Intermediate Cryptology: Specialized Protocols

Introduction to the Tutorial

Navigation

Navigating through the tutorial is easy:

- Use the Next and Previous buttons to move forward and backward through the tutorial.
- 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.

Who is this tutorial for?

The tutorial in front of you builds on the foundation provided by IBM developerWorks' series of two introductory tutorials on general cryptology concepts. Users of this tutorial will not necessarily need to have taken those introductory tutorials, but they should be familiar with general concepts in cryptology, such as: What is a symmetric encryption algorithm? An asymmetric encryption algorithm? What is cryptanalysis? What is an attack? Who are Alice and Bob? What is a message? A hash? A ciphertext? What is keylength, and why is it important? If you feel comfortable answering those questions, you should have no problem following this tutorial. If the answers to those questions seem unclear, take a quick look at our introductory cryptology tutorials by this same author. This first section includes a few reminders about important contepts.

In general, this tutorial is aimed at programmers who would like to familiarize themselves with cryptology, its techniques, its mathematical and conceptual basis, and its lingo. Most users 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 of this tutorial 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 will introduce user to a variety of protocols that are useful for accomplishing specific and specialized tasks. Algorithms as such will not be covered here, but will just be treated as building blocks for larger protocols. For example, a protocol discussed might, as a general assumption, state something like "Assume E() is a strong symmetric encryption algorithm with keylength of 256-bits." It is up to tutorial users to know what it means to be such an algorithm; and it is up to protocol implementors to actually choose an appropriate algorithmic building block. However, the Resources section provides information on a number of common building blocks, so that might prove a good place to start.

The number of things one can accomplish in cryptographic protocols is quite astonishing! We expect users of this tutorial to be surprised that some of the things we discuss are possible at all. The author certainly was when he first encountered many of

them. Moreover, this fairly brief tutorial will not be able to address every protocol and goal cryptologists have developed. If something is not covered here, do not assume that means its goal cannot be accomplished cryptographically. Probably it just means the tutorial author 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 one finds being discussed and requested repeatedly in discussion forums--that simply bang 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 to readers (and tutorial takers). He welcomes any comments, please direct them to <mertz@gnosis.cx>.

Background and Reminders

Protocols and Algorithms

One particular introductory 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 the much narrower 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 a product of a total protocol.

Clearly, using an strong and appropriate algorithm is an important element of creating a strong protocol, but it is not sufficient by itself. The first sections of this tutorial will mostly address how cryptographic algorithms work; the later sections will take a look at the use of some algorithms in actual protocols, particularly protocols combining 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 plaintext input one bit (or one byte) at a time, and output a corresponding ciphertext bit (byte) right away. The manner in which a bit (byte) is encrypted will depend both upon the key used and upon the previous plaintext stream encrypted leading up to this bit (byte).

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

Although stream ciphers have some advantages where immediate responses are required, for example on a socket, the large 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 repeatedly prompt wishful thinking by people getting started with cryptology. In some cases, vendors and "inventors" promise algorithms and protocols that exhibit various impossible properties. A good starting point for cultivating suspicion about impossible claims is the Snake Oil FAQ. It is worth remembering a few impossible (or at least suspicious) things before proceeding with this tutorial.

Impossible things with Random Numbers I

Random numbers are important to a variety of cryptographic protocols, such as randomly generated keys and seeds. A problem one runs up against in wanting random numbers is that it is **impossible** to generate *true* random numbers from deterministic algorithms. Instead, what algorithms get us are called "pseudo-random numbers"--such algorithms are called "psuedo-random generators" (often abbreviated as '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 any given algorithm. After a while, patterns start occurring in pseudo-random streams. By gathering some real-world random seed information (e.g. 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 II

A way of making **one-time pads (OTP's) from PRG's** is something like a philosopher's stone of beginning cryptologists. One-time pads, tutorial users will recall, have the wonderful property of being provably and unconditionally secure. As long as genuinely random data, of the same length as the message to encode, 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, OTP's have the inconvenient quality of requiring 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 wish to combine the provable security of OTP's 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. PRG's 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 keys generators **really** do not have the same deep properties as true random keys. Many PRG's are quite good, but in the end, their entropy is as finite as their algorithms and seeds; they always exhibit cyclic patterns. Mind you, finding these patterns might require the work of serious cryptanalysis. And in the best case (such as many good stream ciphers), the security provided by PRG's is quite adequate--even comparable with other strong systems. But there is no free lunch, PRG is **not** really a OTP.

Provable Security

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

The point of the observations in this panel 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 contrained 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), take claims about provable security with a big mound of salt.

Distributing "secret" software

From a practical standpoint, a cryptographic goal that comes up a lot is to make and distribute software that performs some action, but that prevents users from carrying out that same action without having access to the software. Usually, this type of goal relates to wishes to control distribution and use of mass-produced commercial software, but sometimes it has to do with other security features of the software.

In a general way, this goal is impossible to accomplish. If a determined attacker has access to your software, she inherently has the potential of determining 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 is not going to ever give software the capability of performing non-replicatable magic.

Entropy and Compression

One more matter is worth mentioning that relates only partially to cryptology itself. But it relates enough to be worth mentioning. One sometimes finds claims (less in the last few years) that new lossless compression methods have been discovered that have

fundamentally new properties. In the starkest case, sometimes a compression algorithm is claimed to be able to compress *any* data sequence by some amount. There is a one-line *reductio ad absurdum* for the stark case: Iterate compression of each "compressed" result; if everyting is compressible, you wind up with a one-bit (or zero-bit) representation for every original data sequence. But weaker claims often have similar absurdities contained in them.

The reason a basic understanding of compression is important to cryptology is that both largely come down to the same concept of entropy and information-content. The reason that not all data is compressible is, at heart, the exact same reason that PRG's cannot generate OTP's. The redundancy, entropy, and information-content of data is a fundamental property of that data, and these constrain fundamentally what transformations are possible upon data.

Steganography and Watermarking

What is Steganography?

Steganography (in Greek, "secret"+"writing") is 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 of an attacker; in still others as a way of subverting a channel that an attacker has reasons to leave open.

It is worth giving a couple hypothetical examples of steganography to "get" the concept. In order to pass a secret message, a typed letter includes a number of deliberate "typos," the position of the words with "typos" encodes a subset of the numbers between one and the number of words in the letter. An attacker does not know whether an intercepted letter contains a "sub-text" or "sub-channel"--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 sub-channel. Or again similarly, a sound recording (for example, one played on the radio) has 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 encodes a message (e.g. the millisecond gaps between successive pops encodes a series of numbers). Since regular phono-recordings also contain pops, an attacker does not (immediately) know whether a given song played on the radio actually contains a sub-textual message.

What is Watermarking?

(Digital) watermarking is similar to steganography, but is not really quite the same thing (but you might see them discussed together). In the old fashion case, both invisible ink and an authenticating watermark are features that appear on a sheet of paper that take special procedures to reveal. In digital data, almost exactly the same similarity 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 existence from those unaware of its method of revelation. In digital terms, a digital watermark might be

something a copyright holder puts inside a digital image to prove she is the owner. A steganographic message might be something a political dissident puts inside a digital image to communicate with other dissidents (a repressive government cannot prove the message was sent at all, rather than just a family photo). The techniques for concealing the sub-text might be similar, but the concealer's relation to an attacker is almost exactly opposite.

It is worth contrasting (digital) watermarks with another technique that serves a partially similar purpose: (digital) signatures. In both physical and digital forms, the basic difference is that a watermark is harder to remove than is a signature. A digital file with a specified format can have a digital signature *appendend* to the end of it; this is a way for the signer to purport "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 (scissors can do the same thing for a signed sheet of paper). A watermark is much more closely tied in with the file, ideally one would not be able to remove the watermark without altering the content in an evident way (scissors cannot do this in the paper case, 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 includes a digital signature (from a certain party, if needed); a file without a signature can be considered an automatically invalid file by an application (or operating system).

Problems with Watermarking

Digital watermarking is an increasingly desired, but (in this author's opinion) deeply conceptually flawed, cryptographic demand. 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 in a digital watermark is to scatter some bits in 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 amount to assuring unacceptable degradation of the quality of the source).

The problem with digital watermarking is that it wants 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 detected by a human perceiver (or maybe in certain cases by a machine-perceiver, 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 right sense. An ideal (lossy) compression algorithm for analog data (MP3, 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 one cannot know for certain the "ideality" of a single digital representation, as a comparative matter 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 proved relatively easy to remove once analyzed. In theory, there is an inherent conflict between goals of maximum compression in a format and the goal of a format containing 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 sub-channel in an image is encoded in the "least significant bits" of the image. That is, if each image pixel's color is encoded with a number of bits, often 24, some of those bits cause the 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 finite resolution of cameras and also because of "random" variations in the pictured thing). A steganographic encoding might substitute sub-channel values into that sequence of lowest order red-bits (red variation is the least perceptible of the primary colors). The receiver reads the sub-channel 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 I

Images (and sounds) are often used for digital steganography simply because it is very easy to identify the areas of variability in a purely strucural way. It 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 a grammar, after all. Even within grammatical constraints, most changes to a source code file will result in programs that will not compile or run (e.g. you might be able to change a character in a variable name in a sub-textual way, but most likely doing so will 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 "!=" and "<>" to express inequality. Or at a higher level, one might even automate transformations between different (equivalent) loop structures (e.g. `for(;;){...}` and `while(1){...}`). The pattern of choices between constructs

could contain one-bit of sub-channel for each loop occurrence. Still, the best place to hide a sub-channel in source code is likely to be in the comment fields (but with some subtlety to make it look like real source code comments; you do comment source code, right?)

Digital Steganography using other formats II

Delimited data. Data file formats are even more rigidly structured than source code, in most cases. 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 whitespace, which could make for a good sub-channel). 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 firstname, lastname and SSN, 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 sub-channel 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, that looks suspicious to an attacker). Finding this kind of sub-channel requires a quite specific knowledge of the data format and content being used.

Compressed archives are probably about the very worst format for trying to put a sub-channel in. The problem 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 sub-text might be located is by taking advantage of the error-correcting codes (ECC) some archive formats use. One could introduce an occassional "error" in archives of the type the ECC's would correct upon unpacking. One trick would be to make sure that archives with sub-texts did not have too many more errors than archives without sub-texts (which means introducing random "errors" to all transmitted archives an attacker might intercept).

Natural language text. Natural language is extemely free-form, and apparently an excellent format to embed a sub-channel in. Normal texts contain all sorts of spacing variations, word-choices, types, and other "random" features. But then, a too-obvious sub-channel encoding strategy is easy to detect. Sure people make typos, but not in uniformly 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 sub-text.

Cryptanalysis of Digital Steganography I

How good a sub-channel encoding strategy is is simply a matter of how well it prevent an attacker from proving the existence of the sub-channel. Of course, another desirable feature of a sub-channel is the ability to embed more, rather than less, bandwidth in it. Sometimes a couple bits of sub-text are sufficient for a purpose; but most of the time you would like to be able 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 a sub-channel 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 for it. The key in maintaining the invisibility of a sub-channel is making sure that the *distribution* and *pattern* of sub-channel bits closely matches those in a typical file of the same format.

In many cases, the expected distribution of pre-encoded sub-channels will be uniform and stochastic, but not always. You have to look at whether there is a bias towards 1 (or 0) in the pre-encoded sub-channel 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 sub-channels. A good first step is to extract a large number of pre-encoded sub-channels, 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 II

Pure plaintext messages are absolutely *terrible* candidates for sub-channel encoding. A bit pattern that works out to the ASCII sequence "Secret meeting at 6 p.m." 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 sub-text that begins with "PK" does not look like random data (e.g., PKZip header bytes). The best choice is usually to compress a plaintext first (mostly just to save on a limited sub-channel 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. If you use a symmetric key, this requires a seperate key-negotiation out-of-channel; but use of public-key systems can avoid this need.

The absolutely most important design issue in creating steganographic sub-channels is: **Don't use stock files!** If you use files that to which an attacker has access to the original copy, a simple binary comparison of the original with the new overt message reveals 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, and ones which you have a plausible reason for sending other than to hide sub-channel. Home videos, for example, are bulky files with lots of sub-channel bandwidth, that are unique. Of course, if you leak these original to an attacker, you have destroyed your system; and the same applies if you encode different messages to different parties based on the same original. 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 a possibility, they will differ in much more than just the sub-channel, so a binary comparison just shows them as wholly different files.

Two smaller issues are raised in the above. 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 anounces the likelihood of a sub-channel. The prior discussion of techniques for other file types might be useful in strategizing plausible files for normal transmission. The second issue was mentioned in an earlier panel. If your sub-channel encoding involves altering non-predictable data, can an attacker gain access to the 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? Maybe you want to send an intersecting record-set (either with or without a sub-channel) later on, or already have. If the alterations are inconsistent in

individual records, that provides a clue to a sub-channel (obviously, production data changes occassionally, but within some bounds).

"Exotic" Protocols

Shared Secrets I

The general idea behind **secret sharing** is that you might want to require multiple parties to cooperate in order to decrypt a certain ciphertext. It is not enough for one person to have her key, she needs some help to get at the plaintext. 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 also: e.g. if Alice uses her key, she needs Bob's help; if Carol users her key, she needs Dave's help; other combos will not work.

The simplest case of secret sharing is **secret splitting**. Under this protocol, it requires 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, ..., P
Generates random bit strings R{1}, R{2}, ..., R{N-1}, or length
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 R's. Either way, M can only be constructed by XOR'ing back together the information given to every person. This works exactly the same way as a one-time pad, and has the same degreee of absolute security (it is subject to bad random numbers and human weaknesses, but those contravene the explicit protocol).

Shared Secrets II

Secret splitting is simple and provably secure. It also has some limitations. If any one party loses her 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 also). Furthermore, there are a number of ways by which a malicious party who either genuinely knows a secret share or pretends to, can find other persons' portions without revealing her own 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 look at one (m,n)-threshold scheme, 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 the 256-bits, while messages themselves might well be multiple megabytes in size. The math in most protocols is computationally intractible for the numbers represented by huge files, but reasonable sized keys can act as good proxies to the actual secret message.

Shared Secrets III

The **LaGrange Interpolation Polynomial Scheme** is an easy to understand (m,n)-threshold scheme for secret sharing. The Resources section can lead a user to 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.

- Generate a random prime, **p**, larger than M.
- Generate n-1 random integers, R{1}, R{2}, ..., R{n-1}, each less than p.
- Let **F(x)** be a polynomial in a finite field, defined by:

  F(x) = (R{1}*x^n + R{2}*x^(n-1) + ... + R{n-1}*x + M) mod p
- 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).
- 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).
- Destroy R{1}, R{2}, ..., R{n-1}.
- Destroy or hide M.

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

```
k{l} = (C{1}*x{l}^n + C{2}*x{l}^(n-1) + ... + C{n-1}*x{l} + M)
```

Since these linear equations have n unknowns, C{1}...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, having less than n secret sharers cooperate, even combined with *infinite* computing power, does not allow revelation of M. Without the n'th sharer participating, any possible M (of length less than p) is just as consistent with the (less than n) equations as any other!

Key Escrow I

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 times that the issue comes up is in contexts the author finds undesirable and quite disturbing: providing a (maybe circumscribed) back-door to "secure" communications to a government/police agency and/or to corporate employers. Crytography is a technology that cannot be fully considered apart from its political implications.

However, legitimate reasons for **key escrow** can be imagined also. It may happen that you would like certain people (maybe cooperating in certain ways) to be able to access your secured communications in the even you are no longer able to divulge them yourself (or do not wish to require your effort, given certain circumstances obtain). Two techniques are useful for key escrow goals (either singly or jointly): multiple recipient lists and secret sharing of keys.

Key Escrow II

Tutorial users will probably be aware that most concrete public-key encryption systems actually use symmetric "session keys" to encrypt messages, and the 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 one could send the same entire encrypted message to multiple parties, it might be easier simply to attach multiple versions of the encrypted session key, and allow general distribution. In the concrete, this might look like:

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

Either Alice of Bob can determine S from C. And once they have S, they can decrypt M. Other parties than Alice and Bob have no accesss to S or M (although C *does* use E with three keys over two messages, so this provides a bit of extra ciphertext for attack). A nice property of C is that it is not much bigger than E{Ka}(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 the directly intended recipient, but Bob should be able to get access to M if he needs to (and at his own discretion), this scheme gives Bob an "escrow key." For that matter, we could just send E{Ka}(S) to Bob, and forgo sending E{S}(M) to him at all immediately; this would make sense if he has access to Alice's stored (encrypted files), but not to her key. One can imagine these arrangements might make sense if you wish for an employer to be able to access employees messages should the employees quit (or die, or forget passwords). Of course, it leaves decryption at the employer's discretion (but that might be appropriate for company-related correspondences).

Key Escrow III

The second technique likely to be involved in key escrow 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 if at least five of her ten friends think it appropriate to decrypt her messages, that would be OK (perhaps she is worried about disposition of her secret inventions after her death; or maybe just about the danger she will lose her original private key). In government proposals, the same structure is suggested, wherein in the presence of a warrant multiple non-government agencies would disclose shared-secret keys of citizens. The latter case is politically

worrying, but the cryptographic issue is the same for both cases.

Alice can use a (5,10)-threshold scheme to divide her key among her ten 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 was mentioned earlier, using a threshold scheme for key escrow is consistent with using session keys; depending on the requirment, it might be a message session key rather than Alice' long-term private key that gets distributed in such a scheme.

Zero-Knowledge Proofs I

For this author, probably the most surprising thing cryptography can accomplish is **zero-knowledge proofs**. The idea in a zero-knowledge proof is to prove that you have a certain piece of knowledge, without revealing the content of that knowledge to an interlocuter. The purpose of a zero-knowledge proof is 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 hah! Now I know the key and the message also."
- Alice: "Ooops."

Alice really took a bad approach here, since she failed to keep the confidential message confidential. And she even gave away the key while she was at it (she could have done slightly less badly 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 us 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 II

Zero-knowledge proofs are generalizable 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 the broadest scope exceeds this tutorial, and we will just look at one case (others are similar in form)

Graph isomorphism is a *hard* problem; that is to say, it is NP-complete. Or in other words still, it is one of those problems that can 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 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. What isomorphism means is the following:

- 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 [g1,g2] is an edge in G, [F(g1),F(g2)] 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.

In short-and-sweet form, what this boils down to is that if someone tells you she has two isomorphic graphs with enough thousands of vertices and edges, it is because she constructed the graphs to be isomorphic... not because she 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 us see why all this is important.

Zero-Knowledge Proofs III

Suppose that Peggy claims to know an isomorphism between graphs G and H. In practice this means that she 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 "Peggy is the person who knows the isomorphism of G and H." Obviously, just showing the isomorphism directly allows any observer to thereafter pretend he is Peggy, so that is no good.

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

- 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.
- Peggy gives I to Victor.
- 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 (if he got both, he would have the isomorphism proof of G and H himself).
- Peggy provides the proof requested by Victor.

So far so good. What has this shown. If a Peggy-imposter *did not* know the isomorphism of G and H, the best she can 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. Or alternately, a Peggy-imposter could try to pass off an I she constructed from H, and hope the opposite. But either way, a Peggy-imposter has a 50% chance of getting "caught" by the protocol above.

Victor probably does not find 1/2 confidence sufficient for Peggy to prove she knows the isomorphism, however. 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, 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 tiny amount). As many times as 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.

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 cmpression, this is a good starting point.

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 versitile, 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.