Introduction to cryptology: Pt. 2

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

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

Section 1. Introduction to the tutorial

Navigation

Navigating through the tutorial is easy:

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

Is this tutorial right for you?

This tutorial (along with the others in this series) targets programmers who want to familiarize themselves with cryptology, its techniques, its mathematical and conceptual basis, and its lingo. The ideal user of this tutorial has encountered various descriptions of cryptographic systems and general claims about the security or insecurity of particular software and systems, but may not entirely understand 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).

Part 1 of this three-part tutorial, here on *developerWorks*, introduced the very basic concepts of cryptography (for example, what encryption is and what keys are). Part 1 also covered the basic notions of cryptanalysis -- at least enough to help you understand some typical attacks and what it is to break a protocol.

Part 2 (this tutorial) introduces readers to intermediate cryptographic concepts. Users should feel comfortable with these introductory notions (Part 1 is a good starting point for those not already familiar with these concepts).

In this tutorial, I'll cover cryptographic algorithms and protocols. The goal is not to cover the coding and detailed workings of specific algorithms and protocols, but rather to give users the conceptual background necessary to understand and construct cryptographic algorithms and protocols. Upon completing this tutorial, you will have most of the building blocks needed to consider cryptography.

What tools use cryptography?

Some form of cryptography can be found nearly everywhere in computer technology. Popular standalone programs, like PGP and GPG, aid in securing communications. Web browsers and other network client programs implement cryptographic layers in their channels. Drivers and programs exist to secure files on disk and control access thereto. Some commercial programs use cryptographic mechanisms to limit where they can be installed and how they can be used. Basically, every time you find a need to control the access and usage of computer programs or digital data, cryptographic algorithms are important elements in the protocol for use of these programs or data.

Protocols and algorithms

One particular notion introduced in Part 1 of this tutorial 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 a strong and appropriate algorithm is an important element of a strong protocol, but it is not sufficient by itself. Section 2 of this tutorial mostly addresses how cryptographic algorithms work; section 3 reviews 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 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 upon the encryption of the plain text stream leading up to this bit (byte).

In contrast to stream ciphers, block ciphers require an entire block of plain text input before they will 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 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.)

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. Symmetric encryption algorithms

Getting started

The popular symmetric encryption algorithms in use today have a lot more in common, on a conceptual level, than they have differences between them. A certain set of basic operations and frameworks are known to stand on solid mathematical ground and have withstood years of cryptanalysis. As such, all the general remarks of this section apply almost equally well to any symmetric algorithm you might find yourself working with. Specific details can be found in the documentation associated with a specific algorithm.

Using the right basic operations and frameworks does not guarantee a strong algorithm. There are many subtle ways to put things together that introduce significant vulnerabilities to attack. Understanding this tutorial is not enough to allow you to invent your own strong algorithm; however, "rolling your own" is not a useful goal for most programmers anyway. There are so many well-tested algorithms out there that you are almost surely better off starting with them and worrying only about implementation. This tutorial will allow you to understand the logic behind what you are implementing.

Diffusion and substitution - general

Every modern symmetric encryption algorithm finds ways to work in two basic operations: *diffusion* and *substitution*. That is, the content of a cipher text both replaces the bits and bytes of the plain text with different bits and bytes (substitution), and moves those replaced bits and bytes to different locations in the cipher text (diffusion).

Unlike old-fashioned hand-calculated ciphers, modern algorithms inevitably operate primarily at a bit level. Modern algorithms usually do involve some transforms at the nibble, byte, word, or block level; but these larger transforms usually break down and rearrange the bits within these blocks in other parts of the algorithm.

Most modern algorithms produce cipher texts of exactly the same size as the original plain text. Maybe a bit or two here and there is added for identification or error correction, but overall, bits of plain text have a one-to-one relationship with bits of cipher text.

A good algorithm is one that is entirely key-dependent, whether 1's in the plain text are represented by 0's or by 1's in the cipher text. In addition, with such algorithms one cannot quite tell where in the cipher text to find a particular diffused plain text bit (except by knowing the key, and thereby figuring out just where bits have been pushed).

Diffusion and substitution - Gnosis cipher, part 1

Let's have some fun by developing a very simple algorithm that utilizes diffusion and substitution. It won't be all that strong, but we can understand it easily. Let's call it the "Gnosis cipher," after the name of my company (most anyone with a real interest in breaking it will be able to "know" all the plain text encrypted with it). For pedagogical convenience, the Gnosis cipher will operate on characters rather than on bits. This cipher should not be difficult to encode and decode with pencil and paper. An honorable mention will be given to the tutorial taker who e-mails me the most clear, interesting, or clever "crack" of the Gnosis cipher.

Diffusion and substitution - Gnosis cipher, part 2

The Gnosis cipher combines a simple substitution cipher with diffusion over a block. In fact, the cipher performs two stages: the first for substitution; the second for diffusion (good ciphers mix them together much more). The first stage simply substitutes each alphabetical character with a different character according to a keyed table. The substitution table can be represented by a string of 26 letters, with no repetitions; implicit in the table is a top row of ordered letters, for example:

```
Plain text letter: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Substitution key: BNHULVDZIXKYFMCJEWQOSARPGT
```

The substitution stage simply replaces each letter in the plain text with a different one in the keyed table. The diffusion stage operates on the intermediate substitution text in 10-character blocks. Each indexed position in a substitution text block moves to a different position in the cipher text block. Basically, we just use the same kind of table as with the substitution stage. For example:

```
Starting Index: 0123456789
Diffused Index: 5136097482
```

In each substitution text block, just look at an index position and record the character found there in the corresponding position indicated by the keyed diffusion table. Of course, an encryption will usually involve performing diffusion on multiple sequential blocks.

Reversing the algorithm is simple; just use the tables in the opposite order. A key for the algorithm consists of 26 letters followed by 10 digits, with no value repetitions of either (more compact representations of the key are easily possible). Every key will encode the same plain text to a different cipher text, which is our goal.

The Gnosis cipher is not a particularly *good* algorithm, as they go (although it is not terrible for a pencil-and-paper one). But the nice thing about it is that it implements the most important elements found in strong algorithms.

All praise XOR

One of the most widely used and useful operations in cryptographic algorithms is XOR. It is worth understanding just why XOR is such a helpful operation. XOR, as it is used in cryptography, is a bitwise numeric function with a domain of a bit pair, and the range of a result bit. (It has a slightly different, but isomorphic, use in formal logic.) Most readers are probably already familiar with XOR's result table, but let us take a look at it as a reminder:

```
XOR(1, 1) --> 0
XOR(1, 0) --> 1
XOR(0, 1) --> 1
XOR(0, 0) --> 0
```

We write the XOR function in the above table in a prefix notation, but most programming languages use an infix form. Don't worry about the notation -- the above just helps illustrate the functional nature of XOR. Also, in most programming languages, the operation called XOR (or more accurately "bitwise XOR") does more than the above table shows, but only as a generalization. That is, in an operation like C, Perl, or Python, "^" is actually the Boolean XOR of each corresponding bit in two bit-fields (or ASCII characters, integers, etc., considered as bit-fields). In principle, a language with only a single-bit XOR could simulate the bit-field XOR behavior by looping through each bit position (but computational efficiency benefits greatly from the compound bit-field XOR).

XOR in action

So just what is so special about XOR? First, suppose that we want to perform a cryptographic substitution of a plain text bit. We'd like to prevent an attacker from making any prediction (even statistical) about what the transformed value of our plain text bit will be. With XOR, a plain text 0 bit might become either a cipher text 1 or 0, depending on whether a 0 or 1 is used as the "encryption bit" (the second bit in the domain pair). Likewise for a plain text 1. Complete lack of predictability of the transformation is ideal for cryptography (unless you have access to the encryption bit).

The other crucial feature of XOR is that it is *lossless*. In fact, XOR is directly *reversible*. That is, if we have `Cb = Pb XOR Kb`, then we automatically know that `Pb = Cb XOR Kb`. That is, a reapplication of XOR to the result of a first XOR operation will return to original (plain text) bit if (and only if) the same encryption bit is used both times. Contrast this with the behavior of a different Boolean operation:

```
AND(1, 1) --> 1
AND(1, 0) --> 0
AND(0, 1) --> 0
AND(0, 0) --> 0
```

In performing an AND, we *lose* information. Suppose that we know `0 = Pb AND Kb`. It is true enough that we cannot reconstruct Pb without the encryption bit. However, if Kb happens to be 0, we cannot reconstruct Pb *even with* the encryption bit! We have simply lost any way of getting Pb back.

Snake-oil warning, part 1

As nice as XOR is in its behavior, it is not quite as nice as some folks naively (or maliciously) claim. A surprising number of real-world applications use an encryption that consists of nothing more than XOR. Mind you, there is one perfectly good case where this works -- a one-time pad (OTP). If you happen to have as much key material available as plain text to encrypt, XOR is provably perfect encryption (assuming key material is truly stochastic, that is, it has an entropy equal to its length, and therefore a rate-of-language of 1).

Many flawed algorithms take a fairly small amount of key material, XOR each plain text block with a block of key, and call that result the cipher text. This works fine for a single block. But as soon as you start reusing this same key block to encrypt multiple cipher text, things fall apart.

Snake-oil warning, part 2

Just how does naive XOR encryption show its weaknesses? Basically, it does very little to thwart frequency analysis. Suppose we use 8-byte blocks of plain text and a corresponding 8-byte long encryption key. (It doesn't make much difference if blocks are longer; the same argument applies, although this requires more known cipher text.) Find some cipher text and simply temporarily ignore everything except bytes 1, 9, 17, etc., of the cipher text.

This plain text, corresponding to this first-of-each-block cipher text, will still have the same frequency regularities as the whole plain text. And each identical plain text byte will be transformed into the same cipher text byte. So by knowing that the letter "E" makes up about 13% of plain text (assuming it is English prose), all we need do is look for a cipher text byte value occurring at this same frequency (we simplify here by ignoring case and punctuation, but this is not important for the concept). Once we find these corresponding plain text and cipher text bytes, the key byte is given instantly by `KB = PB XOR CB`, or in the example: `KB = 'E' XOR 'q'`. Once we know the key bytes, we can decipher all the cipher text values whose plain text is not an "E" without further work. Repeat the procedure for cipher text bytes [2,10,18,...] and [3,11,19,...] and so on.

Sub-algorithm rounds

Almost all modern symmetric encryption algorithms consist of multiple "rounds" of a similar sub-algorithm. Sometimes they have special operations at the beginning and/or end of the process, but most of the work consists of repeated iteration of more-or-less the same simpler sub-algorithm. Each round performs a bit of encryption all by itself, but the bits typically become even more diffused with repeated application of the sub-algorithm. In some cases, rounds are slightly different from each other in the sense that they are indexed by different key-derived values or the like. But usually the gist of the sub-algorithm remains the same.

Often cryptanalysts begin attacks on an algorithm by attacking a "simplified" version of the algorithm that has fewer rounds. Well-tested algorithms have a very carefully chosen number of rounds. It is rare that adding more rounds will weaken plausibly strong algorithms. But one thing that adding extra rounds *always* does is add more computational burden to performing the encryption. In practical uses, you always want a faster algorithm rather than slower one, all other things being equal. So the goal in designing an algorithm is to have *enough* rounds to make it secure while having *as few rounds as possible* to keep it fast.

Of course, extra rounds added to a bad starting algorithm will have a limited effect. For example, the Gnosis cipher presented above has a rather undesirable property when it comes to rounds. Performing multiple rounds of the Gnosis cipher is equivalent to performing just one round *using a different initial key*. Adding rounds has no effect whatsoever on the strength. If this is not immediately obvious, it is worthwhile to page back and review the Gnosis cipher in order to understand why this happens. The effect is similar to, but simpler than, problems and limitations encountered by earnest attempts at creating encryption algorithms.

S-boxes, part 1

A typical, although not quite universal, feature of sub-algorithms in symmetric encryption algorithms is the use of "S-boxes" (the "S" stands for "substitution"). S-boxes are in fact just functions. Rather than simply operating on individual bits (as XOR does), an S-box takes N bits of input and produces N bits of output. At their heart, S-boxes must be one-to-one functions because reversibility is required for decryption (but see the "Avalanche effects" panels for some complicating details). Each bit is still transformed to a new value, but its transformation depends on the bits around it.

Actually, once we start to look at S-boxes, the notion of tracing a specific bit as it travels through an algorithm breaks down somewhat. It is not so much that bit-one of an S-box input corresponds to bit-one of its output; rather, the whole output block corresponds (a one-to-one relation) with the whole input block. But whether each individual bit is substituted, moved, both, or neither is irrelevant. As long as the correspondence is one-to-one, we can reverse the operation during decryption.

S-boxes, part 2

The advantage of S-boxes is that they can be hand-tuned to maximize non-linearity of diffusion. Linear relations between inputs and outputs tend to make an attacker's project easier. Most basic algebraic operations that one might perform on a block fail to break up linearity in input/output relations (but some ciphers, like IDEA, nonetheless utilize solely algebraic operations and get their strength via more rounds and other strategies).

The limitation of S-boxes is basically the same as their strength. Since S-boxes are hand-tuned, they must be performed via lookups to tables rather than as fundamental operations. Practical constraints on both design costs and implementation requirements (i.e. memory usage) require that S-boxes operate on comparatively small input blocks. A lookup table with 2^6 entries or even 2^12 entries is not bad, but a lookup table with 2^32 entries is unworkable. Therefore, a number of S-boxes typically transform sub-blocks of a round input in a parallel fashion. The outputs of the collection of S-boxes are then combined and mixed using other operations.

Avalanche effects, part 1

This discussion of avalanche effects somewhat contradicts two simplifying descriptions made in previous panels. Details are always messier.

The idea of an avalanche effect is that we would like every bit in a cipher output to depend not just on the key, but also on every bit of the plain text input. Two plain texts that differ by a single bit should nonetheless produce cipher texts with no predictable similarity, even though they will be encrypted with the same key. To accomplish this goal, encryption algorithms need to recruit input bits to serve a key-like role within the algorithm. But each input bit needs to serve this key-like role in a manner that is diffused throughout the entire cipher text, not just in those cipher text bits that are nearby or that have some other simple relation to the key-like input bits.

Avalanche effects, part 2

The first caveat raised by avalanche effects is in regard to our earlier discussion of how particular plain text bits jump around to specific new positions in the cipher text. This simplification is not really correct. The information in one individual bit of plain text input is not simply moved to a new location in the cipher text, but rather one bit of information is diffused into the entire cipher text. In a very real sense, each bit of cipher text contains, for example, 1/64th bit of information about bit-one of the plain text. It may seem odd to talk about less than one bit of information, but that is fundamentally what we have with cryptographic diffusion.

The second caveat raised by avalanche effects is in regard to S-boxes. The tutorial describes S-boxes as having the same input- and output-block sizes to preserve a one-to-one relation between inputs and outputs. Well, that description is *basically* true, but may not be how you see S-boxes described elsewhere. For example, DES uses S-boxes that are often described as taking 6-bit inputs and producing 4-bit outputs. On the face of it, anything that does that is *necessarily* not fully reversible (so no decryption). But the lookup table for DES S-boxes *really does* have 64 (2^6) entries, and *really does* only have 4-bit outputs listed for each entry!

Avalanche effects, part 3

How does DES actually manage to work? The trick is that DES' S-boxes do not, *in a logical sense*, have 6-bit input blocks. Logically, DES' S-boxes take 4-bit input values; but they also accept two extra bits that *index* which of four possible S-box functions to use for the transform. So, 4-bits are transformed into a different 4-bits, but the manner in which they are transformed depends on two other key-like bits in the lookup table. We maintain reversibility just so long as we are able to find those same two index bits on our way back through the decryption.

Where do DES' S-box index bits come from? One possibility would be to derive these index bits from the key; and such would not be unreasonable in algorithm design. But what DES does instead is *borrow* copies of the bits in neighboring input blocks to the same round of parallel S-boxes, and use those as index bits. A wonderful byproduct of this element of DES' design is that it creates a very strong avalanche effect when round input bits are allowed to affect the transformations other input bits undergo.

Feistel networks

A majority of serious modern encryption algorithms use a structure called Feistel networks. This structure allows each round of an algorithm to introduce new key material, provides additional plain text diffusion, and assures that the overall algorithm remains reversible across multiple rounds (in fact, across as many rounds as you want). It is actually quite a remarkable accomplishment for such a simple structure.

In a Feistel network algorithm, the round input text of each round (including the original plain text) is divided into two equal pieces. Each round swaps the left and right half of the current block around, while introducing keyed XOR substitution in just one of the directions. That is, the output of the "ith" round of a Feistel network is determined from the output of the i-1 round by:

```
L{i} = R{i-1}
R{i} = L{i-1} XOR f(R{i-1},K{i})
```

The right side output moves to the left side with no transformation at this stage. The left side, however, moves back to the right after an XOR with some function **f**. All that constrains **f** is that its domain is the pair of the last right side output and some key material (the ith sub-key, derived from the key in some manner). The design of **f** is where the real work of the cipher goes on. For example, in the case of DES, **f** includes all the S-box transformations and a few other operations.

Why is a Feistel network reversible? Clearly, `R{i-1}` can be obtained from `L{i}` with no work at all. How do we get `L{i-1}`? That, too, is straightforward -- by the nature of XOR:

```
L{i-1} = L{i-1} XOR f(R{i-1},K{i}) XOR f(R{i-1},K{i})
```

Or, simplifying:

```
L{i-1} = R{i} XOR f(L{i},K{i})
```

As long as we can still construct the ith sub-key (which we should have no problem with if we have the key), we have accomplished the reverse algorithm of a Feistel network.

Section 3. Public-key encryption

Getting started

In 1975, Whitfield Diffie and Martin Hellman proposed a different sort of relationship between encryption and decryption keys. What if encryption and decryption were performed using two different, but related, keys? The consequences turned out to be quite radical. What we get is what is known as "public-key" or "asymmetric" algorithms.

The previous section of this tutorial discussed the general concept of public-key encryption. Here, we will hop right into a look at some actual public-key algorithms.

The most popular public-key algorithm by far is called RSA (after its creators, Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman). For years, the only real hindrance to RSA's even more widespread use was its patent status; however, that patent recently expired, and the algorithm is now in the public-domain. The El Gamal scheme runs a somewhat distant second and is based on the difficulty of calculating discrete logarithms in a finite field. RSA is based on the difficulty of factoring, and will be the only public-key algorithm discussed in greater detail in this tutorial.

How RSA works, part 1

The first thing to know about RSA is that no one knows for certain that it is secure. Or more specifically, no one knows for sure that factoring is a difficult problem, which is the assumption that RSA rests upon. In fact, no one knows for sure that factoring is the fastest way to break RSA. Then again, no one knows for sure whether `P = NP`, which largely amounts to the same thing. While theoretical certainty about the strength of RSA remains elusive, the same uncertainty applies to the most basic assumptions of computational complexity theory (i.e. `P = NP`). The security of RSA rests on assumptions that are made by almost all serious mathematicians. But for now it is one of those unproven theorems that mathematicians believe without formal proof.

How RSA works, part 2

The actual operations involved in RSA are remarkably simple, and are elementary ("elementary" in mathematics refers to methods or proofs that use only integers). To generate an RSA key pair, you must first select two primes, **p** and **q**, of the same approximate magnitude. In practice, these primes are selected by choosing random large odd numbers and eliminating composites by iterating probabilistic "primality" tests. Several such tests exist that will, on each iteration, have an X% probability of detecting a composite number (for the better tests, X > 80%). Repeating the test numerous times can eliminate composites with an arbitrarily good guarantee of correctness. The basic math of RSA does not depend on the size of **p** and **q**, but to make it secure practically, you want **p** and **q** to both be 100-200 digits long, or longer.

Several calculations are made once **p** and **q** are chosen. Schneier (see Resources), or another more extended treatment, explains the mathematical grounds in more detail; for now we just show what is calculated. First we calculate `n = p * q`. Next we select an exponent **e** that is relatively prime to (**p**-1) `*` (**q**-1). Common choices for **e** are 3, 17, and 2^16+1 (i.e., 65537) (each of these has only two 1 digits in their binary representation, which speeds exponentiation in practice). After this, we create a decryption key **d** such that:

```
(e * d) mod ((p-1)*(q-1)) = 1
```

Or, in other words:

```
d = e^-1 mod ((p-1)*(q-1))
```

How RSA works, part 3

Once **d** and **n** are calculated, as shown in the previous panel, and **e** is chosen, **p** and **q** themselves are not used, and all trace of their values should be removed to prevent unintended revelation.

Encryption and decryption are performed as follows. In the equations below, M is a number less than n (an entire message may need to be broken into multiple such M's, each one encrypted as a block). Cipher text is denoted as C, as usual:

```
C = M^e mod n
M = C^d mod n
```

The public key in this system consists of **n** and **e**. We will call this key [e, n]. You can reveal these values to the whole world. The private key consists of **d**. Keep this value to yourself, or else anyone will be able to decrypt the private messages sent to you.

How RSA works, part 4

You might wonder why an attacker cannot simply calculate **d** himself, since you have already given him `n = p * q` and **e**. Surely that is enough to reconstruct **d** with a little work! Actually, we have given away little of value. Even though an attacker has `p * q`, he does not have `(p-1)*(q-1)` , which is what he really needs. Unless he can factor **n**, there is no known easy way of deriving the latter from the former. And factoring **n** is believed to be computationally infeasible when **n** is a few hundred digits long. By the way, key lengths of RSA keys are often described by their number of bits rather than by their number of decimal digits (so you may need to divide or multiply by about three-and-a-half to convert between these ways of describing keys).

The lovely effect of this arrangement is that you need not worry at all about the security of your public key; you can send it in unsecured e-mail, or publish it in the newspaper. Anyone who sees your public key can encrypt a message that you alone can decrypt (not even the sender can decrypt it; although the sender could, of course, keep the pre-encrypted original).

An RSA example

Let's look at an example of RSA in action, albeit one with numbers far too small to resist factoring. For this example, I borrow directly from Schneier (see Resources).

Let `p = 47` and `q = 71`. We calculate `n = p * q = 3337`. The encryption exponent **e** must have no factors in common with:

```
(p-1)*(q-1) = 46 * 70 = 3220
```

We may therefore choose `e = 79` (we could have used other values equally well). We now calculate:

```
d = 79^-1 mod 3220 = 1019
```

Publish [e, n], keep **d** secret, and discard **p** and **q**. Each message we can encrypt in the example must be a number smaller than 3337. In other words, we might divide an actual plain text into 11-bit blocks and encrypt each block in the same manner. The cipher text simply concatenates the encrypted blocks (maybe padding to 12 bits to include every cipher text possible).

For example, suppose that one of our correspondent's 11-bit blocks is "01010110000"; in decimal, this is 688. Our correspondent creates a cipher text block by calculating:

```
C = 688^79 mod 3337 = 1570 = "011000100010"
```

Our correspondent sends us the message, "011000100010," and we can decrypt it by calculating:

```
M = 1570^1019 mod 3337 = 688 = "01010110000"
```

Of course, factoring 3337 is hardly an insurmountable obstacle for a determined attacker with a couple of pages of scratch paper and a pencil. By using keys hundreds of times this long, we set the bar higher than even attackers with millions of MIP-years can surmount (or so it is believed).

Signatures, part 1

An observant reader will have noticed something peculiar and useful about our RSA encryption and decryption algorithms. Remember these equations?

```
C = M^e mod n
M = C^d mod n
```

M is what we have thought of as plain text, and C is what we have thought of as cipher text. But mathematically, both M and C are just numbers between 0 and n. Therefore, we could equally well write the equations:

```
M = C^e mod n
C = M^d mod n
```

Here we get a whole new concept just by switching around C and M. Suppose Alice holds the private key **d** and wishes to assure Bob that the message M was really from her, rather than from some imposter (Mallory). All Alice needs to do is calculate `C = M^d mod n` and send C to Bob.

Signatures, part 2

Mallory can easily intercept C, and decrypt it using the public key [e,n] (which everyone knows because it has been published). But with this interception, all Mallory can do is determine M, the same thing Bob can do. Alice makes no secret of the fact she created M, in fact she is trying to *prove* she did so. Suppose Mallory also substitutes a phony C' before forwarding C' to Bob, to try to pass it off as Alice's message. Bob might well be fooled upon initial receipt, but once he tries decrypting it, Bob will find it implausible that C' originated with Alice.

The problem for Mallory is that she has no way of creating a cipher text C' that decrypts to a plausible false message. She can easily create an arbitrary, random C', but this will generally decrypt into gibberish (for widely-used key lengths, the chances of getting non-gibberish with a random C' are minuscule). And Mallory wants to substitute a *specific* false message (e.g., Mallory wants to replace Alice's message "I agree to the contract" with the false message, "I refuse to sign the contract"). Without having **d**, Mallory has no way to create a C' that will decrypt to the desired false message, nor even to any non-gibberish message at all. Once Bob decrypts the note that (purportedly) comes from Alice into something meaningful (and even topical), he can be assured it comes from Alice (or at least from someone who knows **d**; this alone cannot ensure that Mallory has not managed to steal **d** by some other means).

At its heart, what Alice has done is "digitally sign" her message. Real protocols provide additional features and improve efficiency. But RSA-in-reverse is identical in concept to all digital signature procedures.

An e-mail security protocol, part 1

RSA is an extremely useful algorithm; however, a full-fledged messaging protocol will generally involve a number of elements beyond RSA itself. Popular programs like PGP, GPG, and Lotus Notes combine a number of algorithms to form a total e-mail security system. In outline, the programs mentioned have pretty much the same elements. Let's take a look at what these elements are and how we might hypothetically build our own e-mail security protocol.

One important thing we have not yet mentioned about RSA is that it is quite slow in practice. As a mathematical abstraction, RSA looks like a good way to encrypt a message, but in real-life applications, we just do not have the CPU time to spare for RSA. Directly encrypting a message with RSA is likely to be approximately 100 times as slow in software as encrypting with DES (and DES is not a particularly speedy algorithm). By combining bits and pieces of several algorithms, we can create a practical program with desirable performance and security characteristics.

An e-mail security protocol, part 2

Just what would we like to accomplish with an e-mail security protocol? Let's list some goals:

1. We would like to enable correspondents to send private messages to us without requiring separate security procedures for key exchange (and we would like to write back to such correspondents with the same ease).
2. We would like to allow correspondents to "sign" messages and thereby provide a reasonable assurance about the true origin of messages.
3. As a corollary of the first goal, we would like to have a reasonable assurance that the keys we believe to correspond with a certain person really are associated with that person (no spoofing of identities).
4. We would like the whole protocol to make as many limited computational demands as possible while obtaining these other goals.
5. We would like the whole application or system that implements our protocol to be transparent and user-friendly.

The last goal falls outside the scope of this tutorial, but it is not something to ignore when one gets to the actual programming and design.

An e-mail security protocol, part 3

How shall we accomplish our collection of goals? We have examined all the building blocks in this tutorial; let's put them together.

Suppose that Alice wishes to send a private message to Bob, and that she has obtained Bob's public key in a way that reliably links Bob to that key. Let's call Bob's public key PUB_B. Further, let us refer to RSA encryption by the name E_RSA, and to our favorite fast and secure symmetric key algorithms as E_SYM. While we are at it, let us call our favorite pseudo-random number generator PRN. For Alice's message M that she wishes to send to Bob, she calculates and sends:

```
                    [ E_RSA{PUB_B}(PRN), E_SYM{PRN}(M) ]
```

That is to say, Alice: (1) Generates a pseudo-random "session key," which is of a moderate length (e.g. 64, 96, or 128 bits); (2) Encrypts the moderate-sized session key using (slow) RSA encryption and Bob's public key; (3) Encrypts the longer plain text M using a fast symmetric algorithm. Only a little bit of encryption with RSA is necessary: Bob is able to recover the session key because he has his own private key for RSA. And Bob is able to recover M because the protocol specifies the symmetric algorithm used to encrypt it once the session key is known.

We obtain the advantage of RSA in avoiding a requirement for externally-secured key exchange, and we also obtain the speed advantages of symmetric algorithms.

An e-mail security protocol, part 4

What if Alice wants to sign a message to Bob so that he knows with reasonable confidence that the message is genuinely from her? We have already seen that Alice, in principle, could encrypt the whole message with her RSA private key. But this is also unnecessarily CPU-intensive. She has an easier way to go.

Let H refer to our favorite cryptographic hash. And as before, let E_RSA refer to RSA encryption. Here we can refer to Alice's RSA private key as PRIV_A, and her public key as PUB_A. To send a signed message M to Bob, Alice calculates and sends:

```
[ M, S = E_RSA{PRIV_A}(H(M)) ]
```

Notice that the first part of what Alice sends is simply the plain text message itself with no transformation performed whatsoever. The message itself becomes resistant to tampering by virtue of what follows it. The "signature" to M is calculated with two operations. First, a cryptographic hash is calculated on the message. Alice need not send this hash itself to Bob, since he can calculate it equally well himself. Second, the hash is encrypted using Alice's RSA private key. The hash is of moderate length (e.g., 128 or 192 bits), so it does not take too much work to RSA encrypt. An attacker, Mallory, could invent a false message M'; and he can also easily calculate H(M'). But what Mallory cannot do is compute the RSA encryption of H(M') with Alice's private key. Suppose Mallory substitutes the message [M', S'] for Alice's message [M, S]. When Bob decrypts S' using Alice's RSA public key, he will get a value that is *not* equal to H(M'); and Bob will know the whole message [M', S'] was not authentically signed by Alice (this alone does not distinguish an attack from a signal corruption, but at least it shows that something is not right).

Suppose, while we are at it, that Alice wants to keep her signed message to Bob private as well. No problem. All Alice needs to do is substitute [M, S] for M in the encryption protocol described above. In other words:

```
[ E_RSA{PUB_B}(PRN), E_SYM{PRN}([M,E_RSA{PRIV_A}(H(M))]) ]
```

An e-mail security protocol, part 5

In the earlier parts of our e-mail security protocol, we simply assumed that Alice and Bob have a reliable way of knowing each other's RSA public keys, PUB_A and PUB_B, respectively. But a channel over which PUB_A or PUB_B might be transmitted could be subject to falsification. Let us suppose that the protocol is started by Alice sending an unsecured e-mail message to Bob that said, "Hi Bob, My RSA public key is PUB_A, Alice." Assuming Mallory can insert his own false substitute into the channel, he can send the message "Hi Bob, My RSA public key is PUB_M, Alice." (Mallory would also delete Alice's genuine message.)

The next time Bob sends a "private" message to Alice, Mallory can intercept and read it at will. In fact, if Mallory has also thought to send a message to Alice that says, "Hi Alice, My RSA public key is PUB_M, Bob," he can stay in the middle of the channel, decrypt messages from both Alice and Bob, then re-encrypt them using his own private key and/or Bob's and Alice's public keys, then send re-encrypted false messages along (either altered, or with the same M Alice or Bob wrote). Notice that Mallory now knows both PUB_A and PUB_B, while all Bob and Alice know is PUB_M, even though they falsely believe PUB_M to be one of the former things.

An e-mail security protocol, part 6

One thing Alice and Bob could do to make sure they exchange genuine public keys is to meet face-to-face (assuming neither has been coerced into deceiving while face-to-face) and tell each other their public keys. Or Alice and Bob might have a previous secure channel already established (but if so, why would they need our protocol?). However, face-to-face meetings are likely to be inconvenient. Fortunately, Alice and Bob have another option. They can rely on a trusted intermediary, Trent. In real-life transactions, notary publics, banks, escrow lawyers, police, and others play this kind of role. For our protocol, we need some sort of "authenticatable" contact with Trent, as well (and, of course, we need to trust Trent). In "public-key infrastructure" lingo, Trent is known as a "key certifying authority."

Trent has a face-to-face meeting with Alice, and exchanges PUB_A and PUB_T. Trent also has a similar meeting with Bob. In order to reliably obtain Bob's public key, Alice encrypts (but need not sign) the following message with PUB_T: "Hi Trent, what is Bob's public key?" Trent responds with a message signed with PRIV_T: "Bob's public key is PUB_B." In fact, Trent probably will not send this message to Alice personally, but will make it public knowledge via Trent's Web site, newspaper, etc. Mallory can easily get PUB_B this way, but that is fine -- he is welcome to have PUB_B. Since Trent's message is signed with PRIV_T, anyone can determine that the message really comes from Trent (assuming everyone knows they have a genuine copy of PUB_T, hence the face-to-face meetings with Trent).

Protocols like PGP actually distribute Trent's role through a "web of trust" rather than with a hierarchical authority. With a web of trust you can find that a lot of people whom you trust at least a little bit have vouched for Bob's public key. If you have had face-to-face (or other secure) contact with any of these vouchers, you can trust PUB_B.

Section 4. Conclusion

Conclusion

After finishing this tutorial, you should have a good understanding of the principles that underlie symmetric and asymmetric encryption algorithms. Tutorial users have also seen one example of constructing an actual protocol based on the these covered algorithms. In the next installment, users will discover two main things: (1) the ways that fundamental algorithms can be combined to achieve a number of "exotic" and somewhat wondrous goals; (2) some limitations and pitfalls that broad cryptographic goals are subject to. Stay tuned.

Resources and 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**.

Section 5. 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.