to solve. Thus, choosing these parameters weakens the cryptosystem. Ways to generate elliptic curves are being studied [440]. Several parameter sets have been recommended for use. The U.S. NIST recommends curves P-192, P-224, P-256, P-384, or P-521 for elliptic curves using a prime modulus, and degree 163, 233, 283, 409, or 571 binary fields [2148]. Certicom recommends these as well, except that the degree 233 binary field is replaced by a degree 239 binary field [2231]. These curves are widely used. Some questions have been raised about the strength of these curves [181]. Other proposed curves include those of the Brainpool standard [1203], Curve1174 [182], Curve25519 [180], and several others [271].

The advantage to using elliptic curves over other forms of public key cryptography is that the keys can be shorter, and hence the computation time is shorter. As an example, elliptic curve cryptography with a key length of 256 to 383 bits provides a level of security comparable to RSA with a modulus of 3,072 bits [126]. Koblitz, Koblitz, and Menezes [1081] review how elliptic curve cryptography became widely accepted, with a discussion of it and RSA.

10.4 Cryptographic Checksums

Suppose Alice wants to send Bob a message of n bits. She wants Bob to be able to verify that the message he receives is the same one that was sent. So she applies a mathematical function, called a *checksum function*, to generate a smaller set of k bits from the original n bits. This smaller set is called the *checksum* or *message digest*. Alice then sends Bob both the message and the associated checksum. When Bob gets the message, he recomputes the checksum and compares it with the one Alice sent. If they match, he assumes that the message has not been changed; if they do not match, then either the message or the checksum has changed, and so they cannot be trusted to be what Alice sent him.

Of course, an adversary can change the message and alter the checksum to correspond to the message. For the moment, assume this will not happen; we will relax this assumption in Section 10.5.

EXAMPLE: The parity bit in the ASCII representation is often used as a single-bit checksum. If *odd parity* is used, the sum of the 1 bits in the ASCII representation of the character, and the parity bit, is odd. Assume that Alice sends Bob the letter "A." In ASCII, the representation of "A" using odd parity is $p1000001$ in binary, where p represents the parity bit. Because two bits are set, the parity bit is 1 for odd parity.

When Bob gets the message 11000001, he counts the 1 bits in the message. Because this number is odd, Bob believes that the message has arrived unchanged.

To minimize the probability that a change to either the message or the checksum will be detected, the checksum function must satisfy specific properties.

> **Definition 10–4.** A *cryptographic checksum function* (also called a *strong hash function* or a *strong one-way function*) $h : A \rightarrow B$ is a function that has the following properties:
>
> 1. For any $x \in A$, $h(x)$ is easy to compute.
> 2. For any $y \in B$, it is computationally infeasible to find $x \in A$ such that $h(x) = y$.
> 3. It is computationally infeasible to find $x, x' \in A$ such that $x \neq x'$ and $h(x) = h(x')$. (Such a pair is called a *collision*.)

The third requirement is often stated as:

> 3′. Given any $x \in A$, it is computationally infeasible to find another $x' \in A$ such that $x \neq x'$ and $h(x) = h(x')$.

However, properties 3 and 3′ are subtly different. It is considerably harder to find an x' meeting the conditions in property 3′ than it is to find a pair x and x' meeting the conditions in property 3. To explain why, we need to examine some basics of cryptographic checksum functions.

Given that the checksum contains fewer bits than the message, several messages must produce the same checksum. Ideally, the hashes of all possible messages will be evenly distributed over the set of possible checksums. Furthermore, the checksum that any given message produces can be determined only by computing the checksum. Such a checksum function acts as a random function.

The size of the output of the cryptographic checksum is an important consideration owing to a mathematical principle called the *pigeonhole principle*.

> **Definition 10–5.** The *pigeonhole principle* states that if there are n containers for $n + 1$ objects, at least one container will hold two objects.

To understand its application here, consider a cryptographic checksum function that computes hashes of three bits and a set of files each of which contains five bits. This yields $2^3 = 8$ possible hashes for $2^5 = 32$ files. Hence, at least four different files correspond to the same hash.

Now assume that a cryptographic checksum function computes hashes of 128 bits. The probability of finding a message corresponding to a given hash is 2^{-128}, but the probability of finding two messages with the same hash (that is, with the value of neither message being constrained) is 2^{-64} (see Exercise 24).

Definition 10–6. A *keyed* cryptographic checksum function requires a cryptographic key as part of the computation. A *keyless* cryptographic checksum does not.

Many keyless hash functions have been developed. The best known are MD4 [1593] and MD5 [1594] (128-bit checksums), RIPEMD-160 [575] (160-bit checksum), HAVAL [2096] (128-, 160-, 192-, 224-, and 256-bit checksums), and the Secure Hash Algorithm family of hash functions [2214] (SHA-1, SHA-224, SHA-256, SHA-384, SHA-512). Dobbertin devised a method for generating collisions in MD4 [574] and MD5 [573]. Wang and Yu used a differential attack to find collisions in MD4 and MD5 very quickly [1972]; their attack also works on HAVAL-128, RIPEMD, and the original version of SHA (now called SHA-0) [2213]. Various techniques for finding collisions have found collisions in several SHA hash functions, all on versions with a reduced number of steps (such as 58-step SHA-1 [1971], 70-step SHA-1 [516], 24-step SHA-2, SHA-256, and SHA-512 [1649], and 38-step SHA-256 [1317]).

In 2012, the U.S. NIST selected the Keccak hash function as SHA-3 [2229]. The design of Keccak prevents many of the collision-finding attacks that succeeded in previous hash functions [491]. In 2013, however, a different kind of attack succeeded with 3-round Keccak-384 and Keccak-512, and 5-round Keccak-256 [571]. Only the future will tell whether SHA-3 continues to be as robust as is believed.

10.4.1 HMAC

HMAC is a generic term for an algorithm that uses a keyless hash function and a cryptographic key to produce a keyed hash function [1102]. This mechanism enables Alice to validate that data Bob sent to her is unchanged in transit. Without the key, anyone could change the data and recompute the message authentication code, and Alice would be none the wiser.

The need for HMAC arose because keyed hash functions are derived from cryptographic algorithms. Many countries restrict the import and export of software that implements such algorithms. They do not restrict software implementing keyless hash functions, because such functions cannot be used to conceal information. Hence, HMAC builds on a keyless hash function using a cryptographic key to create a keyed hash function.

Let h be a keyless hash function that hashes data in blocks of b bytes to produce a hash l bytes long. Let k be a cryptographic key. We assume that the length of k is no greater than b; if it is, use h to hash it to produce a new key of length b. Let k' be the key k padded with bytes containing 0 to make b bytes. Let *ipad* be a sequence of bytes containing the bits 00110110 and repeated b times; let

opad be a similar sequence with the bits 01011100. The HMAC-h function with key k for message m is

$$\text{HMAC-}h(k, m) = h(k' \oplus opad \;||\; h(k' \oplus ipad||m))$$

where $\oplus$ is exclusive or and $||$ is concatenation.

Bellare, Canetti, and Krawczyk [155] analyze the security of HMAC and conclude that the strength of HMAC depends on the strength of the hash function h. Emphasizing this, attacks on HMAC-MD4, HMAC-MD5, HMAC-SHA-0, and HMAC-SHA-1 have been developed, some of which recover partial [451, 1054] or full keys [710,1965]. Bellare [154] extends the analysis by explaining under what conditions HMAC is secure.

Various HMAC functions are used in Internet security protocols (see Chapter 12).

10.5 Digital Signatures

As electronic commerce grows, so does the need for a provably high degree of authentication and integrity. Think of Alice's signature on a contract with Bob. Bob not only has to know that Alice is the other signer and is signing it; he also must be able to prove to a disinterested third party (called a *judge*) that Alice signed it and that the contract he presents has not been altered since Alice signed it. Such a construct plays a large role in managing cryptographic keys as well. This construct is called a *digital signature*.

> **Definition 10–7.** A *digital signature* is a construct that authenticates both the origin and contents of a message in a manner that is provable to a disinterested third party.

The "proof" requirement introduces a subtlety. Let m be a message. Suppose Alice and Bob share a secret key k. Alice sends Bob the message and its encipherment using k. Is this a digital signature?

First, Alice has authenticated the contents of the message, because Bob deciphers the enciphered message and can check that the message matches the deciphered one. Because only Bob and Alice know k, and Bob knows that he did not send the message, he concludes that it has come from Alice. He has authenticated the message origin and integrity. However, based on the mathematics alone, Bob cannot prove that he did not create the message, because he knows the key used to create it. Hence, this is not a digital signature.

Public key cryptography solves this problem. Let d_{Alice} and e_{Alice} be Alice's private and public keys, respectively. Alice sends Bob the message and its encipherment using d_{Alice}. As before, Bob can authenticate the origin and contents of the message, but in this situation a judge can determine that Alice signed the