Lecture 9:

Network Security

Katerina Argyraki, EPFL

Security properties

- Confidentiality
  - only the sender and the receiver understand the contents of the message
- Authenticity
  - the message is from whom it claims to be
- Integrity
  - the message was not changed along the way

We will now discuss how network protocols can provide three basic security properties: confidentiality, authenticity, and integrity.

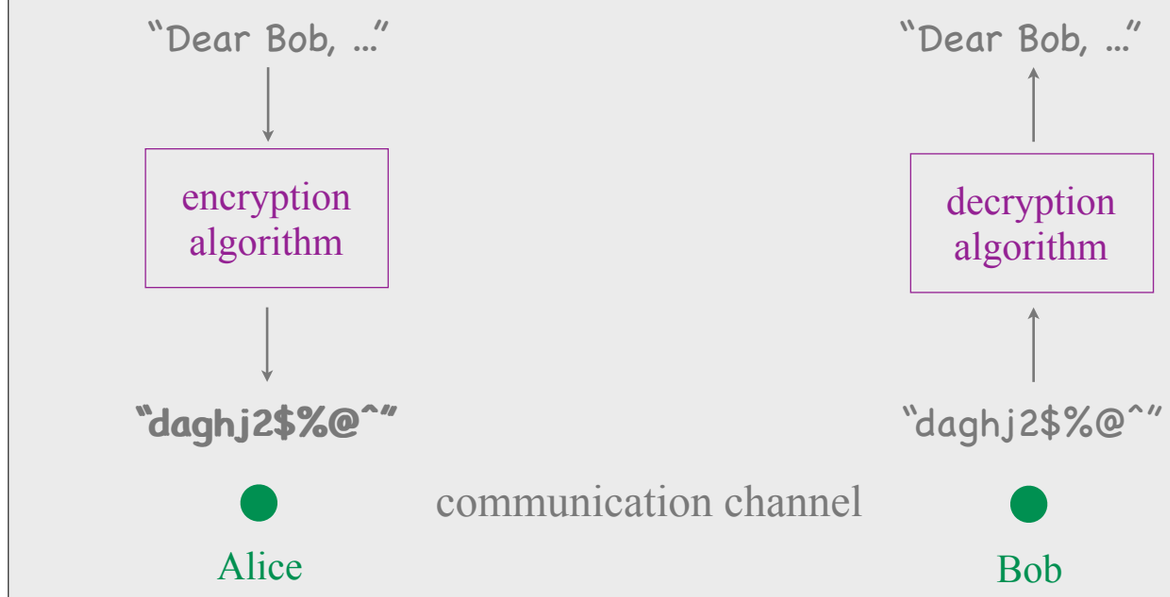
Outline

- Building blocks
- Providing security properties
- Securing Internet protocols
- Operational security

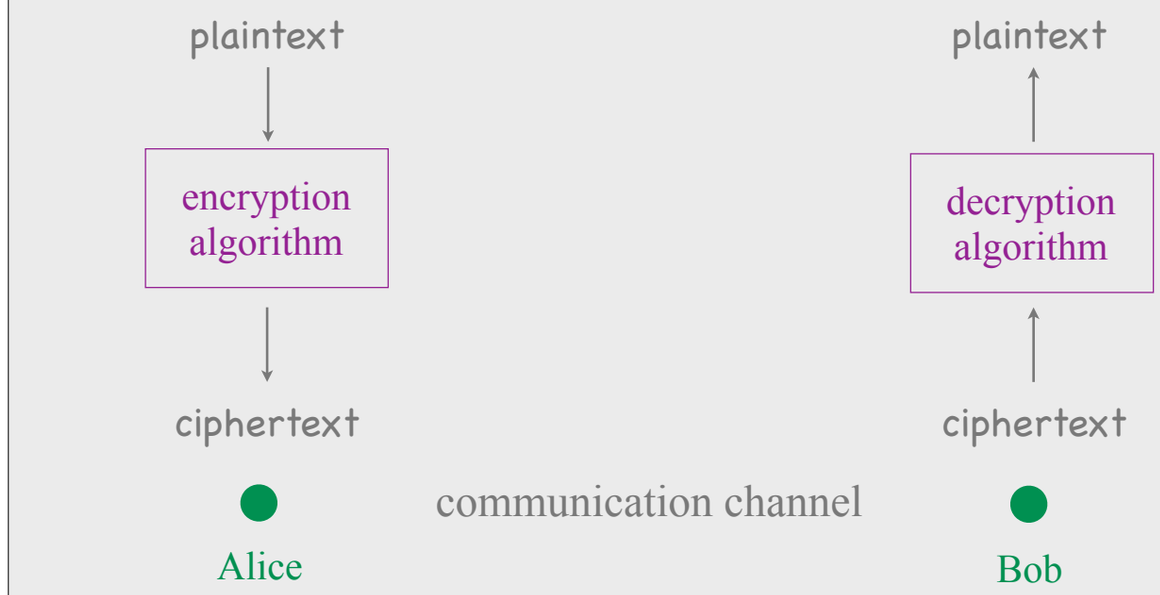
We will first discuss building blocks, then how to use the building blocks in general to provide the target security properties, and then the same thing (how to use the building blocks to provide the target security properties) in the context of very specific network protocols. We will close with a brief discussion on operational network security.

Outline

- Building blocks
- Providing security properties
- Securing Internet protocols
- Operational security



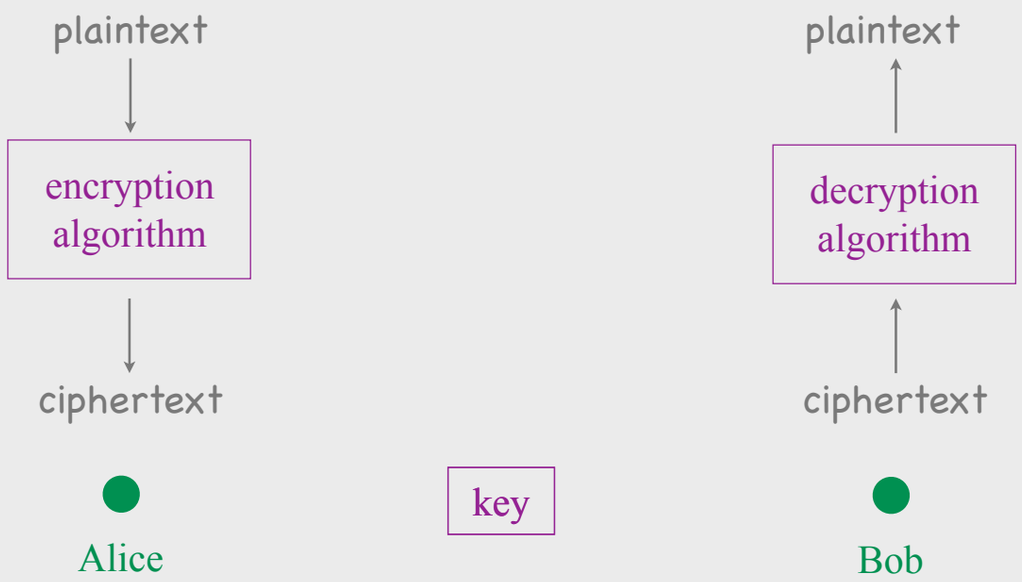
The most basic security building block is encryption and decryption:
Consider Alice and Bob, and a communication channel between them.
Alice has a message to send to Bob.
She first provides the message as input to an encryption algorithm; the algorithm outputs a “scrambled” version of the original message, and Alice sends it to Bob over the communication channel.
At the other end of the channel, Bob receives the message and provides it as input to a decryption algorithm; the algorithm outputs the original message.



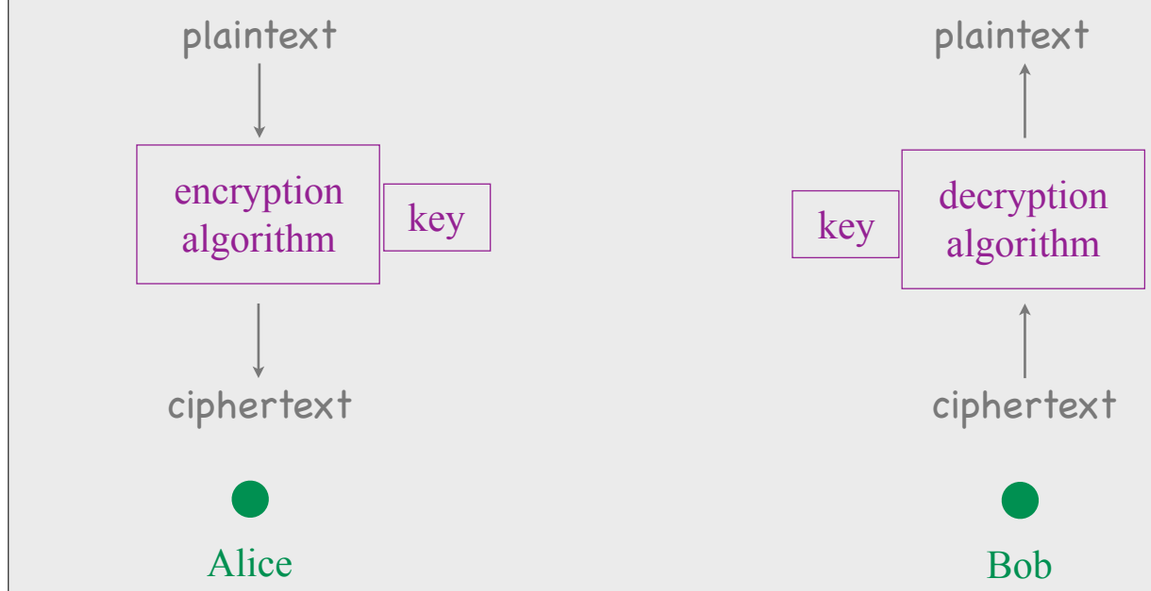
We use the term “plaintext” to refer to Alice’s original message,
and the term “ciphertext” to refer to the scrambled version of the message that is produced by the encryption algorithm.

Encryption & decryption

- Encryption: plaintext in, ciphertext out
- Decryption: ciphertext in, plaintext out
- Ciphertext: ideally, should reveal no information about the plaintext



In symmetric-key cryptography, Alice and Bob share a secret key that is known only to the two of them. This same key is provided as input both to the encryption and the decryption algorithms.



The basic property that needs to hold in a symmetric-key crypto system is that: if we encrypt a plaintext with a key, then decrypt the resulting ciphertext with the same key, we get the plaintext.

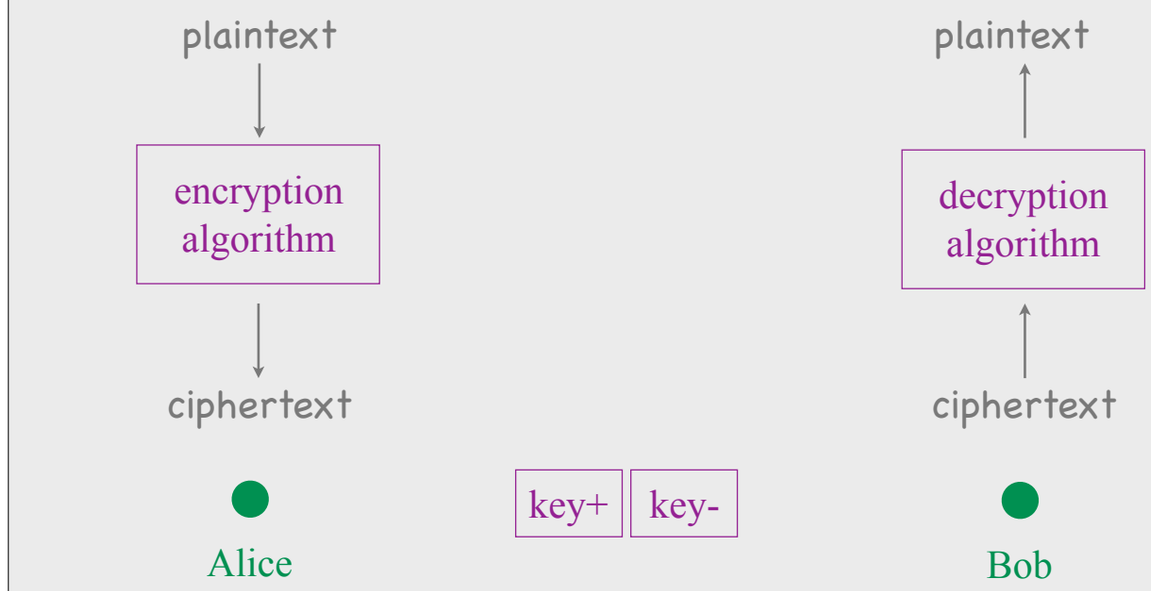
Symmetric key cryptography

- Alice and Bob share the same key
  - used both for the encryption and decryption algorithm
- Use key to "scramble" the plaintext
  - stream ciphers & block ciphers
  - RC4, AES, Blowfish

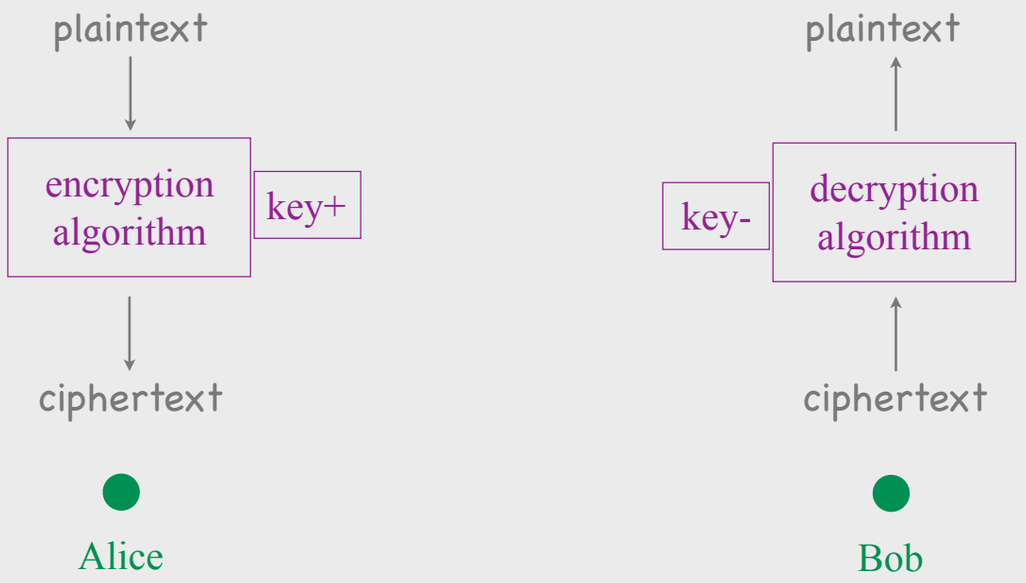
Symmetric key cryptography

- Challenge: how to share a key?
  - out of band
  - not always an option

A fundamental challenge in symmetric-key cryptography is how to share the secret key before the communication begins. Clearly, Alice and Bob cannot exchange the key over the communication channel, because any adversary sitting on that channel could observe the key.



In asymmetric-key cryptography, Alice and Bob use different keys: a public key (denoted by key+) and a private key (denoted by key-).



key-{ key+{ plaintext } } = plaintext

The two basic properties that need to hold in an asymmetric-key crypto system are:

- If we encrypt a plaintext with key+, then decrypt the resulting ciphertext with key-, we get the plaintext.
- If we encrypt a plaintext with key-, then decrypt the resulting ciphertext with key+, we get the plaintext.

These two properties require a very particular mathematical relationship between key+ and key-. So, a pair of public/private keys are always generated together — we obviously cannot use any two keys together as a public/private pair.

Asymmetric key cryptography

- Alice and Bob use different keys
  - public (key+) and private (key-) key
- There is a special relationship between them
  - key-{ key+{ plaintext } } = plaintext
  - key+{ key-{ plaintext } } = plaintext
  - RSA, DSA

Asymmetric key cryptography

- Public key is not secret
  - only private key is secret
  - enough to guarantee secrecy
- But you can't guess one from the other
  - Alice/Bob can share key+ with everyone
  - without revealing information about key-

An important feature of asymmetric key cryptography is that the public key is not a secret — it is OK if adversaries know it.

Also, one cannot guess (but with an insignificant probability) the private key from the public key.

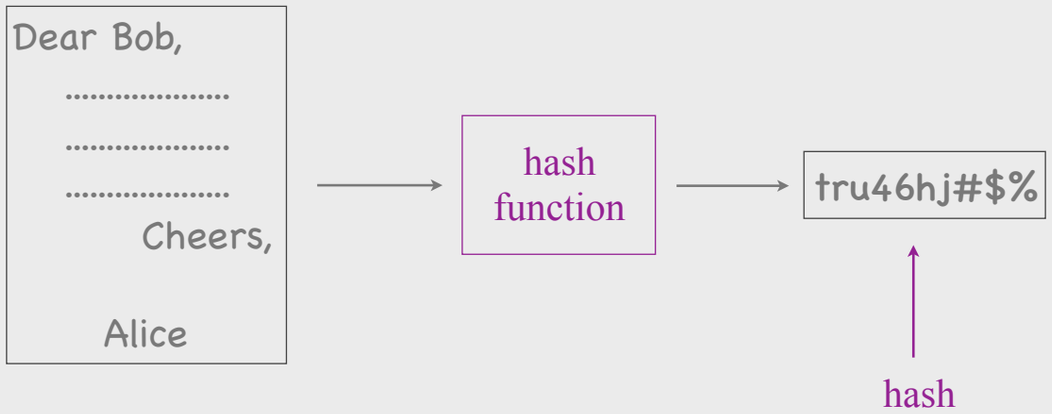
Asymmetric key cryptography

- Challenge: computationally expensive
  - sophisticated encryption/decryption algorithms based on number theory

A disadvantage of asymmetric-key cryptography is that it is more computationally expensive than symmetric-key cryptography, due to the complexity of its encryption/decryption algorithms.

Two approaches to crypto

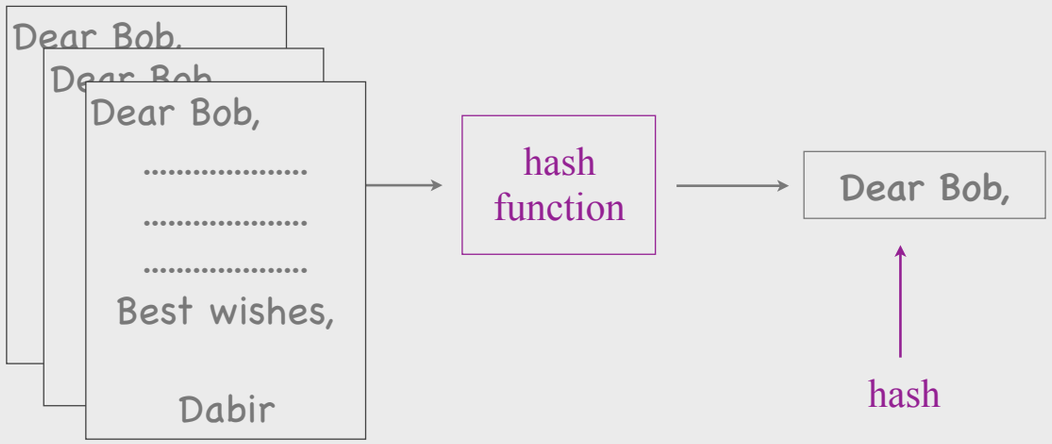
- Symmetric: faster but out-of-band key sharing
- Asymmetric: no out-of-band key sharing
  but slower



Another basic security building block is a cryptographic hash function.

Such a function takes as input a message and produces as output a piece of data (called a “hash”) that is smaller than the input and ideally reveals no information about the input.

When I say that a hash “reveals no information” about the input, I mean that, given the hash, any input is equally likely to have produced the hash.



Here is an example of a very bad hash function: the hash is equal to the first N bytes of the input.

This is bad for two (related) reasons:

- the hash provides significant information about the input
- and it is very easy to come up with several inputs that produce the same hash.

Cryptographic hash function

- Maps larger input space to smaller hash space
- Hash ideally reveals no information on input
- Should be hard to identify two inputs
  that lead to the same hash

How is hashing different from encryption?

A hash function and an encryption algorithm both take as input a plaintext and produce as output a random-looking piece of text that ideally provides no information on the plaintext. So, how are they different?

Building blocks

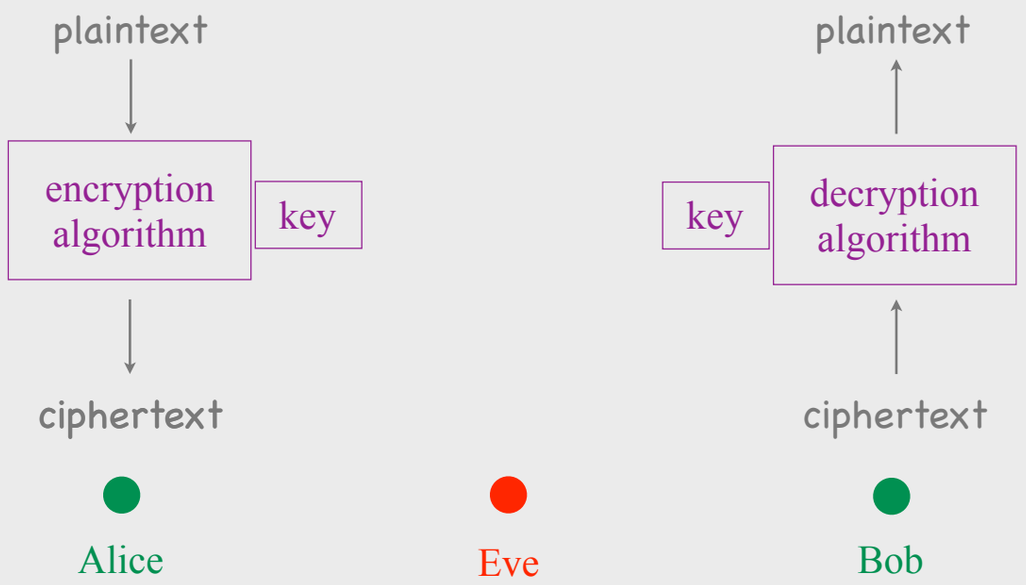
- Symmetric key encryption/decryption
  - Alice and Bob share the same secret key
  - challenge: exchanging the secret key
- Asymmetric key encryption/decryption
  - Alice and Bob use different keys
  - challenge: computationally more expensive
- Cryptographic hash function
  - produces a hash of the original message

Outline

- Building blocks
- Providing security properties
- Securing Internet protocols
- Operational security

Now let’s see how we can use these building blocks to provide security properties.

Providing confidentiality

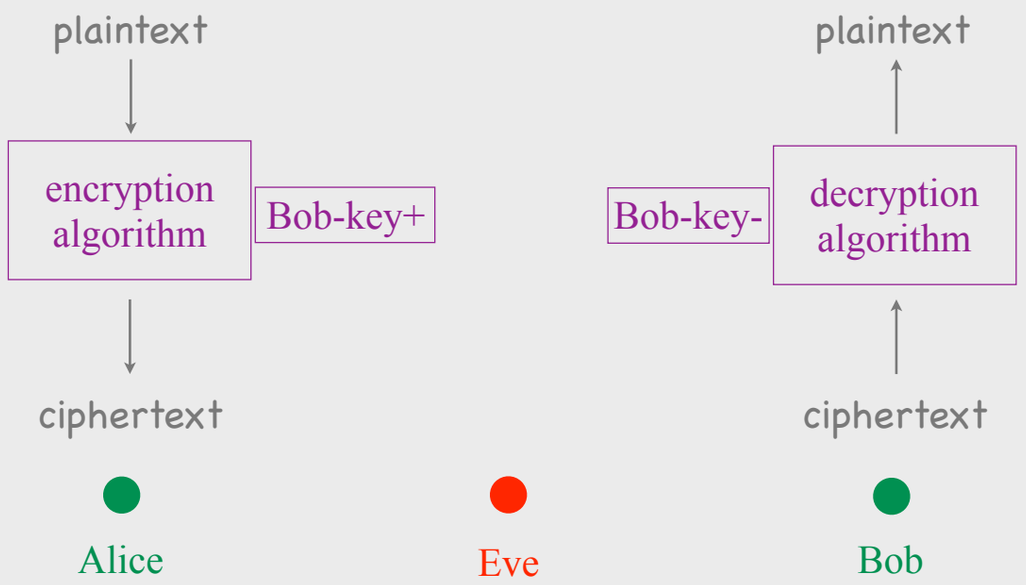


Suppose Alice wants to send a confidential message to Bob.

The adversary is Eve, who is sitting on the communication channel between Alice and Bob (meaning that she can intercept and read their messages, as well as send messages to them), and who wants to read Alice's message.

If Alice and Bob use symmetric-key crypto, Alice can encrypt her message with the key that she shares with Bob. Bob recovers Alice's original message by decrypting the ciphertext with the shared key.

Eve may read the ciphertext but cannot infer from it anything about the plaintext — as long as she does not know the shared key.



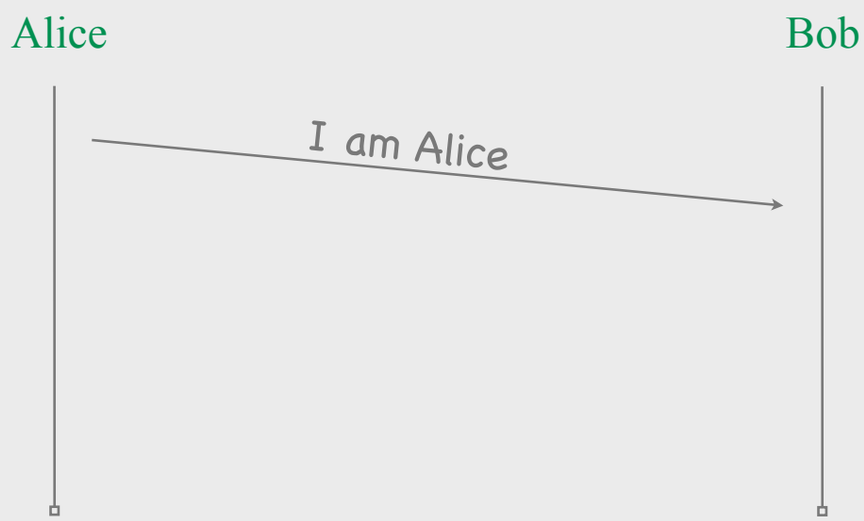
If Alice and Bob use asymmetric-key crypto, Alice can encrypt her message with Bob's public key (Bob-key+). Bob recovers Alice's original message by decrypting the cipheretxt with his private key (Bob-key-).

Again, Eve may read the ciphertext but cannot infer from it anything about the plaintext — as long as she does not know Bob's private key.

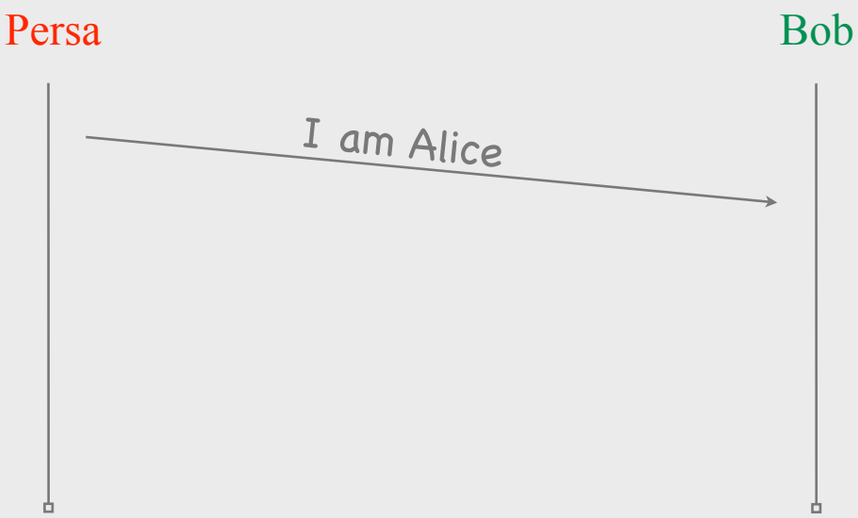
Providing confidentiality

- With symmetric key crypto
  - Alice encrypts message with shared key
  - only Bob can decrypt it (with shared key)
- With asymmetric key crypto
  - Alice encrypts message with Bob's public key
  - only Bob can decrypt it (with his private key)

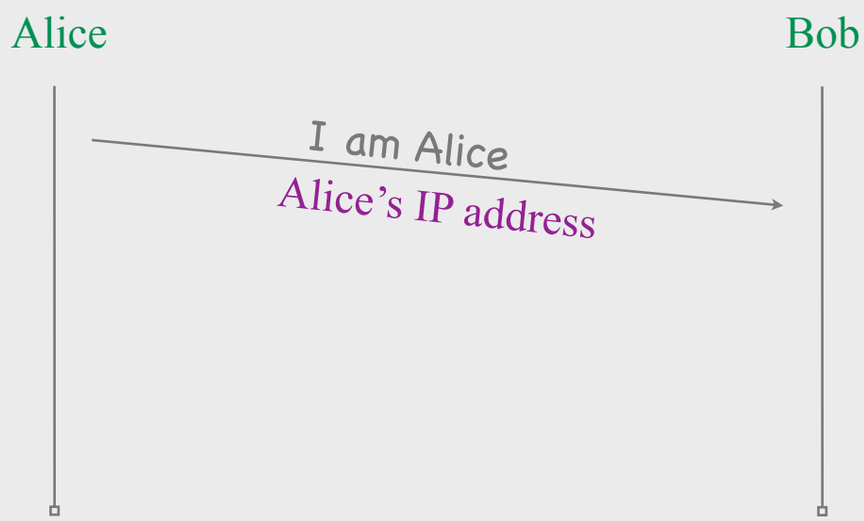
Providing authenticity



Suppose Alice wants to send a message to Bob and at the same time prove to Bob that it is she who sent the message.



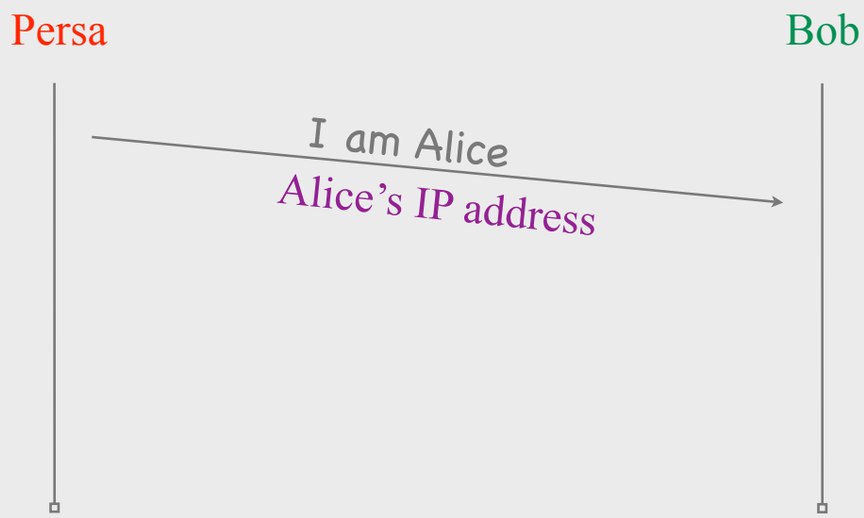
The adversary is Persa, who is sitting on the communication channel between Alice and Bob, and who may try to impersonate Alice, i.e., send a message to Bob and try to convince him that it is Alice who sent it.



One might think that Bob could rely on Alice's source IP address for authentication:

Suppose Bob knows Alice's true IP address.

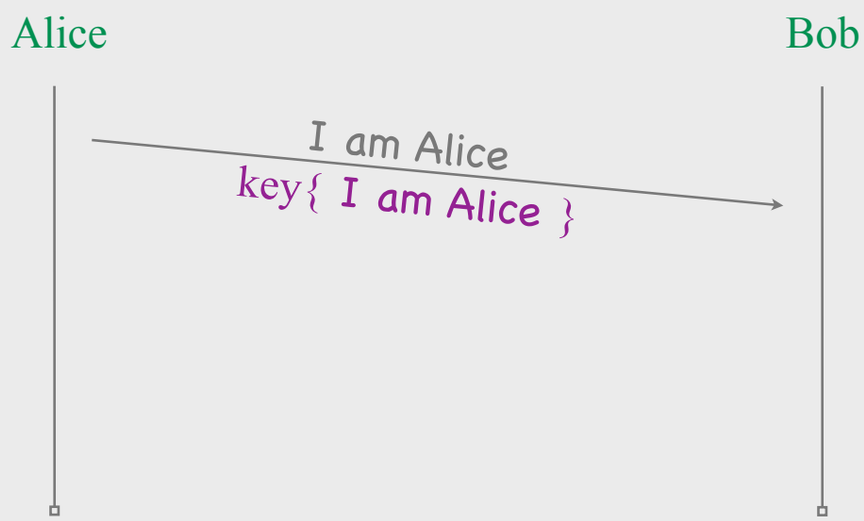
When the packet that contains Alice's message drive, Bob might check the source IP address in the packet's IP header and, if it is Alice's IP address, conclude that the message must be coming from Alice.



Unfortunately, this does not work.

Persa could "spoof" Alice's source IP address, i.e., create a message that claims to be coming from Alice and put it in a packet whose source IP address is Alice's IP address.

The current Internet architecture does not force an end-system to use its own IP address when sending out packets. A network administrator may ensure that all packets leaving her network carry correct IP addresses, but, when Bob receives a packet there is no way to check whether this packet is coming from such a well-behaved network.



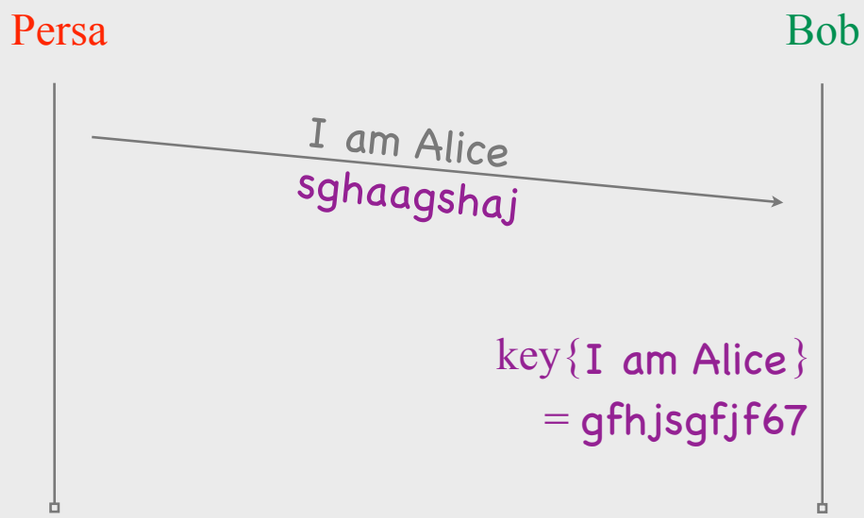
Suppose Alice and Bob share a secret key.

Alice sends a message that consists of a plaintext (I am Alice), followed by the corresponding ciphertext (key{I am Alice}).

This provides authenticity:

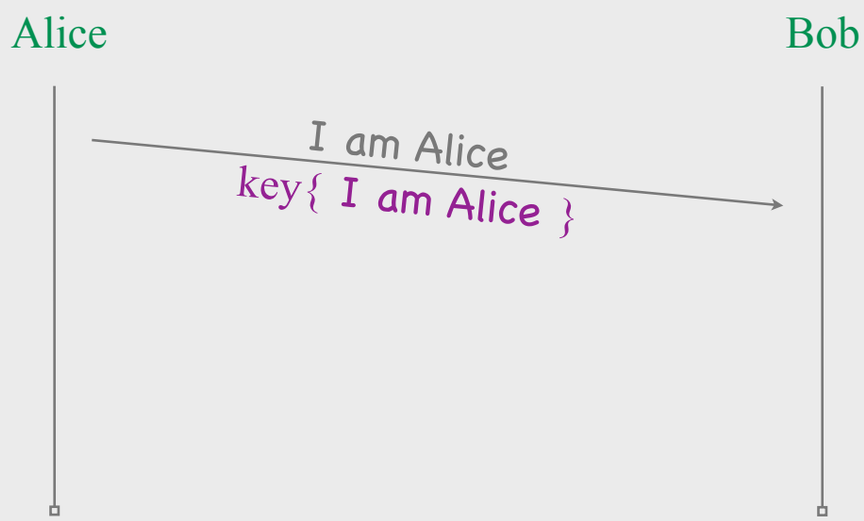
When Bob receives the message, he encrypts the plaintext included in the message using the shared key; if the outcome is the same as the ciphertext included in the message, then it must be Alice who sent the message (because only Alice could have produced a correct plaintext/ciphertext pair, since only Alice knows the shared key).

Note that this use of encryption does not provide confidentiality: Alice's sends *both* an unencrypted and an encrypted version of her message, so an adversary sitting on the channel between Alice and Bob can read Alice's message.

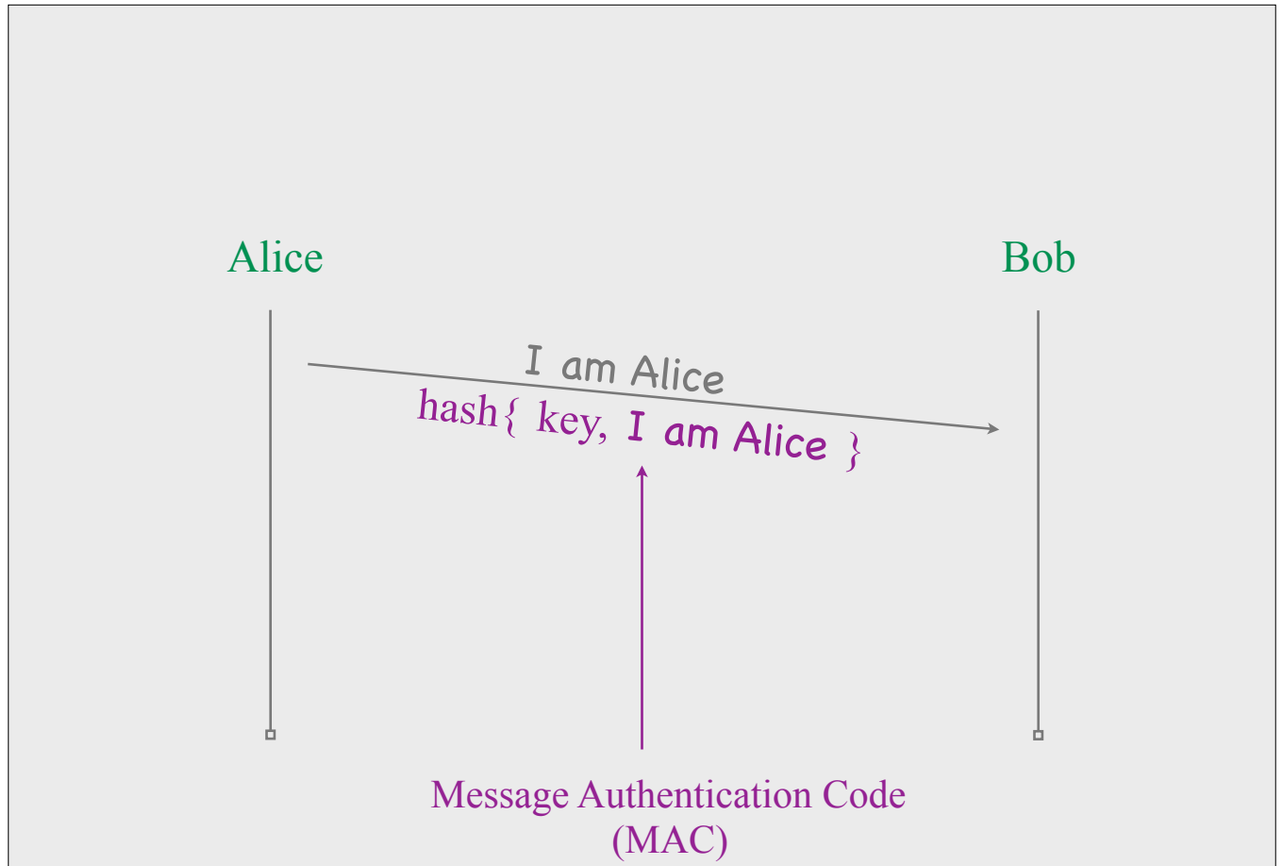


Persa may try to impersonate Alice, but she cannot generate a correct plaintext/ciphertext pair, since she does not know the shared key.

When Bob receives Persa's message, he encrypts it with the key he shares with Alice and sees that the outcome is different from the ciphertext in Persa's message, so he knows that the message cannot possibly come from Alice.

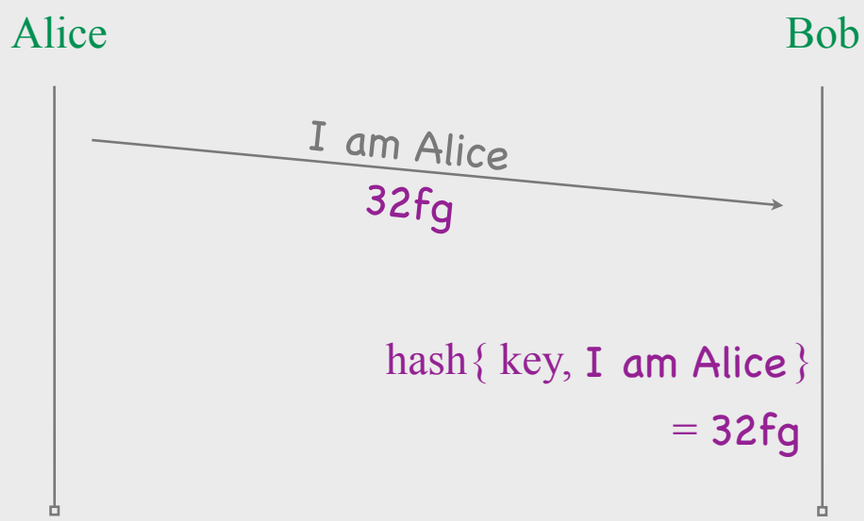


So, this approach provides authenticity, but it also doubles the size of the message that Alice sends.



A better approach is for Alice to send a message that consists of a plaintext and a cryptographic hash of the plaintext combined with the shared key.

The hash of the plaintext-and-key combination is called a “Message Authentication Code” (MAC).

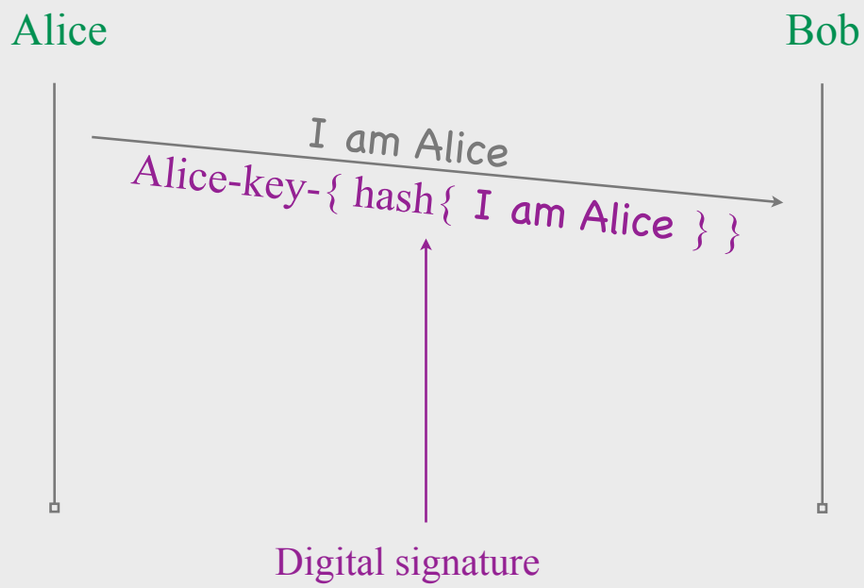


This provides authenticity:

When Bob receives the message, he computes the hash of the plaintext combined with the shared key; if the outcome is the same as the MAC included in the message, then it must be Alice who sent the message.

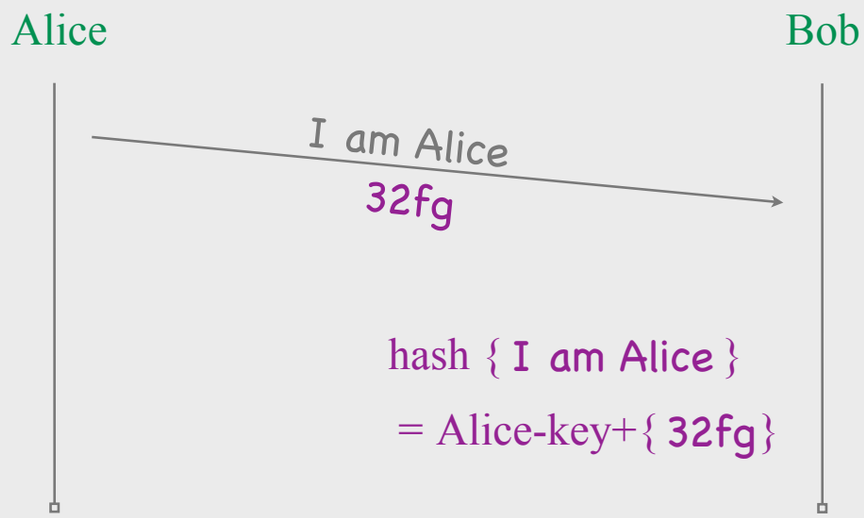
Message Authentication Code

- hash{ key, plaintext }
- Proof that this particular plaintext was sent by an entity that knows the key



Now suppose that Alice and Bob use asymmetric-key cryptography.

Instead of appending to her plaintext a MAC, Alice appends a “digital signature”: she hashes the plaintext and encrypts the hash with her private key.



This provides authenticity:

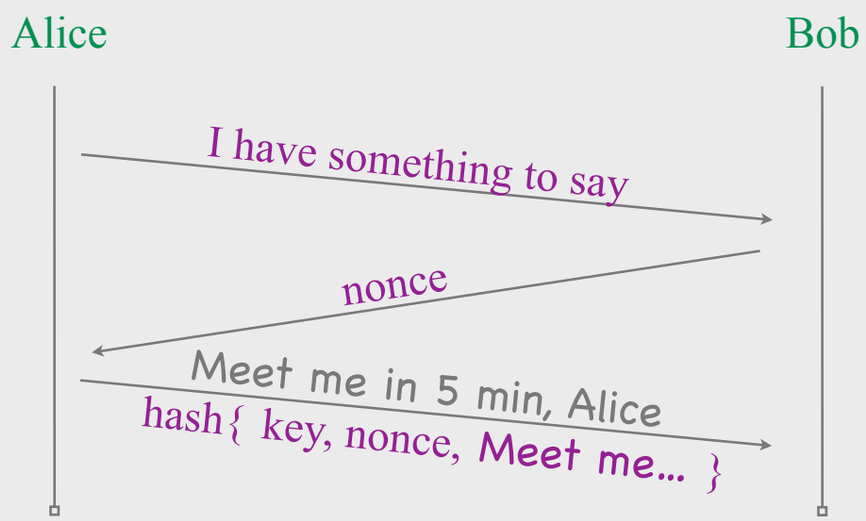
When Bob receives the message, he hashes the plaintext included in the message, and he also decrypts the digital signature with Alice's public key; if the two operations yield the same outcome, then it must be Alice who sent the message (otherwise Bob could not have decrypted the digital signature with Alice's public key).

Digital signature

- Generate: key-{ hash{ message } }
- Verify: key+{…} == hash { message }
- Proof that this particular message was sent by an entity who knows the private key that matches public key key+

Sometimes it is important to prevent Persa not only from impersonating Alice, but also from storing copies of Alice's messages and replaying them.

E.g., if Alice tells Bob "Meet me in 5 min", Persa could store this message and replay it to Bob every day.



Suppose Alice and Bob use symmetric-key crypto, i.e., share a secret key.

Here is one way to prevent replay attacks:

- Alice tells Bob that she has something to say to him.
- Bob responds with a “nonce,” i.e., a number that Alice is supposed to use *once* to generate a message for Bob within a given deadline.
- Alice sends her plaintext followed by a hash of the plaintext combined with the shared key *and the nonce*.

This prevents replay attacks: Alice is not supposed to use the nonce for more than one message. If Persa replays Alice’s message, Bob will detect that all the replayed messages were generated using an already used nonce, and he will reject them.

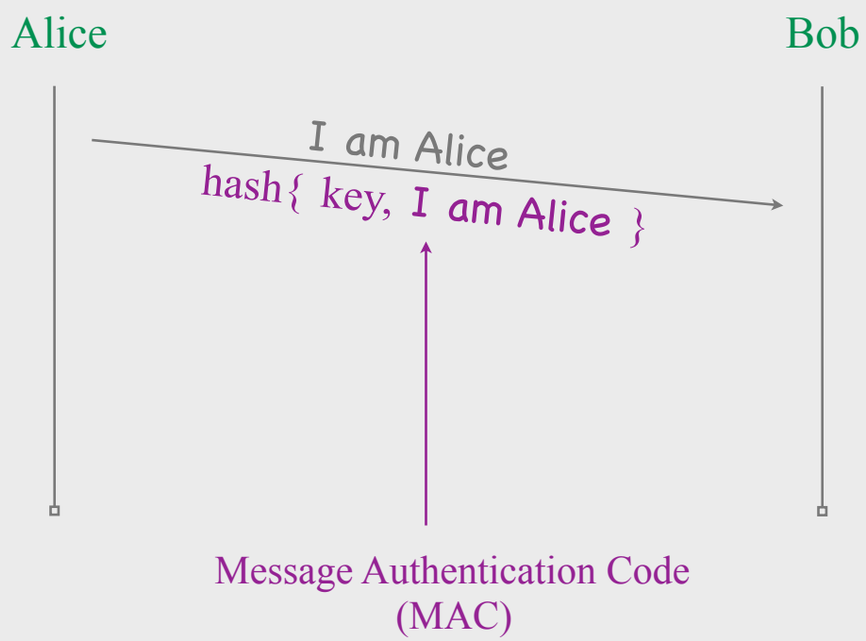
Providing authenticity

- With symmetric key crypto
  - Alice appends MAC
  - Bob checks that it is correct (using shared key)
- With asymmetric key crypto
  - Alice appends digital signature
  - Bob checks that it is correct (using Alice's public key)

Providing authenticity

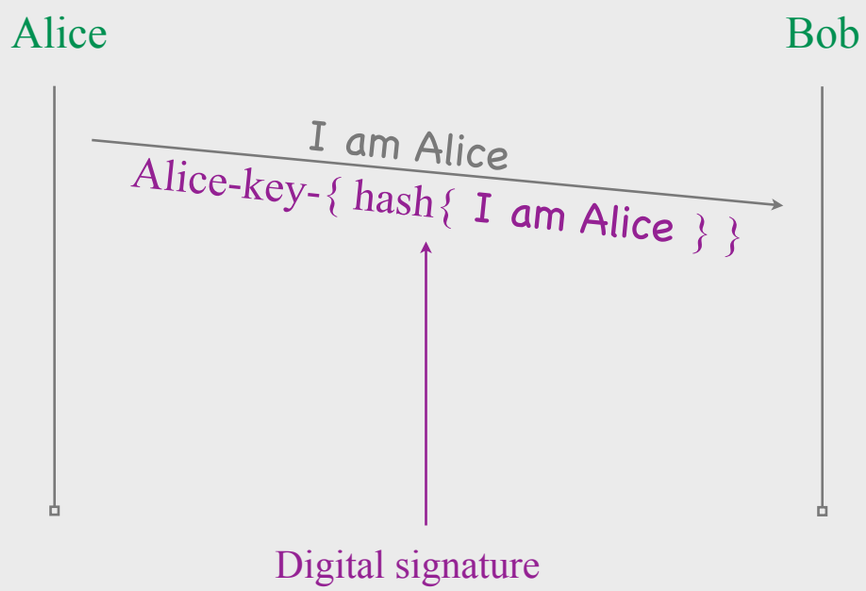
- Use nonce to prevent replay attacks
  - Alice appends MAC or digital signature of nonce + message
  - Bob verifies that it is correct

Providing data integrity



Suppose Alice and Bob use symmetric-key crypto, i.e., share a secret key.
Alice sends a message to Bob that includes a MAC.

The MAC provides not only authenticity, but also data integrity: if Bob validates the MAC, that proves not only that Alice sent this message, but also that nobody modified it (because nobody could have replaced the plaintext, then recomputed the MAC correctly without knowing the shared key).

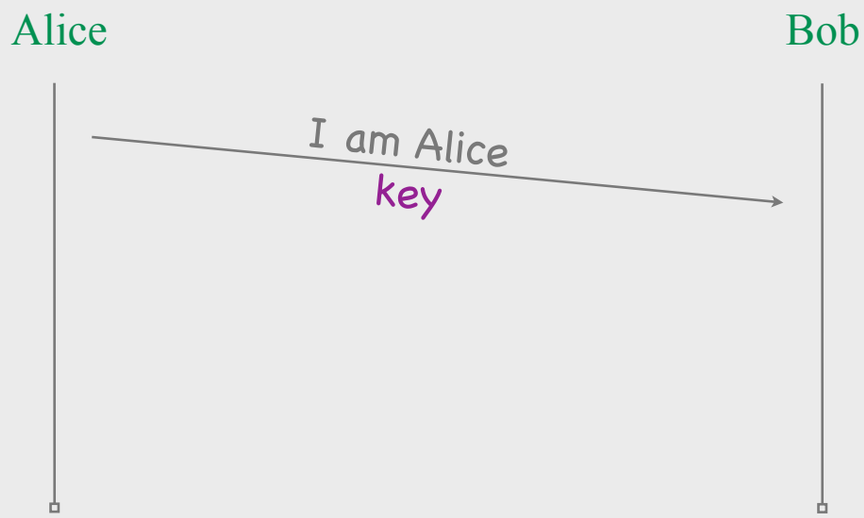


Suppose Alice and Bob use asymmetric-key crypto.
Alice sends a message to Bob that includes a digital signature.

The digital signature provides not only authenticity, but also data integrity: if Bob validates the digital signature, that proves not only that Alice sent this message, but also that nobody modified it (because nobody could have replaced the plaintext, then recomputed the digital signature correctly without knowing Alice's private key).

Providing integrity

- With exactly the same mechanisms that provide authenticity



In fact, it’s hard to think of authenticity and data integrity separately. It’s hard to imagine how we could have one without the other.

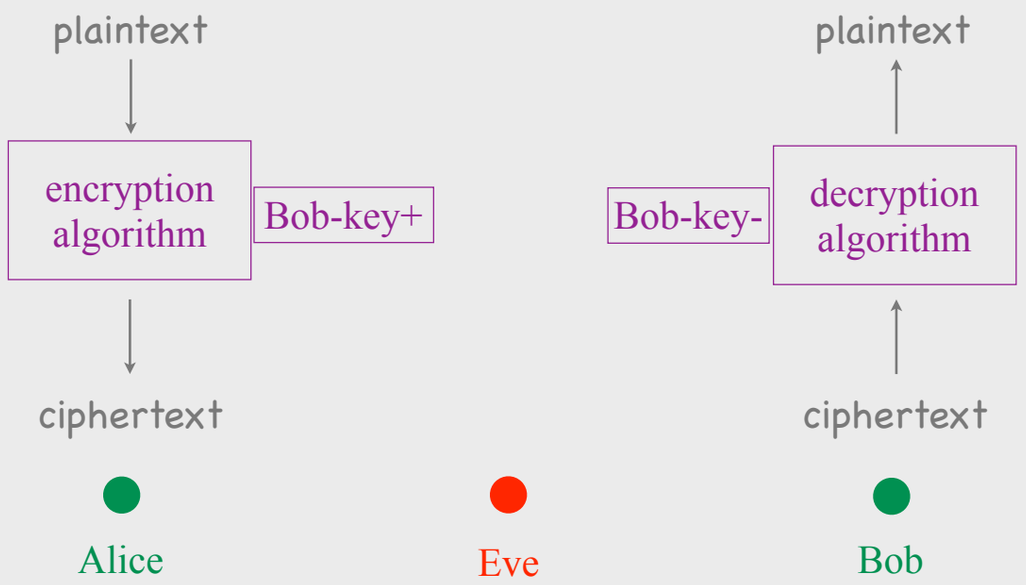
But here’s a contrived example:

Suppose Alice and Bob share a secret key.
Alice sends a plaintext followed by the shared key.

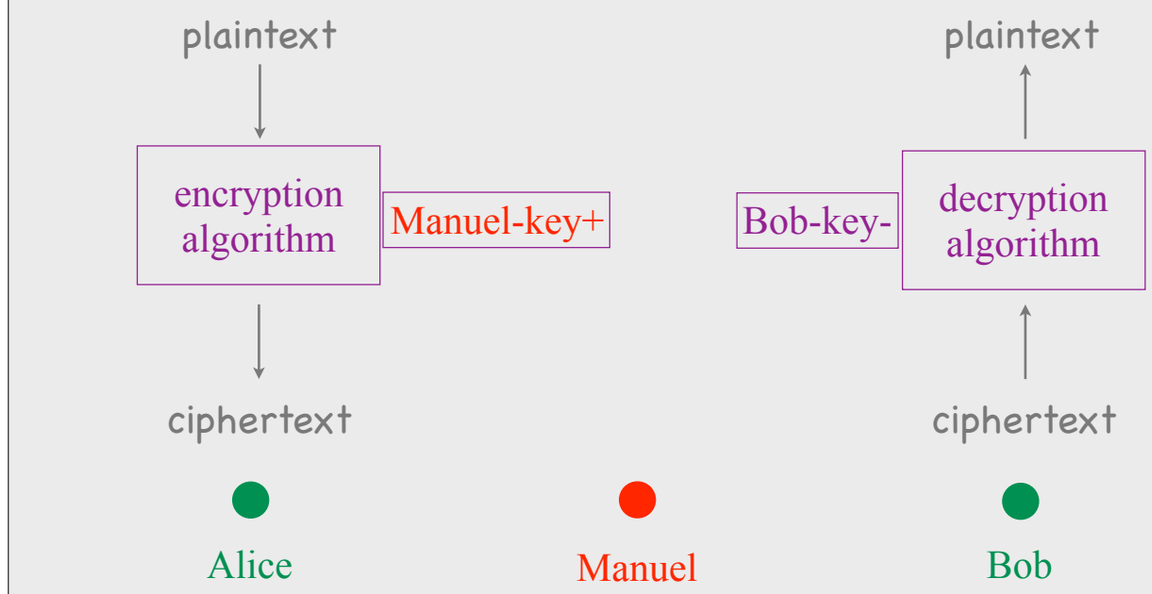
This provides authenticity, in the sense that it proves to Bob that Alice at some point sent him *a* message. It does not provide data integrity, because Persa may have changed the plaintext included in the message.

Also: This approach is pretty silly because it reveals to Persa the shared key, which means that the key could never be used again.

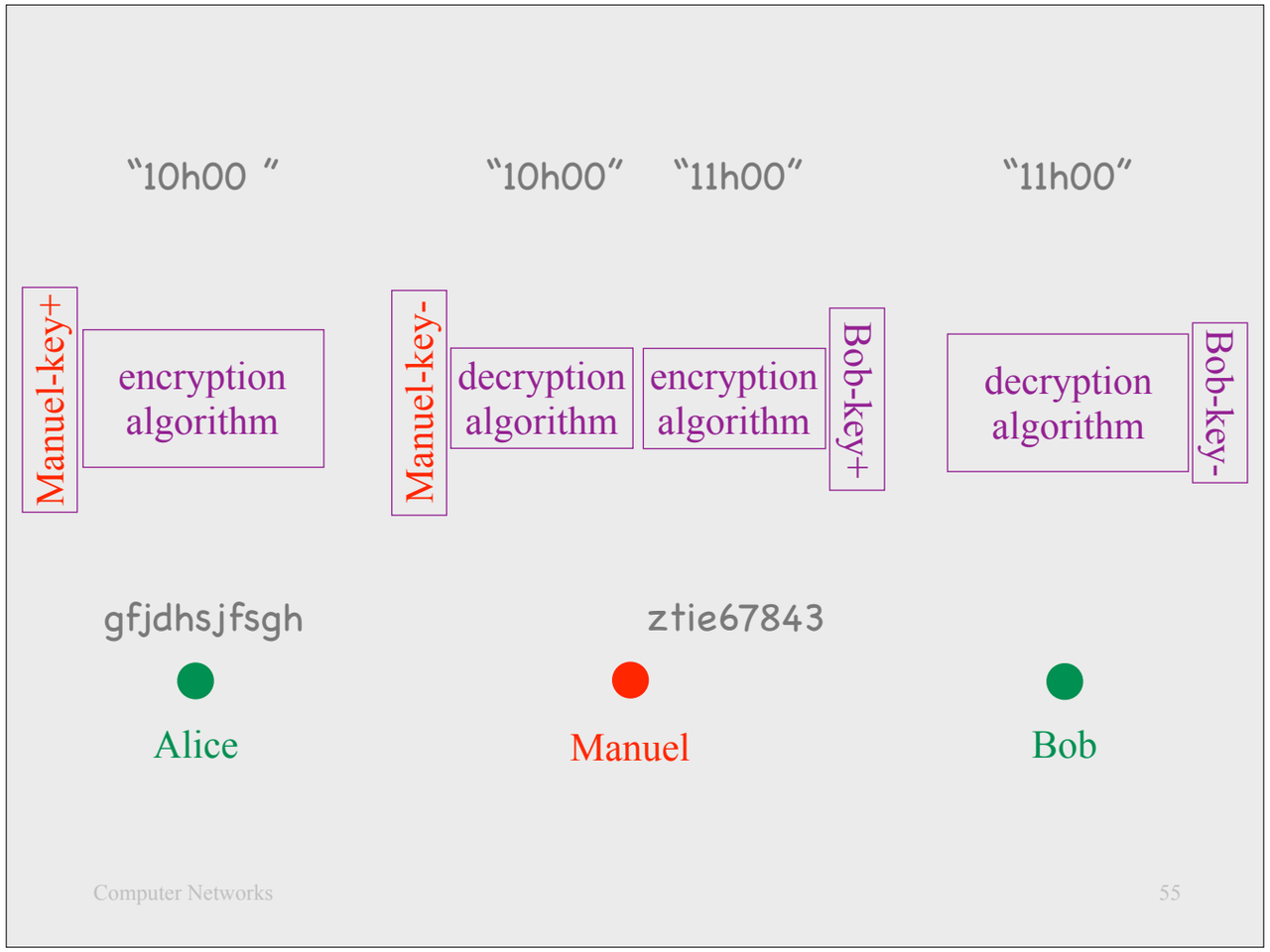
Preventing man-in-the-middle attacks



Suppose Alice and Bob use asymmetric-key crypto.
Alice encrypts messages with Bob's public key and sends them to Bob.
This provides confidentiality in the presence of an adversary like Eve, who passably sits on the communication channel between Alice and Bob.



But what if we have a more active adversary, Manuel, who gives Alice his public key and tricks her into believing that it is Bob's public key.



Now Manuel can read all communication from Alice to Bob:

- Alice encrypts a plaintext with Manuel's public key (believing it to be Bob's) and sends it out.
- Manuel intercepts the message and decrypts it with his private key. At this point confidentiality is compromised.
- Manuel optionally changes the plaintext, encrypts it using Bob's public key, and sends it out.
- Bob receives the message and decrypts it with his private key.

Man in the middle

- Can break confidentiality
  - Manuel convinces Alice to use his public key instead of Bob's
  - decrypts and re-encrypts Alice-Bob messages
- Cause: no way to verify public-keys
  - when Alice learns Bob's public key, she must verify that it is indeed his

A man-in-the-middle attack can break not only confidentiality (as we say in the previous example), but also authenticity. Can you think how?

Solution: public-key certificates

- Rely on trusted certificate authority (CA)
  - an entity that both Alice & Bob trust
- CA produces certificate of Bob's public key
  - { Bob owns Bob-key+, … }
  - CA-key-{ hash{ Bob owns Bob-key+, … } }

To prevent man-in-the-middle attacks, we use public-key certificates.

A public-key certificate is a statement that concerns a public key,
which is digitally signed by a "certificate authority" (CA),
a globally trusted entity, whose public key is generally known.

For example, consider a CA with public/private key pair CA-key+/CA-key-.
To generate a public-key certificate for Bob, the CA puts together:

- The statement "Bob owns public key Bob-key+"
- The digital signature of the statement: CA-key-{hash{Bob owns public key Bob-key+}}

So, a public-key certificate is essentially a message generated by the CA that carries proof that it was indeed generated by the CA.

Solution: public-key certificates

- Alice needs Bob's true public key
  - to produce Bob-key+{ message }
  - to check Bob-key-{ hash{ message } }
- Bob sends public key & certificate
  - CA-key-{ hash{ Bob owns Bob-key+, … } }
  - guarantees this is Bob's public key
- Alice needs CA's true public key
  - to check CA-key-{ hash{ Bob owns Bob-key+, … } }

Here is how Alice can obtain Bob's public key and be certain that it is his true public key:

- Alice asks Bob to send her his public key.
- Bob sends Alice his public key, Bob-key+, and a certificate [statement, signature], where statement="Bob owns public key Bob-key+" and signature = CA-key-{hash{statement}}.
- When Alice receives the certificate, she computes hash{statement} and CA-key+{signature}. If the two yield the same outcome, the certificate is valid.

Important point: Alice must know the CA's true public key, otherwise she will not be able to validate the CA's digital signature on the certificate.

Bootstrapping is unavoidable

- Secure communication requires some form of shared state
- Symmetric crypto: secret key
- Asymmetric crypto: CA's public key
  - typically stored in browser

Asymmetric crypto reduces bootstrapping information

In order for Alice and Bob to establish a secret communication channel (whether to achieve confidentiality or authenticity/data integrity or all of these), they must share some state — some knowledge.

If Alice and Bob use symmetric key cryptography, the shared knowledge is the shared secret.

If they use asymmetric key cryptography, the shared knowledge is the CA's public key (which is not a secret).

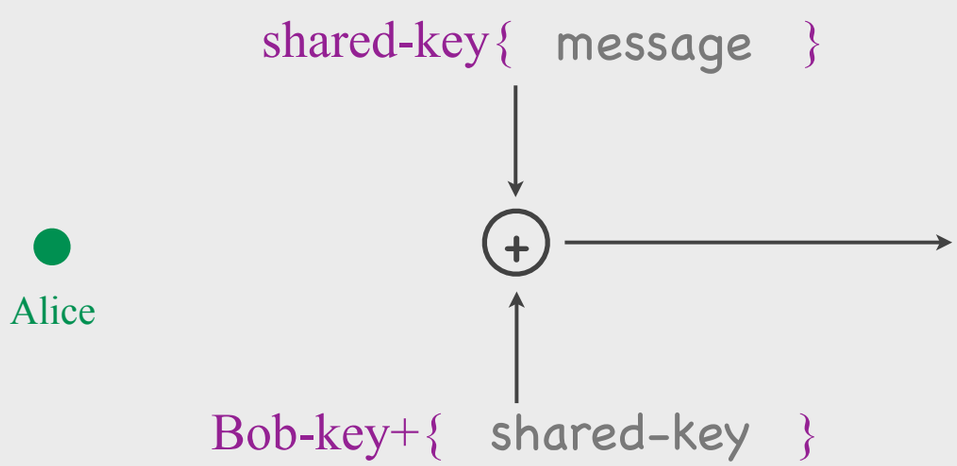
So: Asymmetric-key cryptography does not remove the need for Alice and Bob to share some knowledge. But it makes the shared knowledge non secret, hence easier to share.

Outline

- Building blocks
- Providing security properties
- Securing Internet protocols
- Operational security

Now let’s see how specific network protocols employ the techniques discussed so far to provide security properties.

In all following scenarios, assume an adversary who is sitting on the communication channel between the two communicating parties.



First, email.

Alice wants to send an email to Bob and ensure confidentiality.

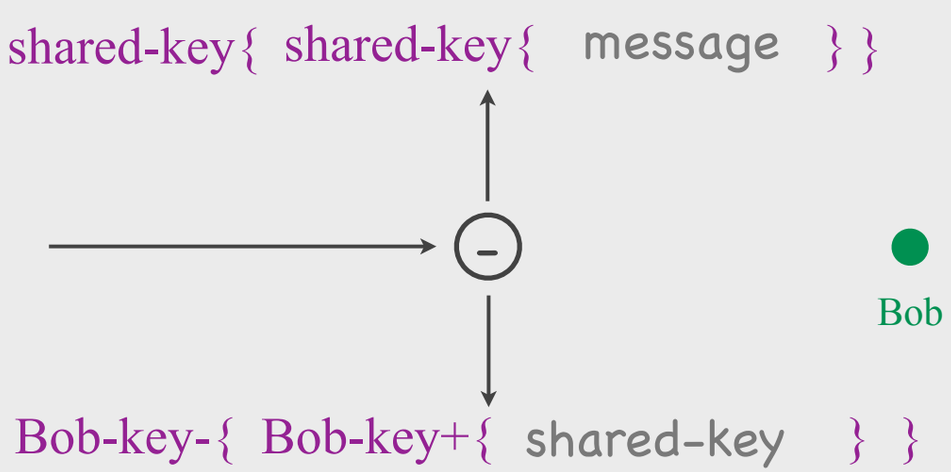
Based on what we have discussed so far, she could use symmetric-key cryptography, i.e., encrypt her email with a secret key that she shares with Bob, or asymmetric-key crypto, i.e., encrypt her email with Bob's public key.

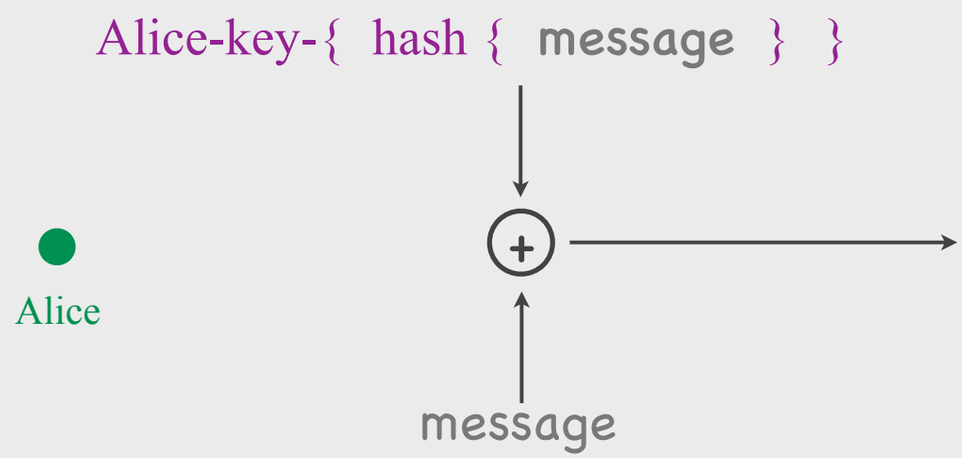
Both approaches have disadvantages: the former requires sharing a secret key, while the latter is more computationally expensive.

So she uses both:

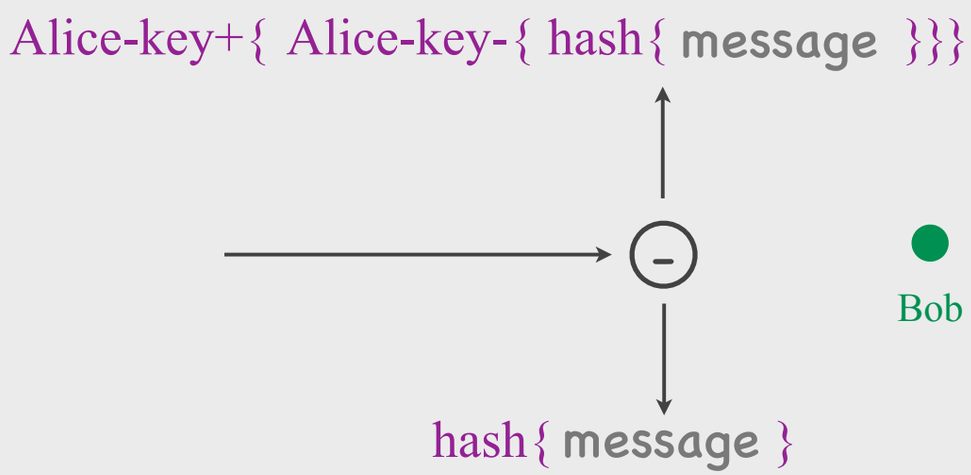
- Generates a key (denoted as "shared-key" on the slide, even though it is still not shared with Bob) and uses it to encrypt her email.
- Encrypts shared-key with Bob's public key.
- Sends to Bob both the encrypted email (encrypted with shared-key) and the encrypted shared-key (encrypted with Bob's public key).

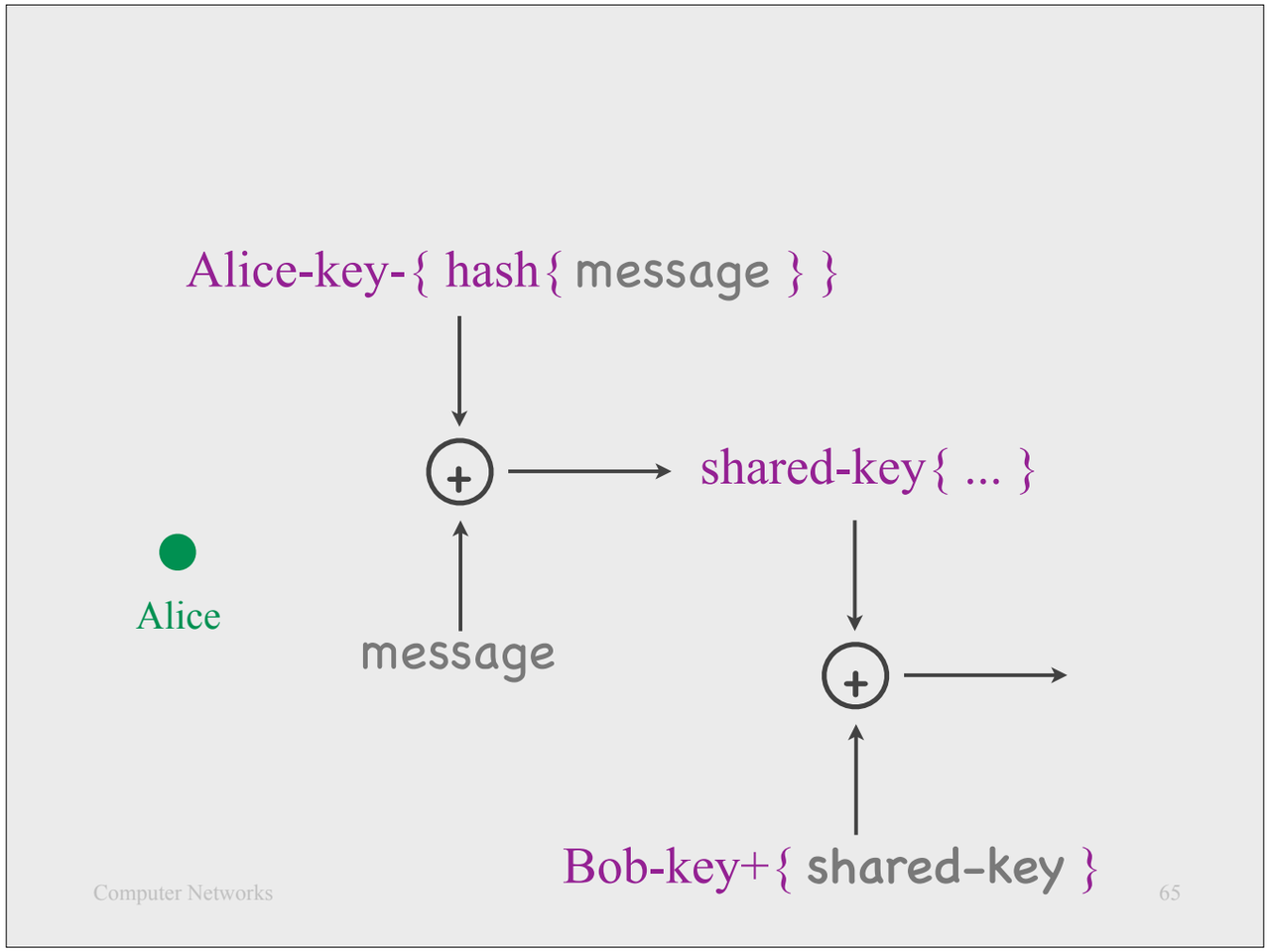
This combines the best of symmetric and asymmetric key cryptography: it does not require Alice and Bob to share a secret key in advance (before the communication begins), and it does not require them to use asymmetric-key encryption/decryption on large amounts of data (just the shared—key).


shared-key{ shared-key{ message } }
Bob
Bob-key-{ Bob-key+{ shared-key } }



Now Alice wants to ensure authenticity/data integrity.
Hence, she sends Bob her email followed by a digital signature of her email.


Alice-key+{ Alice-key-{ hash{ message }}}
Bob
hash{ message }

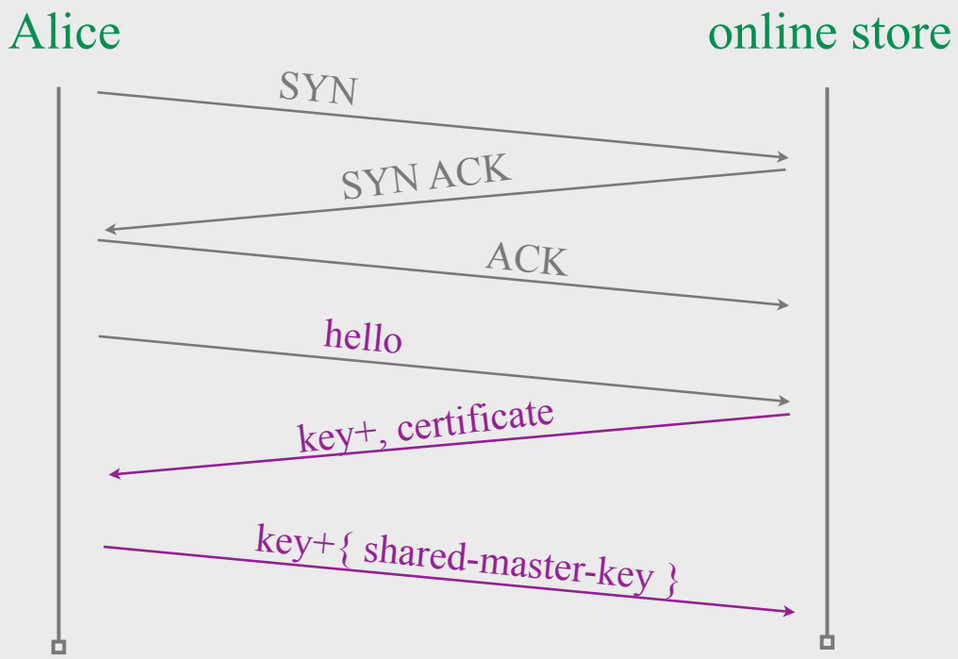


Finally, Alice wants to ensure both confidentiality and authenticity/data integrity.

Here's one way to do this:

- First authenticity/data integrity: Alice prepares her email message, followed by a digital signature of her email message.
- Then confidentiality: Alice encrypts [email, signature] with shared-key, then encrypts shared-key with Bob-key+.
- Alice sends Bob both the encrypted [email, signature] (encrypted with shared-key) and the encrypted shared-key (encrypted with Bob-key+).

This is the main idea behind the PGP protocol (stands for Pretty Good Privacy), which is used to secure email.



Next, TCP applications in general.

Alice has established a TCP connection to an online store in order to buy things. She wants to ensure both confidentiality and authenticity/data integrity.

Here's one way to do it:

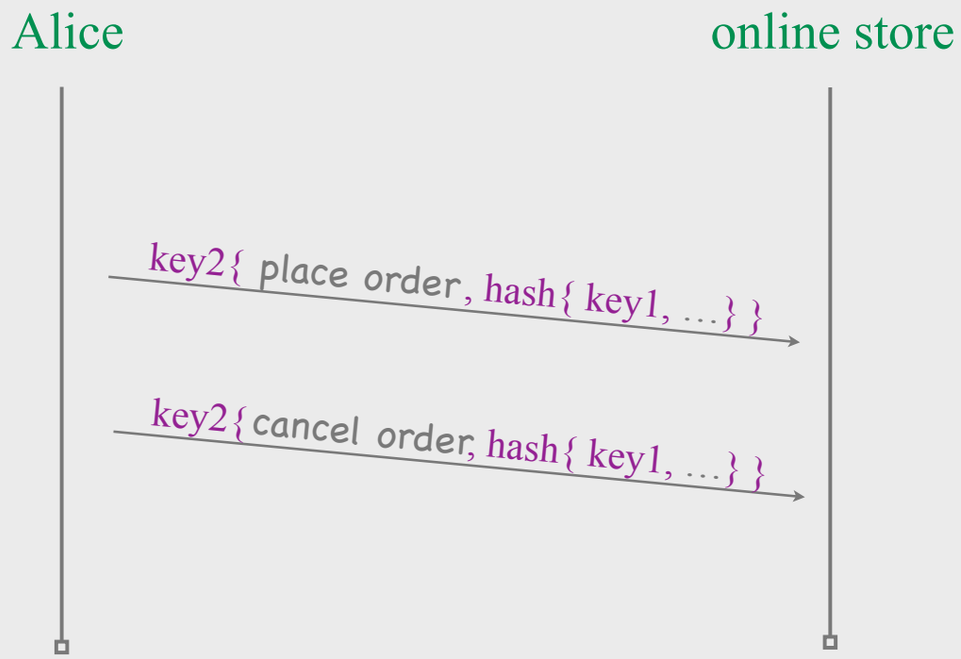
- Alice informs the online store that she wants to establish a secure channel (through the SSL hello message).
- The online store sends Alice its public key and a matching certificate.
- After verifying that the certificate is valid, Alice generates a shared-master-key, encrypts it with the online store's public key, and sends it to the online store.

So, Alice is using the same trick we saw in the PGP slides: she uses asymmetric-key crypto only to communicate a shared key, which can then be used for symmetric-key crypto.

Securing TCP applications

- Server sends its public key & certificate
- Client creates and sends a shared master key
  - encrypts it with server's public key
- Both use master key to create 4 session keys
  - 1 key for encrypting client --> server data
  - 1 key for creating MAC for client --> server data
  - same for server --> client data

Both Alice and the online store use the shared-master-key to generate other shared keys that they use for confidentiality and authenticity/data integrity in each direction (from Alice to the online store and vice versa).



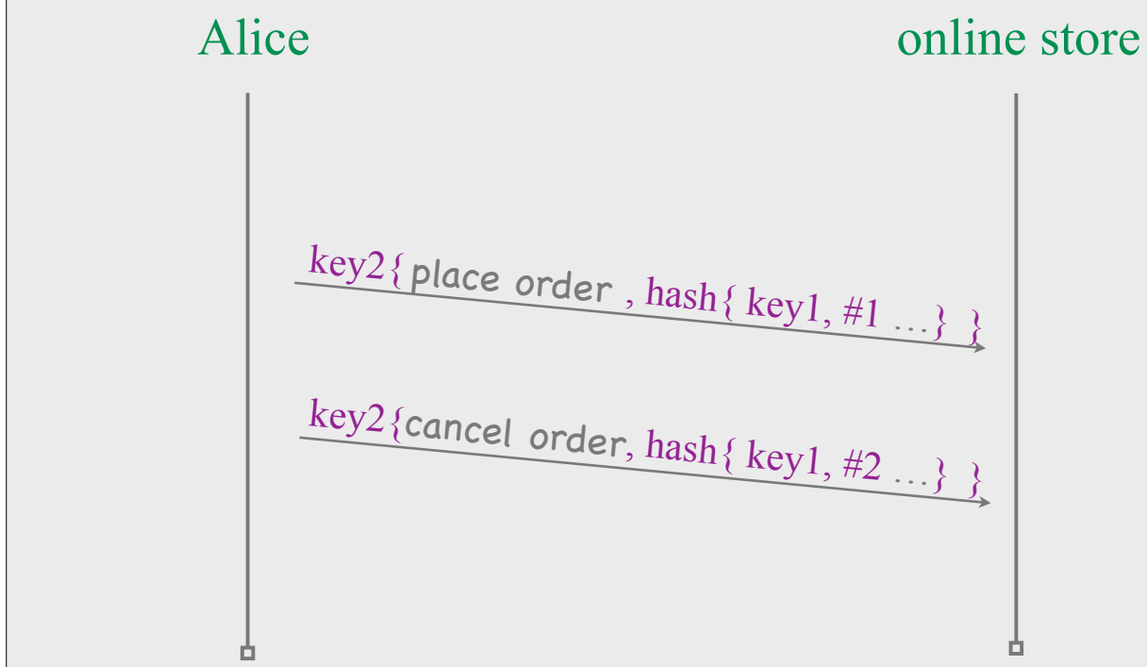
Now that Alice and the online store have established shared keys,
Alice can place orders while ensuring confidentiality and authenticity/data integrity *for each individual request* that she sends to the online store.

For example:

“place order”, hash{ key1, “place order” }
ensures authenticity/data integrity:
“place order” is a plaintext, while hash{ key1, “place order” } is a MAC.

key2{ “place order”, hash{ key1, “place order” } }
ensures confidentiality, because it encrypts the plaintext and MAC.

However, there is still something bad that Persa can do: she can replay a request (e.g., cause Alice to place multiple orders) or change the order of requests.



To prevent Persa from replaying or reordering requests, Alice does the following:

- She implicitly numbers her requests.
- When computing the MAC for each request, she includes in the MAC computation the request's implicit sequence number.

This way, when the online store receives Alice's sequence of requests, it can be certain not only that Alice sent these requests, but that she sent them in this particular order.

To see why, imagine that Persa reorders the two requests shown on the slide, such that the online store receives first the "cancel order" request, then the "place order" request.

Here's what happens next:

- The online store receives the "cancel order" request first.
- It decrypts the request using key2.
- It checks whether the MAC is valid. To do this, it computes hash{ key1, #1, "cancel order" }.
- The MAC validation fails, because the MAC was actually computed as hash{ key1, #2, "cancel order" }.

The online store now knows that its communication with Alice has been tampered with (there's an adversary on their channel) and interrupts the communication.

Securing TCP applications

- Client organizes data in records
  - each record has a sequence number
- Creates MAC for each record + sequence #
  - using one of the 4 session keys
- Encrypts the data + MAC for each record
  - using (another) one of the 4 session keys

This is the main idea behind the Secure Sockets Layer (SSL), which is used to secure TCP applications.

Key ideas

- Combination of symmetric/asymmetric keys
  - asymmetric key crypto to exchange shared keys
  - symmetric key crypto for confidentiality, authenticity, & integrity
  - symmetric key crypto is faster
- Seq. numbers to avoid reordering attacks
  - organize data in records with seq. numbers
  - compute MAC on record data + seq. number

Outline

- Building blocks
- Providing security properties
- Securing Internet protocols
- Operational security

We will close with a few words on operational security, i.e., basic things that network operators do to secure their networks, which have nothing to do with confidentiality, authenticity or data integrity.

action	src IP	dst IP	proto	src port	dst port
allow	167.67/16	any	TCP	> 1023	80
allow	any	167.67/16	TCP	80	> 1023
deny	all	all	all	all	all

A network operator typically wants to allow the minimum amount of traffic to enter/exit their network, in order to minimise risk (e.g., of malicious traffic reaching the end-systems inside their network).

The standard way to do this is to install a filtering table at each border router (each router that sits on the border between their network and the rest of the world).

A filtering table is like a forwarding table. But instead of telling the router how to forward packets, it tells the router which packets to drop and which ones to forward.

E.g., the filtering table on the slide tells the router to deny all traffic except for:

- all TCP traffic with source IP prefix 167.67.0.0/16, source port number >1024, and destination port number 80;
- all TCP traffic with destination prefix 167.67.0.0/16, destination port number >1024, and source port number 80.

(So, this filtering table essentially allows only HTTP traffic from and to a specific IP prefix.)