CHAPTER 16

Public Key Encryption and Signature Algorithms

Chapter Goals

- To present fully secure public key encryption schemes and signature schemes.
- To show how the random oracle model can be used to prove certain encryption and signature schemes are secure.
- To understand how the RSA algorithm is actually used in practice, and sketch a proof of RSA-OAEP.
- To introduce and formalize the notion of hybrid encryption, via KEMs and DEMs.
- To present two efficient KEMs, namely RSA-KEM and DHIES-KEM.
- To explain the most widely used signature algorithms, namely variants of RSA and DSA.
- To present the Cramer–Shoup encryption and signature schemes, which do not require the use of the random oracle model.

16.1. Passively Secure Public Key Encryption Schemes

In this section we present three basic passively secure encryption schemes, namely the Goldwasser–Micali encryption scheme, the ElGamal encryption scheme, and the Paillier encryption scheme.

16.1.1. Goldwasser–Micali Encryption: We have seen that RSA is not semantically secure even against a passive attack; thus it would be nice to give a system which is IND-CPA secure and is based on some factoring-like assumption. Historically the first system to meet these goals was one by Goldwasser and Micali. The scheme is not used in real-life applications, due to its inefficiency, however its simplicity means that it can help solidify ideas about to how construct (and prove secure) the systems we do use in real life.

The security of the Goldwasser–Micali encryption scheme is based on the hardness of the QUADRES problem, namely given a composite integer N and an integer e, it is hard to test whether a is a quadratic residue or not without knowledge of the factors of N. Let us recap, from Chapter 2, that the set of squares in $(\mathbb{Z}/N\mathbb{Z})^*$ is denoted by

$$Q_N = \{x^2 \pmod N : x \in (\mathbb{Z}/N\mathbb{Z})^*\},$$

and J_N denotes the set of elements with Jacobi symbol equal to plus one, i.e.

$$J_N = \left\{a \in (\mathbb{Z}/N\mathbb{Z})^* : \left(\frac{a}{N}\right) = 1\right\}.$$

The set of pseudo-squares is the difference $J_N \setminus Q_N$. For an RSA-like modulus $N = p \cdot q$ the number of elements in J_N is equal to $(p-1)\cdot(q-1)/2$, whilst the number of elements in Q_N is $(p-1)\cdot(q-1)/4$. The QUADRES problem is that given an element x of J_N, it is hard to tell whether $x \in Q_N$, whilst it is easy to tell whether $x \in J_N$ or not.

We can now explain the Goldwasser–Micali encryption system.

N.P. Smart, *Cryptography Made Simple*, Information Security and Cryptography, DOI 10.1007/978-3-319-21936-3_16

Key Generation: As a private key we take two large prime numbers $\mathfrak{sk} = (p, q)$ and then compute the public modulus $N \leftarrow p \cdot q$, and an integer $y \in J_N \setminus Q_N$. The public key is set to be $\mathfrak{pk} \leftarrow (N, y)$. The value of y is computed by the public key owner by first computing elements $y_p \in \mathbb{F}_p^*$ and $y_q \in \mathbb{F}_q^*$ such that

$$\left(\frac{y_p}{p}\right) = \left(\frac{y_q}{q}\right) = -1.$$

Then the value of y is computed from y_p and y_q via the Chinese Remainder Theorem. A value of y computed in this way clearly does not lie in Q_N, but it does lie in J_N since

$$\left(\frac{y}{N}\right) = \left(\frac{y}{p}\right) \cdot \left(\frac{y}{q}\right) = \left(\frac{y_p}{p}\right) \cdot \left(\frac{y_q}{q}\right) = (-1) \cdot (-1) = 1.$$

Encryption: The Goldwasser–Micali encryption system encrypts one bit of information at a time. To encrypt the bit b,

- $x \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$.
- $c \leftarrow y^b \cdot x^2 \pmod{N}$.

The ciphertext is then the value of c. Notice that this is very inefficient since a single bit of plaintext requires $\log_2 N$ bits of ciphertext to transmit it.

Decryption: Notice that the ciphertext c will always be an element of J_N. However, if the message bit b is zero then the value of c will be a quadratic residue, otherwise it will be a quadratic non-residue. So all the decryptor has to do to recover the message is determine whether c is a quadratic residue or not modulo N. But the decryptor is assumed to know the factors of N and so can compute the Legendre symbol

$$\left(\frac{c}{p}\right).$$

If this Legendre symbol is equal to plus one then c is a quadratic residue and so the message bit is zero. If however the Legendre symbol is equal to minus one then c is not a quadratic residue and so the message bit is one.

It is now relatively straightforward to prove that the Goldwasser–Micali encryption scheme is IND-CPA secure, assuming that the QUADRES problem is hard for RSA style moduli N of size v bits.

Theorem 16.1. *Suppose there is an adversary A against the* IND-CPA *security of the Goldwasser–Micali encryption scheme Π for moduli of size v bits, then there is an adversary B against the QUADRES problem such that*

$$\mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}CPA}}(A) = 2 \cdot \mathrm{Adv}_{v}^{\mathsf{QUADRES}}(B).$$

Proof. We describe the algorithm B which will use A as an oracle. To see this in pictures see Figure 16.1. Suppose algorithm B is given N and $j \in J_N$ and is asked to determine whether $j \in Q_N$. Algorithm B first randomizes j to form y, on the assumption that j *does not* lie in Q_N. Thus algorithm B sets $y \leftarrow j \cdot z^2 \pmod{N}$, for some $z \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$. The public key $\mathfrak{pk} \leftarrow (N, y)$ is then passed to algorithm A.

Since the Goldwasser–Micali system only encrypts bits we can assume that the **find** stage of the adversary A will simply output the two messages

$$m_0 = 0 \text{ and } m_1 = 1.$$

We now form the challenge ciphertext

$$c^* \leftarrow y^b \cdot r^2,$$

for some bit $b \leftarrow \{0,1\}$ and some random $r \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$ chosen by algorithm B. We now pass c^* to algorithm A, which will respond with its guess b' for the bit b. If $b = b'$ then algorithm B returns that j is a quadratic residue, otherwise it returns that it is not.

To analyse the probabilities we notice that if j is not a quadratic residue then this value of c^* will be a valid encryption of the message m_b. So if j is a quadratic residue then algorithm B is presenting a valid challenger to algorithm A. However, if j is not a quadratic residue then this is not a valid encryption of anything (since the public key is not even valid). Thus we have

$$\begin{aligned}\mathrm{Adv}_v^{\mathsf{QUADRES}}(B) &= \left|\Pr[b' = b | y \in Q_N] - \Pr[b' = b | y \in J_N \setminus Q_N]\right| \\ &= \left|\Pr[A \text{ wins for a valid challenger}] - \frac{1}{2}\right| \\ &= \frac{1}{2} \cdot \mathrm{Adv}_\Pi^{\mathsf{IND\text{-}CPA}}(A)\end{aligned}$$

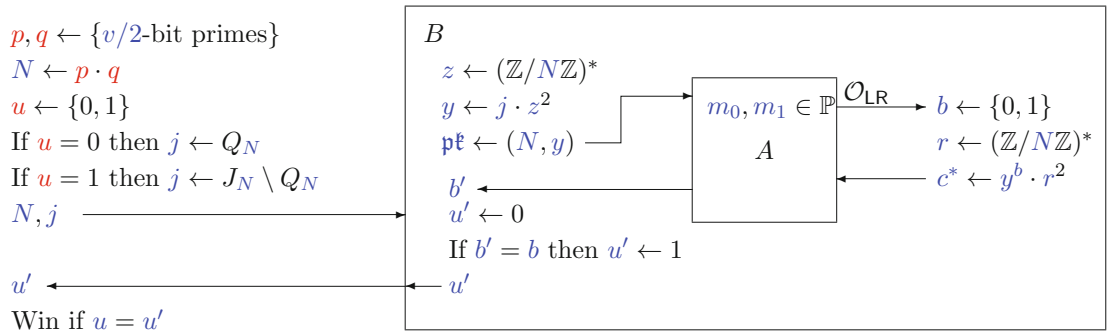

FIGURE 16.1. How B interacts with A in Theorem 16.1

□

Note that the above argument says nothing about whether the Goldwasser–Micali encryption scheme is secure against adaptive adversaries. In fact, one can show it is not secure against such adversaries.

Theorem 16.2. *The Goldwasser–Micali encryption scheme is not* IND-CCA *secure.*

PROOF. Suppose c^* is the target ciphertext and we want to determine what bit b is encrypted by c^*. Recall that $c^* = y^b \cdot x^2 \pmod N$. Now the rules of the game do not allow us to ask our decryption oracle to decrypt c^*, but we can ask our oracle to decrypt any other ciphertext. We therefore produce the ciphertext

$$c = c^* \cdot z^2 \pmod N,$$

for some random value $z \in (\mathbb{Z}/N\mathbb{Z})^*$. It is easy to see that c is an encryption of the same bit b. Hence, by asking our oracle to decrypt c we will obtain the decryption of c^*. □

16.1.2. ElGamal Encryption: The Goldwasser–Micali encryption scheme is passively secure, but not efficient. What we really want is a simple encryption algorithm which is efficient and which is passively secure[1]. The simplest efficient IND-CPA secure encryption algorithm is the ElGamal encryption algorithm, which is based on the discrete logarithm problem. In the following we shall

[1]At this point we still focus on passively secure systems. Once we have solved this, we will turn our focus to actively secure schemes.

describe the finite field analogue of ElGamal encryption; we leave it as an exercise to write down the elliptic curve variant.

Domain Parameters: Unlike the RSA algorithm, in ElGamal encryption there are some public parameters which can be shared by a number of users. These are called the domain parameters and are given by

- p a "large prime", by which we mean one with around 2048 bits, such that $p-1$ is divisible by another "medium prime" q of around 256 bits.
- g an element of $\mathbb{F}_p^*$ of prime order q, i.e. $g = r^{(p-1)/q} \pmod p \neq 1$ for some $r \in \mathbb{F}_p^*$.

The domain parameters create a public finite abelian group G of prime order q with generator g.

Key Generation: Once these domain parameters have been fixed, the public and private keys can then be determined. The private key $\mathfrak{sk}$ is chosen to be an integer $x \leftarrow [0, \ldots, q-1]$, whilst the public key is given by $\mathfrak{pk} := h \leftarrow g^x \pmod p$. Notice that, whilst each user in RSA needed to generate two large primes to set up their key pair (which is a costly task), for ElGamal encryption each user only needs to generate a random number and perform a modular exponentiation to generate a key pair.

Encryption: Messages are assumed to be elements of the group G. To encrypt a message $m \in G$ we do the following:

- $k \leftarrow \{0, \ldots, q-1\}$
- $c_1 \leftarrow g^k$,
- $c_2 \leftarrow m \cdot h^k$,
- Output the ciphertext, $c \leftarrow (c_1, c_2) \in G \times G$.

Notice that since each message has a different ephemeral key k, encrypting the same message twice will produce different ciphertexts.

Decryption: To decrypt a ciphertext $c = (c_1, c_2)$ we compute

$$\frac{c_2}{{c_1}^x} = \frac{m \cdot h^k}{g^{x \cdot k}} = \frac{m \cdot g^{x \cdot k}}{g^{x \cdot k}} = m.$$

ElGamal Example: We first need to set up the domain parameters. For our small example we choose $q = 101$, $p = 809$ and $g = 256$. Note that q divides $p-1$ and that g has order q, in the multiplicative group of integers modulo p. As a public/private key pair we choose

- $x \leftarrow 68$,
- $h \leftarrow g^x = 498$.

Now suppose we wish to encrypt the message $m = 100$ to the user with the above ElGamal public key.

- We generate a random ephemeral key $k \leftarrow 89$.
- Set $c_1 \leftarrow g^k = 468$.
- Set $c_2 \leftarrow m \cdot h^k = 494$.
- Output the ciphertext as $c = (468, 494)$.

The recipient can decrypt our ciphertext by computing

$$\frac{c_2}{{c_1}^x} = \frac{494}{468^{68}} = 100.$$

This last value is computed by first computing 468^{68}, taking the inverse modulo p of the result and then multiplying this value by 494.

We can now start to establish basic security results about ElGamal encryption by presenting two results in the passive security setting. Our first one says that if the Diffie–Hellman problem

is hard then ElGamal is OW-CPA, whilst the second one says that if the Decision Diffie–Hellman problem is hard then ElGamal is IND-CPA.

Theorem 16.3. *If A is an adversary against the* OW-CPA *security of the ElGamal encryption scheme $\Pi(G)$ over the group G, then there is an adversary B against the Diffie–Hellman problem such that*

$$\mathrm{Adv}^{\mathsf{OW\text{-}CPA}}_{\Pi(G)}(A) = \mathrm{Adv}^{\mathsf{DHP}}_{G}(B).$$

Proof. The algorithm A takes as input a public key h and a target ciphertext $c^* = (c_1, c_2)$, and returns the underlying plaintext. We will show how to use this algorithm to create an algorithm B to solve the DHP. We suppose B is given $X = g^x$ and $Y = g^y$, and is asked to solve the Diffie–Hellman problem, i.e. to output the value of $g^{x\cdot y}$; algorithm B then proceeds as in Algorithm 16.1.

Algorithm 16.1: Algorithm to solve DHP given an algorithm to break the one-way security of ElGamal

As input we have $X = g^x \in G$ and $Y = g^y \in G$.
$h \leftarrow X = g^x$.
$c_1 \leftarrow Y = g^y$.
$c_2 \leftarrow G$.
$c^* \leftarrow (c_1, c_2)$.
$m \leftarrow A(c^*, h)$.
return c_2/m.

In words, algorithm B first sets up an ElGamal public key which depends on the input to the Diffie–Hellman problem, i.e. we set $h \leftarrow X = g^x$ (note that algorithm B does not know what the corresponding private key is). Now we write down the target "ciphertext" $c^* = (c_1, c_2)$, where

- $c_1 \leftarrow Y = g^y$,
- $c_2 \leftarrow G$, i.e. a random element of the group.

This ciphertext is sent to algorithm A, along with the public key h. Algorithm A will then output (if successful) the underlying plaintext, We then solve the original Diffie–Hellman problem by computing

$$Z \leftarrow \frac{c_2}{m} = \frac{m \cdot h^y}{m} = h^y = g^{x\cdot y}.$$

□

We can use a similar technique to prove that ElGamal is IND-CPA, but now we have to assume that the Decision Diffie–Hellman problem is hard. Notice that to obtain a stronger notion of security, we have to assume a weaker problem is hard, i.e. make a stronger assumption.

Theorem 16.4. *If A is an adversary against the* IND-CPA *security of the ElGamal encryption scheme $\Pi(G)$ over the group G, then there is an adversary B against the Decision Diffie–Hellman problem such that*

$$\mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi(G)}(A) = 2 \cdot \mathrm{Adv}^{\mathsf{DDH}}_{G}(B).$$

Proof. As usual we will use algorithm A as a subroutine called by algorithm B. Our algorithm B for solving now proceeds as in Algorithm 16.2; to see why this algorithm solves the DDH problem consider the following argument.

- In the case when $z = x \cdot y$ then the encryption input into the guess stage of algorithm A will be a valid encryption of m_b. Hence, if algorithm A can really break the semantic security of ElGamal encryption then the output b' will be correct and the algorithm will return **true**.

Algorithm 16.2: Algorithm to solve DDH given an algorithm to break the semantic security of ElGamal

As input we have $X = g^x, Y = g^y$ and $Z = g^z$.
$h \leftarrow X = g^x$.
$(m_0, m_1, s) \leftarrow A(\textbf{find}, h)$.
$c_1 \leftarrow Y = g^y$.
$b \leftarrow \{0, 1\}$.
$c_2 \leftarrow m_b \cdot g^z$.
$c^* \leftarrow (c_1, c_2)$.
$b' \leftarrow A(\textbf{guess}, c^*, s)$.
if $b = b'$ **then return true**.
else return false.

- Now suppose that $z \neq x \cdot y$, then the encryption input into the guess stage is almost definitely invalid, i.e. not an encryption of m_1 or m_2. Hence, the output b' of the guess stage will be independent of the value of b. Therefore we expect the above algorithm to return **true** or **false** with equal probability, and so $b' = b$ with probability $1/2$.

This is exactly the same argument that we had in the proof of security of the Goldwasser–Micali encryption scheme, and so the relationship between the advantages will follow in the same way. □

Despite the above positive results on the security of ElGamal encryption we still do not have a scheme which is secure against adaptive chosen ciphertext attacks. The main reason for this is that ElGamal is trivially malleable. Given a ciphertext for the message m,

$$(c_1, c_2) = (g^k, m \cdot h^k),$$

one can then create a valid ciphertext for the message $2 \cdot m$ without ever knowing m, nor the ephemeral key k, nor the private key x. In particular the following ciphertext decrypts to $2 \cdot m$,

$$(c_1, 2 \cdot c_2) = (g^k, 2 \cdot m \cdot h^k).$$

One can use this malleability property, just as we did with RSA in Lemma 15.4, to show that ElGamal encryption is not OW-CCA secure. Notice that $(1, 2)$ is a "trivial" encryption of the number 2, and we are in some sense combining the ciphertext $(1, 2)$ with the ciphertext (c_1, c_2) to produce a ciphertext which encrypts $2 \cdot m$. Any two ciphertexts can be combined in this way, to produce a ciphertext which encrypts the product of the underlying plaintexts. An encryption scheme with this property is called *multiplicatively homomorphic*.

Lemma 16.5. *ElGamal is not* OW-CCA.

Proof. Suppose the message the adversary wants to invert is $c^* = (c_1, c_2) = (g^k, m^* \cdot h^k)$. The adversary then creates the related message $c = (c_1, 2 \cdot c_2)$ and asks her decryption oracle to decrypt c to give the message m. Then Eve computes

$$\frac{m}{2} = \frac{2 \cdot c_2 \cdot {c_1}^{-x}}{2} = \frac{2 \cdot m^* \cdot h^k \cdot g^{-x \cdot k}}{2} = \frac{2 \cdot m^* \cdot g^{x \cdot k} \cdot g^{-x \cdot k}}{2} = \frac{2 \cdot m^*}{2} = m^*.$$

□

16.1.3. Paillier Encryption: There is an efficient system, due to Paillier, based on the difficulty of factoring large integers, which can be shown to be IND-CPA. Paillier's scheme has a number of interesting properties, such as the fact that it is additively homomorphic (which means it has found application in electronic voting applications).

Key Generation: We first pick an RSA modulus $N = p \cdot q$, but instead of working with the multiplicative group $(\mathbb{Z}/N\mathbb{Z})^*$ we work with $(\mathbb{Z}/N^2\mathbb{Z})^*$. The order of this last group is given by $\phi(N) = N \cdot (p-1) \cdot (q-1) = N \cdot \phi(N)$. This means, by Lagrange's Theorem, that for all a with $\gcd(a, N) = 1$ we have

$$a^{N \cdot (p-1) \cdot (q-1)} = 1 \pmod{N^2}.$$

The private key for Paillier's scheme is defined to be an integer d such that

$$d = 1 \pmod{N},$$
$$d = 0 \pmod{(p-1) \cdot (q-1)},$$

such a value of d can be found by the Chinese Remainder Theorem. The public key $\mathfrak{pk}$ is just the integer N, and the private key $\mathfrak{sk}$ is the integer d.

Encryption: Messages are defined to be elements of $\mathbb{Z}/N\mathbb{Z}$. To encrypt such a message the encryptor picks an integer $r \in \mathbb{Z}/N^2\mathbb{Z}$ and computes $c \leftarrow (1+N)^m \cdot r^N \pmod{N^2}$.

Decryption: To decrypt one first computes

$$\begin{aligned} t &\leftarrow c^d \pmod{N^2} \\ &= (1+N)^{m \cdot d} \cdot r^{d \cdot N} \pmod{N^2} \\ &= (1+N)^{m \cdot d} \pmod{N^2} && \text{since } d = 0 \pmod{(p-1) \cdot (q-1)} \\ &= 1 + m \cdot d \cdot N \pmod{N^2} \\ &= 1 + m \cdot N \pmod{N^2} && \text{since } d = 1 \pmod{N}. \end{aligned}$$

Then to recover the message we compute $R \leftarrow \frac{t-1}{N}$.

Just like the other schemes presented so far, Paillier encryption is malleable, and hence it cannot be OW-CCA secure. However, unlike RSA, Rabin and ElGamal, the malleability is *additive*. In particular given two ciphertexts, $c_1 = (1+N)^{m_1} \cdot {r_1}^N$ and $c_2 == (1+N)^{m_2} \cdot {r_2}^N$, encrypting messages m_1 and m_2 we can easily form the encryption of the sum of the plaintexts by computing

$$c_1 \cdot c_2 = \left((1+N)^{m_1} \cdot {r_1}^N\right) \cdot \left((1+N)^{m_2} \cdot {r_2}^N\right) = (1+N)^{m_1+m_2} \cdot (r_1 \cdot r_2)^N.$$

Thus we say that Paillier encryption is *additively homomorphic*; this should be compared to the *multiplicatively homomorphic* nature of RSA, ElGamal encryption and Rabin encryption. Finding an IND-CPA secure encryption scheme which is simultaneously both additively and multiplicatively homomorphic was a major open research question in cryptography for over thirty years. Such a Fully Homomorphic Encryption (FHE) scheme, was given in 2009 by Gentry, and we shall return to such FHE schemes in Chapter 17.

The Paillier encryption scheme can be proved to be IND-CPA secure assuming the following generalization of the QUADRES problem is secure. Instead of detecting whether something is a square modulo N, the adversary needs to detect whether something is an Nth power modulo N^2. We present the problem diagrammatically in Figure 16.2, and leave it to the reader to show that Paillier encryption is IND-CPA under this assumption.

16.2. Random Oracle Model, OAEP and the Fujisaki–Okamoto Transform

We have now seen various proofs of security for public key encryption schemes, yet none of them prove security against adversaries which can make chosen ciphertext queries. Let us see why this might be a problem for our existing proof techniques. To recap, we are given an algorithm A and the proof proceeds by trying to create a new algorithm B which uses A as a subroutine. The input to B is the hard mathematical problem we wish to solve (e.g. factoring), whilst the input to A is some cryptographic problem. Since we have a public key algorithm adversary A can make its own

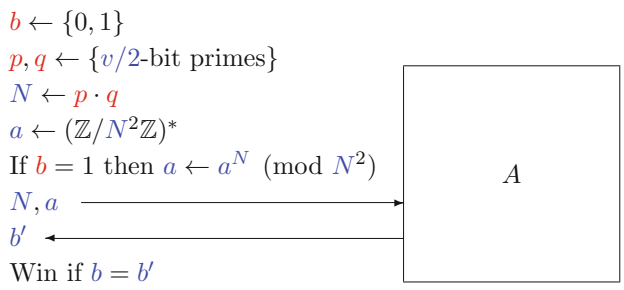

FIGURE 16.2. Security game to define the Decision Composite Residuosity Problem (DCRP)

encryption queries, as soon as it is given the public key. Thus when A is a CPA adversary there are no oracle queries which have to be answered by B. The usual trick in these proofs is that the hard mathematical problem is based on some secret which B does not know, and that this secret becomes the secret key of the public key cryptosystem which A is trying to break.

The difficulty arises when A is a CCA adversary; in this case A is allowed to call a decryption oracle for the input public key. The algorithm B, if it wants to use algorithm A as a subroutine, needs to supply the answers to A's oracle queries. In constructing algorithm B we now have a number of problems:

- Its responses must appear valid (i.e. valid encryptions should decrypt), otherwise algorithm A would notice its decryption oracle was lying. Hence, algorithm B could no longer guarantee that algorithm A was successful with non-negligible probability.
- The responses of the decryption oracle should be consistent with the probability distributions of responses that A expects if the oracle was a true decryption oracle. Again, otherwise A would notice.
- The responses of the decryption oracle should be consistent across all the calls made by the adversary A.
- Algorithm B needs to supply these answers without knowing the secret key. To decrypt we appear to need to know the secret key, which is exactly what B does not have. In most cases if B knew the secret key it would not need A in the first place!

This last point is the most crucial one. We are essentially asking B to decrypt a ciphertext without knowing the private key, but this is meant to be impossible since our scheme is meant to be secure.

To get around this problem it has become common practice to use the "random oracle model", which we introduced in Chapter 11. Recall that a random oracle is an idealized hash function which on input of a new query will pick, uniformly at random, some response from its output domain, and which if asked the same query twice will always return the same response. So to use the random oracle model we need to include a hash function somewhere in the processing of our encryption and/or decryption operations.

In the random oracle model we assume our adversary A makes no use of the explicit hash function being used in the scheme under attack. In other words the adversary A runs, and is successful, even if we replace the real hash function by a random oracle. The algorithm B responds to the decryption oracle queries of A by cheating and "cooking" the responses of the random oracle to suit his own needs.

A proof in the random oracle model is an even more relativized proof than that which we considered before. Such a proof says that assuming some problem is hard, say factoring, then an adversary cannot exist which makes no use of the underlying hash function. This does not imply that an adversary does not exist which uses the real specific hash function as a means of breaking the cryptographic system.

16.2.1. RSA-OAEP: Recall that the raw RSA function does not provide a semantically secure encryption scheme, even against passive adversaries. To make a system which is secure we need either to add redundancy to the plaintext before encryption or to add some other form of redundancy to the ciphertext, so that we can check upon decryption whether the ciphertext has been validly generated. In addition the padding used needs to be random so as to make a non-deterministic encryption algorithm. Over the years a number of padding systems have been proposed. However, many of the older ones are now considered weak.

By far the most successful padding scheme in use today was invented by Bellare and Rogaway and is called OAEP or Optimized Asymmetric Encryption Padding. The general OAEP method is a padding scheme which can be used with any function which is a trapdoor one-way permutation on strings of k bits in length. When used with the RSA trapdoor one-way permutation we need to "tweak" the construction a little since RSA does not act as a permutation on bit strings of length k, it acts as a permutation on the set of integers modulo N. When used with RSA it is often denoted RSA-OAEP.

Originally it was thought that OAEP was a plaintext aware encryption algorithm in the random oracle model, irrespective of the underlying trapdoor one-way permutation, but this claim has since been shown to be wrong. However, one can show in the random oracle model that RSA-OAEP is semantically secure against adaptive chosen ciphertext attacks.

We first give the description of OAEP in general. Let f be any k-bit to k-bit trapdoor one-way permutation. Let k_0 and k_1 denote numbers such that a work effort of 2^{k_0} or 2^{k_1} is impossible (e.g. $k_0, k_1 > 128$). Put $n = k - k_0 - k_1$ and let

$$G : \{0,1\}^{k_0} \longrightarrow \{0,1\}^{n+k_1}$$
$$H : \{0,1\}^{n+k_1} \longrightarrow \{0,1\}^{k_0}$$

be hash functions. Strictly speaking H *is* a hash function as it takes bitstrings and compresses them in length, whereas G is a function more like a key derivation function in that it expands a short bit string into a longer one. In practice for RSA-OAEP both F and G are implemented using hash functions, with G being implemented by repeated hashing of the input along with a counter as in Section 14.6.

Let m be a message of n bits in length. We then encrypt using the function

$$c \leftarrow E(m) = f\left(\{\left(m \parallel 0^{k_1}\right) \oplus G(R)\} \parallel \{R \oplus H\left(\left(m \| 0^{k_1}\right) \oplus G(R)\right)\}\right) = f(A).$$

where

- $m \parallel 0^{k_1}$ means m followed by k_1 zero bits,
- R is a random bit string of length k_0,
- $\parallel$ denotes concatenation.

One can view OAEP as a two-stage Feistel network, as Figure 16.3 demonstrates. To decrypt we proceed as follows:

- Apply the trapdoor to f to recover $A = f^{-1}(c) = \{T \| \{R \oplus H(T)\}\}$.
- Compute $H(T)$ and recover R from $R \oplus H(T)$.
- Compute $G(R)$ and recover $v = m \parallel 0^{k_1}$ from $T = m \parallel 0^{k_1} \oplus G(R)$.
- If v ends in k_1 zeros output m, otherwise return $\perp$.

When applying the OAEP transform to produce an RSA-based version we take $k = 8 \cdot \lfloor \log_8(N) \rfloor$. We then produce the OAEP block A as above, and then we think of A as an integer less than N. It is to this integer we apply the RSA encryption function $f(A) = A^e \pmod N$. Upon decrypting we invert the function, by computing $f^{-1}(c) = c^d \pmod N$ to obtain an integer A'. We then check whether A' has the correct number of zero bits to the left, and if so take A as the k rightmost bits

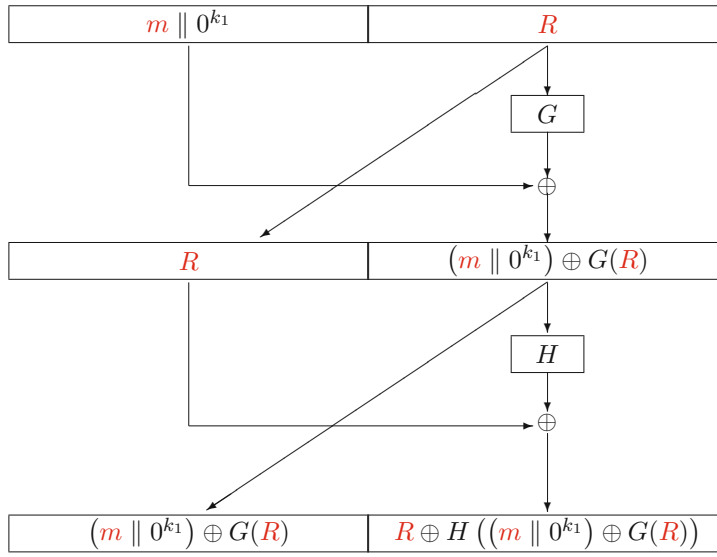

FIGURE 16.3. OAEP as a Feistel network

of A'. However, one needs to be careful about how one reports if the leftmost bits are not zero as expected. Such an error should not be distinguishable from the OAEP padding decoding returning a $\perp$. The main result about RSA-OAEP is the following.

Theorem 16.6. *In the random oracle model, if we model G and H by random oracles then RSA-OAEP is* IND-CCA *secure if the RSA problem is hard.*

PROOF. We sketch the proof and leave the details for the interested reader to look up. We first rewrite the RSA function f as

$$f : \begin{cases} \{0,1\}^{n+k_1} \times \{0,1\}^{k_0} & \longrightarrow & (\mathbb{Z}/N\mathbb{Z})^* \\ (s,t) & \longmapsto & (s\|t)^e \pmod N, \end{cases}$$

assuming the above-mentioned padding to the left is performed on encryption. We then define RSA-OAEP as applying the above function f to the inputs

$$s = (m\|0^{k_1}) \oplus G(r) \text{ and } t = r \oplus H(s).$$

The RSA assumption can be proved to be equivalent to the partial one-wayness of the function f, in the sense that the problem of recovering s from $f(s,t)$ is as hard as recovering (s,t) from $f(s,t)$. So for the rest of our sketch we try to turn an adversary A for breaking RSA-OAEP into an algorithm B which solves the partial one-wayness of the RSA function. In particular B is given $c^* = f(s^*,t^*)$, for some fixed RSA modulus N, and is asked to compute s^*.

Algorithm A works in the random oracle model and so it is assumed to only access the functions H and G via external calls, with the answers being provided by the environment. In addition A expects H and G to "act like" random functions. In our context B is the environment for A and so needs to supply A with the answers to its calls to the functions H and G. Thus B maintains a list of queries to H and G made by algorithm A, along with the responses. We call these lists the H-List and the G-List respectively.

Algorithm B now calls algorithm A, which will make a series of calls to the H and G oracles (we discuss how these are answered below). Eventually A will make a call to its $\mathcal{O}_{\mathsf{LR}}$ oracle by

producing two messages m_0 and m_1 of n bits in length. A bit b is then chosen by B, and B now assumes that c^* is the encryption of m_b. The ciphertext c^* is now returned to A, as the response to its call of the $\mathcal{O}_{\mathsf{LR}}$ oracle. Algorithm A then continues and tries to guess the bit b.

The oracle queries are answered by B as follows:

- Query $G(\gamma)$:
  For any query (δ, δ_H) in the H-List one checks whether
  $$c^* = f(\delta, \gamma \oplus \delta_H).$$
  – If this holds then we have partially inverted f as required (i.e. B can output δ as its solution). We can still, however, continue with the simulation of G and set
  $$G(\gamma) = \delta \oplus (m_b \| 0^{k_1}).$$
  – If this equality does not hold for any value of δ then we choose $\gamma_G = G(\gamma)$ uniformly at random from the codomain of G, and add the pair (γ, γ_G) to the G-List.
- Query $H(\delta)$:
  A random value δ_H is chosen from the codomain of H; the value (δ, δ_H) is added to the H-List. We also check whether for any (γ, γ_H) in the G-List we have
  $$c^* = f(\delta, \gamma \oplus \delta_H),$$
  if so we have managed to partially invert the function f as required, and we output the value of δ.
- Query decryption of c :
  We look in the G-List and the H-List for a pair $(\gamma, \gamma_G), (\delta, \delta_H)$ such that if we set $\sigma = \delta$, $\tau = \gamma \oplus \delta_H$ and $\mu = \gamma_G \oplus \delta$, then $c = f(\sigma, \tau)$ and the k_1 least significant bits of μ are equal to zero. If this is the case then we return the plaintext consisting of the n most significant bits of μ, otherwise we return $\perp$.

Notice that if a ciphertext which was generated in the correct way (by calling G, H and the encryption algorithm) is then passed to the above decryption oracle, we will obtain the original plaintext back.

We have to show that the above decryption oracle is able to "fool" the adversary A enough of the time. In other words when the oracle is passed a ciphertext which has not been generated by a prior call to the necessary G and H, we need to show that it produces a value which is consistent with the running of the adversary A. Finally we need to show that if the adversary A has a non-negligible chance of breaking the semantic security of RSA-OAEP then one has a non-negligible probability that B can partially invert f.

These last two facts are proved by careful analysis of the probabilities associated with a number of events. Recall that B assumes that $c^* = f(s^*, t^*)$ is an encryption of m_b. Hence, there should exist an r^* which satisfies

$$r^* = H(s^*) \oplus t^*,$$
$$G(r^*) = s^* \oplus (m_b \| 0^{k_1}).$$

One first shows that the probability of the decryption simulator failing is negligible. Then one shows that the probability that s^* is actually asked of the H oracle is non-negligible, as long as the adversary A has a non-negligible probability of finding the bit b. But as soon as s^* is asked of H then we spot this and can therefore break the partial one-wayness of f.

The actual technical probability arguments are rather involved and we refer the reader to the paper of Fujisaki, Okamoto, Pointcheval and Stern where the full proof is given. □

16.2.2. The Fujisaki–Okamoto Transform: We end this section with another generic transform which can be applied to one of the IND-CPA secure schemes from Section 16.1, to turn it into an IND-CCA secure encryption scheme. Like the OAEP transform, the transform in this section is secure assuming the adversary operates in the random oracle model.

Suppose we have a public key encryption scheme which is semantically secure against chosen plaintext attacks, such as ElGamal encryption. Such a scheme by definition needs to be non-deterministic hence we write the encryption function as

$$E(m, r),$$

where m is the message to be encrypted and r is the random input, and we denote the decryption function by $D(c)$. Hence, for ElGamal encryption we have

$$E(m, r) = (g^r, m \cdot h^r).$$

Fujisaki and Okamoto showed how to turn such a scheme into one which is IND-CCA secure. Their result only applies in the random oracle model and works by showing that the resulting scheme is plaintext aware. We do not go into the details of the proof at all, but simply give the transformation, which is both simple and elegant.

We take the encryption function above and alter it by setting

$$E'(m, r) = E(m\|r, H(m\|r))$$

where H is a hash function. The decryption algorithm is also altered in that we first compute

$$m' = D(c)$$

and then we check that

$$c = E(m', H(m')).$$

If this last equation holds we recover m from $m' = m\|r$; if the equation does not hold then we return $\perp$. For ElGamal encryption we therefore obtain the encryption algorithm

$$(g^{H(m\|r)}, (m\|r) \cdot h^{H(m\|r)}),$$

which is only marginally less efficient than raw ElGamal encryption.

16.3. Hybrid Ciphers

Almost always public key schemes are used only to transmit a short per message secret, such as a session key. This is because public key schemes are too inefficient to use to encrypt vast amounts of data. The actual data is then encrypted using a symmetric cipher. Such an approach is called a hybrid encryption scheme.

We now formalize this way of designing a public key encryption scheme with a hybrid cipher, via the so-called KEM/DEM approach. A KEM is a Key Encapsulation Mechanism, which is the public key component of a hybrid cipher, whilst a DEM is a Data Encapsulation Mechanism, which is the symmetric component. We have already mentioned DEMs in Chapters 11 and 13, where we constructed DEMs which were ot-IND-CCA secure as symmetric key encryption schemes, namely symmetric key schemes which were only ever designed to encrypt a single message.

We will present a security model for KEMs and then show, without using random oracles, that a suitably secure DEM and a suitably secure KEM can be combined to produce an IND-CCA secure hybrid cipher. This means we only need to consider the symmetric and public key parts separately, simplifying our design considerably. Finally, we show how a KEM can be constructed in the random oracle model using either the RSA or the DLP primitive. The resulting KEMs are very simple to construct and very natural, so we see the simplification obtained by utilizing hybrid encryption.

16.3.1. Defining a Key Encapsulation Mechanism: We define a Key Encapsulation Mechanism, or KEM, to be a mechanism which from the encryptor's side takes a public key $\mathfrak{pk}$ and outputs a symmetric key $k \in \mathbb{K}$ and an encapsulation of that key c for use by the holder of the corresponding private key. The holder of the private key $\mathfrak{sk}$ can then take the encapsulation c and their private key, and then recover the symmetric key k. Thus no message is input into the encapsulation mechanism. We therefore have three algorithms which operate as follows:

- $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \text{KeyGen}(\mathbb{K})$.
- $(c, k) \leftarrow \text{Encap}_{\mathfrak{pk}}()$.
- $k \leftarrow \text{Decap}_{\mathfrak{sk}}(c)$.

For correctness we require, for all pairs $(\mathfrak{pk}, \mathfrak{sk})$ output by $\text{KeyGen}(\mathbb{K})$, that

$$\text{If } (c, k) \leftarrow \text{Encap}_{\mathfrak{pk}}() \text{ then } \text{Decap}_{\mathfrak{sk}}(c) = k.$$

The security definition for KEMs is based on the security definition of indistinguishability of encryptions for public key encryption algorithms. However, we now require that the key output by a KEM should be indistinguishable from a random key. Thus the security game is defined via the following game.

- The challenger generates a random key $k_0 \in \mathbb{K}$ from the space of symmetric keys output by the KEM.
- The challenger calls the Encap function of the KEM to produce a valid key $k_1 \in \mathbb{K}$ and its encapsulation c^*, under the public key $\mathfrak{pk}$.
- The challenger picks a bit b and sends to the adversary the values k_b, c^*.
- The goal of the adversary is to decide whether $b = 0$ or 1.

The advantage of the adversary, against the KEM Π, is defined to be

$$\text{Adv}_{\Pi}^{\text{IND-CPA}}(A) = 2 \cdot \left| \Pr(A(\mathfrak{pk}, k_b, c^*) = b) - \frac{1}{2} \right|.$$

The above only defines the security in the passive case; to define security under adaptive chosen ciphertext attacks one needs to give the adversary access to a decapsulation function. This decapsulation function will return the key (or the invalid encapsulation symbol) for any encapsulation of the adversary's choosing, bar the target encapsulation c^*. In such a situation we denote the advantage by $\text{Adv}_{\Pi}^{\text{IND-CCA}}(A)$, and say the KEM is secure if this advantage is "small" for all adversaries A. We describe the full security model in Figure 16.4.

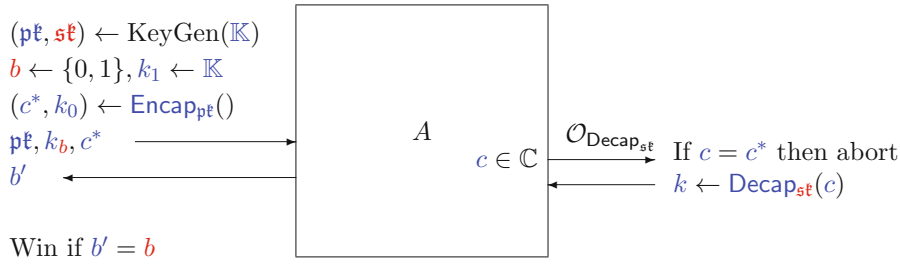

FIGURE 16.4. Security game IND-CCA for a KEM

16.3.2. Generically Constructing Hybrid Encryption: The idea of a KEM/DEM system is that one takes a KEM (defined by the algorithms $\text{KeyGen}, \text{Encap}_{\mathfrak{pk}}, \text{Decap}_{\mathfrak{sk}}$ and which outputs symmetric keys from the space $\mathbb{K}$) and a DEM (defined by the algorithms e_k, d_k and with key space $\mathbb{K}$), and then uses the two together to form a hybrid cipher, a.k.a. a public key encryption

scheme. The key generation method of the public key scheme is simply the key generation method of the underlying KEM. To encrypt a message m to a user with public/private key pair $(\mathfrak{pk}, \mathfrak{sk})$, one performs the following steps:

- $(k, c_1) \leftarrow \mathsf{Encap}_{\mathfrak{pk}}()$.
- $c_2 \leftarrow e_k(m)$.
- $c \leftarrow (c_1, c_2)$.

The recipient, upon receiving the pair $c = (c_1, c_2)$, performs the following steps to recover m.

- $k \leftarrow \mathsf{Decap}_{\mathfrak{sk}}(c_1)$.
- If $k = \perp$ return $\perp$.
- $m \leftarrow d_k(c_2)$.
- Return m.

We would like the above hybrid cipher to meet our security definition for public key encryption schemes, namely IND-CCA.

Theorem 16.7. *The hybrid public key encryption scheme Π defined above is* IND-CCA *secure, assuming the KEM scheme Π_1 is* IND-CCA *secure and the DEM Π_2 is* ot-IND-CCA *secure. In particular if A is an adversary against the* IND-CCA *security of the hybrid public key encryption scheme then there exist adversaries B and C such that*

$$\mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi}(A) \leq 2 \cdot \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi_1}(B) + \mathrm{Adv}^{\mathsf{ot\text{-}IND\text{-}CCA}}_{\Pi_2}(C).$$

Before we give the proof notice that we only need one-time security for the DEM, as each symmetric encryption key output by the KEM is only used once. Thus we can construct the DEM from much simpler components.

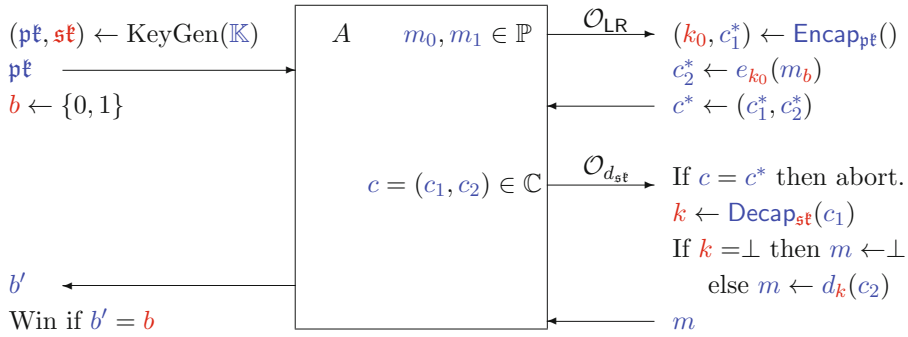

FIGURE 16.5. IND-CCA game G_0 for our hybrid scheme

PROOF. We sketch the proof and direct the reader to the paper of Cramer and Shoup for more details. First consider Figure 16.5; this is the standard IND-CCA game for public key encryption, tailored for our hybrid encryption scheme. Let us call this game G_0. We now modify the game which A is playing; instead of encrypting c_2^* using the valid key k_0 we instead use a new random key called k_1. This modified game we call G_1 and we present it in Figure 16.6.

Game G_1 is relatively easy to analyse so we will do this first. We will show that if A wins in game G_1, then we can construct an adversary C which will use A as a subroutine to break the ot-IND-CCA security of the DEM (e_k, d_k). The key trick we use is that the c_1^* component in game G_1 is unrelated to the key which is used to encrypt m_b. We present algorithm C in Algorithm 16.3. Notice that in Game G_1 when A makes a decryption query for (c_1, c_2), it is validly decrypted by C, as long as $c_1 \neq c_1^*$. When this last condition holds, algorithm C uses its own decryption

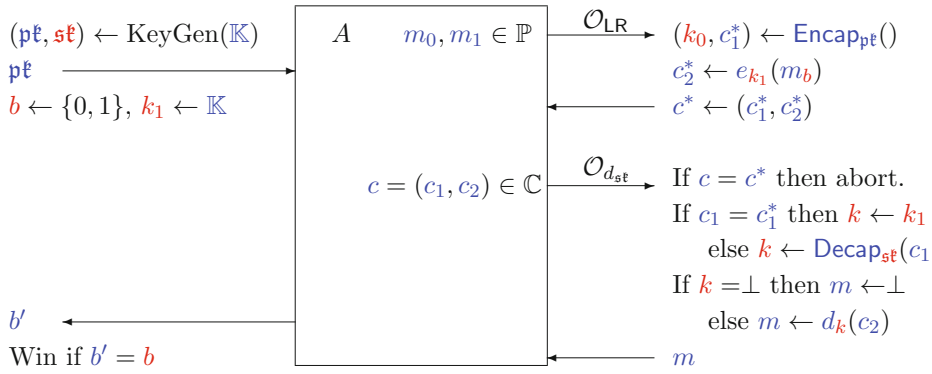

FIGURE 16.6. Game G_1

oracle to return the decryption of c_2. Note that B's target ciphertext c_2^* is never passed to its own decryption oracle, unless B aborts because A made an invalid query. In addition, note that the $\mathcal{O}_{\mathsf{LR}}$ oracle of C is only called once, as is required in the one-time security of a DEM. Finally, note that A winning (or losing) game G_1 directly corresponds to C winning (or losing) game, thus

$$\mathrm{Adv}^{\mathsf{ot\text{-}IND\text{-}CCA}}_{\Pi_2}(C) = 2 \cdot \left| \Pr[C \text{ wins }] - \frac{1}{2} \right| = 2 \cdot \left| \Pr[A \text{ wins in game } G_1] - \frac{1}{2} \right|.$$

Algorithm 16.3: Algorithm C

$(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{KeyGen}(\mathbb{K})$.
Call A with input the public key $\mathfrak{pk}$.

/* A's $\mathcal{O}_{\mathsf{LR}}$ Oracle Queries */
A makes an $\mathcal{O}_{\mathsf{LR}}$ query with messages m_0, m_1.
C passes m_0, m_1 to its own $\mathcal{O}_{\mathsf{LR}}$ oracle to obtain c_2^*.
$c_1^* \leftarrow \mathsf{Encap}_{\mathfrak{pk}}()$.
$c^* \leftarrow (c_1^*, c_2^*)$.
return c^*.

/* A's $\mathcal{O}_{d_k}$ Oracle Queries */
A makes an $\mathcal{O}_{d_k}$ query with ciphertext $c = (c_0, c_1)$.
if $c_0 \neq c_0{}^*$ **then** $k \leftarrow \mathsf{Decap}_{\mathfrak{sk}}(c_0)$, $m \leftarrow d_k(c_1)$.
else if $c_1 \neq c_1{}^*$ **then** C passes c_1 to its $\mathcal{O}_{d_k}$ oracle to obtain m.
else abort.
return m.

/* A's Response */
When A returns b'.
return b'.

We now turn to what is the most complex step. We want to bound the probability

$$\left| \Pr[A \text{ wins in game } G_0] - \Pr[A \text{ wins in game } G_1] \right|.$$

We do this by presenting an algorithm B which uses A to break the IND-CCA security of the KEM. It does this as follows: Algorithm B does not know whether the key/encapsulation pair given to it is a real encapsulation of the key, or a fake one. It constructs an environment for A to play in, where in the first case A is playing game G_0, whereas in the second it is playing game G_1. Hence, any advantage A has in distinguishing the two games it is playing in, can be exploited by B to break the KEM. We give the algorithm for B in Algorithm 16.4.

Algorithm 16.4: Algorithm B

B has as input $\mathfrak{pk}$.
Call A with input the public key $\mathfrak{pk}$.

/* A's $\mathcal{O}_{\mathsf{LR}}$ Oracle Queries */
A makes an $\mathcal{O}_{\mathsf{LR}}$ query with messages (m_0, m_1).
B calls its own $\mathcal{O}_{\mathsf{LR}}$ oracle to obtain (c_1^*, k^*).
$b \leftarrow \{0,1\}$.
$c_2^* \leftarrow e_{k^*}(m_b)$.
$c^* \leftarrow (c_1^*, c_2^*)$.
return c^*

/* A's $\mathcal{O}_{d_k}$ Oracle Queries */
A makes an $\mathcal{O}_{d_k}$ query with ciphertext $c = (c_0, c_1)$.
if $c_0 \neq {c_0}^*$ **then**
  B calls its $\mathcal{O}_{\mathsf{Decap}_{\mathfrak{sk}}}$ oracle on c_0 to obtain k.
  $m \leftarrow d_k(c_1)$.
else if $c_1 \neq {c_1}^*$ **then** $m \leftarrow d_{k^*}(c_1)$.
else abort.
return m.

/* A's Response */
When A returns b'.
$a \leftarrow 0$.
if $b' \neq b$ **then** $a \leftarrow 1$.
return a.

So algorithm B outputs zero if it thinks A is playing in game G_0, i.e. if k^* is the actual key underlying the encapsulation c_1^*, whereas B will output one if it thinks A is playing in game G_1, i.e. if k^* is just some random key. So we have

$$\begin{aligned} 2 \cdot \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi_1}(B) &= 2 \cdot \Big| \Pr[a = 0 \mid A \text{ in game } G_0] - \Pr[a = 0 \mid A \text{ in game } G_1] \Big| \\ &= 2 \cdot \Big| \Pr[A \text{ wins in game } G_0] - \Pr[A \text{ wins in game } G_1] \Big|. \end{aligned}$$

We then have that

$$\begin{aligned} \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi}(A) &= 2 \cdot \Big| \Pr[A \text{ wins in game } G_0] - \frac{1}{2} \Big| \\ &= 2 \cdot \Big| \Pr[A \text{ wins in game } G_0] - \frac{1}{2} \\ &\qquad - \Pr[A \text{ wins in game } G_1] + \Pr[A \text{ wins in game } G_1] \Big| \qquad \text{adding zero} \end{aligned}$$

$$
\begin{aligned}
&\leq 2 \cdot \left| \Pr[A \text{ wins in game } G_0] - \Pr[A \text{ wins in game } G_1] \right| \\
&\quad + 2 \cdot \left| \Pr[A \text{ wins in game } G_1] - \frac{1}{2} \right| && \text{triangle Inequality} \\
&= 2 \cdot \left| \Pr[A \text{ wins in game } G_0] - \Pr[A \text{ wins in game } G_1] \right| \\
&\quad + \mathrm{Adv}^{\mathsf{ot\text{-}IND\text{-}CCA}}_{\Pi_2}(C) \\
&= 2 \cdot \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi_1}(B) + \mathrm{Adv}^{\mathsf{ot\text{-}IND\text{-}CCA}}_{\Pi_2}(C).
\end{aligned}
$$

□

16.4. Constructing KEMs

As previously mentioned, KEMs are simpler to design than public key encryption algorithms. In this section we first look at RSA-KEM, whose construction and proof should be compared to that of RSA-OAEP. Then we turn to DHIES-KEM which should be compared to the ElGamal variant of the scheme based on the Fujisaki–Okamoto transform.

16.4.1. RSA-KEM: Let N denote an RSA modulus, i.e. a product of two primes p and q of roughly the same size. Let e denote an RSA public exponent and d an RSA private exponent. We let $f_{N,e}(x)$ denote the RSA function, i.e. the function that maps an integer x modulo N to the number $x^e \pmod N$. The important point for RSA-KEM is that this is a trapdoor one-way function; only the holder of the secret trapdoor d should be able to invert it. This is summarized in the RSA problem, which is the problem of given an integer y modulo N to recover the value of x such that $f_{N,e}(x) = y$.

We define RSA-KEM by taking a cryptographic key derivation function H which takes integers modulo N and maps them to symmetric keys of the size required by the user of the KEM (i.e. the key size of the DEM). In our security model we will assume that H behaves like a random oracle. Encapsulation then works as follows:

- $x \leftarrow \{1, \ldots, N-1\}$.
- $c \leftarrow f_{N,e}(x)$.
- $k \leftarrow H(x)$.
- Output (k, c).

Since the person with the private key can invert the function $f_{N,e}$, decapsulation is easily performed via

- $x \leftarrow f_N^{-1}(c)$.
- $k \leftarrow H(x)$.
- Output k.

There is no notion of invalid ciphertexts, and this is simpler in comparison to RSA-OAEP. The construction actually works for *any* trapdoor one-way function. We now only need to show that this simple construction meets our definition of a secure KEM.

Theorem 16.8. *In the random oracle model RSA-KEM is an* IND-CCA *secure KEM, assuming the RSA problem is hard. In particular given an adversary A against the* IND-CCA *property of the RSA-KEM scheme Π, for moduli of size v bits, which treats H as a random oracle, then there is an adversary B against the RSA problem for integers of size v such that*

$$\mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi}(A) \leq \mathrm{Adv}^{\mathsf{RSA}}_{v}(B).$$

Proof. Since A works in the random oracle model, we model the function H in the proof as a random oracle. Thus algorithm B keeps a list of triples (z, c, h), which we will call the H-List, of queries to H, which is initially set to be empty. The value z denotes the query to the function H,

the value h the output and the value c denotes the output of $f_{N,e}$ on z. Algorithm B has as input a value y for which it is trying to invert the function $f_{N,e}$. Algorithm B passes the values N, e to A as the public key of the KEM which A is trying to attack. To generate the challenge encapsulation, the challenger generates a symmetric key k at random and takes as the challenge encapsulation the value $c^* \leftarrow y$ of the RSA function for which B is trying to find the preimage. It then passes k and c^* to A.

The adversary is allowed to make queries of H for values z. If this query on z has been made before, then B uses its H-List to respond as required. If there is a value on the list of the form $(\perp, c, h)$ with $f_{N,e}(z) = c$ then B replaces this value with (z, c, h) and responds with h. Otherwise B generates a new random value of h, adds the triple $(z, f_{N,e}(z), h)$ to the list and responds with h.

The adversary can also make decapsulation queries on an encapsulation c. If there is a value (z, c, h), for some c and h, on the H-List it responds with h. Otherwise, it generates h at random, places the triple $(\perp, c, h)$ on the list and responds with h.

Since A is running in the random oracle model, the only way that A can have any success in the game is by querying H on the preimage of y. Thus if A is successful then the preimage of y will exist on the list of triples kept by algorithm B. Hence, when A terminates B searches its H-List for a triple of the form (x, y, h) and if there is one it outputs x as the preimage of y.

In summary algorithm B is presented in Algorithm 16.5. It is easy to see that the calls to H and the calls to $\mathcal{O}_{\mathsf{Decap}_{\mathcal{SE}}}$ are answered by B in a consistent way, due to algorithm B's ability to ensure the required behaviour of the random oracle responses. □

16.4.2. The DHIES Encryption Scheme: The DHIES encryption scheme is the instantiation of our hybrid encryption paradigm, with the DHIES-KEM and the data encapsulation mechanism being the Encrypt-then-MAC instantiation. The scheme was designed by Abdalla, Bellare and Rogaway and was originally called DHAES, for Diffie–Hellman Augmented Encryption Scheme. However, this caused confusion with the Advanced Encryption Standard. So the name was changed to DHIES, for Diffie–Hellman Integrated Encryption Scheme. When used with elliptic curves it is called ECIES. To define the scheme all we need to do is present the DHIES-KEM component, as the rest follows from our prior discussions.

Key Generation: The domain parameters are a cyclic finite abelian group G of prime order q, a generator g and the key space $\mathbb{K}$ for the data encapsulation mechanism to be used. We require a key derivation function H with codomain equal to $\mathbb{K}$, which again we will model as a random oracle. To generate a public/private key pair we generate a random $x \leftarrow \mathbb{Z}/q\mathbb{Z}$ and compute the public key $h \leftarrow g^x$.

Encapsulation: Encapsulation proceeds as follows:

- $u \leftarrow \mathbb{Z}/q\mathbb{Z}$.
- $v \leftarrow h^u$.
- $c \leftarrow g^u$.
- $k \leftarrow H(v \| c)$.

Decapsulation: To decapsulate the KEM one takes c and using the private key one computes

- $v \leftarrow c^x$.
- $k \leftarrow H(v \| c)$.

However, to prove this KEM secure we need to introduce a new problem called the Gap Diffie–Hellman problem. This problem assumes that the Diffie–Hellman problem is hard even assuming that the adversary has an oracle to solve the Decision Diffie–Hellman problem. In other words, we are given g^a and g^b and an oracle $\mathcal{O}_{\mathsf{DDH}}$ which on input of (g^x, g^y, g^z) will say whether $z = x \cdot y$. We

Algorithm 16.5: Algorithm B using an $\mathsf{IND\text{-}CCA}$ adversary A against RSA-KEM to solve the RSA problem

B has input N, e, y and is asked to find x such that $x^e \pmod N = y$.
$(\mathfrak{pk}) \leftarrow (N, e)$.
$k \leftarrow \mathbb{K}$, $c^* \leftarrow y$.
H-List $\leftarrow \emptyset$.
Call A with input $\mathfrak{pk}, k, c^*$.

/* A's H Oracle Queries */
A makes a random oracle query with input z.
if $\exists\ (z, c, h) \in H$-List **then return** h.
if $\exists\ (\perp, c, h) \in H$-List with $z^e \pmod N = c$ **then**
  $H\text{-List} \leftarrow \Big(H\text{-List} \cup \{(z, c, h)\}\Big) \setminus \{(\perp, c, h)\}$.
else
  $h \leftarrow \mathbb{K}$.
  $c \leftarrow z^e \pmod N$.
  $H\text{-List} \leftarrow H\text{-List} \cup \{(z, c, h)\}$.
return h

/* A's $\mathcal{O}_{\mathsf{Decap}_{\mathfrak{sk}}}$ Oracle Queries */
A makes an $\mathcal{O}_{\mathsf{Decap}_{\mathfrak{sk}}}$ query with ciphertext c.
if $\exists\ (\cdot, c, h) \in H$-List **then return** h
$h \leftarrow \mathbb{K}$.
$H\text{-List} \leftarrow H\text{-List} \cup \{(\perp, c, h)\}$.
return h.

/* A's Response */
When A returns b'.
if $\exists\ (x, c^*, \cdot) \in H$-List **then return** x.
return $\perp$.

then wish to output $g^{a \cdot b}$. We define $\mathrm{Adv}_G^{\mathsf{Gap\text{-}DHP}}(A)$ as the probability that the algorithm A wins the Diffie–Hellman problem game, given access to an oracle which solves the Decision Diffie–Hellman problem. It is believed that this problem is as hard as the standard Diffie–Hellman problem. Indeed there are some groups in which the Decision Diffie–Hellman problem is easy and the computational Diffie–Hellman problem is believed to be hard.

We can now prove that the DHIES-KEM is secure. Before stating and proving the theorem we pause to point out why we need the Gap Diffie–Hellman problem. In the proof of security of RSA-KEM, Theorem 16.8, the algorithm B's simulation of a valid attack environment to algorithm A was perfect. In other words A could not notice it was playing against someone trying to solve the RSA problem, and not a genuine encryption system. Algorithm B did this by "cooking" the values output by the random oracle, by computing the RSA function in a forwards direction. In the simulation in the theorem below, algorithm B still proceeds with much the same strategy. However, to do a similar cooking of the random oracle algorithm B needs to be able to distinguish Diffie–Hellman tuples, from non-Diffie–Hellman tuples. Thus algorithm B needs a mechanism to do this. Hence, algorithm B needs access to a $\mathcal{O}_{\mathsf{DDH}}$ oracle, and so B does not solve the Diffie–Hellman problem, but the Gap Diffie–Hellman problem.

Theorem 16.9. *In the random oracle model and assuming the Gap Diffie–Hellman problem is hard, there exists no adversary which breaks DHIES-KEM. In particular if A is an adversary which breaks the* IND-CCA *security of the DHIES-KEM scheme Π for the group G, which treats H as a random oracle, then there is an adversary B against the Gap Diffie–Hellman problem for the group G with*

$$\mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}CCA}}(A) = \mathrm{Adv}_{G}^{\mathsf{Gap\text{-}DHP}}(B).$$

PROOF. We provide a sketch of the proof by simply giving algorithm B in Algorithm 16.6, and presenting some comments. Notice that the public key is g^a and the target encapsulation is $c^* = g^b$. Hence, the Diffie–Hellman value which needs to be passed to the key derivation function H to obtain the target encapsulated key is $v^* = g^{a \cdot b}$. Since H is a random oracle the only way A can find out any information about the encapsulated key is to make the query $H(v^* \| c^*)$. Thus the Diffie–Hellman value, if A is successful, will end up on B's H-List. When looking at Algorithm 16.6 you should compare it with Algorithm 16.5.

Algorithm 16.6: Algorithm B using an IND-CCA adversary A against DHIES-KEM to solve the Gap Diffie–Hellman problem

B has input $A = g^a$, $B = g^b$ and is asked to find $C = g^{a \cdot b}$.
$(\mathfrak{pk}) = h \leftarrow A$.
$k \leftarrow \mathbb{K}$, $c^* \leftarrow B$.
H-List $\leftarrow \emptyset$.
Call A with input $\mathfrak{pk}, k, c^*$.

/* A's H Oracle Queries */
A makes a random oracle query with input $z \| c$.
if $\exists\ (z, c, h) \in H$-List **then return** h.
if $\exists (\perp, c, h) \in H$-List such that $\mathcal{O}_{\mathsf{DDH}}(g, A, c, z)$=**true then**
  H-List $\leftarrow \Big(H\text{-List} \cup \{(z, c, h)\}\Big) \setminus \{(\perp, c, h)\}$.
else
  $h \leftarrow \mathbb{K}$.
  H-List $\leftarrow H$-List $\cup \{(z, c, h)\}$.
return h.

/* A's $\mathcal{O}_{\mathsf{Decap}_{\mathfrak{sk}}}$ Oracle Queries */
A makes an $\mathcal{O}_{\mathsf{Decap}_{\mathfrak{sk}}}$ query with ciphertext c.
if $\exists\ (\cdot, c, h) \in H$-List **then return** h.
$h \leftarrow \mathbb{K}$.
H-List $\leftarrow H$-List $\cup \{(\perp, c, h)\}$.
return h.

/* A's Response */
When A returns b'.
if $\exists\ (C, c^*, \cdot) \in H$-List **then return** C
return $\perp$.

□

16.5. Secure Digital Signatures

We already saw in Chapter 15 how the combination of hash functions and the RSA function could be used to produce digital signatures. In particular we presented the RSA-FDH signature scheme. In this section we first prove that this scheme is secure in the random oracle model. However, RSA-FDH requires a hash function with codomain the RSA group. Since such hash functions are not "natural" we also present the RSA-PSS signature scheme which does not have this restriction, and which is also secure in the random oracle model. Having presented these two variants of signatures based on the RSA problem, we then turn to discussing signature schemes based on the discrete logarithm problem.

16.5.1. RSA-FDH: In Chapter 15 we presented the RSA-FDH signature scheme. The proof we outline below for RSA-FDH bears much in common with the proof for RSA-KEM above, especially in the way the hash function is modelled as a random oracle. For RSA-FDH we assume a hash function

$$H : \{0,1\}^* \longrightarrow (\mathbb{Z}/N\mathbb{Z})^*,$$

where N is the RSA modulus of the public key. Again such hash functions are hard to construct in practice, but if we assume they can exist and we model them using a random oracle then we can prove the RSA-FDH signature algorithm is secure.

As above let $f_{N,e}$ denote the function

$$f_{N,e} : \begin{cases} (\mathbb{Z}/N\mathbb{Z})^* \longrightarrow (\mathbb{Z}/N\mathbb{Z})^* \\ x \longmapsto x^e. \end{cases}$$

The RSA-FDH signature algorithm signs a message m as follows

$$s \leftarrow H(m)^d \pmod N = f_{N,e}^{-1}(H(m)),$$

where the private exponent is d. Verification of a signature is performed by checking whether

$$f_{N,e}(s) = s^e \pmod N = H(m).$$

Recall that the RSA problem is given $y = f_{N,e}(x)$ determine x. One can then prove the following theorem.

Theorem 16.10. *In the random oracle model if we model H as a random oracle then the RSA-FDH signature scheme is secure, assuming the RSA problem is hard. In particular if A is* EUF-CMA *adversary against the RSA-FDH signature scheme Π for RSA moduli of v bits in length which performs q_H distinct hash function queries, then there is an algorithm B for the RSA problem such that*

$$\mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\Pi}(A; q_H) = q_H \cdot \mathrm{Adv}^{\mathsf{RSA}}_{v}(B).$$

Note that in this theorem the advantage term has a "security loss" of q_H, this is because algorithm B in the proof needs to "guess" into which query it should embed the RSA problem challenge.

Proof. We describe an algorithm B which on input of $y \in (\mathbb{Z}/N\mathbb{Z})^*$ outputs $x = f_{N,e}^{-1}(y)$. Without loss of generality we can assume that algorithm A always makes a hash function query on a message m before it is passed to its signing oracle. Indeed, if this is not the case then B can make these queries for A.

Algorithm B first chooses a value $t \in [1, \ldots, q_H]$ and throughout keeps a numbered record of all the hash queries made. Algorithm B takes as input N, e and y and sets the public key to be $pk \leftarrow (N, e)$. Algorithm B maintains a hash list H-List as before, which is initially set to the empty set. The public key is then passed to algorithm A.

When algorithm A makes a hash function query for the input m, algorithm B responds as follows:

- If there exists $(m, s, \perp) \in H$-List then return s.

- If this is the tth distinct query to the hash function then B sets
$$H\text{-List} \leftarrow H\text{-List} \cup \{(m, y, \perp)\}$$
and responds with y. We let m^* denote this message.
- Else B picks $s \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$, sets $h \leftarrow s^e \pmod N$ and sets
$$H\text{-List} \leftarrow H\text{-List} \cup \{(m, h, s)\}$$
and responds with h.

If A makes a signing query for a message m then algorithm B responds as follows.

- If message m is equal to m^* then algorithm B stops and returns fail.
- If $m \neq m^*$ then B returns the value s such that $(m, h, s) \in H\text{-List}$.

Let A terminate with output (m, s) and without loss of generality we can assume that A made a hash oracle query for the message m. Now if $m \neq m^*$ then B terminates and admits failure, but if $m = m^*$ then we have
$$f_{N,e}(s) = H(m_t) = y.$$
Hence we have succeeded in inverting f.

In analysing algorithm B one notices that if A terminates successfully then (m^*, s) is an existential forgery and so m^* was not asked of the signing oracle. The value of t is independent of the view of A, so A cannot always ask for the signature of message m^* in the algorithm rather than not ask for the signature. Hence, roughly speaking, the probability of success of B is $1/q_H$ that of the probability of A being successful. □

16.5.2. RSA-PSS: Another way of securely using RSA as a signature algorithm is to use a system called RSA-PSS, or *probabilistic signature scheme*. This scheme can also be proved secure in the random oracle model under the assumption that the RSA problem is hard. We do not give the details of the proof here but simply explain the scheme, which appears in many cryptographic standards. The advantage of RSA-PSS over RSA-FDH is that one only requires a hash function with a traditional codomain, e.g. bit strings of length t, rather than a set of integers modulo another number.

As usual one takes an RSA modulus N, a public exponent e and a private exponent d. Suppose the security parameter is k, i.e. N is a k-bit number. We define two integers k_0 and k_1 so that $k_0 + k_1 \leq k - 1$, such that a work effort of 2^{k_0} and 2^{k_1} is considered infeasible; for example one could take $k_i = 128$ or 160. We then define two hash functions, one which expands data and one which compresses data (just like in RSA-OAEP):
$$G : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k-k_1-1}$$
$$H : \{0,1\}^* \longrightarrow \{0,1\}^{k_1}.$$
We let
$$G_1 : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k_0}$$
denote the function which returns the first k_0 bits of $G(w)$ for $w \in \{0,1\}^{k_1}$ and we let
$$G_2 : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k-k_0-k_1-1}$$
denote the function which returns the last $k - k_0 - k_1 - 1$ bits of $G(w)$ for $w \in \{0,1\}^{k_1}$, i.e. $G(w) = G_1(w) \| G_2(w)$.

Signing: To sign a message m the private key holder performs the following steps:

- $r \leftarrow \{0,1\}^{k_0}$.
- $w \leftarrow H(m \| r)$.
- $y \leftarrow 0 \| w \| (G_1(w) \oplus r) \| G_2(w)$.
- $s \leftarrow y^d \pmod N$.

Verification: To verify a signature (s, m) the public key holder performs the following

- $y \leftarrow s^e \pmod N$.
- Split y into the components
$$b\|w\|\alpha\|\gamma$$
where b is one bit long, w is k_1 bits long, α is k_0 bits long and γ is $k - k_0 - k_1 - 1$ bits long.
- $r \leftarrow \alpha \oplus G_1(w)$.
- The signature is verified as correct if and only if $b = 0$ and $G_2(w) = \gamma$ and $H(m\|r) = w$.

If we allow the modelling of the hash functions G and H by random oracles then one can show that the above signature algorithm is EUF-CMA secure, in the sense that the existence of a successful algorithm to find forgeries could be used to produce an algorithm to invert the RSA function. For the proof of this one should consult the Eurocrypt 1996 paper of Bellare and Rogaway mentioned in the Further Reading section at the end of this chapter.

16.5.3. The Digital Signature Algorithm: We have already presented two secure digital signature schemes, namely RSA-FDH and RSA-PSS. You may ask why do we need another one?

- What if someone breaks the RSA algorithm or finds that factoring is easy?
- RSA is not suited to some applications since signature generation is a very costly operation.
- RSA signatures are very large; some applications require smaller signature footprints.

One algorithm which addresses all of these concerns is the Digital Signature Algorithm, or DSA. One sometimes sees this referred to as the DSS, or Digital Signature Standard. Although originally designed to work in the group $\mathbb{F}_p^*$, where p is a large prime, it is now common to see it used with elliptic curves, in which case it is called EC-DSA. The elliptic curve variants of DSA run very fast and have smaller footprints and key sizes than almost all other signature algorithms. We shall first describe the basic DSA algorithm as it applies to finite fields. In this variant the security is based on the difficulty of solving the discrete logarithm problem in the field $\mathbb{F}_p$.

Domain Parameters: Just as in ElGamal encryption we first need to define the domain parameters, which are identical to those used in ElGamal encryption. These are

- p a 'large prime', by which we mean one with around 2048 bits, such that $p-1$ is divisible by another 'medium prime' q of around 256 bits.
- g an element of $\mathbb{F}_p^*$ of prime order q, i.e. $g = r^{(p-1)/q} \pmod p \neq 1$ for some $r \in \mathbb{F}_p^*$.
- A hash function H which maps bit strings to element in $\mathbb{Z}/q\mathbb{Z}$.

The domain parameters create a public finite abelian group G of prime order q with generator g. Such domain parameters can be shared between a large number of users.

Key Generation: Again key generation is exactly the same as in ElGamal encryption. The private key $\mathfrak{sk}$ is chosen to be an integer $x \leftarrow [0, \ldots, q-1]$, whilst the public key is given by $\mathfrak{pk} = h \leftarrow g^x \pmod p$.

Signing: DSA is a signature with appendix algorithm and the signature produced consists of two elements $r, s \in \mathbb{Z}/q\mathbb{Z}$. To sign a message m the user performs the following steps:

- $h \leftarrow H(m)$.
- $k \leftarrow (\mathbb{Z}/q\mathbb{Z})^*$.
- $r \leftarrow (g^k \pmod p) \pmod q$.
- $s \leftarrow (h + x \cdot r)/k \pmod q$.

The signature on m is then the pair (r, s). Notice that to sign we utilize a secret ephemeral key on every signature. One issue with DSA is that this ephemeral key k really needs to be kept secret and truly random, otherwise attacks like the earlier partial key exposure attacks from Section 15.5 can be deployed.

Verification: To verify the signature (r, s) on the message m for the public key h, the verifier performs the following steps.

- $h \leftarrow H(m)$.
- $a \leftarrow h/s \pmod q$.
- $b \leftarrow r/s \pmod q$.
- $v \leftarrow (g^a \cdot h^b \pmod p) \pmod q$.
- Accept the signature if and only if $v = r$.

As a baby example of DSA consider the following domain parameters

$$q = 13,\ p = 4 \cdot q + 1 = 53 \text{ and } g = 16.$$

Suppose the public/private key pair of the user is given by $x \leftarrow 3$ and $h \leftarrow g^3 \pmod p = 15$. Now, if we wish to sign a message which has hash value $h = 5$, we first generate an ephemeral secret key, for example purposes we shall take $k \leftarrow 2$, and then we compute

$$\begin{aligned} r &\leftarrow (g^k \pmod p) \pmod q = 5, \\ s &\leftarrow (h + x \cdot r)/k \pmod q = 10. \end{aligned}$$

To verify this signature the recipient computes

$$\begin{aligned} a &\leftarrow h/s \pmod q = 7, \\ b &\leftarrow r/s \pmod q = 7, \\ v &\leftarrow (g^a \cdot y^b \pmod p) \pmod q = 5. \end{aligned}$$

Note $v = r$ and so the signature is verified correctly.

The DSA algorithm uses the subgroup of $\mathbb{F}_p^*$ of order q which is generated by g. The private key can clearly be recovered from the public key if the discrete logarithm problem can be solved in the cyclic group $\langle g \rangle$ of order q. Thus, taking into account our discussion of the discrete logarithm problem in Chapter 3, we require for security that

- $p > 2^{2048}$, to avoid attacks via the Number Field Sieve,
- $q > 2^{256}$ to avoid attacks via the Baby-Step/Giant-Step method.

Hence, to achieve the rough equivalent of 128 bits of AES strength we need to operate on integers of roughly 2048 bits in length. This makes DSA slower than RSA, since the DSA operation is more complicated than RSA. For example, the verification operation for an equivalent RSA signatures requires only one exponentiation modulo a 2048-bit number, and even that is an exponentiation by a small number. For DSA, verification requires two exponentiations modulo a 2048-bit number. In addition the signing operation for DSA is more complicated than the procedure for RSA signatures, due to the need to compute the value of s, which requires an inversion modulo q.

The other main problem is that the DSA algorithm really only requires to work in a finite abelian group of size 2^{256}, but since the integers modulo p is susceptible to an attack from the Number Field Sieve we are required to work with group elements of 2048 bits in size. This produces a significant performance penalty.

Luckily we can generalize DSA to an arbitrary finite abelian group in which the discrete logarithm problem is hard. We can then use a group which provides a harder instance of the discrete logarithm problem, for example the group of points on an elliptic curve over a finite field. To explain this generalization, we write $G = \langle g \rangle$ for a group generated by g; we assume that

- g has prime order $q > 2^{256}$,
- the discrete logarithm problem with respect to g is hard,

- there is a public function f such that

$$f : G \longrightarrow \mathbb{Z}/q\mathbb{Z}.$$

We summarize the differences between DSA and EC-DSA in the following table.

Quantity	DSA	EC-DSA
G	$\langle g \rangle < \mathbb{F}_p^*$	$\langle P \rangle < E(\mathbb{F}_p)$
g	$g \in \mathbb{F}_p^*$	$P \in E(\mathbb{F}_p)$
y	g^x	$[x]P$
$f(\cdot)$	$\cdot \pmod q$	$x\text{-coord}(\cdot) \pmod q$

For this generalized form of DSA each user again generates a secret signing key, x. The public key is again given by $h \leftarrow g^x$. Signatures are computed via the steps

- $h \leftarrow H(m)$.
- $k \leftarrow (\mathbb{Z}/q\mathbb{Z})^*$.
- $r \leftarrow f(g^k)$.
- $s \leftarrow (h + x \cdot r)/k \pmod q$.

To verify the signature (r, s) on the message m the verifier performs the following steps.

- $h \leftarrow H(m)$.
- $a \leftarrow h/s \pmod q$.
- $b \leftarrow r/s \pmod q$.
- $v \leftarrow f(g^a \cdot h^b)$.
- Accept the signature if and only if $v = r$.

You should compare this signature and verification algorithm with that given earlier for DSA and spot where they differ. When used for EC-DSA the above generalization is written additively.

EC-DSA Example: As a baby example of EC-DSA take the following elliptic curve

$$Y^2 = X^3 + X + 3,$$

over the field $\mathbb{F}_{199}$. The number of elements in $E(\mathbb{F}_{199})$ is equal to $q = 197$ which is a prime; the elliptic curve group is therefore cyclic and as a generator we can take $P = (1, 76)$. As a private key let us take $x = 29$, and so the associated public key is given by

$$Y = [x]P = [29](1, 76) = (113, 191).$$

Suppose the holder of this public key wishes to sign a message with hash value $H(m)$ equal to 68. They first produce a random ephemeral key, which we shall take to be $k = 153$, and compute

$$\begin{aligned} r &= x\text{-coord}\left([k]P\right) = x\text{-coord}\left([153](1, 76)\right) \\ &= x\text{-coord}\left((185, 35)\right) = 185. \end{aligned}$$

Now they compute

$$\begin{aligned} s &= (H(m) + x \cdot r)/k \pmod q \\ &= (68 + 29 \cdot 185)/153 \pmod{197} \\ &= 78. \end{aligned}$$

The signature is then the pair $(r, s) = (185, 78)$.

To verify this signature we compute

$$\begin{aligned} a &= H(m)/s \pmod q = 68/78 \pmod{197} = 112, \\ b &= r/s \pmod q = 185/78 \pmod{197} = 15. \end{aligned}$$

We then compute

$$\begin{aligned} Z &= [a]P + [b]Y = [112](1,76) + [15](113,191) \\ &= (111,60) + (122,140) = (185,35). \end{aligned}$$

The signature is now verified since we have

$$r = 185 = x\text{-coord}(Z).$$

It is believed that DSA and EC-DSA do provide secure signature algorithms, in the sense of EUF-CMA, however no proof of this fact is known in the standard model or the random oracle model. However, if instead of modelling the hash function as an ideal object (as in the random oracle model), we model the group as an ideal object (something called the generic group model) then we can show that EC-DSA is EUF-CMA secure. But this uses techniques beyond the scope of this book.

16.5.4. Schnorr Signatures: There are many variants of the DSA signature scheme based on discrete logarithms. A particularly interesting one is that of Schnorr signatures. We present the algorithm in the general case and allow the reader to work out the differences between the elliptic curve and finite field variants.

Suppose G is a public finite abelian group generated by an element g of prime order q. The public/private key pairs are just the same as in DSA, namely

- The private key is an integer x in the range $0 < x < q$.
- The public key is the element $h \leftarrow g^x$.

Signing: To sign a message m using the Schnorr signature algorithm the signer performs the following steps:

(1) $k \leftarrow \mathbb{Z}/q\mathbb{Z}$.
(2) $r \leftarrow g^k$.
(3) $e \leftarrow H(m\|r)$.
(4) $s \leftarrow k + x \cdot e \pmod q$.

The signature is then given by the pair (e, s). Notice how the hash function depends both on the message and the ephemeral public key r; we will see this is crucial in order to establish the security results below. In addition, notice that the signing equation is easier than that used for DSA, as we do not require a modular inversion modulo q.

Verification: The verification step is very simple:

- $r \leftarrow g^s \cdot h^{-e}$.
- The signature is accepted if and only if $e = H(m\|r)$.

Schnorr Signature Example: As an example of Schnorr signatures in a finite field we take the domain parameters $q = 101$, $p = 607$ and $g = 601$. As the public/private key pair we assume $x \leftarrow 3$ and $h \leftarrow g^x \pmod p = 391$. Then to sign a message we generate an ephemeral key, let's take $k \leftarrow 65$, and compute $r \leftarrow g^k \pmod p = 223$. We now need to compute the hash value $e \leftarrow h(m\|r) \pmod q$. Let us assume that we compute $e = 93$; then the second component of the signature is given by

$$s \leftarrow k + x \cdot e \pmod q = 65 + 3 \cdot 93 \pmod{101} = 41.$$

We leave it to the reader to check that the signature (e, s) verifies, i.e. that the verifier recovers the same value of r.

Schnorr Authentication Protocols: Schnorr signatures have been suggested for use in challenge response mechanisms in smart cards since the response part of the signature (the value of s) is particularly easy to evaluate because it only requires the computation of a single modular multiplication and a single modular addition. No matter what group we choose this final phase only requires arithmetic modulo a relatively small prime number.

To see how one uses Schnorr signatures in a challenge response situation we give the following scenario. You wish to use a smart card to authenticate yourself to a building or ATM machine. The card reader has a copy of your public key h, whilst the card has a copy of your private key x. Whilst you are walking around the card is generating commitments, which are ephemeral public keys of the form $r = g^k$.

When you place your card into the card reader the card transmits to the reader the value of one of these precomputed commitments. The card reader then responds with a challenge message e. Your card then only needs to compute

$$s = k + x \cdot e \pmod{q},$$

and transmit it to the reader, which then verifies the 'signature' by checking whether

$$g^s = r \cdot h^e.$$

Notice that the only online computations needed by the card are the computations of the values of e and s, which are both easy to perform.

In more detail, if we let C denote the card and R denote the card reader then we have

$$\begin{aligned} C &\longrightarrow R : r = g^k, \\ R &\longrightarrow C : e, \\ C &\longrightarrow R : s = k + xe \pmod{q}. \end{aligned}$$

The point of the initial commitment is to stop either the challenge being concocted so as to reveal your private key, or your response being concocted so as to fool the reader. A three-phase protocol consisting of

$$\text{commitment} \longrightarrow \text{challenge} \longrightarrow \text{response}$$

is a common form of authentication protocol, and we shall see more protocols of this nature when we discuss zero-knowledge proofs in Chapter 21.

Schnorr Signature Security: We shall now prove that Schnorr signatures are EUF-CMA secure in the random oracle model. The proof uses something called the forking lemma, which we present without proof; see the paper of Pointcheval and Stern mentioned at the end of this chapter.

Lemma 16.11 (Forking Lemma)**.** *Let A be a randomized algorithm with inputs $(x, h_1, \ldots, h_q, r)$, where r is the randomness used by A drawn from a distribution R, and the values $h_1, \ldots, h_q$ are selected from a set $\mathcal{H}$ uniformly at random.*

Assume that A outputs a pair (t, y). Let ϵ_A be the probability that the value t output by A is in the range $[1, \ldots, q]$. Define the algorithm B in Algorithm 16.7, which has input x, and let ϵ_B denote the probability that B outputs a non-zero tuple. Then

$$\epsilon_B \geq \frac{\epsilon_A^2}{q} - \frac{\epsilon_A}{|\mathcal{H}|}.$$

How we think about (and use) the A and B of the Lemma is as follows. Algorithm A is assumed to be running in the random oracle model, and the h_i values are the responses it receives to its random oracle queries. There is a special query which A uses in producing its answer y; this is called the *critical query*, and it is denoted by t in Algorithm 16.7. If $t = 0$ then algorithm A is not successful. We now run A again, with the same random tape and the same random oracle, up until the tth query. At this point the random oracle changes. At this point, which recall was the

Algorithm 16.7: Forking algorithm B

$r \leftarrow R$.
$h_1, \ldots, h_q \leftarrow \mathcal{H}$.
$(t, y) \leftarrow A(x, h_1, \ldots, h_q, r)$.
if $t = 0$ **then return** $(0, 0)$.
$h'_t, \ldots, h'_q \leftarrow \mathcal{H}$.
$(t', y') \leftarrow A(x, h_1, \ldots, h_{t-1}, h'_t, \ldots, h'_q, r)$.
if $t \neq t'$ or $h_t = h'_t$ **then return** $(0, 0)$.
return (y, y').

critical query, the second execution of A "forks" down another path (giving the lemma its name). The lemma tells us a lower bound on the probability that the two runs of A result in the same value for the critical query, assuming it is distinct.

The lemma is important in analysing signature schemes which are of the form *commit*, *challenge*, *response*. To realize such signatures we use the following notation. To sign a message

- The signer produces a (possibly empty) commitment σ_1 (the commitment).
- The signer computes $e = H(\sigma_1 \| m)$ (the challenge).
- The signer computes σ_2 which is the 'signature' on σ_1 and e (the response).

We label the output of the signature schemes as $(\sigma_1, H(\sigma_1 \| m), \sigma_2)$ so as to keep track of the exact hash query; DSA, EC-DSA and Schnorr signatures are all of this form:

- DSA : $\sigma_1 = \emptyset$, $e = H(m)$, $\sigma_2 = (r, (e + x \cdot r)/k \pmod q)$ where $r = \left(g^k \pmod p\right) \pmod q$.
- EC-DSA : $\sigma_1 = \emptyset$, $e = H(m)$, $\sigma_2 = (r, (e + x \cdot r)/k \pmod q)$, where $r = x\text{-coord}([k]G)$.
- Schnorr signatures: $\sigma_1 = g^k$, $e = H(\sigma_1 \| m)$ $\sigma_2 = x \cdot e + k \pmod q$.

In all of these schemes the hash function is assumed to have codomain equal to $\mathbb{F}_q$.

Recall that in the random oracle model the hash function is allowed to be cooked up by the algorithm B to do whatever it likes. Suppose an adversary A can produce an existential forgery on a message m with non-negligible probability in the random oracle model. Hence, the output of the adversary is

$$(m, \sigma_1, e, \sigma_2).$$

We can assume that the adversary makes the critical hash query for the forged message, $e = H(\sigma_1 \| m)$, since otherwise we can make the query for the adversary ourselves.

Algorithm B now runs the adversary A again, just as in the forking lemma, with the same random tape and the modified random oracle. Up until the critical query, the hash queries were answered the same way as before, so we have that the execution of A in both runs is identical up until the critical query. If Algorithm 16.7 is successful this means the two executions output two tuples

$$y = (m, \sigma_1, e, \sigma_2) \text{ and } y' = (m', \sigma'_1, e', \sigma'_2),$$

where e and e' are the outputs from the critical query, but the inputs to this query are *the same*. In other words we have $m = m'$ and $\sigma_1 = \sigma'_1$. This last equation does not give us anything in the case of DSA or EC-DSA, since in those cases σ_1 is always equal to $\emptyset$. However, for Schnorr signatures we obtain something useful, since we find $g^k = g^{k'}$, where k and k' are the underlying ephemeral keys of the two signatures. Note that A might not even know k and k' in running her attack code. This allows us to deduce $k = k'$, and hence to recover the secret key x from the equation

$$x = \frac{e - e'}{\sigma_2 - \sigma'_2}.$$

Notice that the denominator here is non-zero by assumption. This is the basic idea behind the proof of the following theorem.

Theorem 16.12. *In the random oracle model let A denote a* EUF-CMA *adversary against Schnorr signatures with advantage ϵ, making q_H queries to its hash function H. Then there is an adversary B against the discrete logarithm problem with advantage ϵ' such that*

$$\epsilon' \geq \frac{\epsilon^2}{q} - \frac{\epsilon}{q}.$$

Proof. Algorithm B has as input g and $h = g^x$ and it wishes to find x. The value h will be used as the public key by algorithm A. We first 'package' A into an algorithm A' which does not require access to a signing oracle as follows. The algorithm A' will take as input $(h, h_1, \ldots, h_q, r)$, where r is an entry from the set of possible random tapes of algorithm A. When A makes a signing query on the message m then algorithm A' executes the following steps:

- Take the next hash query value input to A', let this be value h_i.
- $s \leftarrow \mathbb{Z}/q\mathbb{Z}$.
- $r \leftarrow g^s/h^e$.
- Define $H(m\|r) = h_i$. If this value has already been defined then pick another value of s.

The hash function queries are handled in the usual way, using the inputs $h_1, \ldots, h_q$. If A does not terminate with a forged signature then A' output $(0, 0)$, otherwise it outputs (t, y) where t is the index of the critical hash query and

$$y = (m, \sigma_1, e, \sigma_2),$$

with $h_t = e$. Algorithm A' is now an algorithm which we can use in the forking lemma. This gives us an algorithm B which will produce two values y and y', from which we can recover x via the above method. □

16.5.5. Nyberg–Rueppel Signatures: What happens when we want to sign a general message which is itself quite short? It may turn out that the signature could be longer than the message. Recall that RSA can be used either as a scheme with appendix or as a scheme with message recovery. So far none of our discrete-logarithm-based schemes can be used with message recovery. We end this section by giving an example scheme which does have the message recovery property, called the Nyberg–Rueppel signature scheme, which is based on discrete logarithms in some public finite abelian group G.

Many signature schemes with message recovery require a public redundancy function R. This function maps actual messages over to the data which is actually signed. This acts rather like a hash function does in the schemes based on signatures with appendix. However, unlike a hash function the redundancy function must be easy to invert. As a simple example we could take R to be the function

$$R : \begin{cases} \{0,1\}^{n/2} \longrightarrow \{0,1\}^n \\ m \longmapsto m\|m. \end{cases}$$

We assume that the codomain of R can be embedded into the group G. In our description we shall use the integers modulo p, i.e. $G = \mathbb{F}_p^*$, and as usual we assume that a large prime q divides $p - 1$ and that g is a generator of the subgroup of order q. Once again the public/private key pair is given as a discrete logarithm problem $\mathfrak{pk} \leftarrow h = g^x$.

Signing: Nyberg–Rueppel signatures are then produced as follows:

(1) $k \leftarrow \mathbb{Z}/q\mathbb{Z}$.
(2) $r \leftarrow g^k \pmod p$.
(3) $e \leftarrow R(m) \cdot r \pmod p$.
(4) $s \leftarrow x \cdot e + k \pmod q$.

The signature is then the pair (e, s).

Verification and Recovery: From the pair (e, s) and the public key h we need to

- Verify that the signature comes from the user with public key h,
- Recover the message m from the pair (e, s).

This is performed as follows:

(1) $u_1 \leftarrow g^s \cdot h^{-e} = g^{s-e\cdot x} = g^k \pmod p$.
(2) $u_2 \leftarrow e/u_1 \pmod p$.
(3) Verify that u_2 lies in the range of the redundancy function, e.g. we must have $u_2 = R(m) = m\|m$. If this does not hold then reject the signature.
(4) Recover the message $m = R^{-1}(u_2)$ and accept the signature.

Example: As an example we take the domain parameters $q = 101$, $p = 607$, $g = 601$, and as the redundancy function we take $R(m) = m + 2^4 \cdot m$, where a message m must lie in $[0, \ldots, 15]$. As the public/private key pair we assume $x \leftarrow 3$ and $h \leftarrow g^x \pmod p = 391$. To sign the message $m = 12$ we compute an ephemeral key $k \leftarrow 45$ and $r \leftarrow g^k \pmod p = 143$. Since $R(m) = m + 2^4 \cdot m$ we have $R(m) = 204$. We then compute $e \leftarrow R(m) \cdot r \pmod p = 36$, $s \leftarrow x \cdot e + k \pmod q = 52$. The signature is then the pair $(e, s) = (36, 52)$.

We now show how this signature is verified and the message recovered. We first compute $u_1 = g^s \cdot h^{-e} = 143$. Notice how the verifier has computed u_1 to be the same as the value of r computed by the signer. The verifier now computes $u_2 = e/u_1 \pmod p = 204$. The verifier now checks that $u_2 = 204$ is of the form $m + 2^4 m$ for some value of $m \in [0, \ldots, 15]$. We see that u_2 is of this form and so the signature is valid. The message is then recovered by solving for m in $m + 2^4 m = 204$, from which we obtain $m = 12$.

16.6. Schemes Avoiding Random Oracles

In the previous sections we looked at signature and encryption schemes which can be proved secure in the so-called 'random oracle model'. A proof in the random oracle model only provides *evidence* that a scheme may be secure in the real world, it does not guarantee security in the real world. We can interpret a proof in the random oracle model as saying that if an adversary against the real-world scheme exists then that adversary must make use of the specific hash function employed.

In this section we sketch recent ways in which researchers have tried to construct signature and encryption algorithms which do not depend on the random oracle model, i.e. schemes in the standard model. We shall only consider schemes which are practical, and we shall only sketch the proof ideas. Readers interested in the details of proofs or in other schemes should consult the extensive literature in this area.

What we shall see is that whilst quite natural encryption algorithms can be proved secure without the need for random oracles, the situation is quite different for signature algorithms. This should not be surprising since signature algorithms make extensive use of hash functions for their security. Hence, we should expect that they impose stricter restraints on such hash functions, which may not actually be true in the real world.

16.6.1. The Cramer–Shoup Encryption Scheme: Unlike the case of signature schemes in the standard model, for encryption algorithms one can produce provably secure systems which are practical and close to those used in 'real life'. The Cramer–Shoup encryption scheme requires as domain parameters a finite abelian group G of prime order q. In addition we require a one-way family of hash functions. This is a family $\{H_i\}$ of hash functions for which it is hard for an adversary to choose an input x, then to draw a random hash function H_i, and then to find a different input y so that

$$H_i(x) = H_i(y).$$

Key Generation: A public key in the Cramer–Shoup scheme is chosen as follows. First the following random elements are selected

$$g_1, g_2 \leftarrow G,$$
$$x_1, x_2, y_1, y_2, z \leftarrow \mathbb{Z}/q\mathbb{Z}.$$

The user then computes the following elements

$$c \leftarrow {g_1}^{x_1} \cdot {g_2}^{x_2},$$
$$d \leftarrow {g_1}^{y_1} \cdot {g_2}^{y_2},$$
$$h \leftarrow {g_1}^{z}.$$

The user finally chooses a hash function H from the universal one-way family of hash functions and outputs the public key $\mathfrak{pk} \leftarrow (g_1, g_2, c, d, h, H)$, whilst keeping secret the private key $\mathfrak{sk} \leftarrow (x_1, x_2, y_1, y_2, z)$.

Encryption: The encryption algorithm proceeds as follows, which is very similar to ElGamal encryption. The message m is considered as an element of G, and encryption proceeds as follows:

$$r \leftarrow \mathbb{Z}/q\mathbb{Z},$$
$$u_1 \leftarrow {g_1}^{r},$$
$$u_2 \leftarrow {g_2}^{r},$$
$$e \leftarrow m \cdot h^{r},$$
$$\alpha \leftarrow H(u_1 \| u_2 \| e),$$
$$v \leftarrow c^{r} \cdot d^{r\alpha}.$$

The ciphertext is then the quadruple (u_1, u_2, e, v).

Decryption: On receiving this ciphertext the owner of the private key can recover the message as follows: First they compute $\alpha \leftarrow H(u_1 \| u_2 \| e)$ and test whether

$${u_1}^{x_1 + y_1\alpha} \cdot {u_2}^{x_2 + y_2\alpha} = v.$$

If this equation does not hold then the ciphertext should be rejected. If this equation holds then the receiver can decrypt the ciphertext by computing

$$m \leftarrow \frac{e}{{u_1}^{z}}.$$

Notice that, whilst very similar to ElGamal encryption, the Cramer–Shoup encryption scheme is much less efficient. Hence, whilst provably secure it is not used much in practice.

Security: To show that the scheme is provably secure, under the assumption that the DDH problem is hard and that H is chosen from a universal one-way family of hash functions, we assume we have an adversary A against the scheme and show how to use A in another algorithm B which tries to solve the DDH problem.

One way to phrase the DDH problem is as follows: Given $(g_1, g_2, u_1, u_2) \in G$ determine whether this quadruple is a random quadruple or we have $u_1 = {g_1}^{r}$ and $u_2 = {g_2}^{r}$ for some value of $r \in \mathbb{Z}/q\mathbb{Z}$. So algorithm B will take as input a quadruple $(g_1, g_2, u_1, u_2) \in G$ and try to determine whether this is a random quadruple or a quadruple related to the Diffie–Hellman problem.

Algorithm B first needs to choose a public key, which it does in a non-standard way, by first selecting the random elements

$$x_1, x_2, y_1, y_2, z_1, z_2 \in \mathbb{Z}/q\mathbb{Z}.$$

Algorithm B_A then computes the following elements

$$c \leftarrow {g_1}^{x_1} \cdot {g_2}^{x_2},$$
$$d \leftarrow {g_1}^{y_1} \cdot {g_2}^{y_2},$$
$$h \leftarrow {g_1}^{z_1} \cdot {g_2}^{z_2}.$$

Finally B chooses a hash function H from the universal one-way family of hash functions and outputs the public key $\mathfrak{pk} \leftarrow (g_1, g_2, c, d, h, H)$. Notice that the part of the public key corresponding to h has been chosen differently than in the real scheme, but that algorithm A will not be able to detect this change. Algorithm B now runs algorithm A, responding to decryption queries of (u_1', u_2', e', v') by computing

$$m \leftarrow \frac{e'}{{u_1'}^{z_1} {u_2'}^{z_2}},$$

after performing the standard check on validity of the ciphertext.

At some point A calls its $\mathcal{O}_{\mathsf{LR}}$ oracle on the two plaintexts m_0 and m_1. Algorithm B chooses a bit b at random and computes the target ciphertext as

$$e \leftarrow m_b \cdot \left({u_1}^{z_1} \cdot {u_2}^{z_2}\right),$$
$$\alpha \leftarrow H(u_1 \| u_2 \| e),$$
$$v \leftarrow {u_1}^{x_1 + y_1 \alpha} \cdot {u_2}^{x_2 + y_2 \alpha}.$$

The target ciphertext is then the quadruple (u_1, u_2, e, v). Notice that when the input to B is a legitimate DDH quadruple then the target ciphertext will be a valid encryption, but when the input to B is not a legitimate DDH quadruple then the target ciphertext is highly likely to be an invalid ciphertext. This target ciphertext is then returned to the adversary A.

If the adversary outputs the correct value of b then we suspect that the input to B is a valid DDH quadruple, whilst if the output is wrong then we suspect that the input to B is not valid. This produces a statistical test to detect whether the input to B was valid or not. By repeating this test a number of times we can produce as accurate a statistical test as we want.

Note that the above is only a sketch. We need to show that the view of the adversary A in the above game is no different from that in a real attack on the system, otherwise A would know something was not correct. For example we need to show that the responses B makes to the decryption queries of A cannot be distinguished from a true decryption oracle. For further details one should consult the full proof in the paper mentioned in the Further Reading section.

16.6.2. Cramer–Shoup Signatures: We have already remarked that signature schemes which are provably secure, without the random oracle model, are hard to come by. They also appear somewhat contrived compared with the schemes such as RSA-PSS, DSA or Schnorr signatures which are used in real life. The first such provably secure signature scheme in the standard model was by Goldwasser, Micali and Rivest. This was however not very practical as it relied on messages being associated with leaves of a binary tree, and each node in the tree needed to be authenticated with respect to its parent. This made the resulting scheme far too slow.

However, even with today's knowledge the removal of the use of random oracles comes at a significant price. We need to make stronger intractability assumptions than we have otherwise made. In this section we introduce a new RSA-based problem called the Flexible RSA Problem. This is a potentially easier problem than ones we have met before, hence the assumption that the problem is hard is a much stronger assumption than before. The *strong RSA assumption* is the assumption that the Flexible RSA Problem is hard.

Definition 16.13 (Flexible RSA Problem)**.** *Given an RSA modulus* $N = p \cdot q$ *and a random* $c \in (\mathbb{Z}/N\mathbb{Z})^*$ *find* $e > 1$ *and* $m \in (\mathbb{Z}/N\mathbb{Z})^*$ *such that*

$$m^e = c.$$

Clearly if we can solve the RSA problem then we can solve the Flexible RSA Problem. This means that the strong RSA assumption is a stronger intractability assumption than the standard RSA assumption, in that it is conceivable that in the future we may be able to solve the Flexible RSA Problem but not the traditional RSA problem. However, at present we conjecture that both problems are equally hard.

The Cramer–Shoup signature scheme is based on the strong RSA assumption and is provably secure, without the need for the random oracle model. The main efficiency issue with the scheme is that the signer needs to generate a new prime number for each signature produced, which can be rather costly. In our discussion below we shall assume H is a 'standard' hash function which outputs bit strings of 256 bits, which we interpret as 256-bit integers as usual.

Key Generation: To generate the public key, we create an RSA modulus N which is the product of two "safe" primes p and q, i.e. $p \leftarrow 2 \cdot p' + 1$ and $q \leftarrow 2 \cdot q' + 1$ where p' and q' are primes. We also choose two random elements

$$h, x \in Q_N,$$

where, as usual, Q_N is the set of quadratic residues modulo N. We also create a random 256-bit prime e'. The public key consists of

$$(N, h, x, e')$$

and the private key is the factors p and q.

Signing: To sign a message the signer generates another 256-bit prime number e and another random element $y' \in Q_N$. Since they know the factors of N, the signer can compute the solution y to the equation

$$y = \left(x \cdot h^{H(x')}\right)^{1/e} \pmod{N},$$

where x' satisfies

$$x' = y'^{e'} \cdot h^{-H(m)}.$$

The output of the signer is (e, y, y').

Verification: To verify a message the verifier first checks that e' is an odd number satisfying $e \neq e'$. Then the verifier computes

$$x' \leftarrow y'^{e'} \cdot h^{-H(m)}$$

and then checks that

$$x = y^e \cdot h^{-H(x')}.$$

Security: On the assumption that H is a collision resistant hash function and the Flexible RSA Problem is hard, one can prove that the above scheme is secure against active adversaries. We sketch the most important part of the proof, but the full details are left to the interested reader to look up in the paper mentioned at the end of this chapter.

Assume the adversary makes t queries to a signing oracle. We want to use the adversary A to create an algorithm B to break the strong RSA assumption for the modulus N. Before setting up the public key for input to algorithm A, the algorithm B first decides on what prime values e_i it will output in the signature queries. Then, having knowledge of the e_i, the algorithm B concocts values for the h and x in the public key, so that it always knows the e_ith root of h and x.

Thus when given a signing query for a message m_i, algorithm B can then compute a valid signature, without knowing the factorization of N, by generating $y_i' \in Q_N$ at random and then computing

$$x_i' \leftarrow y_i'^{e'} \cdot h^{-H(m_i)} \pmod{N}$$

and then

$$y_i \leftarrow x^{1/e_i} \cdot (h^{1/e_i})^{H(x_i')} \pmod{N},$$

the signature being given by

$$(m_i, y_i, y_i').$$

The above basic signing simulation is modified in the full proof, depending on what type of forgery algorithm A is producing. But the basic idea is that B creates a public key to enable it to respond to every signing query in a valid way.

Chapter Summary

- ElGamal encryption is a system based on the difficulty of the Diffie–Hellman problem (DHP).
- Paillier encryption is a scheme which is based on the Composite Decision Residuosity Problem (CDRP).
- The random oracle model is a computational model used in provable security. A proof in the random oracle model does not mean the system is secure in the real world, it only provides *evidence* that it *may* be secure.
- In the random oracle model one can prove that the ubiquitous RSA encryption method, namely RSA-OAEP, is secure.
- We presented the KEM-DEM paradigm for designing public key encryption schemes.
- We gave the RSA-KEM (resp. DHIES-KEM) and showed why it is secure, assuming the RSA (resp. Gap Diffie–Hellman) problem is hard.
- In the random oracle model the two main RSA based signature schemes used in 'real life' are also secure, namely RSA-FDH and RSA-PSS.
- DSA is a signature algorithm based on discrete logarithms; it has reduced bandwidth compared with RSA but is slower. EC-DSA is the elliptic curve variant of DSA; it has reduced bandwidth and greater efficiency compared to DSA.
- In the random oracle model one can use the forking lemma to show that the Schnorr signature scheme is secure.
- The Flexible RSA Problem is a natural weakening of the standard RSA problem.
- The Cramer–Shoup encryption scheme is provably secure, without the random oracle model, assuming the DDH problem is hard. It is around three times slower than ElGamal encryption.

Further Reading

The paper by Cramer and Shoup on public key encryption presents the basics of hybrid encryption in great detail, as well as the scheme for public key encryption without random oracles. The other paper by Cramer and Shoup presents their signature scheme. The DHIES scheme was first presented in the paper by Abdalla et al. A good paper to look at for various KEM constructions is that by Dent. The full proof of the security of RSA-OAEP is given in the paper of Fujisaki et al.

A good description of the forking lemma and its applications is given in the article of Pointcheval and Stern. The random oracle model and a number of applications including RSA-FDH and RSA-PSS are given in the papers of Bellare and Rogaway.

M. Abdalla, M. Bellare and P. Rogaway. *DHAES: An encryption scheme based on the Diffie–Hellman problem*. Submission to IEEE P1363a standard.

M. Bellare and P. Rogaway. *Random oracles are practical: a paradigm for designing efficient protocols*. In Proc. 1st Annual Conf. on Comp. and Comms. Security, 62–73, ACM, 1993.

M. Bellare and P. Rogaway. *The exact security of digital signatures – How to sign with RSA and Rabin*. In Advances in Cryptology – Eurocrypt 1996. LNCS 1070, 399–416, Springer, 1996.

R. Cramer and V. Shoup. *Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack*. SIAM Journal on Computing **33**, 167–226, 2003.

R. Cramer and V. Shoup. *Signature schemes based on the strong RSA assumption*. ACM Transactions on Information and Systems Security, **3**, 161–185, 2000.

A. Dent. *A designer's guide to KEMs*. In Cryptography and Coding – 2003, LNCS 2898, 133–151, Springer, 2003.

E. Fujisaki, T. Okamoto, D. Pointcheval and J. Stern. *RSA-OAEP is secure under the RSA assumption*. In Advances in Cryptology – Crypto 2001, LNCS 2139, 260–274, Springer, 2001.

D. Pointcheval and J. Stern. *Security arguments for digital signatures and blind signatures*. J. Cryptology, **13**, 361–396, 2000.