Chapter 12

Diffie-Hellman

For the presentation of public-key cryptography we're going to follow the historical path. Public-key cryptography was really started by Whitfield Diffie and Martin Hellman when they published their "New Directions in Cryptography" article in 1976 [22].

So far in this book we've only talked about encryption and authentication with shared secret keys. But where do we get those shared secret keys from? If you have 10 friends you want to communicate with, you can meet them all and exchange a secret key with each of these friends for future use. But like all keys, these keys should be refreshed regularly, so then you have to meet and exchange keys all over again. A total of 45 keys are needed for a group of 10 friends. But as the group gets larger, the number of keys grows quadratically. For 100 people all communicating with each other, you need 4950 keys. This quickly becomes unmanageable.

Diffie and Hellman posed the question of whether it would be possible to do this more efficiently. Suppose you have an encryption algorithm where the encryption and decryption keys are different. You can publish your encryption key and keep your decryption key secret. Anyone can now send you an encrypted message, and only you can decrypt it. This would solve the problem of having to distribute so many different keys.

Diffie and Hellman posed the question, but they could only provide a partial answer. Their partial solution is today known as the Diffie-Hellman key exchange protocol, often shortened to DH protocol [22].

The DH protocol is a really nifty idea. It turns out that two people communicating over an insecure line can agree on a secret key in such a way that both of them receive the same key without divulging it to someone who is listening in on their conversation.

12.1 Groups

If you've read the last chapter, it won't surprise you that primes are involved. For the rest of this chapter, p is a large prime. Think of p as being 2000 to 4000 bits long. Most of our computations in this chapter will be modulo p—in many places we will not specify this again explicitly. The DH protocol uses $\mathbb{Z}_p^*$, the multiplicative group modulo p that we discussed in section 11.3.3.

Choose any g in the group and consider the numbers $1, g, g^2, g^3, \ldots$, all modulo p, of course. This is an infinite sequence of numbers, but there is only a finite set of numbers in $\mathbb{Z}_p^*$. (Remember, $\mathbb{Z}_p^*$ is the numbers $1, \ldots, p-1$ together with the operation of multiplication modulo p.) At some point the numbers must start to repeat. Let us assume this happens at $g^i = g^j$ with $i < j$. As we can do divisions modulo p, we can divide each side by g^i and get $1 = g^{j-i}$. In other words, there is a number $q := j - i$ such that $g^q = 1 \pmod{p}$. We call the smallest positive value q for which $g^q = 1 \pmod{p}$ the *order* of g. (Unfortunately, there is quite a bit of terminology associated with this stuff. We feel it is better to use the standard terminology than to invent our own words; otherwise readers will be confused later on when they read other books.)

If we keep on multiplying gs we can reach the numbers $1, g, g^2, \ldots, g^{q-1}$. After that, the sequence repeats as $g^q = 1$. We say that g is a generator and that it generates the set $1, g, g^2, \ldots, g^{q-1}$. The number of elements that can be written as a power of g is exactly q, the order of g.

One property of multiplication modulo p is that there is at least one g that generates the entire group. That is, there is at least one g value for which $q = p - 1$. So instead of thinking of $\mathbb{Z}_p^*$ as the numbers $1, \ldots, p - 1$, we can also think of them as $1, g, g^2, \ldots, g^{p-2}$. A g that generates the whole group is called a *primitive element* of the group.

Other values of g can generate smaller sets. Observe that if we multiply two numbers from the set generated by g, then we get another power of g, and therefore another element from the set. If you go through all the math, it turns out that the set generated by g is another group. That is, you can multiply and divide in this group just as you can in the large group modulo p. These smaller groups are called subgroups (see section 11.3.3). They will be important in various attacks.

There is one last thing to explain. For any element g, the order of g is a divisor of $p-1$. This isn't too hard to see. Choose g to be a primitive element. Let h be any other element. As g generates the whole group, there is an x such that $h = g^x$. Now consider the elements generated by h. These are $1, h, h^2, h^3, \ldots$ which are equal to $1, g^x, g^{2x}, g^{3x}, \ldots$. (All our computations are still modulo p, of course.) The order of h is the smallest q at which $h^q = 1$, which is the same as saying that it is the smallest q such that $g^{xq} = 1$. For any t, $g^t = 1$ is the same as saying $t = 0 \pmod{p-1}$. So q is the smallest q such that $xq = 0 \pmod{p-1}$. This happens when $q = (p-1)/\gcd(x, p-1)$. So q is obviously a factor of $p-1$.

Here's a simple example. Let's choose $p = 7$. If we choose $g = 3$ then g is a generator because $1, g, g^2, \ldots, g^6 = 1, 3, 2, 6, 4, 5$. (Again, all computations modulo p.) The element $h = 2$ generates the subgroup $1, h, h^2 = 1, 2, 4$ because $h^3 = 2^3 \bmod 7 = 1$. The element $h = 6$ generates the subgroup $1, 6$. These subgroups have sizes 3 and 2 respectively, which are both divisors of $p-1$.

This also explains parts of the Fermat test we talked about in section 11.4.1. Fermat's test is based on the fact that for any a we have $a^{p-1} = 1$. This is easy to check. Let g be a generator of $\mathbb{Z}_p^*$, and let x be such that $g^x = a$. As g is a generator of the whole group, there is always such an x. But now $a^{p-1} = g^{x(p-1)} = (g^{p-1})^x = 1^x = 1$.

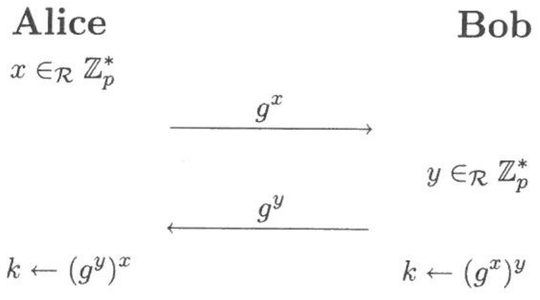
Alice
Bob
$x \in_{\mathcal{R}} \mathbb{Z}_p^*$
g^x
$y \in_{\mathcal{R}} \mathbb{Z}_p^*$
g^y
$k \leftarrow (g^y)^x$
$k \leftarrow (g^x)^y$

Figure 12.1: The original Diffie-Hellman protocol.

12.2 Basic DH

For the original DH protocol, we first choose a large prime p, and a primitive element g which generates the whole group $\mathbb{Z}_p^*$. Both p and g are public constants in this protocol, and we assume that all parties, including the attackers, know them. The protocol is shown in figure 12.1. This is one of the usual ways in which we write cryptographic protocols. There are two parties involved: Alice and Bob. Time progresses from the top to the bottom. First Alice chooses a random x in $\mathbb{Z}_p^*$, which is the same as choosing a random number in $1, \ldots, p-1$. She computes $g^x \bmod p$ and sends the result to Bob. Bob in turn chooses a random y in $\mathbb{Z}_p^*$. He computes $g^y \bmod p$ and sends the result to Alice. The final result k is defined as g^{xy}. Alice can compute this by raising the g^y she got from Bob to the power x that she knows. (High-school math: $(g^y)^x = g^{xy}$.) Similarly, Bob can compute k as $(g^x)^y$. They both end up with the same value k which they can use as a secret key.

But what about an attacker? The attacker gets to see g^x and g^y, but not x or y. The problem of computing g^{xy} given g^x and g^y is known as the Diffie-Hellman problem, or DH problem for short. As long as p and g are chosen correctly, there is no efficient algorithm to compute this—at least, there is none that we know of. The best method known is to first compute x from g^x, after which the attacker can compute k as $(g^y)^x$ just like Alice did. In the real numbers, computing x from g^x is called the logarithm function, which you find on any scientific calculator. In the finite field $\mathbb{Z}_p^*$, it is called a *discrete logarithm*, and in general the problem of computing x from g^x in a finite group is known as the discrete logarithm problem, or DL problem.

The original DH protocol can be used in many ways. We've written it as an exchange of messages between two parties. Another way of using it is to let everybody choose a random x, and publish $g^x \pmod p$ in the digital equivalent of a phone book. If Alice now wants to communicate with Bob securely, she gets g^y from the phone book, and using her x, computes g^{xy}. Bob can similarly compute g^{xy} without any interaction with Alice. This makes the system usable in settings such as e-mail where there is no direct interaction.

12.3 Man in the Middle

The one thing that DH does not protect against is the man in the middle. Look back at the protocol. Alice knows she is communicating with somebody, but she does not know whom she is communicating with. Eve can sit in the middle of the protocol and pretend to be Bob when speaking to Alice, and pretend to be Alice when speaking to Bob. This is shown in figure 12.2. To Alice, this protocol looks just like the original DH protocol. There is no way in which Alice can detect she is talking to Eve, not Bob. The same holds for Bob. Eve can keep up these pretenses for as long as she likes. Suppose Alice and Bob start to communicate using the secret key they think they have set up. All Eve needs to do is forward all the communications between Alice and Bob. Of course, Eve has to decrypt all the data she gets from Alice that was encrypted with key k, and then encrypt it again with key k' to send to Bob. She has to do the same with the traffic in the other direction, but that is not a lot of work.

With a digital phone book this attack is harder. As long as the publisher of the book verifies the identity of everybody when they send in their g^x, Alice knows she is using Bob's g^x. We'll discuss other solutions when we talk about digital signatures and PKIs later on in this book.

There is one setting where the man-in-the-middle attack can be addressed without further infrastructure. If the key k is used to encrypt a phone conversation (or a video link), Alice can talk to Bob and recognize him by

Alice		**Eve**		**Bob**
$x \in_{\mathcal{R}} \mathbb{Z}_p^*$				
	$\xrightarrow{g^x}$			
		$v \in_{\mathcal{R}} \mathbb{Z}_p^*$		
			$\xrightarrow{g^v}$	
				$y \in_{\mathcal{R}} \mathbb{Z}_p^*$
			$\xleftarrow{g^y}$	
		$w \in_{\mathcal{R}} \mathbb{Z}_p^*$		
	$\xleftarrow{g^w}$			
$k \leftarrow (g^w)^x$		$k \leftarrow (g^x)^w$		
		$k' \leftarrow (g^y)^v$		$k' \leftarrow (g^v)^y$

Figure 12.2: Diffie-Hellman protocol with a man in the middle.

his voice. Let h be a hash function of some sort. If Bob reads the first few digits of $h(k)$ to Alice, then Alice can verify that Bob is using the same key as she is. Alice can read the next few digits of $h(k)$ to Bob to allow Bob to do the same verification. This works, but only in situations where you can tie knowledge of the key k to the actual person on the other side. In most computer communications, this solution is not possible. And if Eve ever succeeds in building a speech synthesizer that can emulate Bob, it all falls apart. Finally, the biggest problem with this solution is that it requires discipline from the users. But users regularly ignore security procedures.

12.4 Pitfalls

Implementing the DH protocol can be a bit tricky. For example, if Eve intercepts the communications and replaces both g^x and g^y with the number 1, then both Alice and Bob will end up with $k = 1$. The result is a key negotiation protocol that looks as if it completed successfully, except that

Neither RSA encryption nor signature is generally safe to use on its own. The reason is that, encryption being an algebraic process, it preserves certain algebraic properties. For example, if we have a relation such as $M_1M_2 = M_3$ that holds among plaintexts, then the same relationship will hold among ciphertexts $C_1C_2 = C_3$ and signatures $Sig_1Sig_2 = Sig_3$. This property is known as a *multiplicative homomorphism* (mathematicians describe a function that preserves mathematical structure as a *homomorphism*). The homomorphic nature of raw RSA means that it doesn't meet the random oracle model definitions of public key encryption or signature.

There are a number of standards that try to stop attacks based the homomorphic mathematical structure by setting various parts of the input to the algorithm to fixed constants or to random values. Many of them have been broken. The better solutions involve processing the message using hash functions as well as random nonces and padding before the RSA primitive is applied. For example, in *optimal asymmetric encryption padding* (OAEP), we concatenate the message M with a random nonce N, and use a hash function h to combine them:

$$C_1 = M \oplus h(N)$$

$$C_2 = N \oplus h(C_1)$$

In effect, this is a two-round Feistel cipher, which uses h as its round function. The result, the combination C_1, C_2, is then encrypted with RSA and sent. The recipient then computes N as $C_2 \oplus h(C_1)$ and recovers M as $C_1 \oplus h(N)$ [88].

With signatures, things are slightly simpler. In general, it's often enough to just hash the message before applying the private key: $Sig_d = [h(M)]^d \pmod{N}$. However, in some applications, one might wish to include further data in the signature block, such as a timestamp.

5.7.2 Cryptography Based on Discrete Logarithms

While RSA is used in most Web browsers in the SSL protocol, there are other products (such as PGP) and many government systems that base public key operations on discrete logarithms. These come in a number of flavors, some using "normal" arithmetic, while others use mathematical structures called *elliptic curves*. I'll explain the normal case, as the elliptic variants use essentially the same idea but the implementation is more complex.

A *primitive root* modulo p is a number whose powers generate all the nonzero numbers mod p; for example, when working modulo 7, we find that $5^2 = 25$, which reduces to 4 (modulo 7), then we can compute 5^3 as $5^2 \times 5$ or 4×5, which is 20, which reduces to 6 (modulo 7), and so on, as shown in Figure 5.17.

Thus, 5 is a primitive root modulo 7. This means that given any y, we can always solve the equation $y = 5^x \pmod 7$; x is then called the discrete logarithm of y modulo 7. Small examples like this can be solved by inspection, but for a large random prime number p, we do not know how to do this computation. So the mapping $f : x \rightarrow g^x \pmod p$ is a one-way function, with the additional properties that $f(x + y) = f(x)f(y)$ and

$$\begin{array}{llll}
5^1 & & = 5 & (\mathrm{mod}\ 7) \\
5^2 = & 25 & \equiv 4 & (\mathrm{mod}\ 7) \\
5^3 \equiv & 4 \text{ x } 5 & \equiv 6 & (\mathrm{mod}\ 7) \\
5^4 \equiv & 6 \text{ x } 5 & \equiv 2 & (\mathrm{mod}\ 7) \\
5^5 \equiv & 2 \text{ x } 5 & \equiv 3 & (\mathrm{mod}\ 7) \\
5^6 \equiv & 3 \text{ x } 5 & \equiv 1 & (\mathrm{mod}\ 7)
\end{array}$$

Figure 5.18 Example of discrete logarithm calculations.

$f(nx) = f(x)^n$. In other words, it is a *one-way homomorphism*. As such, it can be used to construct digital signature and public key encryption algorithms.

5.7.2.1 Public Key Encryption: The Diffie-Hellman Protocol

To understand how discrete logarithms can be used to build a public key encryption algorithm, bear in mind that we want a cryptosystem that does not need the users to start off with a shared secret key. Consider the following "classical" scenario.

Imagine that Anthony wants to send a secret to Brutus, and the only communications channel available is an untrustworthy courier (say, a slave belonging to Caesar). Anthony can take the message, put it in a box, padlock it, and get the courier to take it to Brutus. Brutus could then put his own padlock on it, too, and have it taken back to Anthony. Anthony in turn would remove his padlock, and have it taken back to Brutus, who would now at last open it.

Exactly the same can be done using any encryption function that *commutes*, that is, has the property that $\{\{M\}_{KA}\}_{KB} = \{\{M\}_{KB}\}_{KA}$. Alice can take the message M and encrypt it with her key KA to get $\{M\}_{KA}$ which she sends to Bob. Bob encrypts it again with his key KB getting $\{\{M\}_{KA}\}_{KB}$. But the commutativity property means that this is just $\{\{M\}_{KB}\}_{KA}$, so Alice can decrypt it using her key KA getting $\{M\}_{KB}$. She sends this to Bob and he can decrypt it with KB, finally recovering the message M. The keys KA and KB might be long-term keys if this mechanism were to be used as a conventional public-key encryption system, or they might be transient keys if the goal were to establish a key with forward secrecy.

How can commutative encryption be implemented? If we have found values of g and p such that the discrete log problem to the base g modulo p is hard, then we can use discrete exponentiation as our encryption function. For example, Alice chooses a random number x_A, calculates g^{x_A} modulo p and sends it, together with p, to Bob. Bob likewise chooses a random number x_B and forms $g^{x_A x_B}$ modulo p, which he passes back to Alice. Alice can now remove her exponentiation: using Fermat's theorem, she calculates $g^{x_B} = (g^{x_A x_B})^{p-x_A}$ modulo p and sends it to Bob. Bob can now remove his exponentiation, too, and so finally gets hold of g. The security of this scheme depends on the difficulty of the discrete logarithm problem.

In practice, it is tricky to encode a message to be a primitive root; but there is a much simpler means of achieving the same effect. The first public key encryption scheme to be published, by Whitfield Diffie and Martin Hellman in 1976, uses $g^{x_A x_B}$ modulo p as the key to a shared key encryption system. The values x_A and x_B can be the private keys of the two parties.

Let's see how this might work to provide a public-key encryption system. The prime p and generator g are typically common to all users. Alice chooses a secret random number x_A, calculates $y_A = g^{x_A}$ and publishes it opposite her name in the company phone book. Bob does the same, choosing a random number x_B and publishing $y_B = g^{x_B}$. In order to communicate with Bob, Alice fetches y_B from the phone book, forms $y_B^{x_A}$ which is of course $g^{x_A x_B}$, and uses this to encrypt the message to Bob. On receiving it, Bob looks up Alice's public key y_A and forms $y_A^{x_B}$ which is also equal to $g^{x_A x_B}$, so he can decrypt her message.

Slightly more work is needed to provide a full solution. Some care is needed when choosing the parameters p and g; and there are several other details that depend on whether we want properties such as forward security. Variants on the Diffie-Hellman theme include the U.S. government *key exchange algorithm* (KEA) [577], used in network security products such as the Fortezza card, and the so-called Royal Holloway protocol, which is used by the U.K. government and may be used in third generation mobile phones [50].

The biggest problem with such systems is how to be sure that you've got a genuine copy of the phone book, and that the entry you're interested in isn't out of date. I'll discuss that in Section 5.7.4.

5.7.2.2 Key Establishment

Mechanisms for providing forward security in such protocols are of independent interest. As before, let the prime p and generator g be common to all users. Alice chooses a random number R_A, calculates g^{R_A} and sends it to Bob; Bob does the same, choosing a random number R_B and sending g^{R_B} to Alice; they then both form $g^{R_A R_B}$, which they use as a session key.

Alice and Bob can now use the session key $g^{R_A R_B}$ to encrypt a conversation. They have managed to create a shared secret "out of nothing." Even if an opponent had obtained full access to both their machines before this protocol was started, and thus knew all their stored private keys, then, provided some basic conditions were met (e.g., that their random-number generators were not predictable), the opponent still could not eavesdrop on their traffic. This is the strong version of the forward security property referred to in Section 5.6.2. The opponent can't work forward from knowledge of previous keys that he might have obtained. Provided that Alice and Bob both destroy the shared secret after use, they will also have backward security; an opponent who gets access to their equipment subsequently cannot work backward to break their old traffic.

But this protocol has a small problem: although Alice and Bob end up with a session key, neither of them has any idea with whom they share it.

Suppose that in our padlock protocol, Caesar has just ordered his slave to bring the box to him instead; he places his own padlock on it next to Anthony's. The slave takes the box back to Anthony, who removes his padlock, and brings the box back to Caesar who opens it. Caesar can even run two instances of the protocol, pretending to Anthony that he's Brutus and to Brutus that he's Anthony. One fix is for Anthony and Brutus to apply their seals to their locks.

The same idea leads to a middleperson attack on the Diffie-Hellman protocol unless transient keys are authenticated. Charlie intercepts Alice's message to Bob and replies to it; at the same time, he initiates a key exchange with Bob, pretending to be Alice. He ends up with a key $g^{R_A R_C}$, which he shares with Alice, and another key $g^{R_B R_C}$, which he shares with Bob. As long as he continues to sit in the middle of the network and translate the messages between them, they may have a hard time detecting that their communications are being compromised.

In one secure telephone product, the two principals would read out an eight-digit hash of the key they had generated, and check that they had the same value, before starting to discuss classified matters. A more general solution is for Alice and Bob to sign the messages that they send to each other.

Finally, discrete logarithms and their analogues exist in many other mathematical structures; thus, for example, *elliptic curve cryptography* uses discrete logarithms on an elliptic curve—a curve given by an equation such as $y^2 = x^3 + ax + b$. The algebra gets somewhat more complex, but the basic underlying ideas are the same.

5.7.2.3 Digital Signature

Suppose that the base p and the generator g (which may or may not be a primitive root) are public values chosen in some suitable way, and that each user who wishes to sign messages has a private signing key X and a public signature verification key $Y = g^X$. An *ElGamal signature scheme* works as follows: choose a message key k at random, and form $r = g^k$ (modulo p). Now form the signature s using a linear equation in k, r, the message M, and the private key X. There are a number of equations that will do; the particular one that happens to be used in ElGamal signatures is:

$$rX + sk = M \text{ modulo } p - 1$$

So s is computed as $s = (M - rX)/k$; this is done modulo $\phi(p)$. When both sides are passed through our one-way homomorphism $f(x) = g^x$ modulo p we get:

$$g^{rX} g^{sk} \equiv g^M \text{ modulo } p$$

or

$$Y^r r^s \equiv g^M \text{ modulo } p$$

An ElGamal signature on the message M consists of the values r and s, and the recipient can verify it using the above equation.

A few details need to be sorted out to get a functional digital signature scheme. For example, bad choices of p or g can weaken the algorithm; and we will want to hash the message M using a hash function so that we can sign messages of arbitrary length, and so that an opponent can't use the algorithm's algebraic structure to forge signatures on messages that were never signed. Having attended to these details and applied one or two optimisations, we get the *Digital Signature Algorithm* (DSA) which is a U.S. standard and widely used in government applications.