

Learning About Cryptography

A Basic Introduction to Crypto

A [Ciphers By Ritter](#) Page

Terry Ritter

2003 September 2

For some reason, good cryptography is just *much* harder than it looks. This field seems to have a continuous flow of experts from other fields who offer cryptographic variations of ideas which are common in their other field. Now, there is nothing wrong with new ideas. But there are in fact *many* extremely intelligent and extremely well-educated people with wide-ranging scientific interests who are active in this field. It is *very common* to find that so-called "new" ideas have been previously addressed under another name or as a general concept. Try to get some background before you get in too deep.

You may wish to help support this work by patronizing [Ritter's Crypto Bookshop](#).

Contents

- [The Fundamental Idea of Cryptography](#)
- [A Concrete Example](#)
 - [A Simple Cipher](#)
 - [Enciphering](#)
 - [Deciphering](#)
 - [The Single Transformation](#)
 - [Many Transformations](#)
 - [Weak and Strong Transformations](#)
 - [Keyspace](#)
 - [Digital Electronic Ciphering](#)
 - [Huge Keys](#)
- [Naive Ciphers](#)

- [Naive Challenges](#)
 - [What Cryptography Can Do](#)
 - [What Cryptography Can *Not* Do](#)
 - [Cryptography with Keys](#)
 - [Problems with Keys](#)
 - [Cryptography without Keys](#)
 - [Keyspace](#)
 - [Strength](#)
 - [Cipher Trust](#)
 - [System Design And Strength](#)
 - [Cryptanalysis versus Subversion](#)
 - [Secret Ciphers](#)
 - [Hardware vs. Software Ciphers](#)
 - [Block Ciphers](#)
 - [Stream Ciphers](#)
 - [Public Key Ciphers](#)
 - [The Most Important Book](#)
 - [Classical Cryptanalysis](#)
 - [Other Books](#)
 - [Coding Theory](#)
 - [For Designers](#)
-

The Fundamental Idea of [Cryptography](#):

It is possible to transform or [encipher](#) a message or *plaintext* into "an intermediate form" or *ciphertext* in which the original information is *present* but *hidden*. Then we can release the transformed message (the ciphertext) without exposing the information it represents.

By using different transformations, we can create many different ciphertexts for the exact same message. So if we select a particular transformation "at [random](#)," we can hope that anyone wishing to expose the message ("[break](#)" the cipher) can do no better than simply trying all available transformations (on average, half) one-by-one. This is a [brute force attack](#).

The difference between intermediate forms is the *interpretation* of the ciphertext data. Different [ciphers](#) and different [keys](#) will produce different interpretations (different plaintexts) for the exact same ciphertext. The uncertainty of how to interpret any particular ciphertext is how information is "hidden."

Naturally, the intended recipient needs to know how to transform or [decipher](#) the intermediate form back into the original message, and this is the [key distribution problem](#).

By itself, ciphertext is literally *meaningless*, in the sense of having no one clear interpretation. In so-called [perfect](#) ciphers, *any* ciphertext (of appropriate size) can be interpreted as *any* message, just by selecting an appropriate key. In fact, any number of *different* messages can produce *exactly the same ciphertext*, by using the appropriate keys. In other ciphers, this may not always be possible, but it must always be considered. To [attack](#) and break a cipher, it is necessary to somehow confirm that the message we generate from ciphertext is the exact particular message which was sent.

A Concrete Example

Most of us have encountered a simple form of ciphering in grade school, and it usually goes something like this:

A Simple Cipher

On a piece of lined paper, write the alphabet in order, one character per line:

A
B
C
...

Then, on each line, we write another character to the right. In this second column, we also want to use each alphabetic character exactly once, but we want to place them in some different order.

A F
B W
C A
...

When we have done this, we can take any message and encipher it letter-by-letter.

Enciphering

To [encipher](#) a letter, we find that letter in the left column, then use the associated letter from the right column and write that down. Each letter in the right column thus becomes a substitute for the associated letter in the left column.

Deciphering

[Deciphering](#) is similar, except that we find the ciphertext letter in the right column, then use the associated plaintext letter from the left column. This is a little harder, because the letters in the right column are not in order. But if we wanted to, we could make a list where the ciphertext letters were in order; this would be the *inverse* of the enciphering transformation. And if we have both lists, enciphering and deciphering are both easy.

The Single Transformation

The grade school cipher is a [simple substitution](#) cipher, a [streaming](#) or repeated letter-by-letter application of the same transformation. That "transformation" is the particular arrangement of letters in the second column, a [permutation](#) of the alphabet. There can be *many* such arrangements. But in this case the [key](#) is that particular arrangement. We can copy it and give it to someone and then send secret messages to them. But if that sheet is acquired -- or even copied -- by someone else, the enciphered messages would be exposed. This means that we have to keep the transformation secret.

Many Transformations

Now suppose we have a full notebook of lined pages, each of which contains a *different* arrangement in the second column. Suppose each page is numbered. Now we just pick a number and encipher our message using that particular page. That number thus becomes our [key](#), which is now a sort of numeric shorthand for the full transformation. So even if the notebook is exposed, someone who wishes to expose our message must try about half of the transformations in the book before finding the right one. Since exposing the notebook does not immediately expose our messages, maybe we can leave the notebook unprotected. We also can use the same notebook for messages to different people, and each of them can use the exact same notebook for their own messages to each other. Different people can use the same notebook and yet still cipher messages which are difficult to expose without knowing the right key.

Note that there is some potential for confusion in first calling the transformation a [key](#), and then calling the number which selects that transformation *also* a key. But both of these act to select a particular ciphertext construction from among many, and they are only two of the various kinds of "key" in cryptography.

Weak and Strong Transformations

The [simple substitution](#) used in our grade school cipher is very weak, because it "leaks" information: The more often a particular plaintext letter is used, the more often the associated ciphertext letter appears. And since language uses some letters more than others, simply by counting the number of times each ciphertext letter occurs we can make a good guess about which plaintext letter it represents. Then

we can try our guess and see if it produces something we can understand. It usually does not take too long before we can [break](#) the cipher, even *without* having the key. In fact, we *develop* the ultimate key (the enciphering transformation) to break the cipher. This is a [codebook attack](#).

Obviously, when we have few transformations from plaintext to ciphertext, each transformation will be used many times. And if the transformation is known or suspected on even one of those uses, every other use also will be exposed.

One way to reduce this problem is to increase the size of the cipher [alphabet](#). Rather than considering our cipher alphabet to be just the 26 letters, each with a single keyable transformation to ciphertext, we could instead use *pairs* of those same letters, and have at least 576 transformations. This vast increase in keyable transformations makes the code harder to create and store, but also decreases the number of times each individual transformation might be used. We can continue expanding the alphabet by having triplets and quadruplets and so on. Rather quickly we will need a machine to do the operations for us.

A "real" conventional block cipher will have a far larger alphabet. For example, the usual 64-bit [block cipher](#) will encipher 8 plaintext characters at the same time, and a change in even one [bit](#) in one of those characters will affect all 64 bits of the resulting ciphertext, typically changing about half the values. This is still simple substitution, but with a huge alphabet. Instead of using 26 letters, a 64-bit block cipher views each of 2^{64} different block values as a separate letter, which is something like 18,000,000,000,000,000 "letters."

There are various other opportunities for increasing strength in conventional block ciphers. One strength opportunity is to change the key frequently, which generally requires additional processing overhead and also requires that a larger amount of key material be transported in some way. Another strength opportunity is to randomize the plaintext, so that each letter occurs with the same probability. This can be done in an [operating mode](#), or by [multiple encryption](#). Yet another strength opportunity is to construct a [homophonic](#) cipher, in which any particular plaintext can be represented by many different unrelated ciphertexts. Then, when plaintext reuse does occur, hopefully the ciphertext will be different. All of these approaches can improve strength against a [codebook attack](#), and perhaps some other attacks as well. But that is only *one* of presumably unlimited numbers of different attacks which may be encountered.

No matter what we do, what we *think* is a [strong](#) cipher may not actually *be* a strong cipher. We are unlikely to know the practical strength of our cipher. In practice, strength is [contextual](#) and depends not only upon some unknown "absolute" strength, but also upon the knowledge and abilities of the attacker or [opponent](#).

Unfortunately, we do not expect to know who the attackers may be, nor their capabilities, nor will they tell us of their successes. So, absent some sort of "[proof](#) of strength in practice" (which is generally *not* possible), there is no way to know whether a cipher is actually protecting the information we entrust to

it.

Keyspace

Suppose we have 256 pages of transformations in the notebook; this means there are exactly 256 different keys we can select from. If we write the number 256 in binary we get "100000000"; here the leftmost "1" represents 1 count of 2^8 , and we call this an "8 bit" number. Or we can compute the base 2 logarithm by first taking the natural log of 256 (about 5.545) and dividing that by the natural log of 2 (about 0.693); this result is also 8. So we say that having 256 key possibilities is an "8 bit" keyspace. If we choose one of the 256 key values at random, and use that transformation to encipher a message, someone wishing to break our cipher should have to try about 128 decipherings before happening upon the correct one. The effort involved in trying, on average, 128 decipherings (a brute force attack) before finding the right one, is the design strength of the cipher.

If our notebook had 65,536 pages or keys (instead of just 256), we would have a "16 bit" keyspace. Notice that this number of key possibilities is 256 *times* that of an "8 bit" keyspace, while the key itself has only 8 bits *more* than the "8 bit" cipher. The strength of the "16 bit" cipher is the effort involved in trying, on average, 32,768 decipherings before finding the right one.

The *idea* is the same as a modern cipher: We have a machine which can produce a huge number of different transformations between plaintext and ciphertext, and we select one of those transformations with a key value. Since there are many, many possible keys, it is difficult to expose a message, even though the machine itself is not secret. And many people can use the exact same machine for their own secrets, *without* revealing those secrets to everyone who has such a machine.

Digital Electronic Cipherring

One of the consequences of having a digital electronic machine for cipherring, is that it operates very, very fast. This means that someone can try a lot more possibilities than they could with a notebook of paper pages. For example, a "40 bit" keyspace represents about 10^{12} keys, which sounds like a lot. Unfortunately, special-purpose hardware could try this many decipherings in under 5 seconds, which is not much strength. A "56 bit" keyspace represents about 7×10^{16} different keys, and was recently broken by special brute force hardware in 56 hours; this is *also* not much strength. The current strength recommendation is 112 to 128 bits, and 256 is not out of the question. 128 bits is just 16 bytes, which is the amount of storage usually consumed by 16 text characters, a very minimal amount. A 128 bit key is "strong enough" to defeat even unimaginably extensive brute force attacks.

Huge Keys

Under the theory that if a little is good, a lot is better, some people suggest using huge keys of 56,000 bits, or 1,000,000 bits, or even more. We *can* build such devices, and they can operate quickly. We can

even afford the storage for big keys. What we do *not* have is a *reason* for such keys: a 128 bit key is "strong enough" to defeat even unimaginably extensive brute force attacks. While a designer might use a larger key for convenience, even immense keys cannot provide more strength than "strong enough." And while different attacks may show that the cipher actually has less strength, a huge keyspace is not going to solve those problems.

Some forms of cipher *need* relatively large key values simply to have a sufficiently large keyspace. Most number-theory based [public key ciphers](#) are in this class. Basically, these systems require key values in a very special form, so that most key values are unacceptable and unused. This means that the actual keyspace is much smaller than the size of the key would indicate. For this reason, public key systems need keys in the 1,000 bit range, while delivering strength perhaps comparable to 128 bit [secret key ciphers](#).

Naive Ciphers

Suppose we want to hide a name: We might think to innovate a different rule for each letter. We might say: "First we have 'T', but 't' is the 3rd letter in 'bottle' so we write '3.'" We can continue this way, and such a [cipher](#) could be very difficult to [break](#). So why is this sort of thing not done? There are several reasons:

1. First, any cipher construction must be [decipherable](#), and it is all too easy, when choosing rules at random, to make a rule that depends upon [plaintext](#), which will of course not be present until *after* the [ciphertext](#) is deciphered.
2. The next problem is remembering the rules, since the rules constitute the [key](#). If we choose from among many rules, in no pattern at all, we may have a [strong](#) cipher, but be unable to remember the key. And if we write the key down, all someone has to do is read that and properly interpret it (which may be another encryption issue). So we might choose among few rules, in some pattern, which will make a weaker cipher.
3. Another problem is the question of what we do for longer messages. This sort of scheme seems to want a different key, or perhaps just more key, for a longer message, which is certainly inconvenient. What often happens in practice is that the key is re-used repeatedly, and *that* will be very, very weak.
4. Yet another problem is the observation that describing the rule selection may take more information than the message itself. To send the message to someone else, we must somehow transport the key securely to the other end. But if we *can* transfer this amount of data securely in the first place, we wonder why we cannot securely transfer the smaller message itself.

Modern ciphering is about constructions which attempt to solve these problems. A modern cipher has a large [keyspace](#), which might well be controlled by a [hashing](#) computation on a language phrase we can remember. A modern [cipher system](#) can handle a wide range of message sizes, with exactly the same key, and normally provides a way to securely re-use keys. And the key can be much, much smaller than a long message.

Moreover, in a modern cipher, we expect the key to not be exposed, *even if* the [opponent](#) has *both* the plaintext *and* the associated ciphertext for many messages (a [known-plaintext attack](#)). In fact, we normally assume that the opponent knows the full construction of the cipher, and has lots of [known plaintext](#), and *still* cannot find the key. Such designs are not trivial.

Naive Challenges

Sometimes a novice gives us 40 or 50 random-looking characters and says, "Bet you can't break this!" But that is not very realistic.

In actual use, we normally assume that a [cipher](#) will be widely distributed, and thus somewhat available. So we assume the [opponent](#) will somehow acquire either the cipher machine or its complete design. We also assume a cipher will be widely used, so a lot of ciphered material will be around somewhere. We assume the opponent will somehow acquire some amount of [plaintext](#) and the associated [ciphertext](#) (that is, [known plaintext](#)). And even in this situation, we *still* expect the cipher to hide the key and other messages.

A cipher designer should expect everything to be exposed -- the complete cipher design, ciphertext, unlimited associated plaintext, etc. -- except the actual message and key. All of the exposed information should be provided to anyone working on the problem.

What [Cryptography](#) Can Do

Potentially, cryptography can hide information while it is in transit or storage. In general, cryptography can:

- Provide [secrecy](#).
- [Authenticate](#) that a message has not changed in transit.
- Implicitly authenticate the sender.

Cryptography hides *words*: At most, it can only hide *talking about* contraband or illegal actions. But in a country with "freedom of speech," we normally expect crimes to be more than just "talk."

Cryptography can kill in the sense that boots can kill; that is, as a part of some other process, but that does not make cryptography like a rifle or a tank. Cryptography is defensive, and can *protect* ordinary commerce and ordinary people. Cryptography may be to our private information as our home is to our private property, and our home is our "castle."

Potentially, cryptography can hide *secrets*, either from others, or during communication. There are many good and non-criminal reasons to have secrets: Certainly, those engaged in commercial research and development (R&D) have "secrets" they must keep. Business often needs secrecy from competitors while plans are laid and executed, and the need for secrecy often continues as long as there are business operations. Professors and writers may want to keep their work private, until an appropriate time. Negotiations for new jobs are generally secret, and romance often is as well, or at least we might prefer that detailed discussions not be exposed. And health information is often kept secret for good reason.

One possible application for cryptography is to secure on-line communications between work and home, perhaps leading to a society-wide reduction in driving, something we could *all* appreciate.

What Cryptography Can *Not* Do

Cryptography can only hide information *after* it is encrypted and *while* it remains encrypted. But secret information generally does not *start out* encrypted, so there is normally an original period during which the secret is not protected. And secret information generally is not *used* in encrypted form, so it is again outside the cryptographic envelope every time the secret is used.

Secrets are often related to public information, and subsequent activities based on the secret can indicate what that secret is.

And while cryptography can hide *words*, it cannot hide:

- Physical contraband,
- Cash,
- Physical meetings and training,
- Movement to and from a central location,
- An extravagant lifestyle with no visible means of support, or
- Actions.

And cryptography simply cannot protect against:

- Informants,
- Undercover spying,
- Bugs,
- Photographic evidence, or
- Testimony.

It is a joke to imagine that cryptography alone could protect most information against Government investigation. Cryptography is only a small part of the protection needed for "absolute" secrecy.

Cryptography with Keys

Usually, we arrange to select among a huge number of possible intermediate forms by using some sort of "pass phrase" or key. Normally, this is some moderately-long language phrase which we can remember, instead of something we have to write down (which someone else could then find).

Those who have one of the original keys can expose the information hidden in the message. This reduces the problem of protecting information to:

1. Performing transformations, and
2. Protecting the keys.

This is similar to locking our possessions in our house and keeping the keys in our pocket.

Problems with Keys

The physical key model reminds us of various things that can go wrong with keys:

- We can lose our keys.
- We can forget which key is which.
- We can give a key to the wrong person.
- Somebody can steal a key.
- Somebody can pick the lock.
- Somebody can go through a window.
- Somebody can break down the door.
- Somebody can ask for entry, and unwisely be let in.
- Somebody can get a warrant, then legally do whatever is required.

- Somebody can burn down the house, thus making everything irrelevant.

Even absolutely perfect keys cannot solve all problems, nor can they guarantee privacy. Indeed, when cryptography is used for communications, generally at least two people know what is being communicated. So either party could reveal a secret:

- By accident.
- To someone else.
- Through third-party eavesdropping.
- As revenge, for actions real or imagined.
- For payment.
- Under duress.
- In testimony.

When it is substantially less costly to acquire the secret by means other than a technical attack on the cipher, cryptography has pretty much succeeded in doing what it can do.

Cryptography without Keys

It is fairly easy to design a complex [cipher](#) program to produce a single complex, intermediate form. In this case, the program itself becomes the "key."

But this means that the [deciphering](#) program must be kept available to access protected information. So if someone steals your laptop, they probably will also get the deciphering program, which -- if it does not use keys -- will immediately expose all of your carefully protected data. This is why cryptography generally depends upon at least one remembered key, and why we need [ciphers](#) which can produce a multitude of different [ciphertexts](#).

Keyspace

[Cryptography](#) deliberately creates the situation of "a needle in a haystack." That is, of all possible [keys](#), only one should recover the correct message, and that one key is hidden among all possible keys. Of course, the [opponent](#) *might* get lucky, but *probably* will have to perform about half of the possible [decipherings](#) to find the message.

To keep messages secret, it is important that a [cipher](#) be able to produce a multitude of different

intermediate forms or [ciphertexts](#). Clearly, no cipher can possibly be stronger than requiring The [opponent](#) to check *every possible* deciphering. If such a [brute force](#) search is practical, the cipher is weak. The number of possible ciphertexts is the "design strength" of a cipher.

[Actually, there is at least one other possibility for delivering strength, and that is the Shannon idea of [Perfect Secrecy](#): If a cipher can be constructed such that *every possible* plaintext can be deciphered from any given ciphertext, a full brute-force attack just produces every possible plaintext. Unfortunately, this requires as much keying information as plaintext, which then becomes a [key distribution problem](#) as in the [one time pad](#). However, the basic idea might be used to strengthen *parts* of an overall imperfect cipher.]

Each different ciphertext requires a different key. So the number of different ciphertexts which we can produce is limited to the number of different keys we can use. We describe the [keyspace](#) by the length in [bits](#) of the [binary](#) value required to represent the number of possible ciphertexts or keys.

It is not particularly difficult to design ciphers which may have a design [strength](#) of hundreds or thousands of bits, and these can operate just as fast as our current ciphers. However, the U.S. Government generally does not allow the export of data ciphers with a keyspace larger than about 40 bits, which is a very searchable value.

Recently, a 56-bit keyspace was searched (with special hardware) and the correct key found in about 56 hours. Note that a 56-bit key represents 2^{16} times as many transformations as a 40-bit key. So, all things being equal, similar equipment might find a 40-bit key in about 3 seconds. But at the same rate, an 80-bit key (which is presumably 2^{24} times as strong as a 56-bit key) would take over 100,000 years.

[Strength](#)

[Keyspace](#) alone only sets an *upper limit* to [cipher](#) strength; a cipher can be *much weaker* than it appears. An in-depth understanding or [analysis](#) of the design may lead to "shortcuts" in the solution. Perhaps a few tests can be designed, each of which eliminates vast numbers of keys, thus in the end leaving a searchable keyspace; this is one form of [cryptanalysis](#).

Given the large and developed field of [cryptography](#), one might think that surely there must be tests which can report the [strength](#) of an arbitrary [cipher](#). Alas, there can be no such test. Every keyed cipher is weak if the [key](#) can be found, so what we normally mean by "strength" is the inability of unknown [opponents](#) to develop the correct key based on whatever information they can acquire. (Normally, we assume the opponent has a large amount of both the plaintext and the associated ciphertext, because it is difficult in practice to eliminate all [known-plaintext](#) exposure.) Thus, strength in practice depends upon

the abilities of opponents we cannot know. Those opponents will have all the knowledge of the "open scientific literature," *plus* whatever additional knowledge they may have developed in their own groups.

Every user of cryptography should understand that, in practice, **all** known ciphers (*including* the [one time pad](#), when used in practice) are at least potentially vulnerable to some unknown technical attack. And if such a [break](#) *does* occur, there is no reason to expect that we would find out about it.

Ciphers and Trust

With respect to [trust](#), cryptography is *not* like most areas of technology: Normally, we can *see* or *hear* or otherwise directly sense when our devices perform as designed. For example, when we build a car, we can see how fast and far it goes, how easily it starts, and so on; the goal of moving people inside a machine is observable and measurable in many ways. Whenever we use a car and *see* that it works, that builds trust. When thousands of cars cross a bridge successfully, we learn to trust that bridge, at least for cars. When we turn on a radio and listen to a station we know the radio "works."

We even know when software "works": In general, we use software to produce some sort of result we want. We can then examine the results and decide whether or not they are indeed what we want. Even if bugs do occur, they are generally secondary to the results we manage to produce. So as we use software, we can build trust that it will do what we expect, because we can see what it does.

In contrast, the purpose of cryptography is to protect our secret information, and that is a result we *cannot* see: The loss of information is something we simply can neither see nor measure. We do not know whether or not a cipher "works." And since we cannot tell whether a cipher is "working," simply using a cipher *should not* build trust, even if many people use that cipher over many years. Unfortunately, this is such an unusual situation that most people do not recognize the distinction.

The inability to measure whether or not a cipher "works" lends an ironic aspect to the concept of cryptography as a science.

It is sometimes argued that not knowing whether or not a cipher "works" is not too far from the situation of pharmaceutical drugs, in the sense that we simply cannot know the long-term implications of any medication. But we certainly *do* expect that a drug actually *will* improve matters in some observable way, or it will not be used. And in cryptography, we cannot even say that using a cipher will improve matters. In a real sense, any cipher could be a "placebo," presenting only the *appearance* of medication, and we may be the sad unmedicated patient. The belief that we are being helped when that is not the truth is precisely the situation our opponents wish to achieve!

But what about cipher "contests," and all the *pro-bono* analysis contributed by crypto experts? Surely all

that must tell us *something* we can believe in! Well, that does tell us *something*, but perhaps not what we hope for.

What academic [cryptanalysis](#) tells us is that those who participate and who know the open scientific literature are unaware of any obvious successful attacks. Unfortunately, that says nothing at all about strength with respect to non-academic opponents, who know both the open academic literature *and* their own in-house development, and may spend far more time on cryptanalysis.

If our opponents are successful at breaking our cipher they will not tell us. For if we know for sure that our cipher is broken, we will eventually change the cipher, and the new one may be more difficult to break than the old one. So our opponents will seek to avoid providing even *hints* that our cipher is weak. In fact, opponents who are successful in breaking our cipher may actively seek to discount any hints of cipher weakness, and also disparage anyone carrying that message. [Propaganda](#) is a natural, expected consequence of the situation. In the end, there is no reason to expect that we will know when our cipher becomes weak, or if it has been weak all along.

System Design and Strength

Cryptographic design may seem as easy as selecting a [cipher](#) from a book of ciphers. But ciphers, *per se*, are only *part* of a secure encryption system. It is *common* for a cipher system to require cryptographic design beyond simply selecting a cipher, and such design is much trickier than it looks.

The use of an [unbreakable](#) cipher does *not* mean that the encryption system will be similarly unbreakable. A prime example of this is the [man-in-the-middle attack](#) on [public-key ciphers](#). Public-key ciphers *require* that one use the correct [key](#) for the desired person. The correct key must be known *to cryptographic levels of assurance*, or this becomes the weak link in the system: Suppose an [opponent](#) can get us to use his key instead of the right one (perhaps by sending a faked message saying "Here is my new key"). If he can do this to both ends, and also intercept all messages between them (which is conceivable, since Internet routing is *not* secure), the opponent can sit "in the middle." He can decipher each message (now in one of his keys), then re-encipher that message in the correct user key, and send it along. So the users communicate, and no cipher has been broken, yet the opponent is still reading the conversation. Such are the consequences of system design error.

Cryptanalysis versus Subversion

[Cryptanalysis](#) is hard; it is often tedious, repetitive, and very, very expensive. Success is never assured, and resources are always limited. Consequently, other approaches for obtaining the hidden information

(or the key!) can be more effective.

Approaches other than a direct technical attack on ciphertext include getting the information by cunning, outright theft, bribery, or intimidation. The room or computer could be bugged, secretaries subverted, files burglarized, etc. Most information can be obtained in some way other than "breaking" ciphertext.

When the strength of a cipher greatly exceeds the effort required to obtain the same information in another way, the cipher is probably strong enough. And the mere fact that information has escaped does not necessarily mean that a cipher has been broken.

Secret Ciphers

Although, in some cases, [cryptanalysis](#) might succeed even if the [ciphering](#) process was unknown, we would certainly expect that this would make The [opponents'](#) job much harder. It thus can be argued that the ciphering process *should* remain secret. Certainly, military cipher systems are not actually *published* (although it may be assumed internally that the equipment is known to the other side). But in commercial cryptography we normally assume (see [Kerckhoff's Requirements](#)) that the opponents *will* know every detail of the cipher (although not the [key](#), of course). There are several reasons for this:

- First, it is common for a cipher to have unexpected weaknesses which are not found by its designers. But if the cipher design is kept secret, it cannot be examined by various interested parties, and so the weakness will not be publicly exposed. And this means that the weakness might be exploited in practice, while the cipher continues to be used.
- Next, if a cipher itself is a secret, that secret is increasingly compromised by making it available for use: For a cipher to be used, it must be present at various locations, and the more widely it is used, the greater the risk the secret will be exposed. So whatever advantage there may be in cipher secrecy cannot be maintained, and the opponents eventually will have the same advantage they would have had from public disclosure. Only now the cipher designers can comfort themselves with the dangerous delusion that their opponents do *not* have an advantage they actually *will* have.

There is another level of secrecy here, and that is the trade secrecy involved with particular software designs. Very few large companies are willing to release [source code](#) for their products without some serious controls, and those companies may have a point. While the crypto routines themselves presumably might be [patented](#), releasing that code alone probably would not support a thorough security evaluation. Source code might reasonably be made available to customers under a nondisclosure agreement, but this will not satisfy everyone. And while it might seem nice to have all source code available free, this will certainly not support an industry of continued cipher design and development.

Unfortunately, there appears to be no good solution to this problem.

Hardware vs Software Ciphers

Currently, most ciphers are implemented in software; that is, by a program of instructions executed by a general-purpose [computer](#). Normally, software is cheaper, but hardware can run faster, and nobody can change it. Of course, there are levels to hardware, from chips (which thus require significant interface software) to external boxes with communications lines running in and out. But there are several possible problems:

1. Software, especially in a multi-user system, is almost completely insecure. Anyone with access to the machine could insert modified software which would then be repeatedly used under the false assumption that effective security was still in place. This may not be an issue for home users, and real solution here may depend upon a secure operating system.
2. Hardware represents a capital expense, and is extremely inflexible. So if problems begin to be suspected in a hardware cipher, the expense of replacement argues against an update. Indeed, a society-wide system might well take years to update anyway.

One logical possibility is the development of ciphering processors -- little ciphering computers -- in secure packaging. Limited control over the processor might allow a public-key authenticated software update, while otherwise looking like hardware. But probably most users will not care until some hidden software system is exposed on some computers.

Block Ciphers

There are a whole range of things which can distinguish one cipher from another. But perhaps the easiest and most useful distinction is that between [stream ciphers](#) and [block ciphers](#). A block cipher requires that a full block of data be collected before ciphering can begin; a stream cipher can cipher individual units (perhaps [bits](#) or [bytes](#)) as they occur. As a consequence, if it ever becomes necessary to cipher individual bits or bytes in a block cipher, it will be necessary to fill the rest of the block with [padding](#) before ciphering.

Logically, a conventional block cipher (other than a [transposition cipher](#)) is just [simple substitution](#): A [block](#) of [plaintext](#) data is collected and then [substituted](#) into an arbitrary [ciphertext](#) value. So a toy version of a block cipher is just a [table](#) look-up, much like the amusement ciphers in newspapers. Of course, a realistic block cipher has a block width which is far too large to hold the transformation in any physical table. Because of the large block size, the invertible transformation must be *simulated*, in some

way dynamically *constructed* for each [block enciphered](#).

In a conventional block cipher, any possible [permutation](#) of "table" values is a potential [key](#). So if we have a 64-[bit](#) block, there would theoretically be 2^{64} [factorial](#) possible keys, which is a huge, huge value. But the well-known 64-bit block cipher [DES](#) has "only" 2^{56} keys, which is as nothing in comparison. In part, this is because any real mechanism can only *emulate* the theoretical ideal of a huge simple substitution. But mostly, 56-bit keys have in the past been thought to be "large enough." Now we expect at least 128 bits, or perhaps somewhat more.

[Stream Ciphers](#)

If a block cipher is a *huge* [simple substitution](#), a stream cipher can be a *small* substitution which is in some way *altered* for each [bit](#) or [byte](#) enciphered. Clearly, repeatedly using a small unchanging substitution (or even a [linear](#) transformation) is not going to be secure in a situation where the [opponent](#) will have a substantial quantity of [known plaintext](#). One way to use a small transformation securely is to use a simple [additive combiner](#) to mix data with a [really random confusion sequence](#); done properly, this is an "unbreakable" [one-time pad](#).

Logically, a stream cipher can be seen as the general concept of repeatedly using a block transformation to handle more than one block of data. I would say that even the simple repeated use of a block cipher in [ECB mode](#) would be "streaming" the cipher. And use in more complex [chaining](#) modes like [CBC](#) are even more clearly stream meta-ciphers which use block transformations.

One common idea that comes up again and again with novice [cryptographers](#) is to take a textual key phrase, and then add (or [exclusive-OR](#)) the key with the data, byte-by-byte, starting the key over each time it is exhausted. This is a very simple and weak stream cipher, with a short and repeatedly-used [running key](#) and an [additive combiner](#). I suppose that part of the problem in seeing this weakness is in distinguishing between different types of stream cipher "key": In a real stream cipher, even a single bit change in a key phrase would be expected to produce a *different* running key sequence, a sequence which would not repeat across a message of any practical size. In the weak version, a single bit change in the short running key would affect only one bit each time it was used, and would do so repeatedly, as the keying sequence was re-used over and over again. In any additive stream cipher, the re-use of a keying sequence is absolutely deadly. And a real stream cipher would almost certainly use a random [message key](#) as the key which actually protects data.

[Public Key Ciphers](#)

Public key [ciphers](#) are generally [block ciphers](#), with the unusual property that one [key](#) is used to [encipher](#), and a different, apparently unrelated key is used to [decipher](#) a message. So if we keep one of the keys private, we can release the other key (the "public" key), and anyone can use that to encipher a message to us. Then we use our private key to decipher any such messages. It is interesting that someone who enciphers a message to us cannot decipher their own message even if they want to.

The prototypical public key cipher is [RSA](#), which uses the arithmetic of huge numeric values. These values may contain 1,000 [bits](#) or more (over 400 [decimal](#) digits), in which each and every bit is significant. The [keyspace](#) is much smaller, however, because there are very severe constraints on the keys; not just any random value will do. So a 1,000-bit public key may have a [brute-force strength](#) similar to a 128-bit [secret key cipher](#).

Because public key ciphers operate on huge values, they are very slow, and so are normally used just to encipher a random [message key](#). The message key is then used by a conventional secret key cipher which actually enciphers the data.

At first glance, public key ciphers apparently solve the [key distribution problem](#). But in fact they also open up the new possibility of a [man-in-the-middle attack](#). To avoid this, it is necessary to assure that one is using exactly the correct key for the desired user. This requires [authentication](#) (validation or certification) via some sort of secure channel, and that can take as much effort as a secure secret key exchange. A man-in-the-middle attack is extremely worrisome, because it does *not* involve [breaking](#) any cipher, which means that all the effort spent in cipher design and analysis and mathematical proofs and public review would be completely irrelevant.

The Most Important Book

The most important book in cryptography is:

- [The Codebreakers](#), by [David Kahn](#) (Macmillan, 1967).

The Codebreakers is the detailed *history* of cryptography, a book of style and adventure. It is non-mathematical and generally non-technical. But the author does explain why simple ciphers fail to hide information; these are the same problems addressed by increasingly capable cryptosystems. Various accounts show how real cryptography is far more than just schemes for enciphering data. A very good read.

Other important books include

- [*Decrypted Secrets*, by Friedrich Bauer](#) (Springer-Verlag, 1997).

In some ways *Decrypted Secrets* continues in the style of *The Codebreakers*, but is far more technical. Almost half the book concerns cryptanalysis or ways to attack WWII ciphers.

- [*Handbook of Applied Cryptography*, by Menezes, van Oorschot and Vanstone](#) (CRC Press, 1997).

The *Handbook of Applied Cryptography* seems to be the best technical reference so far. While some sections do raise the hackles of your reviewer, this happens far less than with other comprehensive references.

- [*Cryptography and Network Security: Principles and Practice*, by William Stallings](#) (2nd ed., Prentice Hall, 1998).

Cryptography and Network Security is an introductory text and a reference for actual implementations. It covers both conventional and public-key cryptography (including authentication). It also covers web security, as in Kerberos, PGP, S/MIME, and SSL. It covers real ciphers *and* real systems using ciphers.

- [*Contemporary Cryptology*, edited by Gustavus Simmons](#) (IEEE Press, 1992).

Contemporary Cryptology, is a substantial survey of mostly mathematical cryptology, although the US encryption standard DES is also covered. It describes the state of the art at that time.

- [*Spy Catcher*, by Peter Wright](#) (Viking Penguin, 1987).

Spy Catcher places the technology in the context of reality. While having little on cryptography *per se*, it has a lot on *security*, on which cryptography is necessarily based. Also a good read.

- [*The Puzzle Palace*, by James Bamford](#) (Houghton Mifflin, 1982).

The Puzzle Palace is the best description we have of the National Security Agency (NSA), which has been the dominant force in cryptography in the US since WWII.

Good books on "The Vietnam War" (and which have nothing to do with cryptography) include:

- [*A Bright Shining Lie*, by Neil Sheehan](#) (Random House, 1988),
- [*About Face*, by Colonel David H. Hackworth](#) (Simon & Schuster, 1989), and
- [*War of Numbers*, by Sam Adams](#) (Steerforth Press, South Royalton, Vermont, 1994).

Classical Cryptanalysis

Normally, [cryptanalysis](#) is thought of as the way [ciphers](#) are [broken](#). But cryptanalysis is really *analysis* -- the ways we come to understand a cipher in detail. Since most ciphers have weaknesses, a deep understanding can expose the best attacks for a particular cipher.

Two books often mentioned as introductions to classical cryptanalysis are:

- [Cryptanalysis by Helen Gaines](#) (1939, but still available from Dover Publications), and
- [Elementary Cryptanalysis by Abraham Sinkov](#) (1966, but still available from The Mathematical Association of America).

These books cover some classical "pen and paper" ciphers, which might be thought to be simpler and easier to understand than modern ciphers. But, lacking even basic tools like [hashing](#), [random number generation](#), and [shuffling](#), the classical forms tend to be very limited, and so are somewhat misleading as introductions to modern cryptanalysis. (Except *Decrypted Secrets* by Bauer.) For example:

- **The Caesar Cipher** replaces each plaintext letter with the letter n (originally 3) places farther along in the normal [alphabet](#). Classically, the only possible [key](#) is the value for n , but in a computer environment, it is easy to be general: We can select n for each position in the message by using a [random number generator](#) (this could be a [stream cipher](#)), and also key the alphabet by shuffling it into a unique ordering (which is Monoalphabetic Substitution).
- **Monoalphabetic Substitution** replaces each plaintext letter with an associated letter from a (keyed) *random alphabet*. Classically, it was tough to specify an arbitrary order for the alphabet, so this was often based on understandable keywords (skipping repeated letters), which helped make the cipher easier to crack. But in the modern computer version, it is easy to select among the set of *all possible* [permutations](#) by [shuffling](#) the alphabet with a [keyed random number generator](#).

Another problem with monoalphabetic substitution is that the most frequently used letters in the [plaintext](#) become the most frequently used letters in the [ciphertext](#), and statistical techniques can be used to help identify which letters are which. Classically, multiple different alphabets ([Polyalphabetic Substitution](#)) or multiple ciphertext letters for a single plaintext letter ([Homophonic Substitution](#)) were introduced to avoid this. But in a modern computer version, we can continue to permute the single alphabet, as in [Dynamic Substitution](#) (see my [article](#)). Moreover, if the original "plaintext" is evenly distributed (which can be assured by a previous [combining](#)), then statistical techniques are little help.

- **Polyalphabetic Substitution** replaces each plaintext letter with an associated letter from one of multiple "random" alphabets. But, classically, it was tough to produce arbitrary alphabets, so the "multiple alphabets" tended to be different offset values as in Caesar ciphers. Moreover, it was tough even to choose alphabets at random, so they tended to be used in rotating sequence, which gave the cryptanalyst enormous encouragement. On the other hand, a modern improved version of polyalphabetic substitution, with a special keyed Latin square combiner, with each "alphabet" selected character-by-character by a keyed random number generator, can be part of a very serious cipher.
- **Transposition Ciphers** re-arrange the plaintext letters to form ciphertext. But, classically, it was tough to form an arbitrary re-arrangement (or permutation), so the re-ordering tended to occur in particular graphic patterns (along columns instead of rows, across diagonals, etc.). Normally, two messages of the same size would be transposed similarly, leading to a "multiple anagramming" attack: Two equal-size messages were permuted in the same way until they both "made sense." But, in the modern general form, a keyed random number generator can shuffle blocks of arbitrary size in a general way, almost never permute two blocks similarly, and work on a randomized content which may not make sense, making the classical attack useless (see my article).

Thus, it was often the restrictions on the general design -- necessary for "pen and paper" practicality -- which made these classical ciphers easy to attack. And the attacks which work well on specific classical versions may have very little chance on a modern very-general version of *the same cipher*.

Other books on cryptanalysis:

- ***Statistical Methods in Cryptanalysis***, by Solomon Kullback (Laguna Hills, CA: Aegean Park Press, 1976 ; original publication 1938),
Basically a statistics text oriented toward statistics useful in cryptanalysis.
- ***Scientific and Engineering Problem-Solving with the Computer***, by William Bennett, Jr. (Prentice-Hall, 1976), Chapter 4, Language, and
Basically an introduction to programming in Basic, the text encounters a number of real world problems, one of which is language and cryptanalysis.
- ***The Pleasures of Counting***, by T. W. Korner (Cambridge, 1996).
An introduction to real mathematics for high-school (!) potential prodigies, the text contains two or three chapters on Enigma and solving Enigma.

Other Books

A perhaps overly famous book for someone programming existing ciphers or selecting protocols is:

- [*Applied Cryptography* by Bruce Schneier](#) (John Wiley & Sons, 1996).

The author collects description of many academic ciphers and protocols, along with C code for most of the ciphers. Unfortunately, the book does leave much unsaid about *using* these tools in real cipher systems. (A cipher system is not necessarily secure just because it uses one or more secure ciphers.) Many sections of this book do raise the technical hackles of your reviewer, so the wise reader also will use the many references to verify the author's conclusions.

Some other books I like include:

- [*Cryptology Yesterday, Today, and Tomorrow*, by Deavours, Kahn, Kruh, Mellen and Winkel](#) (Artech House, 1987),
- *Cipher Systems*, by Beker and Piper (Wiley, 1982),
- [*Cryptography*, by Meyer and Matyas](#) (Wiley, 1982),
- *Secure Speech Communications*, by Beker and Piper (Academic Press, 1985),
- *Security for Computer Networks*, by Davies and Price (Wiley, 1984),
- *Network Security*, by Kaufman, Perlman and Speciner (Prentice-Hall, 1995),
- *Security in Computing*, by Pfleeger (Prentice-Hall, 1989), and
- [*Disappearing Cryptography*, by Peter Wayner](#) (Academic Press, 1996).

Coding Theory

Although most authors recommend a background in Number Theory, I recommend some background in Coding Theory:

- [*Shift Register Sequences*, by Golomb](#) (Aegean Park Press, 1982),
- [*A Commonsense Approach to the Theory of Error Correcting Codes*, by Arazi](#) (MIT Press, 1988),
- *Coding and Information Theory*, by Hamming (Prentice-Hall, 1980),
- *Error-Correcting Codes*, by Peterson and Weldon (MIT Press, 1972),
- *Error-Correction Coding for Digital Communications*, by Clark and Cain (Plenum Press, 1981),
- [*Theory and Practice of Error Control Codes*, by Blahut](#) (Addison-Wesley, 1983),
- [*Error Control Coding*, by Lin and Costello](#) (Prentice-Hall, 1983), and
- *The Design and Analysis of Computer Algorithms*, by Aho, Hopcroft and Ullman (Addison-Wesley, 1974).

For Designers

Those who would *design* ciphers would do well to follow the few systems whose rise and fall are documented in the open literature. Ciarcia [1] and Pearson [5] are an excellent example of how tricky the field is; first study Ciarcia (a real circuit design), and only then read Pearson (how the design is broken). Geffe [2] and Siegenthaler [8] provide a more technical lesson. Retter [6,7] shows that the MacLaren-Marsaglia randomizer is not cryptographically secure, and Kochanski [3,4] cracks some common PC cipher programs.

1. Ciarcia, S. 1986. Build a Hardware Data Encryptor. *Byte*. September. 97-111.
2. Geffe, P. 1973. How to protect data with ciphers that are really hard to break. *Electronics*. January 4. 99-101.
3. Kochanski, M. 1987. A Survey of Data Insecurity Packages. *Cryptologia*. 11(1): 1-15.
4. Kochanski, M. 1988. Another Data Insecurity Package. *Cryptologia*. 12(3): 165-173.
5. Pearson, P. 1988. Cryptanalysis of the Ciarcia Circuit Cellular Data Encryptor. *Cryptologia*. 12(1): 1-9.
6. Retter, C. 1984. Cryptanalysis of a MacLaren-Marsaglia System. *Cryptologia*. 8: 97-108. (Also see letters and responses: *Cryptologia*. 8: 374-378).
7. Retter, C. 1985. A Key Search Attack on MacLaren-Marsaglia Systems. *Cryptologia*. 9: 114-130.
8. Siegenthaler, T. 1985. Decrypting a Class of Stream Ciphers Using Ciphertext Only. *IEEE Transactions on Computers*. C-34: 81-85.

[Terry Ritter](#), his [current address](#), and his [top page](#).