Developing a Strong Cipher

Rod Hilton

December 20, 2003

Abstract

The vigenere is weak because of key periodicity, but the fake one-time pad based on a random number generator removes that periodicity. This fake one-time pad, however, requires seed-based random number generation and it is therefore a mathematical linear sequence, which lends itself to various flaws as well. I propose that the vigenere can be modified from its base implementation, without relying solely on random number generation, to eliminate key periodicity as well as generally strengthen the cipher by using the key and a series of subsets of the key to encrypt to the message so strongly that no Friedman-like attack can be used against it.

Hypothesis

There seems to be a simple modification to vigenere that has not been studied previously (at least, I cannot find mention of it in my research) that immediately strengthens it. If a message M is encrypted by key A of length a and then key B of length b, as long as the two keys are not the same length, the result is encrypting by a key X of length $lcm(a, b)$. So if a message was encrypted by three keys of different lengths, it would actually be encrypted by a key of the length of the lowest common multiple of those three lengths. Furthermore, if those three numbers are next to each other, because of the rule that no two adjacent numbers can have the same factor (if they did, that factor would divide 1, which is not possible), then for a sequence encrypting by keys of lengths $1, 2, 3, 4...n$, the resulting encryption is using a key of a length that is likely to be very large. Thus, if vigenere was modified to use a password of, say, "PASSWORD", encrypting by "PASSWORD", then "PASSWOR", then "PASSWO", then "PASSW", then "PASS", etc, would be resulting in some non-english key of length 40,320. This method, too, has some flaws, which I propose to find and then eliminate, by modifying the cipher method to no longer manifest those flaws. This method of encryption, combined with token transposition, ought to result in an incredibly strong cipher.

Motivation

For a number of reasons, I have a serious personal investment in developing a stronger vigenere cipher. A while back, I wrote a program called NotezMaker, a little sticky note utility. It was, in actuality, the only real program I've written (I've written hundreds of little utilities to automate tasks, but NotezMaker grew to a point where it was something other people might use). I put this program on my web site, and found that people actually downloaded and used it. Eventually I realized people may want to encrypt their notes. I figured I could find some encryption algorithm on the web, but most of them wound up giving me strange ascii values or making the size of the note larger - I needed to write my own. At this point the only encryption I knew was a monoalphabetic substitution cipher (though I did not know it was called this). I knew that checking how often letters occur could break it. I devised my own system, and it was relatively simple - get the person's password, then encrypt the first letter of the plain text by the first letter of the password, the second by the second, etc.

It wasn't until CS290 that I discovered that this cipher had already been invented, and that there were ways of breaking it. I was skeptical. This seemed like a pretty good cipher to me, especially because I

thought of it. When I was told I needed to crack such a cipher, it was an opportunity to really come to learn about how secure or insecure my cipher was. Upon completion of Assignment 4, I discovered just how truly weak the cipher is, however, I have no intention of giving up. NotezMaker needs a stronger cipher, and that is what I intend to create.

Fake One-Time Pad

First, it is important to establish that the fake one-time pad is not sufficient. The Fake OTP is a lot like the vigenere, so it would seem to be the obvious choice and allow me to bypass re-inventing the wheel. Unfortunately, there is an inherent weakness to the system which comes down to the power of the random number generator. The fake one-time pad, as detailed in the class textbook [2], basically creates a "keystream" by starting with some number seed (your key), and then using a random number generator to get the next element of the key, using that as a seed, etc. There are a number of problems with this. First and foremost, it is always going to be a linear sequence of numbers, with one number depending on the previous one, which makes it susceptible to a mathematical attack[1]. The book, however, does admit that the primary methods of random number generation, such as pRNG or Linear Shift Registers, are not secure enough for cryptograpy, and that only the Blum-Blum-Shub (or Naor-Reingold [5]) generator is. For a lot of cryptology, this would be enough - in BBS, you get two large primes p and q and then multiply them together to get n much like in RSA. Seeds are squared and modded by this n to result in a series of random numbers that are provably random as long as large numbers are hard to factor (which, in cryptograpy, we generally assume to be true).

More importantly, however, there is a crucial problem in the entire system, not so much a flaw, but more a difficulty - something that makes it impractical for a number of purposes, most importantly mine. To make sure the number n is too large to factor, it needs to be significantly more than 32 bits (in fact, p and q should be more than 32 bits) which means that any program implementing the Blum-Blum-Shub requires an infinite precision package. Languages like MATLAB and Python have this built in, but my reasons for wanting a good cipher like this one are for a practical application - one that requires more speed than Python and more functionality than MATLAB. The truth is, even Java's infinite precision package leaves a lot to be desired, and considering the practical need for this cipher, something requiring infinite precision is not really suitable.

While Blum-Blum-Shub is a fantastic way of generating pseudorandom numbers, its implementation is too complex for my purposes and its mathematical requirements too unavailable. While a Fake One-Time Pad would be a very good method of encipherment, it isn't exactly what I want. It seems, actually, that any cipher method relying on pseudorandom numbers will fall prey to mathematical attacks or brute force unless their mathematical basis is in numbers so large that infinite precision is required. In many ways, a complete reliance on random number generation is bad for my purposes no matter how it is done - either requiring infinite precision or severely limiting the keyspace. In any case, pseudorandom number generators appear to be the wrong direction for my purposes, so it would be better to start from scratch with regular Vigenere and modify it to fit my goals.

Goals

What I need in a cipher differs quite significantly from what the needs satisfied by most other ciphers, so it is not surprising that the existing popular ciphers don't quite satisfy my requirements. As a result, I am somewhat forced to explore this issue on my own, developing my own cipher that meets my goals and, at the same time, appears to be unbreakable based on the analytical techniques I have learned through both this course and my own independent research. The goals for my new cipher are as follows:

1. **Unbreakable** - Obviously, as with any cipher, this cipher should make it impossible or at least incredibly difficult to decode a piece of ciphertext without the key.

2. **Password-based** - Unlike many of the contemporary ciphers such as RSA or Fake OTP using BBS, my cipher needs to be encrypted and decrypted with a plain text password, not prime numbers. While it is possible to use some password to generate a prime number, getting a prime number based on *all* of the characters in a password would require an infinite precision math package. Speaking of which...

3. **Implementable With Unaugmented Software** - The cipher needs to be able to work on a message without requiring something like an arbitrary precision arithmetic package. The cipher's primary use will be as a method of good encryption that could be used in any program (specifically NotezMaker) without requiring additional libraries or special hardware.

4. **Preserves Size** - The size of the encrypted message should be identical to the size of the original message. For most ciphers studied in this course, this property is held, but for a number of popular password based encryption libraries, the file size of the encrypted message is significantly larger.

5. **Carries Over Non-alphanumeric Properties** - The punctuation, and spacing need to be preserved from the original plaintext message. The reasons for this are largely aesthetic. In essence, if there was a large set of messages that had been encrypted and forgotten about, there is a good chance that the original encrypter can figure out what the message is about (and therefore remember the password he or she used on it) because, if spacing, punctuation, and letter case are all preserved, the message will actually look just like the original, only unreadable.

6. **Allows More Than 26 Valid Characters** - Somewhat trivial, but the cipher needs to work for upper case and lower case letters, as well as digits, encrypting all of the characters in that set to a different character in that same set (treating digits as capitalized in order to follow the case rule mentioned above, since all digits have the same text height as a capital letter).

7. **Not Vulnerable to Known-Plaintext Attack** - Because of the nature of other requirements, it is conceivable that one could guess at portions of text in an encrypted message (a three letter word after a period is probably a "the", or an 8 letter word with a hyphen after the fourth letter is probably "home-brew"), which would work as partial known-plaintext. To be safe, then, I'd like for a known-plaintext attack to be as difficult as a ciphertext-only attack. Because this is based on vigenere, a lot of work in that regard has already been done, but there is still room for improvement.

Methods

My attempts to remove key periodicity and strengthen the cipher went through numerous revisions and discoveries before arriving in their final state. These revisions are detailed here.

Revision 1

As mentioned in the Hypothesis, the first change to make to Vigenere is the recursive re-application of a key. A message which would normally be encrypted with "SECRET" would be encrypted by "SECRET", then "SECRE", "SECR", "SEC", "SE", and finally a caesar cipher shift of "S". In this example, it would lead to a message being encrypted by the following key (spaced for readability):

```
EMSNON AXXNMM OZUOOX NZYNWZ QAUYBZ OZIAYA QKHACZ YMKBYK DMIAMM ANKLLM EMSNON AXXNMM
OZUOOX NZYNWZ QAUYBZ OZIAYA QKHACZ YMKBYK DMIAMM ANKLLM EMSNON AXXNMM OZUOOX NZYNWZ
QAUYBZ OZIAYA QKHACZ YMKBYK DMIAMM ANKLLM EMSNON AXXNMM OZUOOX NZYNWZ QAUYBZ OZIAYA
QKHACZ YMKBYK DMIAMM ANKLLM EMSNON AXXNMM OZUOOX NZYNWZ QAUYBZ OZIAYA QKHACZ YMKBYK
DMIAMM ANKLLM EMSNON AXXNMM OZUOOX NZYNWZ QAUYBZ OZIAYA QKHACZ YMKBYK DMIAMM ANKLLM
EMSNON AXXNMM OZUOOX NZYNWZ QAUYBZ OZIAYA QKHACZ YMKBYK DMIAMM ANKLLM EMSNON AXXNMM
OZUOOX NZYNWZ QAUYBZ OZIAYA QKHACZ YMKBYK DMIAMM ANKLLM EMSNON AXXNMM OZUOOX NZYNWZ
```

```
QAUYBZ OZIAYA QKHACZ YMKBYK DMIAMM ANKLLM EMSNON AXXNMM OZUOOX NZYNWZ QAUYBZ OZIAYA
QKHACZ YMKBYK
```

The problem here is that the same letters are constantly re-encrypting the same letters (for proof, notice that the first letter merely is encrypted by the character S six times). Another problem is that, due to the repeated letter E, a new, more sparse periodicity is introduced.

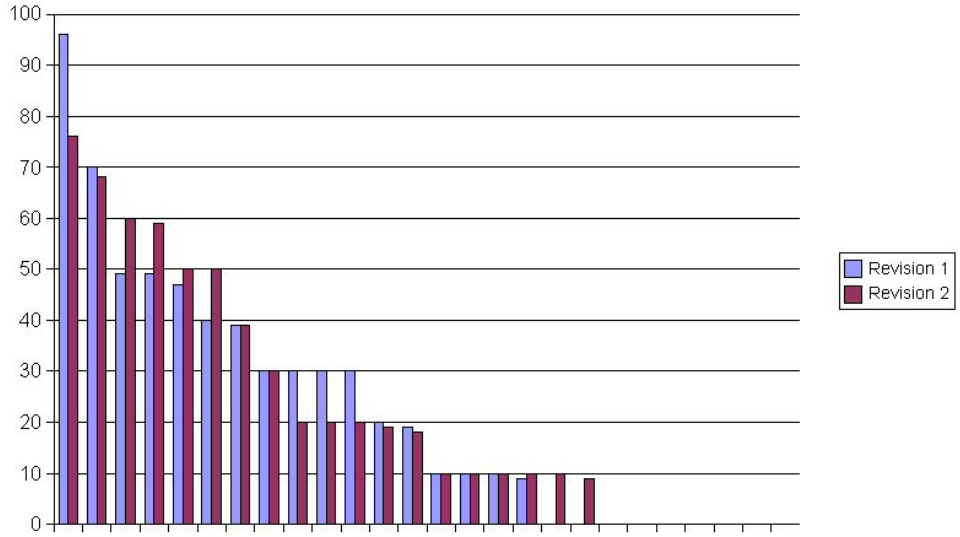
Revision 2

Another modification can potentially solve this. Each recursion into the key, instead of trimming one letter of the end and encrypting, it trims one letter off, reverses the string, and encrypts. So the encryption order would become "SECRET", "ERCES", "ECRE", "RCE", "CR", and finally "R". This basically accomplishes a back-and-forth cutoff of a character from the beginning and end of the string, and thus should eliminate a lot of periodicity.

Indeed, using this new method yields a similar output, one which is mostly just random letters, but it is worthy of a bit more confidence than the previous one.

```
VSDJGW VFSVFL TFTJFU IVSUDY JFSHSK IUQHTY IDFXSJ GHGHSW VTFWQW WHFFFM VSDJGW VFSVFL
TFTJFU IVSUDY JFSHSK IUQHTY IDFXSJ GHGHSW VTFWQW WHFFFM VSDJGW VFSVFL TFTJFU IVSUDY
JFSHSK IUQHTY IDFXSJ GHGHSW VTFWQW WHFFFM VSDJGW VFSVFL TFTJFU IVSUDY JFSHSK IUQHTY
IDFXSJ GHGHSW VTFWQW WHFFFM VSDJGW VFSVFL TFTJFU IVSUDY JFSHSK IUQHTY IDFXSJ GHGHSW
VTFWQW WHFFFM VSDJGW VFSVFL TFTJFU IVSUDY JFSHSK IUQHTY IDFXSJ GHGHSW VTFWQW WHFFFM
VSDJGW VFSVFL TFTJFU IVSUDY JFSHSK IUQHTY IDFXSJ GHGHSW VTFWQW WHFFFM VSDJGW VFSVFL
TFTJFU IVSUDY JFSHSK IUQHTY IDFXSJ GHGHSW VTFWQW WHFFFM VSDJGW VFSVFL TFTJFU IVSUDY
JFSHSK IUQHTY IDFXSJ GHGHSW VTFWQW WHFFFM VSDJGW VFSVFL TFTJFU IVSUDY JFSHSK IUQHTY
IDFXSJ GHGHSW
```

Just as before, it looks like a mostly random sequence of letters. To see if this was any improvement, I checked their frequency distributions and compared them. Of course, which letter occurs x times is not important, but what is important is how often any given letter appears in the key. Both graphs, therefore, will be sorted in descending order, starting with the most common letter and going to the least common.



This distribution is a little more spread-out in the second one. It's maximum is also less than the first maximum. This method of encipherment worked a little better, though not all that much.

Another, more important problem can be seen as well. After 60 characters, there is a period, which needs to be resolved. The reason for the periodicity of 60 is that the key, SECRET, is 6 characters long, and $lcm(6, 5, 4, 3, 2, 1) = 60$. The next important step is to eliminate this problem.

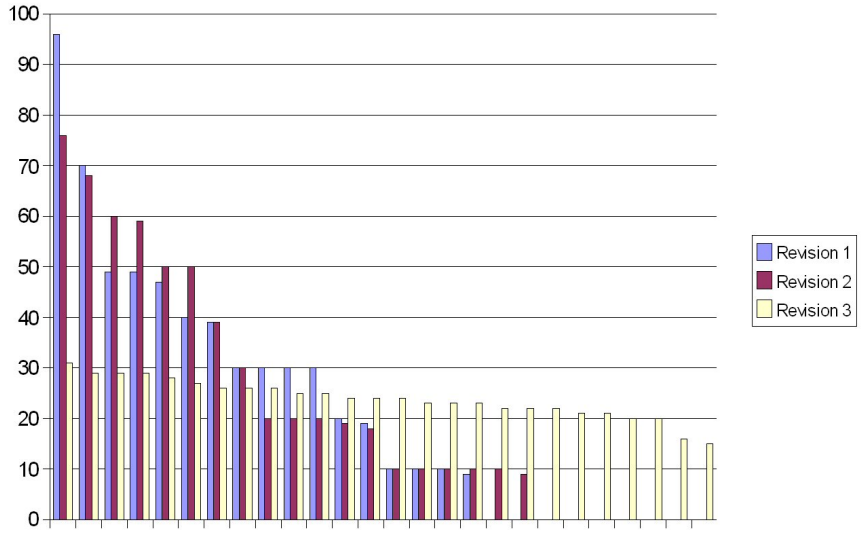
Revision 3

If I am re-applying subkeys for all lengths less than the original length, then it seems clear that, the longer original key, the larger the lcm will be. Thus, a good way to eliminate this periodicity is to first build a very large initial key, then apply the subkeys based on that one. So I decided that, before the key application, I would build a key of length 100.

For my first attempt, the initial key creation consisted of dividing the initial key in half, then setting the key to be the second half concatenated with the first half, concatenated with the reversal of the second half concatenated with the first half. To be put another way, key K is cut in two pieces, $K1$ and $K2$. Then, set $K = K2 + K1 +$ Reversal$(K2 + K1)$.This would double they key size each time through, arriving at the destination size of 100 quickly. Doing this led to the following key:

```
USCXWX WAOHFL NCQZYN HHQLHP CXTSXV TKLOJY EWHPOJ HKWOLB VQHPPW KPADBH QIOLOV CCYVIY
ZPMDAY IMNOKN ENODHH ERUCUF JFEWGM REQEBW YQJYWU LOKHFY TBFOVQ ZWOKDW AYXANS KWKSST
NSKLEE NPCPSI JGBHLI DKYFWI NRFGLY LVUMVH IXYRQI LDKYHY YBMKPA IJUEJH PQWMGD MDVXIN
XRDIDU AQJAXK PTMIQN GRMBQE NIVQRC MFARAR GQZQKG DVAMRH LOJAUF FXWAMC USLDJU LBULAU
ZZADQL NTCPKT STWQRQ RTVEJY TSLCKZ GGIXBQ CFVWDS XWZLMM PHCFHR MBJLNX HNICYT RQMVSW
THUTVF FVMKIH GUXPZR HFOLXW ZUEJEM VPFOSL DJRJEH OMPVNC HQEKZV URWXKZ FOGJDF TZMHUX
IWPHCJ SMXLIJ MYDNRD DMUAHN XMKXMO EWLELY VWBTOE YQPGGQ TFBZAW JSSPXS SLWPJG LAHOBE
DKOGHW SMHQEQ OMTELA OECSXD ARPZIN NVKTLI UOBGEG PJKACL XRWEUG JKREQV ERKDTM GCOBOE
PAFYYE FUAKRA
```

This seems to look much better than the previous two. Adding it's data to the graph, we get this frequency distribution:



This is a vast improvement over the other two - the distribution is almost uniform. I further analyzed the key by looking at more letters than just these few and performing a trigram analysis. Checking the first 2000 characters, all trigrams occured only once, with the exception of three: AQN, QNC, and WGR, which each occured twice. If any portion of the text repeated periodically (that is, not due to random chance), these numbers would be much higher, so it seems clear that there isn't any reasonable periodicity in this key.

Revision 4

It then occured to me, 100 seems like a small cap, when the initial key could be the original size of the message. I made this change, and checked its statistics. As it turns out, the frequency was about the same, but a new problem has been introduced. A slightly longer message length would lead to a significantly longer running time for the encryption, which is very bad. I settled on 200 as the length, as it should be long enough that any key periodicity won't be visible for most messages, and if it was visible for a message, it would be useless, as a "strip" analysis like Friedman would give only three or four characters.

Revision 5

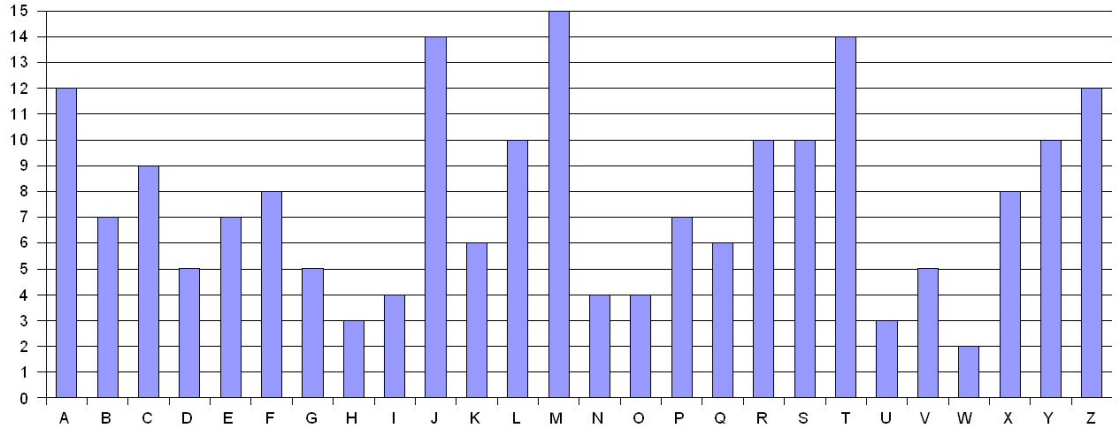
One problem is that the initial key, as "random" as it may be, still will always contain exactly the same characters as the initial passsword. It would be better with a more varied set of characters. To accomplish this, I introduced a salt - some extra, small password that can be used with the original one. For NotezMaker, the salt will be the name of the note, though the username would make a good salt as well.

So the new initial key building works like this: divide the key K into three pieces, $K1$, $K2$, and $K3$. Then set $K = K2 + K3 + K1$. Then, encrypt K by regular vigenere using the salt as a password (for testing purposes, I used "RHILTON") and call that T. Then set $K = K + T$. Because it will be divided into thirds, the divisions will not line up with this concatenation (the concatenation will double it each time). Meaning, the next time through the loop, K1 will be all regular key, K3 will be all encrypted key, and K2 will be a nice mix of the two. The ordering used (2, 3, then 1) will move that nice mix to the front, where much of it will not be used on the next iteration, and the ordering will also move the non-encrypted part of the key to the end. This has the nice effect of leaving a number of characters unencrypted by the salt, while also encrypting a lot of characters by the salt different numbers of times.

Doing this with password SECRET and salt RHILTON yielded this as the final inital key:

```
SETYME LSTYME CZOAMA JCXMGT APMMZV SATYME CZOAMA JCXMGT RWUXSJ FRLZLK HTNSNZ PHRLYU
ZVRGYN TZIWYS BMAPKM XBKYVR ZMERAR CZBJFS PQVIXT XPOTOE TDZDGD DTCRET JLBJFS YJJZPM
JLBJFS YJAGXX QMFHUL CQKDNB LIAZQC ALVFRK QTMCTF PQRKIA JLBJFS YJ
```

The frequency analysis for this initial key looks like this:

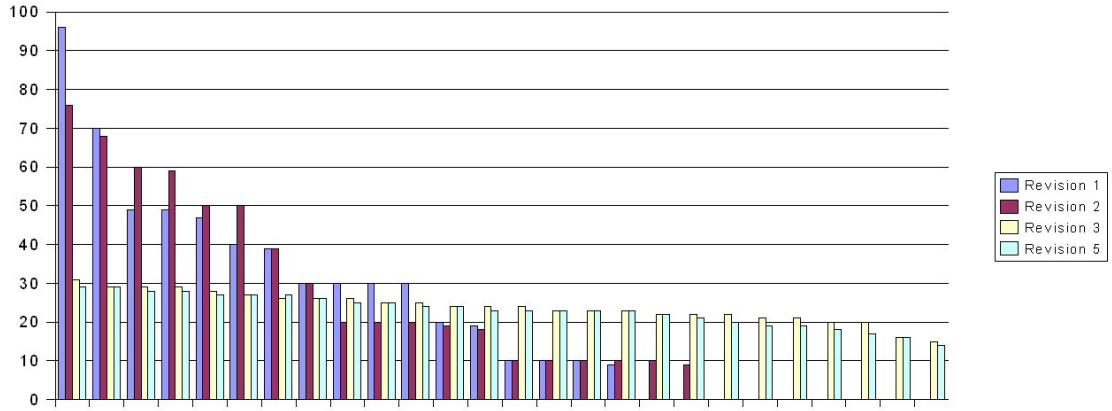


This is much better than the original initial key, which only used the letters S, E, C, R, and T. It would be ideal if this distribution was uniform, but it isn't too terribly important, since this initial key will be combined with the successive key encryptions. If we use this initial key with the rest of the algorithm, the resulting virtual key is this:

```
LMLNET UBFJMX IJYSVZ CYOGUS BCJUST ZIXPKM MNDCDZ TYONLM VWJKBK SJOZJG MJWTOH YSYSTY
MOJXOB PQHJSL GYXLWE CBRCUN QHEJNI ZZMRMS VERVSC CRAWAU JGMDZB CMIUYP IAAIOQ EICLTL
```

```
ZWJNCE JRBNEI ZPLOCL KMSFPN EKALCD UJPOEF ZTOXBR YHJPBJ YTDEKE KAWHGU GWMNDM ROKSCT
EXGQGT KDWKAR GGMLPK DCSDGR RONYBA MDUAAT DFWCNU TBXXIE ADQMQT EDJPSF ZXBTJI OWFWWY
CBORHU XPLOJI KEHEWU ICWIZU NYXSNL KQLPPC RCLMYG XWFTCT KBUOVG BSSKKC GLZYTY OYAAIH
YGTLZH KFVSET QXECPM UNRQWX PXNEUT PWBIKP FYTHPM RVJOZS RRFSIX DIQHLT WTSJJX EHYQWD
LXOMNK ZNEAZK KWJQKC SBHLPM DRVIBX YGRMGS OIFZLR IZOFQF LNPFID JIJKIQ HFDIJW SCOPCV
YVLRVI FAXRTU FFDWRS XQQJYI AFBWEH DCQNVG RPXMNG VIZOOE OYDLLF LHFDLY RJDRTL NTGTAC
FVKRSX PQGSLG
```

When compared to previous revisions, this new revision does pretty well. The letter distribution is a little more uniform, though not much. The graph does not make it clear that this is an improvement, but the very idea of randomizing the initial key using a salt is, on it's surface, going to make it harder to break the cipher. At the very least, it's a little bit of an improvement and it doesn't hurt the security of the cipher at all.



Revision 6

At this point, the letter frequency on the virtual key seems pretty uniform, so it's as good as it's going to get. The next issue that needs to be tackled is design goal 7: that it should not be vulnerable to a known-plaintext attack. If one found in the encrypted note "ASYX-QUZA", it could be deduced that this was "HOME-BREW", the only english word that fits that pattern. From this, there are only so many keys that could have been used to get from H to A, O, to S, etc.. given the position of the word in the text stream. This would make it possible to figure out the initial key, given enough pieces of "known" plaintext, and that is a big problem.

My solution for this was what I called the Vigenere Transposition. A random number generator is a mistake as the primary wall between an attacker and a plaintext message, simply because it, for these purposes, would severely limit the keyspace. However, a random number generator with a specific seed would be useful, if combined with other methods of encipherment, such that the strength of the random number generator is not the only defense.

Any list can be shuffled quickly using the following algorithm: For each element in a list, come up with a random new position in the list and swap it with the element in that position.

One interesting property of this method of shuffling is that, given the same seed and the same length of a list, the shuffle will be the same. I used this to implement the Vigenere Transposition. Instead of a character of the key being used to affect a character in the plaintext (as in regular Vigenere), the transposition uses one character of the key for every token. The ciphertext can be tokenized easily, since spacing and punctuation are preserved. Then a shuffle is performed on this token, based on the appropriate character from the initial key as a seed. This mapping can be generated for decryption as well, and the process can simply be reversed.

To view the results of this, I encrypted the phrase "HELLO, MY NAME IS ROD! THIS IS MY ENORMOUSLY USEFUL? METHOD OF VIGENERE... TRANSPOSITION!" using the password "SECRET".

Note, however, that the key that is used for the vigenere transposition is the initial key that comes to exist using password "SECRET" and salt "RHILTON", not "SECRET" itself. This resulted in the following string of text:

> LHLEO, YM ENMA SI ORD! ISTH IS MY NSROMOEYUL FLUESU? DEHMOT FO EGNRIVEE... TNRISIAOSTONP

This is quite scrambled, and would thus make it basically impossible to know which letter of plaintext maps to which letter of the ciphertext, had it been encrypted. Even if the random number generator was weak, its combination with the regular encryption (plus the fact that it's not generating a stream of integers, but a random index into a variable length array) makes breaking the RNG not sufficient to break the cipher.

Revision 7

In order to tokenize the output as well as accomplish the allowing of more than 26 characters to be encrypted or output, there must be a Valid Character Library, which is used to figure out which letters are valid parts of a token. This library is, by default, A-Z, then a-z, then 0-9. If a cracker, however, did not know the order of the characters in the library, it would make it incredibly difficult to crack. In much the same way as we shuffled tokens, we can use a seed based on the password to shuffle the library as well. Most random number generators allow their seeds to be long integers, but some use integers. Because I want this cipher to be portable, I do not wish to rely on a random number generator that allows the seed to be a long, so I will use an integer. The task here was to devise a way to turn the password into a seed, while using every letter of the password.

If a random number generator is weak, it's primary method of attack is a differential analysis of its values, which are used to predict future values[3]. This is important, as it means that the ability to break the random number generator depends on the ability to look at previous values in a way. This makes the fake one-time pad potentially quite weak, as you can look at a series of a lot of characters and use english frequency distributions to make educated guesses about values, then perform a differential analysis attack. So, if you were to do some act on some input twice, each time using two different seeds, this sort of attack would be unable to succeed. Even if the random number generator is fairly weak, successive applications with different seeds would yield results that could not be subject to a differential analysis attack. This is the basis for how I used the password to shuffle the library.

A simple method would be to group letters together and use the resulting bit string as a seed, but then it would only use as many letters as would fill however many bits are allowed to be used for the seed. The next option was to take groups of that many letters and exclusive or their bit strings together, but that made certain assumptions about the number of bits allowed in the seed. However, using a small seed, but a lot of them would accomplish what I would need to thwart differential analysis. The library, thus, is shuffled as follows: for each letter of the password, use it's ascii value as a seed, then shuffle the list one time. This creates a shuffled list in the end, one that was shuffled n times using n seeds, where n is the length of the original password. Though the seeds are very small (and the seedspace therefore limited), the repeated applications make analysis virtually impossible. Additionally, though the individual seedspace for a single shuffle is quite small, the overall keyspace for the last shuffle is still enormous, eliminating brute force as an option.

One test of this Valid Character Library shuffler resulted in the following library:

> MN5q8aTOPDQjuRhVEboUIFk1ZA0rwBCxnstHKz2leYifXLv7ymG46Jdc3SpgW9

So 'M' would be 0, 'N' would be 1, etc. Encrypting a letter 'X' by 'B' would no longer result in 'Y' ('B' being 1), it would result in encrypting letter 44 by letter 29 mod 62 to get 'j'. Without knowledge of this library, an attack would be almost completely impossible, and each library is generated by the password.

Revision 8

The combination of the random library, vigenere transposition, 100-character initial key, and successive key applications makes this cipher incredibly secure, but one important problem remains. If the original message length is, say, 1000 (not unreasonable), then, after the initial key of length 100 is created, it will apply 100 vigeneres to the 1000-character input, for a total of 100,000 total letter shifts. This can be incredibly slow, and it's even worse for larger inputs. At this point, the successive key application is not the only thing keeping the cipher secure, but it is the thing that slows it down the most. Even if the actual encryption weren't as strong, it would still be impossible to take advantage of key periodicity due to the transposition, and it would be hard to even make sense of the results of any attack due to the library shuffling. The seemingly random initial key, too, makes a large key out of a small one. Though the successive key application was the basis for the strengthening of vigenere, it seems like the least useful, solely due to its negative impact on the algorithm's running time.

Ironically, I decided the best solution was to use the initial key's hashcode as a seed in what can only be defined as a fake one-time pad. Using the random number generator, I applied a "random" offset to each letter. The reason I felt safe doing this is that the combination of this with the random letter library and the vigenere transposition made it so that the strength of the random number generator wasn't really a factor. Even a weak one used on all three of these parts of the cipher would make it impossible to crack, which avoided my problem with the fake one-time pad in the first place. An attacker cannot gain any useful information about a series of randomly generated numbers based solely on the text, because breaking the generation for encipherment is impossible to do since the transposition would disallow one from telling the sequence in which the random numbers were used. One cannot figure out the transpositions without the initial key, which was created using the salt and the regular vigenere, which used the random character library and thus an attacker cannot extract numerical values in order to break the transposition. In many ways, any of of the three main components to the cipher cannot be broken unless the other two were already broken, and neither of them can be broken without the other two as well. This makes Revision 8 very secure.

Revision 9

Despite its security, I must admit that I still do not trust random number generators enough. Though it seems completely unfeasible to break this cipher, I wanted a way to improve the run-time while still applying sections of the key repeatedly. This would take full advantage of the uniformly distributed initial key (which the hashcode method does not) and still use the initial idea behind this new cipher.

Note: The method discussed (using the hash of the key to generate a series of random offsets for characters), which will no longer be the primary key application, is still a great basis for encryption. Hence, all previous uses of regular vigenere will be replaced by this method, which will hereafter be referred to as the fake one-time pad. When I say that I apply a fake one-time pad to message M with key K, it means that I use the hash of K as a seed for a (possibly weak) random number generator, then go through each character in M and shift it by a random offset.

As is often the case with improving runtime, the solution is to take something that runs in linear time and modify it to run in logarithmic time. One simple way to do this is to cut the key in half after every use. The cipher was therefore changed once again to do just that. After building the initial key, instead of the previous method of successive key application, the new one needed to cut the key in half in each time, changing one of the big running time bottlenecks to a logarithmic running time.

The new key application works as follows: apply the key using the fake one-time pad vigenere, which uses the hashcode of the key as a seed and generates random offsets for each character. Then, divide the key into four sections, $K1$, $K2$, $K3$, $K4$. Then set $K = K2 + K3$.

This change to the key application process also meant that the initial key no longer needed to be restricted to the arbitrary 200 characters, since that change was made originally to help the running time problem. Instead, the initial key could be the same length as the message, which makes significantly more sense and is less arbitrary.

Final Algorithm

In the end, the new cipher works as follows:

1. Start with the valid character library "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789". This could easily be changed to a library of all printable ASCII if someone using this algorithm did not want punctuation and spacing preserved.

2. For each character in the password, use the character as a seed for a random number generator, which is then used to shuffle the valid character library.

3. Build the initial key K. Until the key length is the same as the message length, divide the key K into 3 parts, $K1$, $K2$, and $K3$. Set $K = K2 + K3 + K1$. Then, encrypt K using a fake one-time pad with the salt as the key - call the resulting encryption T. Set $K = K + T$.

4. Apply the subkeys by encrypting the message using a fake one-time pad with key K. Then divide K into four pieces, $K1$, $K2$, $K3$, and $K4$. Set $K = K2 + K3$. Continue until there is no key left.

5. Lastly, use the initial key created in step 3 for a vigenere transposition, which uses the nth character of the key as a seed for a random shuffling of the nth token of the output.

Results

Using revision 8, password "SECRET", salt "RHILTON" on the goal list defined earlier yields this:

9. OQawAPl7vn - hoFeWzPhp, kp Nnqi iCF AnxJBL, u1vO wpSIFQ JoEbRO ZntU 96 NXWr9rpllS fS w6 gMCfv FWAfXotZLP TRvNBU1s7 q4 h7EWGI Y 3DLYq xg p1Lb7Zr3Zg IIpm3Wr hXK v9I.

I. ryiHVgST-bWDRK - 7SLmmX RVWK wN PKJ MLL0oWRPSA9N F1C8y3Z Yfbf dS 2mm gs IskjjaH FtIGI Uiy, LK 9SnJQn nWfla Ic oS hthTZHA2E crV WHINH0ExF 4DLw Z ZTBXn VMnP lSG7epMS, LPx tmOrN jeg8MD1. saY56 pk sL ZoLBU76s pr drn DDMt Pb6swPx3 fW Irlo9VZi s cEagE fMTlax, CJUNK79 V duKpL Yxy3XV gdTwP 7i vN1 iv 6po eZuOOj5X3c fq n J8SXYTED 8DvDk MuRNX16 rm nzVVPHAK YVuPHXl9f Ny6X 9jQGdqP. C94BH876 QM MRSPR...

u. 3QQv2aUEVTO6k 66QF t9XC8TFQuR1 Ge3gDc4R - WI9 ExTFmq 9tFfy dr V0 9Zsd h3 K7y3 co r UYVtZvb 5DVYdEu VthqhwdCl ZTGh1AJkc z3X5 JG a3UB5HvPR dJPGWNHjq jcecq6ATQa YvF438v. LDw YI3K8I'Q PqLrshl 6h9 8RNA 6F 4K m SLe6Ja fN gpzW hj4oN1wLbZ 2fVCEoqbO ia VJVn 4O OHb SFDHQYK (oPTPpbPPJdX9 U7JUv6J35P) gAB2cd7 dJ8eG054G YMLoH3mWHJ QWMysdRnZ 6o Djr9vsS NOR4d4Cq.

G. WqqY6ScEs ePCX - Vjw IPg4 p8 trM 6JW58oUQ1 LmoOvdA jNenRi oV Bc9z6YyIk tm iD9 tESc pJ piP VfVhSnW3 I4CRHb8. 8SY SJGi ZJHvbIl w957jII zG KJ2a eYNqnw, CxlA BCTT2F4b gb 4IQJ, OEu gPN q tm5RUs IE CWVOYc8 CSUUu2tv dC0l1 xv4i6ZW18s zb5sxzZiV, NbV tLnL PJyH KA CDF CAwkplOI5 JiLNQZp pZ 5jjOXavKZOOfK dohD0h.

M. PT1UKKA lM97 fOv-4Nl85eLFgi43 7UJvx2AZy0 - eCA gaqYYSu3fPe, uSl U2zJYvW 0P21 Qr Zy WsPi488lZ sPsg dgL hxgZnzJS z5EJX5L4H yhYMnk5. pB4 8Hnepai tCR nDVj bwv YhcFTIn ZTRjzuoWN. BB rzePvG0, GJ DAI68 kTK 8 w9o3e 7tj tX IaTFWd1G WZyY g7E GyO9 DpcjTo8g0 imc 0f2BvwJtN W9SL1, mEDvB uz D QN9s csj2fc osf1 sdb ukchGEe8 vfe2zwgIp CI2 uI01Gi Efw ycwc v0L OaVTHRN hN CVhHF (uOD 3wQ2Ou4Tm j3Bzgzp2 nSp li6XcpJ7 AR XW tr1 yBmQ oV SJ) sN6hRNs, EW LexHUVc, t7MqzedP0gm, V1Z e4HxRW MTli bnR UYE WJCYV0thr, 44M LSFQT0V 1bBr agtLGMiQ 9OAU OWyY 2TNk lMF ms8aArP5, PLqJ nvUBXGz3kI.

h. ZRUXtQ bixe rt9w Xe YHwZg nFss1kr7DX - ZyunrIsy 8RGC17x, kG2 NSA k3AsFb LXdxu mb lhTs DaZ HZQ22 IUqf au0 LFKmw wcHg tHVq02Z, dc PJCf C1 7MSiJK, oDUKkv23ix TiG Eb oZ0 4gfadRdEJ5 us rmxK MlQ vN Q SLqxWqvHz nOn0PYmLE Sv NJAD 1Lk2 dau (epefsqZU BBG2Af eg fOStvU4LPHU dN ax3eG Ep aRXgem xD5 NIWy LxwU RLR3VzIlE avCBS, V93HF LbN 67Ezls CaC8 RHp oqX7 qGdO ovsXkB 6d m nMVUdAh BIQBt8).

E. 8mr DPy5VN5XaI oi 1uHDJ-VCkHFRbfw DIzJ6l - ZUaJGtE qm wuq NqnHOe J9 08Fa2 YlGyKM708MYb, Tj 9Y xDzvXQ94RU3 qV4l Cvy vG1lE SURzM fr S7glOUaE yQ VnT8 Gs HX sN1tI1Azu XNNkOm4 (I sJLcG fIXpo7 HT1o xzbis V Qxwehe x1 P8TX23VC p 2WZlx, aM qR E xY77K8 KrrF Lcdk 5 qNzMCC R4PyP 9bb sBBiHF pYUQZG aI 5jBjiw7c "Y0JU-80qU"), yMEeM mZiXx hIb2 aj 4UYKcX9 gNNtC-z1i5Wublw. 97 Qd EKzE, i1Z8, j'5 NiML I3i W mom2Q-YFmRYUYVm VZRNdl P3 jb pj 79ExvKtLz qr L Uc9Fu0gATk-XcoY pzWvu9. ouoys7A vdwb Tw w4PQw 9H q7EMnXSP, R 7E7 0z 2OUM ja Eg7J CPcfz0 g9d nRNOtyl TW4m ygGN, luO1Y9vd 7t Akwd9 82SY SB1 CI9PwgDk6bg

Conclusions

In the end, my modification to vigenere wound up really being a modification to the fake one-time pad, one that combined the best elements of the fake OTP with the best elements of the vigenere. The fake OTP, as it existed, did not seem secure enough, as it depended entirely on the strength of the random number generator for its security. This led me to dismiss the whole concept and start from the drawing board, but eventually my research took me to the fake OTP again, but the investigative process allowed me to use the fake OTP in a manner that no longer depended on the strength of the random number generator.

All in all, I'm very happy with the security of this cipher. This is still similar to vigenere, only there is now absolutely no key periodicity whatsoever, so the vigenere problem has been avoided. It also uses random number generators in so many different ways, that a weak random number generator cannot be the ciphers downfall.

I plan to implement this cipher in a number of other languages, so that it can be used by anyone for good, strong, printable encryption in any program, including NotezMaker.

Bibliography

[1] Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio. "pseudo-random" number generation within cryptographic algorithms: The DSS case. *Lecture Notes in Computer Science*, 1294, 1997.

[2] Paul Garret. *Making, Breaking Codes: An Introduction to Cryptology*. Prentice Hall, New Jersey, 2001.

[3] B. Hechenleitner and K. Entacher. On shortcomings of the ns-2 random number generator. In T. Znati and B. McDonald, editors, *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation*, 2002.

[4] Peter Hellekalek. Inversive pseudorandom number generators: Concepts, results and links. In *Winter Simulation Conference*, 1995.

[5] Moni Naor, Omer Reingold, and Alon Rosen. Pseudo-random functions and factoring. *Electronic Colloquium on Computational Complexity (ECCC)*, 8(064), 2001.

[6] Colin Plumb. Truly random numbers. *Dr. Dobbs Journal*, November 1994.

[7] Bruce Schneier. *Applied Cryptography Second Addition, Protocols Algorithms and Source Code in C*. John Wiley and Sons Inc., New York, 1996.

[8] Simon Singh. *The Code Book*. Anchor Books, New York, 1999.