

DIGITAL CRYPTOGRAPHY METHODS

AARON ROSENFELD
JUNE 6, 2008
REVISION II

1. INTRODUCTION AND NOTATION

This paper is meant to cover some of the basic methods used in modern Cryptography and Cryptanalysis. It is not meant to cover the topic in great detail but does investigate different methods of securing data.

It is assumed that readers have a basic understanding of boolean logic and base conversion between decimal (base-10), hexadecimal (base-16), and binary (base-2). I will usually denote what base a number is in either with a subscript on the right side or use 0x as a prefix for hex or 0b for binary. Assume base-10 if there is no indication. All numbers are written with their most-significant value on the left, and if given a number N, N0 is the least-significant bit (right-most), N1 is the second-most-least-significant bit (second right-most), etc. I use parallel vertical bars to represent concatenation of two values. That is: $p \parallel q = pq$

2. BASICS

There are three basic requirements for any strong cryptographic algorithm: concealment, dispersion, and avalanching. Concealment simply means, given some plaintext (input text) character it should appear different in the ciphertext (encrypted form). This should seem obvious and I will append to this definition in our first example below.

Dispersion means that, given some string of input characters, the resultant ciphertext should not preserve the ordering of the input (even if they do meet the concealment requirement).

Many people consider avalanching part of dispersion but I like to keep them separate since they can each occur independent of the other. Avalanching is when a very slight change in input (only 1 bit even) causes a large change in the ciphertext. Take two MD5 hashes below. Only one character has been changed in the input:

```
Hello world! 86fb269d190d2c85f6e0468ceca42a20  
Hello wordd! 5447cf2589c0fb0b119cea40f48b9a51
```

3. REQUIREMENT ONE: CONCEALMENT

I think the best way to understand the first requirement is to do a case study. To make things easy, we will examine one of the most basic encryption method available: the exclusive-or. As a quick review, exclusive-or (XOR from here on) takes two bit inputs and returns 1 if one and only one bit is 1. This operation is denoted by circled multiplication sign. It can be more formally described by

$p \otimes q = p\bar{q} + \bar{p}q$. Logically, XOR can be applied to many bits by performing the operation bit-by-bit as in this example:

$$101010_2 \otimes 111000_2 = 010010_2$$

XOR is a very powerful operation because it is reversible. That is, if a bit p is XOR'ed with a bit q , to give a resultant bit y , the bit p can be determined by simply XOR'ing y with q (or q can be found by XOR'ing y and p).

Example: We have a plaintext of $0x61\ 0x61\ 0x72\ 0x6F\ 0x6E$ ("aaron" in hex), denoted P and we want to encrypt it with a single bit $0x11$, denoted K . XOR'ing each bytes can be expressed as:

$$(P_4 \otimes K) \parallel (P_3 \otimes K) \parallel (P_2 \otimes K) \parallel (P_1 \otimes K) \parallel (P_0 \otimes K) = \\ (0x61 \otimes K) \parallel (0x61 \otimes K) \parallel (0x72 \otimes K) \parallel (0x6F \otimes K) \parallel (0x6E \otimes K)$$

This gives us the encrypted text $0x70\ 0x70\ 0x63\ 0x7E\ 0x7F$ (the string is actually not printable in ASCII but that is irrelevant here). To get back to the plaintext P , this encrypted text can just be XOR'ed with the key again.

For the average person, the encryption above is a pretty good way of mixing things up; going from `aaron` to `70 70 63 7F`.

Not only that, it definitely meets the first requirement we set previously; the text is concealed. But I am now going to append to that definition: Concealment not only means that a given input character (or number) should be different after encryption, but all instances of that character in the ciphertext should not be the same (although they don't all have to be different either). In other words, the number of times a letter appears in plaintext should not be easy to determine simply by looking at the ciphertext.

Here is why: Lets look at the ciphertext. There is one very obvious repetition in the ciphertext, the first two bytes, this directly correlates to the first two bytes that were being encrypted! To a trained eye, this will indicate the encryption method may be weak, although this can happen by sheer luck with even very strong encryption methods.

The next logical question is "Who cares? There is still no way to get from $0x70$ back to $0x61$ without the key." This is somewhat true, but, it provides those trying to break the code (or obtain the key) crucial information to use to do something called frequency analysis.

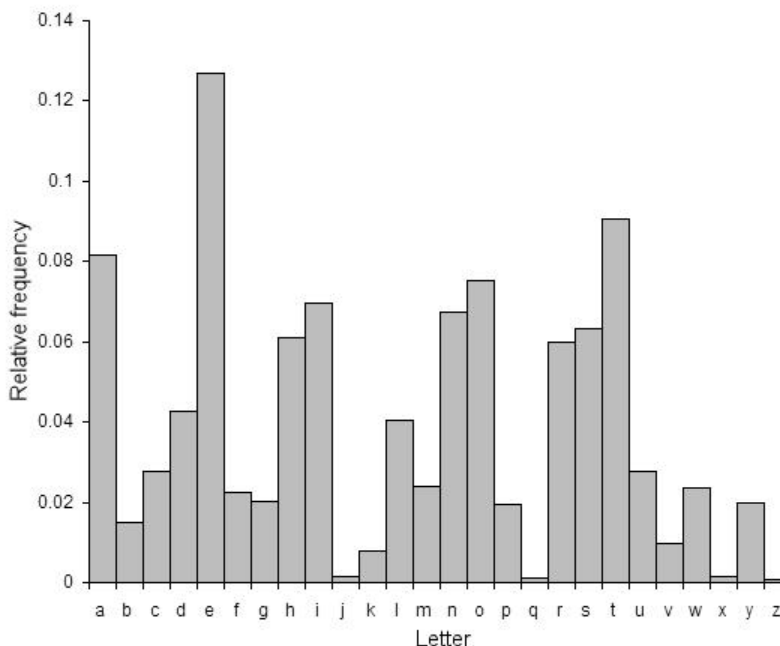
4. FREQUENCY ANALYSIS

Many newspapers have cryptograms-puzzles near the comics section; most of them use basic methods of non-digital cryptography that are based on a simple lookup table ('a' in the ciphertext means 't' in the plaintext, etc.) or other pattern-based method. These are fun but hold little practical value due to a number of cryptanalysis tricks developed over the years. The main one that I will be discussing is called Frequency Analysis.

Lets say we have a big block of ciphertext like the following (the spaces mean nothing and are simply there to break up the text, and the trailing x's simply pad the string):

```
XFUIF QFPQM FPGUI FVOJU FETUB UFTJO PSEFS UPGPS NBNPS FQFSG FDUVO
JPOFT UBCMJ TIKVT UJDFJ OTVSF EPNFT UJDUS BORVJ MJUZQ SPWJE FGPSU
IFDPN NPOEF GFODF QSPNP UFUIF HFOFS BMXFM GBSFB OETFD VSFUI FCMFT
TJOHT PGMJC FSUZU PPVST FMWFT BOEPV SQPTU FSJUJ EPPSE BJOBO EFTUB
```

FIGURE 1. Frequency graph for the English language.



CMJTI UIJTD POTUJ UVUJP OGPSU IFVOJ UFETU BUFTP GBNFS JDBxx xxxxx

To most people this could be solved pretty easily by guessing but we are going to do something a bit more analytic (although guessing is still a big part of it).

The English language is very well documented and information on letter-frequency is readily available as shown in **Figure 1**. From the graph, it is obvious that there some letters are used quite frequently ('E' and 'T') and some that are used very rarely ('Z' and 'J'). This information can be used to make more educated guesses at the encryption scheme's plaintext-to-ciphertext mapping. I won't list them all but the three highest frequency letters in our ciphertext are:

Letter	Occurrences	Frequency
F	39	0.15
U	29	0.11
P	25	0.09

The frequency is the number of occurrences divided by the total number of characters (268). So lets start by matching each of these up with their respective letter in the graph. The most frequently occurring ciphertext letter is F (by a lot), and the most frequently occurring letter in English is E (by a lot) so we will assume F in ciphertext maps to E in plaintext. Yes this is a guess, but a very logical one. We will do the same for 'U' and map it to the second-most-frequently occurring letter in English: T. Same for P but note that, since O and A are both very close in frequency we should try both.

This process continues until either the cipher is broken or enough of a pattern is understood that the rest can be figured out. From what three letters that we have mapped, F to E, U to T, and P to O (it could very well have been A and I had to try both), it may be clear that our cipher simply makes every plaintext letter the next letter in the alphabet. After doing the substitutions and placing spaces properly we end up with:

WE THE PEOPLE OF THE UNITED STATES IN ORDER TO FORM A MORE PERFECT
UNION ESTABLISH JUSTICE INSURE DOMESTIC TRANQUILITY PROVIDE FOR THE
COMMON DEFENCE PROMOTE THE GENERAL WELFARE AND SECURE THE BLESSINGS
OF LIBERTY TO OURSELVES AND OUR POSTERITY DO ORDAIN AND ESTABLISH
THIS CONSTITUTION FOR THE UNITED STATES OF AMERICA

I know I skimmed over this example and made some assumptions but the idea is what is important. In reality, a simple offset cipher like that is very simple to crack. It is harder when letters are randomly assigned or frequencies don't match up as well. In those cases we can use the same exact technique but use the well-documented frequencies of two and three letter strings or turn to other techniques (such as examining different letters' Index of Coincidences).

5. BACK TO THE CASE STUDY

As our detour has shown, a strong cipher cannot preserve the frequencies of letters. In our first example the only reason this occurred, however, was because our key was just 1 character long. Had it been two or more, the first a and second a would have been XOR'ed with different values thus yielding different ciphertexts. Case and point: keys should not be one character long and should minimize the chance of a given letter encrypting to the same thing too often.

6. REQUIREMENT TWO: DISPERSION

Until now I have not addressed the second requirement: dispersion. All the letters in the ciphertext, although in a different form, still represent the character in the equivalent position in the plaintext. There are many easy ways of fixing this but many of them are not reversible or do not shuffle enough for real use. I will explain a couple methods of achieving basic levels of dispersion and will then discuss the primary method that most well-known ciphers use.

Going back to the first example, our plaintext was 0x61 0x61 0x72 0x6F 0x6E and ciphertext calculated to 0x70 0x70 0x63 0x7E 0x7F. A simple method of shuffling is to simply rotate bits or bytes in a predefined pattern. For example, we could move every character left one position to get 0x70 0x63 0x7E 0x7F 0x70. We could do this in any pattern we want as long as it is reversible (in reality it can be irreversible but for our basic methods of encryption we need to have an easy way of going back).

Another method is to shuffle the bits in each byte by a certain amount. For those who are unfamiliar with what bit-shifting does, it simply moves all bits left or right by a certain amount and we denote these with \ll and \gg respectively. It is interesting to note that each bit-shift multiplies the number by 2. Here is an example:

$$1001_2 \ll 1_2 = 10010_2 \Leftrightarrow 9_{10} \ll 1_{10} = 18_{10}$$

Note that when we shift the number 1001_2 once, we just move every bit over to the left and add a 0 to the right side. The equivalent is shown in base-10 on the right. Assume all bit-shifts from here on are done in binary and then converted back to the base in which we are dealing.

This method is very well suited to disguise numbers (and characters by using their ASCII values) because it is seemingly random to the untrained eye. If we did a single left-shift on each character in original ciphertext, instead of $0x70\ 0x70\ 0x63\ 0x7E\ 0x7F$ we would have $0xE0\ 0xE0\ 0xC6\ 0xFC\ 0xFE$. One of the other methods that I personally like is shifting each byte its index plus 1. That is, for the least-significant-byte is shifted 1 time, the second least-significant by 2, etc.

Another important note is some people do shifts in a different fashion. Instead of shifting all the bits over by one, they literally rotate the bits. For our example above:

$$1001_2 \ll 1_2 = 0011_2 \Leftrightarrow 9_{10} \ll 1_{10} = 3_{10}$$

Note that the leftmost bit is not dropped but wraps back around to the right side. This is my preferred way of doing shifts because it guarantees the number of bits will remain unchanged. Note that I put the base-10 equivalent in just for reference. The math behind rotational shifts like this gets a bit messy (although not too complex) and is really unimportant to us since we are simply using it to mask our original bits.

From now on when I talk about bit-shifts I am referring to this rotational method unless I not otherwise.

Bit shifts, byte shifts, and XOR's can be combined in many different ways to make cracking a cipher much more difficult. This is pretty impressive for a single-round encryption that has nothing but an XOR base to it.

7. REQUIREMENT THREE: THE AVALANCHE EFFECT

This the first place I will address the avalanche effect but we will develop the ideas more fully in the final section. Our encryption algorithms so far have basically just masked our input text and moved the bytes around. But imagine this: lets say we have a 1 round encryption scheme that XOR's once with a key given by $0xA1\ 0x12\ 0x41$ and then rotates all the bytes once. As an example we pass it the plaintext $0x01\ 0x02\ 0x03$:

$$\begin{aligned} (0x01 \otimes 0xA1) \parallel (0x02 \otimes 0x12) \parallel (0x03 \otimes 0x41) &= 0xA0 \parallel 0x10 \parallel 0x42 \\ (0xA0 \parallel 0x10 \parallel 0x42) \ll 1 &= 0x10 \parallel 0x42 \parallel 0xA0 \end{aligned}$$

Now imagine we do all of that the same except we feed it $0x42$ instead of $0x41$ for input. Our output would be $0x10\ 0x41\ 0xA0$. Pretty close to the original isn't it? This can become a fatal flaw in an algorithm. By trying many different combinations of inputs (or in this case, one), we can easily determine what output byte corresponds to what input byte. From that, the key itself can be found by calculating the difference between the input and output bits in that byte.

The way to combat this is to assure bytes are dependent on many other bytes. In other words, if we added another step to the encryption process that XOR's every byte with every other byte, a change in any single byte will be propagated to all others and cause a much larger difference between two ciphertexts even if their plaintexts are very similar.

8. ROUND-BASED CIPHERS

Everything we have talked about until now is what's called single-rounded. An input is given to a function that performs some shuffling and distorting of the data and it then gives us a ciphertext all in one go. The next important topic is using more than one round for encryption. Entire books have been dedicated to this topic so I will barely scratch the surface but I will try and cover the general concepts.

A round based cipher essentially just does what we have been discussing a number of times. Each round ciphertext is manipulated by a function (like what we have been talking about) and the output is used as the input for the next round. What is unique about most round based encryption methods is they don't use the same key every round. Instead, a key schedule containing one key for each round is generated from the base key. Many times, additional changes are made at the beginning and end of encryption based on the base key itself, however.

Until we get to the last section of this article, we must make the round function reversible. Likewise, the key schedule must be generated in such a way that, given the same base key, all sub-keys will be the same.

To decrypt, the key schedule is regenerated and the rounds are run in reverse but with the round function this time un-doing whatever the original round function did.

What the problem is with this whole scheme is generally round functions are very complex. They involve bit-shifting, byte-shifting, XORing, masking bytes with other bytes, and generally moving things around a lot. Making all of this reversible is actually much harder than it sounds and, in many cases, impossible. Here is an example.

Say I want to make a round function F that XORs the current input P_i with the current key K_i and then XORs itself with the original round input shifted by one byte $S(P_i)$ (where S just denotes the shift function). We know from the previous section this will promote an avalanche effect.

This seems to be a pretty good method but it cannot be reversed. The general form for a given round i can be shown as (P_0 is our starting plaintext and assume our key schedule has already been determined and the key for round i is denoted by K_i):

$$P_i = P_{i-1} \otimes K_i \otimes S(P_{i-1})$$

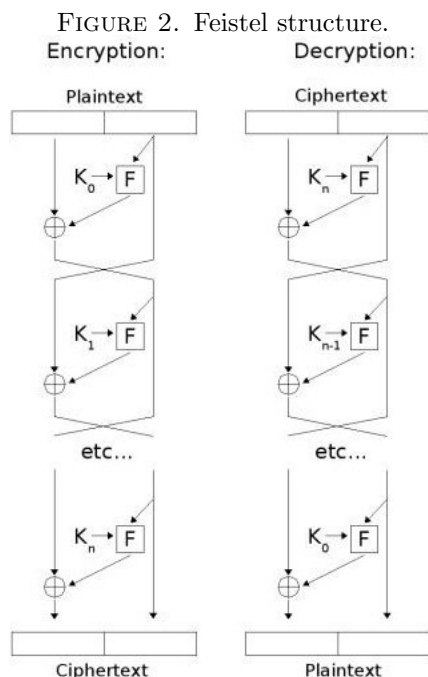
Thus, if we run this cipher for three rounds:

$$\begin{aligned} P_1 &= P_0 \otimes K_1 \otimes S(P_0) \\ P_2 &= P_1 \otimes K_2 \otimes S(P_1) \\ P_3 &= P_2 \otimes K_3 \otimes S(P_2) \end{aligned}$$

Now, if we try and run that in reverse we have a problem. Any given step requires the previous steps' output. When we were encrypting we had access to this. Going the other direction we do not. This would work as a one-way hashing method but it fails for two-way encryption.

9. MORE AVALANCHE EFFECT AND THE FEISTEL STRUCTURE

The way we get around this problem was discovered by Horst Feistel in the 1950's. His method, known as the Feistel Structure, is by far one of the most important algorithms in modern cryptology and is used by some of the big-names such as DES and Blowfish. The beauty of his algorithm is any round function can



be used(it does not have to be reversible), and it maximizing the avalanche effect in only a few rounds.

The way it works is ingenious. The plaintext is split into two equal pieces (halves) denoted L and R. During encryption the right half is sent through the round function along with the round key. That is then XOR'd with the left half. That entire block is then used as the right half of the next round's input and the left half of the current round's input is used as the right. It is easier to understand by analyzing these two equations and referring to **Figure 2**:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \otimes F(R_{i-1}, K_i) = L_{i-1} \otimes F(L_i, K_i) \end{aligned}$$

The most important thing to notice is that the second step does not involve anything from the previous step as in our last example. This can be shown by rearranging the expressions above and changing the indices for easier understanding:

$$\begin{aligned} R_i &= L_{i+1} \\ L_i &= R_{i+1} \otimes F(L_{i+1}, K_i) \end{aligned}$$

This means, to unencrypt we simply iterate from $i=n$ down to $i=0$ and we will arrive back at our plaintext! This is quite amazing seeing that we can use any round function even if data is discarded since they are still preserved (in a different form) in the other half.

REFERENCES

- [1] Jacques Patarin, Luby-Rackoff: 7 Rounds Are Enough for Security, Lecture Notes in Computer Science, Volume 2729, Oct 2003, Pages 513 - 529
- [2] M. Luby and C. Rackoff. "How to Construct Pseudorandom Permutations and Pseudorandom Functions." In *SIAM J. Comput.*, vol. 17, 1988, pp. 373-386.
- [3] Swenson, Christopher, (2008). Modern Cryptanalysis. New York: Wiley.