Chapter 10
Digital Signatures

Digital signatures are one of the most important cryptographic tools they and are widely used today. Applications for digital signatures range from digital certificates for secure e-commerce to legal signing of contracts to secure software updates. Together with key establishment over insecure channels, they form the most important instance for public-key cryptography.

Digital signatures share some functionality with handwritten signatures. In particular, they provide a method to assure that a message is authentic to one user, i.e., it in fact originates from the person who claims to have generated the message. However, they actually provide much more functionality, as we'll learn in this chapter.

In this chapter you will learn:

- The principle of digital signatures
- Security services, that is, the specific objectives that can be achieved by a security system
- The RSA digital signature scheme
- The Elgamal digital signature scheme and two extensions of it, the digital signature algorithm (DSA) and the elliptic curve digital signature algorithm (ECDSA)

Christof Paar et al., *Understanding Cryptography*

10.1 Introduction

In this section, we first provide a motivating example why digital signatures are needed and why they must be based on asymmetric cryptography. We then develop the principles of digital signatures. Actual signature algorithms are introduced in subsequent sections.

10.1.1 Odd Colors for Cars, or: Why Symmetric Cryptography Is Not Sufficient

The crypto schemes that we have encountered so far had two main goals: either to encrypt data (e.g., with AES, 3DES or RSA encryption) or to establish a shared key (e.g., with the Diffie–Hellman or elliptic curve key exchange). One might be tempted to think that we are now in a position to satisfy any security needs that arise in practice. However, there are many other security needs besides encryption and key exchange, which are in fact termed security services; these are discussed in detail in Sect. 10.1.3. We now discuss a setting in which symmetric cryptography fails to provide a desirable security function.

Assume we have two communicating parties, Alice and Bob, who share a secret key. Furthermore, the secret key is used for encryption with a block cipher. When Alice receives and decrypts a message which makes semantic sense, e.g., the decrypted message is an actual (English) text, she can in many cases conclude that the message was in fact generated by a person with whom he shares the secret key[1]. If only Alice and Bob know the key, they can be reasonably sure that an attacking third party has not changed the message in transit. So far we've always assumed that the bad guy is an external party that we often named Oscar. However, in practice it is often the case that Alice and Bob do want to communicate securely with each other, but at the same time they might be interested in cheating each other. It turns out that symmetric-key schemes do not protect the two parties *against each other*. Consider the following scenario:

Suppose that Alice owns a dealership for new cars where you can select and order cars online. We assume that Bob, the customer, and Alice, the dealer, have established a shared secret k_{AB}, e.g., by using the Diffie–Hellman key exchange. Bob now specifies the car that he likes, which includes a color choice of pink for the interior and an external color of orange — choices most people would not make. He sends the order form AES-encrypted to Alice. She decrypts the order and is happy to have sold another model for $25,000. Upon delivery of the car three weeks later, Bob has second thoughts about his choice, in part because his spouse is threatening

[1] One has to be a bit careful with such a conclusion, though. For instance, if Alice and Bob use a stream cipher an attacker can flip individual bits of the ciphertext, which results in bit flips in the received plaintext. Depending on the application, the attacker might be able to manipulate the message in a way that is semantically still correct. However, using block ciphers, especially in a chaining mode, makes it quite likely that ciphertext manipulations can be detected after decryption.

him with divorce after *seeing* the car. Unfortunately for Bob (and his family), Alice has a "no return" policy. Given that she is an experienced car dealer, she knows too well that it will not be easy to sell a pink and orange car, and she is thus set on not making any exceptions. Since Bob now claims that he never ordered the car, she has no other choice but to sue him. In front of the judge, Alice's lawyer presents Bob's digital car order together with the encrypted version of it. Obviously, the lawyer argues, Bob must have generated the order since he is in possession of k_{AB} with which the ciphertext was generated. However, if Bob's lawyer is worth his money, he will patiently explain to the judge that the car dealer, Alice, also knows k_{AB} and that Alice has, in fact, a high incentive to generate faked car orders. The judge, it turns out, has no way of knowing whether the plaintext–ciphertext pair was generated by Bob or Alice! Given the laws in most countries, Bob probably gets away with his dishonesty.

This might sound like a rather specific and somewhat artificially constructed scenario, but in fact it is not. There are many, many situations where it is important to prove to a neutral third party, i.e., a person acting as a judge, that one of two (or more) parties generated a message. By *proving* we mean that the judge can conclude without doubt who has generated the message, even if all parties are potentially dishonest. Why can't we use some (complicated) symmetric-key scheme to achieve this goal? The high-level explanation is simple: Exactly because we have a symmetric set-up, Alice and Bob have the same knowledge (namely of keys) and thus the same capabilities. Everything that Alice can do can be done by Bob, too. Thus, a neutral third party cannot distinguish whether a certain cryptographic operation was performed by Alice or by Bob or by both. Generally speaking, the solution to this problem lies in public-key cryptography. The asymmetric set-up that is inherent in public-key algorithms might potentially enable a judge to distinguish between actions that only one person can perform (namely the person in possession of the private key), and those that can be done by both (namely computations involving the public key). It turns out that digital signatures are public-key algorithms which have the properties that are needed to resolve a situation of cheating participants. In the e-commerce car scenario above, Bob would have been required to digitally sign his order using his private key.

10.1.2 Principles of Digital Signatures

The property of proving that a certain person generated a message is obviously also very important outside the digital domain. In the real, "analog" world, this is achieved by handwritten signatures on paper. For instance, if we sign a contract or sign a check, the receiver can prove to a judge that we actually signed the message. (Of course, one can try to forge signatures, but there are legal and social barriers that prevent most people from even attempting to do so.) As with conventional handwritten signatures, only the person who creates a digital message must be capable of generating a valid signature. In order to achieve this with cryptographic primi-

tives, we have to apply public-key cryptography. The basic idea is that the person who signs the message uses a private key, and the receiving party uses the matching public key. The principle of a digital signature scheme is shown in Fig. 10.1.

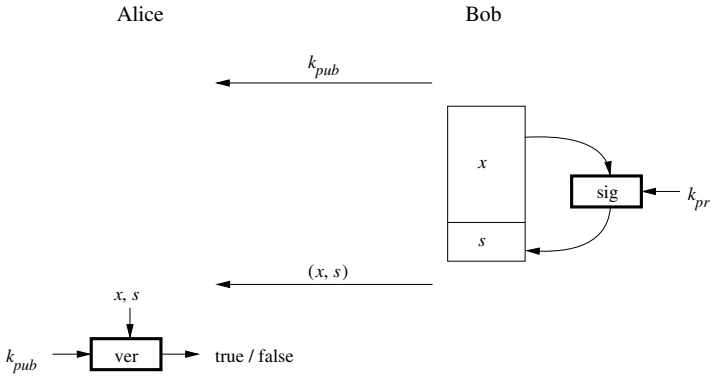
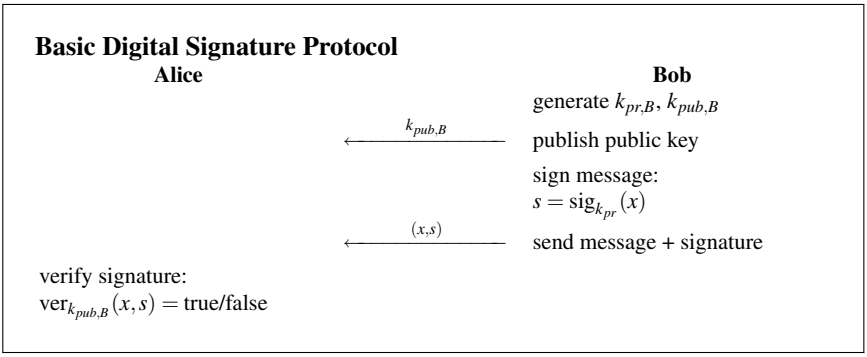


Fig. 10.1 Principle of digital signatures which involves signing and verifying a message

The process starts with Bob signing the message x. The signature algorithm is a function of Bob's private key, k_{pr}. Hence, assuming he in fact keeps his private key private, only Bob can sign a message x on his behalf. In order to relate a signature to the message, x is also an input to the signature algorithm. After signing the message, the signature s is appended to the message x and the pair (x, s) is sent to Alice. It is important to note that a digital signature by itself is of no use unless it is accompanied by the message. A digital signature without the message is the equivalent of a handwritten signature on a strip of paper without the contract or a check that is supposed to be signed.

The digital signature itself is merely a (large) integer value, for instance, a string of 2048 bits. The signature is only useful to Alice if she has means to *verify* whether the signature is valid or not. For this, a verification function is needed which takes both x and the signature s as inputs. In order to link the signature to Bob, the function also requires his public key. Even though the verification function has long inputs, its only output is the binary statement "true" or "false". If x was actually signed with the private key that belongs to the public verification key, the output is true, otherwise it is false.

From these general observations we can easily develop a generic digital signature protocol:



From this set-up, the core property of digital signatures follows: A signed message can unambiguously be traced back to its originator since a valid signature can only be computed with the unique signer's private key. Only the signer has the ability to generate a signature on his behalf. Hence, we can *prove* that the signing party has actually generated the message. Such a proof can even have legal meaning, for instance, as in the Electronic Signatures in Global and National Commerce Act (ESIGN) in the USA or in the *Signaturgesetz*, or Signature Law, in Germany. We note that the basic protocol above does not provide any confidentiality of the message since the message x is being sent in the clear. Of course, the message can be kept confidential by also encrypting it, e.g., with AES or 3DES.

Each of the three popular public-key algorithm families, namely integer factorization, discrete logarithms and elliptic curves, allows us to construct digital signatures. In the remainder of this chapter we learn about most signature schemes that are of practical relevance.

10.1.3 Security Services

It is very instructive to discuss in more detail the security functions we can achieve with digital signatures. In fact, at this point we will step for a moment away from digital signature and ask ourselves in general: What are possible *security objectives* that a security system might possess? More accurately the objectives of a security systems are called *security services*. There exist many security services, but the most important ones which are desirable in many applications are as follows:

1. **Confidentiality:** Information is kept secret from all but authorized parties.
2. **Integrity:** Messages have not been modified in transit.
3. **Message Authentication:** The sender of a message is authentic. An alternative term is *data origin authentication*.
4. **Nonrepudiation:** The sender of a message can not deny the creation of the message.

Different applications call for different sets of security services. For instance, for private e-mail the first three functions are desirable, whereas a corporate e-mail sys-

tem might also require nonrepudiation. As another example, if we want to secure software updates for a cell phone, the chief objectives might be integrity and message authentication because the manufacturer primarily wants to assure that only original updates are loaded into the handheld device. We note that message authentication always implies data integrity; the opposite is not true.

The four security services can be achieved in a more or less straightforward manner with the schemes introduced in this book: For confidentiality one uses primarily symmetric ciphers and less frequently asymmetric encryption. Integrity and message authentication are provided by digital signatures and message authentication codes which, are introduced in Chap. 12. Nonrepudiation can be achieved with digital signatures as discussed above.

In addition to the four core security services there are several other ones:

5. **Identification/entity authentication:** Establish and verify the identity of an entity, e.g., a person, a computer or a credit card.
6. **Access control:** Restrict access to the resources to privileged entities.
7. **Availability:** Assures that the electronic system is reliably available.
8. **Auditing:** Provide evidence about security-relevant activities, e.g., by keeping logs about certain events.
9. **Physical security:** Provide protection against physical tampering and/or responses to physical tampering attempts.
10. **Anonymity:** Provide protection against discovery and misuse of identity.

Which security services are desired in a given system is heavily application-specific. For instance, anonymity might make no sense for an e-mail system since e-mails are supposed to have a clearly identifiable sender. On the other hand, car-to-car communication systems for collision avoidance (one of the many exciting new applications for cryptography we will see in the next ten years or so) have a strong need to keep cars and drivers anonymous in order to avoid tracking. As a further example, in order to secure an operating system, access control to certain parts of a computer system is often of paramount importance. Most but not all of these advanced services can be achieved with the crypto algorithms from this book. However, in some cases noncryptographic approaches need to be taken. For instance, availability is often achieved by using redundancy, e.g., running redundant computing or storage systems in parallel. Such solutions are only indirectly, if at all, related to cryptography.

10.2 The RSA Signature Scheme

The RSA signature scheme is based on RSA encryption introduced in Chap. 7. Its security relies on the difficulty of factoring a product of two large primes (the integer factorization problem). Since its first description in 1978 in [143], the RSA signature scheme has emerged as the most widely used digital signatures scheme in practice.

10.2.1 Schoolbook RSA Digital Signature

Suppose Bob wants to send a signed message x to Alice. He generates the same RSA keys that were used for RSA encryption as shown in Chap. 7. At the end of the set-up he has the following parameters:

RSA Keys
- Bob's private key: $k_{pr} = (d)$
- Bob's public key: $k_{pub} = (n, e)$

The actual signature protocol is shown in the following. The message x that is being signed is in the range $(1, 2, \ldots, n-1)$.

Basic RSA Digital Signature Protocol

Alice		Bob
		$k_{pr} = d$, $k_{pub} = (n, e)$
	$\xleftarrow{(n,e)}$	
		compute signature: $s = \text{sig}_{k_{pr}}(x) \equiv x^d \bmod n$
	$\xleftarrow{(x,s)}$	

verify: $\text{ver}_{k_{pub}}(x, s)$

$x' \equiv s^e \bmod n$

$x' \begin{cases} \equiv x \bmod n & \Longrightarrow \text{valid signature} \\ \not\equiv x \bmod n & \Longrightarrow \text{invalid signature} \end{cases}$

As can be seen from the protocol, Bob computes the signature s for a message x by RSA-encrypting x with his private key k_{pr}. Bob is the only party who can apply k_{pr}, and hence the ownership of k_{pr} authenticates him as the author of the signed message. Bob appends the signature s to the message x and sends both to Alice. Alice receives the signed message and RSA-decrypts s using Bob's public key k_{pub}, yielding x. If x and x' match, Alice knows two important things: First, the author of the message was in possession of Bob's secret key, and if only Bob has had access to the key, it was in fact Bob who signed the message. This is called message authentication. Second, the message has not been changed in transit, so that message integrity is given. We recall from the previous section that these are two of the fundamental security services which are often needed in practice.

Proof. We now prove that the scheme is correct, i.e., that the verification process yields a "true" statement if the message and signature have not been altered during transmission. We start from the verification operation $s^e \bmod n$:

$$s^e = (x^d)^e = x^{de} \equiv x \bmod n$$

Due to the mathematical relationship between the private and the public key, namely that

$$d\,e \equiv 1 \bmod \phi(n),$$

raising any integer $x \in \mathbb{Z}_n$ to the $(d\,e)$th power yields the integer itself again. The proof for this was given in Sect. 7.3. □

The role of the public and the private keys are swapped compared to the RSA encryption scheme. Whereas RSA encryption applies the public key to the message x, the signature scheme applies the private key k_{pr}. On the other side of the communication channel, RSA encryption requires the use of the private key by the receiver, while the digital signature scheme applies the public key for verification.

Let's look at an example with small numbers.

Example 10.1. Suppose Bob wants to send a signed message ($x = 4$) to Alice. The first steps are exactly the same as it is done for an RSA encryption: Bob computes his RSA parameters and sends the public key to Alice. In contrast to the encryption scheme, now the private key is used for signing while the public key is needed to verify the signature.

Alice		**Bob**
		1. choose $p = 3$ and $q = 11$
		2. $n = p \cdot q = 33$
		3. $\Phi(n) = (3-1)(11-1) = 20$
		4. choose $e = 3$
		5. $d \equiv e^{-1} \equiv 7 \bmod 20$
	$\xleftarrow{(n,e)=(33,3)}$	
		compute signature for message $x = 4$:
		$s = x^d \equiv 4^7 \equiv 16 \bmod 33$
	$\xleftarrow{(x,s)=(4,16)}$	
verify:		
$x' = s^e \equiv 16^3 \equiv 4 \bmod 33$		
$x' \equiv x \bmod 33 \Longrightarrow$ valid signature		

Alice can conclude from the valid signature that Bob generated the message and that it was not altered in transit, i.e., message authentication and message integrity are given.

◇

It should be noted that we introduced a digital signature scheme only. In particular, the message itself is not encrypted and, thus, there is not confidentiality. If this security service is required, the message together with the signature should be encrypted, e.g., using a symmetric algorithm like AES.

10.2.2 Computational Aspects

First, we note that the signature is as long as the modulus n, i.e., roughly $\lceil \log_2 n \rceil$ bit. As discussed earlier, n is typically in the range from 1024 to 3072 bit. Even though such a signature length is not a problem in most Internet applications, it can be undesirable in systems that are bandwidth and/or energy constrained, e.g., mobile phones.

The key generation process is identical to the one we used for RSA encryption, which was discussed in detail in Chap. 7. To compute and verify the signature, the square-and-multiply algorithm introduced in Sect. 7.4 is used. The acceleration techniques for RSA introduced in Sect. 7.5 are also applicable to the digital signature scheme. Particularly interesting are short public keys e, for instance, the choice $e = 2^{16} + 1$. This makes verification a very fast operation. Since in many practical scenarios a message is signed only once but verified many times, the fact that verification is very fast is helpful. This is, e.g., the case in public-key infrastructures which use certificates. Certificates are signed only once but are verified over and over again every time a user uses his asymmetric keys (cf. Sect. 13.3.3).

10.2.3 Security

Like in every other asymmetric scheme, it must be assured that the public keys are authentic. This means that the verifying party in fact has the public key that is associated with the private signature key. If an attacker succeeds in providing the verifier with an incorrect public key that supposedly belongs to the signer, the attacker can obviously sign messages. In order to prevent the attack, certificates can be used, a topic which is discussed in Chap. 13.

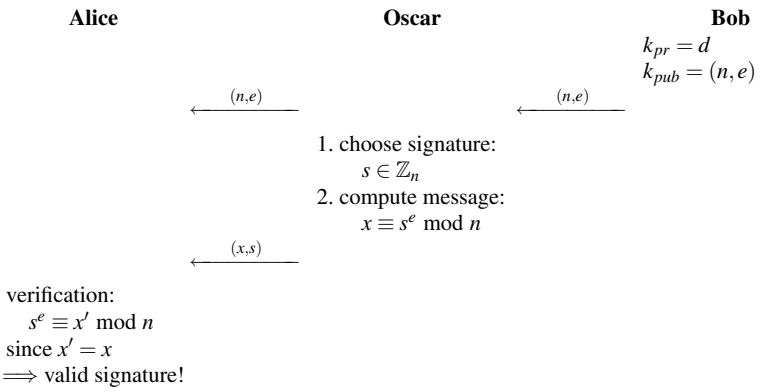
Algorithmic Attacks

The first group of attacks attempts to break the underlying RSA scheme by computing the private key d. The most general of these attacks tries to factor the modulus n into the primes p and q. If an attacker succeeds with this, she can compute the private key d from e. In order to prevent factoring attacks the modulus must be sufficiently large, as discussed in Sect. 7.8. In practice, 1024 bit or more are recommended.

Existential Forgery

Another attack against the schoolbook RSA signature scheme allows an attacker to generate a valid signature for a *random* message x. The attack works as follows:

Existential Forgery Attack Against RSA Digital Signature



The attacker impersonates Bob, i.e., Oscar claims to Alice that he is in fact Bob. Because Alice performs exactly the same computations as Oscar, she will verify the signature as correct. However, by closely looking at Steps 1 and 2 that Oscar performs, one sees that the attack is somewhat odd. The attacker chooses the signature first and then *computes* the message. As a consequence, he cannot control the semantics of the message x. For instance, Oscar cannot generate a message such as "`Transfer $1000 into Oscar's account`". Nevertheless, the fact that an automated verification process does not recognize the forgery is certainly not a desirable feature. For this reason, schoolbook RSA signature is rarely used in practice, and padding schemes are applied in order to prevent this and other attacks.

RSA Padding: The Probabilistic Signature Standard (PSS)

The attack above can be prevented by allowing only certain message formats. Roughly speaking, formatting imposes a rule which allows the verifier, Alice in our examples, to distinguish between valid and invalid messages; this is called *padding*. For example, a simple formatting rule could specify that all messages x have 100 trailing bits with the value zero (or any other specific bit pattern). If Oscar chooses signature values s and computes the "message" $x \equiv s^e \bmod n$, it is extremely unlikely that x has this specific format. If we require a certain value for the 100 trailing bits, the chance that x has this format is 2^{-100}, which is considerably lower than winning any lottery.

We now look at a padding scheme which is widely used in practice. Note that a padding scheme for RSA encryption was already discussed in Sect. 7.7. The *probabilistic signature scheme* (RSA-PSS) is a signature scheme based on the RSA cryptosystem. It combines signature and verification with an encoding of the message.

Let's have a closer look at RSA-PSS. Almost always in practice, the message itself is not signed directly but rather the hashed version of it. Hash functions compute a digital fingerprint of messages. The fingerprint has a fixed length, say 160 or 256 bit, but accepts messages as inputs of arbitrary lengths. More about hash functions and the role the play in digital signatures is found in Chap. 11.

In order to be consistent with the terminology used in standards, we denote the message with M rather than with x. Figure 10.2 depicts the encoding procedure which is known as Encoding Method for Signature with Appendix (EMSA) Probabilistic Signature Scheme (PSS).

Encoding for the EMSA Probabilistic Signature Scheme

Let $|n|$ be the size of the RSA modulus in bits. The encoded message EM has a length $\lceil (|n|-1)/8 \rceil$ bytes such that the bit length of EM is at most $|n|-1$ bit.

1. Generate a random value *salt*.
2. Form a string M' by concatenating a fixed padding $padding_1$, the hash value $mHash = h(M)$ and *salt*.
3. Compute a hash value H of the string M'.
4. Concatenate a fixed padding $padding_2$ and the value *salt* to form a data block DB.
5. Apply a mask generation function MGF to the string M' to compute the mask value $dbMask$. In practice, a hash function such as SHA-1 is often used as MGF.
6. XOR the mask value $dbMask$ and the data block DB to compute $maskedDB$.
7. The encoded message EM is obtained by concatenating $maskedDB$, the hash value H and the fixed padding bc.

After the encoding, the actual signing operation is applied to the encoded message EM, e.g.,

$$s = \text{sig}_{k_{pr}}(x) \equiv EM^d \bmod n$$

The verification operation then proceeds in a similar way: recovery of the *salt* value and checking whether the EMSA-PSS encoding of the message is correct. Note that the receiver knows the values of $padding_1$ and $padding_2$ from the standard.

The value H in EM is in essence the hashed version of the message. By adding a random value *salt* prior to the second hashing, the encoded value becomes probabilistic. As a consequence, if we encode and sign the same message twice, we obtain different signatures, which is a desirable feature.

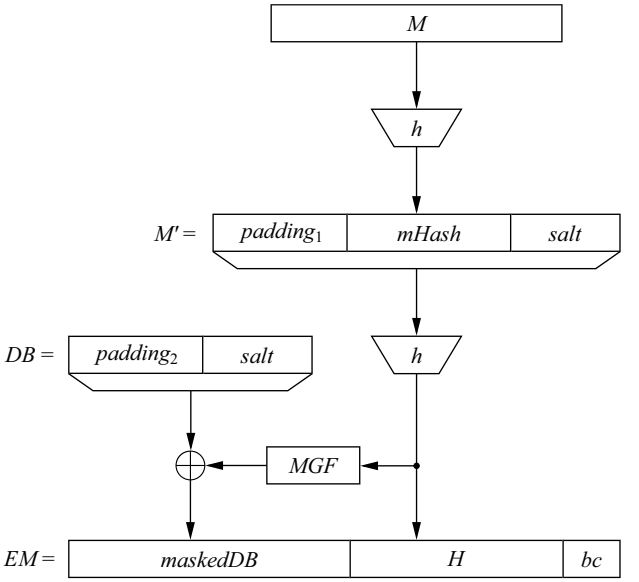


Fig. 10.2 Principle of EMSA-PSS encoding

10.3 The Elgamal Digital Signature Scheme

The Elgamal signature scheme, which was published in 1985, is based on the difficulty of computing discrete logarithms (cf. Chap. 8). Unlike RSA, where encryption and digital signature are almost identical operations, the Elgamal digital signature is quite different from the encryption scheme with the same name.

10.3.1 Schoolbook Elgamal Digital Signature

Key Generation

As with every public-key scheme, there is a set-up phase during which the keys are computed. We start by finding a large prime p and constructing a discrete logarithm problem as follows:

Key Generation for Elgamal Digital Signature

1. Choose a large prime p.
2. Choose a primitive element α of $\mathbb{Z}_p^*$ or a subgroup of $\mathbb{Z}_p^*$.
3. Choose a random integer $d \in \{2, 3, \ldots, p-2\}$.
4. Compute $\beta = \alpha^d \bmod p$.

The public key is now formed by $k_{pub} = (p, \alpha, \beta)$, and the private key by $k_{pr} = d$.

Signature and Verification

Using the private key and the parameters of the public key, the signature

$$\text{sig}_{k_{pr}}(x, k_E) = (r, s)$$

for a message x is computed during the signing process. Note that the signature consists of two integers r and s. The signing consists of two main steps: choosing a random value k_E, which forms an ephemeral private key, and computing the actual signature of x.

Elgamal Signature Generation

1. Choose a random ephemeral key $k_E \in \{0, 1, 2, \ldots, p-2\}$ such that $\gcd(k_E, p-1) = 1$.
2. Compute the signature parameters:

$$r \equiv \alpha^{k_E} \bmod p,$$
$$s \equiv (x - d \cdot r) k_E^{-1} \bmod p - 1.$$

On the receiving side, the signature is verified as $\text{ver}_{k_{pub}}(x, (r, s))$ using the public key, the signature and the message.

Elgamal Signature Verification

1. Compute the value
$$t \equiv \beta^r \cdot r^s \bmod p$$
2. The verification follows from:
$$t \begin{cases} \equiv \alpha^x \bmod p & \Longrightarrow \text{valid signature} \\ \not\equiv \alpha^x \bmod p & \Longrightarrow \text{invalid signature} \end{cases}$$

In short, the verifier accepts a signature (r,s) only if the relation $\beta^r \cdot r^s \equiv \alpha^x \bmod p$ is satisfied. Otherwise, the verification fails. In order to make sense of the rather arbitrary looking rules for computing the signature parameters r and s as well as the verification, it is helpful to study the following proof.

Proof. We'll prove the correctness of the Elgamal signature scheme. More specifically, we show that the verification process yields a "true" statement if the verifier uses the correct public key and the correct message, and if the signature parameters (r,s) were chosen as specified. We start with the verification equation:

$$\begin{aligned}\beta^r \cdot r^s &\equiv (\alpha^d)^r (\alpha^{k_E})^s \bmod p \\ &\equiv \alpha^{d\,r+k_E\,s} \bmod p.\end{aligned}$$

We require that the signature is considered valid if this expression is identical to α^x:

$$\alpha^x \equiv \alpha^{d\,r+k_E\,s} \bmod p. \tag{10.1}$$

According to Fermat's Little Theorem, the relationship (10.1) holds if the exponents on both sides of the expression are identical modulo $p-1$:

$$x \equiv d\,r + k_E\,s \bmod p-1$$

from which the construction rule of the signature parameters s follows:

$$s \equiv (x - d \cdot r) k_E^{-1} \bmod p-1.$$

□

The condition that $\gcd(k_E, p-1) = 1$ is required since we have to invert the ephemeral key modulo $p-1$ when computing s.

Let's look at an example with small numbers.

Example 10.2. Again, Bob wants to send a message to Alice. This time, it should be signed with the Elgamal digital signature scheme. The signature and verification process is as follows:

Alice		Bob
		1. choose $p = 29$ 2. choose $\alpha = 2$ 3. choose $d = 12$ 4. $\beta = \alpha^d \equiv 7 \bmod 29$
	$\xleftarrow{(p,\alpha,\beta)=(29,2,7)}$	
		compute signature for message $x = 26$: choose $k_E = 5$, note that $\gcd(5, 28) = 1$ $r = \alpha^{k_E} \equiv 2^5 \equiv 3 \bmod 29$ $s = (x - d\,r)\,k_E^{-1} \equiv (-10) \cdot 17 \equiv 26 \bmod 28$
	$\xleftarrow{(x,(r,s))=(26,(3,26))}$	
verify: $t = \beta^r \cdot r^s \equiv 7^3 \cdot 3^{26} \equiv 22 \bmod 29$ $\alpha^x \equiv 2^{26} \equiv 22 \bmod 29$ $t \equiv \alpha^x \bmod 29 \Longrightarrow$ valid signature		

◇

10.3.2 Computational Aspects

The key generation phase is identical to the set-up phase of Elgamal encryption, which we introduced in Sect. 8.5.2. Because the security of the signature scheme relies on the discrete logarithm problem, p needs to have the properties discussed in Sect. 8.3.3. In particular, it should have a length of at least 1024 bits. The prime can be generated using the prime-finding algorithms introduced in Sect 7.6. The private key should be generated by a true random number generator. The public key requires one exponentiation using the square-and-multiply algorithm.

The signature consists of the pair (r, s). Both have roughly the same bit length as p, so that the total length of the package $(x, (r, s))$ is about three times as long as only the message x. Computing r requires an exponentiation modulo p, which can be achieved with the square-and-multiply algorithm. The main operation when computing s is the inversion of k_E. This can be done using the extended Euclidean algorithm. A speed-up is possible through precomputing. The signer can generate the ephemeral key k_E and r in advance and store both values. When a message is to be signed, they can be retrieved and used to compute s. The verifier performs two exponentiations that are again computed with the square-and-multiply algorithm, and one multiplication.

10.3.3 Security

First, we must make sure that the verifier has the correct public key. Otherwise, the attack sketched in Sect. 10.2.3 is applicable. Other attacks are described in the following.

Computing Discrete Logarithms

The security of the signature scheme relies on the discrete logarithm problem (DLP). If Oscar is capable of computing discrete logarithms, he can compute the private key d from β as well as the ephemeral key k_E from r. With this knowledge, he can sign arbitrary messages on behalf of the signer. Hence the Elgamal parameters must be chosen such that the DLP is intractable. We refer to Sect. 8.3.3 for a discussion of possible discrete logarithm attacks. One of the key requirements is that the prime p should be at least 1024-bit long. We have also make sure that small subgroup attacks are not possible. In order to counter this attack, in practice primitive elements α are used to generate a subgroup of prime order. In such groups, all elements are primitive and small subgroups do not exist.

Reuse of the Ephemeral Key

If the signer reuses the ephemeral key k_E, an attacker can easily compute the private key a. This constitutes a complete break of the system. Here is how the attack works.

Oscar observes two digital signatures and messages of the form $(x,(r,s))$. If the two messages x_1 and x_2 have the same ephemeral key k_E, Oscar can detect this since the two r values are the same because they were constructed as $r_1 = r_2 = \alpha^{k_E}$. The two s values are different, and Oscar obtains the following two expressions:

$$s_1 \equiv (x_1 - d\,r)k_E^{-1} \bmod p-1 \tag{10.2}$$

$$s_2 \equiv (x_2 - d\,r)k_E^{-1} \bmod p-1 \tag{10.3}$$

This is an equation system with the two unknowns d, which is Bob's private key (!) and the ephemeral key k_E. By multiplying both equations by k_E it becomes a linear system of equations which can be solved easily. Oscar simply subtracts the second equation from the first one, yielding:

$$s_1 - s_2 \equiv (x_1 - x_2)k_E^{-1} \bmod p-1$$

from which the ephemeral key follows as

$$k_E \equiv \frac{x_1 - x_2}{s_1 - s_2} \bmod p-1.$$

If $\gcd(s_1 - s_2, p-1) \neq 1$, the equation has multiple solutions for k_E, and Oscar has to verify which is the correct one. In any case, using k_E, Oscar can now also compute the private key through either Eq. (10.2) or Eq. (10.3):

$$d \equiv \frac{x_1 - s_1 k_E}{r} \bmod p-1.$$

With the knowledge of the private key d and the public key parameters, Oscar can now freely sign any documents on Bob's behalf. In order to avoid the attack, fresh ephemeral keys stemming from a random number generator should be used for every digital signature.

An attack with small numbers is given in the next example.

Example 10.3. Let's assume the situation where Oscar eavesdrops on the following two messages that were previously signed with Bob's private key and that use the same ephemeral key k_E:

1. $(x_1, (r, s_1)) = (26, (3, 26))$,
2. $(x_2, (r, s_2)) = (13, (3, 1))$.

Additionally, Oscar knows Bob's public key, which is given as

$$(p, \alpha, \beta) = (29, 2, 7).$$

With this information, Oscar is now able to compute the ephemeral key

$$\begin{aligned} k_E &\equiv \frac{x_1 - x_2}{s_1 - s_2} \bmod p-1 \\ &\equiv \frac{26-13}{26-1} \equiv 13 \cdot 9 \\ &\equiv 5 \bmod 28 \end{aligned}$$

and finally reveal Bob's private key d:

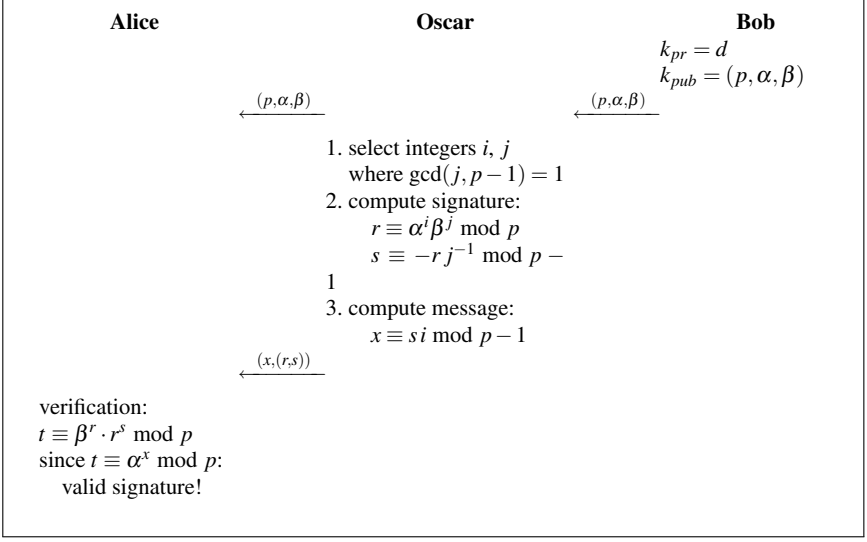
$$\begin{aligned} d &\equiv \frac{x_1 - s_1 \cdot k_E}{r} \bmod p-1 \\ &\equiv \frac{26 - 26 \cdot 5}{3} \equiv 8 \cdot 19 \\ &\equiv 12 \bmod 28. \end{aligned}$$

◇

Existential Forgery Attack

Similar to the case of RSA digital signatures, it is also possible that an attacker generates a valid signature for a *random* message x. The attacker, Oscar, impersonates Bob, i.e., Oscar claims to Alice that he is in fact Bob. The attack works as follows:

Existential Forgery Attack Against Elgamal Digital Signature



The verification yields a "true" statement because the following holds:

$$\begin{aligned} t &\equiv \beta^r \cdot r^s \bmod p \\ &\equiv \alpha^{dr} \cdot r^s \bmod p \\ &\equiv \alpha^{dr} \cdot \alpha^{(i+dj)s} \bmod p \\ &\equiv \alpha^{dr} \cdot \alpha^{(i+dj)(-rj^{-1})} \bmod p \\ &\equiv \alpha^{dr-dr} \cdot \alpha^{-rij^{-1}} \bmod p \\ &\equiv \alpha^{si} \bmod p \end{aligned}$$

Since the message was constructed as $x \equiv s\,i \bmod p-1$, the last expression is equal to

$$\alpha^{si} \equiv \alpha^x \bmod p$$

which is exactly Alice's condition for accepting the signature as valid.

The attacker computes in Step 3 the message x, the semantics of which he cannot control. Thus, Oscar can only compute valid signatures for pseudorandom messages.

The attack is not possible if the message is hashed, which is, in practice, very often the case. Rather than using the message directly for computing the signature, one applies a hash function to the message prior to signing, i.e., the signing equation becomes:

$$s \equiv (h(x) - d \cdot r) k_E^{-1} \bmod p-1.$$

10.4 The Digital Signature Algorithm (DSA)

The native Elgamal signature algorithm described in this section is rarely used in practice. Instead, a much more popular variant is used, known as the *Digital Signature Algorithm (DSA)*. It is a federal US government standard for digital signatures (DSS) and was proposed by the National Institute of Standards and Technology (NIST). Its main advantages over the Elgamal signature scheme are that the signature is only 320-bit long and that some of the attacks that can threaten the Elgamal scheme are not applicable.

10.4.1 The DSA Algorithm

We introduce here the DSA standard with a bit length of 1024 bit. Note that longer bit lengths are also possible in the standard.

Key Generation

The keys for DSA are computed as follows:

Key Generation for DSA

1. Generate a prime p with $2^{1023} < p < 2^{1024}$.
2. Find a prime divisor q of $p-1$ with $2^{159} < q < 2^{160}$.
3. Find an element α with $\text{ord}(\alpha) = q$, i.e., α generates the subgroup with q elements.
4. Choose a random integer d with $0 < d < q$.
5. Compute $\beta \equiv \alpha^d \bmod p$.

The keys are now:
$k_{pub} = (p, q, \alpha, \beta)$
$k_{pr} = (d)$

The central idea of DSA is that there are two cyclic groups involved. One is the large cyclic group $\mathbb{Z}_p^*$, the order of which has bit length of 1024 bit. The second one is in the 160-bit subgroup of $\mathbb{Z}_p^*$. This set-up yields shorter signatures, as we see in the following.

In addition to the 1024-bit prime p and a 160-bit prime q, there are two other bit length combinations possible for the primes p and q. According to the latest version of the standard, the combinations shown in Table 10.1 are allowed.

If one of the other bit lengths is required, only Steps 1 and 2 of the key generation phase have to be adjusted accordingly. More about the issue of bit length will be said in Sect. 10.4.3 below.

Table 10.1 Bit lengths of important parameters of DSA

p	q	Signature
1024	160	320
2048	224	448
3072	256	512

Signature and Verification

As in the Elgamal signature scheme, the DSA signature consists of a pair of integers (r,s). Since each of the two parameters is only 160-bit long, the total signature length is 320 bit. Using the public and private key, the signature for a message x is computed as follows:

DSA Signature Generation

1. Choose an integer as random ephemeral key k_E with $0 < k_E < q$.
2. Compute $r \equiv (\alpha^{k_E} \bmod p) \bmod q$.
3. Compute $s \equiv (SHA(x) + d \cdot r)\, {k_E}^{-1} \bmod q$.

According to the standard, the message x has to be hashed using the hash function SHA-1 in order to compute s. Hash functions, including SHA-1, are described in Chap. 11. For now it is sufficient to know that SHA-1 compresses x and computes a 160-bit fingerprint. This fingerprint can be thought of as a representative of x.

The signature verification process is as follows:

DSA Signature Verification

1. Compute auxiliary value $w \equiv s^{-1} \bmod q$.
2. Compute auxiliary value $u_1 \equiv w \cdot SHA(x) \bmod q$.
3. Compute auxiliary value $u_2 \equiv w \cdot r \bmod q$.
4. Compute $v \equiv (\alpha^{u_1} \cdot \beta^{u_2} \bmod p) \bmod q$.
5. The verification $ver_{k_{pub}}(x,(r,s))$ follows from:

$$v \begin{cases} \equiv r \bmod q \implies \text{valid signature} \\ \not\equiv r \bmod q \implies \text{invalid signature} \end{cases}$$

The verifier accepts a signature (r,s) only if $v \equiv r \bmod q$ is satisfied. Otherwise, the verification fails. In this case, the message or the signature may have been modified or the verifier is not in possession of the correct public key. In any case, the signature should be considered invalid.

Proof. We show that a signature (r,s) satisfies the verification condition $v \equiv r \bmod q$. We'll start with the signature parameter s:

$$s \equiv (SHA(x) + d\,r)\,{k_E}^{-1} \bmod q$$

which is equivalent to:

$$k_E \equiv s^{-1}\,SHA(x) + d\,s^{-1}\,r \bmod q.$$

The right-hand side can be expressed in terms of the auxiliary values u_1 and u_2:

$$k_E \equiv u_1 + d\,u_2 \bmod q.$$

We can raise α to either side of the equation if we reduce modulo p:

$$\alpha^{k_E} \bmod p \equiv \alpha^{u_1 + d\,u_2} \bmod p.$$

Since the public key value β was computed as $\beta \equiv \alpha^d \bmod p$, we can write:

$$\alpha^{k_E} \bmod p \equiv \alpha^{u_1}\beta^{u_2} \bmod p.$$

We now reduce both sides of the equation modulo q:

$$(\alpha^{k_E} \bmod p) \bmod q \equiv (\alpha^{u_1}\beta^{u_2} \bmod p) \bmod q.$$

Since r was constructed as $r \equiv (\alpha^{k_E} \bmod p) \bmod q$ and $v \equiv (\alpha^{u_1}\beta^{u_2} \bmod p) \bmod q$, this expression is identical to the condition for verifying a signature as valid:

$$r \equiv v \bmod q.$$

□

Let's look at an example with small numbers.

Example 10.4. Bob wants to send a message x to Alice which is to be signed with the DSA algorithm. Suppose the hash value of x is $h(x) = 26$. Then the signature and verification process is as follows:

Alice | | **Bob**

		1. choose $p = 59$ 2. choose $q = 29$ 3. choose $\alpha = 3$ 4. choose private key $d = 7$ 5. $\beta = \alpha^d \equiv 4 \bmod 59$
	$\xleftarrow{(p,q,\alpha,\beta)=(59,29,3,4)}$	
		sign: compute hash of message $h(x) = 26$ 1. choose ephemeral key $k_E = 10$ 2. $r = (3^{10} \bmod 59) \equiv 20 \bmod 29$ 3. $s = (26 + 7 \cdot 20) \cdot 3 \equiv 5 \bmod 29$
	$\xleftarrow{(x,(r,s))=(x,(20,5))}$	
verify: 1. $w = 5^{-1} \equiv 6 \bmod 29$ 2. $u_1 = 6 \cdot 26 \equiv 11 \bmod 29$ 3. $u_2 = 6 \cdot 20 \equiv 4 \bmod 29$ 4. $v = (3^{11} \cdot 4^4 \bmod 59) \bmod 29 = 20$ 5. $v \equiv r \bmod 29 \Longrightarrow$ valid signature		

In this example, the subgroup has a prime order of $q = 29$, whereas the "large" cyclic group modulo p has 58 elements. We note that $58 = 2 \cdot 29$. We replaced the function $SHA(x)$ by $h(x)$ since the SHA hash function has an output of length 160 bit.

◇

10.4.2 Computational Aspects

We discuss now the computations involved in the DSA scheme. The most demanding part is the key-generation phase. However, this phase only has to be executed once at set-up time.

Key Generation

The challenge in the key-generation phase is to find a cyclic group $\mathbb{Z}_p^*$ with a bit length of 1024, and which has a prime subgroup in the range of 2^{160}. This condition is fulfilled if $p-1$ has a prime factor q of 160 bit. The general approach to generating such parameters is to first find the 160-bit prime q and then to construct the larger prime p from it. Below is the prime-generating algorithm. Note that the NIST-specified scheme is slightly different.

Prime Generation for DSA
Output: two primes (p,q), where $2^{1023} < p < 2^{1024}$ and $2^{159} < q < 2^{160}$, such that $p-1$ is a multiple of q.
Initialization: $i = 1$
Algorithm:

```
1    find prime q with 2^159 < q < 2^160 using the Miller–Rabin algorithm
2    FOR i = 1 TO 4096
2.1      generate random integer M with 2^1023 < M < 2^1024
2.2      M_r ≡ M  mod 2q
2.3      p − 1 ≡ M − M_r          (note that p − 1 is a multiple of 2q.)
         IF p is prime            (use Miller–Rabin primality test)
 2.4         RETURN (p, q)
2.5      i = i + 1
3    GOTO Step 1
```

The choice of $2q$ as modulus in step 2.3 assures that the prime candidates generated in step 2.3 are odd numbers. Since $p-1$ is divisible by $2q$, it is also divisible by q. If p is a prime, $\mathbb{Z}_p^*$ thus has a subgroup of order q.

Signing

During signing we compute the parameters r and s. Computing r involves first evaluation $g^{k_E} \bmod p$ using the square-and-multiply algorithm. Since k_E has only 160 bit, about $1.5 \times 160 = 240$ squarings and multiplications are required on average, even though the arithmetic is done with 1024-bit numbers. The result, which has also a length of 1024 bit, is then reduced to 160 bit by the operation "$\bmod q$". Computing s involves only 160-bit numbers. The most costly step is the inversion of k_E.

Of these operations, the exponentiation is the most costly one in terms of computational complexity. Since the parameter r does not depend on the message, it can be precomputed so that the actual signing can be a relatively fast operation.

Verification

Computing the auxiliary parameters w, u_1 and u_2 only involves 160-bit operands, which makes them relatively fast.

10.4.3 Security

An interesting aspect of DSA is that we have to protect against two different discrete logarithm attacks. If an attacker wants to break DSA, he could attempt to compute the private key d by solving the discrete logarithm in the large cyclic group modulo p:

$$d = \log_\alpha \beta \bmod p.$$

The most powerful method for this is the index calculus attack, which was sketched in Sect. 8.3.3. In order to thwart this attack, p must be at least 1024-bit long. It is estimated that this provides a security level of 80 bit, i.e., an attack would need on the order of 2^{80} operations (cf. Table 6.1 in Chap. 6). For higher security levels, NIST allows primes with lengths of 2048 and 3072 bit.

The second discrete logarithm attack on DSA is to exploit the fact that α generates only a small subgroup of order q. Hence, it seems promising to attack only the subgroup, which has a size of about 2^{160}, rather than the large cyclic group with about 2^{1024} elements formed by p. However, it turns out that the powerful index-calculus attack is not applicable if Oscar wants to exploit the subgroup property. The best he can do is to perform one of the generic DLP attacks, i.e., either the baby-step giant-step method or Pollard's rho method (cf. Sect. 8.3.3). These are so-called square root attacks, and given that the subgroup has an order of approximately 2^{160}, these attacks provide a security level of $\sqrt{2^{160}} = 2^{80}$. It is not a coincidence that the index calculus attack and the square root attack have a comparable complexity, in fact the parameter sizes were deliberately chosen that way. One has to be careful,

though, if the size of p is increased to 2048 or 3072 bit. This only increases the difficulty of the index-calculus attack, but the small subgroup attack would still have a complexity of 2^{80} if the subgroup stays the same size. For this reason q also must be increased if larger p values are chosen. Table 10.2 shows the NIST-specified lengths of the primes p and q together with the resulting security levels. The security level of the hash function must also match the one of the discrete logarithm problem. Since the cryptographic strength of a hash function is mainly determined by the bit length of the hash output, the minimum hash output is also given in the table. More about security of hash functions will be said in Chap. 11.

Table 10.2 Standardized parameter bit lengths and security levels for DSA

p	q	Hash output (min)	Security levels
1024	160	160	80
2048	224	224	112
3072	256	256	128

It should be stressed that the record for discrete logarithm calculations is 532 bit, so that the 1024-bit DSA variant is currently secure, and the 2048-bit and 3072-bit variants seem to provide good long-term security.

In addition to discrete logarithm attacks, DSA becomes vulnerable if the ephemeral key is reused. This attack is completely analogues to the case of Elgamal digital signature. Hence, it must be assured that a fresh randomly-genererated key k_E is used in every signing operation.

10.5 The Elliptic Curve Digital Signature Algorithm (ECDSA)

As discussed in Chap. 9, elliptic curves have several advantages over RSA and over DL schemes like Elgamal or DSA. In particular, in absence of strong attacks against elliptic curve cryptosystems (ECC), bit lengths in the range of 160–256 bit can be chosen which provide security equivalent to 1024–3072-bit RSA and DL schemes. The shorter bit length of ECC often results in shorter processing time and in shorter signatures. For these reasons, the Elliptic Curve Digital Signature Algorithm (ECDSA) was standardized in the US by the American National Standards Institute (ANSI) in 1998.

10.5.1 The ECDSA Algorithm

The steps in the ECDSA standard are conceptionally closely related to the DSA scheme. However, its discrete logarithm problem is constructed in the group of

an elliptic curve. Thus, the arithmetic to be performed for actually computing an ECDSA signature is entirely different from that used for DSA.

The ECDSA standard is defined for elliptic curves over prime fields $\mathbb{Z}_p$ and Galois fields $GF(2^m)$. The former is often preferred in practice, and we will only introduce this one in what follows.

Key Generation

The keys for the ECDSA are computed as follows:

Key Generation for ECDSA

1. Use an elliptic curve E with
   - modulus p
   - coefficients a and b
   - a point A which generates a cyclic group of prime order q
2. Choose a random integer d with $0 < d < q$.
3. Compute $B = dA$.

The keys are now:
$k_{pub} = (p, a, b, q, A, B)$
$k_{pr} = (d)$

Note that we have set up a discrete logarithm problem where the integer d is the private key and the result of the scalar multiplication, point B, is the public key. Similar to DSA, the cyclic group has an order q which should have a size of at least 160 bit or more for higher security levels.

Signature and Verification

Like DSA, an ECDSA signature consists of a pair of integers (r, s). Each value has the same bit length as q, which makes for fairly compact signatures. Using the public and private key, the signature for a message x is computed as follows:

ECDSA Signature Generation

1. Choose an integer as random ephemeral key k_E with $0 < k_E < q$.
2. Compute $R = k_E A$.
3. Let $r = x_R$.
4. Compute $s \equiv (h(x) + d \cdot r)\, {k_E}^{-1} \bmod q$.

In step 3 the x-coordinate of the point R is assigned to the variable r. The message x has to be hashed using the function h in order to compute s. The hash function output length must be at least as long as q. More about the choice of the hash function will be said in Chap. 11. However, for now it is sufficient to know that the hash function compresses x and computes a fingerprint which can be viewed as a representative of x.

The signature verification process is as follows:

ECDSA Signature Verification

1. Compute auxiliary value $w \equiv s^{-1} \bmod q$.
2. Compute auxiliary value $u_1 \equiv w \cdot h(x) \bmod q$.
3. Compute auxiliary value $u_2 \equiv w \cdot r \bmod q$.
4. Compute $P = u_1 A + u_2 B$.
5. The verification $ver_{k_{pub}}(x,(r,s))$ follows from:

$$x_P \begin{cases} \equiv r \bmod q \Longrightarrow \text{valid signature} \\ \not\equiv r \bmod q \Longrightarrow \text{invalid signature} \end{cases}$$

In the last step, the notation x_P indicates the x-coordinate of the point P. The verifier accepts a signature (r,s) only if the x_P has the same value as the signature parameter r modulo q. Otherwise, the signature should be considered invalid.

Proof. We show that a signature (r,s) satisfies the verification condition $r \equiv x_P \bmod q$. We'll start with the signature parameter s:

$$s \equiv (h(x) + d\,r)\,{k_E}^{-1} \bmod q$$

which is equivalent to:

$$k_E \equiv s^{-1}\,h(x) + d\,s^{-1}\,r \bmod q.$$

The right-hand side can be expressed in terms of the auxiliary values u_1 and u_2:

$$k_E \equiv u_1 + d\,u_2 \bmod q.$$

Since the point A generates a cyclic group of order q, we can multiply both sides of the equation with A:

$$k_E A = (u_1 + d\,u_2)A.$$

Since the group operation is associative, we can write

$$k_E A = u_1 A + d\,u_2 A$$

and

$$k_E A = u_1 A + u_2 B.$$

What we showed so far is that the expression $u_1 A + u_2 B$ is equal to $k_E A$ if the correct signature and key (and message) have been used. But this is exactly the condition that we check in the verification process by comparing the x-coordinates of $P = u_1 A + u_2 B$ and $R = k_E A$.

□

Using the small elliptic curve from Chap. 9, we look at a simple ECDSA example.

Example 10.5. Bob wants to send a message to Alice that is to be signed with the ECDSA algorithm. The signature and verification process is as follows:

Alice		**Bob**
		choose E with $p = 17$, $a = 2$, $b = 2$,
		and $A = (5,1)$ with $q = 19$
		choose $d = 7$
		compute $B = dA = 7 \cdot (5,1) = (0,6)$
	$\xleftarrow{\substack{(p,a,b,q,A,B)= \\ (17,2,2,19,(5,1),(0,6))}}$	
		sign:
		compute hash of message $h(x) = 26$
		choose ephemeral key $k_E = 10$
		$R = 10 \cdot (5,1) = (7,11)$
		$r = x_R = 7$
		$s = (26 + 7 \cdot 7) \cdot 2 \equiv 17 \bmod 19$
	$\xleftarrow{(x,(r,s))=(x,(7,17))}$	
verify:		
$w = 17^{-1} \equiv 9 \bmod 19$		
$u_1 = 9 \cdot 26 \equiv 6 \bmod 19$		
$u_2 = 9 \cdot 7 \equiv 6 \bmod 19$		
$P = 6 \cdot (5,1) + 6 \cdot (0,6) = (7,11)$		
$x_P \equiv r \bmod 19 \Longrightarrow$ valid signature		

Note that we chose the elliptic curve

$$E : y^2 \equiv x^3 + 2x + 2 \bmod 17$$

which is discussed in Sect. 9.2. Because all points of the curve form a cyclic group of order 19, i.e., a prime, there are no subgroups and hence in this case $q = \#E = 19$.

◇

10.5.2 Computational Aspects

We discuss now the computations involved in the three stages of the ECDSA scheme.

Key Generation As discussed earlier, finding an elliptic curve with good cryptographic properties is a nontrivial task. In practice, standardized curves such as the ones proposed by NIST or the Brainpool consortium are often used. The remaining computation in the key generation phase is one point multiplication, which can be done using the double-and-add algorithm.

Signing During signing we first compute the point R, which requires one point multiplication, and from which r immediately follows. For the parameter s we have to invert the ephemeral key, which is done with the extended Euclidean algorithm. The other main operations are hashing of the message and one reduction modulo q.

The point multiplication, which is in most cases by the far the most arithmetic-intensive operation, can be precomputed by choosing the ephemeral key ahead of time, e.g., during the idle time of a CPU. Thus, in situations where precomputation is an option, signing becomes a very fast operation.

Verification Computing the auxiliary parameters w, u_1 and u_2 involves straightforward modular arithmetic. The main computational load occurs during the evaluation of $Pu_1A + u_2B$. This can be accomplished by two separate point multiplications. However, there are specialized methods for simultaneous exponentiations (remember from Chap. 9 that point multiplication is closely related to exponentiation) which are faster than two individual point multiplications.

10.5.3 Security

Given that the elliptic curve parameters are chosen correctly, the main analytical attack against ECDSA attempts to solve the elliptic curve discrete logarithm problem. If an attacker were capable of doing this, he could compute the private key d and/or the ephemeral key. However, the best known ECC attacks have a complexity proportional to the square root of the size of the group in which the DL problem is defined, i.e., proportional to $\sqrt{q}$. The parameter length of ECDSA and the corresponding security levels are given in Table 10.3. We recall that the prime p is typically only slightly larger than q, so that all arithmetic for ECDSA is done with operands which have roughly the bit length of q.

The security level of the hash function must also match that of the discrete logarithm problem. The cryptographic strength of a hash function is mainly determined by the length of its output. More about security of hash functions will be said in Chap. 11.

The security levels of 128, 192 and 256 were chosen so that they match the security offered by AES with its three respective key sizes.

More subtle attacks against ECDSA are also possible. For instance, at the beginning of verification it must be checked whether $r, s \in \{1, 2, \ldots, q\}$ in order to prevent a certain attack. Also, protocol-based weaknesses, e.g., reusing the ephemeral key, must be prevented.

Table 10.3 Bit lengths and security levels of ECDSA

q	Hash output (min)	Security levels
192	192	96
224	224	112
256	256	128
384	384	192
512	512	256

10.6 Discussion and Further Reading

Digital Signature Algorithms The first practical realization of digital signatures was introduced in the original paper by Rivest, Shamir and Adleman [143]. RSA digital signatures have been standardized by several bodies for a long time, see, e.g., [95]. RSA signatures were, and in many cases still are, the de facto standard for many applications, especially for certificates on the Internet.

The Elgamal digital signature was published in 1985 in [73]. Many variants of this scheme are possible and have been proposed over the years. For a compact summary, see [120, Note 11.70].

The DSA algorithm was proposed in 1991 and became a US standard in 1994. There were two possible motivations for the government to create this standard as an alternative to RSA. First, RSA was patented at that time and having a free alternative was attractive for US industry. Second, an RSA digital signature implementation can also be used for encryption. This was not desirable (from the US government viewpoint) since there were still rather strict export restrictions for cryptography in the US at that time. In contrast, a DSA implementation can only be used for signing and not for encryption, and it was easier to export systems that only included signature functionality. Note that DSA refers to the digital signature *algorithm*, and the corresponding standard is referred to as DSS, the digital signature *standard*. Today, DSS includes not only the DSA algorithm but also ECDSA and RSA digital signatures [126].

In addition to the algorithms discussed in this chapter, there exist several other schemes for digital signatures. These include, e.g., the Rabin signature [140], the Fiat–Shamir signature [76], the Pointcheval–Stern signature [134] and the Schnorr signature [150].

Using Digital Signatures With digital signatures, the problem of authentic public keys is acute: How can Alice (or Bob) assure that they possess the correct public keys for each other? Or, phrased differently, how can Oscar be prevented from injecting faked public keys in order to perform an attack? We discuss this question in detail in Chap. 13, where certificates are introduced. Certificates are based on digital signatures and are one of the main applications of digital signatures. They bind an identity (e.g., Alice's e-mail address) to a public key.

One of the more interesting interactions between society and cryptography is digital signature laws. They basically assure that a cryptographic digital signature has a legally binding meaning. For instance, an electronic contract that was digitally

signed can be enforced in the same way as a conventionally signed contract. Around the turn of the millennium, many nations introduced corresponding laws. This was at a time that the "brave new world" of the Internet had opened up seemingly endless opportunities for doing business online, and digital signature laws seemed to be crucial to allow trusted business transactions via the Internet. Examples of digital signature laws are the Electronic Signatures in Global and National Commerce Act (ESIGN) in the US [138], or the corresponding directive of the European Union [133]. A good online source for more information is the Digital Law Survey [167]. Even though much electronic commerce is today conducted without making use of signature laws, there will be without doubt more and more situations where those laws are actually needed.

One crucial issue when using digital signatures in the real world is that the private keys, especially if used in a setting with legal significance, have to be kept strictly confidential. This requires a secure way to store this delicate key material. One way to satisfy this requirement is to employ *smart cards* that can be used as secure containers for secret keys. A secret key never leaves the smart card, and signatures are performed within the CPU inside the smart card. For applications with high security requirements, so called *tamper-resistant* smart cards are protected against several types of hardware attacks. Reference [141] provides excellent insight into the various facets of the highly sophisticated smart card technology.

10.7 Lessons Learned

- Digital signatures provide message integrity, message authentication and nonrepudiation.
- One of the main application areas of digital signatures is certificates.
- RSA is currently the most widely used digital signature algorithm. Competitors are the Digital Signature Standard (DSA) and the Elliptic Curve Digital Signature Standard (ECDSA).
- The Elgamal signature scheme is the basis for DSA. In turn, ECDSA is a generalization of DSA to elliptic curves.
- RSA verification can be done with short public keys e. Hence, in practice, RSA verification is usually faster than signing.
- DSA and ECDSA have the advantage over RSA in that the signatures are much shorter.
- In order to prevent certain attacks, RSA should be used with padding.
- The modulus of DSA and the RSA signature schemes should be at least 1024-bits long. For true long-term security, a modulus of length 3072 bits should be chosen. In contrast, ECDSA achieves the same security levels with bit lengths in the range 160–256 bits.

Problems

10.1. In Sect. 10.1.3 we state that sender (or message) authentication always implies data integrity. Why? Is the opposite true too, i.e., does data integrity imply sender authentication? Justify both answers.

10.2. In this exercise, we want to consider some basic aspects of security services.

1. Does privacy always guarantee integrity? Justify your answer.
2. In which order should confidentiality and integrity be assured (should the entire message be encrypted first or last)? Give the rationale for your answer.

10.3. Design a security service that provides data integrity, data confidentiality and nonrepudiation using public-key cryptography in a two-party communication system over an insecure channel. Give a rationale that data integrity, confidentiality and nonrepudiation are achieved by your solution. (Recommendation: Consider the corresponding threats in your argumentation.)

10.4. A painter comes up with a new business idea: He wants to offer custom paintings from photos. Both the photos and paintings will be transmitted in digital form via the Internet. One concern that he has is discretion towards his customers, since potentially embarrassing photos, e.g., nude photos, might be sent to him. Hence, the photo data should not be accessible for third parties during transmission. The painter needs multiple weeks for the creation of a painting, and hence he wants to assure that he cannot be fooled by someone who sends in a photo assuming a false name. He also wants to be assured that the painting will definitely be accepted by the customer and that she cannot deny the order.

1. Choose the necessary security services for the transmission of the digitalized photos from the customers to the painter.
2. Which cryptographic elements (e.g., symmetric encryption) can be utilized to achieve the security services? Assume that several megabytes of data have to be transmitted for every photo.

10.5. Given an RSA signature scheme with the public key $(n = 9797, e = 131)$, which of the following signatures are valid?

1. $(x = 123, \text{sig}(x) = 6292)$
2. $(x = 4333, \text{sig}(x) = 4768)$
3. $(x = 4333, \text{sig}(x) = 1424)$

10.6. Given an RSA signature scheme with the public key $(n = 9797, e = 131)$, show how Oscar can perform an existential forgery attack by providing an example of such for the parameters of the RSA digital signature scheme.

10.7. In an RSA digital signature scheme, Bob signs messages x_i and sends them together with the signatures s_i and her public key to Alice. Bob's public key is the pair (n, e); her private key is d.

Oscar can perform man-in-the-middle attacks, i.e., he can replace Bob's public key by his own on the channel. His goal is to alter messages and provide these with a digital signature which will check out correctly on Alice's side. Show everything that Oscar must do for a successful attack.

10.8. Given is an RSA signature scheme with EMSA-PSS padding as shown in Sect. 10.2.3. Describe the verification process step-by-step that has to be performed by the receiver of a signature that was EMSA-PSS encoded.

10.9. One important aspect of digital signatures is the computational effort required to (i) sign a message, and (ii) to verify a signature. We study the computational complexity of the RSA algorithm used as a digital signature in this problem.

1. How many multiplications do we need, on average, to perform (i) signing of a message with a general exponent, and (ii) verification of a signature with the short exponent $e = 2^{16} + 1$? Assume that n has $l = \lceil \log_2 n \rceil$ bits. Assume the square-and-multiply algorithm is used for both signing and verification. Derive general expressions with l as a variable.
2. Which takes longer, signing or verification?
3. We now derive estimates for the speed of actual software implementation. Use the following timing model for multiplication: The computer operates with 32-bit data structures. Hence, each full-length variable, in particular n and x, is represented by an array with $m = \lceil l/32 \rceil$ elements (with x being the basis of the exponentiation operation). We assume that one multiplication or squaring of two of these variables modulo n takes m^2 time units (a time unit is the clock period times some constant larger than one which depends on the implementation). Note that you never multiply with the exponents d and e. That means, the bit length of the exponent does not influence the time it takes to perform an individual modular squaring or multiplication.
   How long does it take to compute a signature/verify a signature if the time unit on a certain computer is 100 nsec, and n has 512 bits? How long does it take if n has 1024 bit?
4. Smart cards are one very important platform for the use of digital signatures. Smart cards with an 8051 microprocessor kernel are popular in practice. The 8051 is an 8-bit processor. What time unit is required in order to perform one signature generation in 0.5 sec if n has (i) 512 bits and (ii) 1024 bits? Since these processors cannot be clocked at more than, say, 10 MHz, is the required time unit realistic?

10.10. We now consider the Elgamal signature scheme. You are given Bob's private key $K_{pr} = (d) = (67)$ and the corresponding public key $K_{pub} = (p, \alpha, \beta) = (97, 23, 15)$.

1. Calculate the Elgamal signature (r, s) and the corresponding verification for a message from Bob to Alice with the following messages x and ephemeral keys k_E:

   a. $x = 17$ and $k_E = 31$

b. $x = 17$ and $k_E = 49$
c. $x = 85$ and $k_E = 77$

2. You receive two alleged messages x_1, x_2 with their corresponding signatures (r_i, s_i) from Bob. Verify whether the messages $(x_1, r_1, s_1) = (22, 37, 33)$ and $(x_2, r_2, s_2) = (82, 13, 65)$ both originate from Bob.
3. Compare the RSA signature scheme with the Elgamal signature scheme. Where are their relative advantages and drawbacks?

10.11. Given is an Elgamal signature scheme with $p = 31$, $\alpha = 3$ and $\beta = 6$. You receive the message $x = 10$ twice with the signatures (r, s):

$$(i) \quad (17, 5)$$
$$(ii) \quad (13, 15)$$

1. Are both signatures valid?
2. How many valid signatures are there for each message x and the specific parameters chosen above?

10.12. Given is an Elgamal signature scheme with the public parameters $(p = 97, \alpha = 23, \beta = 15)$. Show how Oscar can perform an existential forgery attack by providing an example for a valid signature.

10.13. Given is an Elgamal signature scheme with the public parameters $p, \alpha \in \mathbb{Z}_p^*$ and an unknown private key d. Due to faulty implementation, the following dependency between two consecutive ephemeral keys is fulfilled:

$$k_{E_{i+1}} = k_{E_i} + 1.$$

Furthermore, two consecutive signatures to the plaintexts x_1 and x_2

$$(r_1, s_1)$$
$$\text{and} \quad (r_2, s_2)$$

are given. Explain how an attacker is able to calculate the private key with the given values.

10.14. The parameters of DSA are given by $p = 59, q = 29, \alpha = 3$, and Bob's private key is $d = 23$. Show the process of signing (Bob) and verification (Alice) for following hash values $h(x)$ and ephemeral keys k_E:

1. $h(x) = 17, k_E = 25$
2. $h(x) = 2, k_E = 13$
3. $h(x) = 21, k_E = 8$

10.15. Show how DSA can be attacked if the same ephemeral key is used to sign two different messages.

10.16. The parameters of ECDSA are given by the curve $E : y^2 = x^3 + 2x + 2 \bmod 17$, the point $A = (5, 1)$ of order $q = 19$ and Bob's private $d = 10$. Show the process of signing (Bob) and verification (Alice) for following hash values $h(x)$ and ephemeral keys k_E:

1. $h(x) = 12, k_E = 11$
2. $h(x) = 4, k_E = 13$
3. $h(x) = 9, k_E = 8$