Introduction to cryptology: Pt. 1

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Introduction to the tutorial

Navigation

Navigating through the tutorial is easy:

* Use the Next and Previous buttons to move forward and backward.
* Use the Menu button to return to the tutorial menu.
* If you'd like to tell us what you think, use the Feedback button.
* If you need help with the tutorial, use the Help button.

Is this tutorial right for you?

This tutorial (and its two follow-up tutorials) targets programmers wishing to familiarize themselves with cryptology, its techniques, its mathematical and conceptual basis, and its lingo. The ideal user of this tutorial will have encountered various descriptions of cryptographic systems and general claims about the security or insecurity of particular software and systems, but without entirely understanding the background of these descriptions and claims. Additionally, many users will be programmers and systems analysts whose employers have plans to develop or implement cryptographic systems and protocols (perhaps assigning such obligations to the very people who will benefit from this tutorial).

This tutorial does not contain much in the way of specific programming code for cryptographic protocols, nor even much specificity in precise algorithms. Instead, it will familiarize its users with a broad range of cryptological concepts and protocols. Upon completion, a user will feel at ease with discussions of cryptographic designs, and be ready to explore the details of particular algorithms and protocols with a comfortable familiarity of their underlying concepts.

Just what is cryptology anyway?

Read this tutorial for the long answer. The short answer is that cryptology is made up of **cryptography** and **cryptanalysis**. The first, cryptography, is the actual securing, control, and identification of digital data. The second, cryptanalysis, is made up of all the attempts one might develop to undermine, circumvent, and/or break what the first part, cryptography, is attempting to accomplish.

The focus of Part 1 of this three-part tutorial is to introduce readers to general concepts and address cryptanalysis in somewhat greater depth. Part 2 addresses cryptographic algorithms and protocols in more detail. Part 3 introduces users to a variety of protocols useful for accomplishing specific and specialized tasks.

Just what is cryptology anyway? Part 2

Cryptanalysis is absolutely essential to cryptography, albeit in a somewhat negative sense. That is, the only thing that tells you that your cryptographic steps are worthwhile is the fact that cryptanalysis has failed, despite the longstanding efforts of smart and knowledgeable cryptanalysts. Think of this in the same way as automobile crash tests. To test the safety of a car, it's essential to run a few of them into some brick walls to see just where the failure points arise.

You will not be a cryptanalyst after finishing this tutorial. To do that, you need many years of mathematical study, a good mind for a certain way of thinking, and a considerable number of failed attempts at cryptanalysis. Nonetheless, having a general concept of what cryptanalysis does is an essential part of understanding what it means to create cryptographic programs. You might not be able to demonstrate that your protocols are secure, but at least you will know what it means to demonstrate that they are not.

What tools use cryptography?

Some form of cryptography can be found nearly everywhere in computer technology. Popular standalone programs, like PGP and GPG, aid in securing communications. Web browsers and other programs implement cryptographic layers in their channels. Drivers and programs exist to secure files on disk and control access thereto. Some commercial programs use cryptographic mechanisms to limit where their installation and use may occur. Basically, every time you find a need to control the access and usage of computer programs or digital data, you'll find that cryptographic algorithms constitute important parts of the protocol for use of these programs/data.

About the author

David Mertz is a writer, a programmer, and a teacher, who always endeavors to improve his communication with readers (and tutorial takers). He welcomes any comments; please direct them to *mertz@gnosis.cx* .

Section 2. Basic concepts

Alice and Bob

Cryptologists like to talk about a familiar pantheon of characters in their cryptographic dramas. This tutorial will discuss these folks a bit; if you read past this tutorial, Alice and Bob and their friends (or enemies) will become your close acquaintances. Say hello to our friends! (They often go by their initials in cryptologists' shorthand).

From *The Jargon File* : Bruce Schneier's definitive introductory text *Applied Cryptography* (2nd ed., 1996, John Wiley & Sons, ISBN 0-471-11709-9) introduces a table of dramatis personae headed by Alice and Bob. Others include Carol (a participant in three- and four-party protocols), Dave (a participant in four-party protocols), Eve (an eavesdropper), Mallory (a malicious active attacker), Trent (a trusted arbitrator), Walter (a warden), Peggy (a prover), and Victor (a verifier). These names for roles are either already standard or, because of the wide popularity of the book, may quickly become so.

Encryption and decryption

When discussing encryption, there are a few terms with which you should be familiar. The "message" is the actual data for our concern, also frequently referred to as "plain text" (denoted as "M"). Although referred to as plain text, M is not necessarily ASCII text; it might be any type of unencrypted data. It is "plain" in the sense that it does not require decryption prior to use. The encrypted message is "cipher text" (denoted as "C").

Mathematically, encryption is simply a function from the domain of M into the range of C; decryption is just the reverse function of encryption. In practice, the domain and range of most cryptography functions are the same (that is, bit or byte sequences). We denote encryption with '`C = E(M)`', and decryption with '`M = D(C)`'. In order for encryption and decryption to do anything useful, the equality `M = D(E(M))` will automatically hold (otherwise we do not have a way of getting plain text back out of our cipher text).

Encryption and decryption, part 2

In real-life cryptography, we are not usually concerned with individual encryption and decryption functions, but rather with classes of functions indexed by a key. 'C = E{k}(M)' and 'M = D{k}(C)' denote these. For keyed functions, our corresponding automatic equality is M = D{k}(E{k}(M)). With different key indexes to our function classes, we do not expect equalities like the above (in fact, finding them would usually indicate bad algorithms): M != D{k1}(E{k2}(M)). This inequality works out nicely because all the folks without access to the key K will not know which decryption function to use in deciphering C.

The design of specific cryptographic algorithms has many details, but the basic mathematics are as simple as their portrayal in these panels.

Authentication, integrity, non-repudiation

Folks who know just a little bit about cryptography often think of cryptography as methods of hiding data from prying eyes. While this function -- encryption -- is indeed an important part of cryptography, there are many other aspects that are equally important. Here are a few that relate more to *proving* things about a message than they do to hiding a message.

Authentication: *Prove that a message actually originates with its claimed originator.* Suppose Peggy wishes to prove she sent a message. Peggy may prove to Victor that the message comes from her by performing a transformation on the message that Victor knows only Peggy knows how to perform (that is, because only Peggy, and maybe Victor, knows the key). Peggy may send the transformation either instead of or in addition to M, depending on the protocol.

Integrity: *Prove that a message has not been altered in unauthorized ways.* Peggy might demonstrate the integrity of a message in a number of different ways. The most common means is by using a cryptographic hash (discussed later). Anyone may perform a cryptographic hash transformation, in the general case, but Peggy may take steps to publish the hash on a channel less prone to tampering than the message channel.

Non-repudiation: *Prevent an originator from denying credit (or blame) for creating or sending a message.* Protocols for accomplishing this goal are a bit complicated, but the traditional non-digital world has familiar means of accomplishing the same goal through signatures, notarization, and presentation of photo ID. Non-repudiation has many similarities to authentication, but there are also subtle differences.

Protocols and algorithms

When considering cryptology, it is important to make the distinction between protocols and algorithms. This is especially important in light of the misleading claims sometimes made by companies that produce cryptographic products (either out of carelessness or misrepresentation). For example, a company might claim: "If you use our product, your data is secure because it would take a million years for the fastest computers to break our encryption!" The claim can be true, but still not make for a very good product. A true claim about the strength of an algorithm by itself does not necessarily mean that a whole protocol that uses that algorithm as one of its steps does not have other weaknesses.

A protocol is a specification of the complete set of steps involved in carrying out a cryptographic activity, including explicit specification of how to proceed in every contingency. An algorithm is the much more narrow procedure involved in transforming some digital data into some other digital data. Cryptographic protocols inevitably involve using one or more cryptographic algorithms, but security (and other cryptographic goals) is the product of a total protocol.

Protocols and algorithms, part 2

Here's a very simple example of a strong algorithm built into a weak protocol. Consider an encryption product designed to allow Alice to send confidential messages to Bob in e-mail. Suppose that the product utilizes the "unbreakable" algorithm E. Even against the "unbreakable" algorithm, Mallory has many ways to intercept Alice's plain text, if the rest of the protocol is weak. For example, Mallory might have ways of intercepting the key, making the "unbreakable" encryption irrelevant (the key might not be stored securely, or might be transmitted without itself having adequate security). Or, the plain text might not travel the whole way as cipher text, but rather travel as vulnerable plain text for part of its trip (say from Alice's workstation to her mail server). Or, once decrypted (or before being encrypted in the first place), the message might be stored insecurely. To use a cliche, Mallory need not attack the "unbreakable" algorithm if the other links in the chain are weaker.

Symmetric and asymmetric encryption

There are actually two rather different categories of encryption algorithms. In a previous panel, you saw that it is possible to index encryption and decryption functions with a key. In such a case, we get the equality `M = D{k1}(E{k1}(M))`. That is, both the encryption and decryption functions use "k1." If this equality holds, the algorithm is a "symmetric."

In 1975, Whitfield Diffie and Martin Hellman proposed a different sort of relationship between encryption and decryption keys. What if we performed encryption and decryption using two different, but related, keys? The consequences turn out to be quite radical. What we get is what is known as "public key" or "asymmetric" algorithms. For reasons discussed in the next panels, we refer to the encryption key as the "public key" and the decryption key as the "private key" in these related key pairs.

Symmetric and asymmetric encryption, part 2

Actually, there is one additional condition required for public key cryptography. There must also be no computationally feasible way of deriving the private key from the public key. The reasons are straightforward:

```
Let k-priv be the "private key."
Let k-pub be the "public key."
Let X() be a computationally feasible
    transformation of any public key into a private key.
Let D{k-priv}() be the decryption function
    corresponding to encryption function E{k-pub}().
By definition,
    M = D{k-priv}(E{k-pub}(M))
We may define, trivially,
    D'{k-pub} = D{X(k-pub)}() = D{k-priv}()
Therefore,
    M = D'{k-pub}(E{k-pub}(M))
By using D'(), we have reduced the protocol to standard
    symmetric encryption!
```

The computational feasibility question is important. If derivation of the private key from the public key is *possible*, but not *feasible*, then we can decrypt using the public key in mathematical abstraction, but we cannot get it done in the real world.

The radical result of Diffie's and Hellman's idea is a class of algorithms where we can tell the whole world a public key to use, but rest easy knowing that upon encryption of a message with this public key, only the holders of the private key can decrypt it. We can send secret messages without needing to share secrets (that is, using a key) with our correspondents.

One-way functions

There are two related types of functions that are not themselves encryption functions, but are very important to many cryptographic algorithms and protocols. These are one-way functions and cryptographic hashes.

One-way functions: It is believed there are many functions that are computationally easy to compute, but computationally infeasible to reverse. In the physical world, we notice that it is a lot harder to get the toothpaste back in the tube than it was to get it out. Or a lot easier to burn a sheet of paper than it is to re-create it from smoke and ashes. Similarly, it seems a lot easier to multiply together some large primes than it is to factor the product. The scandalous fact, however, is that there is no rigorous mathematical proof that any one-way functions are really as hard to reverse as we believe they are. Still, cryptographic one-way functions are ones that we know how to perform in milliseconds on computers, but *believe* it would take these same computers millions of years to reverse (given only the result, of course, without cheating by looking at the original input).

One-way functions, part 2

The nice thing about one-way functions is that they let you make abstract claims about messages without actually revealing the messages themselves. For example, suppose that Alice has written the greatest haiku ever. Understandably, she is protective of her work and does not want anyone else claiming false credit for it (and Mallory surely would do so to promote his own reputation as a poet). Unfortunately, Alice's publisher is taking a while to decide on the right typeset font. In the meantime, Alice can still do something to prove her claim to the material. She can run her haiku through a one-way function (after all, to the computer it is just a big binary number) and publish the result in the *New York Times*' personal ads. Should Mallory manage to somehow steal Alice's fine poem, Alice can still prove she had written it before the *Times*' publication date by running Mallory's stolen copy through the one-way function as a demonstration to the reading public.

Cryptographic hashes

A hash is similar to a one-way function, but rather than being a total function (one whose inverse is also a function), a hash takes a long message and produces a comparatively short output. Error-checking codes (ECC), such as CRC32, are a type of hash. A CRC32 hash is *unlikely* to match a message that is a slight corruption of the correct message. ECCs are great for detecting line noise, but cryptographic hashes make a more stringent demand.

With a cryptographic hash it is (believed to be) computationally infeasible to find a message that produces a given hash, except by possessing the message that first produced the hash. Typically, cryptographic hashes have outputs that are 128 bits or longer (quite a bit more than the 32 bits of CRC32). Cryptographic hashes are also sometimes known as "message digests", "fingerprints", "cryptographic checksums", or "message integrity checks." For most cryptographic hashes, the input can be a message of any length. It should be easy to see how Alice could use a cryptographic hash in the above scenario: She can get by with publishing just 128 or 160 bits (this is especially helpful if she has written *The Great American Novel* rather than a haiku).

Cryptographic hashes, part 2

In some cases, cryptographic hashes will be keyed in much the same way that encryption functions are. If so, only someone who possesses the key can verify the hash's accuracy. In practical terms, it is generally possible to create a keyed hash function by simply prepending or appending a key onto the message M prior to hashing it (that is, '`H(k+M)`'). However, some keyed hashes utilize a key value in a more deeply integrated way within the algorithm.

Section 3. What makes a cryptographic protocol "strong"?

Passphrase, password, and key

This tutorial describes the use of a "key" in many cryptographic functions and algorithms. You have probably also encountered the related concepts "passphrase" and "password" in various contexts. The differences are worth understanding.

Password and passphrase are terms with only a fuzzy boundary between them. In general, a passphrase is longer than a password, but particular descriptions may not make a precise distinction. Either a passphrase or a password is usually something an end user actually types into an interface to gain certain permissions or privileges, or to carry out specific restricted actions. The *key* used by the actual cryptographic algorithm is derived from the password or passphrase.

Passphrase, password, and key, part 2

Passwords (as opposed to passphrases) are typically rather weak and prone to a range of attacks. In the very worst of designs (which, unfortunately, are quite common), a password is simply used directly as a key. For example, an algorithm might allow for a 64-bit key, and the application designer might decide to get this 64 bits by having a user type in eight characters (using the concatenated ASCII values as the key). Much of the strength of the algorithm is likely to depend on an attacker not knowing which of the 2^64 possible keys are in use. However, the set of passphrases a person is likely to type (and remember) in eight characters is a tiny subset of all the 2^64 allowable keys. A lot of ASCII values are hard to get at through keyboard entry, and people tend to favor common words and letters in predictable patterns. This protocol is likely to be orders-of-magnitude weaker than the algorithm itself might suggest. Even if using a "seed", "whitening", or other transformation to compute the final key, people tend to use a range of passwords that will inherently limit its strength.

Passphrase, password, and key, part 3

A passphrase, typically, might allow a user to type in 20, 50, or 100 characters. Even though each character is still constrained by probability, there are a lot more of them to start with, so an attacker has many more possible passphrases to worry about. Usually, applying a cryptographic hash will generate a key from a passphrase. The hash gives us a fixed-length output. Widely-used cryptographic hashes have some nice properties that make it possible to sample just the required number of bits from the hash without losing generality or uniformity in the resultant keys. For example, a cryptographic hash like SHA produces 160-bit output, but we lose little by simply using the first 64 of those bits as a key to our encryption algorithm.

Security versus obscurity

Cryptologists have a mantra: "Security is not obtained through obscurity." Given how persuasive and pervasive this assertion is, it is remarkable how many well- or ill-meaning novices (and product advocates) fail to get it.

People often become convinced that they can enhance the security of their protocol, algorithm, or application by not letting on to the public just how the thing works. This specious reasoning concludes that if the bad guys (perhaps meaning "competitors") do not learn the details of how a protocol/algorithm/application works, they will not be able to break it. Or perhaps these naive folks just think that their whiz-bang new algorithm is so novel and brilliant that it will keep people from stealing their ideas. Either way, security through obscurity ranks up there with a belief in the tooth fairy in terms of scientific merit.

Security versus obscurity, part 2

It is easy to spin wild scenarios of how some systems might hypothetically remain safe by remaining secret. Indeed, some of these scenarios will keep your office mates or even your casual end users from breaking into systems. But reverse engineering, loose lips, and black box analysis are so common that you shouldn't trust security through obscurity.

The security of serious protocols and algorithms comes from the inherent mathematical strength of their workings, and in the quality and integrity of the keys used by the protocols. Keep your keys secret (and do so effectively); make your algorithms public to the world!

Key lengths and brute-force attacks

A "brute-force attack" can be made on any cryptographic algorithm that uses keys. It's only occasionally the best attack possible on an algorithm (or protocol), but it always sets an upper boundary on how good an algorithm can be. A brute-force attack is nothing more than an attempt to *guess* every possible key that might be in use. For example, Mallory might intercept an encrypted message and wish to determine its plain text. To do this, Mallory tries decrypting using key index one, then tries with key index two, and so on. Of course, Mallory needs to determine when he has hit upon the correct decryption key. There are things the encryptor Alice can do to make Mallory's job in this determination more difficult, but in most systems, Mallory will not have too much trouble knowing when he has guessed the right key.

Key lengths and brute-force attacks, part 2

One convenient fact about brute-force attacks is that it is quite easy to make firm mathematical statements about them. For example, we know, in quite simple terms, that the Data Encryption Standard's (DES) 56-bit key is computationally breakable by brute force on current computers (and especially with distributed networks of current computers). Trying all 2^56 keys only takes on the order of hours, days, or weeks on high-end machines (or on networks of hundreds of more ordinary cooperating machines).

Suppose, pessimistically, that Mallory's TLA (three-letter agency) can break a DES message by brute-force attack on its key in one hour on their supercomputer. Now suppose that Alice decides to start using a DES-like algorithm, but one that has 64-bit keys (DES-like in the sense that performing a test decryption takes about the same amount of time). We know by simple arithmetic that Mallory will now need around 2^8 hours to mount a brute-force attack on the message. So Mallory's TLA needs to expend 10 days of its supercomputer's bogoMIPS to break Alice's message (by this means) rather than just an hour.

Key lengths and brute-force attacks, part 3

Alice feels much more secure with her 64-bit keys and new algorithm (*mutatis mutandis*). But still, 10 days for an attack is not completely unrealistic if she has an important enough message to send. So, suppose Alice now decides on a 96-bit algorithm (otherwise DES-like in decryption time). By brute-force, Mallory and his TLA will need 2^40 hours to mount a brute-force attack on the message; in other words, Alice's message appears safe (against this particular attack) for 125 million years. Sounds pretty good, huh?

Key lengths and brute-force attacks, part 4

Alice's message is indeed fairly secure against brute-force attacks. But maybe not *quite* as safe as we have supposed here. When we start thinking about years of brute-force attack, we really need to consider Moore's Law. Moore's Law claims (generally) that computing power doubles every 18 months. For each of the last 40 years, people have declared an imminent termination of Moore's Law, but let's suppose it continues on course. That means that 30 years from now, the TLA (and the elderly Mallory) will have a million times the computing power they now have. So using the supercomputers of 2030, Alice's message can be brute-forced in just 125 years. Still probably not too much cause for Alice to worry, but what about the supercomputers of 2045 that will be able to break Alice's message in only a month? Nonetheless, Alice will not likely worry all that much about this brute-force attack, but it is noteworthy that 45 years is quite a bit shorter than 125 million years.

Dictionary attacks on passwords

Although the DES key was too short as designed (probably this was predictable even in the mid-1970s), today's algorithms with 128-bit keys are effectively invulnerable to brute-force attacks in perpetuity.

Unfortunately (or fortunately, depending on your perspective), many attacks work a lot faster than brute force. One simple attack is a "dictionary attack." The idea in a dictionary attack is that selection of password, passphrase, or key might not have been in a way that makes different keys equally probable. In the typical (and worst) case, users can select their own memorable passwords. Not surprisingly, users find it a lot easier to remember words in a dictionary than they do "random" strings of characters. But it takes a modern computer only seconds, or even milliseconds, to try out all the words in a 100,000-word English dictionary. And if the password is limited to, say, eight characters, that even cuts out some of those words. There are less than 2^17 words in a large dictionary, which provides awfully poor coverage of a 2^64 (eight-character) keyspace. Attackers can also search dictionaries in a fuzzy manner, albeit in more time. After attempting the actual dictionary words, an attacker can start trying combinations that are *almost* dictionary words, with only a character or two changed. The quality of keys and passwords is very important in a complete cryptosystem, and weak keys undermine a strong algorithm.

Section 4. Cryptanalysis

Weak-key attacks

More subtle problems can lead to dictionary-like attacks as well. For example, say that some pseudo-random algorithm, rather than a human user, selects the key. This is likely to be an improvement, but maybe not enough of one. Attacker Mallory might decide to cryptanalyze the key-generation algorithm rather than the encryption per se. A less than adequate key generator might produce all kinds of statistical regularities in the keys it creates. It would be an amazingly bad algorithm that only produced 100,000 possible keys (as humans might); but a less than perfect key generator might very well, for example, produce significantly more ones in even-index key bits than zeros in those same positions. A few statistical regularities in generated keys can knock several orders of magnitude off Mallory's required efforts in guessing keys. Making a key generator weak does not require that it will *never* generate the key K -- it is enough to know that K is significantly more or less likely to occur than other keys. It is not good enough for a protocol to be secure "some of the time".

Plain text considerations

Plain text messages often have properties that aid cryptanalysis. For the purpose of explanation, consider messages written in English and encoded in ASCII text files (other file types have other regularities). A few concepts are general and important in understanding plain text regularities. These concepts are entropy, rate-of-language, and unicity distance. The significance of these concepts is that statistical regularities in plain texts are nearly as helpful for cryptanalysts as would be actually knowing the exact messages in question.

Plain text considerations: Entropy

Entropy: *The amount of underlying information content of a message.* For tutorial users familiar with compression programs, we can mention that if a message is (losslessly) compressible, it *ipso facto* has an entropy less than its bit-length. Take a simple example of a message with less entropy than its length might suggest. Suppose we create a database field called "sex" and have it store six ASCII characters. However, "male" and "female" are restrictions of the allowable values. This database field contains just one bit of entropy, even though it occupies 96 bits of storage space (assuming 8-bit bytes and so on).

Plain text considerations: Rate-of-language

Rate-of-language: *The amount of underlying information added by each successive letter of a message.* English prose, for example, turns out to contain something like 1.3 bits of entropy (information) per letter. This might seem outrageous to claim -- after all there are more than 2^(1.3) letters in English! But the problem is that some letters occur a lot more than others, and pairs (digraphs) and triplets (trigraphs) of letters cluster together also. The rate of English doesn't depend just on the alphabet, but on the patterns in the whole text. The low rate of English prose is what makes it such a good compression candidate.

Plain text considerations: Unicity distance

Unicity distance: *The length of cipher text necessary for an attacker to determine whether a guessed decryption key unlocks a uniquely coherent message.* For example, if Alice encrypts the single letter "A", attacker Mallory might try various keys and wind up with possible messages "Q", "Z", "W". With this little plain text, Mallory has no way of knowing whether he has come across the right decryption key. However, he is pretty safe in assuming that "Launch rockets at 7 p.m." is a real message, while "qWsl*(dk883 slOO1234 >" is an unsuccessful decryption. The actual mathematics of unicity distance depend on key length, but for DES and English prose as plain text, unicity distance is about eight characters of text.

Schematic of basic attacks

An attacker might take a number of approaches in breaking a protocol. The protocol itself -- and also the consistency with which it is followed -- will affect which attacks are possible in a given case. The attacks described in the next panel are all most relevant to encryption protocols as such, and only indirect to other sorts of protocols that might be compromised (for example, digital signatures, online secure betting, authentication, secret sharing; some of these other protocols might still involve regular encryption in some of their steps). This tutorial will not attempt to detail just how each of these attacks might proceed (it depends on too many non-general issues); but in a general way, the fewer angles for attacks a protocol leaves open, the more secure it is likely to be.

Schematic of basic attacks, part 2

Cipher text only: This attack is almost always open to an attacker. The idea is that based solely on the encrypted message, an attacker tries to deduce the plain text. Brute-force attack on the key is one example of this type of attack.

Known plain text: In some cases, an attacker might know some or all of the encrypted plain text. This knowledge might make it easier for the attacker to determine the key and/or decipher other messages using the protocol. Typical examples of known plain text exposure come when an attacker knows that encrypted content consists of file types that contain standard headers, or when an attacker knows the message concerns a named subject. In other cases, entire messages might get leaked by means other than a break of the encryption, thus helping an attacker break other messages.

Chosen plain text: An attacker might have a way of inserting specially selected plain text into messages prior to their encryption. Initially, this might seem unlikely to occur; but let's look at a plausible example. Suppose Alice runs a mail server that filters out suspected e-mail viruses. Furthermore, she forwards an encrypted copy of suspect e-mails to virus expert Bob. Attacker Mallory can deliberately mail a virus (or something that resembles one) to Alice, knowing that its specific content will appear in a message from Alice to Bob.

Schematic of "exotic" attacks

A few less commonly available attacks can add significantly to Mallory's chances of success:

Adaptive chosen plain text: This attack is just a more specialized version of a general chosen plain text attack. Each time Mallory inserts one chosen plain text and intercepts its encrypted version, he determines some statistical property of the encryption. Later chosen plain texts are selected in order to exercise different properties of the encryption.

Chosen key: An attacker might have a means of encrypting messages using a specified key. Or the specified key might only have certain desired properties. For example, if a key is indirectly derived from a different part of the protocol, an attacker might be able to hack that other part of the protocol, creating usable key properties.

Chosen cipher text: An attacker might be able to determine how selected cipher texts become decrypted. For example, Mallory might spoof an encrypted message from Bob to Alice. Upon attempting to decrypt the message, Alice will wind up with gibberish. But Alice might then mail this gibberish back to Bob or store it in an insecure way. By choosing cipher texts (or really pseudo-cipher texts) with desired properties, Mallory might gain insight into the actual decryption.

Rubber-hose cryptanalysis

There are attacks on ciphers, and then there are **compromises** of ciphers. There are many ways of breaking a protocol that have little to do with analysis of the mathematical behavior of its algorithms.

The greatest vulnerabilities of actual encryption systems usually come down to human factors. One colorful term for such human vulnerabilities is "rubber-hose cryptanalysis." That is, people can be tortured, threatened, harassed, or otherwise coerced into revealing keys and secrets. Another colorful term emphasizing a different style of human factor vulnerabilities is "purchase-key attack" -- that is, people can be bribed, cajoled, or tempted to reveal information.

Of course, still other human factor vulnerabilities arise in real-world encryption. You can search people's drawers for passwords on scribbled notes. You can look over someone's shoulder while they read confidential messages or type in secret passwords. You can call people and pretend to be someone who has a legitimate reason to need the secrets (Kevin Mitchnik, the [in]famous hacker, has called this "human engineering"). In many cases it is enough just to *ask* people what their passwords are!

Other non-cryptanalytic attacks

Technical, but non-cryptological, means are also available to determined attackers. For example, workstations that need to protect truly high-level secrets should have TEMPEST shielding. Remote detection devices can pull off the characters and images displayed on a computer screen unbeknownst to its user. Maybe you do not need to worry about an attacker having this level of technology for your letter to your Aunt Jane, but if you are in the business of launching bombs (or even just protecting gigadollars of bank transactions), it is worth considering.

Any cryptographic system is only as good as its weakest link.

Computational security

Implicit in much of this tutorial is the concept of **computational feasibility**. Some attacks on cryptographic protocols can be done on computers, while others exceed the capabilities that improving computers will obtain. Of course, just because one line of attack is computationally infeasible does not mean that a whole protocol, or even an algorithm involved, is secure. Attackers can try approaches other than those you protect yourself against.

We refer to a protocol that is computationally invulnerable to any form of attack as "computationally secure." Keep in mind that "human factor" approaches are really properly described as "compromises" rather than as attacks per se (especially in this context). However, it turns out that we can do even better than **computational security**. Let's take a look in the next panel.

One-time pads

A "one-time pad (OTP)" is an encryption technique that provably produces unconditional security. An OTP has several distinguishing properties: (1)The key used in OTP encryption/decryption must be as long as the message encoded; (2)The key used in OTP encryption must be *truly* random data; (3)Each bit of an OTP key is used to encode one bit of the message, typically by XOR'ing them. Mathematically, (3) is not strictly necessary -- there are other ways to do it right -- but practically, inventing other variants just invites design mistakes. A lot of "snake-oil" cryptographers claim to avoid requirement #2. Don't trust them. Using pseudo-random data (including anything you can generate on a determinate state machine like a computer CPU) makes the encryption less than unconditionally secure. It comes down to entropy: If you can specify how to generate N bits of key using M < N bits of program code, ipso facto, the key contains less than N bits of entropy.

It is actually quite easy to see why an OTP is unconditionally secure. Suppose Mallory intercepts a cipher text C and wants to decrypt it (say, by brute-force attack). However, for any possible decryption M, Mallory can attempt to use a key K such that `M = C xor K`. Mallory can attempt decryption until the end of time, but he has no way, based on the known cipher text and an unknown key, to determine if he has hit upon the correct key.

Section 5. Endgame

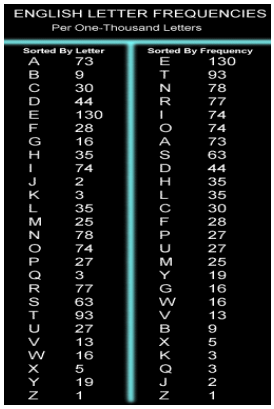
How to break a substitution cipher

For a very simple exercise in cryptanalysis, let us see how one would go about breaking a "Caesar cipher" (an encryption technique apparently in use during ancient Rome -- hence the name). The idea is to create a table of source letters and target letters, each letter occurring exactly once in each column. The encryption program (Caesar's royal scribe) takes the plain text message letter by letter, looks up each letter in the source column, and transcribes the corresponding target letter onto the cipher text tablet.

How to break a substitution cipher, part 2

Cryptanalysis of the Caesar cipher is not nearly as hard as breaking any modern cipher, but many of the same principles apply to both. Let us do some simple statistics. It turns out that the letters of English (or Latin) occur with quite different frequency from each other. For instance, this tutorial has a lot more "E's" in it than it does "Q's". Encrypting a message with a Caesar cipher does not change the statistical distribution of letters in a message, it just makes different letters occupy the same frequencies. That is, if a particular Caesar cipher key transposes E's to Q's, you'll find the encrypted version of this tutorial has exactly as many Q's as the original did E's.

Fair enough, but how does an attacker know how many E's were in the original message without knowing the message? He does not need to know this information *exactly*; it is enough to know that E's make up a whopping 13% of normal English prose (not including punctuation and spaces; just letters). Any letter that occurs in 13% of the cipher text is extremely likely to represent an E. Similarly, the most common remaining letters in the cipher text probably represent "T's" and "N's". This is the low entropy (rate-of-language) of English coming back to haunt us. All you need to do is use up all the letters, make sure the message looks like a message, and you are done!



ENGLISH LETTER FREQUENCIES
Per One-Thousand Letters

Sorted By Letter		Sorted By Frequency	
A	73	E	130
B	9	T	93
C	30	N	78
D	44	R	77
E	130	I	74
F	28	O	74
G	16	A	73
H	35	S	63
I	74	D	44
J	2	H	35
K	3	L	35
L	35	C	30
M	25	F	28
N	78	P	27
O	74	U	27
P	27	M	25
Q	3	Y	19
R	77	G	16
S	63	W	16
T	93	V	13
U	27	B	9
V	13	X	5
W	16	K	3
X	5	Q	3
Y	19	J	2
Z	1	Z	1

How to break a substitution cipher, part 3

Tutorial users who enjoy a simple little game can try the following. Refer to the English letter frequency table to the left and decipher the message:

```
SEVRAQF, EBZNAF, PBHAGELZRA, YRAQ ZR LBHE RNEF!
```

The best answer e-mailed to the author will receive future honorable mention in a forum to be determined!

Conclusion

Following this tutorial look for two related cryptology tutorials. In this part, we have addressed the basic concepts of cryptology; tutorial users should now understand the meaning of concepts like symmetric and asymetric algorithms, key-length, cryptanalysis, and algorithms/protocols. Part 2 will address symmetric and asymmetric (public-key) algorithms in terms of the basic mathematical principles of their construction. Part 3 will move on to describe a number of actual protocols, many of which use the algorithms described in part 2 as their building blocks. After completing all three tutorials, users should have an excellent foundation for constructing working cryptosystems that are tailored to their specific requirements.

Further reading

The nearly definitive beginning book for cryptological topics is Bruce Schneier's *Applied Cryptography* (Wiley). I could not have written this tutorial without my copy on my lap to make sure I got everything just right.

Online, a good place to start in cryptology is the *Cryptography FAQ* .

To keep up on current issues and discussions, I recommend subscribing to the Usenet group **sci.crypt**.

Section 6. Feedback

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.