

Introduction to cryptology: Pt. 3

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.

1. Introduction to the tutorial	2
2. Background and reminders	4
3. Steganography and watermarking	9
4. "Exotic" protocols	16
5. Resources	25
6. Feedback	26

Section 1. Introduction to the tutorial

Is this tutorial right for you?

This tutorial builds on the foundation provided by the two introductory tutorials on general cryptology concepts ([Part 1](#) and [Part 2](#)). You don't necessarily need to have completed the introductory tutorials, but you should be familiar with general cryptology concepts, such as symmetric encryption algorithms, asymmetric encryption algorithms, cryptanalysis, attacks, Alice and Bob, messages, hashes, cipher text, and key length.

If you feel comfortable with these concepts, you should not have difficulty understanding this tutorial. If answers to those questions are unclear, take a quick look at Parts 1 and 2 of this tutorial series by David Mertz. (The section below, "Background and reminders," includes a brief overview of important concepts.)

In general, this tutorial is aimed at programmers who wish to become familiar with cryptology, its techniques, its mathematical and conceptual basis, and its lingo. Most users 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).

What does this tutorial cover?

This intermediate tutorial introduces users to a variety of protocols that are useful for accomplishing specific and specialized tasks. Algorithms as such are not covered here, but are treated as building blocks for larger protocols. For example, a protocol discussed here might, as a general assumption, state something like: "Assume E() is a strong symmetric encryption algorithm with key length of 256 bits." It is up to tutorial users to know what this statement means; and it is up to protocol implementers to actually choose an appropriate algorithmic building block. However, the "Resources" section provides information on a number of common building blocks (so that might be a good place to start).

The number of things you can accomplish with cryptographic protocols is quite astonishing! Many readers will be surprised that some of the matters discussed here are possible at all. The author certainly was when he first encountered many of them. Moreover, this fairly brief tutorial is unable to address every protocol and goal cryptologists have developed. If something is not covered here, please do not assume that means its goal cannot be accomplished cryptographically. It likely means the tutorial author simply did not include it (either because of limits of space or limits of his knowledge). Then again, there *are* certain goals that are easy to state -- and that you might find discussed and requested repeatedly in discussion forums -- that simply bump up against mathematical impossibility. The difference is not always obvious. You might need to think about the issues at some length, and ask questions of folks with some experience.

Contact

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. Background and reminders

Protocols and algorithms

One fundamental notion introduced in the earlier tutorials is worth emphasizing again before we get underway: It is important to make the distinction between protocols and algorithms.

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 a much narrower procedure used to transform 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 a product of a total protocol.

Clearly, using a strong and appropriate algorithm is an important part of creating a strong protocol, but it is not sufficient by itself. The first sections of this tutorial mostly address how cryptographic algorithms work; the later sections take a look at the use of some algorithms in actual protocols, particularly protocols that combine multiple algorithms to accomplish complex goals.

Block ciphers and stream ciphers

Encryption algorithms can be divided into **block ciphers** and **stream ciphers**. Stream ciphers are able to take plain text input one bit (or one byte) at a time and output a corresponding cipher text bit (byte) right away. The manner in which a bit (byte) is encrypted will depend both upon the key used and the previous plain text stream encrypted leading up to that point.

In contrast to stream ciphers, block ciphers require an entire block of plain text input before they can perform any encryption (typically blocks are 64 bits or more). In addition, given an identical plain text input block and an identical key, a block cipher will produce the same cipher text no matter where in an input stream it is encountered.

Although stream ciphers have some advantages in cases where immediate responses are required -- for example on a socket -- the majority of widely-used modern encryption algorithms are block ciphers. In this tutorial, whenever symmetric encryption algorithms are discussed generically, the user should assume the tutorial is referring to block ciphers.

Some impossible things

Many of the things cryptography cannot accomplish are quite simple to understand, but nonetheless they repeatedly prompt wishful thinking by people getting started in cryptology. In some cases, vendors promise algorithms and protocols with properties that are impossible to achieve. A good place to cultivate suspicion about impossible claims is the Snake Oil FAQ (see Resources). A few impossible (or at least suspicious) things are worth considering before proceeding with this tutorial.

Impossible things with random numbers, part 1

Random numbers are important to a variety of cryptographic protocols, such as randomly-generated keys and seeds. A problem one runs up against when seeking random numbers is that it's *impossible* to generate *true* random numbers from deterministic algorithms. Instead, what algorithms produce are "pseudo-random numbers" -- such algorithms are called "pseudo-random generators" (PRGs).

The difference between pseudo-random numbers and genuine random numbers is a basic fact of information theory. A genuine random number contains as much *entropy* or *information* as its bit length. Any algorithm that can be written in a computer language cannot contain more entropy than is contained in its actual source code (including any contained in language libraries and the like). So there is a limit to the number and length of random numbers that can be generated by a given algorithm. After a while, patterns start occurring in pseudo-random streams. By gathering some real-world random seed information (for example, the microsecond timing of a user typing a phrase, or a bit of information about external changes in the Internet), the entropy of a PRG can be improved, but only by the amount of the entropy content of the real-world seed data.

Impossible things with random numbers, part 2

Finding a way of making one-time pads (OTPs) from PRGs is something like a philosopher's stone for beginning cryptologists. One-time pads, you may recall, have the wonderful property of being provably and unconditionally secure. As long as genuinely random data (of the same length as the message being encoded) is used only once, an attacker has absolutely no way of deciphering which message (of the given length) was encoded. Further, an attacker's failure here is not just a computational matter of exceeding the MIPS of all the computers that exist (or might be built), but rather the mathematical fact that nothing distinguishes an actual crack from a false decipherment.

Of course, OTPs have the inconvenient quality of requiring the out-of-channel exchange of a great deal of key material. And the key material gets used up automatically as messages are sent (unlike the keys in other algorithms which can be reused over many messages without being consumed, *per se*). As a consequence, a lot of beginners develop an understandable desire to combine the provable security of OTPs with the finite key distribution requirements of other systems. The result is, frequently, a system that will generate "keys" for a purported OTP system by using pseudo-random generators. PRGs can keep generating new "key" material indefinitely; and at a first pass, these keys appear to have the same statistical and stochastic properties as true random key material.

The catch is that pseudo-random key generators really do *not* have the same deep properties as true random keys. Many PRGs are quite good, but in the end, their entropy is as finite as their algorithms and seeds; they always exhibit cyclical patterns. Mind you, finding these patterns might require serious cryptanalysis. And in the best case (such as with many good stream ciphers), the security provided by PRGs is quite adequate -- even comparable with other strong systems. But this is no free lunch: A PRG is *not* really an OTP.

Provable security

Provable security is another feature that is sought -- and even claimed -- fairly frequently. It turns out that there actually *are* some very interesting proofs for security properties of some algorithms; however, these proofs must be regarded within their precise mathematical context, and what they prove is contingent upon all sorts of assumptions and limitations. Further, most algorithms that have provable properties like this are developed for academic research purposes. In general, none of the algorithms in widespread use (whether public-key or symmetric) has rigorously proven mathematical properties. Instead, we settle for algorithms that have stood up well to years of attack efforts by the best cryptanalysts. This is not the certainty of a mathematical proof, but it is pretty good.

The point of these observations is that you should look with suspicion upon vendors or amateurs who claim to have proven the security of their algorithms. Most likely they have not, except perhaps in highly constrained and circumscribed ways. Unless you are the type of expert cryptanalyst who is able to evaluate such alleged proofs (and if you are, this tutorial is way too basic for you), you should take claims about provable security with a grain of salt.

Distributing "secret" software

Cryptographers often seek to make and distribute software that performs some action, but prevents users from carrying out that same action without having access to the software. Usually, this goal goes hand-in-hand with a desire to control the distribution and use of mass-produced commercial software -- but it sometimes pertains to other security features of the software.

In general, this goal is impossible to accomplish. If a determined attacker has access to your software, he inherently has the ability to determine what the software *does*. If there is a key, or an encryption algorithm, buried within the software (perhaps in obfuscated form), reverse engineering can always reveal that "secret" key/algorithm. It may well be that it is not worth an attacker's *effort* to find your software's secret, but cryptography never gives software the ability to perform non-replicable magic.

Entropy and compression

Another matter worth mentioning, which relates only partially to cryptology itself, is that you might hear claims that new lossless compression methods have been discovered that have fundamentally new properties. In extreme cases, a compression algorithm is sometimes purported to compress *any* data sequence by some amount. There is a one-line *reductio ad absurdum* for this case: Iterate compression of each "compressed" result; if everything is compressible, you wind up with a one-bit (or zero-bit) representation for every original data sequence. But weaker claims are often similarly absurd.

A basic understanding of compression is important to cryptology because both largely come down to the same concept of entropy and information content. Not all data is compressible for the same reason that PRGs cannot generate OTPs -- the redundancy, entropy, and information content of data are fundamental properties of that data, and these factors fundamentally constrain what transformations can be made to data.

Section 3. Steganography and watermarking

What is steganography?

Steganography (in Greek, "secret" + "writing") is the practice of hiding secret information inside non-secret, or less secret, information. Various methods of steganography predate electronic/computer cryptography by centuries: invisible inks, conventions for altering public texts, code words, etc. What distinguishes steganography from plain old encryption is that an attacker does not know with certainty that there is *any* secret message inside another message. In some circumstances this can be important for plausible deniability; in others as a diversion for an attacker; in still others as a way of subverting a channel that an attacker might choose to leave open.

It is worth giving a couple of hypothetical examples of steganography here. In order to pass a secret message, a typed letter might include a number of deliberate "typos," the position of the words with "typos" would encode a subset of the numbers between one and the number of words in the letter. An attacker would not know whether an intercepted letter contains a subtext or subchannel -- or whether it simply has typos (as do many letters with no hidden message). Obviously, the recipient must be aware of the protocol used to encode the subchannel. Similarly, a sound recording (for example, one played on the radio) might have a number of clicks and pops added to it that are indistinguishable from scratches on a vinyl record (in fact, they could be produced by making scratches in such a vinyl record before playing it). The exact timing of the pops would encode a message (for example, the millisecond gaps between successive pops encodes a series of numbers). Because regular phonograph recordings also contain pops, an attacker would not (immediately) know whether a particular song played on the radio actually contains a subtextual message.

What is watermarking?

Watermarking is similar to steganography, but is not quite the same thing. (You might, however, see them discussed together). In the old-fashioned case, both invisible ink and an authenticating watermark might appear on a sheet of paper and require special procedures to reveal. In digital data, almost exactly the same situation exists. But the purpose of a watermark is always to be implicitly available for revelation in appropriate circumstances; the purpose of steganography is to hide its own existence from those unaware of its method of revelation. In digital terms, a watermark might be something a copyright holder puts inside a digital image to prove she is the owner, whereas a steganographic message might be something a political dissident puts inside a digital image to communicate with other dissidents (in which case, a repressive government could not prove the message was sent at all, that it's not simply a family photo). The techniques for concealing the subtext might be similar, but the concealer's relation to an attacker is almost exactly opposite.

Watermarking vs. signatures

It is worth contrasting watermarks with another technique that serves a somewhat similar purpose: signatures. In both physical and digital forms, the basic difference is that a watermark is harder to remove than a signature. A digital file with a specified format can have a digital signature *appended* to the end of it; in this way, the signer purports: "I (signer) agree to/authorize the content/meaning of this digital file." But it is simple to utilize the digital file and discard the signature; doing so removes the claim made by the signature (in the same way that scissors can be used on a signed sheet of paper). A watermark is much more closely tied in with the file; ideally you would not be able to remove the watermark without altering the content in any obvious way (this doesn't hold true in the paper/scissors example, and some watermarks are designed to photocopy in a way that makes copying evident). Of course, if you have the option of defining what constitutes a valid digital file format, you can explicitly specify that it include a digital signature (from a certain party, if necessary); a file without a signature can be considered automatically invalid by an application (or operating system).

Problems with watermarking, part 1

Digital watermarking is an increasingly desired, but (in this author's opinion) conceptually flawed cryptographic technique. Overwhelmingly, digital watermarking is proposed as a way to prevent (or at least identify) unauthorized reproduction of digital information. A prominent and recent example is the Recording Industry Association of America's (RIAA) Secure Digital Music Initiative (SDMI). The idea behind a digital watermark is to scatter some bits into a digital file in such a way that the scattered bits cannot be identified by an attacker, and therefore cannot be removed or altered without making the changes evident (in the case of analog source media, such as sound, video, and images, this amounts to assuring unacceptable degradation of the quality of the source).

Problems with watermarking, part 2

The problem with digital watermarking is that it seeks to break mathematics and information theory. The trick is to keep in mind perceptual content and compressibility. The real meaningful information in a digital file that *represents* an analog source is whatever features can be perceived by a human (or perhaps in certain cases by a machine, but the issue is the same). Anything that cannot be perceived is *noise*, not meaningful content; not every bit in a digital representation of analog data is necessarily *information* in the proper sense. An ideal (lossy) compression algorithm for analog data (MP3 and Ogg Vorbis come close for sound; JPEG comes close for images; video compression techniques are still subject to large improvements) keeps every perceptible feature of the representation while discarding every non-perceptible feature. While you cannot know for certain the "ideal-ness" of a single digital representation, a smaller representation producing the same perceptible features is closer to this "ideal." A digital watermark is, by definition, a non-perceptible feature (otherwise the perceiver could simply remove it). In other words, the watermark adds entropy to the digital encoding, while doing nothing to add *meaningful information* to the representation.

SDMI is a good illustration. In developing a music format that includes copyright identification (digital) information, the RIAA has *exactly* two choices at a conceptual level: (1) They can increase the size of music files over the size of an ideally compressed format in order to include the copyright identification; (2) They can replace some of the analog information in the digital representation with copyright information (in other words, make the format *sound worse* to a discerning ear). The exact same tradeoff exists for watermarks in images and other analog sources. In practice, no digital watermarking format has ever stood up to any serious scrutiny, and watermarks have always proven to be relatively easy to remove once analyzed. In theory, there is an inherent conflict between the goals of maximum compression and inclusion of a watermark.

Digital steganography using images

In order for steganography to find a handle in digital data files, the format of those files must contain a degree of non-predictable variation. Steganography operates by substituting desired bit values in unpatterned bit positions. Fortunately, many file formats contain quite a bit of non-predictable variation. The most commonly-used file formats for steganography are those that encode real-world (analog) data, such as image and sound formats. Typically, a subchannel in an image is encoded in the "least significant bits" of that image. That is, if each image's pixel color is encoded with a number of bits (often 24) some of those bits cause less color variation of the pixel than others do. Specifically, 24-bit images usually have 8-bit values devoted to each primary color (red, green, blue). If the image is generated through a real-world process (such as taking a photograph), the sequence of lowest-order red bits will be largely random to start with (because of the finite resolution of cameras and also because of "random" variations in the pictured object). A steganographic encoding might substitute subchannel values into that sequence of lowest order red bits (red variation is the least perceptible of the primary colors). The receiver reads the subchannel back out of a received image by stripping out everything other than the sequence of lowest order red bits (which are identified purely positionally by the file-format structure).

Digital steganography using other formats, part 1

Digital steganography is often used with images and sounds simply because these file formats make it easy to identify the areas of variability in a purely structural way. This might be as simple as knowing that every 24th bit in the file (after some initial offset) is a lowest order red bit. Other file formats can be used, but often require more semantic consideration of the file contents. Let us look at a few examples.

Source code. Programming languages have fairly strict structural constraints. That is the point of grammar, after all. Even within grammatical constraints, most changes to a source code file will result in programs that do not compile or run (for example, you might be able to change a character in a variable name in a subttextual way, but doing so will most likely break the program logic in some manner). Even so, there are a number of areas of non-predictable variation even in source code files; the trick is that encoding them involves "understanding" the code in a richer way than changing recurrent bit positions. Many programming languages offer several equivalent constructs for the same operation: for example, both " \neq " and " $<>$ " to express inequality. Or at a higher level, you might even automate transformations between different (equivalent) loop structures (for example, `for(;;){...}` and `while(1){...}`). The pattern of choices between constructs could contain one bit of subchannel for each loop occurrence. Still, the best place to hide a subchannel in source code is likely to be in the comment fields -- but with some subtlety to make it look like actual source code comments (you *do* comment source code, right?).

Digital steganography using other formats, part 2

Delimited data. Data file formats are, in most cases, even more rigidly structured than source code. Delimited data is a good example, but the same line of thought applies to many other data formats (XML, however, has a lot of optional white space, which could make for a good subchannel). At the level of *content*, however, data file formats have non-predictable variation *by definition*. After all, the point of actually *sending* a data file is to convey the information in it that the recipient does not know. For example, a row record for a person might have a first name, last name, and social security number, each of which must look a fairly specific way. But the actual SSN a person has is not predictable from the other information. A possible subchannel exists in subtly varying this data content. However, a danger of revelation exists if an attacker has independent ways of correlating data (if no one in your data file has the true match between name and SSN, this looks suspicious to an attacker). Finding this kind of subchannel requires specific knowledge of the data format and content being used.

Digital steganography using other formats, part 3

Compressed archives are probably the very worst format for trying to insert a subchannel. The problem here is that almost every bit change in an archive has an effect on many bits in the unpacked contents, and in a way that depends on the whole archive contents. Changing a bit or two at random is extremely likely to produce unpacked files that have invalid file formats (or just corrupt archives). This is easy for an attacker to notice. About the only place a few bits of subtext might be located is by taking advantage of the error-correcting codes (ECC) some archive formats use. You could introduce an occasional "error" in archives of the type the ECCs would correct upon unpacking. One trick would be to make sure that archives with subtexts do not have many more errors than archives without subtexts (which means introducing random "errors" to all transmitted archives that an attacker might intercept).

Natural language text. Natural language is extremely free-form, and thus is an excellent format in which to embed a subchannel. Normal texts contain all sorts of spacing variations, word-choices, types, and other "random" features. But then, a too-obvious subchannel encoding strategy is easy to detect. Sure, people make typos, but not in uniformity in every third word. Too much pattern in the "random" variations is easy for a machine scan, or a human reader, to identify as a probable subtext.

Cryptanalysis of digital steganography, part 1

The efficacy of a subchannel encoding strategy is simply a measure of how well it prevents an attacker from proving the existence of the subchannel. Of course, another desirable feature of a subchannel is the ability to embed more, rather than less, bandwidth in it. Sometimes a couple of bits of subtext are sufficient for a particular purpose; but most of the time you are more likely to need the ability to send more extensive messages. Unfortunately, the goals of bandwidth and invisibility tend to pull in opposite directions: More fiddling with bits makes detection more likely and easier.

Your first assumption in designing subchannel encoding should be that an attacker is at least as able to identify non-predictable variation as you are. Do not try to hide the message simply by assuming an attacker will not know where to look. The key in maintaining the invisibility of a subchannel is to make sure that the *distribution* and *pattern* of subchannel bits closely match those in a typical file of the same format.

In many cases, the expected distribution of pre-encoded subchannels will be uniform and stochastic -- but not always. You have to look at whether there is a bias toward 1 (or 0) in the pre-encoded subchannel slot (the bits or variations you have identified as encoding sites); but you also have to look at whether there is a frequency shift between the start and end of a file and/or whether cyclicalities exist in bit frequencies of pre-encoded subchannels. A good first step is to extract a large number of pre-encoded subchannels, and see if this data is compressible (if so, it is not purely stochastic and uniform, and you need to look more closely at the patterns).

Cryptanalysis of digital steganography, part 2

Pure plain text messages are absolutely *terrible* candidates for subchannel encoding. A bit pattern that works out to the ASCII sequence "Secret meeting at 6 PM" is a dead giveaway (maybe literally!). Assuming you are aiming for stochastic-looking bit patterns, compression removes much redundancy. But watch out for compression headers: A subtext that begins with "PK" does not look like random data (for example, PKZip header bytes). The best choice is usually to compress plain text first (mostly just to save on a limited subchannel bandwidth), then to encrypt the compressed text second. Of course, you also have to watch out for encryption format headers (i.e., choose a format that is headerless). The use of a symmetric key requires a separate key negotiation out-of-channel; but use of public-key systems can avoid this requirement.

Without a doubt, the most important design issue in creating steganographic subchannels is: *Don't use stock files!* If you use files to which an attacker has access to the original copy, a simple binary comparison of the original with the new overt message will reveal that the file has been tampered with. This applies especially to image or sound files that exist in the public domain (or generally, in public, even if copyrighted). If you downloaded an image from the Web, so can an attacker. What you really need are entirely original files -- ones that you have a plausible reason for sending other than to hide subchannels. Home videos, for example, are bulky files with lots of unique subchannel bandwidth. Of course, if you leak these originals to an attacker, you have destroyed your system; the same applies if you encode different messages to different parties based on the same original. You should treat a steganographic overt message much like you would a one-time pad: Use once and destroy! However, multiple digitizations of the same analog original are possible; they will differ in much more than just the subchannel, so a binary comparison just shows them as wholly different files.

Cryptanalysis of digital steganography, part 3

Two smaller issues are raised in the previous panel. One is that the files you send need to be plausible. Do you generally send pictures of your family to your business associates? Maybe yes, but if not, sending them just announces the likelihood of a subchannel. The prior discussion of techniques for other file types might be useful in strategizing plausible files for normal transmission. The second issue was mentioned earlier: If your subchannel encoding involves altering non-predictable data, can an attacker gain access to that same data in other non-identical files? For example, suppose you have a strategy for altering information in transmitted flat-file records. Good enough, so far. But can an attacker gain access to individual records by other means, or at other times? Perhaps you have sent an intersecting record set (either with or without a subchannel), or want to later on. If the alterations are inconsistent in individual records, this provides a clue to a subchannel (obviously, production data changes occasionally, but within some limits).

Section 4. "Exotic" protocols

Shared secrets, part 1

The general idea behind **secret sharing** is that you might want to require multiple parties to cooperate in order to decrypt a certain cipher text. It is not enough for one person to have her key, she needs some help accessing the plain text. It turns out that you can design schemes of arbitrary complexity that specify exactly who has to cooperate to decrypt a particular message. For example, you could specify a "Chinese menu" approach, where you need two from column A, three from column B, and one from column C, to decrypt a message. Even more complex dependencies are possible as well. For example, if Alice uses her key, she needs Bob's help; if Carol uses her key, she needs Dave's help (only one combination will work).

The simplest case of secret sharing is **secret splitting**. This protocol requires the cooperation of all parties (two or more) to decrypt a message. The protocol is quite simple:

```
Given a secret M, of length n.
Given N persons who will share the secret (named P1, P2, ..., PN).
Generate random bit strings R{1}, R{2}, ..., R{N-1}, of length n.
Calculate S = M XOR R{1} XOR R{2} ... XOR R{N-1}.
Destroy or hide M.
Give S to P1
Give R{1} to P2
[...]
Give R{N-1} to PN
```

The secret splitters need not even know which one receives S, and which ones receive the Rs. Either way, M can only be constructed by XOR-ing back together the information given to every person. This works just like a one-time pad, and has the same degree of absolute security (it is subject to bad random numbers and human weaknesses, but those contravene the explicit protocol).

Shared secrets, part 2

Secret splitting is simple and provably secure, but it also has some limitations. If any one party loses his portion, or becomes unwilling or unable to share it, no sharer can get at the secret message. The secret splitting protocol also puts total power in the hands of the person who originally generates the split secret (but then, M belonged to that person as well). Furthermore, there are a number of ways in which a malicious party, who either genuinely knows a secret share or pretends to, can find another person's portion without revealing her own portion and/or the message. All of these limitations can be avoided in other (more complex) protocols. The "Resources" section can lead tutorial users to many of these specifics; here we will only discuss **(m,n)-threshold schemes**.

Before we do, though, it is worth making a general observation. The secret shared in secret sharing schemes need not be the ultimate interesting content. In practical terms, the size of calculations and distributed portions can be limited by letting $C = E\{K\}(M)$ for a strong symmetric-key algorithm. C can be revealed to everyone (even those not involved in the secret sharing), while K rather than M becomes the secret to use in a sharing scheme. Good encryption algorithms use keys of less than 256 bits, while messages themselves might well be multiple megabytes in size. The math in most protocols is computationally intractable for the numbers represented by huge files, but reasonable-sized keys can act as effective proxies for the actual secret message.

Shared Secrets, part 3

The **LaGrange Interpolation Polynomial Scheme** is an easy-to-understand (m,n)-threshold scheme for secret sharing. The "Resources" section includes others.

Suppose you want to share a secret, M, among n people, such that any m of them will be able to reveal the secret by cooperating.

1. Generate a random prime, **p**, larger than M.
2. Generate n-1 random integers, $R\{1\}$, $R\{2\}$, ..., $R\{n-1\}$, each less than p.
3. Let **F(x)** be a polynomial in a finite field, defined by: $F(x) = (R\{1\} * x^n + R\{2\} * x^{(n-1)} + \dots + R\{n-1\} * x + M) \bmod p$
4. Generate m "shadows" of F, defined by: $k\{i\} = F(x\{i\})$ where each $x\{i\}$ is distinct (using successive integer values [1,2,3,...] is a fine choice for x's).
5. Give [p, $x\{i\}$, $k\{i\}$] to each of the m secret sharers, for i corresponding to the number of each sharer (the enumeration is arbitrary).
6. Destroy $R\{1\}$, $R\{2\}$, ..., $R\{n-1\}$.
7. Destroy or hide M.

Given the information provided, each secret sharer is able to write out a linear equation. For example, Linda, who was enumerated as sharer #1, can construct the equation:

$$k\{1\} = (C\{1\} * x\{1\}^n + C\{2\} * x\{1\}^{(n-1)} + \dots + C\{n-1\} * x\{1\} + M) \bmod p$$

Because these linear equations have n unknowns -- $C\{1\} \dots C\{n-1\}$ and M -- it requires the n equations with these same unknowns to solve the system of equations, and thereby reveal M (and also the $C\{i\}$'s, but these are not interesting once we have M).

Because the coefficients of F were chosen randomly, the cooperation of less than n secret sharers -- even combined with *infinite* computing power -- does not allow revelation of M. Without the nth sharer participating, any possible M (of a length less than p) is just as consistent with the (less than n) equations as any other!

Key escrow

There may be times when it is desirable to give a secret key, or indirect access to a secret key, to parties other than those directly involved in a secured communication. Unfortunately, most of the time the issue comes up in a context the author finds undesirable, such as providing a (possibly circumscribed) backdoor to "secure" communications to a government/police agency and/or to corporate employers. Cryptography is a technology that cannot be fully considered apart from its political implications.

However, there are other legitimate reasons for **key escrow**. It may happen that you would like certain people to have the ability to access your secured communications in the event you are no longer able to divulge them yourself (or do not wish to require your effort, given certain circumstances). Two techniques are useful for key escrow goals (and can be used individually or jointly): multiple recipient lists and secret sharing of keys.

Key escrow: Multiple recipient lists

You are probably aware that most concrete public-key encryption systems actually use symmetric "session keys" to encrypt messages, and public keys only to encrypt these session keys. Computational speed considerations are the main motivation behind such split systems, but they also have desirable side effects. In fact, even when using entirely symmetric-key systems, the same sort of split systems can be useful. It is possible to encrypt a session key for multiple recipients, not merely for one. While you could send the same encrypted message to multiple parties, it might be easier to simply attach multiple versions of the encrypted session key, and allow general distribution. This might look like the following:

```
Let  $E\{k\}$  be a symmetric-key encryption algorithm.
Let  $S$  be a random session key.
Let  $M$  be a message.
Let  $K_a$  be Alice's public or symmetric key.
Let  $K_b$  be Bob's public or symmetric key.
Generate  $C = [E\{S\}(M), E\{K_a\}(S), E\{K_b\}(S)]$ .
Make  $C$  available to both Alice and Bob.
Destroy  $S$ .
```

Either Alice or Bob can determine S from C . And once they have S , they can decrypt M . Other parties besides Alice and Bob have no access to S or M (C *does* use E with three keys over two messages, so this provides a bit of extra cipher text for attack). A nice property of C is that it is not much bigger than $E\{K_a\}(M)$, which would be a direct way of encrypting for Alice only. Certainly, for megabyte messages and 128-bit keys, the extra session key encryption is insignificant.

If Alice is intended as the direct recipient, but Bob should be able to get access to M if he needs to (and at his own discretion), then this scheme would give Bob an "escrow key." For that matter, we could just send $E\{K_b\}(S)$ to Bob, and forgo sending $E\{S\}(M)$ to him at all; this would make sense if he had access to Alice's stored encrypted files, but not to her key. You can imagine these arrangements might make sense if you wish for an employer to have access to employees' messages should an employee quit (or die, or forget passwords). Of course, it leaves decryption at the employer's discretion (but this might be appropriate for company-related correspondence).

Key escrow: Secret sharing of keys

The second key escrow technique is secret sharing of key material (either session keys or private keys). Suppose that Alice does not wish to disclose her secret key to anyone directly, but does feel that it would be okay for at least five of her 10 friends to decrypt her messages (perhaps she is worried about disposition of her secret inventions after her death; or maybe just about the possibility that she will lose her original private key). In government proposals the same structure is suggested: In the presence of a warrant, multiple non-government agencies would disclose citizens' shared-secret keys. The latter case is politically troubling, but the cryptographic issue is the same in both cases.

Alice can use a (5,10)-threshold scheme to divide her key among her 10 friends. No one except Alice has access to the whole private key, but five friends can recover it by working together (and thereafter decrypt any messages Alice has encrypted using the key). More complex threshold schemes can also be used if the requirements for key revelation are more structured than this. As mentioned earlier, using a threshold scheme for key escrow is consistent with using session keys; depending on the requirement, it might be a message session key rather than Alice's long-term private key that gets distributed in such a scheme.

Zero-knowledge proofs, part 1

For this author, probably the most surprising thing cryptography can accomplish is **zero-knowledge proofs**. The idea behind a zero-knowledge proof is to prove that you have a certain piece of knowledge without revealing the content of that knowledge to an interlocutor. The purpose of a zero-knowledge proof is to demonstrate access to some secret information without *giving* that access to someone else. For example, imagine a conversation between Alice and Bob:

- * Alice: "I can decrypt the confidential message encrypted as C."
- * Bob: "I do not believe you. Prove it!"
- * Alice (bad response): "The key is K, and therefore, as you can see the message decrypts to M."
- * Bob: "Ah ha! Now I know the key and the message also."
- * Alice: "Oops!"

Alice really took a bad approach here, since she failed to keep the message confidential. And she even gave away the key while she was at it (she could have done slightly better if, for example, the cryptographic hash of M could be verified instead of revealing the key; but the idea is the same). A much better conversation for Alice to engage in is:

- * Alice: "I can decrypt the confidential message encrypted as C."
- * Bob: "I do not believe you. Prove it!"
- * Alice (good response): "Let's engage in a zero-knowledge protocol, and I will demonstrate my knowledge with an arbitrarily high probability (but not reveal anything about the message to you)."
- * Bob: "OK."
- * Alice and Bob go through the protocol...

Zero-knowledge proofs, part 2

Zero-knowledge proofs can be generalized to a wide range of information. In fact, it turns out that *any* mathematical theorem with a proof can have a zero-knowledge "proof of the proof." That is, Alice can claim to have a proof of theorem T, but not wish to state it (she wants to wait for publication). Nonetheless, Alice can prove that she *has* proved T without revealing the proof. This very general fact is broad enough to cover specific cases like factoring large numbers and the like, which are involved in many cryptographic algorithms. The broadest scope exceeds this tutorial; we will just look at one case (others are similar in form).

Zero-knowledge proofs, part 3

Graph isomorphism is a *hard* problem; that is to say, it is NP-complete. It is one of those problems that could take millions of computers millions of years to solve, even though *constructing* a problem takes only a moderate amount of time and space. A graph is a collection of vertices connected by a collection of edges. Every edge connects exactly two vertices, but not all pairs of vertices are (necessarily) connected by an edge. Some graphs are *isomorphic* to other graphs, meaning:

For isomorphic graphs G and H , there exists a one-to-one function F such that:

- * The domain of F is the set of vertices of G .
- * The range of F is the set of vertices of H .
- * If and only if $[g_1, g_2]$ is an edge in G , $[F(g_1), F(g_2)]$ is an edge in H .

Obviously enough, if G and H do not have the same number of vertices and edges as each other, they are not isomorphic. But assuming G and H meet this minimum condition of "plausible" isomorphism, determining whether they really are isomorphic basically means attempting every mapping from G onto H , and checking whether it creates an isomorphism.

What this boils down to is that if someone tells you he has two isomorphic graphs with enough thousands of vertices and edges, it is because he constructed the graphs to be isomorphic -- not because he discovered the isomorphism. On the other hand, it is trivial to construct isomorphic graphs with thousands of vertices and edges; you could do it on paper without using a computer if you spent a bit of time on it! Next, let's see why all this is important.

Zero-knowledge proofs, part 4

Suppose that Peggy claims to know an isomorphism between graphs G and H . In practice this means that she has constructed the graphs herself (for large graphs), or at least was provided the isomorphism by someone who did. Knowing this isomorphism might be Peggy's way of proving her identity if it has been published previously that she is the person who knows the isomorphism of G and H . Obviously, just showing the isomorphism directly allows any observer to pretend he is Peggy, so that is no good.

Here is what Peggy does to prove she knows the isomorphism:

1. Peggy randomly permutes G to produce another isomorphic graph I . Since Peggy knows the isomorphism between G and H , it is easy for her to simultaneously find the isomorphism between H and I .
2. Peggy gives I to Victor.
3. Victor may ask Peggy to prove *either* (a) that I and G are isomorphic, *or* (b) that I and H are isomorphic. But Victor may not ask for both proofs (were he to obtain both, he would have the isomorphism proof of G and H himself).
4. Peggy provides the proof requested by Victor.

Zero-knowledge proofs, part 5

So far, so good. What has this shown? If a Peggy imposter *did not* know the isomorphism of G and H, the best the imposter could do is to try to pass off an I that is isomorphic with G (she knows G and H, as does Victor), and just hope Victor doesn't ask for the isomorphism of H and I. Alternately, a Peggy imposter could try to pass off an I she constructed from H, and hope the opposite. Either way, the imposter has a 50% chance of getting caught by the protocol above.

Victor, however, probably does not find 1/2 confidence sufficient for Peggy to prove she knows the isomorphism. Fortunately, Victor can simply demand that Peggy now generate an I' and undergo the protocol again. If she passes now, Victor can be 3/4 confident about Peggy. If that's not good enough, she can do a third pass of the protocol with I'', and obtain a 7/8 confidence; or a 15/16 confidence, a 31/32 confidence, and so on. By iterating the protocol, Peggy can prove that she knows the isomorphism for an arbitrary confidence requirement by Victor (but always less than 100% by some amount). No matter how many times the protocol is iterated, Victor gains no knowledge that helps him in constructing his own G/H isomorphism, so "zero knowledge" is leaked to Victor.

Conclusion

Having finished this tutorial, you have received a good glimpse of many of the surprisingly powerful things that can be accomplished by combining cryptographic building blocks. On the other hand, you have also seen a few cases where mathematical facts limit the wished for -- and sometimes claimed -- capabilities of cryptography. The development of cryptological protocols is beautiful, fascinating, and even fun; but if you're interested, you should investigate further the mathematical and programming details in the Resources section to gain a more complete understanding of the subject.

Section 5. Resources

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**.

A nice [Web page with both good explanations and links to a variety of cryptological topics](#) is provided by John Savard.

For topics related to compression, the author is particularly fond of his own [A Data Compression Primer](#). For background on the several topics in this tutorial that touch on compression, this is a good starting point.

[Snake Oil FAQ](#)

Popular Symmetrical Algorithms

The [National Institute of Standards and Technology](#) has recently completed selection of an algorithm for its Advanced Encryption Standard (AES). The winner was [Rijndael](#) which is thereby guaranteed to become a widely-used algorithm. Rijndael is both powerful and versatile, and makes a good choice for the AES selection, and for general use.

Counterpane's [Blowfish](#) has been popular for a number of years. Its successor, [Twofish](#) was another AES finalist that is likely to continue in widespread use (despite the selection of Rijndael as the winner).

The most widely-used symmetrical encryption algorithm has almost certainly been NIST's (formerly called National Bureau of Standards) [Data Encryption Standard \(DES\)](#). Although DES has developed key length problems with the advancement of computer capabilities, triple-DES is still viable, and even single-DES is an algorithm you are likely to come across in existing products.

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.