ABSTRACT

Application Level Cryptography

Combinational Stream & Block Ciphering Using Double Encryption Algorithms

Ashish Anand
Computer Science Engineering
Maharishi Dayanand University, India
ashishanand25@gmail.com

Scenario:

- Mails sent from a browser (using Yahoo! etc.) are broken down into TCP packets that contain the body as plain text.
- Instant messages sent using clients (MSN, Yahoo! messenger etc.) are also sent as plain text.

Vulnerability:

- TCP packets can be intercepted by monitoring the originating interface or by having access to any of the routers that they pass through. **Privacy** loses its charm when even your ISP can easily monitor all your data.
- Sad but true, inspite of using *decent* hardware (3Com HiperARC Dial-In PPP RASs') ISP's like MTNL have proved to show oblivion towards elementary security measures.

My work:

- I successfully managed to access most of their routers and *monitor each interface*, thus enabling me to view mails & chat sessions. The best part is that they can't sue me for this!
- In the process of coding a Linux based server in C++ using secure TCP/IP sockets
- Developing a TSR application with Windows-DOS inter-portability to make available cipher-text in Windows Clipboard
- Key generation by measuring keyboard latency & tracking mouse movements. Technique is immune to *physical attacks*. Statistical study verifies randomness of technique used. Working on making key generation even more secure.
- "Private key" transfer using "public key"
- New "Session Key" after random time intervals, generated from the IM
- Modified IDEA & DESX standards to develop a simpler, faster, yet secure ciphering technique using *whitening, transposition, block & stream ciphering, and compression algorithms.*
- Resistance to brute force attacks.

Real time working of all concepts would be demonstrated…

(250 Words)

Application Level Cryptography

Combinational Stream & Block Ciphering Using Double Encryption Algorithms

Ashish Anand
Maharishi Dayanand University, India
E-mail: ashish.anand@titsbhiwani.org

"If I take a letter, lock it in a safe, hide the safe somewhere in New York, then tell you to read the letter, that's not security. That's obscurity. On the other hand, if I take a letter and lock it in a safe, and then give you the safe along with the design specifications of the safe and a hundred identical safes with their combinations so that you and the world's best safecrackers can study the locking mechanism – and you still can't open the safe and read the letter – that's security!" **[1]**

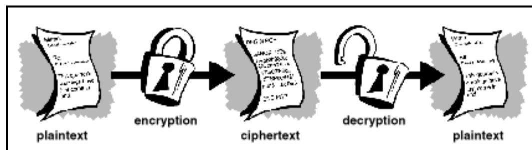

Fig. 1. Basic Cryptography [7]

In addition to providing confidentiality, cryptography is often asked to do other jobs **[1]**:

Authentication: It should be possible for the receiver of a message to ascertain its origin; an intruder should not be able to masquerade as someone else.

Integrity: It should be possible for the receiver of a message to verify that it has not been modified in transit; in intruder should not be able to substitute a false message for a legitimate one.

Nonrepudiation: A sender should not be able to falsely deny later that he sent a message.

This paper is inclined primarily towards the *integrity* aspect of a successful cryptosystem.

Algorithms & Keys

A cryptographic algorithm, also called a **cipher**, is the mathematical function used for encryption and decryption. If the security of an algorithm is based on keeping the way that algorithm works a secret, it is a **restricted algorithm.** Restricted algorithms are woefully inadequate by today's standards. A large or changing group of users cannot use them, because every time a user leaves the group, everyone else must switch to a different algorithm. If someone accidentally reveals the secret, everyone must change their algorithm.

Modern cryptography solves this problem with a **key**. All of the security in key based algorithms is based in the key (or keys); none is based in the details of the algorithm. This means that the algorithm can be **published and analyzed.** Products using the algorithm can be mass produced. It doesn't matter if an eavesdropper knows your algorithm; if he doesn't know your particular key, he can't read your messages.

"If privacy is outlawed, only outlaws will have privacy..."

Choosing an algorithm

When it comes to evaluating and choosing algorithms, people have several alternatives:

1. They can chose a published algorithm, based on the belief that a published algorithm has been scrutinized by many cryptographers; if no one has broken the algorithm yet, then it must be pretty good

2. They can trust a manufacturer, based on the belief that a well-known manufacturer has a reputation to uphold and is unlikely to risk that reputation by selling equipment or programs with inferior algorithms.

3. They can trust a private consultant, based on the belief that an impartial consultant is best equipped to make a reliable evaluation of different algorithms.

4. They can trust the government, based on the belief that the government is trustworthy and wouldn't steer its citizens wrong.

5. They can write their own algorithms, based on the belief that their cryptographic ability is second-to-none and that they should trust nobody but themselves.

How I decided to design my own algorithm considering the above mentioned:

1. The DES (Digital Encryption Standard) and IDEA (International Data Encryption Standard) are the most popular patented algorithms around today. I chose a much simpler and thus faster implementation of a combination of what these algorithms do, including *whitening, transposition, block and stream ciphering* resulting in the evolution of an inherited version of the DES variations (DESX) and IDEA used in PGP. Thus besides originality, elements of something widely published make the project qualitative.

2. Trusting a manufacturer would mean purchasing a hardware implementation. Practically speaking, o*ne man alone (me)*; cannot design an implementation of this magnitude, so the second option was ruled out.

3. Hiring a private consultant seems feasible, but cryptography is not my profession, I haven't dedicated my life to it either. You're probably insane if you're expecting someone to come up with a radical new idea all by himself and I'd rather not say why!

4. The Indian Government is hardly involved in such projects on a scale comparable to the US or the Europeans. Besides, this project is not intended to be implemented on a scale so large that it would require the intervention of a nations Government.

5. Writing my own algorithm…sounds fun, though the decision is certainly not based on the belief that my ability is second-to-none and that I don't trust anyone but myself!

"Don't worry that you're reinventing the wheel all over again, that's what learning is all about…!"

Compression & Encryption

Using a data compression algorithm together with an encryption algorithm makes sense for two reasons:

1. Cryptanalysis relies on exploiting redundancies in the plaintext; compressing a file before encryption reduces these redundancies

2. Encryption is time-consuming; compressing a file before encryption speeds up the entire process

The important thing to remember is to compress before encryption. If the encryption algorithm is any good, the cipher-text will not be compressible; it will look like random data. (This makes a reasonable test of an encryption algorithm; if the cipher-text can be compressed, then the algorithm probably isn't very good).

Table 1. Observations upon encrypting a sample text file without any compression using the program I developed

Size of Plain Text:	16,947	43,843
Size of Zipped Plain Text:	5,045	14,485
*__Percentage Compression of Plain Text:__	**71%**	**67%**
Size of Cipher Text:	29,923	80,143
Size of Zipped Cipher Text:	11,833	29,000
*__Percentage Compression of Cipher Text:__	**61%**	**64%**
$**Ratio of Cipher Text & Plain Text:**	**1.76**	**1.82**
^**Ratio of Cipher Text & Plain Text:**	**2.34**	**2.00**

$ *Before Compression*
^ *After Compression*
* *Compression using WinZip 8.1*

The catch here is that the input text consisted of the source code of my program, which has about 1000 line breaks. Combine that with multiple repetitions of keywords like *cout, printf, get, read, write* and so on, and you have more redundancies than one could imagine. More the redundancies, greater is the compression.

In a way, this means that there are no bounds to the range of the input character set (by no bounds, I mean that it ranges across the *entire* ASCII character set; codes 0-255). Now that's **at least** about **200** different symbols for any conventional input stream. But after the second round of my block ciphering algorithm, I'm limiting the output cipher text to an ASCII character set ranging from codes 33 – 132, that's just about **100** different symbols. This leads to one observation and one question:

There would be **a lot of** character repetitions in the resulting cipher text. Imagine representing 16,947 (~17,000) characters consisting of 200 different symbols using a character set of just 100 symbols. No wonder the size of the cipher text is almost twice as that of the plain text!

Which one of the following ideal cases would lead to greater compression?

> Compressing an input plain text of 200 characters in which each symbol is unique (because it uses the 0-255 range, with some exceptions)
>
> Compressing an input plain text of 200 characters in which at least 100 symbols are unique (33-132) and each symbol occurs exactly twice

If you couldn't already guess, it's the latter. Better still, a practical example would verify that symbol occurrences would actually be more than twice, thus leading to even great amounts of compression.

What conventional block ciphers don't do is limit their output to a smaller character set. And that's exactly what I've done. Thus, I'm purposely compressing after encryption, rather than the other way round. One may argue that if the entire algorithm is intended to be made public, then compressing the data wouldn't add to security as an eavesdropper could uncompress it to obtain the cipher text with all the redundancies. Now that would be undesirable.

To overcome this issue, I decided to treat the compressed data with a transposition function that would take the private key as an argument. Thus without the key, it would be impossible to uncompress the cipher-text.

IDEA - Block Cipher Algorithm

The first incarnation of the IDEA cipher, by Xuejia Lai and Kames Massey, surfaced in 1990. It was called PES (Proposed Encryption Standard). After subsequent strengthening of the algorithm, it was renamed to IDEA (International Data Encryption Algorithm) in 1992. IDEA is patented in Europe and the United States; the patent is held by Ascom-Tech AG.

IDEA's key length is 128 bits. Assuming that a brute force attack is the most efficient, it would require 2^{128} (10^{38}) encryptions to recover the key. Design a chip that can test a **billion** keys per second and throw a **billion** of them at the problem, and it will still take 10^{13} years – that's **longer than the age of the universe.** An array of 10^{24} such chips can find the key in a day, but there aren't enough silicon atoms in the universe to build such a machine.

This remarkable observation about this algorithm prompted me to devise a software implementation of something similar, but something that would work much faster, since IDEA uses 8 rounds of XOR and modular addition and multiplication operations along with 52 sub keys derived from the 128 bit key. I cut it down to just one round of XOR operations, while using just 2 sub keys derived from a single 10 bit key. Though increasing the key length to about 32 or 64 bits wouldn't hurt, I shall stick to just one round of XOR operations. (Inspite of this simplification, encryption seems to remain a slow process for comparatively large amounts of data, as well as theoretically much less secure. But my implementation is obviously not intended to aim at achieving a Patent or anything, and the outputs produced from my program will appear secure enough to be implemented on a personal basis as well as within internal networks of an organization).

As already stated, one may keep adding rounds and rounds of complexity to your algorithm, but the power of your cryptosystem depends more on how you generate the key.

Some concepts involved

Transposition Ciphers [Discussed later]

Whitening

Whitening is the name given to the technique of XORing some key material with the input to a block algorithm, and XORing some other key material with the output. This was first done in the DESX variant developed by RSA Data Security, Inc.

$C = K_3 \otimes E_{K2}(M \otimes K_1)$
$M = K_1 \otimes D_{K2}(C \otimes K_3)$

Measuring Keyboard Latency

People's typing patterns are both random and nonrandom. They are nonrandom enough that thy can be used as a means of identification, but they are random enough that they can be used to generate random bits. Here's what I did:

Measure the time between successive keystrokes, then take the least significant bits of those measurements. These bits are going to be pretty random. This technique may not work on a UNIX system, since the keystrokes pass through filters and other mechanisms before they get to the program, but it will work just fine in a DOS/Windows environment.

Next, find the sum of these random numbers. Divide it by another random number between 1 – 11 generated by the C++ internal randomize() function. If this number is greater than 5, then divide the sum by this number, else multiply the sum with it.

Calculate the 10-bit binary of the result and use it as the private key.

Other sources of obtaining random data **[1]**:

- The sector number, time of day, and seek latency for every disk operation
- Actual mouse position
- Number of current scan line of monitor
- Contents of the actually displayed image
- Sizes of FATs, kernel tables, and so on
- CPU Load
- Arrival times of network packets
- Input from a microphone (noise)

"Random Number Generators in your Compiler are NOT a source of randomness…!"

Statistics

During the program testing phase, the random key generator's results at each run were recorded, based on the typing patterns of a number of different users. 5 different users were made to generate 20 keys each, and the total number of keys (100), were entered into a database for further analysis.

Each user's data set was filtered out to obtain the number of repetitions in the 20 keys generated.
The efficiency of my technique was obtained by dividing the number of unique keys generated with the total number of keys generated.
Individual efficiencies for each user's data were calculated, along with the average efficiency of the entire key generation algorithm. Some users were made to type different strings each time, while others were asked to monotonously repeat the same string each time.

Table 2. Key Generation Statistics

	USER 1	USER 2	USER 3	USER 4	USER 5
KEY 01:	0000011100	0000101111	0010110110	0110011000	0000110111
KEY 02:	0000111101	1101001010	0000111100	0001011001	0011000000
KEY 03:	0010111011	0000011111	0000101101	1100011100	0000111100
KEY 04:	0101101101	1110011000	0000110001	0100101101	1111010110
KEY 05:	0000110010	1100010100	1110111110	0000111111	**0000111110**
KEY 06:	0000101100	**0001000000**	1101101010	0000101001	**0000111100**
KEY 07:	1011010010	**0001000000**	0000101010	1110100100	**0000101001**
KEY 08:	0001010110	0101011010	**0001000110**	0001000101	0101011000
KEY 09:	1111111100	0001001111	0001001111	0110110000	0100110110
KEY 10:	0000111011	0000111000	0001011001	**0000111001**	0000100100
KEY 11:	0001111000	0111101100	0001000001	0000101101	0000100110
KEY 12:	0000101001	**0000100000**	0011110100	0110110101	0000110101
KEY 13:	0111011010	0001011010	0000110110	1110100000	0001001110
KEY 14:	0001101010	**0000100000**	1101101100	**0000111001**	0000111111
KEY 15:	0000110001	1001011111	1101001100	0000111100	0000101111
KEY 16:	**0000111010**	0001001011	0001001011	0100111000	0110010100
KEY 17:	**0000111010**	0001010011	**0001000110**	0000111011	0011110100
KEY 18:	1110011110	0111001110	0000111011	0000011010	0011001100
KEY 19:	0000100111	0000101000	0000111000	0000111010	0001001101
KEY 20:	0001000111	0100001100	1101100110	0101100101	**0000101001**
INPUT STRING	*Same*	*Same*	*Different*	*Different*	*Different*
REPEATS	*1 Repeat*	*2 Repeats*	*1 Repeat*	*1 Repeat*	*2 Repeats*
EFFICIENCY	*95%*	*90%*	*95%*	*95%*	*90%*

If all 100 items are filtered, 23 repetitions are found. This implies an average efficiency of 77%

PGP – Pretty Good Privacy

"If all the personal computers in the world—260 million—were put to work on a single PGP-encrypted message, it would still take an estimated 12 million times the age of the universe, on average, to break a single message."

PGP is an e-mail security program written by Phil Zimmermann, based on the IDEA algorithm for encryption of plaintext and uses the RSA Public Key algorithm for encryption of the private key.

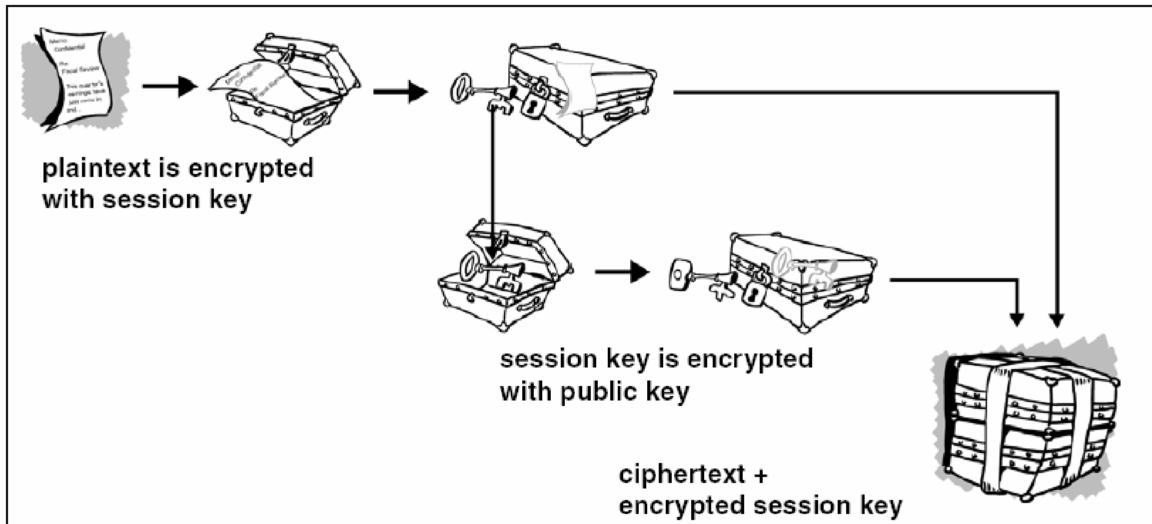

Fig. 2. The *details* of how PGP works, is out of the scope of this document **[6]**

PGP uses a pass phrase to encrypt your private key on your machine. You use the pass phrase to decrypt and use your private key. A pass phrase should be hard for you to forget and difficult for others to guess. It should be something already firmly embedded in your long-term memory, rather than something you make up from scratch. Why? Because if you forget your pass phrase, the game's over! Your private key is totally and absolutely useless without your pass phrase and nothing can be done about it. PGP is cryptography that will keep major governments out of your files. It will certainly keep you out of your files, too!

Sounds pretty neat! Top that up with the fact that it's all Open Source, which has it's own plethora of advantages which we don't want to get into right now.

PGP Vulnerabilities

Fake PGP: Since it's all open source, there are fake versions of the famous software floating about the net. Unless you're sure that your copy of the program is from a trusted source, it wouldn't be surprising to realize one day that your pass phrase was sent to an attacker via e-mail the moment you went online! Once he has your pass phrase, he has your private key. Key transfer using public key cryptography is useless after this point.

Tempest attacks: Another kind of attack that has been used by well-equipped opponents involves the remote detection of the electromagnetic signals from your computer. This **expensive** and somewhat labor-intensive attack is probably still cheaper than direct cryptanalytic attacks. An appropriately instrumented van can park near your office and remotely pick up all of your keystrokes and messages displayed on your computer video screen. This would compromise all of your passwords, messages and so on. This attack can be thwarted by properly shielding all of your computer equipment and network cabling so that it does not emit these signals. This shielding technology, known as 'tempest' is used by some government agencies and defense contractors. There are hardware vendors who supply tempest shielding commercially.

Keyloggers: Consider a keystroke recorder logging your pass phrase and emailing it to an eavesdropper. What's the purpose of complex algorithms and 128 bit public keys when your own private key is in someone else's hands?

My program overcomes this issue with the mere fact that the only input the user will provide to the system are a bunch of random keystrokes for key generation. Any sensitive pass phrase or the private key is never "typed" by the user. There's no question of encrypting the key and storing it locally. I'm essentially using a different key for each transaction. Since it's a one time session key, it gets erased from the memory the moment the encryption operation is completed. (Whether the key is physically; actually; erased from the hard disk or not – is another issue, which may be exploited by data recovery packages; but then, this issue exists in PGP as well, and besides, clearing out swap files or

overwriting on virtual memory…is **not** what my project is about!).

Given below is an outline of the algorithm for the system I designed:

Step 1: *Processing of Plain Text*

Read input file "PLAIN.TXT" sequentially (bit by bit) and generate it's binary output. Since the ASCII Character Set consists of 256 characters (0-255), it follows that 8-bits are required to represent each character in its binary form. (2^8 = 256)

Step 2: *Addition of One Time Pads*

Pad each binary number with subsequent zero's towards the Most Significant Bit (MSB) until the total number of the binary bits equals ten. This is required as the binary data is being encrypted using a 10-bit private key. Output is saved in file "BINARY.TXT".

Step 3: *Key Generation By Measuring Keyboard Latency*

Take random input data from the keyboard. The time difference between each keystroke serves as an excellent source of randomness. *The beauty of encryption lies not in the complexity of its algorithm, but how the key is generated.* My key generation technique is based on a function that sums the millisecond field's value of the time difference between each keystroke and either divides or multiplies the sum with another random number generated based on its magnitude and converts the final output into a 10-bit binary that is used as the private key. This key is saved in another file "PVTKEY.TXT" in binary form for future reference.

Step 4: *Encrypting Plain Text (Phase I: Stream Ciphering)*

Each 10-bit binary block is sequentially read from "BINARY.TXT" and XORed with the 10-bit key as read from "PVTKEY.TXT". The resulting 10-bit data is stored in "XOR.TXT". This process is often referred to as *whitening*.

Step 5: *Addition of Salt*

We shall be using 16-bit long blocks of data during the second round of encryption. Hence the number of bits in "XOR.TXT" must essentially be an integral multiple of 16. For this purpose, n-bits of 0's are padded at the end of "XOR.TXT" where 'n' is calculated by the following formula:

n = 16-[{(count-1)*10}%16]

where,
(count-1): represents the number of characters in plain text
***** **:** represents ordinary multiplication
% **:** represents remainder after ordinary division

To make matters more complex, one may add salt values other than a continuous stream of zeros. Having another function to decide what salt values to use, based on the value of 'n' generated from the above mentioned formula wouldn't hurt, although this change would require alterations in the proceeding steps. I would probably incorporate this, in the next version of my program.

Step 6: *Generation of Sub-Keys for Block Ciphering*

Two sub-keys K1 and K2 are generated using the private key. The first 2 bits from the MSB of the Private Key are discarded. The remaining 8 bits are complemented and saved as K1. The contents of K1 are reversed to obtain K2. Agreed this means of generating sub-keys is lame. But it's no big a deal complicating this function, and would be taken care of, in the next version.

Step 7: *Division of Stream Cipher into blocks for double encryption*

The number of bits in the file "XOR.TXT" are now divisible by 16. Blocks of 16-bits are processed sequentially. Each 16-bit block is divided into two equal halves X1 and X2.

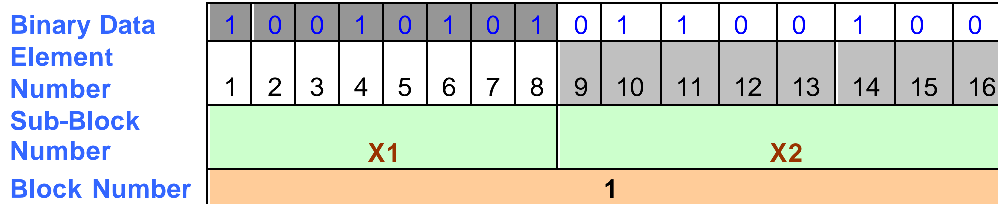

Fig. 3. Diagram showing 1 block of length 16-bits being sub-divided into X1 & X2, each 8-bit long

Step 8: *Encrypting Stream Cipher (Phase II: Block Ciphering)*

The following operations are performed using X1, X2, K1 and K2 and the result is stored in "CIPHER2.TXT"

$Y1 = X1 \otimes K1$
$Y2 = X2 \otimes K2$

where,
$\otimes$ **:** represents XOR operation

Y1 and Y2 are concatenated to obtain a 16-bit binary block. This process is applied on the entire stream cipher.

Step 9: *Processing of Block Cipher - I*

The length of the binary file "CIPHER2.TXT" is essentially a multiple of 16. Thus it is implied that it is also a multiple of 8. Now we sequentially read 8-bits at a time and convert each 8-bit number into its decimal equivalent. The value of the decimal equivalent will always range between 0-256. Each corresponding decimal equivalent is padded to 3 bits of length (for instance, 3 is converted to 003, 52 is converted to 052, 145 is left as it is) and written into the file "CIPHER3.TXT".

Step 10: *Some more salt please!*

The number of characters obtained from block ciphering the file "CIPHER3.TXT" needs to be a multiple of 2 for the next round of block ciphering. Since each character obtained from the previous file is being represented by 3 numbers in "CIPHER3.TXT", the length of "CIPHER3.TXT" must essentially be a multiple of 3. I'm making it a multiple of 2 by padding the end of the file with a zero.

(I realized later that if suppose there are 2 characters obtained from processing the file "CIPHER2.TXT"; that would hold true if it consisted of just 16 bits, then the number of characters in "CIPHER3.TXT" would be 6, which is already a multiple of 2. Thus, subsequent addition of a zero would ruin things up a little, by printing a garbage character at the end of the decrypted text. This minor bug shall be removed in the next version of the program)

Step 11: *Processing of Block Cipher – II*

The file "CIPHER3.TXT" is now sequentially read, but this time 2 bits at a time. Thus at each read operation, we essentially obtain an integer between 00 and 99. 33 is added to each number and the ASCII character corresponding resulting number is written into the file "CIPHER.TXT". The basic issue was to avoid low range codes as they contain non-printable characters, escape key codes, line breaks, spaces etc.

The cipher-text would then be compressed.

The next version of the program would incorporate a final *transposition ciphering function* at this stage. I'm working on a transposition function that would play around with the values just obtained and side by side make sure that the transposed resultant wouldn't violate the *'avoiding a low ASCII code range'* condition. The output of the transposition function is dependent on the private key. Thus, it isn't possible to uncompress without the key, even if the algorithm is made public and an eavesdropper has access to the compressed cipher-text.

Thus the file "CIPHER.TXT" contains the cipher text corresponding to the contents of "PLAIN.TXT".

Sample Plaintext

LZW Compression

The LZW compression method maps strings of text characters into numeric codes. To begin with, all characters that may occur in the text file are assigned a code. For example, suppose the text file to be compressed is the string:

aaabbbbbbaabaaba

The string is composed of the characters 'a' and 'b'. 'a' is assigned the code 0 and 'b' the code 1. The mapping between character strings and their codes is stored in a dictionary. Each dictionary entry has two fields: KEY and CODE. The character string represented by CODE is stored in the field KEY.

Corresponding Cipher text

[Note: Cipher text not compressed yet]

```
6!V(]H7@l.]ƒ2iJ7,s3rt/"b!qƒ3|H4Tr2740Jt(Is+}/5+W7"„/^`(@o/UK7-O-
J„/Tt(5c/UN6{W#sD7@a!r?4iK5+X7"h/,b!r/0AK5+W#s<7@_(6•7}N5+W7#40r`5"?.iN
6^45Jt7@b!r/6AG5+W"6l/^c!|o1-;5+W7"„0Tc!|`3|^65W#}T0hc!s;1-
H5+X0_8.|b(]t.A96qW%UT0^b(6?5U96iP27H/,`/"O6AB6h42AT/^d!{$.AG6iO7"l/,t(
5#3|^7!W+sH7@_(6•7}N5+W/#8/hc"6O-
}L6^/8_X/"`.]#0AH6^32AT0rt(6_3U>6^/"AT3Ta.hp.A=7?W:7(/@a(6€3,^7-
P3rx/@a.h_/T^7!W+sH7@_(6•7}N5+W/#8/hc"6O6AG5+W5KH7@d.]ƒ2iJ7,s3rt/"c!s,.
AA7-K8^h0,c"6O5-N7,s-J€06n5]T$@E4T-(gt9^z"I$$@E6h3:7X0hd5"o/-
<6hs:7X0hd!r?/-94Tm(gt2,b(6€.AK7!X5K8/Tc/"O1-K5+W7"„/^`(5ƒ5-
=6]S8^„0@t(A+1U=5+W7#40r`5"?.iN6^45Jt7@s.^@0@^6h3"7D7@s.^p0@l5+S0_X6^t(
5#5,^6h47"t0"c.^//U>5+X2740Jt(6_3U>6^/8hx7@d!r//|^5#O5U@7@_(53/T^6iO#sD
0Jt(h@3|^65W+sH7@a!r?4iJ6|3"7@7@d5"•6AO6^33r€7@d.]3-}L6h37"h0J`5JO5-
N7,s-J€06`/"O-}H6]S8^h0,c!q#5T^6iO#sD0J`/"O1-
K5+X7"h/J`5"•/|^6|3"AT0rt(7+1-;7!W-
J„/Td!|o7Tl5+U3sX0^b(^O/}A6iP278/Ja5"?5UQ5+W3r€.|`5+$.AB6h47-
T.|_.]„.A@6|33s$0T`/+T.A#5r2-
UT0ra5#,.@•6#M26,5|t(U+1U=5+W7#40r`5"?.iN6^45UT/"_(@o1-
H6_K8^p0J`(@o/UK6^3"6h0Jc(^O/-Q5+U7!h3hk"6O1-
K5+X7"h/,a.^•/|^6|3"AT.|b(6€.A@6|33s$0Tt(IC"i156p*
```

Private Key Used

0 0 0 0 1 1 0 0 1 0

Disclaimer

This project, for me, is an ongoing process of learning; and like any other open source project, would claim to remain 'under development' till the end of time! The next version of my program would incorporate the following points I currently have in mind:

Compressing data *after* encryption to reduce redundancy and provide more immunity to frequency analysis attacks

Longer key length (so that a brute force attack would theoretically take longer than the age of the universe to complete)

Applying more randomization functions on the data obtained after calculating keyboard latency in order to make the process of key generation even more secure

Performing modular (clocked) arithmetic operations for key generation

Use a transposition ciphering function after the process of whitening and multiple block ciphering

Minor bug fixes, as discussed above

A *'Terminate and Stay Resident'* version would make it possible to read the encrypted file in DOS and transfer its contents to the Windows Clipboard. This is done by playing around with hardware interrupts at the OS level.

Another side project that I have initiated is making a TSR version of the program. This would make it highly inter-portable between Windows & DOS. My aim was to code in C++ (DOS based, not the Windows based VC++) but still make it possible for a user to "Ctrl. + V" the encrypted message into his browser when he arrives at the 'Compose Message' screen. This would eliminate the need to send the cipher-text as an attachment, thus making the entire process more convenient.

Since the instant messaging server is being developed on a Linux platform, coding a Linux based version of the cryptosystem wouldn't hurt either. Further ideas about the client-server model would be discussed in the presentation.

"Those who claim to have an unbreakable cipher simply because they can't break it are either geniuses or fools. Unfortunately, there are more of the latter in the world."

Wireless Security: A short note

The IEEE 802.11b standard uses WEP (Wired Equivalent Protocol) to send packets over a wireless network that are converted into a format equivalent to the regular Ethernet packets. The catch is that the moment a wireless base station is connected to the Ethernet, all the traffic on the Ethernet suddenly appears on the wireless interface (exceptions might arise, but this would certainly hold true in case of a non-switched topology). This is similar to the case where a LAN Interface can be set to *'promiscuous mode'* thus making it possible to intercept packets that are not intended to arrive at that interface.

What this boils down to is that if I take my laptop to the basement of your office that uses a Wireless LAN, and if your DHCP server permits an additional node to be added to the network (your administrator didn't consider defining IP Leases!), or if it were possible that my laptop got assigned an IP address under the same subnet as that of the wireless network, I could then simply run a network analyzing utility on my laptop and filter out packets and be able read plain text mails traveling within and out of your office's network. Sounds cool doesn't it? By the way, the same would hold true for being able read chat sessions (though messengers aren't permitted to be installed in an ideal office environment!).

Now that's what happens when you simply 'plug and play' your brand new wireless station, with the security options turned off as factory defaults. That's what happens at MTNL!

RSA Security came out with algorithms such as the RC2, RC4, RC5 that are used to encrypt packets flowing across a wireless network. The base station and the node share a secret key which is used for encryption/decryption of packets **[4]**.

To sum up, wireless security is yet another discussion altogether, and of course, is out of the scope of this paper.

Software implementation

For those avid cryptographers who don't find my results convincing but are still curious about what this is all about, here's an encrypted version of my programs source code:

http://ashishanand2.tripod.com/source/source.zip

References

[1] Applied Cryptography – Bruce Schneier

[2] D. Balenson, Feb. 1993. RFC#1423: *Privacy Enhancement for Internet Electronic Mail:* Part III: Algorithms, Modes, and Identifiers - Network Working Group

[3] D. Comer, D. Stevens, *Internetworking with TCP/IP* (Client - Server Programming & Applications)

[4] A. S. Tanenbaum, Computer Networks

[5] RFC#2440: *OpenPGP Message Format*

[6] www.pgp.com, www.openpgp.org