

Cryptography and Public Keys

Jean Mark Gawron

Linguistics

San Diego State University

gawron@mail.sdsu.edu

<http://www.rohan.sdsu.edu/~gawron>

2004 June 1

Contents

1	Subject Matter of Cryptography	2
2	Encryption	2
2.1	The solution to several problems: Encryption	2
2.2	The Decryption Key	4
2.3	The problem with keys	5
3	One-way functions	6
3.1	Fair play: A coin-tossing game problem	6
3.2	A solution: A one-way function	7
3.3	Verifiable key exchange: A practical almost one-way almost-function	8
3.4	Two true-to-life one-way functions	10
3.5	Verifiable key exchange: A one-way function protocol	10
3.6	Diffie-Hellman Protocol	12
3.7	RSA protocol	13
4	Bibliography	14

1 Subject Matter of Cryptography

The general situation: Communication over a problematic channel. The channel may not be secure (any communication may be overheard); the participants in the communication may not be trustworthy. Assurances of various sorts would be valuable.

- **Confidentiality:** assurance that only the participants will know the contents of their communication. Business or government secrets.
- **Integrity:** assurance that the message sent is exactly the message received by the intended participant.
- **Authenticity:** assurance that the message really comes from who it says it comes from
- **Non-repudiation:** assurance that the real sender cannot deny a file was sent.
- **Warrantability:** Alice commits to the truth of some proposition p but does not reveal p . Bob wants some warranty now that commits Alice to p before he commits to some important action. That warranty should be verifiable at a later time.

Generally: verification and securing of communications over channels that are not secure and cannot be assumed to transmit information reliably.

2 Encryption

2.1 The solution to several problems: Encryption

Encryption:

1. Plain text message. We call this M (for message)

2. Encoding algorithm. We call this **E**.
3. Decoding algorithm (perhaps even more important than 2!) We call this **D**.
4. Key. We call this **K**. In general the longer the key, the more secure the messages sent with it.
5. Cipher text. The result of encoding the message with the key. We call this **C**.

That's all.

Convention: $E(M,K) = C$.

Convention: $D(C,K) = M$.

Why **a key**? Why not just two secret algorithm? so that:

$$E(M)=C$$
$$D(C)=M$$

1. Short answer: Secrets always get out. Keys are easier to change than algorithms.
2. Longer answer: Kerckoff's Law.

A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.

Make the algorithms public. Let the smartest mathematicians in the world amuse themselves by trying to find a weakness. Makes for the strongest possible encryption algorithms.

Encryption solves two of the problems we started with:

1. **Confidentiality:** Only those who know the key can read the message.

2. **Authenticity:** Bob wants to verify that a message came from Alice and Alice and Bob share a secret key known only to them. Bob challenges Alice to encrypt a random message, say, “2 is fun!” Only Alice can encrypt it correctly.

Integrity, nonrepudiation, and warrantability each call for a little something more, but secret keys can enter into solving these problems as well, as we shall see.

2.2 The Decryption Key

We can and should distinguish between encryption keys and decryption keys.

One important reason is that it gives a more general way of characterizing decryption systems. The key K as a whole can then be viewed as a pair E and D , the E half of which is used for encryption and the D half for decryption.

Another very important reason for spitting K into E and D is that a mathematically nice way to design an encryption system is to use the same function for encoding and decoding. As before:

$$\begin{aligned} E(M) &= C \\ D(C) &= M \end{aligned}$$

But we have keys K_E and K_D , encryption and decryption keys respectively, and an encryption operation O , such that:

$$\begin{aligned} E(M) &= O(M, K_E) = C \\ D(C) &= O(C, K_D) = M \end{aligned}$$

We saw this setup with several kinds of ciphers.

1. **Shift cipher:** O was modular addition. K_E was some number E and K_D was $E^{-1} \pmod{26}$.

2. Affine cipher: O was linear combination using the equation:

$$C = e_1 * M + e_2$$

K_E was $[e_1, e_2]$ and K_D was $[e_1^{-1}, e_1^{-1} * e_2]$.

3. Hill cipher: O was matrix multiplication. K_E was the matrix:

$$\begin{bmatrix} e_1 & e_2 \\ e_3 & e_3 \end{bmatrix}$$

K_D was K_E^{-1} .

Mathematically this brings us in view of a kind of mathematical system called a **group**. This is a system with a 2-place operation on a set of elements. In this system there is an identity element and inverses (elements which, combined with the operation give the identity element), In terms of our decryption system, K_E and E and K_D are inverses. Hill and shift encryption systems are very naturally formulated as groups, as are transposition codes, which we have not looked at.

We shall see this setup again with RSA public key cryptography where the concept of a group and of inverses will be very important.

2.3 The problem with keys

The problem with keys is exchanging them.

Consider: The internet. Many to many communications among perfect strangers who cannot trust each other (but must in order for the American way to survive). For a community of n users,

$$\frac{n(n-1)}{1 * 2}$$

meetings are necessary to secure all pairwise communications.

The solution to the problem of key exchange is to make public key exchange possible. The idea is to only exchange K_E publicly, and to keep K_D secret.

Of course this is not possible with shift ciphers and Hill ciphers. Once he knows the encryption key for a shift cipher, any bozo who knows modular arithmetic can figure out the decryption key. Similarly for Hill ciphers and any bozo who knows matrix theory.

This is because there is an easy-to-compute function that takes you from K_E to K_D .

But what if there weren't? What if we could find a system in which getting from K_E to K_D was very difficult? But remember that

$$O(M, K_E) = C$$

What we would also want of course is that, even given C and K_E , M is very hard to compute.

This brings us to the topic of one-way functions, functions that are easy to compute in one direction but very hard in the other. Basically we are working our way toward a system in which the encryption function is a one-way function, even when

3 One-way functions

3.1 Fair play: A coin-tossing game problem

Alice and **Bob** (the main characters of our little drama) play a game of heads-or-tails.

Version 1: Alice flips coin. Bob calls it in the air. If Bob gets it right he wins. If not, Alice wins.

Issues:

- Fair coin
- Bob calls coin at right time

- Alice and Bob agree on outcome

Item 3 sounds silly.

But now add: Alice and Bob are playing over the phone.

How do we do this reliably, assuming Alice and Bob can't trust each other completely?

3.2 A solution: A one-way function

Wanted: a function f such that:

- For any x , $f(x)$ is pretty easy to compute
- For any $f(x)$, it is impossible to determine what x is.
- For any x and y , if $x \neq y$, then $f(x) \neq f(y)$

Never mind for now whether such a thing exists. Suppose we had one. Then agree on the following:

- An even x is heads
- An odd x is tails

Then we could play the game as in Table 3.2

The moral: One of the first problems we could solve with a one-way function is **warrantability**.

Step One	Alice chooses an even/odd x (H or T)
Step Two	She sends $f(x)$ to Bob
Step Three	Bob calls H or T (odd or even), essentially guessing what kind of x produced the $f(x)$ he just got.
Step Four	Alice informs Bob whether he's right by sending him x
Step Five	Bob computes $f(x)$ and verifies that it agrees with the $f(x)$ Alice sent before.

Figure 1: Head/Tails Game protocol

3.3 Verifiable key exchange: A practical almost one-way almost-function

In reality it just has to be easy to compute the one-way function in one direction and way hard in the other. And in reality it doesn't have to be a function to do much of what we want.

Here is what seems to be the simplest idea, due to Ralph Merkle¹ For the basic idea, see Figure 3.3.

Alice has to do some work to get her key. In fact for a key 20 bits long the task of breaking the code takes about 2^{20} (roughly a million) steps. But our evil eavesdropper Eve has to perform that same task for about 1,000,000 messages, so that's

$$2^{20} * 2^{20} = 2^{40}$$

steps, which is a **lot** more work.

This **might** be good enough. If Alice and Eve can both try 10K keys per second, Alice can do her decoding in about a minute on average, but Eve's decoding takes her about a year on average.

¹ The tale that is told among cryptographers about this is that Merkle took a course in Computer Security at Berkeley in 1974 [taught by Lance Hoffman]. He submitted a paper describing this idea; Hoffman couldn't understand it and Merkle dropped the course and eventually went on to become one of the people who developed the idea of public key cryptography. Retelling due to (Schneier 1996)

Step One	Bob sends Alice 1,000,000 messages of the form <i>This is message number x. Use 128-bit key K_x.</i> So each message gives Alice a different 128-bit key. He encrypts each of these messages using a randomly chosen 20-bit key [easily broken] whose identity doesn't matter.
Step Two	Alice randomly picks one of Bob's million messages and decrypts it using a brute force method based on the fact that it was encoded using a 20-bit key. She learns K_x and uses it to encrypt a message m to Bob, sends the encrypted message, $E(K_x, m)$, along with the message number x (not encrypted).
Step Three	Bob decrypts the message by looking up K_x in message x .

Figure 2: Merkle secret key protocol

Note that there is no function here, one way or otherwise. What there is is a lot of values (Bob's 1,000,000 messages) and a random choice among them. The problem is determining what the result of that random choice is. Note that that determination isn't impossible, merely computationally difficult. Note that the amount of security depends on the difficulty of the computation for Eve. If Eve works for the NSA she might have computational resources several orders of magnitude better than Alice's. Then the margin starts to dwindle away quickly.

The hassle: Bob has to do a lot of computing to generate 1,000,000 appropriate random messages, and he has to tie up a lot of bandwidth to send all of them to Alice. Alice has to do a lot of computing to get her key. Increasing the amount of security can be done by increasing the number of initial messages or increasing the size of the keys Bob sends (from 20 to, say, 25), thus increasing the amount of hassle.

Moral: A function that is **difficult** to undo would be better and could maybe secure a safer security margin without increasing the hassle.

3.4 Two true-to-life one-way functions

Here are two examples of one-way functions that might give us a better way to accomplish what Merkle's protocol was trying to accomplish in the last section.

1. Multiplication. Multiplication is easy. Factoring is hard. Factoring a number usually is taken to mean finding prime integers that divide it evenly. For very large numbers this is very hard. The idea is roughly this: Pick two primes p, q such that

$$pq = n \approx 10^{200}$$

For example:

$$p = 8273488995874738949376376607$$

$$q = 15263784900967433245564811$$

$$n = 126284756413553050978360366248406608817836065776277$$

Multiplying p and q together to get n is easy (at least for a computer). Factoring n to find p and q is very hard.

2. Finding logarithms in modular arithmetic (for some large n). Raising a number a to a power x to get result r is easy in modular arithmetic, in fact, easier than in regular arithmetic. Finding what power a has to be raised to to get r is hard. This is called the *discrete logarithm* problem.

Note that what counts here is not possibility or impossibility but the relative amount of computing.

3.5 Verifiable key exchange: A one-way function protocol

We want Alice and Bob to construct a key K they can use for their encryption. We want them to construct it by communications on an insecure channel. And of course we want K to be a secret.

Alice and Bob each contribute half the bits, we'll call Alice's bits A and Bob's bits B . Then we need some easy-to-compute function *construct-key* which computes K :

$$\text{construct-key}(x, y) = K$$

As before algorithms aren't assumed to be secret so construct-key is public.

Alice can't send A. That's insecure. So we assume a one-way function f as before.

Alice sends

$$f(x)$$

Bob sends

$$f(y)$$

The exact values of A and B are still secret because f is one-way.

What we now need is for Alice and Bob to both be able to construct the same key from what each has:

$$\begin{array}{ll} \text{Alice} & x \quad f(y) \\ \text{Bob} & y \quad f(x) \end{array}$$

So we need a version of construct-key which works even though one of its arguments has gone through f :

$$\text{alt-construct-key}(f(x), y) = \text{alt-construct-key}(f(y), x) = \text{construct-key}(x, y) = K$$

$$\begin{array}{ll} \text{Alice} & x \quad f(y) \quad \text{alt-construct-key}(f(y), x) \\ \text{Bob} & y \quad f(x) \quad \text{alt-construct-key}(f(x), y) \end{array}$$

For security we also want a kind of one-wayness for alt-construct-key. It should be impossible to construct K without having either A or B. In particular, the information sent on the insecure channel, $f(A)$ and $f(B)$, shouldn't suffice. So there should be no easy-to-compute function break-key, such that:

$$\text{break-key}(f(x), f(y)) = K$$

In fact, we have already discussed some functions. in Section 3.4, that have the properties we need. These were multiplication in regular arithmetic and raising to a power in modular arithmetic; both have which inverses (factoring and discrete logarithms) which are hard to compute.

The reason these fit the bill is that we don't need to reach the ideal of functions that are impossible to run backwards. We just need a function that, given the complexity of the particular inputs we use, will take an impractical amount of computing time to compute, say 1000 years.

In fact, the protocol we have just described is the Diffie-Hellman protocol for key exchange. The one-way function used there is the discrete power function (inverse: (discrete log function)). Details to be given in the next section.

3.6 Diffie-Hellman Protocol

First, Alice and Bob publically agree on on a large prime p and a generator g of Z_p^* . The protocol goes like this:

1. Alice chooses a large random number x and sends Bob:

$$X = g^x \pmod p$$

2. Bob chooses a large random number y and sends Alice:

$$Y = g^y \pmod p$$

3. Alice computes

$$K = Y^x \pmod p$$

4. Bob computes

$$K = X^y \pmod p$$

Note that Alice can do her key computation because she knows x , and Bob can do his because he knows y , but evil eavesdropper Eve doesn't know either of those numbers. Moreover Alice and Bob compute the same key because:

$$g^{xy} = g^{yx} = g^{xy}$$

What Eve does know is g , p , X , and Y . But in order to compute the K she has to get back from X to x or from Y to y , and in order to do that she has to find

what power g is raised to to give either X or Y . This is what is called the *discrete logarithm problem* and it is, as far as we know, very difficult to compute.

So raising to a power in modular arithmetic is our one way function. Recap- ping the original idea:

We needed Alice to send

$$f(x)$$

The f here was $g^x \bmod p$ In step 2, Bob sent

$$f(y) = g^y \bmod p$$

The exact values of A and B are still secret because f is one-way.

What we needed next was for Alice and Bob to both be able to construct the same key from what each has:

$$\begin{array}{ll} \text{Alice } x & f(y) = Y \\ \text{Bob } y & f(x) = X \end{array}$$

So we needed an alternative construct key function which works even though one of its arguments has gone through f :

$$\text{alt-construct-key}(Y, x) = \text{alt-construct-key}(X, y) = \text{construct-key}(x, y) = K$$

In the Diffie-Hellman protocol, alt-construct-key is just raising the last received message to a power mod p :

$$\begin{array}{ll} \text{alt-construct-key}(X, y) & X^y \bmod p \\ \text{alt-construct-key}(Y, x) & Y^x \bmod p \\ \text{construct-key}(x, y) & g^{xy} \bmod p \end{array}$$

And that was all there was to it, given that:

$$X^y = g^{xy} = Y^x = g^{yx} = g^{xy} \bmod p$$

3.7 RSA protocol

The RSA protocol (named for its creators, Ron Rivest, Adi Shamir, and Leonard Adleman)

1. Choose two large random numbers p and q and multiply them together to give n

$$n = p \cdot q$$

2. Randomly choose an encryption key e such that e and $(p - 1)(q - 1)$ are relatively prime.
3. Use the extended Euclidean Algorithm to compute the inverse of e mod $(p - 1)(q - 1)$. Note that e and $(p - 1)(q - 1)$ are also relatively prime. Why?

4 Bibliography

Schneier, Bruce. 1996. *Applied Cryptography*. New York: John Wiley & Sons.