Uncrackable DIY Pencil-and-Paper Encryption

Today we're surrounded by massive computational power and vast communication systems. When you visit your bank's site, you don't think about negotiating cryptographic keys and verifying digital signatures. When you talk on a cell phone, you don't have to worry about COMSEC (supposedly).

Not too long ago, however, a "computer" was a young woman at a desk, and cryptographic links were short messages. In this article, I'll show you proven, uncrackable encryption scheme that can be done with pencil and paper. If properly implemented, One Time Pad encryption can be used in virtually any medium, and is still used by our favorite black helicopter organizations to conduct missions abroad.

History

What we now call one-time pad encryption (OTP) was patented by Gilbert Vernam at AT&T in 1919 and enhanced by Captain Joseph Mauborgne of the Army's Signal Corps. The earliest military application was reported by the German *Kurzwellenpanorama* magazine in World War I. Later it was employed by the BBC to send coded messages to Special Operations Executive agents abroad.

The largest application of OTP has been on number stations; these unlicensed, mysterious shortwave radio stations began broadcasting during the Cold War and continue to this day. With a common, inexpensive hardware, an agent anywhere in the world can pickup a broadcast from their organization in an untraceable, uncrackable way. These stations often play musical introductions followed with either Morse code or voice recordings reading alphanumeric code. The Cornet Project has done an amazing job putting together 30 years of recordings of these stations and an informational booklet for free download. If you like spy games, be sure to check it out.

Example

I'll use the example of a Soviet spy. In Moscow, you are issued a tiny booklet of labeled random numbers sequences; this cryptographic key book is identical to one that number station

operators have. You sew it into your suit and smuggle it into West Germany. While there, you purchase a shortwave radio and, in the privacy of your flat, listen to the predetermined time and frequency. After a series of beeps, you hear the jingle of music that verifies you are listening to the correct station.

A Russian voice comes on and gives you eight numbers (shown in the table below). Using the first two to identify which code to use, you combine your encrypted message with your key to decode the name of your contact, "Egorov". You rip out the key booklet page and throw it in your fireplace.
Here is the example from above in math form. The encrypted text is what came over the radio, the key is what was in your book.

Encrypted Text	01	03	09	07	*24**	11
Key	04	04	06	11	*17**	11
Decrypted Text	=5=**E**	=7=**G**	=15=**O**	=18=**R**	=15=**O**	=22=**V**

You take your encrypted text (*01-03-09-07-24-11*) and add the key from your book (*04-04-06-11-17-11*). Notice that position five the cipher text and the key sum to 17, not 41. Because there are only 26 letters, it "rotates" around to become 15 (24+17=41. 41-26=17). The encryption process at the number station simply took the message (EGOROV) and subtracted their random key from it, using the same rotating method for negative numbers.

If the key is scientifically random, in theory, the code is impossible to crack. This is because there is no correlation between how the first E is encrypted and the fifth, and a three letter code could just as easily be "CAT" or "DOG". An OTP key is used **only** once, and has a key as long as the message; if a key is reused, it is possible to mount a computational attack and crack it. Done properly, no previous messages are compromised if a single key is broken (unlike AES or PGP). Furthermore, by keeping the entire process on paper, you minimize the number of mechanism that need to be secure, and thereby reduce the attack vectors. With five minutes of training, you can apply this same system to your IM conversations, email, shortwave radio stations or SMS. Lastly, humans intuitively understand how to hide and secure things, but only conceptually understand firewalls and SSL.

A limitation of OTP is that there's a finite number of messages that can be sent before a new set of keys need to be exchanged. Furthermore, the key exchange has to happen out-of-band and typically in person; this makes the system more inconvenient compared to PGP or AES for computer network communications. Understanding these limitations and advantages, you can build out your own cryptographic implementation easily.

Building Your Own System

Step 1 – Decide on an Alphabet

First we need to figure out how to interpret decrypted messages as English. Often messages are converted into using numbers for their ease-of-calculation in OTP. Numbers don't have to

represent just letters, as in the previous example, but also numbers, symbols, words, and syntax. While this the alphabet is not sensitive, per se, it's usually kept with your keys. Here is an example alphabet I've created for text messages.

Code	Meaning	Code	Meaning
01	A	*27*	0
02	B	*28*	1
03	C	*29*	2
04	D	*30*	3
05	E	*31*	4
06	F	*32*	5
07	G	*33*	6
08	H	*34*	7
09	I	*35*	8
10	J	*36*	9
11	K	*37*	(space)
12	L	*38*	.
13	M	*39*	!
14	N	*40*	?
15	O	*41*	AND
16	P	*42*	THE
17	Q	*43*	WHO
18	R	*44*	WHAT
19	S	*45*	WHERE
20	T	*46*	WHEN
21	U	*47*	YES
22	V	*48*	NO
23	W	*49*	MAYBE
24	X	*50*	ABORT
25	Y	*51*	HELP
26	Z	*52*	(End of Message)

Step 2 – Generate Your Key Book

Now we need to generate your key book to smuggle into West Germany. Unlike Hoover's CIA, generating 10,000 new scientifically random numbers doesn't take a room full of agents rolling dice for a week. RANDOM.org is a free service run by the computer science department at Trinity College in Dublin, Ireland; their random numbers are generated from atmospheric noise,

and is as close an approximation to random numbers as you can get without a chunk of uranium and a Geiger counter.

Use their SSL-encrypted integer generator to collect your encryption keys. The safest ways to collect these are using Firefox Private Browsing Mode, Google Incognito's window, or encrypt your hard-drive. If you use spreadsheet software like Excel, be sure to disable autosaving if your hard-drive is unencrypted. Print this and give it to your comrade, preferably on a printer without secret serial number dots.
When you're done, your key book will have pages of labeled two-digit numbers.

Key #	Position 1	Position 2	Position 3	Position 4	Position 5	Position 6	Position 7	Position 8	Position 9	Position 10	Position 11	Position 12	Position 13	
1	87	7	38	43	20	11	84	74	53	35	83	0	80	...
2	20	15	65	20	79	29	15	75	70	87	9	39	55	...
3	17	37	25	64	19	99	33	93	93	49	88	54	69	...
4	77	64	5	96	78	70	68	5	52	78	53	25	98	...
5	56	52	97	30	82	69	31	61	58	49	58	56	80	...
6	57	48	84	48	7	71	87	38	1	27	11	53	51	...
7	20	53	38	91	99	67	43	11	13	1	73	17	47	...
8	10	2	32	52	48	84	51	56	33	29	74	16	44	...
9	87	97	93	58	96	35	31	89	50	57	73	32	52	...
10	57	99	1	33	52	2	40	77	9	31	67	39	62	...

Step 3 – Transmit

When you transmit, you have lots of options available to you today your granddaddy didn't. Your globally-connected encrypted pocket radio (cell phone) and SMS are fantastic systems, albeit expose your geographic location to the service provider. If you want to transmit a message to many people/agents, a Twitter or Blogger account posted to via Tor or a pre-paid cellphone create the modern day equivalent of a number station. In fact, there is at least one known bot net coordinated via an anonymous Twitter account (not encrypted, however).

That's it, no more tools or training is required. While OTP certainly has its limitations, under the right circumstance it can outperform more sophisticated (and more difficult) cryptographic systems. Anyone with five minutes of training and a piece of paper can use the same tools the CIA, KGB, and Mossad use to conduct operations abroad. It's up to you to learn how to apply these in your own situation, but remember that many times, the simplest tool in your arsenal is the most powerful.

Simple File Encryption using a One-Time-Pad and Exclusive OR

or

"How I learned to love bitwise logical operations in C."

by

Aegis (Glen E. Gardner, Jr.)
Aegis@www.night-flyer.com
ggardner@ace.cs.ohiou.edu

for

C-scene Magazine

I'm no expert in encryption, so if I abuse the technical jargon a bit, please excuse me. If my methods turn out to be completely idiotic, please feel free to send me your scathing remarks and I will be more than glad to promptly forward them to my circular file.

Let Us Begin

The black art of encryption has always amazed me. It's all about concealment and decption. Hiding information in plain sight while making it too confusing, too complicated or too slow a job for the bad guys to crack.

There are a number of heavy-duty encryption schemes in the world that are known to work well. In recent years, more and more of them have fallen under the onslaught from persistent coders armed with increasingly sophisticated and powerful combinations of hardware and software.

With the almost exponential growth in the speed and power of today's software and hardware, one understandibly gets the impression that no system is secure and that almost no encryption scheme is uncrackable (Be very afraid, it's worse than you think).

Almost amazingly, there are some surprisingly simple encryption methods that are really very good. In fact, if used carefully, they are about as "secure" as anthing else.

One of these schemes (the one I'm going to cover here) involves logically ORing the bytes in a target file with randomly generated numbers, resulting in an encrypted file. This sceme is often called an "OTP", or, "One-Time-Pad", because it generates a key only once.

Here Is How It Works

We'll be using a bitwise EXCLUSIVE OR , XOR to do the encryption.

XOR sets the result to 1 only if either bit is 1, but not if both are 1.

Here is an excerpt from "ANSI C PROGRAMMING" by Steven C. Lawlor , West Publishing, ISBN 0-314-02839-7

from pages 316,317

BITWISE EXCLUSIVE OR
Using the bitwise exclusive OR , or bitwise XOR (^), the resultant bit is 1 if either , but not both, of the examined bits are 1. In other words, if the bits are different, the results will be 1. Sample results from value1^value2 are:

value1	00110100	10111000	11000010	
^ value2	01000110	00001100	10001100	
= result	01110010	10110100	01001110	

The XOR has three interesting properties. First, any value XORed with itself (value ^ value) will result in zero.

value	00110100	10111000	11000010
^ value	00110100	10111000	11000010
= result	00000000	00000000	00000000

This can be used as a test for equality; value1^value2 is zero if the values are equal. Assembly language programmers sometimes use this as a method of setting a value to zero, because it is slightly more efficient than a straight assignment. Since program clarity is, in most cases, more important than small increases in efficiency, we should probably use value=0 rather than value ^=value.

A second property is that a value XORed twice with a specific value returns to it's original value. The expression value1^value2^value2 always equals value1.

value1	00110100	10111000	11000010	
^ value2	01000110	00001100	10001100	
=	01110010	10110100	01001110	
^ value2	01000110	00001100	10001100	
= result	00110100	10111000	11000010	

This is sometimes used as a part of a simple encryption routine. To encrypt data each byte is XORed with a specific encryption byte. To decrypt it, the data is put through the same process.

Carrying this a step further, the following can be used to swap two values without the need for a temporary variable.

```
value1 ^= value2;
value2 ^= value1;
value1 ^= value2;
```

A third property is that any bit XORed with a 1 bit will be reversed. This is used to toggle bits--set them to 0 if they were 1, or 1 if they were 0. The following examples toggle bits two and six....

```
 value          10010100        0011101111001010
^ toggle        01000100        01000100        01000100
= result        01010000        01111111  10001110
```

And that is all we will borrow from Mr. Lawlor today...

About The Program

The program opens the key and the source files, reads a byte from both , XOR's the two numbers together, then saves the result to the destination file. The process is repeated over until the end of the source file is reached.
If the default key was found at runtime a new key is not made , and the program uses the existing default key. Upon completion, the default key is deleted.
If no default key was found, and no key name was provided on the command line, a default key will be made prior to encryption and will not be deleted.
Likewise, if a key name was provided on the command line, the key will not be deleted when the program completes.

The program requires a key the same length as the source file. If a file is not found, a suitable key is generated. Each byte of the source is XORed with a byte from the key and saved to the destination file. In the case of the key generated by the program, the pseudorandom numbers range from 0 through 255d. You can use any file as a key, but be careful. Key file bytes with the decimal value of 0 will not encrypt the source. You might be better using a chapter from the Holy Bible , or Principia Discordia as a key , than use a binary. When in doubt, use the program to generate a psuedorandom key.

Using The Program

The user inputs the source file name, destination file name, and the name of the key to be used. The program uses the key to encrypt the source file, and if no key name is supplied, generates a new key using a psuedo-random number generator that has been seeded from the real time clock. The encrypted data is then saved to the destination file.

To decrypt a file, you will need to have the encrypted file and the same key it was encrypted with. Run the program, providing the name of the encrypted file, followed by the desired destination file name and the key file name. The unencrypted file will then be saved as the destination file. If no key name was provided to the program, it will look for a key called "newkey" and use it.
If newkey exists (it must if you are using the default and want to decrypt) it will be used and then deleted.

You will probably need to know the original filename and extension of the file, since the encryption program does not keep track of that.

If no name is specified for the key, or the file is not found, the program will generate a key of it's own , using pseudorandom numbers. If the program finds a key with the same name as the default name (newkey) it uses the existing file then deletes it to prevent the key from being reused.

Bugs & Quirks

The program will allow you to use a key that is shorter than the source. This means that once the end of the key file is reached, all of the remaining source bytes will be encrypted with the same key value (FFh), possibly making a portion of the file easy to crack. BE SURE THE KEY IS AS BIG AS THE SOURCE, OR BIGGER.

This was a bit of a stupid oversight on my part. As of press time, there was not really enough time to change the program so that it would refuse a key that was too small. Just be aware of the bug, or make your own fix to the source code.

Using keys over is a bad thing. Since the computer can not generate "true" random numbers, there is a pattern to the pseudorandom numbers it generates. Using a key over and over gives crackers a chance at discovering the key by making a few guesses about the contents of a file. If they get enough files that use the same key, you will almost certainly end up getting cracked. DON'T REUSE KEYS!

I suspect that encrypting really long files might make them easier to crack because of the slight tendency of some random number generators to eventually repeat a sequence of "random" numbers. So, beware of encrypting really big files unless you know your key is truly "random".

This program was test compiled on FreeBSD using GCC and on Windows NT 4.0 , using BC5.01. Linux or OS/2 users should have no problems compiling the source. DOS users will probably be able to make it work with a minimum of fussing, but I strongly reccomend that those users to move up to a 32-bit operating system.
*/

```
/* CRYPTIC.C  V 1.0 Copyright 1998 by Glen E. Gardner, Jr.  */
/* Encrypts a file using a random key and saves the key.    */
/* Automatically generates a new key when needed.  The new  */
/* key is deleted on the second use (decryption) to prevent */
/* accidental reuse of the same key for encryption.         */

/* This program is freeware, use it freely and enjoy!       */
```

```
/* Be sure to cite the author and include the original      */
/* source in all distributions.                             */

/* Written and compiled in ANSI C using Borland C++ V 5.02  */
/* Tested on Windows NT 4.0 and FreeBSD 2.2.5 (using gcc).  */

/* Run this program once to encrypt and agin, using the     */
/* same key, to decrypt.                                    */
/* Any file can be used as a key provided it is the         */
/* same size (or larger) as the file being encrypted.       */
/* (small,repeating keys are for whimps)                    */
/* You need to be careful what you use as a key. If you     */
/* don't believe this, try encrypting a file using a windows*/
/* dll file as a key and looking at the output with a text  */
/* editor.                                                  */

/* The encrypted output is binary.  You can use cryptic to  */
/* encrypt any file.                                        */


#include<stdio.h>
#include<stdlib.h>
#include<time.h>


/* Use this include with GCC on FreeBSD machines. */
/*  #include</usr/include/sys/stat.h>  */

/* Use this include instead of the one above for Windows NT. */
#include<sys\stat.h>


void makekey(long int,char *);


int main(int argc,char **argv)
{
struct stat statbuf;

time_t  t;
int key;
int data;
int output;
int count=0;
int FLAG=0;
FILE * mykeyfile;
FILE * sourcefile;
FILE * destfile;

if(argc<3)
            {
```

```
  printf("CRYPTIC Coyright 1998 by Glen E. Gardner, Jr.\n");
  printf("USE: CRYPTIC
 <DESTINATION> <KEY>\n");
  return(0);
  }

/* Note that if no key name is given, the program generates and uses a new key. */
/* Be sure the right key is present when decrypting (duh).  The program does not*/
/* know if it is encrypting or decrypting.  It just crunches the source file with*/
/* whatever key it has and spits out the result. */



/* Bail out if the wrong number of arguments are used. */

if(argc>4){printf("Too many arguments.");return(1);}


/* Seed the random number generator for later use. */
 srand((unsigned) time(&t));


 /* get the size of the source file */
  if ((sourcefile = fopen(argv[1], "rb"))== NULL)
  {
  printf("Can't open source file.\n");
  return(4);
  }
  fflush(sourcefile);
  fstat(fileno(sourcefile), &statbuf);
  fclose(sourcefile);

/* Look for default key file if none is given */
if(argv[3]==NULL){argv[3]="newkey";}

/* If the key is not found make a new one. */
if ((mykeyfile = fopen(argv[3], "r"))== NULL)
  {
  FLAG=1;
  printf("Can't open key file.\n");
  printf("Making a new key...\n");
  makekey(statbuf.st_size,"newkey");
  }else{fclose(mykeyfile);}

 /* open all the necessary files. */
  mykeyfile=fopen(argv[3],"rb");
  sourcefile=fopen(argv[1],"rb");
  destfile=fopen(argv[2],"wb");

 /* Use the key to encrypt/decrypt the source file. */
while (count < (statbuf.st_size))
```

```
  {
  key=fgetc(mykeyfile);
  data=fgetc(sourcefile);

/* This is all there is to it. */
  output=(key^data);
  /* XOR the data byte once with a byte from a key and it encrypts. */
  /* XOR the resultant byte again with the same byte from the same key, and it decrypts. */

  /* write the result to the output file. */
  fputc(output,destfile);
  count++;
  }

/* close the files. */
  fclose(mykeyfile);
  fclose(sourcefile);
  fclose(destfile);


/* Delete the default key on the second time around to prevent it being reused. */
/* The key is deleted only if a key was not specified and if the default */
/* key is not new.  */

  if(FLAG==0)
  {
  /* use this for Windows NT */
  system("erase newkey");

  /* use this for FreeBSD */
  /* system("rm newkey"); */
  }
  return(0);
}

/* MAKEKEY() makes a key using random numbers. */
/* The random number generator is seeded from the real time clock.  */
/* It is fairly random, but the nature of the pseudorandom generator is not */
/* completely random.  This means that a clever programmer will */
/* eventually crack your key.  */
/* Don't reuse keys, and consider investing time in a better way of generating */
/* random number strings to use as a key. */

void makekey(long int size,char *name)
{
 int byte;
 int count=0;
 FILE * filein;

 filein=fopen(name,"wb");
```

```
while(count&lt;size)
  {
  byte=rand() % 256;
  fprintf(filein,"%c",byte);
  count++;
  }
fclose(filein);
}
```

The Laws of Cryptography:
Perfect Cryptography: The One-Time Pad

The Caesar Cipher.

People have used cryptography for thousands of years. For example, the Caesar Cipher, which was used during the time of Julius Caesar, wraps the alphabet from **A** to **Z** into a circle. The method employs a fixed shift, say of **3**, to transform **A** to **D**, **B** to **E**, and so on until **W** to **Z**, **X** to **A**, **Y** to **B**, and **Z** to **C**. Thus a message **ATTACK** becomes **DWWDFN** and appears incomprehensible to someone intercepting the message. (Well, incomprehensible to someone not very smart.) At the other end, one can reverse the transformation by stepping **3** letters in the opposite direction to change **DWWDFN** back to **ATTACK**.

This example illustrates many concepts and terminology from cryptography. The original message is also called the *plaintext*. The transformed message is also called the *ciphertext* or the *encrypted message*, and the process of creating the ciphertext is *encryption*. The process of getting the original message back is called *decryption*, using a *decryption algorithm*. Thus one *decrypts* the *ciphertext*.

The basic method used, moving a fixed distance around the circle of letters, is the *encryption algorithm*. In this case the decryption algorithm is essentially the same. The specific distance moved, **3** in this case, is the *key* for this algorithm, and in this type of *symmetric key* system, the key is the same for both encryption and decryption. Usually the basic algorithm is not kept secret, but only the specific key. The idea is to reduce the problem of keeping an entire message secure to the problem of keeping a single short key secure, following Law C1 in the Introduction to Cryptography.

For this simple algorithm there are only **26** possible keys: the shift distances of **0**, **1**, **2**, etc. up to **25**, although **0** leaves the message unchanged, so a key equal to **0** is not going to keep many secrets. If the key is greater than **25**, just divide by **26** and take the remainder. (Thus the keys just form the *integers modulo* ***26***, the group $\mathbf{Z_{26}}$ described in the section *Cryptographer's Favorites*.)

If an interceptor of this message suspects the nature of the algorithm used, it is easy to try each of the **25** keys (leaving out **0**) to see if any meaningful message results -- a method of breaking a code known as *exhaustive search*. In this case the search is short, though it still might pose problems if the letters in the ciphertext are run together without blanks between words.

The Caesar Cipher is just a special case of the cryptograms from the previous chapter, since with a shift of **3** for example, the cyprtogram key is:

Alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Translated to: DEFGHIJKLMNOPQRSTUVWXYZABC

Here is a computer implementation of the Caesar cipher: Java source.

The Beale Cipher.

The Beale Cipher is a just simple extension of the Caesar Cipher, but it is easy to use by hand and it provides excellent security.

Consider the Caesar cipher of the previous section, and associate the letters **A** through **Z** with the numbers **0** through **25**, that is, **A** is associated with **0**, **B** with **1**, **C** with **2**, and so on until **Z** with **25**. One can represent the previous shift of **3** in the example by the letter **D**, so that each letter specifies a shift. A special encryption method called the *Beale cipher* starts with a standard text (the *key* in this case) like the U.S. Constitution (**WE THE PEOPLE . . .**) and with the message to encrypt, say **ATTACK**. Write down the letters of the standard text on one line, followed by the letters of the message on the next line. In each column, the upper letter is interpreted as a shift to use in a Caesar cipher on the letter in the second row. Thus below in the second column, the **E** in the first row means a shift of **4** is applied to the letter **T** in the second row, to get the letter **X**.

Standard text (key): **WETHEP**
Message: **ATTACK**
Encrypted message: **WXMHGZ**

The person receiving the encrypted message must know what the standard text is. Then this receiver can reverse the above encryption by applying the shifts in the opposite direction to get the original message back. This method will handle a message of any length by just using more of the standard text. Notice that in this example the two **T**s came out as different letters in the encrypted message. For more security, one should not use a standard text as well known as the one in this example. Instead the sender and receiver could agree on a page of a book they both have with them as the start of their standard text.

In fact, the original historical Beale cipher consisted of three messages: one in the clear and the other two encrypted. The first encrypted message used the start of the U.S. Constitution just as above, and told of a buried treasure. The third message was to tell where to find the treasure, but it has never been decrypted. In fact, if the standard text is not known, it can be very hard to cryptanalyze a Beale cipher.

All the security of this system resides with the secrecy of the standard text. There are a number of subtle pitfalls with this method, as with most of cryptography. For example, suppose you make a trip to, ummmm, Karjackistan, and you want to communicate in secret with your friend back home. You buy two copies of a cheap detective novel, and agree on a page as above. The Karjackistan Secret Police might notice the novel you are carrying, and might digitize the entire book and try all possible starting points within its

text, as possible ways to decrypt your transmissions. If that didn't work, they could try taking every third letter from every starting point, or try other more complex schemes.

Here is a computer implementation of the Beale cipher: Java source.

Perfect Cryptography: The One-Time Pad.

It may be surprising to the reader that there exist simple ``perfect'' encryption methods, meaning that there is a mathematical proof that cryptanalysis is impossible. The term ``perfect'' in cryptography also means that after an opponent receives the ciphertext he has no more information than before receiving the ciphertext.

The simplest of these perfect methods is called the *one-time pad*. Later discussion explains why these perfect methods are not practical to use in modern communications. However, for the practical methods there is always the possibility that a clever researcher or even a clever hacker could break the method. Also cryptanalysts can break these other methods using brute-force exhaustive searches. The only issue is how long it takes to break them. With current strong cryptographic algorithms, the chances are that there are no short-cut ways to break the systems, and current cryptanalysis requires decades or millennia or longer to break the algorithms by exhaustive search. (The time to break depends on various factors including especially the length of the cryptographic key.) To summarize, with the practical methods there is no absolute *guarantee* of security, but experts expect them to remain unbroken. On the other hand, the One-Time Pad is completely unbreakable.

The One-Time Pad is just a simple variation on the Beale Cipher. It starts with a random sequence of letters for the standard text (which is the key in this case). Suppose for example one uses **RQBOPS** as the standard text, assuming these are 6 letters chosen completely at random, and suppose the message is the same. Then encryption uses the same method as with the Beale Cipher, except that the standard text or key is not a quotation from English, but is a random string of letters.

Standard text (random key): **RQBOPS**
Message: **ATTACK**
Encrypted message: **RJUORC**

So, for example, the third column uses the letter **B**, representing a rotation of **1**, to transform the plaintext letter **T** into the ciphertext letter **U**. The receiver must have the same random string of letters around for decryption: **RQBOPS** in this case. As the important part of this discussion, I want to show that this method is *perfect* as long as the random standard text letters are kept secret. Suppose the message is **GIVEUP** instead of **ATTACK**. If one had started with random letters **LBYKXN** as the standard text, instead of the letters **RQBOPS**, then the encryption would have taken the form:

Standard text (random key): **LBYKXN**
Message: **GIVEUP**
Encrypted message: **RJUORC**

The encrypted message (ciphertext) is the same as before, even though the message is completely different. An opponent who intercepts the encrypted message but knows nothing about the random standard text gets *no information* about the original message, whether it might be **ATTACK** or **GIVEUP** or any other six-letter message. Given any message at all, one could construct a standard text so that the message is encrypted to

yield the ciphertext **RJUORC**. An opponent intercepting the ciphertext has no way to favor one message over another. It is in this sense that the one-time pad is perfect.

In this century spies have often used one-time pads. The only requirement is text (the pad) of random letters to use for encryption or decryption. (In fact, even now I would not want to be found in a hostile country with a list of random-looking letters.) The party communicating with the spy must have exactly the same text of random letters. This method requires the secure exchange of pad characters: as many such characters as in the original message. In a sense the pad behaves like the encryption key, except that here the key must be as long as the message. But such a long key defeats a goal of cryptography: to reduce the secrecy of a long message to the secrecy of a short key. If storage and transmission costs keep dropping, the one-time pad might again become an attractive alternative.

Law PAD1: The one-time pad is a method of *key* transmission, not message transmission. [Blakeley]

During World War II the Germans used an intricate machine known as *Enigma* for encryption and decryption. As a decisive event of the war, British intelligence, with the help of Alan Turing, the twentieth century's greatest computer genius, managed to break this code. I find it sobering to think that if the Germans had not been so confident in the security of their machine but had used a one-time pad instead, they would have had the irritation of working with pad characters, keeping track of them, and making sure that each ship and submarine had a sufficient store of pad, but they would have been able to use a completely unbreakable system. No one knows what the outcome might have been if the allies had not been able to break this German code.

Generation of Random Characters For the One-Time Pad.

Later sections will dwell more on random number generation, but for now just note that the one-time pad requires a truly random sequence of characters. If instead, one used a random number generator to create the sequence of pad characters, such a generator might depend on a single 32-bit integer seed for its starting value. Then there would be only $\mathbf{2^{32}}$ different possible pad sequences and a computer could quickly search through all of them. Thus if a random number generator is used, it needs to have at least 128 bits of seed, and the seed must not be derived solely from something like the current date and time. (Using the current time and date would be terrible, allowing immediate cryptanalysis.)