Introduction to Cryptology Concepts I

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 follow-up) 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 I of a two-part tutorial is to introduce readers to general concepts and address cryptanalysis in a somewhat greater depth. Part II addresses cryptographic algorithms and protocols in more detail.

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, an essential exercise is to run a few of them into some brick walls to see just where the failure points arise (using crash-test dummies as proxies in our analogy for test, rather than production, data).

You will not be a cryptanalyst after finishing this tutorial. To do that, you will 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 (after this tutorial).

What tools use cryptography?

Some form of cryptography is 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, cryptographic algorithms wind up constituting important parts of the protocol for use of these programs/data.

Contact

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

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, given the wide popularity of the book, may quickly become so.

Encryption and Decryption I

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

Encryption and Decryption II

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)`**' denotes 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 since all the folks without access to the key K will not know which decryption function to use in deciphering C.

There are lots of details in the design of specific cryptographic algorithms, 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 things one can do 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 (e.g., 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.* There are a number of ways by which Peggy might demonstrate the integrity of a message. 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 measures to publish the hash on a channel less subject 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 picture ID. Non-repudiation has many similarities to authentication, but there are also subtle differences.

Protocols and Algorithms I

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 cryptographic product makers (either out of carelessness or [illegible] misrepresentation). For example, a maker might claim: "If you use our product, your data is secure since 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 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 II

It is worth thinking about 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 email. Suppose that the product utilizes the "unbreakable" algorithm E. Even against the "unbreakable" algorithm, Mallory has many ways to intercept Alice's plaintext, 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 plaintext might not travel the whole way as ciphertext, but rather travel as vulnerable plaintext 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 I

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 by 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 relation 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 panel, 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 II

Actually, there is one extra condition required for public-key cryptography. It must also happen that there is 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 in the assurance 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 (i.e., a key) with our correspondents.

One-way Functions

There are two related types of functions that are not themselves encryption functions, but that 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 which 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 recreate 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 allowing cheating by looking at the original input).

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 poem and does not want anyone else claiming false credit for it (and Mallory surely would do so to promote his own poetic reputation). Unfortunately, publishers being as they are, Alice's press is taking a while deciding on the right typeset font. In the meantime, Alice can still do something to prove her claim. 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, she 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 an 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 producing 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 wrote *The Great American Novel* rather than a haiku).

In some cases, cryptographic hashes will be keyed, much as are encryption functions. In such a case, only someone who possesses the key can verify the hash 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 (i.e., '`H(k+M)`'). However, some keyed hashes utilize a key value in a more deeply integrated way within the algorithm.

What makes a cryptographic protocol "strong"?

Passphrase, Password, and Key I

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 somehow derived from the password or passphrase.

Passphrase, Password, and Key II

Passwords (versus passphrases) are typically rather weak and prone to several attacks. In the very worst of designs (but these worst designs are quite common, unfortunately), a password is simply used directly as a key. For example, an algorithm might allow for a 64-bit key, and the application designer decides to get this 64-bits by having a user type in 8 characters (and using their 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 8 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-magnitude-weaker than the algorithm itself might suggest. Even if using a "seed," "whitening," or other transformation to compute the final key, the range of passwords people will tend to type in will inherently limit the strength.

A passphrase, typically, might allow a user to type in 20, 50, or 100 characters. Even though each character is still probabilistically constrained, 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 whereby it is possible to sample just the needed 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 outputs, 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.

What often happens is that people 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 (maybe meaning "competitors") do not learn the details of how a protocol/algorithm/application works, they will not be able to break it. Or maybe the naive folks just think that their whiz-bang new algorithm is so novel and brilliant that they don't want anyone to steal their ideas. Either way, security-through-obscurity ranks up there with a belief in the tooth fairy in terms of scientific merit.

It is easy to spin wild scenarios of how some system 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, loosening lips, and black box analysis just are not difficult or rare enough to trust serious security concerns against.

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 a very good job of this secrecy); make your algorithms public to the world!

Key Lengths and Brute-Force Attacks I

A "brute-force attack" is one attack available for 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-bound 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 plaintext. 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 basically, in most systems, Mallory will not have too much trouble knowing when he has guessed the right key.

Key Lengths and Brute-Force Attacks II

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). We are a little fuzzy on the exact times, but we can see why that doesn't matter.

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 III

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 that 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?

Alice's message is indeed fairly secure against brute-force attacks. But maybe not *quite* as safe as we have supposed above. When we start thinking about years of brute-force attack, we really need to think about Moore's Law in the package. Moore's Law claims (roughly) 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 do. 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? Alice nonetheless 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), a lot of 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 may 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, 8 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 (8 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 combos 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.

Cryptanalysis

Weak-key Attacks

More subtle problems can create dictionary-like attacks also. For example, say that some pseudo-random algorithm rather than human users select the keys. 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."

Entropy, Rate-of-Language, Unicity Distance

Plaintext 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 plaintext regularities. The significance of these concepts is that statistical regularities in plaintexts are nearly as helpful for cryptanalysts as would be actually knowing the exact messages in question.

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).

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 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.

Unicity distance: *The length of cyphertext 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 plaintext, 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 plaintext, unicity distance is about 8 characters of text.

Schematic of Basic Attacks

There are a number of approaches an attacker might take 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 are all most relevant to encryption protocols as such, and only indirect to other sorts of protocols that might be compromised (e.g., 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.

Ciphertext-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 plaintext. Brute-force attack on the key is one example of this type of attack.

Known-plaintext: In some cases, an attacker might know some or all of the encrypted plaintext. 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-plaintext 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-plaintext: An attacker might have a way of inserting specially selected plaintext into messages prior to their encryption. Initially, this might seem unlikely to occur; but let us give a plausible example. Suppose Alice runs a mail server that filters out suspected email viruses. Furthermore, Alice forwards an encrypted copy of suspect emails to virus expert Bob. Attacker Mallory can deliberately mail a virus (or just 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-plaintext: This attack is just a more specialized version of a general chosen-plaintext attack. Each time Mallory inserts one chosen plaintext and intercepts its encrypted version, he determines some statistical property of the encryption. Selection of later chosen-plaintexts are chosen 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 utilizable key properties.

Chosen-ciphertext: An attacker might be able to determine how selected ciphertexts get 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 and/or store it in an insecure way. By choosing ciphertexts (or really pseudo-ciphertexts) 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 vulnerability of actual encryption systems usually comes 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." People can be bribed, cajoled, and tempted.

Of course, still other human factor vulnerabilities arise in real-world encryption. You can search people's drawers for passwords on scribble 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 a lot of 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.

Unconditional Security, 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 infeasible to attack (by any style 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 ciphertext C and wants to decrypt it (say by brute-force attack). However, for any possible decryption M, Mallory can attempt using 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 ciphertext and unknown key, to determine if he has hit upon the correct key.

Endgame

How to Break a Substitution Cipher I

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 plaintext message letter by letter, looks up each letter in the source column, and transcribes the corresponding target letter onto the ciphertext tablet.

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. 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 ciphertext is extremely likely to represent an E. Similarly, the most common remaining letters in the ciphertext 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!

How to Break a Substitution Cipher II

Tutorial users who enjoy a simple little game can try the following. Refer to this English letter frequency table (http://gnosis.cx/download/letterfrequency.gif) and decipher the message:

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

The best answer emailed to the tutorial author will receive future honorable mention in a forum to be determined!

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 of Schneier 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**.