Computer Security and the Internet: Tools and Jewels

Chapter 2: Cryptographic Building Blocks

Sept 24, 2018

Comments, corrections, and suggestions for improvements are welcome and appreciated.
Please send by email to: paulv@scs.carleton.ca

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* 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 panel-board, nor all the electrical footnotes in the building code. So 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 algorithms, nor your own cryptographic protocols.[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, implemented in a software program or in hardware. *Encryption* algorithms, and corresponding *decryption* algorithms, are the fundamental means for providing data confidentiality. They are parameterized by a *cryptographic key*; think of a key as a binary string representing a large, secret number.

[1]This follows the TIME-TESTED-TOOLS principle P9 from Chapter 1.

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 out to storage media if there is concern of adversaries accessing the media.

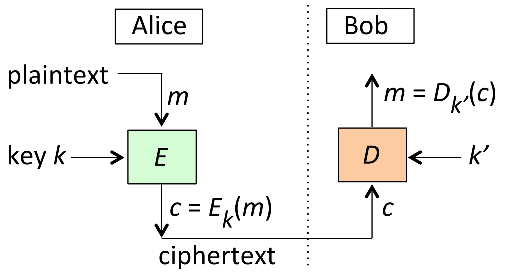

Figure 2.1: Generic encryption (E) and decryption (D). For symmetric encryption, E and D use the same shared (symmetric) key $k = k'$, and are thus inverses under that parameter; one is sometimes called the "forward" algorithm, the other the "inverse". The original Internet threat model (Chapter 1) and 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: $(\mathcal{P} \times \mathcal{K}) \rightarrow \mathcal{C}$ and corresponding decryption mapping D: $(\mathcal{C} \times \mathcal{K}) \rightarrow \mathcal{P}$. 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 keyspace $\mathcal{K}$, parameterizing D with each k sequentially,

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

computing each $D_k(c)$ and looking for some meaningful result; we call this an *exhuastive key search*. If there are no algorithmic weaknesses, then no algorithmic "short-cut" attacks exist, and the whole keyspace 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 even if the attacker is able to use all computers in existence for this task, 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, because modern processors make exhaustive key search of spaces this size too easy!)

‡CIPHER ATTACK MODELS.[4] In a *cipher-text 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 a fixed amount of 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 defeating weaker algorithms. Yet another attack model is *chosen-ciphertext* attack, wherein 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 "short-cuts", 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. Symmetric encryption is most easily introduced with the following example.

[3]Our sun's lifetime is $\approx$ (10 billion years) $<$ (2^{60} seconds). Thus even if $10^{15} \approx 2^{50}$ keys were tested in one second, the expected time to find the correct key would exceed $2^{17} = 128$ thousand lifetimes of the sun. Nonetheless, many standards currently recommended that symmetric keys be at least 128 bits.

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

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 $m_1 m_2 ... m_t$, using key $k = k_1 k_2 ... k_t$, the algorithm is bitwise exclusive-OR: $c_i = m_i \oplus k_i$ yielding ciphertext $c = c_1 c_2 ... c_t$. Plaintext recovery is again by exclusive-OR: $m_i = c_i \oplus k_i$. If k is randomly chosen and never re-used, 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 $c = c_1 c_2 ... c_t$. For every possible plaintext $m = m_1 m_2 ... m_t$, 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.

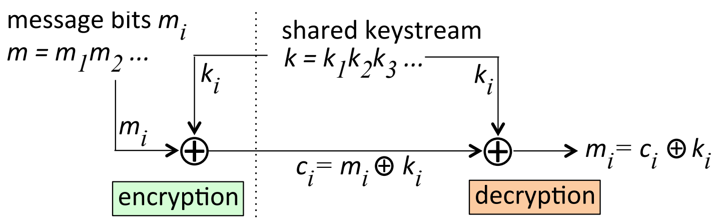

Figure 2.2: Vernam cipher. If the keystream is a sequence of truly random, independent bits that is never re-used, then this an unbreakable "one-time pad". Practical encryption systems aim to mimic the one-time pad using shortcuts without compromising security.

Example *(One-time pad has no integrity).* Since the one-time pad is theoretically unbreakable, it is impossible 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 to 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 (see above), 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 theoretically unbreakable—even given unlimited computational power and time, an attacker without the key cannot recover plaintext from ciphertext—and is thus said to be *information-theoretically secure*

for confidentiality. Ciphers commonly used in practice offer less: *computational security*,[5] i.e., protection against attackers modelled to have fixed computational resources, and thus assumed unable to exhaustively try all keys in huge keyspaces. Such ciphers may fall to unknown weaknesses, or if a key space is so small that all keys can be tested within a reasonable time. Exhaustive key-guessing attacks require an automated method to signal if a key-guess is correct—this is often done using a known plaintext-ciphertext pair, or recognition of redundancy such as ASCII coding in the decrypted bitstream.

STREAM CIPHERS. The Vernam cipher is an example of a *stream cipher*, which in simplest form, involves generating a keystream simply XOR'd onto plaintext bits; decryption involves XORing the ciphertext with the same keystream. In contrast to block ciphers (below), there are no requirements that the plaintext length be a multiple of, e.g., 64 or 128 bits. Thus stream ciphers are often favoured when there is need to encrypt plaintext one character or even a single bit at a time, e.g., in some real-time applications. A simplified view of stream ciphers is that they turn a fixed-size secret (symmetric key) into an arbitrary-length secret keystream unpredictable to adversaries. The mapping of the next plaintext bit to ciphertext is a position-varying transformation dependent on the input key.

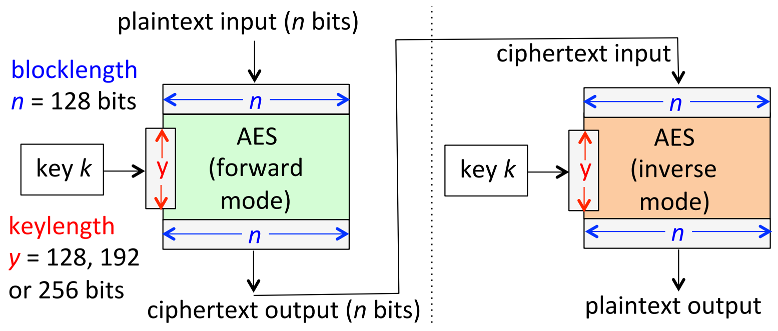

Figure 2.3: AES input-output interface (a block cipher example). For a fixed key k, a block cipher with n-bit blocklength is a permutation that maps each of 2^n possible input blocks to a unique n-bit output block, and the inverse mode does the reverse mapping (as required to recover the plaintext). Ideally, each k defines a different permutation.

BLOCK CIPHERS, BLOCK SIZE, KEY SIZE. In contrast, a second class of symmetric ciphers, *block ciphers*, processes plaintext in fixed-length chunks or *blocks*. Each block, perhaps a group of ASCII-encoded characters, is encrypted with a fixed transformation dependent on the key. From a black-box (input-output) perspective, a block cipher's main properties are *blocklength* (block size in bits) and *keylength* (key size in bits). When using a block cipher, if the last plaintext block has fewer bits than the block size, it is *padded* by "filler" characters. A common non-ambiguous padding rule is to always append a 1-bit, followed by zero or more 0-bits as necessary to fill out the block.

AES BLOCK CIPHER. Today's most widely used block cipher is AES, specified by

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

the *Advanced Encryption Standard*. Created by researchers at Flemish university K.U. Leuven, the algorithm itself (Rijndael) was selected after an open, multi-year competition run by the (U.S.) National Institute of Standards and Technology (NIST). Similar NIST competitions resulted in SHA-1, SHA-2 and SHA-3 (Section 2.5). Figure 2.3 shows the input-output interface of AES; Table 2.2 (Section 2.7) compares with other algorithms.

‡MESSAGE EXPANSION. Symmetric ciphers are typically *length-preserving*, i.e., the ciphertext consumes no more space than the plaintext, in which case *in-place encryption* is possible (e.g., in a storage context, plaintext may be replaced by ciphertext without requiring additional memory). If, however, integrity guarantees are required (AEAD below), the ciphertext is accompanied by an authentication tag implying *message expansion*. Additional space may also be needed for related parameters (e.g., IVs or nonces, below).

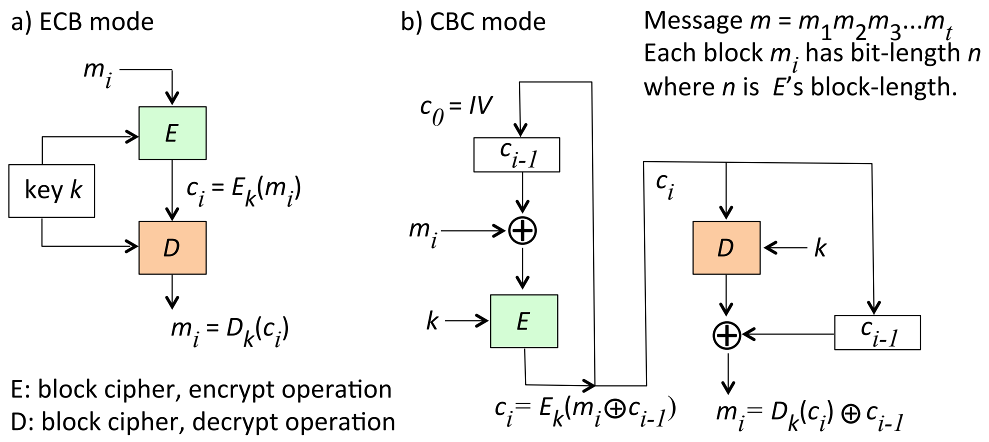
a) ECB mode
m_i
E
key k
$c_i = E_k(m_i)$
D
$m_i = D_k(c_i)$
E: block cipher, encrypt operation
D: block cipher, decrypt operation
b) CBC mode
Message $m = m_1m_2m_3...m_t$
Each block m_i has bit-length n
where n is E's block-length.
$c_0 = IV$
c_{i-1}
m_i
k
E
$c_i = E_k(m_i \oplus c_{i-1})$
c_i
D
k
c_{i-1}
$m_i = D_k(c_i) \oplus c_{i-1}$

Figure 2.4: ECB and CBC modes of operation. The plaintext $m = m_1m_2\cdots m_t$ becomes ciphertext $c = c_1c_2\cdots c_t$ of the same length. $\oplus$ denotes bitwise exclusive-OR. Here the IV (*initialization vector*) is a bitstring of length equal to the cipher's blocklength (e.g., $n = 128$). In CBC mode, if a fixed m is encrypted under the same key k and same IV, the resulting ciphertext blocks are the same each time; changing the IV disrupts this.

ECB ENCRYPTION AND MODES OF OPERATION. Let E denote a block cipher with blocklength n. Say $n = 128$. If a plaintext m has bitlength exactly n also, equation (2.1) is used directly with just one 128-bit "block operation". Longer plaintexts are broken into 128-bit blocks for encryption—so a 512-bit plaintext is processed in four blocks. The block operation maps each of the 2^{128} possible 128-bit input blocks to a distinct 128-bit ciphertext block (this allows the mapping to be reversed; the block operation is a *permutation*). Each key defines a fixed such "code-book" mapping. In the simplest case (Figure 2.4a), each encryption block operation is independent of adjacent blocks; this is called *electronic code-book* (ECB) mode of the block cipher E. If a given key k is used to encrypt several identical plaintext blocks m_i, then identical ciphertext blocks c_i result; ECB mode does not hide such patterns. This information leak can be addressed by including random bits within a reserved field in each block, but that is inefficient and awkward. Instead, various methods called *modes of operation* (below) combine successive n-bit block operations such that the encryption of one block depends on other blocks.

BLOCK CIPHER MODE EXAMPLES: CBC, CTR. For above reasons, ECB mode is discouraged for messages exceeding one block, and if one key is used for multiple messages. Instead, standard block cipher *modes of operation* are used to make block encryptions depend on adjacent blocks (the block encryption mapping is then context-sensitive). Figure 2.4 illustrates the historical *cipher-block chaining* (CBC) mode of operation; others, including CTR mode (Figure 2.5), are now generally recommended over CBC, for technical reasons beyond our scope. Some modes, including CTR, use the block cipher to produce a keystream and effectively operate as a stream cipher processing "large" characters; the distinction between block and stream ciphers then becomes fuzzy.

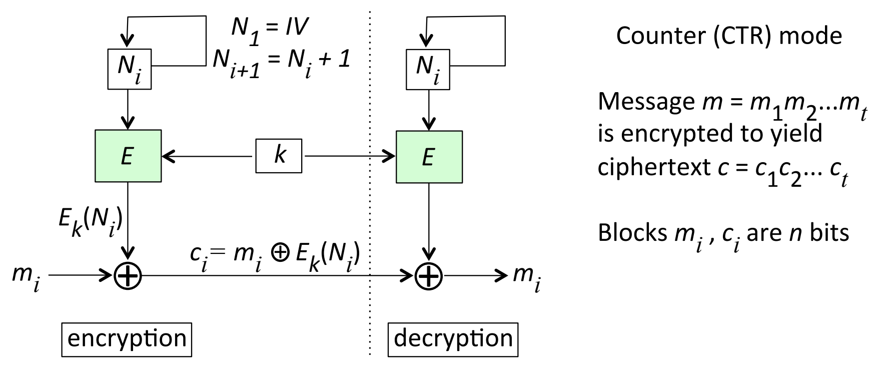
$N_1 = IV$
$N_{i+1} = N_i + 1$
N_i
N_i
E
k
E
$E_k(N_i)$
$c_i = m_i \oplus E_k(N_i)$
m_i
m_i
encryption
decryption
Counter (CTR) mode
Message $m = m_1m_2...m_t$ is encrypted to yield ciphertext $c = c_1c_2... c_t$
Blocks m_i, c_i are n bits

Figure 2.5: Counter (CTR) mode of operation. E denotes a block cipher (encrypt operation) with blocklength n ($n = 128$ is common). CTR mode ECB-encrypts an incrementing index (counter) to generate a keystream of blocks to XOR onto corresponding plaintext blocks. To reverse the process, decryption regenerates the same keystream using ECB encryption (note that the "inverse" algorithm, or ECB decryption, is not used in this case).

‡**Exercise** (ECB leakage of patterns). For a picture with large uniform colour patterns, obtain its uncompressed bitmap image file (each pixel's colour is represented using, e.g., 32 or 64 bits). Encrypt the bitmap using a block cipher of blocklength 64 or 128 bits. Report on any data patterns evident when the encrypted bitmap is displayed.

‡**Exercise** (Modes of operation: properties). Summarize the properties, advantages and disadvantages of the following long-standing modes of operation: ECB, CBC, CTR, CFB, OFB (hint: [22, pp.228-233] or [26]). Do the same for XTS (hint: [11]).

ENCRYPTION IN PRACTICE. In practice today, symmetric-key encryption is almost always accompanied by means to provide integrity protection (not just confidentiality). Such *authenticated encryption* is discussed in Section 2.7, after an explanation of message authentication codes (MACs) in Section 2.6.

2.3 Public-key encryption and decryption

For symmetric-key encryption, k denoted a key shared between two parties. For public-key encryption, we label keys with a subscript denoting the single party they belong to. In fact each party has a *key pair*, e.g., (e_B, d_B) for Bob, consisting of an *encryption public key* e_B which can be publicized as belonging to Bob, and a *decryption private key* d_B which Bob should keep secret and share with no one. (Of course, it may be prudent for Bob to back up d_B; and neither the primary copy, nor the backup, should ever appear in plaintext form in untrusted storage. Practical issues start to complicate things quickly!)

To public-key encrypt a message m for Bob, Alice obtains Bob's public key e_B, uses it to parameterize the associated public-key encryption algorithm E, encrypts m to ciphertext c per (2.2), and sends c to Bob (Figure 2.6). Bob recovers m using the corresponding known public-key decryption algorithm D, parameterized by his private key d_B.

$$c = E_{e_B}(m); \qquad m = D_{d_B}(c) \tag{2.2}$$

INTEGRITY OF PUBLIC KEY IS IMPORTANT. A public key can be published, for example like a phone number in an old-style phonebook. It need not be kept secret. But its integrity is critical—for, if Charlene could replace Bob's public key by her own, then someone who thought they were encrypting something under a public key for Bob's eyes only, would instead be making the plaintext recoverable by Charlene.

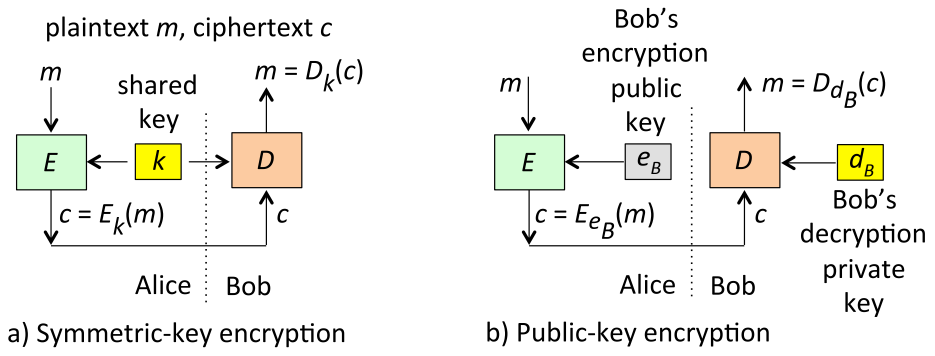

Figure 2.6: Comparing symmetric-key to public-key encryption-decryption. In the symmetric-key case, the same shared key is used to encrypt and decrypt. The public-key (asymmetric) case uses distinct encryption and decryption keys; one public, one private. Some people use "private key" to refer to the secrets in asymmetric systems, and "secret key" to refer to those in symmetric-key systems.

KEY DISTRIBUTION—SYMMETRIC VS. PUBLIC-KEY. If a group of n people wish to use symmetric encryption for pairwise confidential communications, then each pair should use (shared between the pair) a different symmetric key. This requires $\binom{n}{2} = n(n-1)/2$ keys, i.e., $O(n^2)$ keys. For $n = 4$ this is just 6, but for $n = 100$ this is already 4950.

As n grows, keys become unwieldy to distribute and manage securely. In contrast for public-key encryption, each party needs only one set of (public, private) keys in order to allow all other parties to encrypt for them—thus requiring only n key pairs in total.

HYBRID ENCRYPTION. Symmetric-key algorithms are typically faster than public-key algorithms. On the other hand, public-key methods are convenient for establishing shared secret keys between end-points (as just noted). Therefore, to send encrypted messages, often public-key methods are used to establish a shared symmetric-key k (*session key*) between communication end-points, and k is then used in a symmetric-key algorithm for efficient "bulk encryption" of a payload message m. See Figure 2.7. Thus a primary use of RSA encryption (below) is to encrypt relatively short data keys or session keys, i.e., for *key management* (Chapter 4), rather than for bulk encryption of messages themselves.

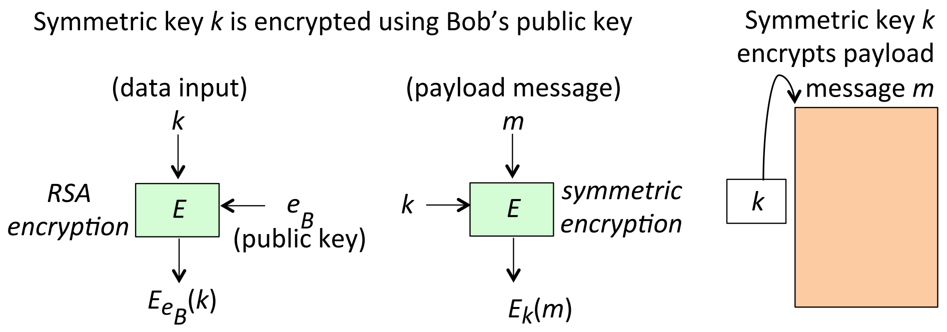

Figure 2.7: Hybrid encryption. The main data (payload message m) is encrypted using a symmetric key k, and k is made available by public-key methods. Here Alice creates k and using it with a symmetric-key algorithm, encrypts m. The message key k is encrypted for Bob alone using his RSA public key e_B. Alternatively, k is a shared Alice-Bob key established by Diffie-Hellman key agreement (Chapter 4).

‡MATH DETAILS: RSA PUBLIC-KEY ENCRYPTION. Here we outline the technical details of RSA, the first popular public-key encryption method. Per notation above, a party A has a public key e_A and private key d_A. When used to parameterize the corresponding algorithms E and D, the context is clear and we can use E_A and D_A to denote the parameterized algorithms, e.g., $E_{e_A}(m) \equiv E_A(m)$. For RSA, $e_A = (e, n)$. Here $n = pq$, and parameters e, d, p, q must satisfy various security properties. Those of present interest are that p and q are large primes (e.g., 1000 bits each), and e is an integer chosen such that:

$$gcd(e, \phi(n)) = 1 \qquad \text{where here, } \phi(n) = (p-1)(q-1)$$

gcd is the greatest common divisor function, and $\phi(n)$ is the *Euler phi function*, the number of integers in $[1, n]$ relatively prime to n; its main properties of present interest are that for a prime p, $\phi(p) = p - 1$, and that if p and q have no factors in common other than 1 then $\phi(pq) = \phi(p) \cdot \phi(q)$. For RSA, $d_A = (d, n)$ where d is computed to satisfy $ed \equiv 1(\bmod\ \phi(n))$, i.e., $ed = 1 +$ (some integer multiple of $\phi(n)$). Now let m be a message whose binary representation, interpreted as an integer, is less than n (e.g., 2000 bits).

RSA encryption of plaintext m: $c \equiv m^e \pmod n$
RSA decryption of ciphertext c: $m \equiv c^d \pmod n$

By this we mean, assign to c the number resulting from the modular exponentiation of m by the exponent e, reduced modulo n. Operations on numbers of this size require special "big number" support, provided by crypto libraries such as OpenSSL. Using RSA in practice is somewhat more complicated, but the above gives the basic technical details.

‡**Exercise** (RSA decryption). Using the above equations, show that RSA decryption actually works, i.e., recovers m.

‡**Exercise** (RSA toy example). You'd like to explain to a 10-year old how RSA works. Using $p = 5$ and $q = 7$, encrypt and decrypt a "message" (use a number less than n). Here $n = 35$, and $\phi(n) = (p-1)(q-1) = (4)(6) = 24$. Does $e = 5$ satisfy the rules? Does that then imply $d = 5$ to satisfy the required equation? Now with pencil and paper—yes, by hand!—compute the RSA encryption of $m = 2$ to ciphertext c, and the decryption of c back to m. The exponentiation is commonly done by repeated multiplication, reducing partial results mod n (i.e., subtract off multiples of the modulus 35 in interim steps). This example is so artificially small that the parameters "run into each other"—so perhaps for a 12-year-old, you might try an example using $p = 11$ and $q = 13$.

2.4 Digital signatures and verification using public-key

Digital signatures, typically computed using public-key algorithms, are tags (bitstrings) that accompany messages. Each tag is a mathematical function of a message (its exact bitstring) and a unique-per-sender private key. A corresponding public key, uniquely associated with the sender, allows automated verification that the message originated from that individual, since only that individual knows the private key needed to create the tag.

The name originates from the idea of a replacement (for digital documents) for handwritten signatures, with stronger assurances. The late 1990s saw considerable international effort towards deploying digital signatures as an actual (legally binding) replacement for handwritten signatures, but many legal and technical issues arose; in current practice, digital signatures are most commonly used for authentication purposes.

SIGNATURE PROPERTIES. Digital signatures provide three properties.

1. *Data origin authentication*: assurance of who originated (signed) a message or file.
2. *Data integrity*: assurance that received content is the same as that originally signed.
3. *Non-repudiation*: strong evidence of unique origination, making it hard for a party to digitally sign data and later successfully deny having done so. This is an important advantage over MACs (Section 2.6), and follows from signature verification not requiring the signer's private key—verifiers use the signer's public key.

NON-REPUDIATION IN PRACTICE. This property assumes that only the legitimate party has access to their own signing private key. One might try to deny having executed a signature by claiming "my private key spilled onto the street—someone else must be using it!" This assertion will raise suspicion if repeated, but highlights a critical requirement for digital signatures: ordinary users must somehow have the ability, by appropriate technology or training, to prevent others from accessing their private keys. Arguably, this has posed a barrier to digital signatures replacing handwritten signatures on legal documents, while their use for computer-related authentication applications faces lower barriers.

DETAILS OF PUBLIC-KEY SIGNATURES. Public-key methods can be used to implement digital signatures by a process similar to encryption-decryption, but with subtle differences (which thoroughly confuse non-experts). The public and private parts are used in reverse order (the originator uses the private key now), and the key used for signing is that of the message originator not the recipient. The details are as follows.

In place of encryption public keys, decryption private keys, and algorithms E, D (encrypt, decrypt), we now have *signing* private keys for signature generation, *verification* public keys to validate signatures, and algorithms S, V (sign, verify). To sign message m, Alice uses her signing private key s_A to create a tag $t_A = S_{s_A}(m)$ and sends (m, t_A). Upon receiving a message-tag pair (m', t'_A) (the notation change allowing that the pair sent might be modified en route), any recipient can use Alice's verification public key v_A to verify if t'_A is a matching tag for m' from Alice, by computing $V_{v_A}(m', t'_A)$. This returns VALID (if matching), otherwise INVALID. Just as for MAC tags (later), even if verification succeeds, in some applications it may be important to use additional means to confirm that (m', t'_A) is not simply a replay of an earlier legitimate signed message. See Figure 2.8.

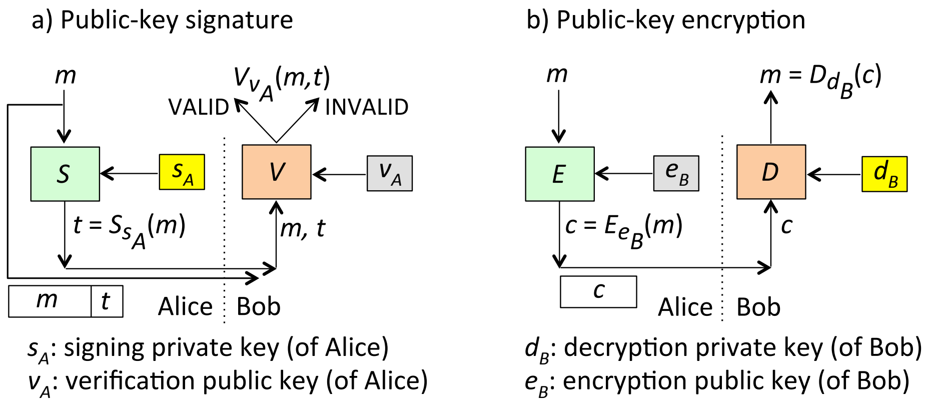

Figure 2.8: Public-key signature generation-verification vs. encryption-decryption. For Alice to encrypt for Bob, she must use *his* encryption public key; but to sign a message for Bob, she uses *her own* signature private key. In a), Alice sends to Bob a pair (m,t) providing message and signature tag, analogous to the (message, tag) pair sent when using MACs (Figure 2.11). Internal details on signature algorithm S are given in Figure 2.10.

Exercise (Combining signing and encrypting). Alice wishes to both encrypt, and sign, a message m for Bob. Specify the actions that Alice must carry out, and the data values to be sent to Bob. Explain your choice of whether signing or encryption should be done first. Be specific about what data values are included within the scope of the signature operation, and the encryption operation; use equations as necessary. Similarly specify the actions Bob must carry out to both decrypt the message and verify the digital signature. (Note the analogous question in Section 2.7 on how to combine MACs with encryption.)

DISTINCT TERMINOLOGY FOR SIGNATURES AND ENCRYPTION. Even among university professors, great confusion is caused by misusing encryption-decryption terminology to describe operations involving signatures. For example, it is common to hear and read that signature generation or verification involves "encrypting" or "decrypting" a message or its hash value. This unfortunate wording unnecessarily conflates distinct functions (signatures and encryption), and predisposes students—and more dangerously, software developers—to believe that it is acceptable to use the same (public, private) key pair for signatures and confidentiality. (Some signature algorithms are technically incompatible with encryption; the RSA algorithm can technically be used to provide both signatures and encryption, but proper implementations of these two functions differ considerably in detail, and it is prudent to use distinct key pairs.) Herein, we carefully avoid the terms *encryption* and *decryption* when describing digital signature operations, and also encourage using the terms *public-key operation* and *private-key operation*.

DIGITAL SIGNATURES IN PRACTICE. For efficiency reasons, digital signatures are commonly used in conjunction with hash functions, as explained in Section 2.5. This is one of several motivations for discussing hash functions next.

2.5 Cryptographic hash functions

Cryptographic hash functions help solve many problems in security. They take as input any binary string (e.g., message or file) and produce a fixed-length output called a *hash value*, *hash*, *message digest* or *digital fingerprint*. They typically map longer into shorter strings, as do other (non-crypto) hash functions in computer science, but have special properties. Hereafter, "hash function" means cryptographic hash function (Figure 2.9).

A hash value is ideally an efficiently computable, compact representation meant to be, in practice, uniquely identifiable with its input. For a good hash function, changing a single binary digit (*bit*) of input results in entirely unpredictable output changes (50% of output bits change on average). Hashes are often used as a type of secure checksum whose mappings are too complex to predict or manipulate—and thus hard to exploit.

PROPERTIES OF CRYPTOGRAPHIC HASH FUNCTIONS. We use H to denote a hash function algorithm. It is generally assumed that the details of H are openly known. We want functions H such that, given any input m, the computational cost to compute $H(m)$ is relatively small. Three hash function security properties are often needed in practice:

(H1) *one-way property* (or *preimage resistance*): for essentially all possible hash values h, given h it should be infeasible to find any m such that $H(m) = h$.

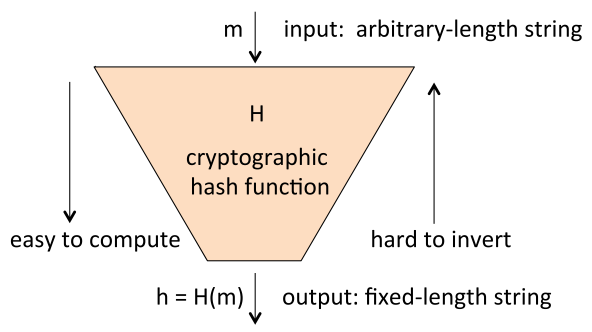
m
input: arbitrary-length string
H
cryptographic hash function
easy to compute
hard to invert
h = H(m)
output: fixed-length string

Figure 2.9: Cryptographic hash function. A base requirement is a "one-wayness" property. Depending on the application of use, additional technical properties are required.

(H2) *second-preimage resistance*: given any first input m_1, it should be infeasible to find any distinct second input m_2 such that $H(m_1) = H(m_2)$. (Note: there is free choice of m_2 but m_1 is fixed. $H(m_1)$ is the target image to match; m_1 is its *preimage*.)

(H3) *collision resistance*: it should be infeasible to find any pair of distinct inputs m_1, m_2 such that $H(m_1) = H(m_2)$. (Note: here there is free choice of both m_1 and m_2.)

The properties required vary across applications. As examples elaborated later, H1 is required for password hash chains and also for storing password hashes; for digital signatures, H2 is required if an attacker cannot choose a message for others to sign, while H3 is required if an attacker can choose the message to be signed by others.

COMPUTATIONAL SECURITY. The one-way property (H1) implies that given a hash value, an input that produces that hash cannot be easily found—even though many, many inputs do indeed map to each output. To see this, restrict your attention to only those inputs of exactly 512 bits, and suppose the hash function output has bitlength 128. Then H maps each of these 2^{512} input strings to one of 2^{128} possible output strings—so on average, 2^{384} inputs map to each 128-bit output. Thus enormous numbers of collisions exist, but they should be hard to find in practice; what we have in mind here is called *computational security*. Similarly, the term "infeasible" as used in (H1)-(H3) means computationally infeasible in practice, i.e., assuming all resources that an attacker might be able to harness over the period of desired protection (and erring on the side of caution for defenders).[6]

‡**Exercise** (CRC vs. cryptographic hash). Explain why a cyclical redundancy code (CRC) algorithm, e.g., a 16- or 32-bit CRC commonly used in network communications for integrity, is not suitable as a cryptographic hash function (hint: [22, p.363]).

Hash functions fall into two broad service classes in security, as discussed next.

ONE-WAY HASH FUNCTIONS. Applications in which "one-wayness" is critical (e.g., password hashing, below), require property H1. In practice, hash functions with H1 of-

[6]In contrast, in *information-theoretic security*, the question is whether, given unlimited computational power or time, there is sufficient information to solve a problem. That question is of less interest in practice.

ten also provide H2. We prefer to call the first property *preimage resistance*, because traditionally functions providing both H1 and H2 are called *one-way hash functions*.

COLLISION-RESISTANT HASH FUNCTIONS. A second usage class relies heavily on the requirement (property) that it be hard to find two inputs having the same hash. If not so, then in some applications using hash functions, an attacker finding such a pair of inputs might benefit by substituting a second such input in place of the first. As it turns out, second-preimage resistance (H2) fails to guarantee collision resistance (H3); for an attacker trying to find two strings yielding the same hash (i.e., a *collision*), fixing one string (say m_1 in H2) makes collision-finding significantly more costly than if given free choice of both m_1 and m_2. The reason is the *birthday paradox* (below). When it is important that finding collisions be computationally difficult even for an attacker free to choose both m_1 and m_2, *collision resistance* (H3) is specified as a requirement. It is easy to show that H3 implies second-preimage resistance (H2). Furthermore, in practice,[7] hash functions with H2 and H3 also have the one-way property (H1), thus giving all three. Thus as a single property in a hash function, H3 (collision-resistance) is most desirable.

Example *(Hash function used as modification detection code).* As an example application involving properties H1–H3 above, consider an executable file corresponding to program P with binary representation p, faithfully representing legitimate source code at the time P is installed in the file system. At that time, using a hash function H with properties H1-H3, the operating systems computes $h = H(p)$. This "trusted-good" hash of the program is stored in memory that is safe from manipulation by attackers. Later, before invoking program P, the operating system recomputes the hash of the executable file to be run, and compares the result to stored value h. If the values match, there is strong evidence that the file has not been manipulated or substituted by an attacker.

The process in this example provides a *data integrity* check for one file, immediately before execution. Data integrity for a designated *set* of system files could be provided as an ongoing background service by similarly computing and storing a set of trusted hashes (one per-file) at some initial time before exposure to manipulation, and then periodically recomputing the file hashes and comparing to the known-good stored values.[8]

‡**Exercise** (Hash function properties—data integrity). In the above example, was the collision-resistance property (H3) actually needed? Give one set of attack circumstances under which H3 is necessary, and a different scenario under which H needs only second-preimage resistance to detect an integrity violation on the protected file. (An analogous question arises regarding necessary properties of a hash function when used in conjunction with digital signatures, as discussed shortly.)

Example *(Using one-way functions in password verification).* One-way hash functions H are often used in password authentication as follows. A userid and password p entered on a client device are sent (hopefully over an encrypted link!) to a server. The server hashes the p received to $H(p)$, and uses the userid to index a data record contain-

[7] There are pathological examples of functions having H2 and H3 without the one-way property (H1), but in practice collision-resistance (H3) almost always implies H1 [22, p.330].

[8] An example of such a service is Tripwire, as detailed by Kim [15].

Family name	**Year**	**Output size**		**Alternate names and notes**
		bitlength	bytes	
SHA-3	2015	224, 256	28, 32	SHA3-224, SHA3-256
		384, 512	48, 64	SHA3-384, SHA3-512 (NOTE 1)
SHA-2	2001	256, 512	32, 64	SHA-256, SHA-512
SHA-1	1995	160	20	Theoretical attacks are known
MD5	1992	128	16	Deprecated for many applications

Table 2.1: Common hash functions and example parameters. Additional SHA-2 variants include SHA-224 (SHA-256 truncated to 224 bits), SHA-384 (SHA-512 truncated), and further SHA-512 variations which use specially computed initial values and truncate to 224 or 384 bits resp., giving SHA-512/224 and SHA-512/256. NOTE 1: SHA-3's most flexible variation allows arbitrary bitlength output; SHA-3 is based on the *Keccak* family.

ing the (known-correct) password hash. If the values match, login succeeds. This avoids storing, at the server, plaintext passwords which might be directly available to disgruntled administrators, anyone with access to backup storage, or breaking into the server database.

Exercise (Hash function properties—password hashing). In the example above, would a hash function having the one-way property, but not second-preimage resistance, be useful for password verification? Explain.

Exercise (Password hashing at the client end). The example using a one-way hash function in password verification motivates storing password hashes (vs. clear passwords) at the server. Suppose instead that passwords were hashed at the client side, and the password hash was sent to the server (rather than the password itself). Would this be helpful or not? Should the password hash be protected during transmission to the server?

Example *(Hash examples).* Table 2.1 shows common hash functions in use: SHA-3, SHA-2, SHA-1 and MD5. Among these, the more recently introduced versions, and those with longer outputs, are generally preferable choices from a security viewpoint. (Why?)

BIRTHDAY PARADOX. What number n of people are needed in a room before it is expected (i.e., with probability $p = 0.5$) to have a shared birthday among them? (Alternatively, given n people in a room, what is the probability that two of them have the same birthday?) Only about 23, and the probability rises rapidly with n: $p = 0.71$ for $n = 30$, and $p = 0.97$ for $n = 50$. The *birthday paradox* is so-named because the result surprises many. Our interest in it stems from analogous surprises arising frequently in computer security: attackers can often solve problems more efficiently than expected (e.g., arranging hash function collisions as in property H3 above). The key point is that the "collision" here is not for one pre-specified day (e.g., *your* birthday); any matching pair will do, and as n increases, the number of pairs of people is $C(n,2) = n(n-1)/2$, so the number of pairs of days grows as n^2. From this it is not surprising that further analysis shows that (here with $m = 365$) a collision is expected when $n \approx \sqrt{m}$ (rather than $n \approx m$, as is a common first impression).

Digital signatures with hash functions. Most digital signature schemes are implemented using mathematical primitives that operate on fixed-size input blocks. Breaking a message into blocks of this size, and signing individual pieces, is inefficient. Thus commonly in practice, to sign a message m, a hash $h = H(m)$ is first computed and h is signed instead. The details of the hash function H to be used are necessary to complete the signature algorithm specification, as altering these details alters signatures (and their validity). Here, H should be collision resistant (H3). Figure 2.10 illustrates the process.

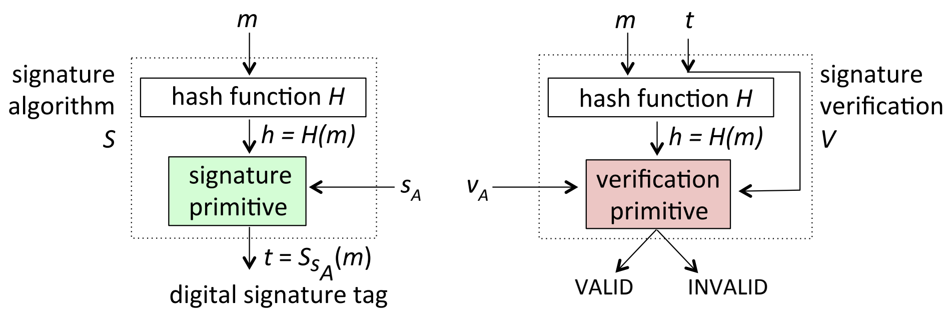

Figure 2.10: Signature algorithm with hashing details. The process first hashes message m to $H(m)$, and then applies the core signing algorithm to the fixed-length hash, not m itself. Signature verification requires the entire message m as input, likewise hashes it to $H(m)$, and then checks if an alleged signature tag t for that m is VALID or INVALID (e.g., returning boolean values TRUE or FALSE). s_A and v_A are Alice's signature private key and signature verification public key, respectively. Compare to Figure 2.8.

‡**Exercise** (Hash properties for signatures). For a hash function H used in a digital signature, outline distinct attacks that can be stopped by hash properties (H2) and (H3).

‡**Exercise** (Three methods for data integrity). Outline three methods, involving different cryptographic primitives, for providing data integrity on a digital file f.

‡**Exercise** (Precomputation attacks on hash functions). The definition of the one-way property (H1) has the curious qualifying phrase "for essentially all". Explain why this qualification is necessary (hint: [22, p.337]).

‡**Exercise** (Merkle hash trees). Explain what a *Merkle hash tree* is, and how BitTorrent uses this hash function construction to efficiently verify the integrity of data pieces in peer-to-peer downloads. (Hint: see the Wikipedia entry on "torrent file".)

‡**Exercise** (biometrics and *fuzzy commitment*). For password-based authentication, to avoid administrators having access to cleartext user passwords w, systems stores not w but its hash $H(w)$ as noted earlier. For biometric authentication, by combining error-correcting codes and one-way hash functions, explain how a *biometric template* can similarly be stored in protected form—and why error correction is needed. (Hint: [13].)

2.6 Message authentication (data origin authentication)

Message authentication is the service of assuring the integrity of data (i.e., that it has not been altered) and the identity of the party that originated the data, i.e., *data origin authentication*. This is done by sending a special data value, or tag, called a *message authentication code* (*MAC*), along with a message. The algorithm computing the tag, i.e., the MAC function, is a special type of hash function whose output depends on not only an input message but also on a secret number (secret key). The origin assurance derives from the assumption that the key is known only to the originator who computes the tag, and any party they share it with to allow tag verification; thus the recipient assumes that the originator is a party having access to, or control of, this MAC key.

If Alice sends a message and matching MAC tag to Bob (with whom she shares the MAC key), then he can verify the MAC tag to confirm integrity and data origin. Since the key is shared, the tag could also have been created by Bob. Between Alice and Bob, they know who originated the message, but if Alice denies being the originator, a third party may be unable to sort out the truth. Thus, MACs lack the property of *non-repudiation*, i.e., do not produce evidence countering false denial of previous actions (repudiation). Public-key signatures provide both data origin authentication and non-repudiation.

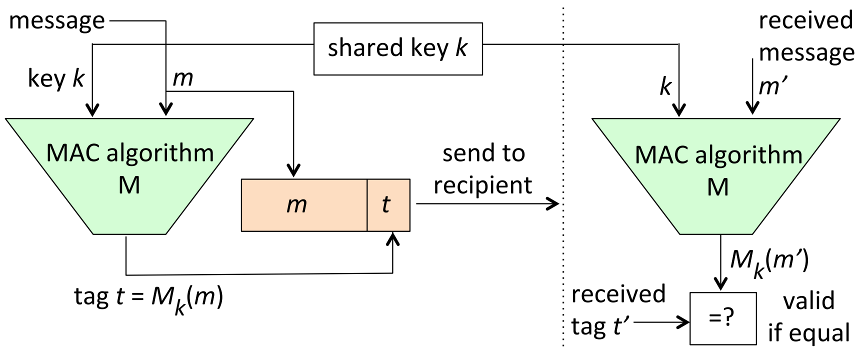

Figure 2.11: Message authentication code (MAC) generation and verification. As opposed to unkeyed hash functions, MAC algorithms take as input a secret (symmetric key) k, as well as an arbitrary-length message m. By design, with high probability, an adversary not knowing k will be unable to forge a correct tag t for a given message m; and be unable to generate any new pair (m, t) of matching message and tag (for an unknown k in use).

MAC DETAILS. Let M denote a MAC algorithm and k a shared MAC key. If Alice wishes to send to Bob a message m and corresponding MAC tag, she computes $t = M_k(m)$ and sends (m,t). Let (m',t') denote the pair actually received by Bob (allowing that the legitimate message might be modified en route, e.g., by an attacker). Using his own copy of k and the received message, Bob computes $M_k(m')$ and checks that it matches t'. Beyond this basic check, for many applications further means should ensure "freshness"—i.e., that (m',t') is not simply a replay of an earlier legitimate message. See Figure 2.11.

Example *(MAC examples).* An example MAC algorithm based on block ciphers is CMAC (Section 2.9). In contrast, HMAC gives a general contruction employing a generic hash function H such as those in Table 2.1, leading to names of form HMAC-H (e.g., H can be SHA-1, or variants of SHA-2 and SHA-3). Other MAC algorithms as noted in Table 2.2 are Poly1305-AES-MAC and those in AEAD combinations in that table.

‡**Example** *(CBC-MAC).* From the CBC mode of operation, we can immediately describe a MAC algorithm called CBC-MAC, to convey how a MAC may be built using a block cipher.[9] To avoid discussion of padding details, assume $m = m_1 m_2 \cdots m_t$ is to be authenticated, with blocks m_i of bitlength n, matching the cipher blocklength; the MAC key is k. Proceed as if carrying out encryption in CBC-mode (Figure 2.4) with IV = 0; keep only the final ciphertext block c_t, and use it as the MAC tag t.

‡**Exercise** (HMAC). Explain the general construction by which HMAC converts an unkeyed hash function into a MAC (hint: [17]).

‡**Exercise** (MAC truncation). The bitlength of a MAC tag varies by algorithm; for those built from hash functions or block ciphers, the default length is that output by the underlying function. Some standards truncate the tag somewhat, for technical reasons. Give security arguments both for, and against, truncating MAC outputs (hint: [34, 17]).

Exercise (Data origin authentication and integrity). Is it possible to provide data origin authentication, without data integrity? Why or why not?

2.7 Authenticated encryption and further modes of operation

Having now explained MACs, we proceed to discuss how they are commonly combined with symmetric-key encryption, and then consider several additional modes of operation of symmetric-key ciphers.

AUTHENTICATED ENCRYPTION. Encryption, when stated as a requirement, usually implies encryption with guaranteed integrity, i.e., the combination of encryption and data origin authentication. This allows detection of unauthorized ciphertext manipulation, including alteration and message forgery. The combined functionality, called *authenticated encryption* (AE), can be achieved by using a block cipher for encryption, and a separate MAC algorithm for authentication; this is called *generic composition* (Exercise below). However, a different approach, preferred for technical reasons, is to use a custom-built algorithm which does both. Such *integrated* AE algorithms are designed to allow safe use of a single symmetric key for both functions (whereas using one crypto key for two purposes is generally discouraged as bad practice[10]). Integrated AE algorithms are identified by naming a block cipher and an AE family—see Table 2.2.

AUTHENTICATED ENCRYPTION WITH ASSOCIATED DATA (AEAD). In practice, the following situation is common: message data is to be encrypted, and accompanying data should be authenticated (e.g., to detect any tampering) but not encrypted—e.g.,

[9]In practice, CMAC is recommended over CBC-MAC (see endnotes in Section 2.9).

[10]An example of what can go wrong is relatively easy to follow [22, p.367, Example 9.88].

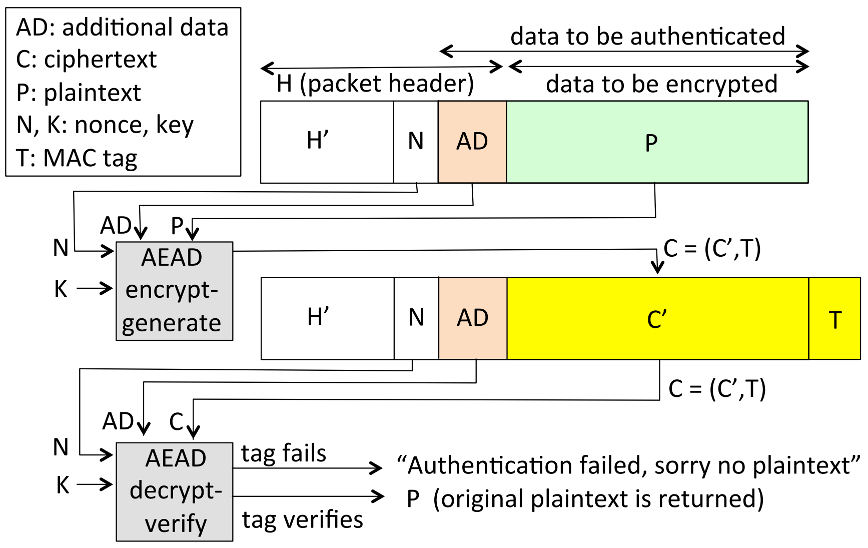

Figure 2.12: Authenticated encryption with associated data (AEAD). If the MAC tag T is 128 bits, then the ciphertext C' is 128 bits longer than the plaintext. A protocol may pre-allocate a field in sub-header H' for tag T. AEAD functionality may be provided by generic composition, e.g., generating C' using a block cipher in CBC mode, and T by an algorithm like CMAC (Section 2.9) or HMAC-SHA-2. The authenticated data (AD) need not be physically adjacent to the plaintext as shown (provided that logically, they are covered by MAC integrity in a fixed order). The *nonce* N, e.g., 96 bits in this application, is a number used only once for a given key K; re-use puts confidentiality at risk. If P is empty, the AEAD algorithm is essentially a MAC algorithm.

packet payload data must be encrypted, but header fields containing protocol or routing information may be needed by (visible to) intermediate networking nodes. Rather than use a separate MAC to detect integrity violations of such information, a special category of AE algorithms, called *authenticated encryption with associated data* (AEAD) algorithms, accommodate such additional data (AD) as shown in Figure 2.12.

‡CCM MODE OF OPERATION. An AEAD method called *Counter mode with CBC-MAC* (CCM) combines the CTR mode of operation (Figure 2.5) for encryption—in essence a stream cipher—with CBC-MAC (above) for authentication. The underlying block cipher used is commonly AES. Use in practice requires agreement on application-specific details for security and interoperability, e.g., input formatting and MAC tag post-processing (reducing its length in some cases); Sec. 2.9 gives references to CCM-related standards.

‡CHACHA20 AND POLY1305. ChaCha20 is a stream cipher involving 20 rounds of an underlying cipher, ChaCha. It was created by University of Illinois at Chicago professor Dan Bernstein, who also created the Poly1305 MAC algorithm. Both were designed to deliver high security with improved performance over alternatives that make

heavy use of AES (e.g., CCM above), for environments that lack AES hardware support. Poly1305 MAC requires a 128-bit key for an underlying block cipher (AES is suitable, or an alternate cipher with 128-bit key and 128-bit blocklength). For an AEAD algorithm, ChaCha20 is paired with Poly1305 as listed in Table 2.2.

Example *(Symmetric algorithms and parameters).* Table 2.2 gives examples of well-known symmetric-key algorithms with blocklengths, keylengths, and related details.

Name of cipher, MAC or AEAD combination	**Cipher specs**		**Other specs, notes**			**Cipher details**
	n (bits)	y (bits)	mode	v	t	
AES-128, AES-192, AES-256	128	128-256	–	–	–	block
ChaCha20	–	256	–	96	–	stream
Poly1305-AES-MAC	128	256	custom	96	128	AES-128
AEAD_ChaCha20_Poly1305	128	256	custom	96	128	ChaCha
AEAD_AES_128_CCM	128	128	CCM	96	128†	AES-128
AEAD_AES_256_CCM		256				AES-256
AEAD_AES_128_GCM	128	128	GCM	96	128†	AES-128
AEAD_AES_256_GCM		256				AES-256
DES	64	56	core of triple-DES			block
triple-DES (3-key)	64	3x56	NIST SP 800-67r2			block
RC4	–	–	legacy use			stream

Table 2.2: Common ciphers, AEAD algorithms, and example parameters. Blocklength n, keylength y, nonce/IV bitlength v, MAC tag bitlength t (†may be reduced by post-processing). Triple-DES involves three rounds of DES under three distinct keys, and remains of interest for legacy reasons. GCM is Galois/Counter Mode (Section 2.9).

‡**Exercise** (Authenticated encryption: generic composition). To implement authenticated encryption by serially combining of a block cipher and a MAC algorithm, three options are: 1) MAC-plaintext-then-encrypt (MAC the plaintext, append the MAC tag to the plaintext, then encrypt both); 2) MAC-ciphertext-after-encrypt: (encrypt the plaintext, MAC the resulting ciphertext, then append the MAC tag); and 3) encrypt-and-MAC-plaintext (the plaintext is input to each function, and the MAC tag is appended to the ciphertext). Are all of these options secure? (Hint: [1, 16, 4], and [37, Fig.2].)

2.8 Certificates, elliptic curves, and equivalent keylengths

CERTIFICATES. A *digital certificate* is a data structure whose primary fields are a subject name, a public key asserted to belong to that subject, and a digital signature (covering these and other fields) by a third party called a *certification authority* (CA). The intent is that the signature conveys the CA's attestation that it has verified that the named subject is the legitimate party associated with that public key, thus binding subject and public key.

Parties that rely on the certificate (*relying parties*) require an authentic copy of the CA's verification public key to verify the CA's signature, and thus the certificate's integrity.

CERTIFICATION AUTHORITIES. The CA's role is critical for trustworthy certificates. Before signing a certificate, the CA is expected to carry out appropriate due diligence to confirm the identity of the named subject, and their association with the public key. For example, to obtain evidence of control of the corresponding private key, the CA may send the subject a challenge message whose correct response requires use of that private key (without disclosing it); the CA uses the purportedly corresponding public key in creating the challenge, or verifying the response.

CERTIFICATE FIELDS. Other fields in a certificate include: a serial number to uniquely identify the certificate, expiry date, identity information for the CA, algorithm identifiers (for the embedded public-key, and the CA's signature), and revocation information. The latter relates to the possibility that a certificate's validity, which by default continues until the expiry date, may need to be terminated earlier (e.g., if the private key is reported compromised, or the named subject ceases to continue in the role for which the public key was certified). The revocation information indicates how relying parties can get further information, e.g., a signed list of *revoked certificates*, or the URL of a trusted site to contact for a real-time status check of a certificate's validity. Digital certificates allow relying parties to gain trust in the public keys of many other parties, through pre-existing trust in the public key of a signing CA. Trust in one key thus translates into trust in many. Certificates and CAs are discussed in greater detail in Chapter 8.

NIST-RECOMMENDED KEYLENGTHS. For public U.S. government use, NIST recommends (as of November 2015) at least 112 bits of "security strength" for symmetric-key encryption and related digital signature applications. Here "strength" is not necessarily the raw symmetric-key bitlength, but an estimate of security based on best-known attacks (e.g., triple-DES has three 56-bits keys, but its estimated strength is only 112 bits). To avoid obvious "weak-link" targets, multiple algorithms used in conjuction should be of comparable strength,[11] including when symmetric- and public-key algorithms are used together. Giving "security strength" estimates for public-key algorithms requires a few words. The most effective attacks against strong symmetric algorithms like AES are exhaustive search attacks on their key-space—so a 128-bit AES key is expected to be found after searching 2^{127} keys, for a security strength of 127 bits (essentially 128). In contrast, for public-key cryptosystems based on RSA and Diffie-Hellman (DH), the best attacks do not require exhaustive search over private key spaces, but instead faster number-theoretic computations involving *integer factorization* and computing *discrete logarithms*. This is the reason that, for comparable security, keys for RSA and DH must be much larger than AES keys. Table 2.3 gives rough estimates for comparable security.

ELLIPTIC CURVE PUBLIC KEY SYSTEMS. Public-key systems are most easily taught using implementations over number systems that students are already somewhat familiar with, e.g., arithmetic modulo $n = pq$ for RSA (above), and (later in Chapter 4) Diffie-Hellman (DH) over integers modulo a large prime p. By their underlying mathe-

[11]This follows the principle of DEFENCE-IN-DEPTH P13 (Chapter 1).

Symmetric key security strength	**RSA** modulus	**DH** modulus	private key	**ECC**
112 (triple-DES)	2048	2048	224	224-255
128 (AES)	3072	3072	256	256-383

Table 2.3: Recommended keylengths for comparable algorithm strengths. Numbers denote parameter bitlengths. A symmetric-key of 128 bits corresponds to the lowest of three keylengths supported by AES. For RSA and DH, the modulus implies the size of the public key. RSA entries are for encryption, signatures, and key agreement/key transport (Chapter 4). These are recommended pairings, rather than exact security equivalents.

matical structures, RSA and DH are respectively classified as *integer factorization cryptography* (IFC) and *finite field cryptography* (FFC). Public-key functionality—encryption, digital signatures, and key agreement—can analogously be implemented using operations over sets of elements defined by points on an *elliptic curve*. Such *elliptic curve cryptography* (ECC) implementations offer as a main advantage computational and storage efficiencies due to smaller key sizes (Table 2.3). In certain situations ECC also brings disadvantages. To mention one, in many RSA implementations the public-key operation is relatively inexpensive compared to the private key operation (as for technical reasons, short public exponents can be used); the reverse is true for ECC, a drawback in certificate-based infrastructures where signature verification (using the public key) is far more frequent than signing (using the private key). ECC involves more complex mathematics, but this is easily overcome by the availability of standard toolkits and libraries. In this book, we use RSA and Diffie-Hellman examples that do not involve ECC.

2.9 ‡Endnotes and further reading

The classic treatment of cryptography through the ages is Kahn [14]. Diffie [7] gives a more academic introduction. Diffie and Hellman [8] introduced public-key cryptography, as well as DH key exchange (Chapter 4). RSA encryption and signatures are due to Rivest, Shamir and Adleman [36]. Boneh [5, 6] surveys attacks on RSA, and explains the difference between theory and practice in implementing public key algorithms. For extensive background in applied cryptography, see Menezes [22] (book, free online). Its section 7.3 (*Classical ciphers and historical development*) reviews "toy ciphers" and how simple, elegant attacks defeat historical *transposition ciphers* and *polyalphabetic substitution*, e.g., using the *method of Kasiski* and related *index of coincidence*. Such attacks clarify how the one-time pad fails if the key is re-used (it is not a two-time pad). Other recommended books include Ferguson [9] for practical cryptography, and Menezes [21, 10] for elliptic curve cryptography.

For triple-DES and its status circa 2017, see NIST [31]; newer alternatives are preferred. CMAC is a block cipher construction improving on CBC-MAC; NIST 800-38B

[24] gives details and supporting literature (approving it for use with AES, and also 3-key triple-DES); see also RFC 4493 for AES-CMAC. See RFC 7539 [23] for the ChaCha20 stream cipher and Poly1305 MAC, their combined AEAD algorithm due to Langley, and motivation (advantages over AES); the original proposals are by Bernstein [2, 3]. Use of HMAC [17, 28] with MD5, i.e., HMAC-MD5, is discouraged [39] towards ending MD5's ongoing use (especially for signature applications). The widely-implemented RC4 stream cipher is now precluded by TLS [32]. Digital signatures *with appendix*, per Section 2.4, require the message itself for signature verification. Digital signatures *with message recovery* [22, p.430] (suitable only for short messages) do not—the signature verification process recovers the original message (the tag conveys both signature and message), but whereas a hash function is not needed, a customized redundancy function is.

Preneel [33] undertook the first systematic study of cryptographic hash functions; Chapter 9 of Menezes [22] gives an early overview. For finding hash function collisions in practice, see van Oorschot [40]. For cryptanalysis of methods attempting to create MAC algorithms by concatenating secret keys into the input of unkeyed hash functions, see Preneel [35]. Rogaway [37] formalized AEAD (authenticated encryption with associated data); for interface definitions per Table 2.2, see RFC 5116 [19]. NIST-specified AEAD modes include CCM [25] (see Jonsson [12] for security analysis, and the original Whiting [41] proposal) and GCM [27] (see also McGrew [20]); the OCB mode of Rogaway [38] is faster than both (Krovetz [18] compares the three), but patent entanglement issues impaired adoption of it and several earlier methods. Table 2.3's recommended keylength pairings are from NIST [30].

References

[1] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT—Advances in Cryptology*, pages 531–545, 2000. Extended version in: J. Cryptology, 2008.

[2] D. J. Bernstein. ChaCha, a variant of Salsa20. 28 Jan 2008 manuscript; see also `https://cr.yp.to/chacha.html`.

[3] D. J. Bernstein. The Poly1305-AES Message-Authentication Code. In *Fast Software Encryption*, pages 32–49, 2005. See also `https://cr.yp.to/mac.html`.

[4] J. Black. Authenticated encryption. In *Encyclopedia of Cryptography and Security*. Springer (editor: Henk C.A. van Tilborg), 2005. Manuscript also online, dated 12 Nov 2003.

[5] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society (AMS)*, 46(2):203–213, 1999.

[6] D. Boneh, A. Joux, and P. Q. Nguyen. Why textbook ElGamal and RSA encryption are insecure. In *ASIACRYPT—Advances in Cryptology*, pages 30–43, 2000.

[7] W. Diffe and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, March 1979.

[8] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Info. Theory*, 22(6):644–654, 1976.

[9] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley Publishing, 2003.

[10] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography*. Springer, 2004.

[11] IEEE Computer Society. IEEE Std 1619-2007: Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. 18 April 2008. Defines the XTS-AES encryption mode, and is the base document for NIST SP 800-38E [29].

[12] J. Jonsson. On the security of CTR + CBC-MAC. In *SAC—Workshop on Selected Areas in Cryptography*, pages 76–93, 2002.

[13] A. Juels and M. Wattenberg. A fuzzy commitment scheme. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 28–36. ACM, 1999.

[14] D. Kahn. *The Codebreakers*. Macmillan Publishing, 1967.

[15] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 18–29. ACM, 1994.

[16] H. Krawczyk. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In *CRYPTO—Advances in Cryptology*, pages 310–331, 2001.

[17] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104: HMAC: Keyed-Hashing for Message Authentication, Feb. 1997. Informational; updated by RFC 6151 (March 2011).

[18] T. Krovetz and P. Rogaway. The software performance of authenticated-encryption modes. In *Fast Software Encryption*, pages 306–327, 2011.

[19] D. McGrew. RFC 5116: An Interface and Algorithms for Authenticated Encryption, Jan. 2008. Proposed Standard.

[20] D. A. McGrew and J. Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In *INDOCRYPT*, pages 343–355, 2004.

[21] A. Menezes. *Elliptic curve public key cryptosystems*. Springer, 1993.

[22] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Freely available online, `http://cacr.uwaterloo.ca/hac/`.

[23] Y. Nir and Langley. RFC 7539: ChaCha20 and Poly1305 for IETF Protocols, May 2015. Informational.

[24] NIST. Special Pub 800-38B: Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication. May 2005, with updates 6 Oct 2016.

[25] NIST. Special Pub 800-38C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. May 2004, with updates 20 Jul 2007.

[26] NIST. Special Pub 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques, Dec. 2001.

[27] NIST. Special Pub 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, Nov. 2007.

[28] NIST. FIPS 198-1: The Keyed-Hash Message Authentication Code (HMAC). National Inst. Standards and Tech., U.S. Dept. of Commerce, July 2008.

[29] NIST. Special Pub 800-38E: Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices. National Inst. Standards and Tech., U.S. Dept. of Commerce, Jan. 2010. This points to IEEE Std 1619-2007 [11] for technical details.

[30] NIST. Special Pub 800-57 Part 1 r4: Recommendation for Key Management (Part 1: General). National Inst. Standards and Tech., U.S. Dept. of Commerce, Jan 2016. (Revision 4).

[31] NIST. Special Pub 800-67 r2: Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher. National Inst. Standards and Tech., U.S. Dept. of Commerce, Nov 2017. (Revision 2).

[32] A. Popov. RFC 7465: Prohibiting RC4 Cipher Suites, Feb. 2015. Proposed Standard.

[33] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, Belgium, Jan. 2003.

[34] B. Preneel and P. C. van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In *CRYPTO—Advances in Cryptology*, pages 1–14, 1995.

[35] B. Preneel and P. C. van Oorschot. On the security of iterated message authentication codes. *IEEE Trans. Info. Theory*, 45(1):188–199, 1999.

[36] R. L. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[37] P. Rogaway. Authenticated-Encryption with Associated-Data. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 98–107, 2002.

[38] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: a block-cipher mode of operation for efficient authenticated encryption. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 196–205, 2001. Journal version in: ACM TISSEC, 2003.

[39] S. Turner and L. Chen. RFC 6151: Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms, Mar. 2011. Informational.

[40] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[41] D. Whiting, R. Housley, and N. Ferguson. RFC 3610: Counter with CBC-MAC (CCM), Sept. 2003. Informational RFC.