Chapter 10

Public key tools

We begin our discussion of public-key cryptography by introducing several basic tools that will be used in the remainder of the book. The main applications for these tools will emerge in the next few chapters where we use them for public-key encryption, digital signatures, and key exchange. Since we use some basic algebra and number theory in this chapter, the reader is advised to first briefly scan through Appendix A.

We start with a simple toy problem: generating a shared secret key between two parties so that a passive eavesdropping adversary cannot feasibly guess their shared key. The adversary can listen in on network traffic, but cannot modify messages en-route or inject his own messages. In a later chapter we develop the full machinery needed for key exchange in the presence of an active attacker who may tamper with network traffic.

At the onset we emphasize that security against eavesdropping is typically not sufficient for real world-applications, since an attacker capable of listening to network traffic is often also able to tamper with it; nevertheless, this toy eavesdropping model is a good way to introduce the new public-key tools.

10.1 A toy problem: anonymous key exchange

Two users, Alice and Bob, who never met before talk on the phone. They are worried that an eavesdropper is listening to their conversation and hence they wish to encrypt the session. Since Alice and Bob never met before they have no shared secret key with which to encrypt the session. Thus, their initial goal is to generate a shared secret unknown to the adversary. They may later use this secret as a session-key for secure communication. To do so, Alice and Bob execute a protocol where they take turns in sending messages to each other. The eavesdropping adversary can hear all these messages, but cannot change them or inject his own messages. At the end of the protocol Alice and Bob should have a secret that is unknown to the adversary. The protocol itself provides no assurance to Alice that she is really talking to Bob, and no assurance to Bob that he is talking to Alice — in this sense, the protocol is "anonymous."

More precisely, we model Alice and Bob as communicating machines. A **key exchange protocol** P is a pair of probabilistic machines (A, B) that take turns in sending messages to each other. At the end of the protocol, when both machines terminate, they both obtain the same value k. A **protocol transcript** T_P is the sequence of messages exchanged between the parties in one execution of the protocol. Since A and B are probabilistic machines, we obtain a different transcript every time we run the protocol. Formally, the transcript T_P of protocol P is a random variable, which is a function of the random bits generated by A and B. The eavesdropping adversary $\mathcal{A}$ sees the entire transcript T_P and its goal is to figure out the secret k. We define security of a key exchange protocol using the following game.

Attack Game 10.1 (Anonymous key exchange). For a key exchange protocol $P = (A, B)$ and a given adversary $\mathcal{A}$, the attack game runs as follows.

- The challenger runs the protocol between A and B to generate a shared key k and transcript T_P. It gives T_P to $\mathcal{A}$.
- $\mathcal{A}$ outputs a guess $\hat{k}$ for k.

We define $\mathcal{A}$'s advantage, denoted $\text{AnonKE}\mathsf{adv}[\mathcal{A}, P]$, as the probability that $\hat{k} = k$. □

Definition 10.1. *We say that an anonymous key exchange protocol P is secure against an eavesdropper if for all efficient adversaries $\mathcal{A}$, the quantity* $\text{AnonKE}\mathsf{adv}[\mathcal{A}, P]$ *is negligible.*

This definition of security is extremely weak, for three reasons. First, we assume the adversary is unable to tamper with messages. Second, we only guarantee that the adversary cannot guess k in its entirety. This does not rule out the possibility that the adversary can guess, say, half the bits of k. If we are to use k as a secret session key, the property we would really like is that k is indistinguishable from a truly random key. Third, the protocol provides no assurance of the identities of the participants. We will strengthen Definition 10.1 to meet these stronger requirements in Chapter 21.

Given all the tools we developed in Part 1, it is natural to ask if anonymous key exchange can be done using an arbitrary secure symmetric cipher. The answer is yes, it can be done as we show in Section 10.8, but the resulting protocol is highly inefficient. To develop efficient protocols we must first introduce a few new tools.

10.2 One-way trapdoor functions

In this section, we introduce a tool that will allow us to build an efficient and secure key exchange protocol. In Section 8.11, we introduced the notion of a one-way function. This is a function $F : \mathcal{X} \to \mathcal{Y}$ that is easy to compute, but hard to invert. As we saw in Section 8.11, there are a number of very efficient functions that are plausibly one-way. One-way functions, however, are not sufficient for our purposes. We need one-way functions with a special feature, called a **trapdoor**. A trapdoor is a secret that allows one to efficiently invert the function; however, without knowledge of the trapdoor, the function remains hard to invert.

Let us make this notion more precise.

Definition 10.2 (Trapdoor function scheme). *Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets. A* ***trapdoor function scheme*** *$\mathcal{T}$, defined over $(\mathcal{X}, \mathcal{Y})$, is a triple of algorithms (G, F, I), where*

- *G is a probabilistic key generation algorithm that is invoked as $(pk, sk) \xleftarrow{R} G()$, where pk is called a* ***public key*** *and sk is called a* ***secret key****.*
- *F is a deterministic algorithm that is invoked as $y \leftarrow F(pk, x)$, where pk is a public key (as output by G) and x lies in $\mathcal{X}$. The output y is an element of $\mathcal{Y}$.*

- *I is a deterministic algorithm that is invoked as $x \leftarrow I(sk, y)$, where sk is a secret key (as output by G) and y lies in $\mathcal{Y}$. The output x is an element of $\mathcal{X}$.*

Moreover, the following ***correctness property*** *should be satisfied: for all possible outputs (pk, sk) of $G()$, and for all $x \in \mathcal{X}$, we have $I(sk,\ F(pk, x)\ ) = x$.*

Observe that for every pk, the function $F(pk, \cdot)$ is a function from $\mathcal{X}$ to $\mathcal{Y}$. The correctness property says that sk is the trapdoor for inverting this function; note that this property also implies that the function $F(pk, \cdot)$ is one-to-one. Note that we do not insist that $F(pk, \cdot)$ maps $\mathcal{X}$ onto $\mathcal{Y}$. That is, there may be elements $y \in \mathcal{Y}$ that do not have any preimage under $F(pk, \cdot)$. For such y, we make no requirements on algorithm I — it can return some arbitrary element $x \in \mathcal{X}$ (one might consider returning a special reject symbol in this case, but it simplifies things a bit not to do this).

In the special case where $\mathcal{X} = \mathcal{Y}$, then $F(pk, \cdot)$ is not only one-to-one, but onto. That is, $F(pk, \cdot)$ is a *permutation on the set $\mathcal{X}$*. In this case, we may refer to (G, F, I) as a **trapdoor permutation scheme** defined over $\mathcal{X}$.

The basic security property we want from a trapdoor permutation scheme is a one-wayness property, which basically says that given pk and $F(pk, x)$ for random $x \in \mathcal{X}$, it is hard to compute x without knowledge of the trapdoor sk. This is formalized in the following game.

Attack Game 10.2 (One-way trapdoor function scheme). For a given trapdoor function scheme $\mathcal{T} = (G, F, I)$, defined over $(\mathcal{X}, \mathcal{Y})$, and a given adversary $\mathcal{A}$, the attack game runs as follows:

- The challenger computes

$$(pk, sk) \xleftarrow{\text{R}} G(), \quad x \xleftarrow{\text{R}} \mathcal{X}, \quad y \leftarrow F(pk, x)$$

  and sends (pk, y) to the adversary.

- The adversary outputs $\hat{x} \in \mathcal{X}$.

We define the adversary's advantage in inverting $\mathcal{T}$, denoted $\mathrm{OW}\mathsf{adv}[\mathcal{A}, \mathcal{T}]$, to be the probability that $\hat{x} = x$. $\Box$

Definition 10.3. *We say that a trapdoor function scheme $\mathcal{T}$ is* ***one way*** *if for all efficient adversaries $\mathcal{A}$, the quantity $\mathrm{OW}\mathsf{adv}[\mathcal{A}, \mathcal{T}]$ is negligible.*

Note that in Attack Game 10.2, since the value x is uniformly distributed over $\mathcal{X}$ and $F(pk, \cdot)$ is one-to-one, it follows that the value $y := F(pk, x)$ is uniformly distributed over the image of $F(pk, \cdot)$. In the case of a trapdoor permutation scheme, where $\mathcal{X} = \mathcal{Y}$, the value of y is uniformly distributed over $\mathcal{X}$.

10.2.1 Key exchange using a one-way trapdoor function scheme

We now show how to use a one-way trapdoor function scheme $\mathcal{T} = (G, F, I)$, defined over $(\mathcal{X}, \mathcal{Y})$, to build a secure anonymous key exchange protocol. The protocol runs as follows, as shown in Fig. 10.1:

- Alice computes $(pk, sk) \xleftarrow{\text{R}} G()$, and sends pk to Bob.

- Upon receiving pk from Alice, Bob computes $x \xleftarrow{\text{R}} \mathcal{X}, y \leftarrow F(pk, x)$, and sends y to Alice.

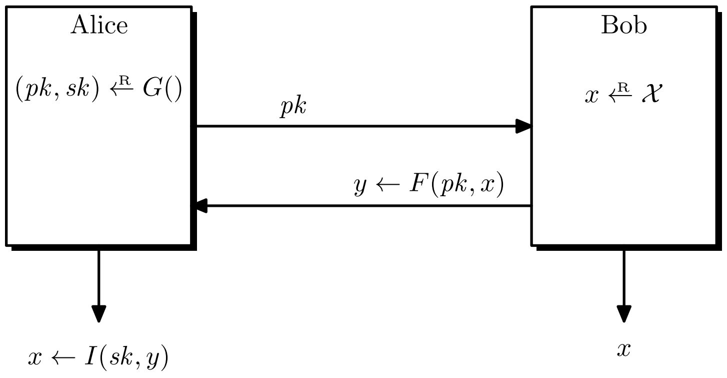

Figure 10.1: Key exchange using a trapdoor function scheme

- Upon receiving y from Bob, Alice computes $x \leftarrow I(sk, y)$.

The correctness property of the trapdoor function scheme guarantees that at the end of the protocol, Alice and Bob have the same value x — this is their shared, secret key. Now consider the security of this protocol, in the sense of Definition 10.1. In Attack Game 10.1, the adversary sees the transcript consisting of the two messages pk and y. If the adversary could compute the secret x from this transcript with some advantage, then this very same adversary could be used directly to break the trapdoor function scheme, as in Attack Game 10.2, with exactly the same advantage.

10.2.2 Mathematical details

We give a more mathematically precise definition of a trapdoor function scheme, using the terminology defined in Section 2.4.

Definition 10.4 (Trapdoor function scheme). *A* ***trapdoor function scheme*** *is a triple of efficient algorithms (G, F, I) along with families of spaces with system parameterization P:*

$$\mathbf{X} = \{\mathcal{X}_{\lambda,\Lambda}\}_{\lambda,\Lambda}, \mathbf{Y} = \{\mathcal{Y}_{\lambda,\Lambda}\}_{\lambda,\Lambda}.$$

As usual, $\lambda \in \mathbb{Z}_{\geq 1}$ is a security parameter and $\Lambda \in \text{Supp}(P(\lambda))$ is a domain parameter. We require that

1. *$\mathbf{X}$ is efficiently recognizable and sampleable.*
2. *$\mathbf{Y}$ is efficiently recognizable.*
3. *G is an efficient probabilistic algorithm that on input λ, Λ, where $\lambda \in \mathbb{Z}_{\geq 1}$, $\Lambda \in \text{Supp}(P(\lambda))$, outputs a pair (pk, sk), where pk and sk are bit strings whose lengths are always bounded by a polynomial in λ.*
4. *F is an efficient* deterministic *algorithm that on input λ, Λ, pk, x, where $\lambda \in \mathbb{Z}_{\geq 1}$, $\Lambda \in \text{Supp}(P(\lambda))$, $(pk, sk) \in \text{Supp}(G(\lambda, \Lambda))$ for some sk, and $x \in \mathcal{X}_{\lambda,\Lambda}$, outputs an element of $\mathcal{Y}_{\lambda,\Lambda}$.*

5. *I is an efficient* deterministic *algorithm that on input λ, Λ, sk, y, where $\lambda \in \mathbb{Z}_{\geq 1}$, $\Lambda \in$* Supp$(P(\lambda))$*, $(pk, sk) \in$* Supp$(G(\lambda, \Lambda))$ *for some pk, and $y \in \mathcal{Y}_{\lambda,\Lambda}$, outputs an element of $\mathcal{X}_{\lambda,\Lambda}$.*

6. *For all $\lambda \in \mathbb{Z}_{\geq 1}$, $\Lambda \in$* Supp$(P(\lambda))$*, $(pk, sk) \in$* Supp$(G(\lambda, \Lambda))$*, and $x \in \mathcal{X}_{\lambda,\Lambda}$, we have $I(\lambda, \Lambda; sk, F(\lambda, \Lambda; pk, x)) = x$.*

As usual, in defining the one-wayness security property, we parameterize Attack Game 10.2 by the security parameter λ, and the advantage $\mathrm{OW}\mathsf{adv}[\mathcal{A}, \mathcal{T}]$ is actually a function of λ. Definition 10.3 should be read as saying that $\mathrm{OW}\mathsf{adv}[\mathcal{A}, \mathcal{T}](\lambda)$ is a negligible function.

10.3 A trapdoor permutation scheme based on RSA

We now describe a trapdoor permutation scheme that is plausibly one-way. It is called RSA after its inventors, Rivest, Shamir, and Adleman. Recall that a trapdoor permutation is a special case of a trapdoor function, where the domain and range are the same set. This means that for every public-key, the function is a permutation of its domain, which is why we call it a trapdoor permutation. Despite many years of study, RSA is essentially the only known reasonable candidate trapdoor permutation scheme (there are a few others, but they are all very closely related to the RSA scheme).

Here is how RSA works. First, we describe a probabilistic algorithm RSAGen that takes as input an integer $\ell > 2$, and an odd integer $e > 2$.

```
RSAGen($\ell, e$) :=
    generate a random $\ell$-bit prime $p$ such that $\gcd(e, p-1) = 1$
    generate a random $\ell$-bit prime $q$ such that $\gcd(e, q-1) = 1$ and $q \neq p$
    $n \leftarrow pq$
    $d \leftarrow e^{-1} \bmod (p-1)(q-1)$
    output $(n, d)$.
```

To efficiently implement the above algorithm, we need an efficient algorithm to generate random ℓ-bit primes. This is discussed in Appendix A. Also, we use the extended Euclidean algorithm (Appendix A) to compute $e^{-1} \bmod (p-1)(q-1)$. Note that since $\gcd(e, p-1) = \gcd(e, q-1) = 1$, it follows that $\gcd(e, (p-1)(q-1)) = 1$, and hence e has a multiplicative inverse modulo $(p-1)(q-1)$.

Now we describe the RSA trapdoor permutation scheme $\mathcal{T}_{\mathrm{RSA}} = (G, F, I)$. It is parameterized by fixed values of ℓ and e.

- Key generation runs as follows:

$$\begin{aligned} G() := \quad & (n,d) \xleftarrow{\mathrm{R}} \mathrm{RSAGen}(\ell, e), \quad pk \leftarrow (n, e), \quad sk \leftarrow (n, d) \\ & \text{output } (pk, sk). \end{aligned}$$

- For a given public key $pk = (n, e)$, and $x \in \mathbb{Z}_n$, we define $F(pk, x) := x^e \in \mathbb{Z}_n$.

- For a given secret key $sk = (n, d)$, and $y \in \mathbb{Z}_n$, we define $I(sk, y) := y^d \in \mathbb{Z}_n$.

Note that although the encryption exponent e is considered to be a fixed system parameter, we also include it as part of the public key pk.

A technicality. For each fixed $pk = (n, e)$, the function $F(pk, \cdot)$ maps $\mathbb{Z}_n$ into $\mathbb{Z}_n$; thus, the domain and range of this function actually vary with pk. However, in our definition of a trapdoor permutation scheme, the domain and range of the function are not allowed to vary with the public key. So in fact, this scheme does not quite satisfy the formal syntactic requirements of a trapdoor permutation scheme. One could easily generalize the definition of a trapdoor permutation scheme, to allow for this. However, we shall not do this; rather, we shall state and analyze various schemes based on a trapdoor permutation scheme as we have defined it, and then show how to instantiate these schemes using RSA. Exercise 10.24 explores an idea that builds a proper trapdoor permutation scheme based on RSA.

Ignoring this technical issue for the moment, let us first verify that $\mathcal{T}_{\text{RSA}}$ satisfies the correctness requirement of a trapdoor permutation scheme. This is implied by the following:

Theorem 10.1. *Let $n = pq$ where p and q are distinct primes. Let e and d be integers such that $ed \equiv 1 \pmod{(p-1)(q-1)}$. Then for all $x \in \mathbb{Z}$, we have $x^{ed} \equiv x \pmod{n}$.*

Proof. The hypothesis that $ed \equiv 1 \pmod{(p-1)(q-1)}$ just means that $ed = 1 + k(p-1)(q-1)$ for some integer k. Certainly, if $x \equiv 0 \pmod{p}$, then $x^{ed} \equiv 0 \equiv x \pmod{p}$; otherwise, if $x \not\equiv 0 \pmod{p}$, then by Fermat's little theorem (Appendix A), we have

$$x^{p-1} \equiv 1 \pmod{p},$$

and so

$$x^{ed} \equiv x^{1+k(p-1)(q-1)} \equiv x \cdot \left(x^{(p-1)}\right)^{k(q-1)} \equiv x \cdot 1^{k(q-1)} \equiv x \pmod{p}.$$

Therefore,

$$x^{ed} \equiv x \pmod{p}.$$

By a symmetric argument, we have

$$x^{ed} \equiv x \pmod{q}.$$

Thus, $x^{ed} - x$ is divisible by the distinct primes p and q, and must therefore be divisible by their product n, which means

$$x^{ed} \equiv x \pmod{n}. \quad \square$$

So now we know that $\mathcal{T}_{\text{RSA}}$ satisfies the correctness property of a trapdoor permutation scheme. However, it is not clear that it is one-way. For $\mathcal{T}_{\text{RSA}}$, one-wayness means that there is no efficient algorithm that given n and x^e, where $x \in \mathbb{Z}_n$ is chosen at random, can effectively compute x. It is clear that if $\mathcal{T}_{\text{RSA}}$ is one-way, then it must be hard to factor n; indeed, if it were easy to factor n, then one could compute d in exactly the same way as is done in algorithm RSAGen, and then use d to compute $x = y^d$.

It is widely believed that factoring n is hard, provided ℓ is sufficiently large — typically, ℓ is chosen to be between 1000 and 1500. Moreover, the only known efficient algorithm to invert $\mathcal{T}_{\text{RSA}}$ is to first factor n and then compute d as above. However, there is no known *proof* that the assumption that factoring n is hard implies that $\mathcal{T}_{\text{RSA}}$ is one-way. Nevertheless, based on current evidence, it seems reasonable to conjecture that $\mathcal{T}_{\text{RSA}}$ is indeed one-way. We state this conjecture now as an explicit assumption. As usual, this is done using an attack game.

Attack Game 10.3 (RSA). For given integers $\ell > 2$ and odd $e > 2$, and a given adversary $\mathcal{A}$, the attack game runs as follows:

- The challenger computes

$$(n, d) \xleftarrow{R} \text{RSAGen}(\ell, e), \quad x \xleftarrow{R} \mathbb{Z}_n, \quad y \leftarrow x^e \in \mathbb{Z}_n$$

  and gives the input (n, y) to the adversary.

- The adversary outputs $\hat{x} \in \mathbb{Z}_n$.

We define the adversary's advantage in breaking RSA, denoted RSA$\mathsf{adv}[\mathcal{A}, \ell, e]$, as the probability that $\hat{x} = x$. □

Definition 10.5 (RSA assumption). *We say that the RSA assumption holds for (ℓ, e) if for all efficient adversaries $\mathcal{A}$, the quantity* RSA$\mathsf{adv}[\mathcal{A}, \ell, e]$ *is negligible.*

We analyze the RSA assumption and present several known attacks on it later on in Chapter 17.

We next introduce some terminology that will be useful later. Suppose (n, d) is an output of RSAGen(ℓ, e), and suppose that $x \in \mathbb{Z}_n$ and let $y := x^e$. The number n is called an **RSA modulus**, the number e is called an **encryption exponent**, and the number d is called a **decryption exponent**. We call (n, y) an **instance** of the **RSA problem**, and we call x a **solution** to this instance of the RSA problem. The RSA assumption asserts that there is no efficient algorithm that can effectively solve the RSA problem.

10.3.1 Key exchange based on the RSA assumption

Consider now what happens when we instantiate the key exchange protocol in Section 10.2.1 with $\mathcal{T}_{\text{RSA}}$. The protocol runs as follows:

- Alice computes $(n, d) \xleftarrow{R} \text{RSAGen}(\ell, e)$, and sends (n, e) to Bob.
- Upon receiving (n, e) from Alice, Bob computes $x \xleftarrow{R} \mathbb{Z}_n$, $y \leftarrow x^e$, and sends y to Alice.
- Upon receiving y from Bob, Alice computes $x \leftarrow y^d$.

The secret shared by Alice and Bob is x. The message flow is the same as in Fig. 10.1. Under the RSA assumption, this is a secure anonymous key exchange protocol.

10.3.2 Mathematical details

We give a more mathematically precise definition of the RSA assumption, using the terminology defined in Section 2.4.

In Attack Game 10.3, the parameters ℓ and e are actually poly-bounded and efficiently computable functions of a security parameter λ. Likewise, RSA$\mathsf{adv}[\mathcal{A}, \ell, e]$ is a function of λ. As usual, Definition 10.5 should be read as saying that RSA$\mathsf{adv}[\mathcal{A}, \ell, e](\lambda)$ is a negligible function.

There are a couple of further wrinkles we should point out. First, as already mentioned above, the RSA scheme does not quite fit our definition of a trapdoor permutation scheme, as the definition of the latter does not allow the set $\mathcal{X}$ to vary with the public key. It would not be too difficult to modify our definition of a trapdoor permutation scheme to accommodate this generalization. Second, the specification of RSAGen requires that we generate random prime numbers of a given bit length. In theory, it is possible to do this in (expected) polynomial time; however, the most practical algorithms (see Appendix A) may — with negligible probability — output a number that is not a prime. If that should happen, then it may be the case that the basic correctness requirement — namely, that $I(sk, F(pk, x)) = x$ for all pk, sk, x — is no longer satisfied. It would also not be too difficult to modify our definition of a trapdoor permutation scheme to accommodate this type of generalization as well. For example, we could recast this requirement as an attack game (in which any efficient adversary wins with negligible probability): in this game, the challenger generates $(pk, sk) \overset{\text{R}}{\leftarrow} G()$ and sends (pk, sk) to the adversary; the adversary wins the game if he can output $x \in \mathcal{X}$ such that $I(sk, F(pk, x)) \neq x$. While this would be a perfectly reasonable definition, using it would require us to modify security definitions for higher-level constructs. For example, if we used this relaxed correctness requirement in the context of key exchange, we would have to allow for the possibility that the two parties end up with different keys with some negligible probability.

10.4 Diffie-Hellman key exchange

In this section, we explore another approach to constructing secure key exchange protocols, which was invented by Diffie and Hellman. Just as with the protocol based on RSA, this protocol will require a bit of algebra and number theory. However, before getting in to the details, we provide a bit of motivation and intuition.

Consider the following "generic" key exchange protocol the makes use of two functions E and F. Alice chooses a random secret α, computes $E(\alpha)$, and sends $E(\alpha)$ to Bob over an insecure channel. Likewise, Bob chooses a random secret β, computes $E(\beta)$, and sends $E(\beta)$ to Alice over an insecure channel. Alice and Bob both somehow compute a shared key $F(\alpha, \beta)$. In this high-level description, E and F are some functions that should satisfy the following properties:

1. E should be easy to compute;
2. given α and $E(\beta)$, it should be easy to compute $F(\alpha, \beta)$;
3. given $E(\alpha)$ and β, it should be easy to compute $F(\alpha, \beta)$;
4. given $E(\alpha)$ and $E(\beta)$, it should be hard to compute $F(\alpha, \beta)$.

Properties 1–3 ensure that Alice and Bob can efficiently implement the protocol: Alice computes the shared key $F(\alpha, \beta)$ using the algorithm from Property 2 and her given data α and $E(\beta)$. Bob computes the same key $F(\alpha, \beta)$ using the algorithm from Property 3 and his given data $E(\alpha)$ and β. Property 4 ensures that the protocol is secure: an eavesdropper who sees $E(\alpha)$ and $E(\beta)$ should not be able to compute the shared key $F(\alpha, \beta)$.

Note that properties 1–4 together imply that E is hard to invert; indeed, if we could compute efficiently α from $E(\alpha)$, then by Property 2, we could efficiently compute $F(\alpha, \beta)$ from $E(\alpha), E(\beta)$, which would contradict Property 4.

To make this generic approach work, we have to come up with appropriate functions E and F. To a first approximation, the basic idea is to implement E in terms of exponentiation to some fixed base g, defining $E(\alpha) := g^{\alpha}$ and $F(\alpha, \beta) := g^{\alpha\beta}$. Notice then that

$$E(\alpha)^{\beta} = (g^{\alpha})^{\beta} = F(\alpha, \beta) = (g^{\beta})^{\alpha} = E(\beta)^{\alpha}.$$

Hence, provided exponentiation is efficient, Properties 1–3 are satisfied. Moreover, if Property 4 is to be satisfied, then at the very least, we require that taking logarithms (i.e., inverting E) is hard.

To turn this into a practical and plausibly secure scheme, we cannot simply perform exponentiation on ordinary integers since the numbers would become too large. Instead, we have to work in an appropriate finite algebraic domain, which we introduce next.

10.4.1 The key exchange protocol

Suppose p is a large prime and that q is a large prime dividing $p-1$ (think of p as being very large random prime, say 2048 bits long, and think of q as being about 256 bits long).

We will be doing arithmetic mod p, that is, working in $\mathbb{Z}_p$. Recall that $\mathbb{Z}_p^*$ is the set of nonzero elements of $\mathbb{Z}_p$. An essential fact is that since q divides $p-1$, $\mathbb{Z}_p^*$ has an element g of order q (see Appendix A). This means that $g^q = 1$ and that all of the powers g^a, for $a = 0, \dots, q-1$, are distinct. Let $\mathbb{G} := \{g^a : a = 0, \dots, q-1\}$, so that $\mathbb{G}$ is a subset of $\mathbb{Z}_p^*$ of cardinality q. It is not hard to see that $\mathbb{G}$ is closed under multiplication and inversion; that is, for all $u, v \in \mathbb{G}$, we have $uv \in \mathbb{G}$ and $u^{-1} \in \mathbb{G}$. Indeed, $g^a \cdot g^b = g^{a+b} = g^c$ with $c := (a+b) \bmod q$, and $(g^a)^{-1} = g^d$ with $d := (-a) \bmod q$. In the language of algebra, $\mathbb{G}$ is called a subgroup of the group $\mathbb{Z}_p^*$.

For every $u \in \mathbb{G}$ and integers a and b, it is easy to see that $u^a = u^b$ if $a \equiv b \bmod q$. Thus, the value of u^a depends only on the residue class of a modulo q. Therefore, if $\alpha = [a]_q \in \mathbb{Z}_q$ is the residue class of a modulo q, we can define $u^\alpha := u^a$ and this definition is unambiguous. From here on we will frequently use elements of $\mathbb{Z}_q$ as exponents applied to elements of $\mathbb{G}$.

So now we have everything we need to describe the Diffie-Hellman key exchange protocol. We assume that the description of $\mathbb{G}$, including $g \in \mathbb{G}$ and q, is a system parameter that is generated once and for all at system setup time and shared by all parties involved. The protocol runs as follows, as shown in Fig. 10.2:

1. Alice computes $\alpha \overset{\text{R}}{\leftarrow} \mathbb{Z}_q$, $u \leftarrow g^\alpha$, and sends u to Bob.
2. Bob computes $\beta \overset{\text{R}}{\leftarrow} \mathbb{Z}_q, v \leftarrow g^\beta$ and sends v to Alice.
3. Upon receiving v from Bob, Alice computes $w \leftarrow v^\alpha$
4. Upon receiving u from Alice, Bob computes $w \leftarrow u^\beta$

The secret shared by Alice and Bob is

$$w = v^\alpha = g^{\alpha\beta} = u^\beta.$$

10.4.2 Security of Diffie-Hellman key exchange

For a fixed element $g \in \mathbb{G}$, different from 1, the function from $\mathbb{Z}_q$ to $\mathbb{G}$ that sends $\alpha \in \mathbb{Z}_q$ to $g^\alpha \in \mathbb{G}$ is called the **discrete exponentiation function**. This function is one-to-one and onto, and its inverse function is called the **discrete logarithm function**, and is usually denoted Dlog_g; thus, for $u \in \mathbb{G}$, $\mathsf{Dlog}_g(u)$ is the unique $\alpha \in \mathbb{Z}_q$ such that $u = g^\alpha$. The value g is called the **base** of the discrete logarithm.

If the Diffie-Hellman protocol has any hope of being secure, it must be hard to compute α from g^α for a random α; in other words, it must be hard to compute the discrete logarithm function. There are a number of candidate group families $\mathbb{G}$ where the discrete logarithm function is believed to be hard to compute. For example, when p and q are sufficiently large, suitably chosen primes,

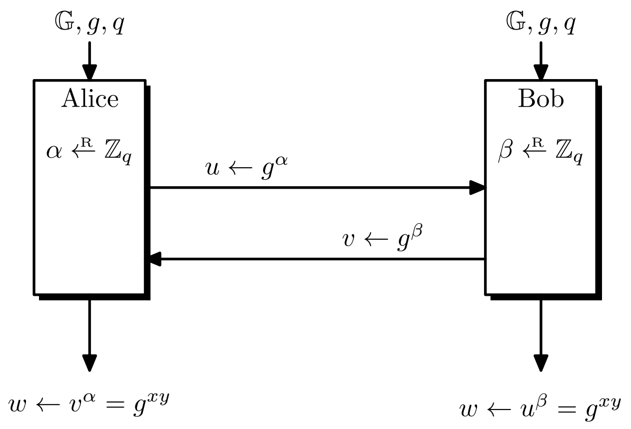
$\mathbb{G}, g, q$
$\mathbb{G}, g, q$
Alice
$\alpha \xleftarrow{R} \mathbb{Z}_q$
Bob
$\beta \xleftarrow{R} \mathbb{Z}_q$
$u \leftarrow g^{\alpha}$
$v \leftarrow g^{\beta}$
$w \leftarrow v^{\alpha} = g^{xy}$
$w \leftarrow u^{\beta} = g^{xy}$

Figure 10.2: Diffie-Hellman key exchange

the discrete logarithm function in the order q subgroup of $\mathbb{Z}_p^*$ is believed to be hard to compute (p should be at least 2048-bits, and q should be at least 256-bits). This assumption is called the **discrete logarithm assumption** and is defined in the next section.

Unfortunately, the discrete logarithm assumption by itself is not enough to ensure that the Diffie-Hellman protocol is secure. Observe that the protocol is secure if and only if the following holds:

given $g^{\alpha}, g^{\beta} \in \mathbb{G}$, where $\alpha \xleftarrow{R} \mathbb{Z}_q$ and $\beta \xleftarrow{R} \mathbb{Z}_q$, it is hard to compute $g^{\alpha\beta} \in \mathbb{G}$.

This security property is called the **computational Diffie-Hellman assumption**. Although the computational Diffie-Hellman assumption is stronger than the discrete logarithm assumption, all evidence still suggests that this is a reasonable assumption in groups where the discrete logarithm assumption holds.

10.5 Discrete logarithm and related assumptions

In this section, we state the discrete logarithm and related assumptions more precisely and in somewhat more generality, and explore in greater detail relationships among them.

The subset $\mathbb{G}$ of $\mathbb{Z}_p^*$ that we defined above in Section 10.4 is a specific instance of a general type of mathematical object known as a *cyclic group*. There are in fact other cyclic groups that are very useful in cryptography, most notably, groups based on *elliptic curves* — we shall study elliptic curve cryptography in Chapter 15. From now on, we shall state assumptions and algorithms in terms of an abstract cyclic group $\mathbb{G}$ of prime order q generated by $g \in \mathbb{G}$. In general, such groups may be selected by a randomized process, and again, the description of $\mathbb{G}$, including $g \in \mathbb{G}$ and q, is a system parameter that is generated once and for all at system setup time and shared by all parties involved.

We shall use just a bit of terminology from group theory. The reader who is unfamiliar with the concept of a group may wish to refer to Appendix A; alternatively, for the time being, the reader may simply ignore this abstraction entirely: