Chapter 2

Cryptographic Building Blocks

This chapter introduces basic cryptographic mechanisms that serve as foundational building blocks for computer security: symmetric-key and public-key encryption, public-key digital signatures, hash functions, and message authentication codes. Other mathematical and crypto background is deferred to specific chapters as warranted by context. For example, Chapter 3 provides background on (Shannon) *entropy* and one-time password *hash chains*, while Chapter 4 covers *authentication protocols* and key establishment including Diffie-Hellman key agreement. Digital certificates are introduced here briefly, with detailed discussion delayed until Chapter 8.

If computer security were house-building, cryptography might be the electrical wiring and power supply. The framers, roofers, plumbers, and masons must know enough to not electrocute themselves, but need not understand the finer details of wiring the main panelboard, nor all the electrical footnotes in the building code. However, while our main focus is not cryptography, we should know the best tools available for each task. Many of our needs are met by understanding the properties and interface specifications of these tools—in this book, we are interested in their input-output behavior more than internal details. We are more interested in helping readers, as software developers, to properly use cryptographic toolkits, than to build the toolkits, or design the algorithms within them.

We also convey a few basic rules of thumb. One is: do not design your own cryptographic protocols or algorithms.[1] Plugging in your own desk lamp is fine, but leave it to a master electrician to upgrade the electrical panel.

2.1 Encryption and decryption (generic concepts)

An *algorithm* is a series of steps, often implemented in software programs or hardware. *Encryption* (and *decryption*) algorithms are a fundamental means for providing data confidentiality, especially in distributed communications systems. They are parameterized by a *cryptographic key*; think of a key as a binary string representing a large, secret number.

[1] This follows principle P9 (TIME-TESTED-TOOLS) from Chapter 1. The example on page 33 illustrates.

PLAINTEXT AND CIPHERTEXT. Encryption transforms data (*plaintext*) into an unintelligible form (*ciphertext*). The process is reversible: a *decryption key* allows recovery of plaintext, using a corresponding decryption algorithm. Access to the decryption key controls access to the plaintext; thus (only) authorized parties are given access to this key. It is generally assumed that the algorithms are known,[2] but that only authorized parties have the secret key. Sensitive information should be encrypted before transmission (assume communicated data is subject to *eavesdropping*, and possibly modification), and before saving to storage media if there is concern about adversaries accessing the media.

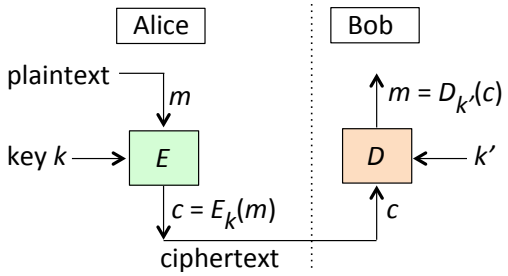

Figure 2.1: Generic encryption (E) and decryption (D). For symmetric encryption, E and D use the same shared (symmetric) key $k = k'$, and are thus inverses under that parameter; one is sometimes called the "forward" algorithm, the other the "inverse". The original Internet threat model (Chapter 1) and conventional cryptographic model assume that an adversary has no access to endpoints. This is false if malware infects user machines.

GENERIC ENCRYPTION NOTATION. Let m denote a plaintext message, c the ciphertext, and E_k, $D_{k'}$ the encryption, decryption algorithms parameterized by symmetric keys k, k' respectively. We describe encryption and decryption with equations (Figure 2.1):

$$c = E_k(m); \qquad m = D_{k'}(c) \tag{2.1}$$

Exercise (Caesar cipher). Caesar's famous cipher was rather simple. The encryption algorithm simply substituted each alphabetic plaintext character by that occurring three letters later in the alphabet. Describe the algorithms E and D of the Caesar cipher mathematically. What is the cryptographic key? How many other keys could be chosen?

In the terminology of mathematicians, we can describe an encryption-decryption system (*cryptosystem*) to consist of: a set $\mathcal{P}$ of possible plaintexts, set $\mathcal{C}$ of possible ciphertexts, set $\mathcal{K}$ of keys, an encryption mapping E: $(\mathcal{P} \times \mathcal{K}) \to \mathcal{C}$ and corresponding decryption mapping D: $(\mathcal{C} \times \mathcal{K}) \to \mathcal{P}$. But such notation makes it all seem less fun.

EXHAUSTIVE KEY SEARCH. We rely on cryptographers to provide "good" algorithms E and D. A critical property is that it be infeasible to recover m from c without knowledge of k'. The best an adversary can then do, upon intercepting a ciphertext c, is to go through all keys k from the *key space* $\mathcal{K}$, parameterizing D with each k sequentially, computing each $D_k(c)$ and looking for some meaningful result; we call this an *exhaustive key search*. If there are no algorithmic weaknesses, then no algorithmic "shortcut" attacks

[2] This follows the OPEN-DESIGN principle P3 from Chapter 1.

exist, and the whole key space must be tried. More precisely, an attacker of average luck is expected to come across the correct key after trying half the key space; so, if the keys are strings of 128 bits, then there are 2^{128} keys, with success expected after 2^{127} trials. This number is so large that we will all be long dead (and cold!)[3] before the key is found.

Example *(DES key space).* The first cipher widely used in industry was DES, standardized by the U.S. government in 1977. Its key length of 56 bits yields 2^{56} possible keys. To visualize key search on a space this size, imagine keys as golf balls, and a 2400-mile super-highway from Los Angeles to New York, 316 twelve-foot lanes wide and 316 lanes tall. Its entire volume is filled with white golf balls, except for one black ball. Your task: find the black ball, viewing only one ball at a time. (By the way, DES is no longer used, as modern processors make exhaustive key search of spaces of this size too easy!)

‡**CIPHER ATTACK MODELS.**[4] In a *ciphertext-only* attack, an adversary tries to recover plaintext (or the key), given access to ciphertext alone. Other scenarios, more favorable to adversaries, are sometimes possible, and are used in evaluation of encryption algorithms. In a *known-plaintext* attack, given access to some ciphertext and its corresponding plaintext, adversaries try to recover unknown plaintext (or the key) from further ciphertext. A *chosen-plaintext* situation allows adversaries to choose some amount of plaintext and see the resulting ciphertext. Such additional control may allow advanced analysis that defeats weaker algorithms. Yet another attack model is a *chosen-ciphertext* attack; here for a fixed key, attackers can provide ciphertext of their choosing, and receive back the corresponding plaintext; the game is to again deduce the secret key, or other information sufficient to decrypt new ciphertext. An ideal encryption algorithm resists all these attack models, ruling out algorithmic "shortcuts", leaving only exhaustive search.

PASSIVE VS. ACTIVE ADVERSARY. A *passive adversary* observes and records, but does not alter information (e.g., ciphertext-only, known-plaintext attacks). An *active adversary* interacts with ongoing transmissions, by injecting data or altering them, or starts new interactions with legitimate parties (e.g., chosen-plaintext, chosen-ciphertext attacks).

2.2 Symmetric-key encryption and decryption

We distinguish two categories of algorithms: symmetric-key or *symmetric* encryption (also called *secret-key*), and *asymmetric* encryption (also called *public-key*). In symmetric-key encryption, the encryption and decryption keys are the same, i.e., $k = k'$ in equation (2.1). In public-key systems they differ, as we shall see. We introduce symmetric encryption with the following example of a stream cipher.

Example *(Vernam cipher).* The *Vernam cipher* encrypts plaintext one bit at a time (Figure 2.2). It needs a key as long as the plaintext. To encrypt a t-bit message $m_1 m_2 ... m_t$,

[3] Our sun's lifetime, approximately 10 billion years, is $< 2^{60}$ seconds. Thus even if $10^{15} \approx 2^{50}$ keys were tested per second, the time to find the correct 128-bit key would exceed $2^{17} = 128,000$ lifetimes of the sun. Nonetheless, for SUFFICIENT-WORK-FACTOR (P12), and mindful that serious attackers harness enormous numbers of processors in parallel, standards commonly recommend symmetric keys be *at least* 128 bits.

[4] The symbol ‡ denotes research-level items, or notes that can be skipped on first reading.

using key $k = k_1k_2...k_t$, the algorithm is bitwise exclusive-OR: $c_i = m_i \oplus k_i$ yielding ciphertext $c = c_1c_2...c_t$. Plaintext recovery is again by exclusive-OR: $m_i = c_i \oplus k_i$. If k is randomly chosen and never reused, the Vernam stream cipher is called a *one-time pad*. One-time pads are known to provide a theoretically unbreakable encryption system. As a proof sketch, consider a fixed ciphertext $c = c_1c_2...c_t$. For every possible plaintext $m = m_1m_2...m_t$, there is a key k such that c decrypts to m, defined by $k_i = c_i \oplus m_i$; thus c may originate from any possible plaintext. (Convince yourself of this with a small example, encoding lowercase letters a-z using 5 bits each.) Observing c tells an attacker only its length. Despite this strength, one-time pads are little-used in practice: single-use, long keys are difficult to distribute and manage, and if you can securely distribute a secret key as long as the message, you could use that method to deliver the message itself.

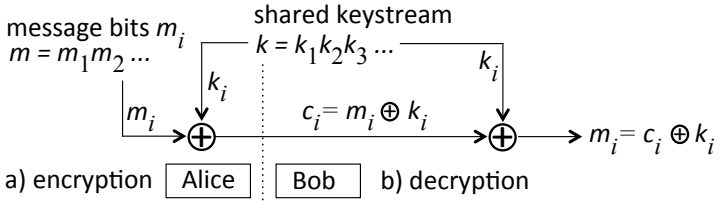

Figure 2.2: Vernam cipher. If the keystream is a sequence of truly random, independent bits that is never reused, then this is an unbreakable one-time pad. Practical encryption systems aim to mimic the one-time pad using shortcuts without compromising security.

Example *(One-time pad has no integrity).* The one-time pad is theoretically unbreakable, in that the key is required to recover plaintext from its ciphertext. Does this mean it is secure? The answer depends on your definition of "secure". An unexpected property is problematic here: *encryption alone does not guarantee integrity*. To see this, suppose your salary is \$65,536, or in binary `(00000001 00000000 00000000)`. Suppose this value is stored in a file after one-time pad encryption. To tamper, you replace the most significant ciphertext byte by the value obtained by XORing a 1-bit anywhere other than with its low-order bit (that plaintext bit is already 1). Now on decryption, the keystream bit XOR'd onto that bit position by encryption will be removed (Fig. 2.2), so regardless of the keystream bit values, your tampering has flipped the underlying plaintext bit (originally 0). Congratulations on your pay raise! This illustrates how intuition can mislead us, and motivates a general rule: use only cryptographic algorithms both designed by experts, and having survived long scrutiny by others; similarly for cryptographic protocols (Chapter 4). As experienced developers know, even correct use of crypto libraries is challenging.

‡**CIPHER ATTACKS IN PRACTICE.** The one-time pad is said to be *information-theoretically secure* for confidentiality: even given unlimited computing power and time, an attacker without the key cannot recover plaintext from ciphertext. Ciphers commonly used in practice offer only *computational security*,[5] protecting against attackers modeled as having fixed computational resources, and thus assumed to be unable to exhaustively try all keys in huge key spaces. Such ciphers may fail due to algorithmic weaknesses, or

[5]Computational security is also discussed with respect to hash functions in Section 2.5.