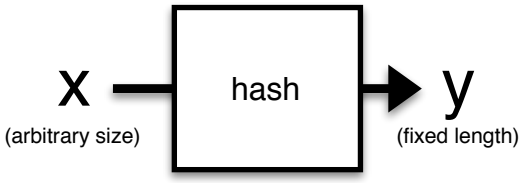
# Cryptographic Primitives

These primitives are algorithms that compute some useful cryptographic function. They should be treated as opaque boxes—-we won't care how they work, only that they satisfy the properties that we need from them.

Throughout these definitions, we refer to a task being EASY or HARD. EASY means that a function can be computed efficiently enough for whatever application we need. HARD means practically impossible on any useful human time scale. To be more specific, all of these cryptographic primitives can be defeated by *brute force*: I can always try to decrypt a message with every possible key until I guess the right key. Breaking a cipher is HARD if there is no trick or technique that is faster than brute force. We can control the difficulty of brute force by dictating the size of keys or outputs.

## Hashes

A hash function maps any input string to a fixed-length output, typically a few hundred bits in length. The purpose of a cryptographic hash is to conceal the input value even if someone knows the output.



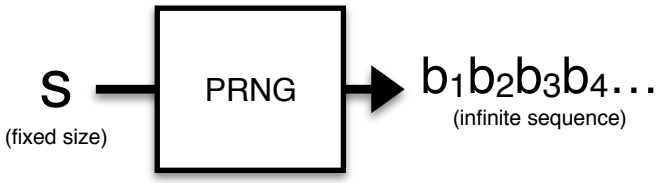

Under the *random oracle model*, a hash function appears to assign every input to a string generated by flipping a bunch of fair coins. Under this model, the output does not depend on the content of the input, so it can not be analyzed to learn anything about the input. This also means that two similar inputs should not hash to similar outputs, since each is assigned a separate random string. A real-world hash function can not work in this way, but rather scrambles an input so well that the result is indistinguishable from a random oracle.

**Security properties**

0. The function hash[x] is EASY to compute
1. (Weak collision-free property) given y=hash[x], it is HARD to find x, or to find any other input that hashes to y.
2. (Strong collision-free property) in general it is HARD to find any two inputs that hash to the same value.

## Pseudo-random number generators (PRNGs)

A PRNG is a function that transforms a "seed" value into an inexhaustible supply of random-looking bits. In terms of the input and output length it is somewhat dual to the concept of a hash; like a hash, it has the goal of concealing the input even if the output is known.

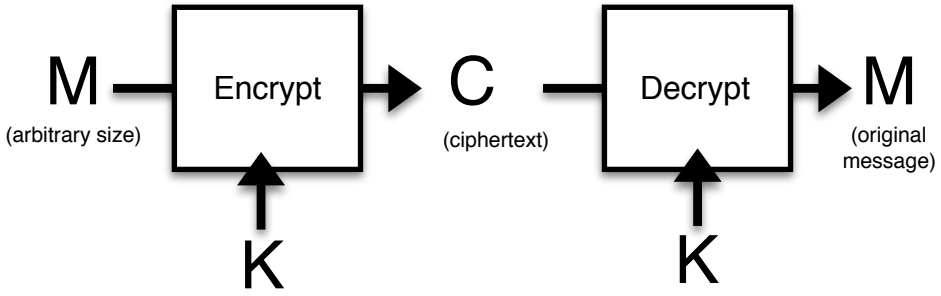



**Security properties**

1. Given a seed s, bits of PRNG(s) are EASY to compute.
2. (Weak property): given any output bits of PRNG(s), it is HARD to find the seed s.
3. (Strong property): given any output bits of PRNG(s), it is HARD to predict or estimate any other bits of the PRNG output sequence. To an adversary lacking the seed, every subsequent bit should be as unpredictable as a fair coin flip.

Essentially, the output of a PRNG should perfectly simulate flipping a coin, with the one exception that we can re-run a PRNG with the same seed and get the same random-looking sequence. Thus two people sharing the same secret seed can run a PRNG function, each person generating "the same random."

## Symmetric encryption




An encryption system consists of two functions* we will call Encrypt() and Decrypt(), each of which requires both a message and a key to drive the encryption and decryption. These two functions have the property of being inverses, as diagrammed.

**Security properties (symmetric ciphers):**

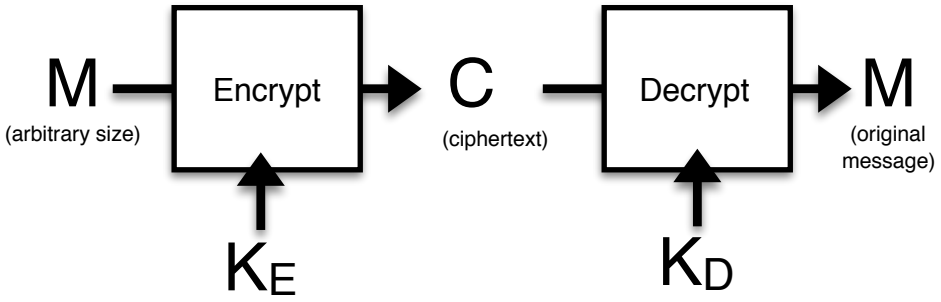
1. Given a key K, it is EASY to compute C=Encrypt(M,K) and M=Decrypt(C,K).
2. Without the key K, given C=Encrypt(M,K), it is HARD to determine any part of M.
3. Without the key K, given M, it is HARD to compute an encryption of M.
4. Without the key K, given both M and C=Encrypt(M,K), it is HARD to determine K.

The last requirement is necessary because an adversary often knows some or all of the content of one encrypted message, and should not be able to read any other encrypted messages.

{*}: It is common for encryption methods to include some randomness, so that the same input message with the same key encrypts to a different ciphertext every time it is encrypted. This is a desirable property because it prevents an adversary from knowing if the same message was sent twice. Mathematically, this means that Encrypt(M,K) is not a function, because a single input can map to multiple outputs; but even in this case Decrypt(C,K) is a function.

## Asymmetric encryption

This is the same as symmetric encryption, except there is a key pair, with an encryption key corresponding to a decryption key. These are designed so that knowing the encryption key doesn't tell you the decryption key.




**Security properties (asymmetric ciphers):**

1. Given encryption key $K_E$, it is EASY to compute C=Encrypt(M,$K_E$).
2. Given decryption key $K_D$, it is EASY to compute M=Decrypt(C,$K_D$).

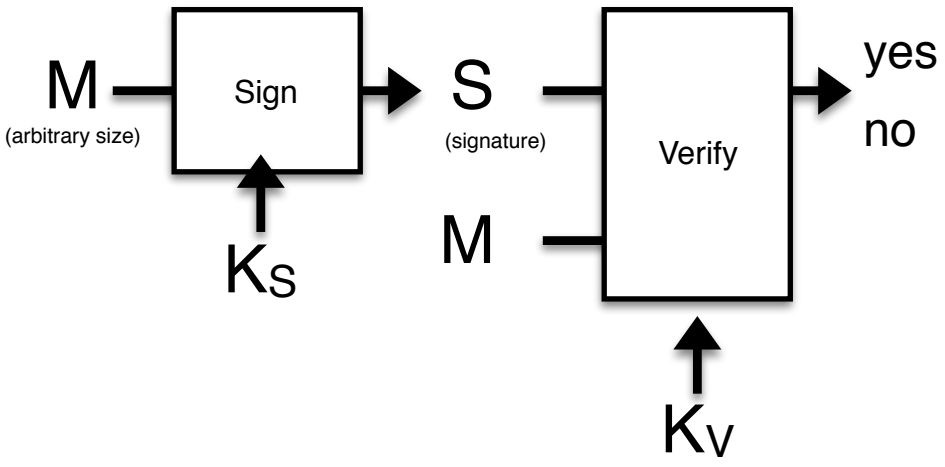
3. Without the key $K_D$, given C=Encrypt(M,$K_E$), it is HARD to determine any part of M, even if someone knows $K_E$.

Property 3 implies that someone who has $K_E$ cannot determine $K_D$. This allows Alice to produce a key pair ($K_E$, $K_D$) and publish $K_E$. Thus anyone can encrypt a message for Alice, that only Alice can decrypt, analogous to a drop-box that only Alice can open. For this reason, asymmetric encryption is sometimes called *public-key encryption*.

Asymmetric encryption solves a long-standing problem in cryptography, that of arranging for two people to share a secret key for communication. There is no need for such an arrangement if we can simply publish an encryption key. However, in practice these algorithms are slow enough, and symmetric algorithms are fast enough, that we rarely use asymmetric cryptography to send actual messages. Instead, we will use an asymmetric encryption system to send a *session key*, a temporary key for encrypting with a symmetric cipher. Thus instead of sending C=Encrypt(M,$K_E$) we will send the pair {$C_1$=Encrypt$_{Symmetric}$(M,K),$C_2$=Encrypt$_{Asymmetric}$(K,$K_E$)}.

# Signature schemes

A signature scheme is very similar to an asymmetric encryption system, so similar that we can implement a signature by using an asymmetric system with encryption and decryption reversed.




Here, a message is digested into a signature using a secret signing key $K_S$, and the digest can be verified by anyone using a public verification key $K_V$ corresponding to the signing key—-the signing key and verification key form a key pair ($K_S$,$K_V$).

**Security properties:**

1. Given signing key $K_S$, it is EASY to compute S=Sign(M,$K_S$).
2. Given verification key $K_V$, M and S, it is EASY to compute Verify(M,S,$K_V$).

3. Without the key $K_S$, it is HARD to determine $K_S$, or to compute any signatures with $K_S$, even if one has $K_V$, or samples of messages and their signatures.

Given an asymmetric encryption system, if encryption and decryption are reversible—-that is, if Decrypt( Encrypt( M, $K_E$ ) $K_D$ ) = M and Encrypt( Decrypt(C, $K_D$) $K_E$) = C—-then we can use it to sign messages as follows. To sign a message, use the secret decryption key as the signing key and compute S=Decrypt( hash[M], $K_D$ ). The hash is present for purposes of efficiency, and if the hash is secure then we can use hash[M] as a surrogate for M. To verify a signature, we use the public encryption key $K_E$ as the verification key; given the message M and signature S we can verify that Encrypt(S,$K_E$) = hash[M].

What does a signature mean? A signature of a message M is a string that only the holder of $K_S$ can generate. This is taken as proof that message M came from that key holder. This can be used to prevent impersonation, to prevent tampering with a message, and like a physical signature to show some kind of commitment, such as signing a contract.

Note that the signature scheme does not make the message M secret or unreadable. It provides *integrity*, rather than confidentiality.