computed.

3.7.3 Surjection definition

By definition a function f from the set X to the set Y is a surjection if every y in Y is a value f(x) for at least one x in X (Biggs, 1990).

3.7.4 Bijection definition:

The function f is a bijection from X to Y if every y in Y is a value f(x) for exactly one x in X.

3.8 Getting a permutation according to a key:

In the following sections we will present 2 algorithms that will give a permutation according to a key. The reason that we give 2 algorithms is to facilitate in one case the key computation and in the other case the key exchange.

The first algorithm will give a permutation according to a key string. The second algorithm will give a key according to an integer. Later on we will see that to have a permutation of sufficient length the key has to be more than 64 bits. Currently, most personal computers don't support computations of keys more than 64 bits. Therefore we will give an algorithm that gives a permutation according to a string which can be any length and therefore can give a permutation of any length.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The drawbacks of using a string will be in exchanging the key string. Many public key exchanges are performed using integers and mathematical computations (Diffie, 1978). In general the key exchange can be performed in one unique session. If one has to exchange a key string using public key exchange one may have to perform x sessions where x is the length of the key string. On the other hand using the key string can allow any key size and consequently any length of permutation.

3.9 Getting a permutation using a key string:

We take an alphabet of cardinality n. The set of keys $K(\alpha,l)$ is the set of all possible keys of length l constructed with the alphabet α.

The cardinality of $K(\alpha,l)$ if n^l.

Since α can be an ordered set, then $K(\alpha,l)$ can also be ordered.

Example:

We take the ordered alphabet $\alpha = \{a,b\}$ where $a < b$.

We construct $K(\alpha,2)$:

$K(\alpha,2) = \{ aa, ab, ba, bb\}$ where $aa < ab < ba < bb$, or we can say that

aa is element 0 of $K(\alpha,3)$

ab is element 1 of $K(\alpha,3)$

ba is element 2 of $K(\alpha,3)$

bb is element 3 of $K(\alpha,3)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.10 Definition of a surjection F between a set of keys and a set of permutations:

Let's define the function F as a function between the 2 sets $K(\alpha, \ell)$ and P_ℓ. Later we will prove that F is a surjection.

P_ℓ will be the set of permutations constructed from the set $K(\alpha, \ell)$.

For the definition of F if we want keys of length ℓ then we will choose an alphabet of cardinality ℓ ($\ell = n$).

Before defining the function F we have to define 2 other functions $\mathcal{W}$ and [illegible].

3.11 Definition of the function $\mathcal{W}$:

$\mathcal{W}$ takes 3 arguments: the first argument is a set of keys, the second argument is a key from the set of keys and the last element is an integer.

$\mathcal{W}(K(\alpha, \ell), k_i, j)$ where α is an alphabet, $k_i \in K(\alpha, \ell)$ and $0 \leq j \leq \ell - 1$ and $\mathcal{W}$ returns the jth- 1 letter of the key k_i.

(In order to simplify the notation we will denote by K the first argument of $\mathcal{W}$.

3.11.1 Example of $\mathcal{W}$:

Let's take the alphabet $\alpha = \{a, b\}$ and ab an element of α.

$\mathcal{W}(K,ab,0) = a$ and $\mathcal{W}(K,ab,1) = b$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.12 Definition of the function $\mathcal{V}$:

[illegible] takes 2 arguments: the first argument is an alphabet and the second argument a letter of the alphabet. More generally, [illegible] takes as a first argument any ordered set.

[illegible](α, L) where α is an alphabet and L is a letter of the alphabet α and [illegible] returns the position of L in the alphabet α.

3.12.1 Example:

If we take the alphabet $\alpha = \{a, b, c, d, e, f, g, h\}$

[illegible]$(\alpha, a) = 0$, [illegible]$(\alpha, b) = 2$, [illegible]$(\alpha, h) = 7$.

Now that we have defined the functions [illegible] and [illegible] we can define [illegible].

3.13 Definition of $\mathcal{F}$:

F: K(α,[illegible]) → P[illegible].

Let's take $k_i \in$ K(α,[illegible]).

P[illegible] is the set of all the possible permutations of length [illegible].

We take the set I[illegible] = {0 , 1 , .., [illegible] - 1}.

-We take the *first* letter of k_i and we compute [illegible]([illegible](K , k_i , 0)).

By definition [illegible]([illegible](K , k_i , 0)) ∈ I[illegible] since we have the condition that the cardinality of the alphabet is the same as the length of the keys. We call [illegible]([illegible](K , k_i , 0)) = p_0.

Then we form the set I[illegible] - {p_0}.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

-We take the *second* letter of k_i and we compute $\mathcal{V}(\mathcal{W}(K, k_i, 1))$.

By definition $\mathcal{V}(\mathcal{W}(K, k_i, 1)) \in I^{\ell-1}$ or $\mathcal{V}(\mathcal{W}(K, k_i, 1)) \notin I^{\ell-1}$.

If $\mathcal{V}(\mathcal{W}(K, k_i, 1)) \in I^{\ell-1} - \{p_0\}$ then we form the set $I^{\ell-1} - \{p_0, p_1\}$ where $\mathcal{V}(\mathcal{W}(K, k_i, 1)) = p_1$.

Otherwise we take the first element of $I^{\ell-1} - \{p_0\}$ and we call it p_1 then we form the set $I^{\ell-1} - \{p_0, p_1\}$.

.

.

-We take the nth letter of k_i and we apply $\mathcal{V}(\mathcal{W}(\alpha, k_i, n))$

If $\mathcal{V}(\mathcal{W}(K, k_i, n-1)) \in I^{\ell-1} - \{p_0\}$ then we form the set $I^{\ell-1} - \{p_0, p_1, .., p_{n-1}\}$ where $\mathcal{V}(\mathcal{W}(K, k_i, n-1)) = p_{n-1}$.

Otherwise we take the first element of $I^{\ell-1} - \{p_0, p_1, .., p_{n-2}\}$ and we call it p_{n-1} then we form the set $I^{\ell-1} - \{p_0, p_1, .., p_{n-1}\}$.

.

.

-We take the ℓth-1 letter of k_i and we apply $\mathcal{V}(\mathcal{W}(K, k_i, \ell-1))$

If $\mathcal{V}(\mathcal{W}(\alpha, k_i, \ell-1)) \in I^{\ell-1} - \{p_0, p_1, .., p_{\ell-2}\}$ then we form the set $I^{\ell-1} - \{p_0, p_1, .., p_{\ell-1}\}$ where $\mathcal{V}(\mathcal{W}(\alpha, k_i, \ell-1)) = p_{\ell-1}$.

Otherwise we take the first element of $I^{\ell-1} - \{p_0, p_1, .., p_{\ell-2}\}$ and we call it $p_{\ell-1}$ then we form the set $I^{\ell-1} - \{p_0, p_1, .., p_{\ell-1}\}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Then we take the set $\{p_0, p_1,.., p_{\ell-1}\}$ and we form the permutation$\{p_0, p_1,.., p_{\ell-1}\}$ as defined in 6.1.(do that)

3.14 Proof that F is a surjection:

By definition any element of the set $K(\alpha,\ell)$ corresponds to one permutation, i.e., one transposition of length ℓ.

Therefore, it will suffice to prove that the cardinality of the set $K(\alpha,\ell)$ is greater than or equal to the cardinality of P_ℓ.

By definition the cardinality of the set $K(\alpha,\ell)$ depends of the cardinality of the alphabet α and the length ℓ of each key. Therefore the cardinality of the set $K(\alpha,\ell)$ is the number of keys of length ℓ that can be constructed from the alphabet α. Therefore $|K(\alpha,\ell)| = |\alpha|^\ell$.

By definition we have decided that the length of the key will be equal to the cardinality of the alphabet. Then, $|K(\alpha,\ell)| = |\alpha|^\ell = \ell^\ell$.

By definition a key of the set $K(\alpha,\ell)$ will give a permutation which is equivalent to a transposition of length ℓ. There are $\ell!$ such transpositions, consequently there are $\ell!$ permutations, making the cardinality of the set P_ℓ equal to $\ell!$.

Since we show that $|K(\alpha,\ell)| = \ell^\ell$ and that $|P_\ell| = \ell!$ and since by definition $\ell^\ell \geq \ell!$ then F is a surjection.

Let's prove anyway that $\forall\ \ell \geq 0$ then $\ell^\ell \geq \ell!$.

$0^0 = 0 \geq 0! = 0$.

We suppose that for $j > 0$ $j^j \geq j!$ is true.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If we can have $j^j \geq j! \Rightarrow (j+1)^{j+1} \geq (j+1)!$ then we can say that $\forall$ [illegible] ≥ 0 then [illegible] $\geq$ [illegible]!.

If $j^j \geq j!$ then $j^j * (j+1) \geq j! * (j+1)$

$j^j * (j+1) \geq j! * (j+1) \Leftrightarrow j^j * (j+1) \geq (j+1)!$

We know also that:

$(j+1)^j \geq j^j$ then

Since we have $j^j * (j+1) \geq (j+1)!$ and $(j+1)^j * (j+1) \geq j^j * (j+1)$ then

$(j+1)^j * (j+1) \geq (j+1)! \Leftrightarrow (j+1)^{j+1} \geq (j+1)!$

Therefore, we have proved that $j^j \geq j! \Rightarrow (j+1)^{j+1} \geq (j+1)!$ then $\forall$ [illegible] ≥ 0 then [illegible] $\geq$ [illegible]!.

3.15 Proof that the complexity of F is logarithmic:

To establish the complexity of F let's establish the complexity of [illegible] and [illegible]

3.15.1 Complexity of [illegible]

The function [illegible] would have to traverse at most the alphabet α that has a cardinality of [illegible].

Example:

If we take the alphabet $\alpha = \{a, b, c, d, e, f, g, h\}$

[illegible]$(\alpha, h) = 7$ therefore, in order to obtain the number 7 one could traverse the alphabet and increment a counter for each letter encountered until the appropriate letter is found.

Therefore O([illegible]) $\leq |\alpha| =$ [illegible]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Complexity of

The function would have to traverse at most a word of length .

Example:

If we take the word "badchef" constructed from the alphabet

α = {a, b, c, d, e, f, g, h}

(K,badchef,7) = a

Therefore, in order to obtain the letter a one could traverse the word badchef and increment a counter for each letter encountered until the appropriate value of the counter is reached, and the corresponding letter returned.

Therefore O( ) ≤

3.15.3 Complexity of

Let's follow the description of the algorithm for a letter of a key step by step:

(1) We take the nth letter of k_i and we apply $(\alpha, (K, k_i, n))$

In (1) we use and to obtain the integer $(\alpha, (K, k_i, n)$.

Clearly. obtaining that integer takes a complexity of O( ) + O( ) ≤ 2 .

(2) If $(\alpha, (K, k_i, n)) \in I - (p_0, p_1, ..., p_{n-1}\}$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In (2) we have to verify whether that integer belongs to a set that has a cardinality of at most [illegible] (here the set $I^{[illegible]-1}$ - ($p_0, p_1, .., p_{n-2}$}). This action also has a complexity of [illegible] since we would have to traverse at most a set of cardinality [illegible]. Now we have a complexity of

O([illegible]) + O([illegible]) + [illegible] ≤ 3[illegible]..

(3a) then we form the set $I^{[illegible]-1}$ - $\{p_0, p_1,..,p_{n-1}\}$ *where* ([illegible](K , k_i , n-1)) = p_{n-1}.

In (3a) if the integer [illegible](α , [illegible](K , k_i , n-1) belongs to the set then we have to remove that integer from a set. Clearly, this operation would take at most [illegible] steps. Now we have a complexity of O([illegible]) + O([illegible]) + [illegible] ≤ 3[illegible]..

If the integer does not belong to the set then our algorithm branches to a different complexity:

(3b) If the integer [illegible](α , [illegible](α , k_i , n-1) does not belong to the set then we have to take the first element of $I^{[illegible]-1}$ - {$p_0, p_1, .., p_{n-2}$}.

If the integer does not belong to the set then we have to take the first element of a set which has a constant complexity C and then remove that integer from the set. Clearly, this operation would take at most 2[illegible] steps.

For that branch of the algorithm, i.e., (1) (2) (3b) we have a complexity of

O([illegible]) + O([illegible]) + [illegible] + 2[illegible] ≤ 4[illegible].

This part of the algorithm will at most be applied to each letter of the key and there are [illegible] letters.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Therefore the total complexity of the algorithm is:

O( ) + O( ) + 2) ≤ (4).

Therefore O( ) = O( ) + O( ) + 2 ≤ (4) = 4^2.

So, O( ) ≤ ^{2}.

We have proved that F has a logarithmic complexity.

3.16 Example using :

Let's take the following alphabet = {a, b, c, d}.

Let's consider the following bijection:

: {a, b, c, d} ↔ {0, 1, 2, 3}

Therefore, since the cardinality of the alphabet is 4, the key has to have a length of 4.

Using the alphabet α we have the set of keys K(α, 4) which has a cardinality of 4^4 = 256.

Let's choose the key bacc of K(α, 4).

Let's compute (bacc).

(1) we have to compute (α, (K, bacc, 0)).

(K, bacc, 0) = b and (α, b) = 1.

We take the set I^3 = {0, 1, 2, 3} and we construct the set I^3 - {1} = {0, 2, 3}.

(2) we have to compute (α, (K, bacc, 1)).

(K, bacc, 1) = a and (α, b) = 0.

We construct the set I^3 - {1, 0} = {2,3}.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(3) we have to compute [illegible]$(\alpha$, [illegible]$(K$, bacc, $2))$.

[illegible]$(K$, bacc, $2) = c$ and [illegible]$(\alpha, b) = 2$.

We construct the set $I^3 - \{1, 0, 2\} = \{3\}$.

(4) we have to compute [illegible]$(\alpha$, [illegible]$(\alpha$, bacc, $3))$.

[illegible]$(K$, bacc, $3) = c$ and [illegible]$(\alpha, b) = 2$. Since c does not belong to the set $I^3 - \{1, 0, 2\}$ we take the first element of that set which is 3.

We construct the set $I^3 - \{1, 0, 2, 3\}$ which is the empty set.

(5) we take the set $\{1, 0, 2, 3\}$ and we construct the corresponding permutation (1, 0, 2, 3) which can be alternatively noted as (01) (2) (3).

3.17 Complexity of enumerating all the permutations constructed with the function F and a set of key:

In 3.14 we proved that F is a surjection. We also showed that the cardinality of the set of keys $K(\alpha$,[illegible]$)$ = [illegible] and that [illegible] $K(\alpha$,[illegible]$)) = P$[illegible] has a cardinality of [illegible]!.

The number n! can be approximated by computing $s_n = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ where

$n! \sim s_n$.

This approximation is very good since it already approximates 5! Since $5! = 120$ and $s_n = 118.02$. (also $10! = 3{,}628{,}000$ and $s_n = 3{,}598{,}600$).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ we can clearly see that enumerating permutations is exponential and that the enumeration has a complexity of $(n/e)^n$.

Therefore the complexity of enumerating of all the permutations obtained from a set of keys of cardinality [illegible] has a complexity of at least ([illegible]/e)[illegible] which is an exponential complexity.

3.18 Definition of a bijection B between a set of keys and a set of permutations:

Now we will define a function that is a bijection between the 2 sets $K(\alpha_{y}$[illegible]$)$ and P_K, P_K being a set of permutations.

Each key will correspond to one and only one permutation, unlike the preceding algorithm.

Therefore the complexity of enumerating the number of permutations constructed with such a function is the same as the enumeration of all the keys that belong to the set used to construct the set of permutations.

3.18.1 Length of the permutations generated by the bijection [illegible]:

Let's note $\Theta(P_K)$ being the unique length of all the permutations of P_K.

Since we want a bijection between the sets $K(\alpha_{y}$[illegible]$)$ and P_K then the permutations of the set P[illegible] have to be such that

$(\Theta(P_K) - 1)! < |K(\alpha_{y}$[illegible]$)| \leq \Theta(P_K)!$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proof:

Suppose that $\Theta(P_K)! < |K(\alpha,l)|$ therefore the cardinality of P_K is less than the cardinality of the set of keys making the bijection between the 2 sets impossible.

Example:

Suppose that the keys are constructed with an alphabet of cardinality 4, then the cardinality of all the possible keys is = 256.

Consequently, since $5! = 120 < 4^4 < 6! = 720$ the bijection B has to correspond to a set of permutations such that all the permutations of the set are of length 6.

If the length of the permutation is less than 6 then the cardinality of the set of permutations is less than the set of keys, making a bijection of the 2 sets trivially impossible.

3.18.2 Getting $\Theta(P_K)$:

$\Theta(P_K)$ can be computed with linear complexity.

Here is the function GetLengthPermutationSet that returns $\Theta(P)$ according to the cardinality of the alphabet used and the length of a key constructed with that alphabet.

```
ULONG GetLengthPermutationSet
(
      ULONG ulAlphabetCadinality,
      ULONG ulKeyLength
)
{
     ULONG ulCounter;
     double dKeyCardinality = 1;
     double dLengthSetPermutation = 1;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
	for(ulCounter=0;ulCounter<ulKeyLength;ulCounter++)
	dKeyCardinality=dKeyCardinality*ulAlphabetCardinality;
	ulCounter = 1;
	do
	{
	dLengthSetPermutation=dLengthSetPermutation*++ulCounter;
	}while(dLengthSetPermutation < dKeyCardinality);
	return ulCounter;
}
```

We can see that the first *for* loop is used to compute the cardinality of the set of keys. The *while* loop iterates until the theoretical cardinality of the set of permutations exceeds the cardinality of the set of keys.

Then the number of iterations is returned which corresponds to the cardinality of the set of permutations. We clearly see that the number of iterations of the loop is the length of the permutations therefore the function proposed has a linear complexity.

3.18.3 Algorithm for finding $\Theta(P_K)$:

Once $\Theta(P)$ has been found then a permutation from an integer has to be found.

Let's call $\Theta(P) = o$ where $o > 0$.

Let's define the function $C: K \Leftrightarrow N$

Suppose that we use an alphabet of cardinality m and that the key length is L and suppose that we take a key $k = a_1a_2..a_l$ where a_i is an element of the set of keys for I greater than or equal to 1 and less than or equal to L, then we compute:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$C(k) = C(a_1a_2..a_l) =$ [illegible]$(\alpha, a_l)*m^0 +$ [illegible]$(\alpha, a_{l-1})*m^1 + .. +$ [illegible]$(\alpha, a_1)*m^{L-1}$

Example:

$\alpha = \{a, b, c\}$

m = 4 and L = 5.

k = abaca.

Then $C(k) = C(abaca) = 0*5^4 + 1*5^3 + 0*5^2 + 2*5^1 + 2*5^0 = 0 + 75 + 0 + 10 + 0 =$ 85.

Since the permutation that has to be found according to an integer is of length o then o iterations have to be executed.

3.18.4 Algorithm of the function [illegible]

Let's take a key k from a set of key K. Suppose that the set of permutations corresponding to the set of key K is of length n, let's compute [illegible](k) = $p_0p_1..p_{n-1}$.

In all the definitions below we will choose n greater than 1 (meaning that we will only consider permutations of length 2 or greater).

Here is the algorithm that will give the unique permutation.

First we are going to compute q_0, q_1 .. q_n in the following way:

Iteration 0:

We have $r_0 = C(k) - q_0(n - 1)!$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Where $q_0 = \left\lfloor \frac{C(k)}{(n-1)!} \right\rfloor$.

In fact $r_0 = C(k) - \left\lfloor \frac{C(k)}{(n-1)!} \right\rfloor (n-1)!$

Iteration 1:

We have $r_1 = r_0 - q_1(n-2)!$

Where $q_1 = \left\lfloor \frac{r_0}{(n-2)!} \right\rfloor$

ith iteration:

We have $r_i = r_{i-1} - q_i(n-i-1)!$

Where $q_i = \left\lfloor \frac{r_{i-1}}{(n-i-1)!} \right\rfloor$

Last iteration:

$q_{n-1} = 0$;

Therefore we have the following set $Q = \{q_0, q_1, .., q_{n-1}\}$

Now we have to find the corresponding permutation $p_0p_1..p_n$ corresponding to the key k.

Using the function see 3.12 we will use at iteration 0 the following sets:

$S = \{0, 1,.., n-1\}$ and $Q = \{q_0, q_1, .., q_{n-1}\}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Iteration 0:

$p_0 = (S, (Q, 0)) = (S, q_0)$

then we form the set $S_0 = Q - \{q_0\}$

Iteration 1:

$p_1 = (S_0, (Q, 1)) = (S_0, q_1)$

then we form the set $S_1 = Q - \{q_0, q_1\}$

Iteration i :

$p_i = (S_{i-1}, (Q, 2)) = (S_{i-1}, q_i)$

then we form the set $S_i = Q - \{q_0, q_1, .., q_{i-1}\}$

Last iteration:

$p_{n-1} = (S_{n-2}, (Q, n-1)) = (S_{n-2}, q_{n-1})$

In fact for the last iteration there is only one element in the set S_{n-2} since the set has a cardinality of n.

Therefore p_{n-1} will always be the last element of the set S_{n-2}.

Before we prove that B is a bijection, let's establish the following propositions:

Proposition 1:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Suppose that for 2 keys k and k' where $k, k' \in K$ and that the length of the corresponding set of permutations is n, we have the following 2 iterations:

For k:

$$r_0 = C(k) - q_0(n-1)! \text{ where } q_0 = \left\lfloor \frac{C(k)}{(n-1)!} \right\rfloor.$$

$$r_i = r_{i-1} - q_i(n-1)! \text{ where } 0 \leq i \leq n\text{-}1 \text{ and } q_i = \left\lfloor \frac{r_{i-1}}{(n-i-1)!} \right\rfloor$$

For k':

$$r'_0 = C(k) - q'_0(n-1)! \text{ where } q'_0 = \left\lfloor \frac{C(k')}{(n-1)!} \right\rfloor.$$

$$r'_i = r'_{i-1} - q'_i(n-1)! \text{ where } 0 \leq i \leq n\text{-}1 \text{ and } q'_i = \left\lfloor \frac{r'_{i-1}}{(n-i-1)!} \right\rfloor$$

We have the following proposition:

If there exists i such that $0 \leq i \leq n\text{-}1$ and $r_i \# r'_i$ then $B(k) \# B(k')$.

Proof of proposition 1:

Taking all the conditions defined in proposition 1, if $r_i \# r'_i$ then we have two cases:

(1) $q_i \# q'_i$ then $p_i \# p'_i$ consequently $B(k) \# B(k')$.

(2) $q_i = q'_i$ then by definition $r_{i+1} \# r'_{i+1}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In that case, at one point if we continue the iteration to (n - 1) we are going to show that the condition:

"there exists j where $q_j \# q'_j$ and $i < j \leq n\text{-}1$" is true.

Proof by contradiction:

Suppose that until iteration n - 1 we always have the condition $q_j = q'_j$.

Let's see what happens at iteration (n - 2).

In that case:

$$q_{n-2} = \left\lfloor \frac{r_{i-1}}{(n - n + 2 - 1)!} \right\rfloor = r_{n-2} \text{ and } q'_{n-2} = \left\lfloor \frac{r'_{i-1}}{(n - n + 2 - 1)!} \right\rfloor = r'_{n-2}$$

we have the condition:

$q_{n-3} = q'_{n-3}$ therefore $r_{n-2} \# r'_{n-2}$ consequently $r_{n-1} \# r'_{n-2}$.

In that case $(q_{n-2} \# q'_{n-2}) \rightarrow (p_{n-2} \# p'_{n-2}) \rightarrow (B(k) \# B(k'))$.

Q.E.D.

Proposition 2:

$\forall\, i > 0$ and $i \leq n - 1$ we have $r_i \leq r_0$.

Proof of proposition 2:

In spite of the triviality of proposition 2 we will prove it by a recursive proof.

Let us see if $r_1 \leq r_0$ is true.

$r_1 = r_0 - q_1(n\text{-}2)!$ and $r_0 = C(k) - q_0(n\text{-}1)!$

note: by definition $\forall\, i > 0$ and $i \leq n - 1$, $q_i \geq 0$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$r_1 = C(k) - q_0(n\text{-}1)! - q_1(n\text{-}2)! = C(k) - (q_0(n\text{-}1)! + q_1(n\text{-}2)!)$.

According to the above note

$(q_0(n\text{-}1)! + q_1(n\text{-}2)!) \geq q_0(n\text{-}1)! \rightarrow C(k) - (q_0(n\text{-}1)! + q_1(n\text{-}2)!) \geq C(k) - q_0(n\text{-}1)! \rightarrow r_1 \leq r_0$

Since r_1 is true then we suppose that $r_j \leq r_0$ for $j > 0$ and $j < n - 1$.

Let's prove that if $r_j \leq r_0$ is true this implies $r_{j+1} \leq r_0$ is true.

By definition $r_{j+1} = r_j - q_{i+1}(n - j - 2)$ and $q_{i+1}(n - j - 2) \geq 0$.

Since we have $r_j \leq r_0$ then $r_j - q_{i+1}(n - j - 2) \leq r_0$ therefore $r_{j+1} \leq r_0$.

Since we proved that if $r_j \leq r_0$ is true that implies $r_{j+1} \leq r_0$ is true for

$j > 0$ and $j < (n - 1)$, then proposition 2 is true.

Q.E.D.

Proposition 3:

$\forall$ i where $0 \leq i \leq n - 1$ and $\forall$ $k \in K$, if $C(k) < (n - i - 1)!$ then $q_i = 0$.

Proof of proposition 3:

If $C(k) < (n - i - 1)!$ then $C(k) < (n - i)! < .. < (n - 1)!$ therefore $q_0 \geq q_1 \geq .. \geq q_i$ and $q_0 =$

0 therefore $q_i = 0$.

Q.E.D.

Proposition 4:

$\forall$ $k \in K$ such that $C(k) > 0$ then $\exists$ i where $0 < i \leq (n - 1)$ such that $qi > 0$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proof of proposition 4:

Proposition 4 can be proved by the fact that in the worst case

$q_0, q_1, .., q_{n-3} = 0$ and $r_0, r_1, .., r_{n-3} = C(k)$

At one point we will have the condition:

$$i = (n - 2) \text{ then } q_{n-2} = \left\lfloor \frac{r_{n-3}}{(n-(n - 2) - 1)!} \right\rfloor = \left\lfloor \frac{r_{n-3}}{1} \right\rfloor = C(k)$$

Since $C(k) > 0$ then $q_{n-2} > 0$

Q.E.D.

Proof that B is a bijection:

To prove that B is a bijection we have to prove that

(a) If $:(k_1) = :(k_2) \rightarrow k_1 = k_2$

(b) If $k_1 = k_2 \rightarrow :(k_1) = :(k_2)$.

Proof by contradiction:

We are going to suppose that:

$\exists\, k_1, k_2 \in K$ and $k_1 \# k_2$ such that $:(k_1) = :(k_2)$

Suppose that $:(k_1) = :(k_2) = p_0 p_1 .. p_{n-1}$

By definition if $k_1 \# k_2$ then $C(k_1) \# C(k_2)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Therefore we have to compute $q_0, q_1 .. q_{n-1}$ for k_1.

Remember the first iteration:

$$r_0 = C(k_1) - q_0(n - 1)! \text{ where } q_0 = \left\lfloor \frac{C(k_1)}{(n - 1)!} \right\rfloor.$$

We are going to examine different cases relative to C(k).

We have to keep in mind that in all the cases if $C(k_1) \geq (n - 1)!$ then

Case 1: $C(k_1) \geq (n - 1)!$ and $C(k_2) < (n - 1)!$:

If $C(k_1) > (n - 1)!$ therefore by definition $q_0 = \left\lfloor \frac{C(k_1)}{(n - 1)!} \right\rfloor > 0$.

If $C(k_2) < (n - 1)!$ therefore $q'_0 = \left\lfloor \frac{C(k_2)}{(n - 1)!} \right\rfloor = 0$.

In that case we clearly see that q_0 and q'_0 are different.

Therefore $p_0 = W(S, q_0)$ is different from $p'_0 = W(S, q'_0)$

Therefore, since we have [illegible]$(k_1) = p_0p_1..p_{n-1}$ and [illegible]$(k_2) = p'_0p'_2..p'_{n-1}$ in the preceding case,

p_0 and p'_0 are different which is equivalent of [illegible](k_1) and [illegible](k_2) are different.

Case 2: $C(k_1) \geq (n - 1)!$ and $C(k_2) \geq (n - 1)!$ we have to keep in mind that C(k) # C(k').

We know already that $q_0 > 0$

If $C(k_2) > (n - 1)!$ therefore $q'_0 = \left\lfloor \frac{C(k_2)}{(n - 1)!} \right\rfloor > 0$.

Since k_1 # k_2 are different then $C(k_1)$ # $C(k_2)$.

Since $C(k_1)$ # $C(k_2)$ then q_0 # q'_0.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Therefore, if we have $B(k_1) = p_0p_1..p_{Ord}$ and $B(k_2) = p'_0p'_2..p'_{Ord}$ in the preceding case, p_0 and p'_0 are different which is the equivalent of $B(k_1)$ and $B(k_2)$ being different.

In fact $r_0 = C(k) - \left\lfloor \frac{C(k)}{(n-1)!} \right\rfloor (n-1)!$

Case 3: $C(k_1) < (n-1)!$ and $C(k_2) \geq (n-1)!$.

In that case trivially $q_0 = 0$ therefore $r_0 = C(k1)$.

If $C(k_1) < (n-1)!$ then $q'_0 > 0$ which are the same conditions as case 1.

Therefore $B(k_1) \# B(k_2)$.

Case 4: $C(k') < (n-1)!$ and $C(k') \# 0$ and $C(k) < (n-1)!$ and $C(k) \# 0$

Since $C(k) < (n-1)!$ then $q_0 = \left\lfloor \frac{C(k)}{(n-1)!} \right\rfloor = 0$.

Since $C(k') < (n-1)!$ then $q'_0 = \left\lfloor \frac{C(k')}{(n-1)!} \right\rfloor = 0$.

Therefore we have $q_0 = q'_0$

According to proposition 4:

$\exists$ i such that $0 < i < (n-1)$ and $q_i > 0$.

Since $q_0 = 0$ we can say that (taking proposition 4 into consideration):

$\exists$ i such that $0 < i < (n-1)$ and $q_0, q_1,..,q_i > 0$.

we can say the same thing for i':

$\exists$ i' such that $0 < i' < (n-1)$ and $q'_0, q'_1,..,q'_{i'} > 0$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Let's consider such i and i':

By definition:

$r_0 = C(k)$ since $q_0 = 0$

$r_1 = r_0 = C(k)$

.

.

$r_{i-1} = r_{i-2} = C(k)$.

By definition:

$r'_0 = C(k)$ since $q_0 = 0$

$r'_1 = r_0 = C(k')$

.

.

$r'_{i-1} = r_{i-2} = C(k')$.

Therefore:

$$q_i = \left\lfloor \frac{C(k)}{(n - i - 1)!} \right\rfloor = 0 \text{ and } q_{i'} = \left\lfloor \frac{C(k')}{(n - i - 1)!} \right\rfloor = 0.$$

- Suppose that $i = i'$ in that case $r_{i-1} = C(k)$ and $r'_{i-1} = C(k')$ then $r_{i-1} \# r'_{i-1}$.

Consequently, according to proposition 1 since $r_{i-1} \# r'_{i-1}$ then $B(k) \# B(k')$.

- Suppose that $i \# i'$ and $i < i'$.

In that case $q_i > 0$ and $q'i = 0$.

Therefore $r_{i-1} = C(k)$ and $r'_{i-1} = C(k')$ then $r_{i-1} \# r'_{i-1}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Consequently according to proposition 1, since r_{i-1} # r'_{i-1} then B(k) # B(k').

- Suppose that i # i' and i > i'

(using the same trivial reasoning as in i # i' and i < i').

Case 5: C(k') < (n - 1)! and C(k') = 0 and C(k) < (n - 1)! and C(k) # 0

If C(k) = 0 then q0, q1, ..qn-1 = 0.

If C(k') = 0 then according to proposition 4:

$\exists$ i' such that 0 < i' < (n - 1) and $q'_{i'} > 0$.

therefore since qi' = 0 and $q'_{i'} > 0$ then B(k) # B(k').

(a) If [illegible]$(k_1) =$ [illegible]$(k_2) \rightarrow k_1 = k_2$

To prove (a) we suppose that B(k) = p1p2..pn-1 and B(k') = p1p2..pn-1 where n is the length of the permutations.

If B(k) $\neq$ B(k') then $\exists$ 1 $\leq$ i < n -1 such that pi $\neq$ p'i. If pi $\neq$ p'i.

Since $p_i =$ [illegible](S_i, q_i) and $p'_i =$ [illegible](S'_i, q'_i) which means that if S_i = S'i then qi $\neq$ q'i and if $S_i \neq$ S'i then $\exists$ 1 $\leq$ j < i such that qj $\neq$ q'j.

In any case we can say that $\exists$ 1 $\leq$ j $\leq$ i such that qj $\neq$ q'j.

Therefore since qj $\neq$ q'j and $q_j = \left\lfloor \frac{r_{j-1}}{(n - j - 1)!} \right\rfloor$ and $q'_j = \left\lfloor \frac{r'_{j-1}}{(n - j - 1)!} \right\rfloor$

then rj-1 $\neq$ r'j-1.

To further continue the proof we are going to prove the following proposition:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proposition 5:

if $r_i \neq r'_i$ and $i > 1$ then $r_{i-1} \neq r'_{i-1}$.

proof of the proposition:

We know that

$ri = r_{i-1} - q_i(n - i - 1)!$

$r'_i = r'_{i-1} - q'i(n - i - 1)!$

We suppose that $r_{i-1} = r'_{i-1}$ then since $r_i != r'_{i-1}$ then $q_i != q'_i$.

Since $q_i = \left\lfloor \frac{ri-1}{(n - i - 1)!} \right\rfloor$ and $q'_i = \left\lfloor \frac{r'i-1}{(n - i - 1)!} \right\rfloor$ and $q_i \neq q'_i$ then $r_{i-1} != r'_{i-1}$ which is a contradiction, therefore $r_{i-1} \neq r'_{i-1}$.

Q.E.D.

We continue the preceding proof.

Remember that we are in the case where $r_i != r'_i$ therefore according to proposition 5, we have $r_{i-1} != r'_{i-1}, .., r_0 \neq r'_0$.

By definition we have:

$r_0 = C(k) - q_0(n-1)!$ and $r'_0 = C(k') - q'_0(n-1)!$

Let us suppose that $C(k) = C(k')$ therefore since $r_0 \neq r'_0$ therefore $q_0 \neq q'_0$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since $q_0 = \left\lfloor \frac{C(k)}{(n-1)!} \right\rfloor$ and $q'_0 = \left\lfloor \frac{C(k')}{(n-1)!} \right\rfloor$ therefore $C(k) \neq C(k')$ which is a contradiction, therefore $C(k) \neq C(k')$ which is equivalent to $k \neq k'$ since C is a bijection.

Q.E.D.

This finalizes the proof that B is a bijection.

3.19 Lehmercode of a permutation:

The preceding algorithm gives the ith permutation of an index of permutation writen in lexicographic order (Biggs, 1988).

In the algorithm we obtain the set {q0,q1..qi-2} which is called the Lehmercode of a permutation. The Lehmercode of a permutation is obtained by the representation of an integer j, element of the symmetric group n = {1,..,n-1} in base n!

For example, when we follow the preceding algorithm for a symmetric group of cardina;ity 4, if we compute B(15) we will have q0 = 2, q1 = 1, q3 = 1 where we can write

15 = q0.3! + q1.2! + q0.1.

Then, obtaining the corresponding permutation of a Lehmercode, we use function V. Algorithms have been proposed to obtain the corresponding permutation of a Lehmercode. All the algorithms proposed (Flajolet, 1996) consist of the search and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

traversal of a set of cardinality ≤ n where n is the symmetric group {0,..,n-1} at least n-1 times bringing the complexity of the algorithm to $O(n^2)$.

In this section we are going to give an algorithm that obtains the corresponding permutation of a Lehmercode in O(n), which is a dramatic improvement.

3.20 Obtaining a permutation from Lehmercode in O(n):

3.20.1 Algorithm and complexity:

1) First we represent the set $I^n = \{1,..,n\}$ in base (n + 1) the following way:

$$PermBase = c = 1 * (n + 1)^{n-1} + 2*(n + 1)^{n-2} + .. + i * (n+1)^{n-i} + .. + n * (n+1)^0$$

Trivially, for any positive integer n there is one and only one representation of the set I^n.

The following C function computes the PermBase of the set I^n:

```
void GetPermBase(__int64 n, __int64 *PermBase)
{
   __int64 Base = n + 1;
   *PermBase = n;
   int x;
   for(x = (n - 1); x > 0; x--)
   {
      *PermBase = (x * Base) + (*PermBase);
      Base = Base * (n + 1);

   }
}
```

Clearly the complexity of the function is O(n) since we have to perform a constant number of arithmetic operations in a *for* loop that iterates (n - 1) times.

2) we create the following array:

0	....	i	...	n-1
(n+1)		$(n+1)^{i+1}$		$(n+1)^n$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The following C code performs the initialization of the array of integers Baseth:

```
__int64 *Baseth;
Baseth = (__int64 *) malloc (sizeof(__int64) * n);
Baseth[0] = n + 1;
int x;
for(x = 1; x < n; x++)
     Baseth[x] = Baseth[x - 1] * (n + 1);
```

The complexity of the preceding function is O(n) since it contains only a for loop with (n - 1) iterations.

3) we perform the following operation that extracts the permutation from PermBase:

```
__int64 i, y = 1, a, b, c = PermBase, p;
for(x = 0; x < n - 1; x++)
{
     i = (Baseth[n - LehmerCode[x] - 1] / y);
     a = c % i;
     b = c % (i / (n + 1));
     p = (a - b) / (i / (n + 1));
     Permutation[x] = p;
     c = ((c - a) / (n + 1)) + b;
     y = y * (n + 1);
}
Permutation[x] = c;
```

In the preceding piece of code LehmerCode is an array of integers where the Lehmercode of the lexicographic order of a permutation of the symmetric group n has been stored.

Remember:

PermBase = c = $1 * (n + 1)^{n-1} + 2*(n + 1)^{n-2} + .. + i * (n+1)^{n-i} + .. + n * (n+1)^0$

which can be writen in base (n + 1) as the following vector: [1,2,..,n].

The Lehmercode is represented by an array of n - 1 elements.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To compute the permutation we use the value of the first element of the Lehmercode and so on until all the elements of the Lehmercode have been used.

Therefore when LehmerCode[x] = j we will get the jth value v_j of the vector represented by c, i.e we extract the factor v_j of $(n + 1)^{n-1-LehmerCode[x]}$ from c the following way:

we compute $c \bmod (n + 1)^{n-LehmerCode[x]} - c \bmod ((n + 1)^{n-1-LehmerCode[x]} / (n + 1)) =$

$v_j(n + 1)^{n-1-LehmerCode[x]}$

then we compute $v_j(n + 1)^{n-1-LehmerCode[x]} / (n + 1)^{n-1-LehmerCode[x]} = v_j$

Then c will be computed to represent the preceding vector without the jth value v_j (the first value of a vector is 0) the following way:

$c = (c - (c \bmod (n + 1)^{n-LehmerCode[x]})/ (n + 1) + c \bmod ((n + 1)^{n-1-LehmerCode[x]} / (n + 1))$

Then we use the next element of the Lehmercode to get the value of the new vector represented by c and so on until all the elements of the Lehmercode have been exhausted. Since the powers of (n + 1) have been stored initially in an array of size (n - 1) they can be accessed in one step in order to extract the values of the vectors represented by c

Example:

First iteration:

PermBase = c = $3 * 1 + 2 * 4 + 1 * 4^2$

Here n = 3, if LehmerCode[0] = 1 then we compute

$c \bmod 4^{(3-1)} = 3 * 1 + 2 * 4$ and we compute

$c \bmod 4 = 3 * 1$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Then we compute $p = ((c \bmod 4^2) - (c \bmod 4))/4^2 = 2$ then we compute a new c:

$c = ((3 * 1 + 2 * 4 + 1 * 4^2)/4) - (3 * 1 + 2 * 4) + 3 * 4 = 3*1 + 1 * 4$

we remember $p = 2$.

Second iteration:

If LehmerCode[1] = 0 then we compute $p= 1$ and c is represented as $3 * 1$

Last iteration:

$p = c$.

At each iteration p builds the permutation array : 2 1 3.

Therefore the Lehmercode (2 , 0) gives the corresponding permutation 2 1 3

Consequently, according to the preceding algorithm we can see clearly that we compute a constant number of arithmetic operations n -1 times. Therefore the complexity of the algorithm is O(n).

Remark:

The preceding algorithm is bound to the fact that we will represent permutations of length n starting at integer 1 to n. Trivially the algorithm can represent permutations of length n starting at integer 0 to n-1 without any major modifications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.20.2 Complexity of :

We proved that the complexity for obtaining the Lehmercode of a permutation according its lexicographic position has a complexity of O(n) and we proved that obtaining the corresponding permutation from a Lehmercode has a complexity of O(n) therefore the bijection B has a complexity of O(n).

3.20.3 Example of using bijection :

F: $K(\alpha_r) \rightarrow P_K$

The length of the key is $= 4$.

The alphabet $\alpha = \{a, b, c, d\}$.

Therefore the cardinality of P_4 is $4^4 = 256$.

Since $5! < 256 < 6!$ Then the permutations of P_4 are of length 6.

Let's take the key abbc. We know that $(\alpha, a) = 0$, $(\alpha, b) = 1$ and $(\alpha, c) = 2$.

$C(abca) = 0*4^3 + 1*4^2 + 1*4^1 + 2*4^0 = 22$.

Since we have $3! < 22 < 4!$. We take $n = 4$.

Step 1

We compute $\left\lfloor \frac{C(abbc)}{(Length(P4) - 1)!} \right\rfloor = \left\lfloor \frac{22}{5!} \right\rfloor = 0 = q_0$.

Then we compute $C(abbc) - \left\lfloor \frac{C(abbc)}{(Length(P4) - 1)!} \right\rfloor = 22 = r_0$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since $C(abbc) = q_0 * (Length(p_4) - 1)! + C(abba) \Leftrightarrow 22 = 0 * 5! + 22$

then we take [illegible](I_5, q_0) = [illegible]$(\{0, 1, 2, 3, 4, 5\}, 0) = 0 = p_0$.

Step 2

We compute $\left\lfloor \frac{r_0}{(Length(P_4) - 2)!} \right\rfloor = \left\lfloor \frac{22}{4!} \right\rfloor = 0 = q_1$.

Then we compute $r_0 - (\left\lfloor \frac{r_0}{(Length(P_4) - 2)!} \right\rfloor * (Length(P_4) - 2)!) = 22 = r_1$.

Since $22 = 0 * 4! + 22$ then we take [illegible]$(\{1, 2, 3, 4, 5\}, 0) = 1 = p_1$.

Step 3

We compute $\left\lfloor \frac{r_1}{(Length(P_4) - 3)!} \right\rfloor = \left\lfloor \frac{22}{3!} \right\rfloor = 3 = q_2$.

Then we compute $r_1 - \left\lfloor \frac{r_1}{(Length(P_4) - 3)!} \right\rfloor * (Length(P_4) - 3)!) = 4 = r_2$.

Since $22 = 3 * 3! + 4$ then we take [illegible]$(\{2, 3, 4, 5\}, 3) = 5 = p_2$.

Step 4

We compute $\left\lfloor \frac{r_2}{(Length(P_4) - 4)!} \right\rfloor = \left\lfloor \frac{4}{2!} \right\rfloor = 2 = q_3$.

Then we compute $r_3 - \left\lfloor \frac{r_2}{(Length(P_4) - 4)!} \right\rfloor * (Length(P_4) - 4)!) = 0 = r_3$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since 4 = 2 * 2! + 0 then we take [illegible]({2, 3, 4}, 2) = 4 = p_3

Step 4

We compute $\left\lfloor \frac{r_3}{(Length(P_4) - 5)!} \right\rfloor = \left\lfloor \frac{0}{1!} \right\rfloor = 0 = q_4$.

Then we compute r_4 - $\left\lfloor \frac{r_3}{(Length(P_4) - 5)!} \right\rfloor$ * (Length(P_4) - 5)!) = 0 = r_4.

Since 0 = 0 * 1! + 0 then we take [illegible]({2, 3}, 0) = 2 = p_4.

Step 5:

By default p_5 = 3.

Solution:

Therefore the corresponding permutation of the key abbc is 015423 = (0) (1) (2534).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 4

4. How to transform a bad password into a good one:

In chapter 3 we defined all the tools that will permit us to transform badly chosen passwords into good ones. Remember that we have two basic tools: a homophonic code (see 3.1) to represent words or passwords, and functions that will give us a permutation in logarithmic time according to a key (see 3.13 and 3.18).

In this chapter we are going to describe how to transform a password such that the password does not belong to the best cracker's hit list. We are also going to prove that the password transformation defies statistical analysis and also resists plain text attack and chosen plaintext attack.

4.1 What kind of permutations defy statistical analysis?

In order to transform a password into a word that defies statistical analysis we will chose a specific permutation that we will use later on to transform the password. Therefore in the next section we define what permutation to choose and how to choose it.

4.1.1 Statistical analysis definition:

Statistical analysis is based on the frequency of letters encountered in the cyphertext.

Also it consists of the analysis of the frequency of n-grams in the ciphered text for $n \geq 2$.

In order to eliminate the frequency of n-grams for $n \geq 2$ it is possible to use permutations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to encipher a plain text. Also, it would be very useful to use permutations that eliminate all the bigrams of a plain text each time (Welsh, 1993).

Examples:

Let's choose the following plaintext:

"ATTACK AT 1 PM."

Suppose that we use the following permutation (2 3 4 5 6 7 8 9 10 11 12 13 14 1). The preceding plaintext would be encrypted the following way:

"TTACK AT 1 PM.A"

It is clear that the preceding permutation did not eliminate any bigram except for the first 2 letters of the plain text.

Suppose that we use the following permutation:

(1 4) (2 7) (3 6) (10 13) (5 11) (8 12) (9 14) which corresponds to the following transposition:

4 7 6 1 11 3 2 12 14 13 5 8 10 9

ATTACK AT 1 PM.

The preceding text would be encrypted the following way:

A		K	A	1	T	T	P		M	C	A		T

We claim the last permutation eliminated all the bigrams of the plain text.

It is not obvious at first sight since we still have the bigram TT.

But we have to base this on the following principle:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Principle 1:

A permutation eliminates all the bigrams of a plaintext if and only if it eliminates all the bigrams of a plaintext that do not have the same letter twice and such that the plaintext has the same length as the length of the permutation.

Let's take a cipher text that has all the properties described in the preceding claim: "ABCDEFGHIJKMNO"

Let's take the preceding permutation and permute ABCDEFGHIJKMNO. The result is the following cyphertext:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
4	7	6	1	11	3	2	12	14	13	5	8	10	9
A	B	C	D	E	F	G	H	I	J	K	M	N	O
D	G	F	A	K	C	B	M	O	N	E	H	J	I

We clearly see that the permutation eliminated all the bigrams of the plaintext. Therefore if we want to produce a permutation that will satisfy principle 1 it suffices that:

Proposition 6:

If a permutation has a corresponding transposition $p_0p_1..p_n$ where

$0\leq p_i \leq n$ and $\forall\ 0< j \leq n$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$p_{j-1} \neq p_j - 1$ therefore such a permutation satisfies principle 1.

Proof:

Suppose that we have a permutation $p = p_0p_1..p_n$ such that $\forall\ 0 < j \leq n$, $p_{j-1} \neq p_j - 1$ and does not satisfy principle 1.

In that case it means that if we take a text and $\forall\ 0 \leq j \leq n$ and $\forall\ 0 \leq i \leq n$ where $i \neq j$ then $Ti \neq Tj$ (meaning that all the letters of text T are different) then when we permute T using the permutation p we have the following cyphertext $T' = T'_1T'_2..T'_n$ such that $\exists\ 0 < m \leq n$ such that $T'_m = T_i$ and $T'_{m-1} = T_{i-1}$ (meaning at least one of the bigrams was preserved). If one of the bigrams was preserved then it means that the permutation of T_i was transposed to position v then T_{i-1} was transposed to position v-1. In that case the permutation $p = p_0p_1.v_{i-1}v_i.p_n$ and $v_{i-1} = v_i - 1$ which is a contradiction.

Q.E.D.

4.2 Number of permutations of a set P satisfying Proposition 6:

We call the set of all permutations of length n P_n and the subset $\not{P}_n$, the permutations that belong to Pn and that satisfy proposition 6. We will denote by $\not{n!}$ the cardinality of $\not{P}_n$

We will try to find an upper bound and a lower bound of $|\not{P}_n|$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.1 An upper bound of $|\not{P}_n|$

We can already assert that:

$\forall\, n \geq 2$ then $|Pn| > |\not{P}_n|$.

The proof is very simple since:

$\forall\, n \geq 2$ then the permutation $0\ 1\ 2\ ..\ n\text{-}1 \in P_n$ and $0\ 1\ 2\ ..\ n\text{-}1 \notin \not{P}_n$ then $|P_n| > |\not{P}_n|$.

Therefore we have the following upper bound $n! > |\not{P}_n|$.

Let's try to find a lower bound.

4.2.2 A lower bound of $|\not{P}_n|$

For the simplicity of the computation we will define the function *Successor.*

When we use permutations we represent them by integers. A permutation of length n will be represented by the integers of the set {0,1,..,n-1}.

Definition of the function Successor:

$\forall\, i \in \{0,1,..,n\}$, $\text{Successor}(i) = i + 1 \bmod(n)$.

From now on we will abbreviate the function "Successor" by "Scsr"

As an example if we take the set {0,1,2} we have Scsr(0) = 1, Scsr(1) = 2, Scsr(2) = 0.

Let's compute 3!:

Before we will enumerate all the permutations of P3:

0 1 2   0 2 1   1 0 2   1 2 0   2 0 1   2 1 0

Since Scsr(0) = 1 the permutation 0 1 2 does not belong to $\not{P}3$

Since Scsr(1) = 2 the permutation 1 2 0 does not belong to $\not{P}3$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since Scsr(2) = 0 the permutation 2 0 1 does not belong to P3

therefore !3! = 3.

Let's compute 4!:

First we will enumerate all the permutations of P4:

0123	1023	2013	3012
0132	1032	2031	3021
0213	1203	2103	3102
0231	1230	2130	3120
0312	1302	2310	3201
0321	1320	2301	3210

~~0123~~	~~1023~~	~~2013~~	~~3012~~
~~0132~~	1032	2031	~~3021~~
0213	~~1203~~	2103	3102
~~0231~~	~~1230~~	~~2130~~	~~3120~~

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

~~0312~~	~~1302~~	~~2310~~	~~3201~~
0321	1320	~~2301~~	3210

therefore 4! = 8.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For P5 we will just count permutations starting with 0:

01234	02134	03124	04123
01243	02143	03142	04132
01324	02314	03214	04213
01342	02341	03241	04231
01423	02413	03412	04312
01432	02431	03421	04321

~~01234~~	~~02134~~	~~03124~~	~~04123~~
~~01243~~	02143	03142	04132
~~01324~~	~~02314~~	03214	04213
~~01342~~	~~02341~~	03241	~~04231~~
~~01423~~	02413	~~03412~~	~~04312~~
~~01432~~	02431	~~03421~~	04321

We have 9 permutations starting with 0 that belong to ~~P5~~. It will be the same number for permutations starting with 1, 2, 3 and 4 therefore ~~5!~~ = 45.

We see that:

~~3!~~ = 3 > 2!

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

~~4~~! = 8 > 3!

~~5~~! = 45 > 4!

Let's prove that $\forall$ n ≥ 3 then ~~n~~! ≥ ~~(n-1)~~!(n-2).

We have $\mathcal{P}$n-1 = ~~(n-1)~~! Where n ≥ 3.

Let's take a permutation p ∈ $\mathcal{P}_{n-1}$.We can see clearly that if p = $p_0\ p_1$ .. a_{n-2} we can construct a permutation $p^+ \in \mathcal{P}_n$ by inserting the integer n-1, n different ways into p.

Example:

p = 0 2 4 1 3

let's take 5

we can form 5 0 2 4 1 3, 0 5 2 1 3, 0 2 5 4 1 3, 0 2 4 5 1 3, 0 2 4 1 5 3,

0 2 4 1 3 5.

With that permutation p only (n-2) permutations can be constructed such that they belong to $\mathcal{P}_n$ since necessarily one of the integers will be Scsr(n-1) and one of the integers i of p will be such that Scrs(i) = n-1. This is valid for all the permutations of $\mathcal{P}_{n-1}$ and there are ~~(n-1)~~! of them therefore ~~n~~! ≥ ~~(n-1)~~!(n-2).

Q.E.D.

Remember we were at:

~~5~~! = 45 > 4!

Since $\forall$ n ≥ 3 then ~~n~~! ≥ ~~(n-1)~~!(n-2) we can write that:

~~6~~! ≥ ~~5~~! (6 - 2) and ~~5~~! > 1.8*4! Then

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$6! \geq 4! * 4 * 1.8 \geq 5!$

$7! \geq 6! * 5 \geq 5! * 5$

therefore for !n!:

$!n! \geq !(n-1)!(n-2) \geq !(n-2)!(n-3)(n-2) \geq !(n-3)!(n-4)(n-3)(n-2).$

$!n! \geq 7! * 6 * 7 * 8 * .. *(n-2).$

$!n! \geq 5! * 5 * 6 * ..*(n-2) = 5 * (n-2)!.$

Therefore:

$n! > |Ᵽ_n| \geq 5(n-2)!$

Therefore according to the sterling approximation of $n! \cong \left(\sqrt{2\pi n}\right)\frac{n}{e}^{n}$ enumerating the set $Ᵽ_n$ would take an algorithm of complexity of at least $O(a^n)$ where a is greater than 1.

Open question:

Is $n! > |Ᵽ_n| \geq (n-1)!$

4.2.3 Characteristic function of Ᵽn:

The characteristic function of Ᵽn works solely on the set Pn and is defined as

$\forall\ p \in Pn,\ C(Pn, p) = 0$ if $p \notin !Pn$, 1 otherwise.

4.2.4 Complexity of the characteristic function C:

The characteristic function will traverse the set P_n and for each integer it will apply the function Scsr. The traversing of the set P_n is linear and the function Scsr is constant. As a code example for the characteristic function C see figure ?.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Therefore the complexity of $C(\text{Ꝑ}_n, p)$ is linear, i.e., $O(C(\text{Ꝑ}_n, p)) = n$.

// returns the successor of an integer

```
INT Scsr(INT i, INT &CardPn)
{
   return (++i) % CardPn;
}
```

// returns the cardinality of an array of positive integers

// the array of integers terminates with the flag -1

```
INT Card(INT *A)
{
      INT Cnt = -1;
      while(A[++Cnt] != -1);
      return Cnt;
}
```

// Characteristic function of Pn

```
INT C(INT *Pn, INT *p)
{
   INT CardPn = Card(Pn);
   INT Cnt;
   for(Cnt = 0; Cnt < CardPn; Cnt ++)
      if(Scsr(p[Cnt], CardPn) == Pn[(Cnt + 1) % CardPn])
return 0;
   return 1;
}
```

4.3 Extracting the set Ꝑn from Pn:

Suppose that ones chooses an integer i and applies B(i), it is possible that

$B(i) = p \notin !Pn$. Therefore one wants to produce a permutation p from i

such that $p \in !Pn$ each time.

Algorithm:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1) Take an integer.
2) Apply the function B(i).
3) Apply the function *GenerateDn* (B(i)).

The function *(B(i))* will take any permutation of Pn and produce a permutation of !Pn Before, the function will apply the characteristic function on !Pn. If the function returns 1 then the function returns p otherwise the function applies the following algorithm: It traverses the permutation $p = p_0..p_{n-1}$. Each time it encounters an integer p_i of p such that $Scsr(p_i) = p_{i+1}$ it does the following:
it switches p_i with p_{i+1} and then when the entire permutation is traversed it returns the result.

4.3.1 Example:

p = 0 1 2 3 4. Since C(p) = 0, we traverse p. First we apply Scsr(0) = 1. Since 0 is followed by 1 in the permutation we switch 0 and 1 and we have 1 0 2 3 4. Then since we switched the integers 0 and 1 we apply Scsr(2) = 3. Since 2 is followed by 3 then we switch 2 and 3 and we have 1 0 3 2 4. Then we return the permutation 1 0 3 2 4 which belongs to !P5.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3.2 Proof that the algorithm always returns an element of $\mathbb{P}_n$.

The only problem encountered when p_i and p_{i+1} are switched is, if p_{i-1} and p_{i+2} exist. But since p_{i+1} is successor of p_i and by definition $p_{i-1} \neq p_i$ and $p_{i+2} \neq p_{i+1}$ then it is impossible to have p_{i+1} and p_{i+2} respectively Successors of p_{i-1} and p_i.

Also this algorithm will give the entire scope of the set $\mathbb{P}_n$.

4.3.3 Complexity of the GenerateDn:

Since the algorithm traverses p and we apply only 2 functions namely *Scsr* and *Switch*, which are constant functions, then the function *GenerateDn* is linear.

Remark: in the function GenerateDn, the function Switch is embedded in the function itself and characterized by the following lines:

```
INT p1 = p[Cnt], p2 = Pn[(Cnt + 1)];
p[Cnt] = p2;
p[Cnt + 1] = p1;
```

The function GenerateDn:

```
INT * GenerateDn(INT *Pn, INT *p)
{
   INT CardPn = Card(Pn);
   INT Cnt;
   if(C(Pn, p)) return Pn;
   else
   {
      for(Cnt = 0; Cnt < CardPn; Cnt++)
      {
         if(Scsr(p[Cnt], CardPn) == Pn[(Cnt + 1) % CardPn])
         {
            INT p1 = p[Cnt], p2 = Pn[(Cnt + 1)];
            p[Cnt] = p2;
            p[Cnt + 1] = p1;
            Cnt ++;
         }
      }
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    }
    return p;
}
```

We just show that from a positive integer we can get a permutation that satisfies proposition 6 with an algorithm of logarithmic complexity. We also proved that enumerating the corresponding set of permutations satisfying proposition 6 would require an algorithm of exponential complexity, i.e., if the permutations are of length n then the complexity of the algorithm is at least $O(a^n)$ for a greater than 1.

4.4 Remarks on principle 6:

Cryptanalysts verify if the encrypted plain text is not in reverse. They may be also interested in the reverse bigram statistics. Therefore we could strengthen choosing the permutations such that

Principle 2:

We choose only permutations of corresponding transposition $p_0 p_1 .. p_n$ *where* $0 \le p_i \le n$ *and* $\forall$ $0 < j \le n$ *then* $p_{j-1} \ne p_j - 1$ *and* $p_{j-1} \ne p_j + 1$.

The scope of our discussion is, in fact, to use permutations that would give the most chances of transforming existing words into non-existent ones. Although one could use principle 2 to select a permutation, we have to emphasize the fact that even if one has chosen a permutation that will give a palindrome, all the probable reverse digram statistics would be hidden when the plain text is permuted at the bit level. Nevertheless the cardinality of a set of permutations satisfying principle 2 would have to be studied in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

order to find the length of the permutation that would make impossible for the fastest computer to enumerate in reasonable time that particular set of permutations.

4.5 Transforming passwords : the complete algorithm:

Remember all the tools that we have:

I. A homophonic code and an alphabet α used to create a key.

II. A function that maps the key created from α into a set of permutations. We can use either:

   A. the surjection F.

   B. the bijection B.

III. Once we have a permutation we have the function GenerateDn that will give us a permutation that will eliminate digram statistics when used to permute a plain text.

To transform the password we will need to employ keys $k_1 \in K_1$ and $k_2 \in K_2$ that will give us corresponding permutations needed to transform the password.

Algorithm outline of password transformation:

I. Take 3 good big enough keys k_1, k_2 and k_3

II. Choose a password Pwd (bad or good) and apply the following algorithm:

   A. Compute [illegible](k_1), [illegible](k_2) and get the corresponding permutation p_1 and p_2.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

B. Compute GenerateDn(p) and get the good permutation p_1^+ and p_2^+

C. If the password Pwd does not have the critical length then we concatenate the password and k_3 to get the appropriate length.

D. Otherwise keep Pwd.

E. Permute the letters of the password using the permutation p+

F. Permute the bits of the password using permutation p+.

The sections 1, 2.a, 2.b, 2.c, 2.d have been described in the preceding chapter, let's describe section 2.e and 2.f.

4.5.1 Permute the letters of the password using the permutation p^+:

Example:

Suppose that we have a proper permutation, the plain text is going to be transformed the following way:

We have the permutation 4021 and the plain text "CRYP".

C is at position 0, R is at position 1, Y is at position 2 and P is at position 3.

Therefore, according to the permutation 4021 we are going to permute C at position 2, R at Position 1, Y at position 0 and P at position 3 see Table 7:

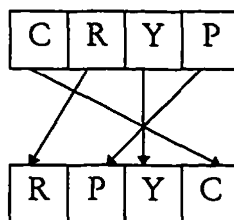

TABLE 8 : PLAIN TEXT PERMUTATION EXAMPLE

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.5.2 Permuting the bits of the password using permutation p^+:

The permutation at the bit level is very similar than the one at the byte level. Taking the preceding example and following the homophonic representation described in Table 6, the password "CRYPTO" is represented by the following bit string:

0000110111 0001110110 0010011101 0001110011 0001111010 0001101110

Let's permute the first 8 bits of the bit string:

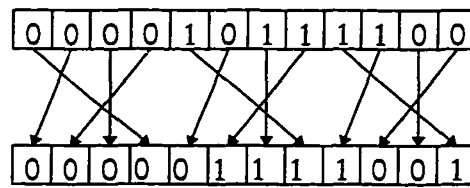

TABLE 9 : BIT PERMUTATION

In the algorithm we use 3 different keys k_1, k_2 and k_3, respectively those 3 keys belong to the sets K_1, K_2 and K_3.

4.5.3 *Retrieving the original password:*

Once the password has been permuted using the permutation p it is trivial to retrieve the original password with the same permutation.

When we transform a password with the permutation 4021 we put letter at position 0 to position 4, letter at position 1 at position 0, letter at position 2 at position 2 and so on. To retrieve back the original password we take letter at position 4 and we put it at

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

position 0, we take letter at position 0 and we put it back to 1 and so on. For the complete algorithm of permuting back the original password see the function DecryptoByte in section 5.8.

As a generality if we have a permutation $p = p_1p_2..p_{n-1}$ we can represent the preceding permutation in cycle notation the following way:

$(0,p_1)\ (1,p_2)..(n\text{-}1,p_{n-1})$ (Biggs, 1989, pp.55-59).

If we use the preceding permutation to transform a password trivially we can use the corresponding permutation to retrieve the preceding password:

$(p_1,0)\ (p_2,1)..(p_{n-1}, n\text{-}1)$.

4.5.4 The computational advantages of the password transformation:

The password transformation described above has the following computational advantages:

It has a linear complexity, meaning that it has a trivial complexity of O(n) where n is the size of the password. Therefore we can claim that any other existing algorithm has the same or a worse complexity.

There is no limitation of the length of the permutation that can be used (except possibly the capacity of the hard drive of the computer).Since we can consider a permutation as a key, we can say that there is no limitation on the size of the key that can be used regardless of the computational power of the computer that is used.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

No arithmetic operation is used when transforming the password (only simple swapping); therefore, a very weak computational system can perform password transformation in linear time.

4.5.5 Definitions of the key sets K_1 and K_2:

The sets K_1 and K_2 are respectively the set of keys to perform the byte permutation and the set of keys to perform the bit permutation. As we already know those 2 sets have different properties in terms of cardinality. The set K_3 is the set of keys used to compensate the length of the password if the password is not big enough (see the critical cardinality of K_2).

4.5.6 On the critical Cardinality of K_1 and K_3:

As described in 1.6 the length of a password should be at least 11 letters so that it is impossible for the fastest computer on earth to enumerate all the possible words of 11 letters.

Let's call L(Pwd) the critical length of a password (here we decided that

L(Pwd) = 11)

We already know that k_3 is appended to the password when the password is less than 11 letters. Since we accept any kind of password (even passwords of length 0) the key k_2 should be of at least of length L(Pwd).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Also we are going to use k_1 to permute a password of critical length L(Pwd), therefore k_1 should generate a permutation of length L(Pwd). In that case remember since we are using permutation that eliminates digram statistics we have the following inequality:

$n! > |P_n| \geq 5(n-2)!$

Therefore to be totally on the safe side k_1 should generate a permutation of length L(Pwd + 2).

Consequently, if it has been computed that the length of a password should be 11 letters so that it is impossible for the fastest computer on earth to enumerate all the possible words of 11 letters, we will have to consider passwords of 13 letters.

Let's call the critical length of k_1 $L(k_1)$.

In section 15.1.1 we have the relation:

$(\Theta(P_L) - 1)! < |K(\alpha,L)| \leq \Theta(P_L)!$ consequently we have the following equality:

We have an alphabet of cardinality c (c different letters).

$c^{L(k1)} =$ L(Pwd + 2)! therefore L(k1) $= \lceil \log_c(L(Pwd + 2)!) \rceil$.

Example of critical size of k_1 and k_2:

We suppose that L(Pwd) = 12. So we will consider L(Pwd + 2) = 14

Therefore the length of k_2 is 14.

Suppose that we use an alphabet of 26 letters then L(Pwd + 2)! =. 87 178 291 200

Therefore $L(k_1) = \lceil \log_{14}(87178291200) \rceil = 10$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We see that the length of k_1 and k_3 are directly computed according to the length of the password. The length of a password is determined solely by the capability of a machine to enumerate all the possible potential passwords in a reasonable time. We consider that reasonable time is less than a century.

In the algorithm outline the Verifier has to take "2 good big enough keys k_1 and k_3." We meant by "good big enough length" respectively $L(k_1)$ and L(Pwd).

The size of k_3 is also an important factor and has to be discussed separately since it will determine the ability of the algorithm to resist plaintext and chosen plaintext attacks.

4.5.7 On the critical cardinality of K_2

Resisting plaintext attack and chosen plaintext attack depends solely on the fact that the bits of the password are permuted. Since the permutation is given according to the key k_3 it is very important to know the length of the corresponding permutation that will give the property of the algorithm of resisting plaintext and chosen plaintext attacks.

4.5.8 How big should the permutation be that permutes bits to resist a plaintext attack?

When transforming the password at the bit level we will have keys that transform the password the same way.

For simplification we will use a homophonic code to represent the password that has to be transformed. With a homophonic code the password that has to be transformed will be represented by the same number of 0's and 1's.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Suppose that the number of bits that represent a password is m (remember that m has to be necessarily even).

Since we are performing a permutation of a bit string of 0's and 1's, let us see in how may ways we can permute a bit string of even length that has the same number of 0s and 1s.

We have m/2 zeros and m/2 ones.

Permuting such a string is the same as choosing m/2 different objects out of m objects.

There are $\binom{m}{m/2}$ ways of choosing m/2 objects out of m objects.

We know that:

$$\binom{m}{m/2} = \frac{m!}{(m-\frac{m}{2})!\frac{m}{2}!} = \frac{m!}{\left(\frac{m}{2}!\right)^2}$$

Therefore we will have $M_m = \frac{m!}{\left(\frac{m}{2}!\right)^2}$ different ways to transform a password at the bit level.

Since we have m! different permutations and M_m different bit strings that have the same number of 0' and 1's level then if X_m is the number of permutations that encrypt a given homophonic bit string the same way we have

m! = * Mm therefore X_m = m!/M_m = $\left(\frac{m}{2}!\right)^2$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Therefore if we have an homophonic bit string of 0's and 1's there are M_m different ways to permute it and for a given bit string there are X_m permutations that will transform that homophonic bit string into the another given homophonic bit string.

Therefore we should chose M_m and X_m such that M_m and X_m cannot be enumerated in reasonable time by the fastest computer.

We are going to prove that $X_m > M_m$. $\forall$ m > 4.

Since m is a even number let's take m = 6, we have $X_6 = 36$, and $M_6 = 20$.

Therefore $X_m > M_m$ is true for m = 6.

Suppose that it is true for j > 6 i.e $X_j > M_j$

Let's see if $X_j > M_j \Rightarrow X_{j+2} > M_{j+2}$

Trivially we have:

$M_{j+2} = M_j(16/(m+2)(m+1))$ and X_{j+2} = since m > 4 we have

$[(m+2)(m+1)]^2/16 > (16/(m+2)(m+1))$

therefore since $X_j > M_j$ then

$X_j[(m+2)(m+1)]^2/16 > M_j[(m+2)(m+1)]^2/16 > M_j(16/(m+2)(m+1))$

then $X_{j+2} > M_{j+2}$

We proved that $X_j > M_j \Rightarrow X_{j+2} > M_{j+2}$

Consequently we have $X_m > M_m$. $\forall$ m > 4

In that case we are only interested in M_m such that it should be impossible for the fastest computer on earth to enumerate M_m

Computing the length of the permutation:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

According to the sterling approximation of $m! \approx \left(\sqrt{2\pi m}\right)\frac{m}{e}^{m}$ (Roberts, 1984) then:

$$\frac{m}{2}^{2}! \approx \pi m\left(\frac{m}{2e}\right)^{m} \text{ then } M_m \approx \frac{\left(\sqrt{2m\pi}\right)\left(\frac{m}{e}\right)^{m}}{(m\,\pi)\left(\frac{m}{2e}\right)^{m}} = \frac{\sqrt{2m\pi}}{m\pi}2^{m}$$

We can say that $M_m > \frac{2^m}{4m}$ since $\frac{\sqrt{2m\pi}}{m\pi} > \frac{1}{4m}$ for $m \geq 1$.

Therefore it would be safe to choose

$$\frac{2^m}{4m} = \frac{2^{m-2}}{m} = M_m$$

We then have the equality:

$m - 2 = \log_2 m + \log_2 Z \Leftrightarrow m = \log_2 m + \log_2 Z + 2.$

Evaluation of $\log_2 m$:

Suppose that $m = 10^3$. It means that the fastest computer is trying to perform in reasonable time 10^3! Which is a number that is greater than all the atoms in the universe.

Therefore in the computation of m we could safely choose:

$m = 7 + \log_2 Z + 2 = 9 + \log_2 Z$

Example:

If a computer can perform a permutation per computer cycle, but it cannot enumerate in reasonable time all the passwords of length 12 constructed from an alphabet of 26 letters, then it would be safe to choose

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$m = 9 + \log_2(12^{26}) = 9 + \frac{12\ln(26)}{\ln 2} \approx 65$$

4.5.9 Plaintext attack:

Suppose that we have a password P and a permutation of length m.

We apply the algorithm to P and P is transformed into P' just permuting the bits. We give P and P' to cryptanalysis. The cryptoanalyst finds a key such that it transforms P into P'.

As an example suppose that there are x keys where x > 1 that will transform P into P' the same way. Therefore, the probability that the key k is the right key is 1/x. In the preceding example we found that it is safe to choose a permutation of length 65. Consequently, a cryptoanalyst performing a plaintext attack has one chance out of $\left(\frac{m}{2}!\right)^2$ to have the right permutation (m being the length of the permutation)! Therefore his next hope is to perform chosen plaintext attack.

4.5.10 Chosen Plaintext Attack:

In that case the attacker has the capability to find the cyphertext corresponding to an arbitrary plaintext message of his choosing. Therefore, since we represent each plaintext with an homophonic code, if the attacker choses the plaintext, it will be represented by a bit string that has the same number of 0's and 1's. Consequently, the attacker will be confronted with the same problem of the plaintext attack, he can find the key that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

encrypts the chosen plaintext into the encrypted text but the probability that it is the right key is again $1/\left(\frac{m}{2}!\right)^2$.

4.5.11 Chosen cyphertext attack:

In that case the attacker can arbitrarily choose a cyphertext and find the corresponding decrypted plaintext. Therefore the attacker has to provide a cyphered text that has the same number of 0's and 1's. Consequently, decryptor will use the key to decrypt the text.

According to (Biggs, 1989, p. 57):

The following properties hold in the set Sn of all permutations of {1, 2, .., n}.

Theorem:

If π and σ are in Sn, so is $\pi\sigma$.

For any permutation π, σ and τ in Sn we have: $(\pi\sigma)\tau = \pi(\sigma\tau)$

The identity function, denoted by id and defined by id(f) = f for all f in Nn is a permutation and for any σ in Sn we have $id\sigma = \sigma id = \sigma$.

For every permutation π in Sn there is one and only one inverse permutation π^{-1} such that $\pi\pi^{-1} = \pi^{-1}\pi = id$.

Part 4 of the theorem shows that if x permutations encrypt a plaintext the same way then x permutation decrypt the cyphertext the same way, then for chosen ciphertext attack the attacker will be confronted with the same problem of the chosen cipher text attack:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

he can find the key that decrypts the chosen cipher text into the plain text but the probability that it is the right key is again $1/\left(\frac{m}{2}!\right)^2$.

Therefore, the next and last hope for the attacker is to proceed to an adaptive chosen plaintext attack.

4.5.12 Adaptive chosen plaintext attack:

Adaptive chosen plaintext:

The attacker can determine the cipher texts of chosen plain texts in an interactive or iterative process based on previous results.

Therefore, suppose that the attacker submit a chosen plain text for encryption. Suppose that the attacker found a key k permuting the plain text into the corresponding cipher text.

In that case k is part of the set of keys that can encrypt the plain text into the cipher text the same way, as we know there are $x = \left(\frac{m}{2}!\right)^2$ of them.

Suppose that the attacker chooses another plaintext, then the probability that k will encrypt successfully the second plain text is:

$p(k) = 1/\left(\frac{m}{2}!\right)^2 = 1/x$..

Therefore the probability that k will encrypt a third plain text is $p(k)^2$ and so on for different plain texts.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As a generality, the probability for k to encrypt the nth plaintext is $p(k)^{n-1}$. Suppose that as a generalization, the hit list of a cracker has a cardinality of 300 000 then the probability that a key k which is different from the original key encrypts the hit list the same way is $1/x^{300000}$.

Therefore, even if the cryptanalyst found a key k he would have to try k on a second chosen plaintext. If the key does not encrypt the second plaintext the right way, he then would be reasonable to find another key that encrypts the first chosen plain text and try it on the second plaintext. We see that brute force has to be used for this kind of chosen plaintext attack.

Adaptive chosen plain text attack using a probabilistic approach:

Suppose that the attacker decides that a bit b is permuted to a certain position p.

How many times do we have to try bit submit a plaintext to be sure the bit b is not permuted with the real key to position p?

Let's call X the number of all possible strings strings bit of length L represented by an homophonic code.

By definition half of all the possible strings bit of length L represented by an homophonic code will be 0 at the bit b and the other half will be 1 since all the possible homophonic bit strings have the same number of 0's and 1's. Therefore, if we suppose that we have the wrong bit b permuting to position p, this means that another bit b′ will be swapped at that position. In that case there are X/2 homophonic bit strings that will

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

swap bit b' at the right position and there are X/2 homophonic bit strings that will swap bit b' at the wrong position. Consequently, to be sure that bit b does not swap at position p we have to submit for encryption at least X/2 different homophonic bit strings of length L. Since the length of the permutation has been chosen such that it is impossible to enumerate in reasonable time X/2 homophonic bit string, we can see that this approach is unfeasible.

But, the attacker can try a probabilistic approach.

If the attacker suppose that the bit b is permuted at position p and that bit b is the wrong bit, what is the probability that X/2 successive homophonic bit string will be the ones such that bit b will be permuted at position p?

For the first round, since there are X/2 out of X bit string that have the property of being permuted properly then the probability is ½.

The second round the probability is ¼ and so on. Therefore, an attacker can legitimaly think that he has the right bit after n trials where n is a lot less than X/2.

As an example: suppose that the attacker tries 10 times with 10 different bit strings and that 10 times the bit b is permuted successfully then the chance that the attacker would have submitted the right strings if the bit b was wrong is $1/2^{10}$. Therefore there are $(1 - 1/2^{10}) = 0.9990234375$chances that the attacker has the right bit swapping.

To get that high probability, if the length of the permutation is o then the attacker would have to try only o*10 times.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The attacker can do this for o bits and therefore can find a permutation that has a probability of being the right one of 0.9990234375^o. As an example if o = 65 then the probability is $0.9990234375^{65} \approx 0.55$ which in fact is not so great.

Therefore, we can assess that our bit transformation schema resists the 2 adaptive chosen plaintext attack presented above, considering especially with the second method that the attacker want to be 100% sure that he found the right permutation. But we can see that it may be possible to use a probabilistic approach. We leave this as an open question for further research.

Remark:

we can use the same conclusions for an adaptive chosen ciphertext attack since we know that (according to Part 4 of the theorem of section 4.5.11) if x permutations encrypt a plaintext the same way then x permutation decrypt the cyphertext the same way.

Since we are not using the permutations to encrypt any text, but that we use permutations to transform a password and consequently increasing exponentially the probable password space, we will see that even if the password transformation may not resist adaptive chosen plaintext or adaptive chosen cipher text attack then it is always possible to use an encryption for authentication that can resist the adaptive attacks.

Therefore we consider that our plain text transformation resists plaintext and chosen plaintext attack and also cipher text and chosen plaintext attack. We proved in two cases

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that it also resists adaptive chosen plaintext and ciphertext attacks. The potentiality of using a probabilistic attack correlated with an adaptive attack successfully is left as an open question for further research.

4.5.13 Does the transformed password have to look random?

The answer is absolutely no. The only criteria for the password is that it has to be transformed into a word that is not part of a hit list of a cracker. In that case it has to have the most chances of belonging to a set of words that cannot be enumerated in reasonable time by the fastest computer. Since we have proved that our password transformation resists plaintext attack and chosen plaintext attack, and we supposed that the keys are secret, then we have to see what is the probability of a transformed password not to be part of a hit list of a cracker.

2 tests have been been conducted on a 150 MHz pentium using the Crypto library (see 5.8):

We used the password "ABADIE" and transformed it with permutations of length 19 selected randomly which satisfied proposition 6 (see Appendix 1 for the result). We chose as a hit list the dictionary of Microsoft word 7.0 that has 100 000 words. On the 1 million word transformation, none was part of the dictionary.

We conducted another test where the keys to transform the word ABADIE where not selected randomly but selected starting from integer 0 and incremented by 1 for each iteration. Again, of the one million word transformation, none was part of the dictionary.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.5.14 Transforming the password back to the original:

Since we are using permutations to transform the password it is trivial to transform the password back to the original by using the corresponding inverse permutations a second time.

Therefore we proved that using our password transformation generates passwords that are not part of the hit list of a cracker, that resist statistical analysis, plaintext and chosen plaintext attacks and that the transformation can be performed in linear time.

In that case password transformation can be used to perform safe authentication.

In the next chapter we are going to propose an authentication scheme that uses password transformation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 5

5. Using password transformation for authentication:

In this section we will propose an authentication method that uses the password transformation described in chapter 4. We will prove that the authentication can be done in logarithmic time and that the security depends solely on the security of the algorithm used for key exchange and the algorithm used to generate the keys.

In this section we are going to use the word User and the word Verifier in order to define the authentication algorithm.

5.1 User definition:

A User is an entity that wants to be granted some services that a specific system provides.

5.2 Verifier definition:

The Verifier is an entity that grants the specific services of a system to a User. This is done by authenticating the User. Note that the positive authentication can be done only if the Verifier and the User have agreed respectively to granting and receiving specific services.

In fact the User has to prove to the Verifier that he is himself. Once the Verifier is convinced of the identity of the User then he grants the corresponding services to the User (most commonly called authorization).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A lot of systems (Internet, bank accounts, etc.) authenticate their User according to a password. There are many other means of authentication that do not require passwords (zero knowledge, etc.). Unfortunately, most of the systems that grant services still require the User to give their password at one point in their authentication schema. As described in Chapter 1, we proved the weakness of using a password: Users have the tendency to use badly chosen passwords most of the time.

In this section, we will propose a method of authentication that can make safe use of badly chosen passwords. Then we will compare this method with other well known authentication methodologies.

In the algorithm definition we make the use of the keys k_1, k_2 and k_3 already defined in 4.5.5, and of another key called the key signature k_s. The key signature is a one time key that is used only for the purpose of one authentication.

In the algorithm, we also speak of "public key exchange". We are not restricting the public key exchange that we can use with our algorithm. But for our algorithm we use the Diffie-Hellman algorithm described in the next section.

5.3 Public key exchange:

Our system requires a private key. We suggest the use of public key exchange once to exchange the private key that will be used by the User to transform the password and by the receiver to find the original password.

For public key exchange we will give as an example an adaptation of the Diffie-Hellman key exchange (Diffie, 1979).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Nevertheless the public key exchange requires a big prime number p (Pohlig, 1978), the bigger the better. In our case the size of the prime number is well defined. We have to remember that the private key used for password transformation is quite long (14 bytes for a permutation at the byte level and more than 40 bytes for a permutation at the bit level). We have to point out that the prime number has to be greater than or equal to the cardinality of the key sets K_1 and K_2 (see 4.5.6).

It also requires a positive integer s such that s is relatively prime with $\phi(p-1)$.

Remember that $\phi(n)$ is the totient function and that it is the cardinality of the group formed by all integers that are coprime to n.

p and s are the public keys.

The User and the Receiver each choose an integer, a and b respectively, less than p.

The User computes:

$\alpha = s^a \bmod p$

The Receiver computes:

$\beta = s^b \bmod p$

Then they send the result to each other and they compute the key:

$k = \alpha^b \bmod p = \beta^a \bmod p$.

The Diffie-Hellman exchange key stops here.

Since our system needs two keys k_1 and k_2 we have to perform 2 public key exchanges.

The fundamental issue is how to pass a big key between systems that cannot support such big integers. As an example, most compilers support only 64 bit integers. Those integers

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are not big enough to compute permutations of length 40. Consequently, whenever one has to use integers so big that they are not supported by the compiler, one has to emulate such integers at the hardware level. This thesis is not intended to propose software solutions to handle integers that are not supported at the hardware level and many solutions have already been proposed to handle those problems at the software level. Once the key exchange has been performed we can use either function F or function B to find the corresponding permutations then the function GenerateDn to find the permutations that will eliminate digram statistics.

Before doing any authentication, the User and the Verifier have to exchange 2 kinds of information. The User has to give her password to the Verifier and the Verifier has to give to the User the 3 keys k_1, k_2 and k_3, keys that will be used by the User to transform her password. Consequently, the User has to keep the 3 keys secret. The User just has to remember her own password.

5.4 Keeping secret k_1, k_2 and k_3:

It is almost impossible to ask a user to remember the keys that will be used to perform authentication. Nevertheless such systems have been proposed where the user is responsible for remembering a key (Konheim, 1980). Another solution is to store the key in a smart card (Fumy, 1990) or magnetic stripe card (Feistel, 1975).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.4.1 Using Smart Cards or magnetic stripe card:

As stated in *Cryptology* (Beutelspacher, p 86) "Smart cards are ideal for the human user because they are extremely easy to use. The only burden upon the user is that he must remember his secret number in order to authenticate himself against the card".

Here we propose the use of smart cards to store the secret keys. One can think of a smart card as a portable diskette where the keys are stored. To augment security on the key storage, one can use the User's password to encrypt the keys. In that case, this is not a perfect strategy to avoid a cracker who is in possession of the smart card or the diskette to decipher the secret keys, since nothing prevents him from using a traditional hit list. But it allows the User, while the cracker tries to decipher the keys, to take the necessary steps to prevent his card or diskette from being used or report where they were stolen. It is not within the scope of this paper to discuss how a User will prove to the Verifier that his smart card was stolen, since authentication also has to be done in that case. We leave this as an open question for further study.

Therefore, before presenting the authentication algorithm, we have a User who has some portable device (smart card, diskette or other) in whose memory are stored the encrypted keys k_1, k_2 and k_3.

In the portable device a User number is also stored, which is unique for each User.

5.5 Algorithm layout:

1. The User authenticates himself to the portable device by entering his password and using its password to decrypt the keys k_1, k_2 and k_3.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. The User sends his unique index to the Verifier by using the following message: " I am User 6453 and I want to prove it".
3. Use public key exchange to exchange a key signature k_s. (the key signature has to be bigger than the permutation generated by k_2).
4. Use public key exchange to exchange a key k that is used for an encryption algorithm
5. The Verifier and the User have k_1, k_2, k_3 and k_s.
6. The User transforms his password using k_1 k_2 and k_3 by performing a byte and a bit level permutation (see 4.5).
7. Then the User uses the transformed password to encrypt k_s and then performs a second encryption of k_s with k.
8. At the same time the Verifier knows the index of the User and takes the password corresponding to the index.
9. The Verifier has the same keys: k_1, k_2, k_3 and k_s as the User.
10. The Verifier transforms the User's password using k_1 k_2 and k_3 by performing a byte and a bit level permutation.
11. The User uses the password to encrypt k_s and then performs a second encryption of k_s with k and he gets $E(k_s)$.
12. The User sends the encrypted $E(k_s)$ to the Verifier.
13. The Verifier verifies that $E(k_s)$ is the same as the one he computed.
14. If it is the same then the Verifier grants the User corresponding services.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It is the responsibility of the Verifier to keep the passwords secret. It is not within the scope of the thesis to propose a solution for that purpose.

In the preceding algorithm it is assumed that the User and the Verifier the following information have in common:

1. the User's password
2. the keys k_1, k_2 and k_3
3. the unique User identification number.

In that case the User has to send his password secretly once. The verifier has to send the keys k_1, k_2 and k_3 secretly.

The user can use a public key encryption algorithm (Rivest, 1978) to transmit its password once to the Verifier. We have to keep in mind that public key encryption cannot be used for authentication and must be used once to exchange the key (see 6.2).

Also, the keys k_1, k_2 and k_3 can be securely exchanged using a secure public key exchange algorithm (Diffie, 1978).

5.6 An authentication example:

In the following example, for purposes of simplification we will assume that the User's password is big enough that the key k_2 will not be used. In order to shorten the example we are going to perform the password transformation only at the byte level. Therefore only the key k_1 will be used to transform the user's password.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In this part we will use function F (see 3.12) and will hypothetically suppose that it is impossible to enumerate the set of permutations of length 4 in reasonable time (of course everybody knows that this is not true). In that case we have to use a key of length 10. We know that if the key is of length 10 then the cardinality of the alphabet has to be 10.

To be more realistic, we are going to use 2 alphabets $\propto$ and $\propto'$ and the 2 bijections W and W':

$\propto = \{A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z\}$

$I_{26} = \{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25\}$

W is a bijection between $\propto$ and I_{26}, W: $\propto \leftrightarrow I_{26}$

$\propto' = \{A,B,C,D,E,F,G,H,I,J\}$.

$I_{10} = \{0,1,2,3,4,5,6,7,8,9\}$

W' is a bijection between $\propto'$ and I_{10}, W': $\propto' \leftrightarrow I_{10}$

Performing the public key exchange:

The prime number that has to be chosen for the public key exchange has to be greater than or equal to the cardinality of $\propto$. Therefore we can choose the prime number 11.

Exchanging key k

The User and the Verifier have to agree upon another integer s such that s is relatively prime to $\phi(11-1)$. We can take 2.

The User chooses 2 as her private key and computes: $3^2 \bmod 11 = 9$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The Verifier chooses 5 as his private key and computes: $3^5 \bmod 11 = 1$

The User sends her result to the Verifier and the Verifier sends his result to the User.

Then the User computes $1^2 \bmod 11 = 1$.

Then the User computes $9^5 \bmod 11 = 1$.

Now the User and the Verifier have the key k = 1 in common.

Exchanging the key ks:

The User chooses as her private key 7 and computes: $3^7 \bmod 11 = 9$

The Verifier chooses as his private key 2 and computes: $3^2 \bmod 11 = 9$

The User sends her result to the Verifier and the Verifier sends his result to the User.

Then the User computes $9^7 \bmod 11 = 4$.

Then the User computes $9^2 \bmod 11 = 4$.

Now the User and the Verifier have in common the key ks = 4.

5.6.1 Transforming the User's password:

Suppose that the verifier has the following password: CRYPTOLOGY.

Suppose that the User uses the key $k_1 = 3$ to transform the password. First the Verifier has to use F to compute the permutation that corresponds to the key k_1.

Computing F(3):

Since $3 = 0*10^9+0*10^8+0*10^7+0*10^6+0*10^5+0*10^4+0*10^3+0*10^2 + 0*10^1+3*10^0$ then 3 corresponds to the word CAAAAAAAAA.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We apply F(AAAAAAAAAC) = 3 0 1 2 4 5 6 7 8 9.

We apply GenerateDn to 3 0 1 2 4 5 6 7 8 9 10 and we obtain

3 1 0 4 2 6 5 8 7 9.

Then we transform the password "CRYPTOLOGY" at the byte level:

"YRCCPLOGOY"

The User uses "YRCCPLOGOY" as a key to encrypt $k_s = 4$.

At that point we can use any kind of encryption. If one wants to use DES, one can use the first 64 bits of the string "YRCCPLOGOY". For our example let's use a simple encryption methodology which will permute all the bytes of the key using the transformed password: "YRCCPLOGOY". For that we will compute the corresponding integer values of the transformed password modulo 11 using the function W.

As an example W(Y) = 24 so we compute W(Y) mod 11 = 2. The result of W(YRCCPLOGOY) is the word "CGCCEADGAC". Consequently we will compute F(CGCCEADGAC) corresponds to the permutation 2 6 0 1 4 3 5 7 9 8. Then we apply the function GenerateDn to the preceding permutation and we obtain 2 6 1 0 4 3 5 7 9 8.

Then we use the preceding permutation to permute the bits of ks = 4.

4 in binary is the string 0000000100. Let's use its homophonic representation which is 0000111101. Let's permute the preceding bit string with the permutation 2610435798 and we obtain the bit string 0001110101 which corresponds to 16.

Then the User uses the key k to encrypt 16 again and sends the result to the Verifier E(16).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The Verifier knows the User's password. He has the same keys as the User, therefore he obtains "CGCCEADGAC" and the corresponding permutation to encrypt the key ks = 4. Since the Verifier uses the same algorithm as the User he also obtains 16. Then the Verifier encrypts 16 to obtain E(16).

The User sends E(16) to the Verifier. The Verifier tries to match E(16) with his own computation. If it is the same, the Verifier decides according to the encryption algorithm that is in use to grant the privileges to the User or to redo the authentication scheme with a new public key exchange.

If the Verifier cannot match whatever is sent by the User then the Verifier does not grant the services to the User.

Note:

In order to simplify the example we omitted to transform the User's password at the bit level. This transformation has to be done in order to eliminate the possibility of finding the keys k_1 and k_2 with a plain-text and chosen plain-text attack.

Since we proved that the password transformation resists statistical analysis, plain-text and chosen plain-text attack, the security of the algorithm only resides in the security of the key generation and on the security of the public key exchange algorithm By the nature of our authentication algorithm it is easy to use any random generating function to generate the keys k_s and the key k for each session. One can argue that generating fast random keys generates keys that can be easily guessed compromising the authentication algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Let's assume that the keys k and k_s are badly generated, the strength of our algorithm still resides in the keys k_1, k_2 and k_3. Since those keys have to be generated once for many sessions it is possible to spend adequate time to generate them and be sure to generate truly random keys (Knuth).

Therefore, since we use the transformed password to encrypt k, a cracker would have to guess the transformed password to crack the authentication.

In order for the cracker to guess the transformed password, he would have to guess the keys k_1, k_2 and k_3. Then, as we stated above, the strength of the algorithm resides in k_1, k_2 and k_3 being properly generated.

5.7 Random generating functions:

In various places we use a random generating function to generate some numbers. Particularly to choose the private key that will generate the Diffie-Hellman public key. A cryptoanalyst in that case, instead of attacking a transformed text using plain text attack, etc., would choose the solution of trying to find the algorithm of the pseudo-random generator that generates the keys that are used to transform a plaintext. Therefore, the pseudo-random generator that is used has to satisfy some conditions.

Let's say that a generator G is cryptographically secure if it passes all polynomial-time statistical tests. In other words, pseudo-random generator G is cryptographycally secure if there is no effective algorithm which in polynomial time will separate the output of G from that produced by a truly random source. (Goldreich, Goldwasser and Micalli, 1984).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.8 Crypto:

Crypto is the implementation using the C programming language that enables a developer to do the following:

Generate and exchange big public keys using a variation of DIFFIE-HELLMAN public key exchange.

Generate permutations that eliminate digram statistics when applied to a plain text.

Transform a plain text using the preceding permutations.

Transform to plain text a transformed text using the preceding permutation.

Note that in the code we use the biggest available integer on an Intel based 64 bit compiler. The algorithm can generate big keys by generating a set of smaller keys (see function MakeBigKeys( ) and ComputeBigKeys( )).

It would be trivial to port the code to a system that can support 128 bit keys. Also, bigger integers could be simulated at software level. Nevertheless, the following code can generate permutations of length 20 and greater.

5.8.1 Crypto's header:

The system uses a set of functions already described in chapter 2 and 3. The functions are described in Table 7 which is a copy of the crypto.h file.

TABLE 10: CRYPTO FUNCTION (FUNCTIONS DECLARED IN CRYPTO.H)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// Put zero in an array of integers
void Z_ARRAY(INT *A, INT Size);

// Put -1 in an array of integers
void N_ARRAY(INT *A, INT Size);

// Returns the successor of an integer
INT Scsr(INT i, INT &CardPn);

// Returns the cardinality of an array of positive integers
the array of integers terminates with the flag -1
INT Card(INT *A);

// Characteristic function of Pn  it returns 1 if it is
acceptable 0 otherwise
INT C(INT *Pn, INT *p);

//Generates a permutation that is acceptable Pn is the
unacceptable permutation and p will be the acceptable one
INT * GenerateDn(INT *Pn, INT *p);

// Computes the Deffie-Hellman key,  P is a prime number
(public),  S is another public key, A is the power
__int64 DEKeyToSend(__int64 P, __int64 S, __int64 A);

// Computes the length of the permutation,  card is the
cardinality of the set of keys used to  compute the
corresponding permutations
__int64 PermutationLength(__int64 card);

// Computes factorial the result is a 64 bit integer
__int64 Factorial(INT n);

// Bijection F that gives according to the key k the
cardinality of the set of keys the corresponding permutation
in S1
void F(__int64 card, __int64 k, INT *S1);

// Encrypts Buf1 and returns the encrypted buffer. Length is
the length of Buf1 in bytes. RtrnLen is the length of then
encrypted buffer, p is the permutation used for the
encryption
char * CryptoByte(char *Buf1, __int64 Length, __int64
*RtrnLen, INT *p);

// Decrypts Buf1 and returns the Decrypted buffer. Length is
the length of Buf1 in bytes, p is the permutation used for
the encryption
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
char * DeCryptoByte(char *Buf1, long Length, INT *p);

// Returns the bit representation of Buf. Buf is the buffer
that will be represented in bits, Length is the length in
byte of buf, RtrnLength is the size of the buffer of the bit
representation of Buf
char *TransformBit(char *Buf, long Length, __int64
*RtrnLength);

// Returns the byte representation of Buf. Buf is the buffer
that will be represented in bytes, Buf has to be string of 0
and 1's, Length is the length in byte of buf, RtrnLength is
the size of the buffer of the byte representation of Buf
char *TransformByte(char *Buf, long Length, __int64
*RtrnLength);

// Returns a set of 10, 32 bit signed integers using the rand
function
void MakeBigKeys(unsigned long *Res2);

// Returns a set of 10 Diffie-Hellman public keys in DHKeys,
p is a public prime number, s is a public key
void ComputeBigKeys
(unsigned long *BigKeys, __int64 *DHKeys, __int64 p, __int64
s);

// returns a set of 10 Diffie-Hellman privare keys in DHKeys,
p is a public prime number, s is a public key, DHKeys are the
set of 10 public keys
void ComputePrivateKeys(__int64 *DHKeys, __int64 *PrivKeys,
__int64 p, __int64 s);

// returns a 19 digit private key according to the set of
keys stored in PrivKeys
__int64 ComputeOneKey(__int64 *PrivKeys);
```

5.8.2 Crypto's functions description:

The function Z_ARRAY initializes to 0 a set of integers (array A).

```
void Z_ARRAY(INT *A, INT Size)
{
   int Count;
   for(Count = 0; Count < Size; Count++)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      A[Count] = 0;
}
```

The function N_ARRAY initializes to 0 a set of integers (integer array A). This function is used in function C which computes the characteristic function of Pn (see below).

```
void N_ARRAY(INT *A, INT Size)
{
   int Count;
   for(Count = 0; Count < Size; Count++)
      A[Count] = -1;
}
```

The function Scsr returns the successor of an integer according to the cardinality of the set of integers. This function is used in function C that computes the characteristic function of Pn (see below)

```
INT Scsr(INT i, INT &CardPn)
{
   return (++i) % CardPn;
}
```

The function Card returns the cardinality of a set of positive integers (array of integers I). The array of integers I has to have a terminator flag which is -1. The function Card traverses all the integers of the array until it meets the flag

-1 and returns the result.

```
INT Card(INT *A)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
{
   INT Cnt = -1;
   while(A[++Cnt] != -1);
   return Cnt;
}
```

Characteristic function of Pn returns TRUE (1) if the permutation Pn represented by the array of integers Pn satisfies proposition 1 defined in chapter 3. Otherwise it returns 0.

```
INT C(INT *Pn, INT *p)
{
   INT CardPn = Card(Pn);
   INT Cnt;
   for(Cnt = 0; Cnt < CardPn; Cnt ++)
      if(Scsr(p[Cnt], CardPn) == Pn[(Cnt + 1) % CardPn])
return 0;
   return 1;
}
```

GenerateDn generates a permutation that satisfies principle 1 described in Chapter 3. First the function verifies whether the permutation satisfies principle 1. If principle 1 is satisfied by permutation Pn then the permutation is returned by the function, otherwise a new permutation satisfying principle 1 is generated.

```
INT * GenerateDn(INT *Pn, INT *p)
{
   INT CardPn = Card(Pn);
   INT Cnt;
   N_ARRAY(p, CardPn + 1);
   if(C(Pn, p)) return Pn;
   else
   {
      for(Cnt = 0; Cnt < CardPn; Cnt++)
      {
         int x = (int) Pn[Cnt];
         int y = (int) Pn[Cnt + 1];
         if(x + 1 == y)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        {
          p[Cnt] = Pn[Cnt + 1];
          p[Cnt + 1] = Pn[Cnt];
          Cnt ++;
        }
        else p[Cnt] = Pn[Cnt];
      }
    }
    return p;
}
```

DEKeyToSend Computes the Deffie-Hellman key. The computation follows the principles described in 3.21.3. Note that P is a big public prime number and S is a public key. In that algorithm, S is raised to the power A. The complexity of the function is alogA, where a is a constant. That complexity is achieved by computing the key the following way:

The first iteration computes S^2.

The second iteration computes S^4.

The ith iteration such computes S^b where $b = 2^i$. Iterations are conducted until the critical iteration j such that 2^j and $2^{j+1} > A$.

If $2^j = A$ then the result is returned otherwise S at the power 2^j is saved in an array. Then we compute a new power which is $A' = A - 2^{(j-1)}$ and we compute $S^{A'}$ the same way we started S^A until we meet an iteration j' such that $2^{j'} \leq A'$ and $2^{j'+1} > A'$ and we save the result in an array. This function follows the principle that any integer $A > 0$ can be decomposed the following way:

$A = 2^{a0} + .. 2^{ai} .. + 2^{an}$ where

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$2^{a0} \leq A$ and $2^{a0+1} > A$.

$2^{a1} \leq A_1$ and $2^{a1+1} > A_1$ where . $A_1 = A0 - 2^{a0}$.

$2^{ai} \leq A_i$ and $2^{ai+1} > A_i$ where . $A_i = A_{i-1} - 2^{ai-1}$.

Then all the results are saved in an array (i) and multiplied by each other.

Note that the array I can have only 18 elements. We consider that 18 elements are sufficient for any computations. If all the elements of the array were filled then A would be greater than 2^{60} since $60 \cong 18 / \log 2$.

```
__int64 DEKeyToSend(__int64 P, __int64 S, __int64 A)
{
   __int64 Result = S, Result1 = S, Result2 = S, OldResult =
S;
   INT Power = 1, WPower = 1;
   int flag = 0;
   __int64 i[18] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
   int j = 0;
   int k = 0;
   while(1)
   {
      Power = Power * 2;
      if(Result > 9000000000) flag = 1;
      else
         Result = (Result * Result) % P;
      if((Power > A) || flag)
      {
         flag = 0;
         i[j++] = OldResult;
         Result = OldResult = S;
         A = A - (Power/2);
         Power = 1;
      }
      else if(Power < A)
         OldResult = Result;
      if(A == 0) break;
      else if(A == Power)
      {
         i[j++] = Result;
         break;
      }
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    }
    Result = 1;
    for(k = 0; k < j; k++) Result = ((Result * i[k]) % P);
    return Result;
}
```

PermutationLength computes the length of the permutation of cardinality Card. The function computes in the first iteration 2, second iteration 2*3, ith iteration 2*3*4*..i until an iteration j is such that j! ≥ Card. Therefore, the function returns j which is $\Theta(P_j)$ described in 3.17.1.

```
__int64 PermutationLength(__int64 card)
{
    __int64 y = 2;
    __int64 x = 1;
    do{ x = x * y++;}
    while(x < card);
    return y - 1;
}
```

This function returns n! in a 64 bit integer. This function is important since almost all C library functions computing factorials do so with long integers which are only 32 bits long.

```
__int64 Factorial(INT n)
{
    long y = 2;
    __int64 x = 1;
    do{ x = x * y++;}
    while(y <= n);
    return x;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bijection B gives the corresponding permutation in S1 according to the key k (lexicographic position) and the cardinality of the set of keys Card. The function B is the direct implementation of function B defined in 3.18. We can verify the complexity of function B. We can see that function B uses while loops that have at most n = Length iterations. The most complex while loop is a loop that has an embedded while loop a complexity of O(n) where n = Length. As a reminder n = Length is the Length of the permutation that is used to perform plain text transformation. As stated in 4.5.8, it is totally safe to use permutations of length 65 to perform bit permutations. Therefore the function would have to perform at most a $65^2 = 4225$ where a is less than 4 since we have at most 4 while loops in the function B. Nowadays most PC's achieve at least a speed of 15 MIPS. If each while iteration takes 1 cycle, it would take 4 * 4225 / 15000000 = 0.001126666666667 which takes approximately one hundredth of a second. For the code that computes the permutation according to its lexicographic position in O(n) see functions: `GetLehmerCode(..)`and `LehmercodeToPerm(..);`

```
void B(__int64 card, __int64 k, INT *S1)//, INT *p)
{
   __int64 q0;
   __int64 r0;
   __int64 Length = PermutationLength(card);
   __int64 x = 1;
   INT S2[50], S3[50];
   int y = 0;
   Z_ARRAY(S2, (int) Length);
   while(y < Length)
   {
      S3[y] = y;
      y++;
   }
   S3[y] = -1;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    S1[Length] = S2[Length] = -1;
    __int64 w = Factorial((int) (Length - x));
    x ++;
    q0 = (k/w);
    r0 = k - (q0 * w);
    S1[x - 2] =  (int) q0;
    S2[q0] =  -1;
    int z = 0;
    y = 0;
    while(y < Length)
    {
       if(S2[y] != -1)
       {
          S3[z] = y;
          z++;
       }
       y++;
    }

    do
    {
       __int64 w = Factorial((int) (Length - x));
       x++;
       q0 = (r0/w);
       r0 = r0 - (q0*w);
       S1[x - 2] = S3[q0] ;
       S2[S3[q0]] =  -1;
       y = z = 0;
       while(y < Length)
       {
          if(S2[y] != -1)
          {
             S3[z] = y;
             z++;
          }
          y++;
       }
    }
    while(x <= Length);
}
```

The function CryptoByte transforms the plain text in Buf1 using the permutation p. The parameter Length is the length of the array Buf1. For convenience, Buf1 is padded with characters '/' if the length of the buffer is not a multiple of the cardinality of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

permutation p. The function follows the algorithm fully described in 4.5.1. We can clearly see that the CryptoByte function has a linear complexity Length.

```
void CryptoByte(
     char *Buf1, char *Result, __int64 Length,
     __int64*RtrnLen, INT *p)
{
     char Pad[100], Buf[1000];
     int x, Crd = Card(p);
     __int64 w = Crd - (Length % Crd);
     memset(Pad, '/', (unsigned long) w);
     Pad[w] = '\0';
     memset(Buf, '\0', (unsigned long) (Length + w + 1));
     memcpy(Buf, Buf1, (unsigned long)Length);
     strcat(Buf, Pad);
     memset(Result, '\0', (unsigned long)(Length + w + 1));
     int Chunk = (int) (Length + w) / Crd;
     int y = 0, z = 0;
     while( y < Chunk)
     {
          for(x = 0; x < Crd; x++)
               Result[z + x] = Buf[z + p[x]];
          z = z + x;
          y ++;
     }
     *RtrnLen = Length + w;
}
```

The function DeCryptoByte transforms the cipher text in Buf1 using the permutation p. The parameter Length is the length of the array Buf1. The function follows the algorithm fully described in 4.5.3. We can clearly see that the DeCryptoByte function has a linear complexity Length.

```
char * DeCryptoByte(char *Buf1, char *Result, long Length,
INT *p)
{
     char *Pad, *Buf;
     int x, Crd = Card(p), w = Crd - (Length % Crd);
     Pad = (char *) malloc (w + 1);
     memset(Pad, '/', w);
     Pad[w] = '\0';
     Buf =  (char *) malloc (Length + w + 1);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        memset(Buf, '\0', Length + w + 1);
        memcpy(Buf, Buf1, Length);
        strcat(Buf, Pad);
        memset(Result, '\0', Length + w + 1);
        int Chunk = (Length + w) / Crd, y = 0, z = 0;
        while( y < Chunk)
        {
             for(x = 0; x < Crd; x++)
                  Result[z + p[x]] = Buf[z + x];
             z = z + x;
             y ++;
        }
        free(Pad);
        free(Buf);
}
```

The function TransformBit, transforms the buffer Buf into a string of zeros and ones. It returns the length of the string of zeros and ones in RtrnLength.

Example:

If Buf = "AB" then the string "0101100001101101" is returned. In general if Buf has a byte length of n then a string of 0's and 1's of length n * 8 is returned.

```
void TransformBit(char *Buf, char *Result, long Length,
__int64 *RtrnLength)
{
     __int64 BitSize = Length * 8;
     __int64 x;
     int Masks[8] = {0x80, 0x40, 0x20, 0x10, 0x8, 0x4, 0x2,
0x1};
     int WhatByte = 0, WhatMask = 0;
     for(x = 0; x < BitSize; x++)
     {
          WhatByte = (int) ( x / 8 );
          WhatMask = (int) ( x % 8 );
          if((Buf[WhatByte] & Masks[WhatMask]))
               Result[x] = '1';
          else Result[x] = '0';
     }
     Result[BitSize] = '\0';
     *RtrnLength =  BitSize;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The function TransformByte is the counterpart of TransformBit, it transforms the buffer Buf which is a string of 0s and 1s to its ASCII representation equivalent. It returns the length of the ASCII represented string in RtrnLength.

Example:

If Buf = "0101100001101101" then the string "AB" is returned. In general if Buf has a byte length of n then the ASCII equivalent string has a length n /8..

```
void TransformByte(char *Buf, char *Result, long Length,
__int64 *RtrnLength)
{
     int Masks[8] = {0x80, 0x40, 0x20, 0x10, 0x8, 0x4, 0x2,
0x1};
     __int64 ByteSize = Length / 8;
     __int64 x;
     int y;
     char Byte = (char) 0x0;
     for(x = 0; x < ByteSize; x++)
     {
          for(y = 0; y < 8; y++)
          {
               if(Buf[(x * 8) + y] == '1')
                    Byte = Byte | Masks[y];
          }
          Byte = Byte % 127;
          if(Byte < 0) Byte = -Byte;
          if(Byte < 33)
               Byte = Byte + 33;
          if(Byte == 7)
               int d = 0;
          Result[x] = Byte;
          Byte = (char) 0x0;
     }
     Result[ByteSize] = '\0';
     *RtrnLength =  ByteSize;
}
```

The function returns a set of 10, 32 bit signed integers using the rand function.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This function is used to generate big keys in order to pass the barrier of 64 bit integers of most popular compilers.

Since the rand function cannot generate long integers but only integers, 20 random integers are stored in an array of 20 integers. Then 10 long integers are formed using the array of 20 integers

```
void MakeBigKeys(unsigned long *Res2)
{
   int i;
   int Res1[20];
   srand( (unsigned)time( NULL ) );
   for( i = 0; i < 20;i++ )
      Res1[i] = rand();
   int x = 0;
   for( i = 0; i < 20;i = i + 2 )
   {
      Res1[i+1] = Res1[i+1] * 10;
      if(Res1[i+1] > 10)
         Res1[i+1] = Res1[i+1] * 100;
      else if(Res1[i+1] > 100)
         Res1[i+1] = Res1[i+1] * 1000;
      Res2[x] = Res1[i] + Res1[i+1];
      //printf( "%lu\n", Res2[x]);
      x++;
   }
   int c = 0;
}
```

The function ComputeBigKeys returns a set of 10 Diffie-Hellman public keys in DHKeys. It uses the function DEKeyToSend described above to computes the 10 public keys.

```
void ComputeBigKeys
   (unsigned long *BigKeys, __int64 *DHKeys, __int64 p,
__int64 s)
{
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
   int i;
   for( i = 0; i < 10;i++ )
   {
      DHKeys[i] = DEKeyToSend(p, s, BigKeys[i]);
   }
}
```

The function ComputePrivateKeys returns a set of 10 Diffie-Hellman private keys in PrivKeys. It uses the function DEKeyToSend described above to compute the 10 private keys. The set of keys in DHKeys had to be computed with ComputeBigKeys.

Example:

The User computes a set of public keys using ComputeBigKeys and the public keys p and s and a set of private keys stored in BigKeys.

The Verifier computes a set of public keys using ComputeBigKeys and the public keys p and s and a set of private keys stored in BigKeys.

Then the User receives the set of public keys from the Verifier and uses the function ComputePrivateKeys.

At the same time the Verifier receives the set of public keys from the User and uses the function ComputePrivateKeys.

Now the User and the Verifier should have the same set of private keys in the array PrivKey.

```
void ComputePrivateKeys
   (__int64 *DHKeys, __int64 *PrivKeys, __int64 p, __int64 s)
{
   int i;
   for( i = 0; i < 10;i++ )
   {
      PrivKeys[i] = DEKeyToSend(p, s, DHKeys[i]);
   }
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The function ComputeOneKey returns a 19 digit private key according to the set of keys stored in PrivKeys.

```
__int64 ComputeOneKey(__int64 *PrivKeys, __int64 Prime)
{
     int i = 0, s = 0;
     __int64 t = 10, Res = 0, IntKey;
     for( i = 0; i < 10; i = i + 2 )
     {
          while(1)
          {
               if(Res < t)
               {
                    IntKey = (t * 10) % Prime;
                    IntKey = (IntKey * PrivKeys[i]) * Prime;
                    Res = (IntKey + Res) % Prime;
                    break;
               }
               t = t * 10;
          }
     }
     if(Res < 0) Res = -Res;
     if(Res > Prime) return ((Res - Prime) % Prime);
     return Res;
}
```

The following section is the implementation of finding a permutation according to its lexicographic position in O(n)

```
#include <stdio.h>
#include <math.h>
#include <malloc.h>

__int64 factoriel(__int64 n)
{
   int x = 1, y = 2;
   while(y <= n)
   {
      x = y * x;
      y++;
   }
   return x;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void GetLehmerCode(__int64 index, __int64 n, __int64
*LehmerCode)
{
   __int64 b;
   int x = 0;
   while(--n)
   {
      b = factoriel(n);
      LehmerCode[x++] = index / b;
      index = index % b;
   }
}


void GetPermBase(__int64 N, __int64 *PermBase)
{
   __int64 Base = N + 1;
   *PermBase = N;
   int x;
   for(x = (N - 1); x > 0; x--)
   {
      *PermBase = (x * Base) + (*PermBase);
      Base = Base * (N + 1);

   }
}

void LehmercodeToPerm(__int64 N, __int64 *LehmerCode, __int64
*Permutation)
{
   __int64 PermDecimal, *Baseth;
   Baseth = (__int64 *) malloc (sizeof(__int64) * N);
   GetPermBase(N, &PermDecimal);
   Baseth[0] = N + 1;
   int x;
   for(x = 1; x < N; x++)
      Baseth[x] = Baseth[x - 1] * (N + 1);
   __int64 i, y = 1, a, b, c = PermDecimal, p;
   for(x = 0; x < N - 1; x++)
   {
      i = (Baseth[N - LehmerCode[x] - 1] / y);
      a = c % i;
      b = c % (i / (N + 1));
      p = (a - b) / (i / (N + 1));
      Permutation[x] = p;
      c = ((c - a) / (N + 1)) + b;
      y = y * (N + 1);
   }
   Permutation[x] = c;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    free(Baseth);
}

void PrintResult(__int64 N, __int64 *LehmerCode, __int64
*Permutation)
{
    int x;
    printf("LehmerCode: ");
    for(x = 0; x < N - 1; x++)
        printf("%ld ", LehmerCode[x]);
    printf(" -> Permutation: ");
    for(x = 0; x < N; x++)
        printf("%ld ", Permutation[x]);
    printf("\n");
}

void main()
{
    __int64 N = 11, Index = 0;
    __int64 LehmerCode[20], Permutation[20];
    __int64 p;
    while(Index < 24)
    {
        GetLehmerCode(Index++, N, LehmerCode);
        LehmercodeToPerm(N, LehmerCode, Permutation);
        //GetPermBase(N, LehmerCode, Permutation);
        PrintResult(N, LehmerCode, Permutation);

    }
}
```

5.9 An example using the crypto library:

```
void main()
{
    unsigned long UserKeys[10], VerifKeys[10];
    __int64 DHUserK[10], DHVerifK[10];
    __int64 PUserK[10], PVerifK[10];
    INT SETX1[50], SETX2[50];
    MakeBigKeys(VerifKeys);
    MakeBigKeys(UserKeys);
    ComputeBigKeys(VerifKeys, DHVerifK,11318687, 17);
    ComputeBigKeys(UserKeys, DHUserK,11318687, 17);
    ComputePrivateKeys(DHVerifK, PUserK,11318687, 17);
    ComputePrivateKeys(DHUserK, PVerifK,11318687, 17);
    __int64 UserK = ComputeOneKey(PUserK);
    __int64 VerifK = ComputeOneKey(PVerifK);
    F(11138674513759432 14, UserK, SETX1);
    GenerateDn(SETX1, SETX2);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    char Buf[100];
    printf("Enter Your Password: ");
    gets(Buf);
    int Len = strlen(Buf);
    __int64 RtrnLen;

    char *Temp1, *Temp2, *Temp3, *Temp4;
    __int64 x = GetLengthTransformByte(Len, SETX2);
    Temp1 =  (char *) malloc ( x + 1);
    memcpy(Temp1, CryptoByte(Buf, Len, &RtrnLen, SETX2), x);
    Temp1[x] = '\0';

    x = GetLengthTransformBit(x);
    Temp2 =  (char *) malloc ( x + 1);
    memcpy(Temp2, TransformBit(Temp1, RtrnLen, &RtrnLen), x);
    Temp2[x] = '\0';

    Temp3 =  (char *) malloc (x + 1);
    memcpy(Temp3, CryptoByte(Temp2, x, &RtrnLen, SETX2), x);
    Temp3[x] = '\0';

    x = (x/8);
    Temp4 =  (char *) malloc ( x + 1);
    memcpy(Temp4, TransformByte(Temp3, x * 8, &RtrnLen), x);
    Temp4[x] = '\0';
    // end of encryption
    printf("%s\n", Temp4);
    //decryption
    x = GetLengthTransformBit(x);
    Temp2 =  (char *) malloc ( x + 1);
    memcpy(Temp2, TransformBit(Temp4, RtrnLen, &RtrnLen), x);
    Temp2[x] = '\0';

    Temp3 =  (char *) malloc (x + 1);
    memcpy(Temp3, DeCryptoByte(Temp2, x, SETX2), x);
    Temp3[x] = '\0';

    x = (x/8);
    Temp4 =  (char *) malloc ( x + 1);
    memcpy(Temp4, TransformByte(Temp3, x * 8, &RtrnLen), x);
    Temp4[x] = '\0';

    Temp1 =  (char *) malloc (x + 1);
    memcpy(Temp1, DeCryptoByte(Temp4, x, SETX2), x);
    Temp1[x] = '\0';
    printf("%s\n", Temp1);
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 6

6. Advantage of the authentication schema over other known algorithms:

We are going to compare the authentication algorithm presented in chapter 5 to widely used authentication schema, and we will prove that none of the existing authentication schema takes into consideration the use of badly chosen passwords (see Chapter 1).

6.1 Sending plain passwords:

Old IBM machines and UNIX machines still send clear passwords over a network to perform authentication and authorization.

These methods, of course, are the worst since they don't take into consideration badly chosen passwords, and they don't prevent taping of the line where the password is sent. Our methodology does not send a clear password and does take into consideration badly chosen passwords, since a clear password is always transformed in order to be used.

6.2 Sending passwords using public key encryption:

This methodology has the advantage that, if somebody taps the line, the password is encrypted and he cannot know the password. But this methodology does not prevent anybody from sending the same encrypted password and being authenticated and authorized if he taps a line and captures the encrypted password, since the key is public. Also, it does not take care of badly chosen passwords: since the public key is known, any hacker can encrypt his hit list of passwords and submit them to the Verifier.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.3 Key crunching:

Key crunching is a technique that converts easy to remember text strings into random keys. It uses a one-way hash function (Preneel, 1993) to transform an arbitrary length text string into a pseudo-random bit string. The text string is often referred to as a passphrase. For a 64 bit key, a passphrase of about 10 to 15 normal English words should be sufficient. One could use key crunching to transform a password, unfortunately the password has to be at least 10 words. With our technique we only need to remember one password of regular length.

6.4 Wide-Mouth Frog:

The Wide-Mouth Frog protocol (Burrows, 1990) uses a trusted server called T. We are going to describe the protocol by showing how A authenticate himself to B using T.

1. A concatenates a timestamp t_a, B's password and a random session key and encrypts the whole message with the key she shares with T. She sends this to T with a public identifier.

2. T concatenates a time stamp t_b, A's password and the random session key and encrypts the whole message with the key he shares with B and sends it to B.

3. B decrypts A's password and the key that he will use to communicate if he recognizes A.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We can see several problems:

The algorithm requires a trusted party that could be easily bribed without A or B knowing it. Since our authentication scheme does not use a trusted party the weakness of the trusted party being bribed is automatically eliminated.

Also, in the process the real password is encrypted, therefore easily guessed if badly chosen. Also, the time stamp can be found easily if the message is intercepted. Thus, the plaintext secrecy resides only in the random session key. Therefore the message can be attacked using a part of the encrypted message that can be easily guessed (password and time stamp). In our protocols none of the plaintext can be guessed since each time it uses a random key and does not send any encrypted password.

But the biggest assumption made in this protocol is that A can generate good session keys and we know that random numbers are not easy to generate. In our system, even if the keys k and k_s are badly randomly generated, a cracker would have to find k_1, k_2 and k_3 to defy the authentication schema. Since those keys have to be generated once, they can be generated one time from a true random source by passing all the known random tests (Knuth, 1981). A cannot afford to try to generate a key that is truly random by performing all the time consuming tests each time she wishes to communicate with B.

6.5 Other protocols using a trusted party:

The following widely used protocols also use a trusted party:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Yahalom, Needham and Shroeder (Needham, 1987), Otway-Rees (1987), SPX (Tardo, 1991) all use a trusted party and the security of the authentication resides in generating random keys for each session. Therefore, they all share the same weaknesses.

6.6 PGP:

PGP Authentication, according to Cooper, et. al., (pp. 111-112):

1. The sender creates a message.
2. PGP uses MD5 to generate a 128 bit hash code of the message.
3. The sender specifies the private key to be used for this operation and provides a passphrase, enabling PGP to decrypt the sender's private key.
4. PGP encrypts the hash code with RSA using the sender's private key and attaches the result to the message. The key ID of the corresponding sender's public key is attached to the signature.

Handling of the signature by the receiving PGP:

1. PGP takes the key ID attached to the signature and uses that to grab the correct public key from the public key ring.
2. PGP uses RSA with the sender's public key to decrypt and recover the hash code.
3. PGP generates a new hash code for the message and compares it with the decrypted hash code. If the 2 match, the message is accepted as authentic.

The MD5 hash code used by PGP is a one way function that produces a 128 bit hash (Shneier, 1991).

RSA is the encryption algorithm used by PGP. The security of RSA is based on the difficulty of factoring large numbers and the generation of big prime numbers (Gardner, 1977).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.6.1 PGP authentication weaknesses:

PGP uses a passphrase. It is called a passphrase rather than a password because it does not need to be a single world. Unfortunately, 80 % of PGP users (Cooper, et.al., 1995) do not use a sentence or phrase, they still use a single word as a password. Consequently, password crackers can use their technique 80% of the time. Since PGP uses the passphrase to generate a 128-bit IDEA key and since badly chosen passwords are used at least 15% of the time (see 1.3) it is obvious that PGP restricts its 128-bit IDEA key space. Also, the MD5 algorithm requires that the input text should be 512 bits to produce a 128 bit hash value. Since 512 bits is equivalent to a word of 64 letters, the user would have to remember a passphrase of 64 letters, otherwise the passphrase has to be padded. The average computer user is not expected to remember passphrases of more than 12 letters. Therefore, 4/5 of the passphrase has to be padded with a padding text known by the sender and the receiver. Consequently 4/5 of the security depends on the security of the padding and 1/5 on the security of a passphrase that is badly chosen 15% of the time.

If one knows a PGP private key, one can then fake authentication. As we see in the presentation of the PGP algorithm, a PGP private key is only protected by the passphrase, i.e., a badly chosen password in most cases, therefore the weakness of PGP still resides in the fact that it does not solve the problem of badly chosen passwords, unlike our authentication schema.

6.7 Using one way hashing functions for password transformation:

Here is the list of the most secure one way functions currently used (Menezes, 1996):

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Hash function	n	m
MD4	512	128
MD5	512	128
RIPEMD 128	512	128
SHA-1, RIPEMD 160	512	128

TABLE 11: HASH FUNCTIONS (SIZE OF THE INPUT AND THE OUTPUT)

n is the size of the input key and m is the size of the output key. Passcode generators have been proposed to generate passcodes (Davies, 1989). They use a hash function that transforms a PIN number of a length no more than 8. As we see in the preceding table the input text for using any secure hashing function has to be of at least 64 bytes. Therefore any algorithm that uses a proven secured hash function has to input a string of 64 bytes, otherwise the input text has to be padded. The advantages of our password transformation are that it requires a password of a length of 12 bytes, and that the complexity of the algorithm for the password transformation is linear.

6.8 Kerberos:

6.8.1 The Kerberos protocol:

The Kerberos protocol is one of the most widely used systems to perform authentication and to grant services to Users (Kohl, 1991).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A User, X, logs on to the workstation and requests access to a specific server. The User sends a message to the Authentication Server (AS) that includes, the User's ID and request for certain services. Those services will be delivered by the AS by giving the User a ticket-granting-ticket (TGT). Then the server checks its database to find the password of this User. Next, the AS responds with a one time encryption key and a TGT encrypted with the user's password. When the message returns to the User, the User is prompted to give his or her password. The User generates the key and attempts to decrypt the incoming message.

The AS has been able to verify the User's identity because this User knows the correct password, but it has been done in such a way that the password is never passed over the network. The encryption uses DES.

6.8.2 System weaknesses:

The AS uses the User's password to encrypt the TGT and the key that will be used later on by the User to use his or her TGT. This method does not prevent the use of badly chosen passwords. An experienced cracker could tap a line and then capture the encrypted message. He could then use a hit list of passwords to try to decrypt the message sent by the AS to the User. As we described in the first chapter, the pool of keys to decrypt a message encrypted with a password is reduced tremendously and decryption can be done in reasonable time. Once the password is known, a cracker can tap a line, decrypt a User's TGT and the key sent by the AS when authentication is performed, and use the TGT to acquire the User's services. Otherwise the protocol shares some similarities with our protocol.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Both protocols use a one time key to avoid a cracker simulating message sending during other sessions.

The password is never sent over the network.

Clearly, the weak point of the system is using the exact password to encrypt a message that contains the core information to access a system.

6.9 General advantages of our authentication scheme:

1) It uses a one time key that prevents a User from pretending to be another User operating from a workstation.

2) The password is never sent over the network.

3) No keys, encrypted or not, are sent over the network.

4) The password used has been transformed such that no cracker could use a hit list.

5) The transformation is done with a linear complexity.

6) The Key computation for the password transformation is done once and has a complexity of O(n) where n is the size of the expected password length.

6.9.1 An apparent weakness:

The User needs a private key to transform her password. But as we have already described in the algorithm, the private key can be stored in a smart card or any device no bigger than a credit card.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We also suggested that the private key could be encrypted using the User's password. This would give the User enough time to signal to the authentication verifier that he had lost his card, while a potential cracker could be trying to decrypt his private key.

One could say that we need an authentication algorithm to prove that we are a User who has lost his smart card. A person could pretend that he had lost a smart card. The worst that could happen is that the User would lose the system services for a while.

6.9.2 Why use a personal password?

A question that is commonly asked is: Why use a personal password, since a portable device is used to store private keys? Why not use only private keys?

Indeed we could perform authentication very securely without using a personal password, but using a personal password has many advantages.

Suppose that an authentication system uses a private key stored on a smart card and no personal password. If the User looses the smart card, then anybody who found the smart card could enter the User's account and use it.

Therefore, if the User's smart card requires a password, then if the User looses the smart card anybody who finds the smart card would have to guess the User's password.

Nevertheless, this does not prevent the smart card from being found by a smart cracker. Therefore, the smart cracker could assume that the User was using a bad password. Here is the advantage of using a personal password.

Suppose that the User was using a bad password (see Table 1: (bad passwords list)). The cracker would have to guess a lot of passwords in order to be successfully authenticated.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Thus, the verifier could keep track of how many times an authentication has failed for the same User during a certain lapse of time. The Verifier could refuse to perform more authentications after a certain number of failed attempts, considering that the User is not the right person. Also, the verifier could try to trace the fake user electronically.

6.10 Other uses of password transformation:

We proved that our password transformation schema would resist plaintext attack and chosen plain text attack. We also proved that the changes whereby a real word would be transformed into an unknown word are absolutely astronomical. Therefore, since the transformation algorithm has a linear complexity it would be very useful prior to encryption to transform with our methodology the plaintext that has yet to be encrypted. This would certainly destroy at least all the digram statistics. Also, we can leave as an open question the capabilities of such transformations in terms of pure encryption.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Conclusion

In the preceding discussion we gave a secure authentication algorithm that can use badly chosen passwords by transforming them into secure ones. We proved that the password transformation algorithms have a complexity of no more than O(n) where n is the length of a password when the permutation used for password transformation is already known. We also proved that obtaining the permutation from a key has a complexity O(n) where n is the length of the permutation. We showed that the password transformation algorithm can use any key length regardless of the computational capacity of the system in use, unlike other algorithms which have to use adapted hardware to perform their algorithm when bigger keys must be used.

We proved that the password transformation that is used for the authentication algorithm defies statistical analysis (by using homophonic code) and resists plain text and chosen plain text attack.

We also proved that it can resist the crackers' techniques that are generally used to crack a password.

We described an authentication technique that uses password transformation and we proved that the authentication technique uses algorithms that have a complexity of no more than O(n) where n is the size of the biggest key used by the algorithm. We also proved that the security of the authentication resides solely in the security of the pseudo-random generator used to generate keys and in the security of the public key exchange used to exchange private keys.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We compared our authentication schema against other widely used authentication algorithms and showed that either none of them solve the problem of badly chosen passwords or that they require a passphrase or password 5 time larger than ours. We also showed that the algorithm of password transformation can be used in conjunction with other encryption algorithms to strengthen them against cryptanalysis attacks such as statistical attacks, plain text and chosen plain text attacks. Therefore, we sincerely think that our authentication algorithm will help the average computer system user to legitimately use a password that is easy to remember.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Here is the program using the crypto library that generated 1 000 000 password transformation using random keys:

```
void main()
{
   int x = 0, y = 0, z = 0;
   unsigned int Num = (time( NULL ));
   srand(unsigned (time( NULL )));
   FILE *fp;
   fp = fopen("TEST1.LOG", "a+");
   char *Templ = NULL, *Temp2 = NULL, *Temp3 = NULL, *Temp4 =
NULL;
   Templ = (char *) malloc (1000);
   Temp2 = (char *) malloc (1000);
   Temp3 = (char *) malloc (1000);
   Temp4 = (char *) malloc (1000);
   while(y++ < 1000000)
   {
      INT Pn[] = {0,1,2,3,-1}, p[] = {0,1,3,2,-1};
      unsigned long ClientKeys[10], VerifKeys[10];
      __int64 DHClientK[10], DHVerifK[10];
      __int64 PClientK[10], PVerifK[10];
      INT SETX1[50], SETX2[50];
      MakeBigKeys(VerifKeys);
      MakeBigKeys(ClientKeys);
      ComputeBigKeys(VerifKeys, DHVerifK,11318687, 17);
      ComputeBigKeys(ClientKeys, DHClientK,11318687, 17);
      ComputePrivateKeys(DHVerifK, PClientK,11318687, 17);
      ComputePrivateKeys(DHClientK, PVerifK,11318687, 17);
      __int64 ClientK = ComputeOneKey(PClientK,
11138674513759432l4);
      __int64 VerifK = ComputeOneKey(PVerifK,
11138674513759432l4);
      ClientK = (ClientK + 100) % 1113867451375943214;
      F(1113867451375943214, ClientK, SETX1);
      GenerateDn(SETX1, SETX2);
      char Buf[100];
      strcpy(Buf, "ABADIE");
      int Len = strlen(Buf);
      __int64 RtrnLen, x = GetLengthTransformByte(Len,
SETX2);
      CryptoByte(Buf, Templ, Len, &RtrnLen, SETX2);
      Templ[x] = '\0';
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      x = GetLengthTransformBit(x);
      TransformBit(Temp1, Temp2, RtrnLen, &RtrnLen);
      Temp2[x] = '\0';
      CryptoByte(Temp2, Temp3, x, &RtrnLen, SETX2);
      Temp3[x] = '\0';
      x = (x/8);
      TransformByte(Temp3, Temp4, x * 8, &RtrnLen);
      Temp4[x] = '\0';
      if(!(y % 100))
      {
        printf("%s\n", Temp4);
        fflush(stdout);
      }
      fprintf(fp, "%s ", Temp4);
      if(z == 3)
      {
        fprintf(fp, "\n");
        z = 0;
      }
      z++;
      fflush(fp);
    }
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Exerpt of the first few thousand transformations of the word "ABADIE":

```
P#:|*(/a/'(/i!:8?cg! 3#)C>|xiC>89@/$3#)C> .#",Y.Z|4_*B'f/*B$$r
7)09$YU/;e]~S'Oy6&9K >$@@)>$_@)>J!B(:$^Z) ,!(V#Y/SV'IQVZRyACV&
k<c5fk<c5dZ)>#fe4o.h HNltl(M2t*hM*DM1OtD2 z`M\Obr1\.tN2]RzN2U.
TrT,kPu8/?p/H18t.H"i yyJn2yyJn2k1AAWY7A/' =Tk)j"J*7a@^7X7C7i7a
a'A?%Ada?%)/`i1A4cc| "my\l'0SDey;Wd2y;Wd2 ;!7$aw*W><W*WCASBBCE
t>i?D4=3+>->i?D:2#92 x-7/)b*}/)FT5@)^vu@) !d,SL!d*4P1T"4L?DCSP
jMy5:I?bBZ7MuJ;i7f6; 13%1/S"7a^14ul/S_&1/ @jAEG@&"%G>.$'_@!!%A
^oua%f2ur-L@;KxL@ka% <pK,RK-k,UK-klU@,+,E Ki1:6KIN%Sck!*5Ki@79
&d+Ny+(,4!6=)4b+(.M} 4?-!'O6E>-K>M>%_YA/w g4>~deL7T.v4KT&R`,T&
~AY<NV8)_-~WEO5V8)4M gav~1@.*~1H.+(?YsV"2 v$A+S#AJdX8!-*;+IA+S
:893P#ac3L4y>;"chs;" 86*O5!=&0%@:%/*!=/a& \%$2\{[-QLsx7AL{s"7\
J!0'*'CAGFy#d4*'cD4* z,9u+m,?32(;{MN4;~1v bL2eKEBQw[>L/eKILQu_
=f2;09xKnp=f?Lr@3;:. S-K25AWX@;s6G2Ks8XQO #0,r:3Ow>B3O}b2~1*r9
hp'_IbRz_I6LzN<az^_m !v34$Q\^4$o+2;(!*;4< 3A?/y"`EQ&"`E%ym`a{&
9[I?u9\IOU5MT4r5MY?6 @;!UJ$;!U2%3M{2%3IUn 6u>'86u>7D7,/787,%gQ
CM+&OC.kL4GN6&OB',&O E~._CB}4!9B}4!9l}3$, ZI)!tn5)-Sn5)I\o=!w7
5DisD9^4<%@&4;|5T*{% A,s"N?l(=3?lOCM?E5^^ 7v+)K/v/!x16!MtR*+){
(3+3oOk+3T`2<8oOk<8k K@gU9YGgV&YGgECYGgU= <:aB/,N@Q/88cA-,N;n/
x}#'2{m/Gjx})E2tm)uR oO.E#o'/>#8(Vd*o'*c3 3"$i'$VMy99A#i'h`+W:
p~%*_t~%I>p>=Kcp}%I> 6!T^7?cR~Y?cR#YH#|^2 .F*%$-T4>X.F-%*&Fwm(
B98>wP21<bD;(R`l:9>w gqUg-QuKc$gqRom7!R+i r'mY[s'46HuAm5C>Gm5C
%F1daFSKDaFsKDaF?1(q &Rw%%j7"26FBUI=FBW%5 -010@z!9!{%0I]E?y2_;
)]wp8AgoKii]lu`i]nki A0iF>!!M?s";S'>a[4'> &"0[5{AH/X{ACiTAIF_l
v'3S('(5=%#86-#NI4S& ^21!Td/V"T|(F!/|80ov 4iH&C0o46g1y46g27.A7
d?l]EK6-N"CT%N},/L]J Q({/:U(z/#>~6/#M~6}" @(EF}4\EF}L8*<E@7UFQ
FDKHKrxAH,.$QVQ/daH, -P.rz<=%3<<=#zz<=#:z W!as9X#giV"=!s9)693=
p,/304z3X/4z5"a1>39/ @x;X1:p<2G:p:`':ps:% 2Pk#o2B}o+5C-o+5C)+Z
7%^avV%brfDXBbf3e^av 0=2!1?_"*9&s"mG&s"{K OzV/?1%v.?K97>AK:'CE
p!x4[A#&s5'>z{''>z{' x>K6kpI-Uk]71jC]73jc *9>VRh9w@-!U?wTh)?wT
A0?9g-s?:gL%?=#dg>=# Uk]67UkY.^]cYR:!X=2' \<9&D)=;ZD!^:(8uV9E8
5P!77;XH[}&Fwk7;H_77 0>uw+B7i7.b>2/.b>87s MF(e2M>"20MF(:"E-#o0
)6=Yr,&>Bv6:+bv&;#Bv $0,$@@a3#>]AN#>$q3#> 9aMW%W@.W%'=0Q[0=.W%
`Oi'7p5*'7!NSj/P]DF3 ,:Y*:rc[!+Z:<*0ccY#+ 2-USP1xKW~5f[W~6bU4p
bMwF0LM7E.6pAEqlu6GN R;2!Q*4@!S*C1?A^4<!S nB-9LBCj{)3B-yWBCn>9
38PzB38Gl%31E;d31Mt# Q1=335!:23ymi2My3l~4 aC75$C=W5$aCQ7}`G7>c
x=xG2Pdz&2HA|E#Pez&2 -4n3g)Tx<)<4D,i<4Nuo Cc2;"\"k;2_Ck;2d^5Uk
)`!('<?~G$4=F=-4C\=% %w!]%%J#_![W%O,%M#][ =lwdx]Xj52=l}-1@n_%x
3qd^=3qb1;+,|::3qpYG s|*Gqe**LFmV*+amR*+a **.xo5-}"85<xcH5-}"8
`"1CLiE@6H-;!6H-,~#L {3/7%>j>7%>j>(^Jp,?V `C.(vHcu$7BA+|jfbE$7
]tAK5Yr>Q"RzkO5Yr6gI P7z'6cCI>~rWZ'6P8x.6 h;f)$-BF)$p[fh}H'&0'
K"t>%4+57$$~4:%86%80 73!W#D+7,);u7$(;u3S$ (:6$ilW69j`0e9jlW69k
afu/)k^rKqBxTKuk^qNw 8y)*PD~%'9%y$*9$y$*9 j6*$mk)-$}k:.$})0*$i
~4wW6~49$:~4?.$vmw$: el>=#<18H>el>=#m5'wp @".g*(|)g'e"t;*(|*:7
fLKW<&$(,[k^K]<k^MZc mn>#$$>=3H25/#$ma&!L 0l=0K(v:HI(v>&I(v:t(
j5<:o\]6zv?]::o$12.< >o/F6scN+UWcF:C:.N>5 t.?TSC+-qS"`jAsC+*As
`1.?Md4.?MS%49VC&@?M "jE:&#$bL&8lE6''df]& .25="i1~-~31R5Ui1~)#
"o4r4rA6r3"ocDl,mF6: zxt{+x<3CJ)9d{+)9f0s =s]7~=c6-,<5.%~*7U;.
78O's!"28.Ox7]rOx7]S '5/kb'7+5b<W?+/'7)6b 7/?g'5/6$I%+>!'')5@Y
0vO3@MvO3@0VK2<>J+2< @4m".C4hb)=4.".:4.Q. )5^3&3R~s(?&nq6?&n1%
|4/z#8mRuk8mR728mR4y 1yB.>2+'&A)Bfe@4+3/= K3}F'o=!('K3{('L1t-3
>70<V.3-$2.0M$2>6MPY B]!A#FcuAcBk5/EgY?Ac 5+rT2=y;To]q49%Us>T2
V0a.OFA<0$IP%1}*pe0$ )j-ud)bnU/w#C(#)bbY? >bY0\DwFxiDwB*8FgA(j
9<K"U9[[4Q">{3U@&[3Q nG(oqj7c2ynC("q!s("q q!{Gn-i#_'*K6=%-i#G3
-wx:>-wL4:-WxrbGW44: .o29)m&R##1hSg7.nR<> 6k0i7@>.:;/`:8m@>.:3
XGD}=xupl4XGC}gHusYc 3[l(f{;$WQtV}Y23S}Y2 "l9]{p@+]{6(']{>(3>{
|}x>2]?tc2]?vkj>FN67 5_3C;=:~X:"e+C:`e+C: }1Kj!u2NJ{k[DJ@y[Kj!
fVw;&nwxWzn)g;?fVw;6 =:ax+4`^0*9=b18$qax+ H_u|5H#}~5)=6>l98}~5
\:UH>zQ5''zQu9>zQ5'' (?B#C:%B#C&^"#C},AF3 %0<D:V{z&%F{eD:dieD:
@]($7p8jK?tkR([p8:$[ "Ci#/2n>3?:R=3Q-!*#/ Ayk4|GV<H_G)a4~G)e*8
HU@?8&*A#GFUA#G&0`cI Mxl2{MTVYZ3xl2{?4P3- <OUpA=gqn5DHm#5=gEB,
)p?}*)3<00|P!ygxp?}* |0Q:1zb)*.'D6m/~"),- :R@<%J2n&AMS<*sJS4".
u6/3=U'`2=u8p;fu6%3R &=]#y63!M8)lOM8&<_#y <`C[k9*:.U'*9/T9*:QT
/+&]s(;@]S/<@X=,:?G- >2\1[1.A;7>2R1[/>a;7 B=/:A.L5Au*l>)5B=/:A
/E.6&=E.6&/".6&MWDu3 =y5or1$|ok5#|oa5"uov ~:1"u3I%2w#*$/6~:3l5
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ms]7T*wz']msh_ymsU7T
dp5-XD&@>1D.VM&h17M&
.>qAND!$RV&c"Z3&c(3#
=UN7%4B~,q>EN4[=UF-&
=@"7vC@#"2Ba#?o.7#"2
F$6i'jv[i'zx<jxNu(%*
`:Yj>?I"=2"a!;o?Ko+2
k&cLO6u7Rk<d/DK65*LO
9.:x3!&o,49.@|29.5)*
$%6o:#-;o*+%<[O#-;o+
#vdwNNa9~+H<t+_NedwJ
,"tM.{v*x3{rsL"2SsL"
,;l2Oj+F2G&;7"_X!:"_
F$$s'3d%'(V#%W!X?5W!
t2!3>69)3]o?#^@o?*3]
7b*qMw^WaIw^Wq]EnXL$
?baG0?bg*<@ZaA<2rz+<
)A]F"zQ?N#!R?N#!76FX
-+91F-+;!'1,F!guW{9@
+Y8Ya+Y18282'Y$kQ8\e
&V^6&|3T>{x3^6&xn5.{
d1q.fg1q&f(92>k(:2>k
37T_U37\s/99_U#+,*W
3S,R:#UxN:w"uR:#U|f}
r3\%D0ia-D0ia-Bu4anR
^V.#5@EM7N>T!'L`(!'L
yVP&$}PyvSyWQV#yWQV#
0/0Ko0/+a?7^mK/(?-0!
=,0eI=,.5I?,-Zv=`09}
0oI.Ec=q>Us=lhO3@i.E
`</c(Yd#{M`<*y3Yd#{M
X%qmwFNmBvj$amwz"i}s
/1>Jq*4BN4-0>"2.4@,$
?5E}1(iY<??5&];9=Y$;
`A]E2jNw&g~O8&g'O>E:
e}j:2aQZ'nE}kY2eYY:'
-$kE+,Al]4,bg:+T#g8+
_I&,Cg)8<_WFfa?WE;=^
`ji&!+HQ6{k1*&!c&1f{
Pr;'|t-+7$$V8P$5X+7$
TM]K'TMW&h,1.6h"MmK'
Va!@4W@3PlU>;84Qn;H5
A/7#=!,7%,?YMz?~\I#=
1&ls<|:a>:)"l3<x9ms(
(V2pTlIb&C,_"&C\V#6@
!,Qcs#?1cs!#2cs/B3S#
\zl4(4q|g901|*54qrw(
3h!.c386.s.AU.s59].c
~$9*(U0<00)0s.yU0:*(
#<{=Q>"(1)>255">"(1)
Qno#^0m(3^03&BPQmm3?
@]Or4I!8zmH#3rHIp6jm
3A[l%4!Y/z467Gl357/r
&!#)+Y~$)+I~L_3(!#;*
hZ&$(D}E4PHY&3Xd"0uV
xm$sA^44c@6<1975=3sA
8$8?Sb5$cT'(0!$b'A!$
>La:fF%a:v|%w_IF%ao*
yAA+!})"+{})"H8YXH{!
pSs%-7&55Rpq6%.v33L]
9tN:W5Br;U*;r:W&E|,@
P+F,B<##3lP+F}jP+N}j
"+M)a`9\zs".M)ah!>)a
}h"48P~]44PBY%EP~_$4
<Sd-?<Sl]*:wdK!9w"=?
W<E>I}2/?/,,1>A,?Q>A
9VTz/c[Nzog[Rb=-E.zo
Wew{D`m1QXw$q30Wew{D
Ai}'I'X%7}@"}'IPq}7]
,A6]70C6+?q,Mii{QS68
a1,IEIkLIEIy@i-P(dI;
2m^'`2*N*An-~&(n-z)(
cW5&tkWn)FkWjEVk<~*/
X/;^*D+|+*D,_A.D,\+S
GBd'&9Bd'&\/D'&:R`U<
boFw.b?*w.*@j87$@d2'
="%/%MR'bRm.W?3e.%/%
S!-_Zh,Y.Qh,Y.Qh,])u
&l-d8CL5bIF%5bIN|;to
8y!zz"5qHz+D1Xz8y!zz
F8\~6:'4~614\~6@(;Q,
p5/t>xDn<?9cl~?(8dz;
^A7;0}m)(J}m)'?ym?;:
-<0(s7ZO(sn:o(s-@0Yg
3(Oj'v`]P'n.UP8v._j'
C}a6#6-i>j2R<&#rP;&#
o?lu$G?d0YG?d}Mp@d}3
%7NUWK;2Q>%+nTcK%3Q>
E#8=0Gq>AIGq8-qA75%0
.-B#?.=`sTh>^#?h5N/;
{t>?[%#.x_%#6?_4f6?[
M"[b+J![rY-2lq2Iolq2
`Og#!['GP#['MP"[.L0&
:ACV7*%A*7:M8:%*%A6M
D4e+Mfb='u`J[jVfb=+m
)hUz(N[i+8%(7Z=N[kz(
4><ot=57K%!+3%(=57K%
oG-[^Mf`jLoG/:WMFr:W
gd*&<5te&B:P-&<5tfH-
%,EWg-E#Wg=ia8g9ma8g
18;IX-*;aX|:<5,|:<5$
l*+z%G;c{AW;-!_jA+{"
:i/j0V((0o4).-oR(00a
np6%:ZvZjc;v3%:nrNHK
Ka)GoGn/OOOn(2LG[00,
uq.db9)'e!,Q(e!9X)da
!r-5"!pJIM!pD-u!r%T*
9(=/"=)])FSc8(]Cb=/"
W*kA94$,aXO*K8Bv*&A9
/(-T'/(-31m~V"1/(+31
.:;xP.g!&/.g!-I.jiS(
oAhESpV"5PpV"$PQv6$S
!O&Fw+O9*)rF9$.+W9*)
%bMFg}C:/!q]:g}AYF
:Jl,A(JL,B:Jd,BNDh-w
%!oaOBe<C_~D4W1fd1aO
>fe:0>fi!csDMlosBA%D
STJ":->6G9STJ":rAJb6
j=M%7j=Mk39PM%7HNs%7
X~A()BjD"!Rj"(_BjD"!
KN%5"B@45"a!O'w5N-CB
Z`<WF/{=y-5[m)+5[gWf
aU)&.!\:h#7_#b!7^;j!
)650<8>thL)7S.)(6}.)
`8@<=`,+*mzo5+m\SK*m
pf),&p5(<0pD(,&Pj<,$
T2*b)0P5cAC&*b,G&5C&
<Se'Q9s6z.5S;'QV22Z.
]?'"S^31Lj]?!bm^*7LQ
!;6>]}y+6]][Tbe^%SZ9
/U0Zo/Grtne9rr.E#0Zo
+?g77+ZbBv+?gf7|@dBR
QX\@"1%X*>"G\Vb1%\:"
{dUT7['GT!&BFvoOj(vk
0"Y)kLO!"UO"R'iP"R6m
RZJA-VZJoR67fA-U7f_R
y|=N<04A(G,2IK7+nYK3
8C"m5I84U7Ox1]"(I4]"
2"(M&o$&l-6#jMk6_j.i
'~sY",~xib+~sY"}~Hw3
9(n{-<(K.p<-j~0<(B>1
"a,:X:Q^O.:Q^:R:a\Hp
tg$;btg-IB!/s)Ltg"+l
TN6{C.N6{Cd@6{C8!Gk=
?)>c#Z+95Hf/9#Qf/9#U
"Gy";!3i";!3w";o{o3!
"&1AI5{sQI6A!):5!oAI
-6xB9=\Vv?%\;B?-]7n1
c%%g,%@9/I==Tp:==X/M
3_8(@b68#+H))$>b67/=
4:!IM3vs(1L:#I(3vs[(
V1<";b=64<~66,Bb={";
:%a(:>sb{>-[=4f>sa!;
#A0@8N':0iN'60i0E3@/
;")q62IEh+2IDf#/HWq6
O+V^)",5X#!,TV~O*T'#
`$5_J}4[O6k4y<6k4y4>
\Lm^6L,LIA$tlIA44oMM
>7{b'<7!"Y[(!"Y?O|$"
2)}N`5)s%,t.kWi5)s#,
""}2A*D#v?`-=v?*D#v?
HMg4[$n!9;"bA3_$.I
!dg%pCZ7'qcZOJ'#^#&w
Zz_w?ZzQb];:QbQ:*\/[
^>VRMh1muv*.mu6*.mv1
XzYtb%-Ze^X+Z3RX+bu"
h3@>||3o>|>=>8P=/o>|
R;+/O*5u$Cr7U+_r7T'!
@ZK(dOT:6f@+>(bA;,|?
r4%0<#4OWmr4)U=23&=I
yQ-G(y1Nu7yC*MhyQ$u7
1wXA~%vp\O1!XA~-v0'c
g6z6%P`595gj39Kgj|AY
wlSH/y52y?}52q?062B/
Rx>i:&5IJT.6C%]/bSk]
,[-d]i/:+]!9?3=i/?m]
h|+,gwz],e1#m<Z9Z|UU
5TA/{u~c!G2jc!GlTB_G
(Qd62Q?Lp"TOk)?D62
L#,$$Hc&$$d",#L1B;$$
Ml;9$>l'9<Ml;9$<"@-g
O|+~NFjU+"FjU.3Fz+wB
o)tt+>hC7+>hC4s>hDBg
8+8-u:2W=u:2S=uR5&-m
p$)S$>-$iC2$%k=|-,:I
jM?I4zMBA@@4A]LjM4+:
Zi?*c#[/::zJq7+ZI/zg
N}x8%Fm'31A./8!A>~3:
Jcu?!Ncu?!V3s?]%ry@6
3Z1/M+-8H=3-1Rr361/M
_!_b;<1!b*mpbj*<1"vs
52so%52so%5}$mtD2A:&
+&e@w38^+K38^,<28E@w
JBj4#BD=K6B@/4A6F|[5
9dS{()D1k6R$${(.D`{6
&}!f@6}ao/&}BK.g}#0.
}44=-z|@d-&%?L?,(/L?
T6KQ^L!*9]+{*)]L~k2_
B$sp}y[2<Sy[2[uy[4[U
"LeymuY:]Z+:G}ew!:}e
#&zk*#/",1#/zL)#0nd/
9F"%Y!~o'59F"%Y)B"5U
D*gQlT<Ku=_4[A-R;DRP
>=k.P0+*<ZBd-8;B<i*Z
i^77*Xn6'"cJ&'ycJ$|8
?m!>97-2.97-?02V-bp+
V=2(ov-&a:*>602P`+02
)S]$''3_8~>4_8~PI08~
jnq<S?N!Lg?N70%?Nr0'
OQTA6%6%&}['4A6KCt&}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
-LAqDr;Yi:-LAarzr*cr
v%RnCS>/nC[:-Ka<=Znu
;ju&]$@#Q}+7#Q}d_*'Q
'DT:C;]D<K<lL:C$<H0G
^^x\N^Xr]'^Xr\'PHb\'
C*[av$+[86=8Y9UC+[83
?0=7i@(-7k!\S68?@5?&
p$2/odH?/op$2?o$(!N,
(<=,3hMG+7A{0*.h^3,3
\-0cqNg.9r3'vi&#'.=v
"04&9pv&l}0*4&9!-<2}
t*A|;K"F|&HMR\{GNV|&
$8C.-<#qX7<#qZ34XUX;
V@S5kV@4WOVDm_-+l46K
*A.7w>Rs7I&q.7I6A.7I
h+q8$??v8!3MY6g3MY6g
'_B}!(m:d.<l~y}(8@=.
}<;6D)|s+FuB.7,}<4+R
\|5;>3-}.a7TYlM3,53?
2>0.%$S@e&2>"<Z2C!e&
/#k0$#!;0=!+26>.*k0$
s!:S!#x7S!+)3So6ATXW
u!<5~4!e5~A}ETn6p;5~
!.A<XwIgm%7p<m[wIgm%
#{qvm,=.nikG,~akG-ne
m-weOmVWk*@rPcWmVWE)
9k9d+9k;-R9k=/.:tqjE
P22BYt5Nxy4Ibj'P52BY
Sq!-TSq!,8Sq-5Wxr?7m
s";23}=.j4{=*:EM~>k4
3<E#5X:%Ru3<ER%X;%n(
1x,"|`x/,\)wL"|p+,+r
#@E=6"AA{)8@!;'7A*|>
,Q=&.-m=&.J2"8T+m5(*
SSC7=USE^#SSJ5%u3?5,
/-NYUN,I!(N-O}+X_o}+
JTo6=6YM6\6I'ETJE%;t
=9./a4,//a]7rN0]7~0<
xo}x-{ho97wj/X-tmsyR
!`da8]j[;V]j_ax]j[y~
,>.?Z8_.6R<:,~n8_.'S
MHU;B34e;Bw<\$v34a"j
=%jAv-'6;u-8:A!l&b;v
p06o'y(VOFApT=&ApV9&
(A{8*yA{8*ya|6+YAl5*
>R}?iq/q965Rq9&`ZY^i
\""^zf=o^zr<u^zf-?^z
l(K.&YX:X.myK.&QxK.&
zd>I;$;|'/!8*J;&c}I;
_\&87|X,&,@L"{0<H"[0
l[Hegl[C&fp[P^{`%s&#
5XA@y,'$!y=JC!y3J6{1
-q`$&i2:!<i2F5/daZ!<
Jf:A?jf(d_JF@=_!]_=_
U#/[9kH3t8kH2<0kH25Q
9\<"l!]~"l!^v(,1w#(T
038)R@-8)Rj)8)$:181v
o"o9;=]eq7$|41+$tml+
Tu$F]g\0VGs^/F_+_/>;
D)&/S!96/SFp-F4@G&'*
s\KG&{$KG&)w7Nzr,F(6
>F>5gR^|5gw-/.;R^~%;
iow2952'395OU;jq2%;7
O<IALT<JOJ\\WALV@IFP
>'uKzcY8*z@'u<:cM=@>
0@,"8p8,28p4%28rt)!^
-'?;"$iL~93jN;"wj;tb
4V2;!'P$@y3j2;!3L4&'
)0!]I/p6oo9?#!I'p>!I
`1'k5dl'c4}!;C8m!&Sc
HLGhBWp3J%S0e&%+LE&%
6(K[9&M&,<6MKUB&)"S9
(I#}29i#u>hb#}2()bo=
14q;'Z7RZ!%.*X_^*.Z!
=43#aC43#a>6S4_A:@6G
1J3=EWn3=ClhS}CWn3$8
jh)Qunh9GEjP()+@A()+
9;Lp2A?M[A9G3PQ9G3u6
0*8'=P{3()P{4&>0+4"m
S!<#Gu"&=8Kl<#GLjA>K
tphH+F:sTKtPmD.F:3D.
HV{#bLip"=Hik#bHil\G
nD<jAG*^~Ag*Kj3t<CjM
?PsKzLM9&9jMlky?OvN>
8zMw't0-w'Hz06~@:sU'
0A+>!1)n-0o`_.e1)k>!
9$@T_&0"AA'.*?`'.-ad
A9u10/)*9=c?xd9c?u10
*%.Y\@7rBK<7rBK863B4
V\'[;"D$_zA/-_z7X,L^
@eC64vC]US@$`{8>$C64
/v"?{1Tk?&:jI?s4,,?&
"v9-NPL[/2xr%"2Hz&"2
;~MA3d:#s3"+3A3E}MA3
oZ%jPM4D*4M\B$LM\I#l
`gW(=0g8Vr|.s.L\sw'2
0~S1}/3V({0~V({0%Q0~
6fC_oJvQ"}5;AY{JeC_o
.>j*!-/F:We2F:WHlj*!
*M.J)>mSJ):O/'?=n5j/
$<I?J$ty?JD<PA#D<NP!
0g=*K8%Y#B85$4u85#27
"W,=eZ"B}ml[t=mj,?}m
TVJ3C5v;,yQwj3CTVJ5i
="=?=T=<i5T=l:S,=-?=
7g>BbwWAB?7g>Bb</]r6
|W+,kV6G)k)?0(iR7W)k
u3i|#$MYlYuN}-HsMYlY
-|QFU<LcC;<Lc&&pP#BE
d$+sQp().*p('%-t(9.m
.J"#0X4]$qlI!#0RI"$,
(,\!Ix%xE0|,|"Ku,xM.
nplK'09+GV,YnC#,Jnk'
(wG)6$uG9WtUVi"0cgY?
{,5Tv,-|eYK.KAJK.LVG
nCV~.J=^SzfA^CzfAT~6
:)^C56{'S=?9~C5'o~Hn
<U)<->;FD>>;E%=#xE#]
<#]&he7)R;$6(#G%6(R;
:6B4ZZ&3H>:6AdB:5O|!
[U_['%Y>|%[GV!6[GV!n
^(/(-"+2(-'o4!!'o2&)
@AO{sH_0{sXoo{shgMs>
t5u=?t5w=?t5w,>A@?;;
|f!H&pSaH&l3iH#lv!Xf
r'X=c=/B=a|0Q6F=/J5*
,`,{qu+/Ka!B*_2u+-0&
&T0x,?51x,/t1(\Zt3j0
/"{xw8'Tl!*1-ywlUSyw
/;}>F0<w<.([,/M&70?N
,7@X!+)_X!+WVD!)M8Xk
`b83,ySW3@q9x9+hB@3O
c}/Z!^0..0@-.Z!VX**0
y/!94eN+)39..0}9..py
25qE;291L+gT"o;e,0o4
&#:#>#%#[^&#:#>6$?$^
f"z&;ZI){<@|3(fXM6&;
s,k'<3cy*WI<y/:M-y*W
a6<|5Q55,4a5;|4h,;|4
1A*O-)A38$/A38$MA9@%
/@NOU"Bd*=.92)5.9<*J
){}"[A!5$"&6Ma%'&E!`
(U>Jy0^_+5$YaK{,^)[J
%YKn^%Y}1s%Y}&B%YC[2
w79":./8S"k/7S",07S"
(do,?z<i<;(Do,?)L!<+
8R55(@X>5G8p8;/8p55(
"j08-#*A%4#9A%4#*Q7t
%P!@{.pF*+.pFj<.0`V&
t@4U*tao_#%"O?K%!FRK
AVa?$c%-@m3/*>$rl">0
Y;1{v{P%k?{Rn{tjD&=+
<b?[:,9G[D4b=!:4cc!:
W[pvT8[`1)5!@*T8[e,C
!N/_t(L!:Yy(|:]y(!y}
4>)[M|6W&Wy6`Y]9K"&S
00lI5(|l*=p$AW68lt*=
k!99T8<=9T8<=9,B-7-D
,C238,}0"",C2"@;Fl"v
;NKZ!Z~x>(\#KZ!J~mR%
t];;!j7:K*jU:5uj79|.
"D8lC}'+':B,K$CCIk$C
g.&x!OYoJ/<90v!OYjus
;@"Eadf5EA#d"x{#d"Ee
A)q9'A)q9'01)9x01(|S
9@~)(p#Zzipb^)(|p~)(
g30&WtW&oCsg9oU:'9oU
,`!Pw>G:[.6E2[*>G<[.
}_"XUug:Uy#5gkQug9kQ
?N_P&h,8;"'.i;"'.nNu
ZO/E,RG5q*^/J"<^/D&,
50!6c#tuNA^tuNA6T]:M
%lwRSet(?bet&Ywet(Ts
(/w)26$A^$=-E!6=0M)2
[k<8oSH6L.S)>9j[k2m/
QA&!3'"%_lB#f!3q"&$5
W.s}X9BQ}(wP!~(1RU}(
X!aZ}0AMK3+-Q[3+-QZy
Qbg:{En%(sEn'9[pX+(3
}/'O\e;%G]a3+/7a3(o3
'E<>@i<A-0i<<>q;EQ>0
L%|"5J%d"596L"5Z5d!6
eD?U^mN_U^-2Y8X=6F,h
9\)UG5`;WC"h<WC25+TG
149N.5`RFE7r2FE7"2)v
>0eUH1/W<J@/s$O1/RBG
0DM85e@3R%e@32<g<7.%
r`3k!vtM|{#)3k!!T&l{
2`/;3$#G2x8iQ;35)/;3
A0/s5*(As5I'As5*(F)R
"{:>ex>g6&Sfg7sx0g6&
VCxn<WGZ={VCxIgVCxIc
;@?71mm77!m=;&03px8.
k3tq^k<0Q^m<*2pl.b<*
E=#+3E='+3.}5R^0}(c!
4]-o6a@nUI>:/e6j]de6
'Y/5~V9;9v7)?5~%y)X*
t3Iz0L%Uz0tK+s_D.5s_
"B[(5xA;(5xgk(5"C?Gv
>=;*x>m;:%}j"Y_}:{<5
"1%g"=\Ld&$L-d(y!Dg"
T!}G23c3G2;s3Gi<3)G"
<*&":~+5{{ZKK":YCPaJ
@I:%+#3K&"#;i%+>[K&W
`!qm8i(lm8W)#elwi*e,
-:?@L%]N}"<;?="<c0="
9'_-%-Z!-%0Z0E70Z.-r
kQ=d_7^=b!jQ5c/7[=d[
/&IlbNoQ'c/&In9/&In9
%b#*j0&z37"l#Fm!d##j
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

99I')8>@m.h9)((?)0o& uA1W&{tlL({th_&8()_& ;=Q&L[SWAH]!7{L[s!a(
)i0N>%6RM}l'RM}#e^My *=_i'9B>@i*=_i]*=`@S <!*S6|2WH8|2WG1DkWH8
@n;]Z@(gM<_),5|Oj?]| 6//"/$)='/2K?"/4A;'/ KRy>)I?s=-519>)I?xm}
(f*]#j;=Y7zS:/~zS=]? qA{G%uAzW"$`9<-p@{GX 2%)1&"1@1%<0@0M&281&

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BIBLIOGRAPHY

Agnew, C.B. "Random Sources for Cryptographic Systems." Advances in Cryptology: EUROCRYPT 1987 Proceedings. Berlin: Springer-Verlag, 1987(pp 77-81).

Beutelspacher, A. Cryptology. Washington, D.C.: The Mathematical Association of America; 1994.

Biggs N. L. Discrete Mathematics. Oxford: Oxford Science Publication; 1990.

Biham E. and Shamir, A.Differential Cryptanalysis of the Data Encryption Standard. New York: Springer-Verlag; 1993.

Blum, M. and GoldWasser, S. "An Efficient Public Key Encryption Scheme Which Hides All Partial Information." Advances in Cryptology: Proceedings of CRYPTO 84, Berlin Springer Verlag, 1985 (pp 289-299).

Brickell, E.F. and McCurley, K.S. "An Interactive Identification Scheme Based on Discrete Logarithms and Factoring." Advances in Cryptology: EUROCRYPT 90 Proceedings. Berlin: Springer Verlag; 1991, (pp 63-71).

Burrows, M., Abadi, M. and Needham, R. A Logic Authentication, Nashua, NH: Digital Equipment Corporation Systems Research Center; Feb. 1990.

Cooper, F.J., Goggans, C., Halvey, J.K., et al. Implementing Internet Security, Indianapolis: New Riders Publishing, 1995.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Diffie W. and Hellman M. E., "Privacy and Authentication: An Introduction to Cryptography." Proceedings of the IEEE, v, 67, n. 3, March 1979, pp 397-427.

Dolev, D., Yao, A.C. "On the Security of Public Key Protocols." 22nd Annual Symposium Foundation of C.S.: IEEE Oct 1987..

Durstenfeld, R. "Algorithm 235: Random Permutation," Communication of the ACM, vol 7, num 7, Jul 1964, (p. 420).

Feistel H., Notz ,W. A. and Smith, J. L., "Some Cryptographic Techniques for Machine to Machine Data Communications," Proceedings of the IEEE, v 63, n. 11, Nov 1975, (pp. 1545-1554).

Fumy, W, Pfau, A. (1990). "Assymetyric Authentication schemes for Smart Cards-- Dream or Reality?" Proceedings IFIP SEC '90, Espoo, Finland.

Gardner, M. (1977) "A New Kind of Cypher That Would Take a Million Years To Break," Scientific American, v. 237, n.8, Aug, (pp. 120-124).

Garey, M.R. and Jhonson, D. S. Computers and intractability: A guide to the Theory of NP-Completeness. San Francisco: W.H. Freeman and Co.; 1979.

Garfinkel, S. and Spafford, G., Practical UNIX Security. Sebastopol, CA: O'Reilly and Associates, Inc.; 1994.

Knuth, D.E. The Art of Computer Programming. Reading, MA: Addison-Wesley; 1973.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Kohl, J. T., "The Use of Encryption in Kerberos for Network Authentication," Advances in Cryptology: CRYPTO '89 Proceedings. Berlin: Springer-Verlag; 1990 (pp35-43).

Konheim A.G., Mack, R. K., Mac Neil, B., Tuckerman, N., and Waldbaum, G., "The IPS Cryptographic Programs," IBM System Journal, v 19, n. 2, 1980 (pp 253-283).

L'Ecuyer, P. "Efficient and Portable Combined Random Number Generator," Communications of the ACM; vol 31, # 6, 1988.

Levin L. A. "One-Way Functions and Pseudo Random Generators." Proceedings of the 17th ACM Symposium on the Theory of Computing: 1985 (pp 363-365).

Lewis, R. L., Denemberg, L. Data Structures & Their Algorithms. New York: Harper Collins; 1991.

Luby, M. and C. Rackoff, "How to Construct Pseudo-Random Permutations from Pseudorandom Functions," SIAM Journal on Computing, Apr 1988 (pp. 373-386).

Maor, M. and M. Yung, "Universal One-Way Hash Functions and Their Cryptographic Application," Proceedings of the 21st Annual ACM Symposium on the Theory of Computing, 1989 (pp. 203-210).

Matyas, S.M., C.H. Jeyer and J. Oseas, "Generating Strong One-Way Functions with Cryptographic Algorithms," IBM Technical Disclosure Bulletin, v. 27, n. 10A, Mar 1985, (pp 5658-5659).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Menezes, A. J., van Oorshot, P.C. and Vanstone, S. A. (1996). Handbook of Applied Cryptography. Boca Raton: CRC Press; 1996.

Merkle, R. C. "One Way Hash Functions and DES," Advances in Cryptology. CRYPTO '89 Proceedings, Berlin: Springer-Verlag; 1990 (pp2-13).

Needham, R. M. and Shroeder, M. D. "Authentication Revisited," Operating Systen Review, 1987, v. 21, n.1, (p. 7).

Okamoto, T. and Ohta, K. "Disposable Zero-Knowledge Authentication and Their Applications to Untraceable Electronic Cash." Advances in Cryptology, CRYPTO '91 Proceedings. Berlin: Springer-Verlag; 1990 (pp 134-149).

Otway, D. and Rees, O. "Efficient and Timely Mutual Authentication", Operating Systems Review. 1987, v. 21, n.1, (pp. 8-10).

Park, S.K. and K.W. Miller, "Random Number Generators: Good Ones are Hard to Find," Communications of the ACM, v. 31, n. 10, Oct 1988 (pp 1192-1201).

Pohlig C., and Hellman, M. E., "An Improved Algorithm for Computing Logarithms in GF(p) and its Cryptographic Significance," IEEE Transactions on Information Theory, v, 24, n, 1 Jan 1978 (pp.106-111).

Preneel B., Analysis and Design of Cryptographic Hash Functions, Ph.D. dissertation, Katholieke Universiteit, Leuven, January 1993.

Rivest, R., Shamir, A. and Adleman "A Method for Obtaining Digital Signature and Public Key Cryptosystems," Communication of the ACM,1978 v.21, n.2,pp. (120-126).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

________. "On Digital Signature and Public-Key Cryptosystems", MIT Laboratory of Computer Science, Technical Report, MIT/LCS/TR-212, Jan. 1979.

Roberts F. S. Applied Combinatorics. Englewood Cliffs, NJ: Prentice-Hall; 1984.

Santha, M. and U.V. Vizirani, "Generating Quasi-Random Sequences from Slightly Random Sources," Journal of Computer and System Sciences, 1986, v.33, (pp 75-87).

Shneier, B."One-Way Hash Function," Dr. Dobbs Journal, v. 16, n. 9, Sep 1991 (pp. 148-151).

Schnorr, C.P. "On the Construction of Random Number Generators and Random Functions." Advances in Cryptology-EUROCRYPT 88 Proceedings. Berlin: Springer-Verlag, 1988, (pp 225-232).

Sinkov T., Elementary Cryptanalysis. Washington, D.C.: Mathematical Association of America; 1966.

Tardo, J. and Alagappan, K. "SPX, Global Authentication Using Public-Key Certificates", Proceedings of the 1991 Symposium on Security and Privacy of the IEEE, Apr. 1991 (pp. 232-244).

Tsudik, G. "Message Authentication with One Way Functions," INFOCOM 1992. Florence; May 1992.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Vazirani, U.V. and V.V. Vazirani, "Efficient and Secure Pseudo-Random Number Generation," Proceedings of the 25th IEEE Symposium on the Foundations of Computer Science, 1984 (pp. 458-463).

Welsh, D. Codes and Cryptography. Oxford: Oxford Science Publications; 1993.

Wichmann, B.A. and I.D. Hill, "An Efficient and Portable Pseudo-Random Number Generator," Applied Statistics, v. 31, 1982, (pp. 188-190).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

INDEX:

alphabet, 22
analysis
  statistical, 53
authentication, 82

bad password, 1
badly chosen passwords, 83
big O, 20
bigram, 54
bijection, 21
brute force, 9, 12

Ciphertext:, 12
computer
  fastest, 10
  super, 10
computer cracker's, 5
coprime, 84
cracker, 1
Cryptanalysis, 11
Cryptography, 11
Cryptology, 12

Decryption, 12
Diffie-Hellman, 84

Encryption, 12

gigaFLOPS, 5

homophonic, 15, 16, 17
  code, 15
  permuting, 18
  word, 18

IDEA, 116

Joe account, 4
Julius Caesar, 11

Kerberos, 117
Key crunching, 113
key string, 21

Lehmercode, 46
lexicographic order, 46
logarithmic, 20

MD5, 115
MIPS, 1

Oxford English Dictionary, 5

passphrase, 115
password
  bad, 3
  good, 9, 10
password space, 12
Pentium, 7
permutation, 19
  according to a key, 21
Plaintext, 12
polynomial, 20
private key, 119
probabilistic approach, 77
public key encryption, 88
public key exchange, 83

RSA, 115

secret keys, 86
statistical
  tests, 92
statistical analysis, 19
strong cryptosystem, 12
surjection, 21
symmetric group, 46

TGT, 118
totient function, 84
transposition, 19
transposition cyphers, 15
trusted party, 114

User, 82

Verifier, 82, 89, 90, 91, 112

Wide-Mouth Frog, 113

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.