Lecture 5: Secure Channel, TLS/SSL & Miscellaneous Cryptography Topics

5.1 Strong authentication

5.2 Key exchange & authenticated key exchange

5.3 Putting all together: Securing a communication channel

5.4 Putting all together: TLS/SSL

5.5 Authenticated encryption

5.6 Birthday attack variant

5.7 Remarks on Security Reduction

5.8 Other interesting cryptography topics

5.9 Time-memory tradeoff for dictionary attack (optional)

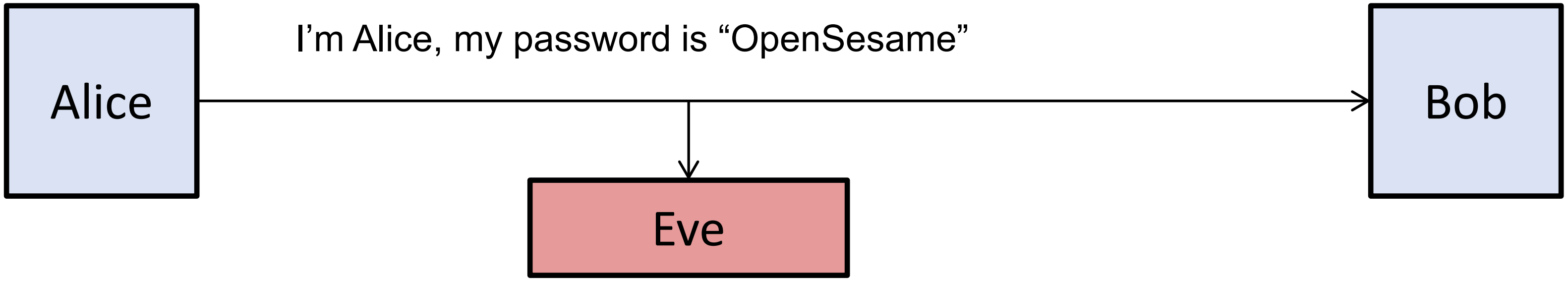
5.10 Summary of cryptography

5.1 Strong Authentication

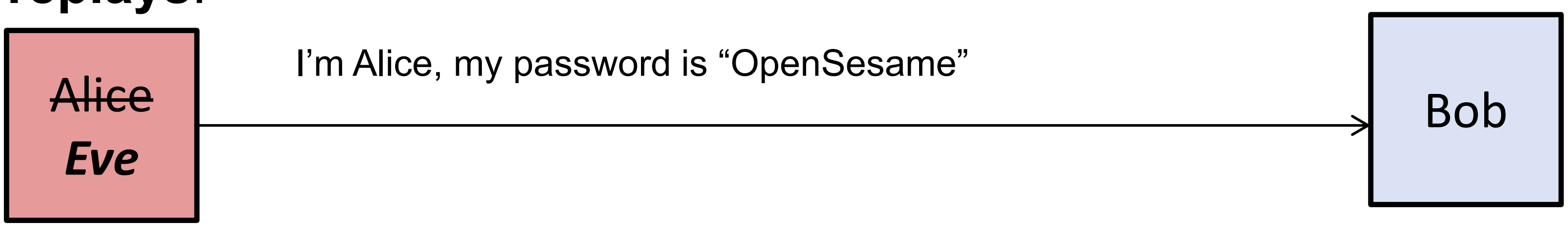
Password and Weak Authentication

- Password is a **weak** authentication system:
  an eavesdropper can get the password, and ***replay*** it

Eve **eavesdrops**:



Eve replays:



- The issue: the shared secret **is *exchanged*** between Alice & Bob
- Is it possible to have a mechanism that Alice can "prove" to Bob that she ***knows*** the secret ***without revealing*** the secret?
- This seems impossible, but there is **an easy way**

Strong Authentication: SKC-based Challenge-Response

Suppose Alice and Bob have a shared secret key ***k***,
and both of them agree on an **encryption scheme**, say AES

(1) Alice sends to Bob a hello message:

"Hi, I'm Alice"

(2) (**Challenge**) Bob randomly picks ***m***, and sends to Alice:

$$\boldsymbol{y = E_k(m)}$$

(3) (**Response**) Alice decrypts ***y*** to get ***m,*** and then sends ***m*** to Bob

(4) Bob verifies that the message received is indeed ***m***
If so, accepts;
otherwise rejects

Strong Authentication: SKC-based Challenge-Response (Analysis)

- Even if Eve can obtain all the communication between Alice and Bob, Eve still **can't get** the **secret key *k***
- Eve **can't replay** the **response** either:
  Because the challenge ***m*** is randomly chosen and
  likely to be **different** in the next authentication session
  → The ***m*** ensures ***freshness*** of the authentication process
- The protocol **only** authenticates Alice:
  Hence it is call ***unilateral authentication***
- There are also protocols to verify both parties,
  which are called ***mutual authentication***

Question:
What is "freshness" in the context of authentication protocol?

Strong Authentication: PKC-based Challenge-Response

Suppose Alice wants to authenticate herself to Bob using PKC

(1) Alice sends to Bob a hello message:

"Hi, I'm Alice"

(1) (**Challenge**) Bob chooses a **random number *r***, and sends to Alice:

"Alice, here is your challenge", ***r***

(2) (**Response**) Alice uses her ***private key*** to sign ***r***, and sends to Bob:

sign(*r*), Alice's certificate

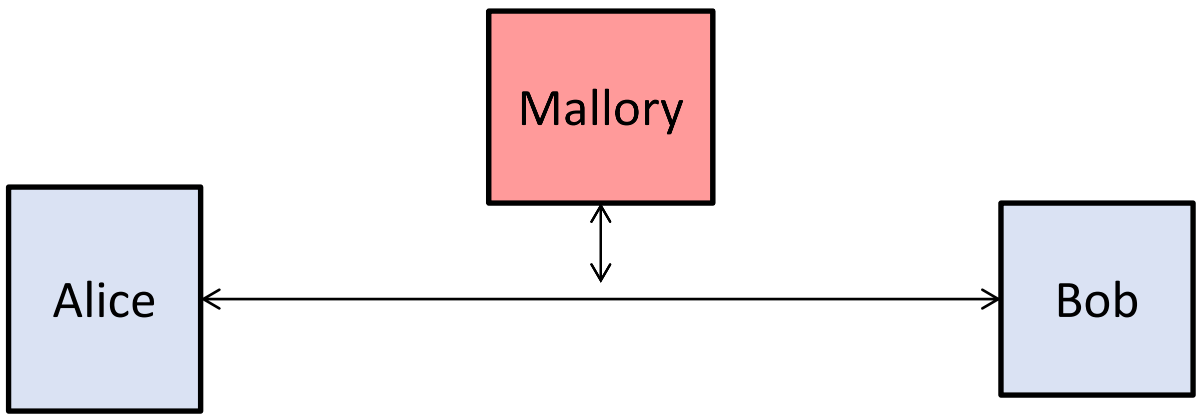
(3) Bob verifies Alice's certificate, extracts Alice's public key from the certificate, and then verifies that the **signature is correct**

Strong Authentication: PKC-based Challenge-Response (Analysis)

- An eavesdropper can't know/derive **Alice's private key** and replay the response because the challenge ***r*** is likely to be different
- The value ***r*** is also known as the ***cryptographic nonce*** (or simply ***nonce***)
- ***Question***: *Which component in the above ensures freshness?*
- **Remarks**:
  - The shown protocols have omitted many details
  - Designing a secure authentication protocol is **<u>not easy</u>**

Is Authentication Alone Sufficient?

- You may wonder what come next ***after*** an authentication
- Consider the typical setting of Alice, Bob and Mallory
- Mallory (who can modify messages) wants to impersonate Alice



- Imagine that Mallory allows Alice and Bob to carry out a strong authentication
- ***After*** Bob is convinced that he is communicating with Alice, Mallory **interrupts and takes over** the channel
- Subsequently Mallory can pretend to be Alice! *(duh)*

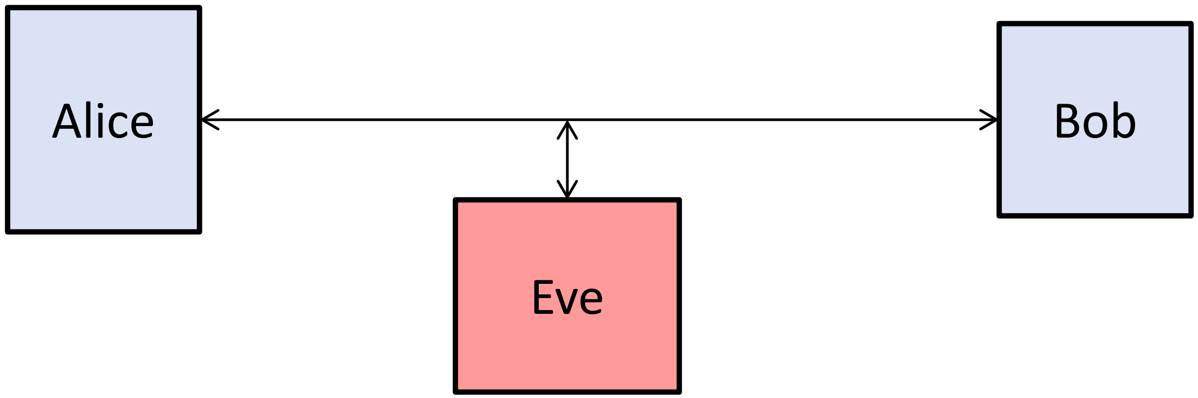
Is Authentication Alone Sufficient?

- Strong authentication, in its **basic form**, assumes that Mallory is ***unable*** to interrupt the session:
  e.g. a terminal access to server in a secure server room
- For applications whereby Mallory **can** interrupt the session, we thus need ***something more***!
- The **outcome** of the authentication process must be:
  a new secret ***k*** (a.k.a. ***session key***) established by Alice & Bob
- The process of establishing a secret between Alice & Bob is called ***key exchange***, ***key establishment***, or ***key agreement***
- Subsequent communication between Alice & Bob must be **secured using the session key**

5.2 Key Exchange & Authenticated Key Exchange

Basic/Unauthenticated Key Exchange (No Authentication)

- Alice & Bob want to establish a **common key**: both are assumed to be *authentic*
- The channel could be eavesdropped by ***Eve***

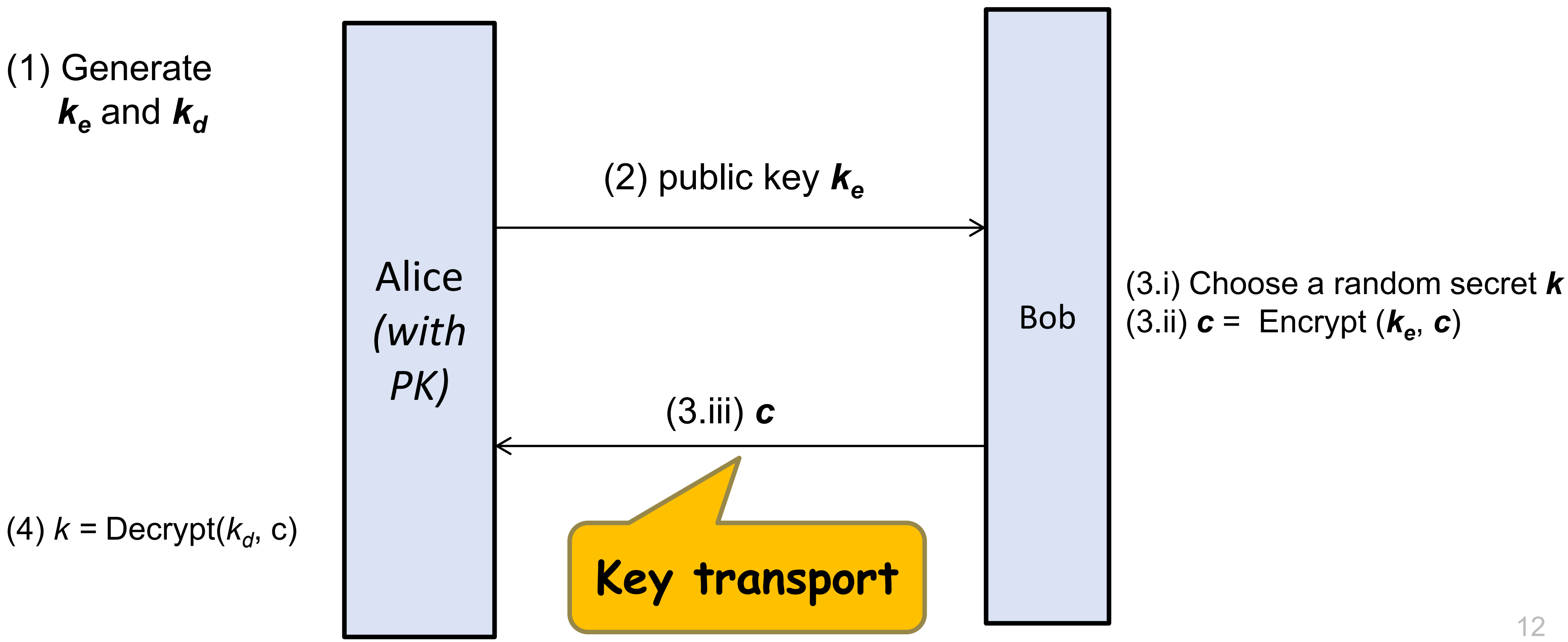


- They want a ***key exchange protocol***, such that Eve is unable to **extract any information** of the established key:
  the established key can be used to protect (e.g. via cipher, MAC) subsequent communication between Alice & Bob

*Note: Here we only consider **Eve** who can sniff,*
and not the malicious Mallory who can modify the communication!

1: PKC-based *Unauthenticated* Key Exchange

1. Alice generates a pair of private/public key
2. Alice sends the public key $\boldsymbol{k_e}$ to Bob
3. Bob carries out the following:
   i. Randomly choose a secret $\boldsymbol{k}$
   ii. Encrypt $\boldsymbol{k}$ using $\boldsymbol{k_e}$
   iii. Send the ciphertext $\boldsymbol{c}$ to Alice
4. Alice uses her private key $\boldsymbol{k_d}$ to decrypt and obtain $\boldsymbol{k}$.

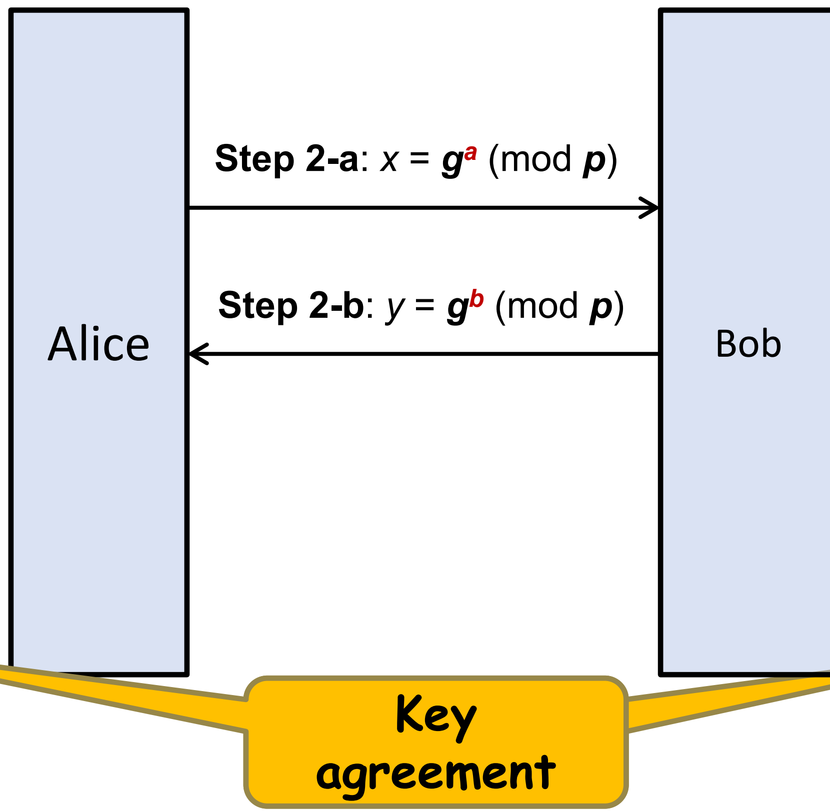


2: *Basic/Unauthenticated* Diffie-Hellman Key Exchange

- We assume both Alice and Bob have agreed on two ***publicly-known*** parameters: a ***generator g,*** and a large (e.g. 1,000-bit) prime ***p***

Step 1-a:
- Randomly choose a
- Compute $x = g^a \pmod{p}$

Step 1-b:
- Randomly choose b
- Compute $y = g^b \pmod{p}$



Step 3-a:
- Compute $k = y^a \pmod{p}$

Step 3-b:
- Compute $k = x^b \pmod{p}$

$k = g^{ab} \pmod{p}$

$k = g^{ab} \pmod{p}$

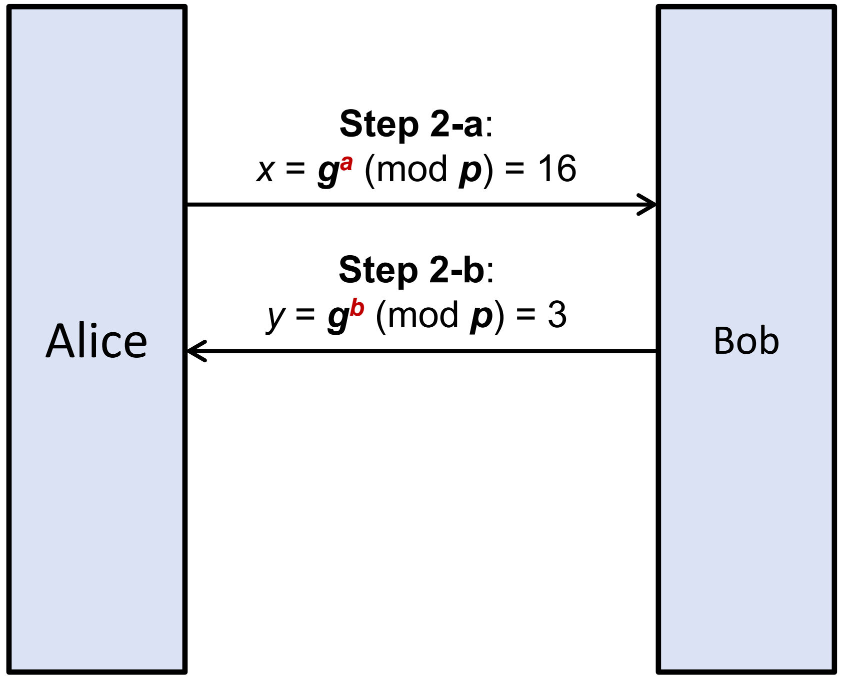
Remark:
Steps 1-a & 1-b, Steps 2-a & 2-b, and Steps 3-a & 3-b can be carried out ***in parallel***

Diffie-Hellman Key Exchange: Example

- Suppose $g = 2$, and $p = 23$

Step 1-a:
- Randomly choose $\boldsymbol{a}$=15

Step 1-b:
- Randomly choose $\boldsymbol{b}$=8



Step 3-a:
- Compute
  $k = y^{\boldsymbol{a}} \text{ (mod } \boldsymbol{p})$
  $= 3^{15} \text{ (mod 23)}$
  $= \mathbf{12}$

Step 3-b:
- Compute
  $k = x^{\boldsymbol{b}} \text{ (mod } \boldsymbol{p})$
  $= 16^{8} \text{ (mod 23}$
  $= \mathbf{12}$

Remark (optional):
DH key exchange achieves additional security property called ***forward secrecy (FS)***:
https://en.wikipedia.org/wiki/Forward_secrecy

Diffie-Hellman Key Exchange: Computational Analysis

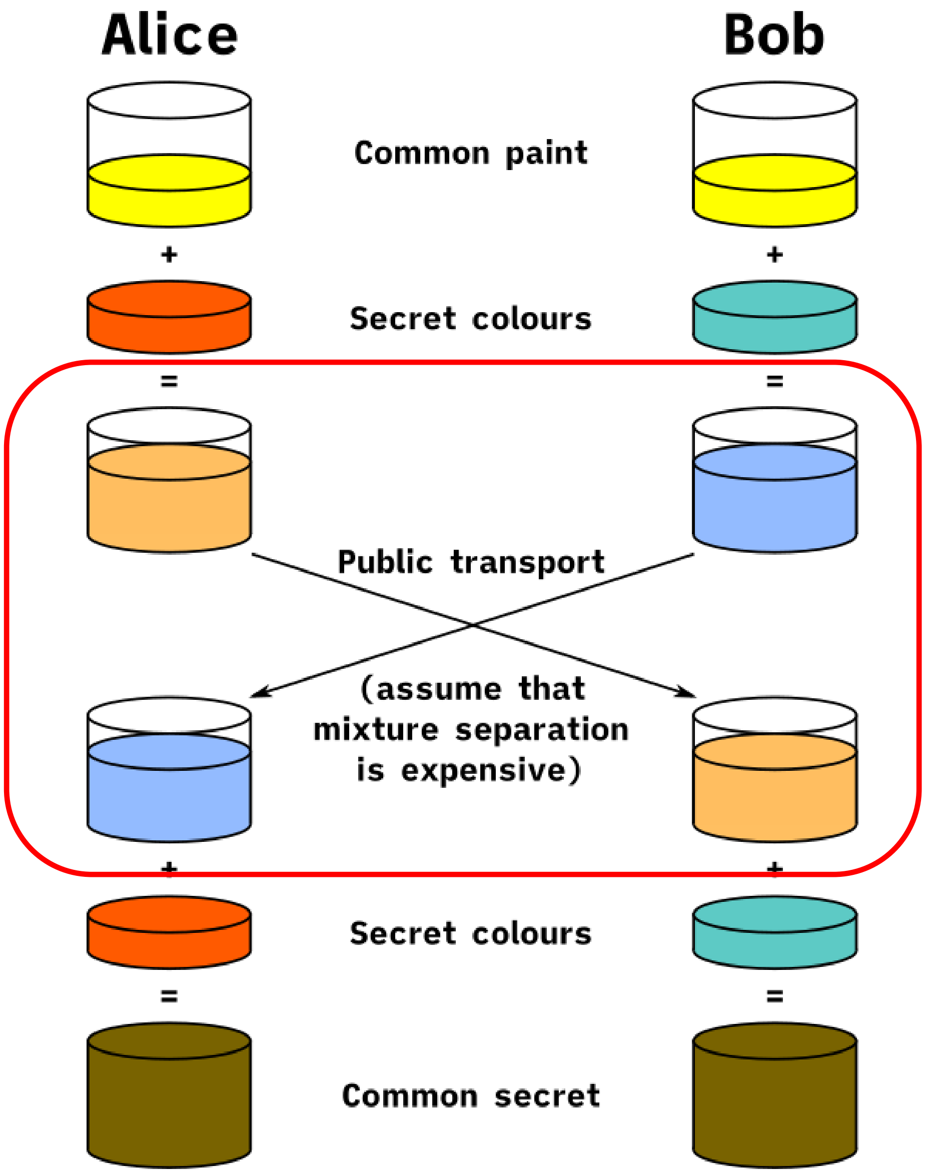
Optional

- Hard problems in **RSA**:
  - **Factoring** problem: given $n=p.q$; find p and q
  - **RSA** problem: given n, e, and $c = m^e \pmod{n}$; find m
- **Discrete Log (DL)** problem:
  - A ***cyclic group***: (optional https://en.wikipedia.org/wiki/Cyclic_group)
  - Let g be the ***generator*** of the cyclic group:
    g can derive all other elements in the set, e.g. by using exponentiation
  - **Given n, g, and $x = g^e \pmod{p}$; find e**
- **Computational Diffie-Hellman (CDH)** problem:
  - **Given p, g, $x=g^a \pmod{p}$, $y=g^b \pmod{p}$; find $k=g^{ab} \pmod{p}$**
- Diffie-Hellman key exchange works by assuming **Discrete Log** and **Computational Diffie-Hellman** are computationally hard

Remarks:

1. The assumption seems self-fulfilling. Nonetheless, there are much evidence that it holds.
2. The "exponentiation" operation can be applied to any algebraic group, i.e. not necessary integers.
   Note that CDH doesn't hold in certain groups.
   The crypto community actively searches for groups that CDH holds:
   e.g. Elliptic Curve Cryptography (ECC) is based on **elliptic curve group** where CDH is believed to hold.

Diffie-Hellman Key Exchange: Analogy/Visualization



Two assumptions:
- Colour mixing is easy
- Mixture separation is hard

Source: Wikipedia

Diffie-Hellman Key Exchange: Another Example

- From Wikipedia:
  g = 5, p = 23, a=6, b=15, and the key s=2

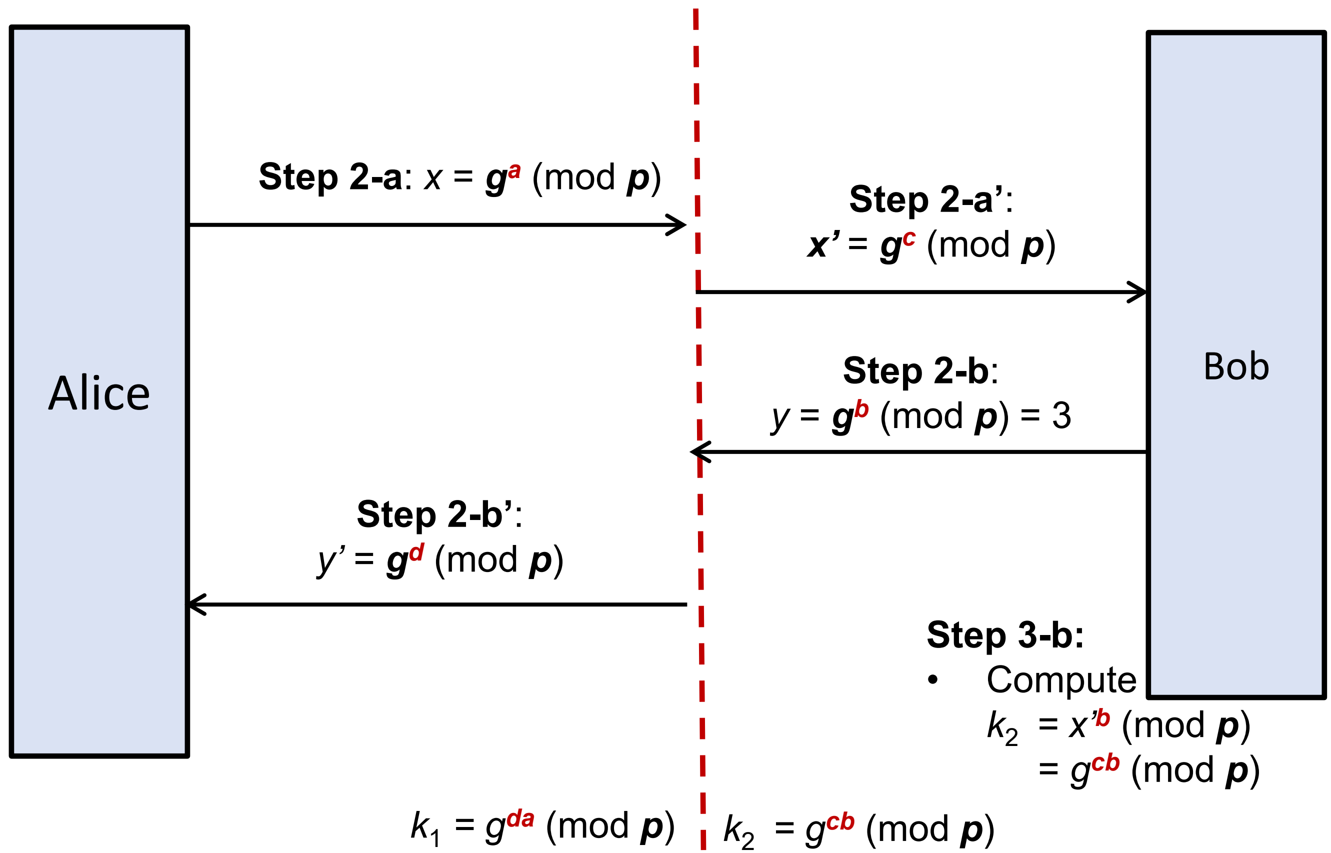
Alice		Bob		Eve	
Known	**Unknown**	**Known**	**Unknown**	**Known**	**Unknown**
$p = 23$		$p = 23$		$p = 23$	
$g = 5$		$g = 5$		$g = 5$	
$\boldsymbol{a} = \mathbf{6}$	$\boldsymbol{b}$	$\boldsymbol{b} = \mathbf{15}$	$\boldsymbol{a}$		$\boldsymbol{a}$, $\boldsymbol{b}$
$A = 5^{a} \bmod 23$		$B = 5^{b} \bmod 23$			
$A = 5^{6} \bmod 23 = 8$		$B = 5^{15} \bmod 23 = 19$			
$\boldsymbol{B} = \mathbf{19}$		$\boldsymbol{A} = \mathbf{8}$		$A = 8$, $B = 19$	
$\boldsymbol{s} = B^{a} \bmod 23$		$\boldsymbol{s} = A^{b} \bmod 23$			
$\boldsymbol{s} = 19^{6} \bmod 23 = \mathbf{2}$		$\boldsymbol{s} = 8^{15} \bmod 23 = \mathbf{2}$			$\boldsymbol{s}$

Source: Wikipedia

Is *Basic/Unauthenticated* DH Key Exchange Secure?

- Now, assume the presence of **Mallory**

Step 1-a:
- Randomly choose $\boldsymbol{a}$
- Compute $x = \boldsymbol{g}^{a}$ (mod $\boldsymbol{p}$)

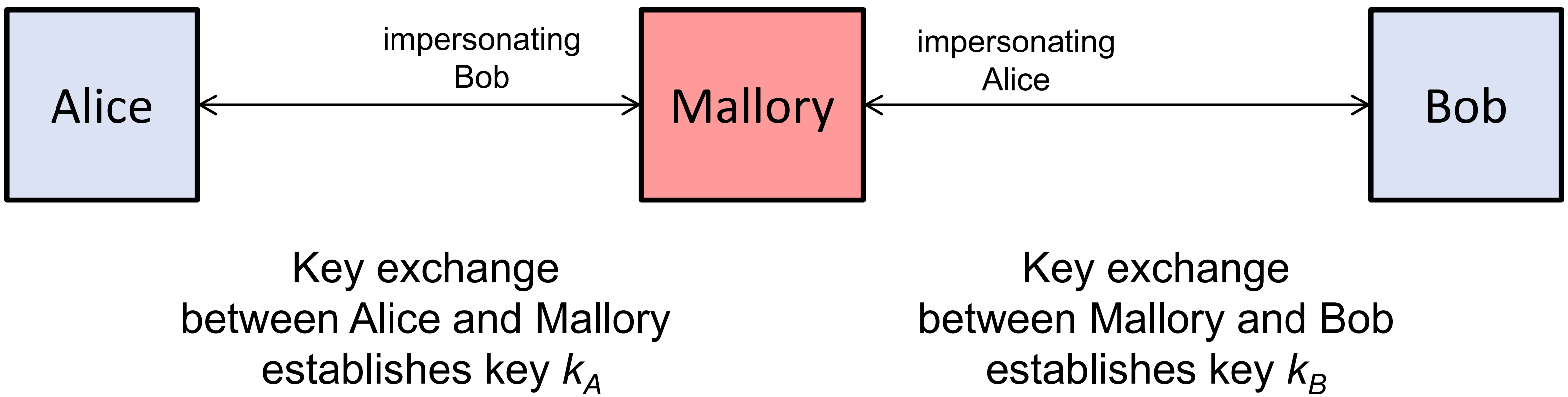


Step 3-a:
- Compute
  $k_1 = y'^{a}$ (mod $\boldsymbol{p}$)
  $= g^{da}$ (mod $\boldsymbol{p}$)

From the attack: DH does ***not*** achieve entity authentication

Is *Basic/Unauthenticated* Key Exchange Secure?

- With **Mallory** as a man-in-the-middle



- In this case, Alice mistakes Mallory for Bob
- Since communication from Alice is encrypted using k_1, Mallory **can decrypt** it using k_1 and re-encrypt using k_2
- Hence, Mallory can **see and modify** the message
- The same attack technique applies to communication from Bob
- (It turns out that **PKC-based authenticated key-exchange** can be easily derived from existing the key-exchange: simply **sign** all communications)

Why is That so?

- Question: *so, what's still wrong really?*
- Think about these 2 **different** goals/ requirements for an entity *A* in communicating with *B*:
  - ***Key Secrecy (Confidentiality)***:
    Is *k* a "good key" to talk to *B*?
  - ***Entity Authenticity***:
    Can *A* be sure that it's really talking to *B*?
- **Solution**:
  Incorporate the key exchange process with authentication → ***authenticated key exchange (AKE)***

Station-to-Station (STS) Protocol

- Here, we consider unilateral authentication: Alice want to authenticate Bob
- Adding **signature** to DH: **authenticated** key-exchange based on DH

Alice knows Bob's public key: Bob_{public}

$(Bob_{public}, Bob_{private})$

Step 1-a:
- Randomly choose $\boldsymbol{a}$
- Compute $x = \boldsymbol{g}^{\boldsymbol{a}} \pmod{\boldsymbol{p}}$

Step 1-b:
- Randomly choose $\boldsymbol{b}$
- Compute $y = \boldsymbol{g}^{\boldsymbol{b}} \pmod{\boldsymbol{p}}$
- **Sign** y to derive the signature $\boldsymbol{s}$

Alice

Bob

Step 2-a: $x = \boldsymbol{g}^{\boldsymbol{a}} \pmod{\boldsymbol{p}}$ →

← **Step 2-b**: **(*y*, *s*)**

Step 2-b':
y', s'?

Step 3-a:
- **Verify** signature $\boldsymbol{s}$
- Compute $k = y^{a} \pmod{\boldsymbol{p}}$

Step 3-b:
- Compute $k = x^{b} \pmod{\boldsymbol{p}}$

$k = g^{ab} \pmod{\boldsymbol{p}}$

$k = g^{ab} \pmod{\boldsymbol{p}}$

How about the unsigned x in Step 2-a?

Unilateral vs Mutual Authenticated Key Exchange

- The unilateral authentication protocol earlier can be extended to a **mutual authentication**: make Alice sign her message in Step 2-a
- **Requirements** for carrying out an authenticated key exchange:
  - For a mutual authentication, Alice and Bob must have a way to know **each other's public key** (e.g. using PKI)
  - For a unilateral authentication, only **one party** needs to have public key
  - *Qn: only 1 session key used to secure subsequent communication?*
  - After the protocol has completed, **a set of session keys** is established using a ***Key Derivation Function (KDF)*** like HKDF (based on HMAC) https://en.wikipedia.org/wiki/Key_derivation_function, e.g.:
    - 1 key for encryption & 1 key for MAC
    - Even more: 1 key for client-encryption, 1 key for server-encryption, 1 key for client-MAC, 1 key for server-MAC

Summary: Mutual Authenticated Key Exchange

- *Before the protocol:*
  - *A* has a pair of public/private key pair (A_{public}, $A_{private}$)
  - *B* has a pair of public/private key pair (B_{public}, $B_{private}$)
  - *A* knows *B* public key and vice versa
- Carry out the authenticated key exchange protocol:
  if an entity is not authentic, the other will halt
- *After the protocol*:
  - Both *A* and *B* obtain a shared key ***k***, known as the ***session key***
  - Other secret keys can be derived from *k*
- Security requirements:
  - Authenticity-1:
    A is assured that it is communicating with an entity who knows $B_{private}$
  - Authenticity-2:
    B is assured that it is communicating with an entity who knows $A_{private}$
  - Key confidentiality: the attacker is unable to get the session key *k*

Additional Remarks (Optional)

Password Authenticated Key Exchange (PAKE)

Optional

- The authenticated key exchange can also be used in **symmetric-key setting**. In symmetric-key setting, both *A* and *B* share a common secret, typically a **password**. Both conduct authentication + key exchange based on this secret.
- When the shared secret is a password, it is often called ***Password Authenticated Key Exchange*** (PAKE):
  - See https://en.wikipedia.org/wiki/Password-authenticated_key_agreement
  - A crucial difference between human-generated password and machine-generated key: passwords are typically shorter and vulnerable under dictionary attack.
    (Of course, if the password is strong, e.g. 20 characters that are randomly chosen, then dictionary attack is infeasible).
  - Offline dictionary attacks: After the attacker logged the traffic, it carries out offline exhaustive/dictionary attack by trying all possible passwords.
    The process could take longer time, but it doesn't need to remain connected to the victim.
- A **secure password authenticated key exchange** ensures that even if the passwords are short, the adversary (Eve or Mallory who pretends to be one of the authenticating party) can't carry out offline dictionary attack.
- Note that simply adding **MAC** using passwords cannot prevent offline dictionary attack. Since MAC is deterministic, Eve, after capturing the MAC, can test it on the dictionary in offline. So, the design of password authenticated key exchange is more complicated. Details are omitted.

Password Authenticated Key Exchange (PAKE)

Optional

- Some examples:
  - Encrypted Key Exchange (EKE)
  - PEAP (Protected Extensible Authentication Protocol)

- Some protocols like LEAP (**Lightweight Extensible Authentication Protocol)** are vulnerable to offline dictionary attacks.
  In contrast, PEAP is secure against offline dictionary attacks.

- ***Question***: *Which protocol does NUS WiFi employ?*
  In our previous example of an attacker spoofing a fake NUS hotspot in NUH bus stop, can the attacker successfully steal password?
  (See NUS WiFi setup instruction at: http://www.nus.edu.sg/comcen/gethelp/guide/itcare/wireless/NUS-WPA2%20Network%20Configuration%20Guide%20for%20Android%207.0.pdf)

5.3 Putting All Together: Securing a Communication Channel

Secure Communication Channel Problem

- Consider a communication channel that is subjected to **sniffing and spoofing**: does this reminds us of the Internet?
- **Question**: How can we securely communicate over it using cryptographic primitives?
- *A “**secure channel**”*:
  establishes, between 2 programs, a data channel that has **confidentiality, integrity, authenticity** against a computationally-bounded network attacker (i.e. Mallory)
- *Question: what cryptographic primitives to use?*

Confidentiality → Encryption

Integrity → MAC

Integrity → Digital signature

Authenticity → MAC

Authenticity → Digital signature

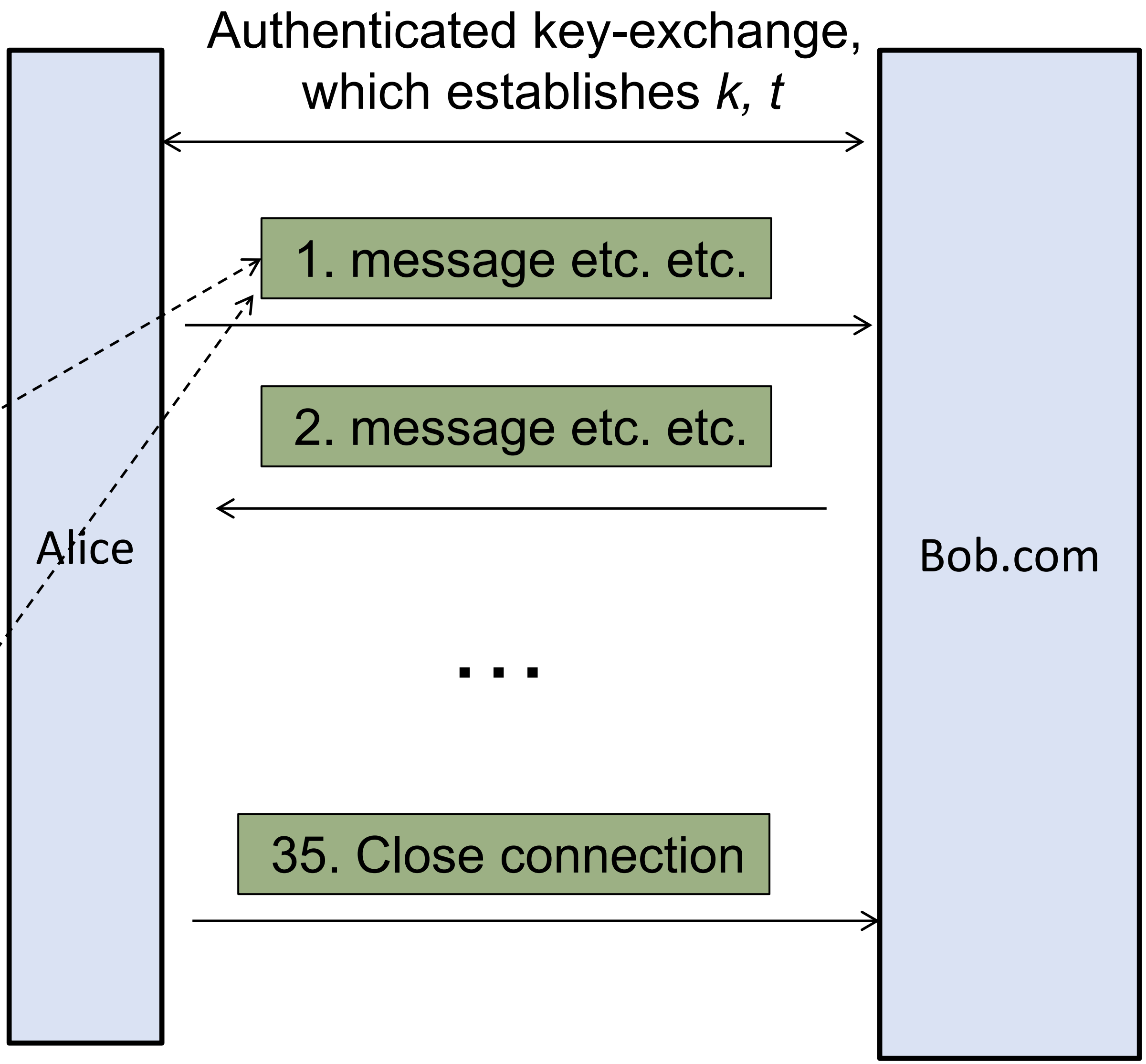
Secure Communication Channel Problem

- A common **example**:
  Imagine that Alice wants to visit a website Bob.com.
- ***Question***: How to protect the **authenticity** (that Bob.com is authentic), and **confidentiality** & **integrity** of the communication?
- ***Answer***: we need to establish a **secure channel** between Alice (i.e. her browser) and Bob.com (i.e. its web server)
- We have discussed some important necessary mechanisms: authenticated key exchange, PKI, encryption, … *(message-ordering protection?)*
- Let’s now see how we can establish a secure channel

Secure Channel between Alice and Bob.com

Confidentiality is protected by encryption using *k* as key

Authenticity is protected by MAC using *t* as key



Question: Why do we need the sequence number?

Secure Channel between Alice and Bob.com: The Steps

(Step 1)

Alice and Bob.com carry out a **unilateral authenticated key exchange** using Bob's private/public key

After authentication, both Bob and Alice know two randomly selected ***session keys k***, ***t***
where: ***k*** is the secret key of a symmetric-key encryption, e.g. AES

t is the secret key of a MAC

(Details of how k and t can be established are omitted here: *see, for instance, station-to-station protocol for details)*

Secure Channel between Alice and Bob.com: The Steps

(Step 2)

Subsequent communication between Alice and Bob.com will be protected by ***k***, ***t*** and *a* ***sequence number i***

Suppose m_1, m_2, m_3, … are the sequence of message exchanged, the **actual data** to be sent for m_i will be:

$$E_k\,(\,i\;||\,m_\mathrm{i}\,)\;||\;MAC_t\,(\,\mathrm{E_k}\,(\,i\;||\,m_\mathrm{i})\,)$$

where: i is the sequence number,

|| refers to concatenation

Note: The technique above is known as "**encrypt-then-MAC**". There are other variants of **authenticated encryption** called "MAC-then-encrypt" and "MAC-and-encrypt": *see later in this lecture*

Secure Channel and PKI Usage

- Recall that in order to carry out an authenticated key-exchange, some mechanism of **distributing public keys** is required
- Very often, **PKI** is employed to distribute the public key: the authenticated key-exchange is thus likely to involve ***certificate***
- Example (a case of unilateral authentication) :
  - Suppose **Alice** visits **Bob.com**, and wants to verify that the entity she's communicating is with indeed is **Bob.com**
  - **Alice** then needs to know **Bob.com**'s **public key**
  - Right in the beginning of the authentication protocol, **Bob.com** directly sends **its *certificate*** (which contains his public key) to Alice

5.4 Putting All Together: TLS/SSL

HTTPS & TLS/SSL Protocols

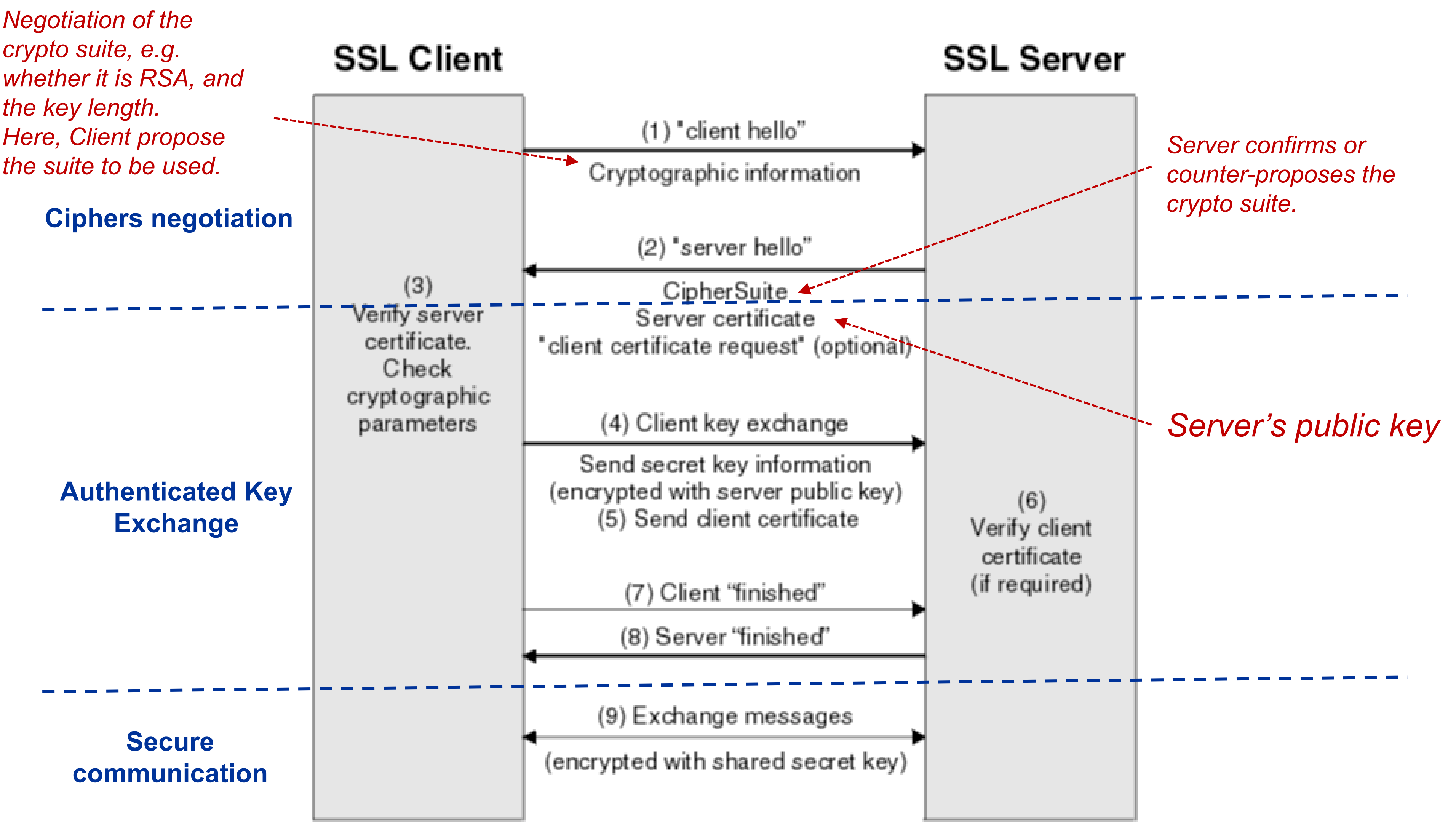
- **HTTPS** (HTTP Secure) is widely used to secure Web traffic
- HTTPS is built on top of SSL/TLS: **HTTPS = HTTP + SSL**
  Hence, HTTPS is also called: HTTP over SSL,
  or HTTP over TLS
- Transport Layer Security (**TLS**) is a protocol to secure communication using cryptographic means:
  TLS 1.2 [2008], TLS 1.3 [Aug 2018]
- **SSL** is predecessor of TLS: Netscape SSL 2.0 [1993]
- TLS/SSL adopts similar framework as in the previous part to **establish a secure communication channel**
- TLS/SSL sits in between TCP/transport & application layers

TLS Protocol (Suite)

- How does TLS work **at the high level**?
  1. **Ciphers negotiation**
  2. **Authenticated Key Exchange**: the exchange of session key, which also authenticates the identities of parties involved
  3. Symmetric-key based **secure communication**
  4. **Re-negotiation** (if needed)
- Two (sub) protocols (see https://docs.microsoft.com/en-us/windows/win32/secauthn/transport-layer-security-protocol):
  - **Handshake Protocol**: for 1, 2, 4 above; also uses the Record Protocol
  - **Record Protocol**: for 3; a lower-layered protocol

Question: Alice is in a café. She uses the free WiFi to upload her assignment to IVLE (which uses HTTPS). The café owner controls the WiFi router, and thus can inspect every packet going through the network. Can the café owner get Alice's report?

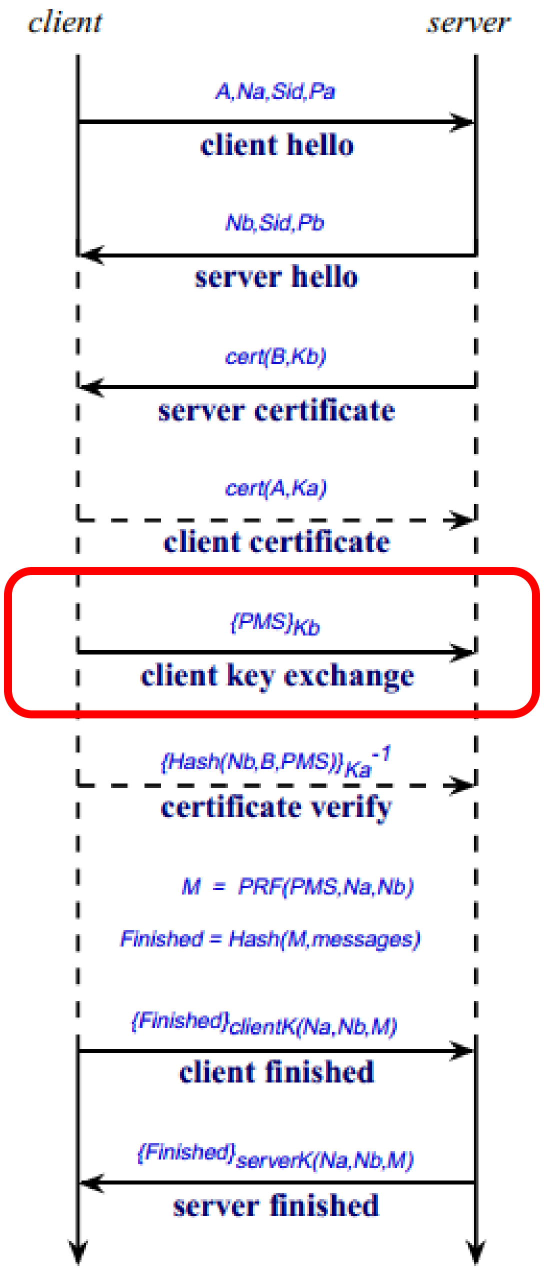
TLS Handshake (Ciphers Negotiation & Authenticated Key Exchange)



https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm

TLS Handshake (Ciphers Negotiation & Authenticated Key Exchange)

Optional



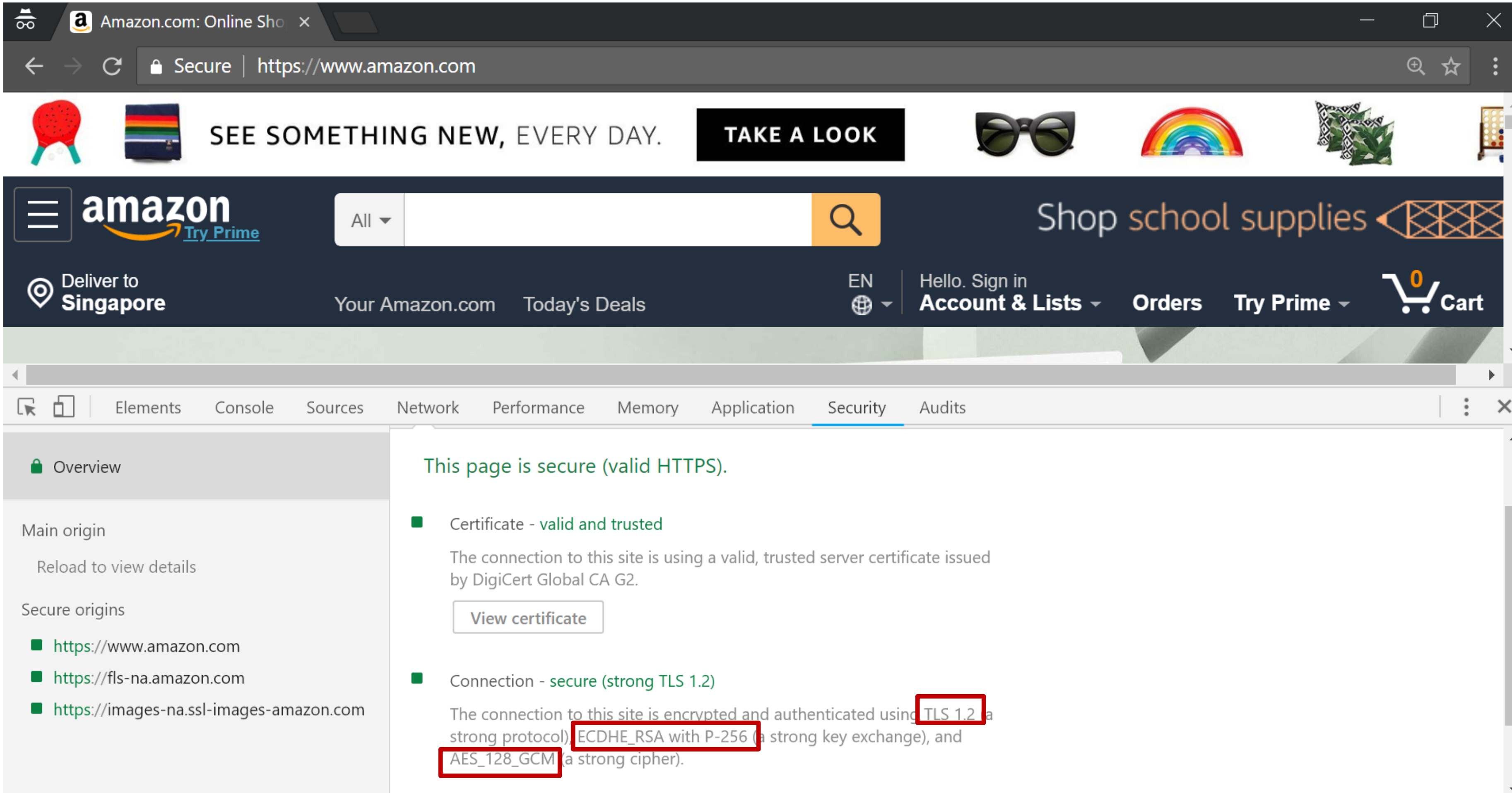
PMS can be derived using DH KE too

Notes on notation:

- *Na*, *Nb*: nonces from A & B, respectively
- *Sid*: session ID
- *Pa*, *Pb*: a set of preferences from A & B, respectively
- *PMS*: **Pre-Master Secret**
- PRF: Pseudo Random Function
- *M*: ***Master secret***

See: http://www.cl.cam.ac.uk/~lp15/papers/Auth/tls.pdf

HTTPS Protocol in Action: An Example



See https://tools.ietf.org/html/rfc5246 for the details of TLS Protocol

Additional Remarks on TLS (Optional)

Optional

- Previous **attacks** on vulnerable TLS implementations:
  - TLS 1.2 supports AES in CBC mode: vulnerable to padding oracle attack
  - Some attacks: Heartbleed (“buffer over-read”), BEAST, CRIME, POODLE
- The **latest TLS 1.3** version:
  - Ditches **insecure** schemes:
    e.g. MD5, SHA-1, RC4, AES in CBC mode
  - Ditches **unnecessary** (legacy) features:
    e.g. optional data compression, which enables the CRIME attack
  - Uses state-of-the-art **ciphers**:
    AES-GCM, AES-CCM, ChaCha20+Poly1305 authenticated ciphers
  - Adds useful protection mechanisms:
    e.g. downgrade protection
  - Performance improvement:
    e,g, session resumption

Side Remark: What is a Protocol?

Protocol Definition and Example

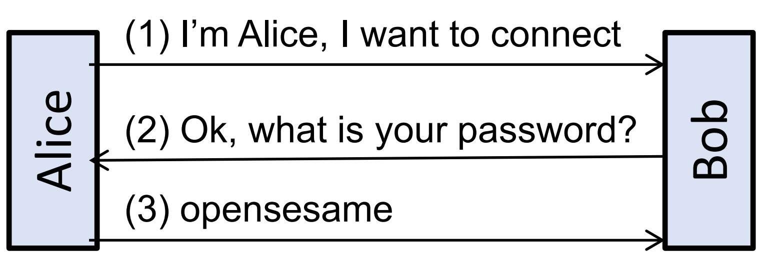
- Slides 4, 6, 12, 13, and 21 illustrate a "**protocol**"
- In computer networking, a ***protocol*** is a set of rules for exchanging information between **multiple entities**
- A protocol is often described as **steps of actions** to be carried by the entities, and the **data to be transmitted**
- For example:

  Alice → Bob: "I'm Alice, I wants to connect"

  Alice ← Bob: "Ok, what is your password?"

  Alice → Bob: "opensesame"

- It can also be visually shown as below:



5.5 Authenticated Encryption

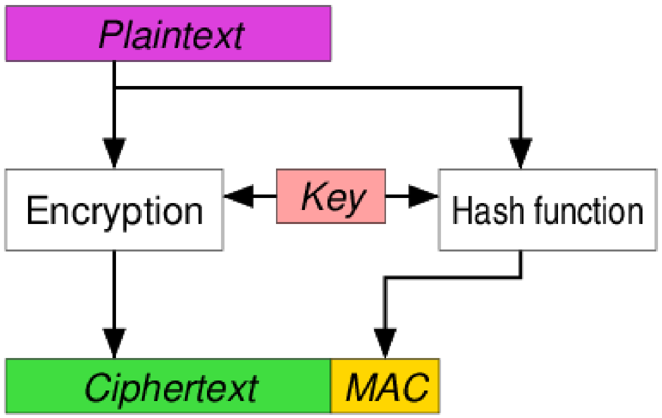
What is Authenticated Encryption?

- ***Authenticated encryption***:
  symmetric encryption that returns **both** ciphertext and authentication tag
- It combines **cipher** and **MAC**:
  ensures message confidentiality and authenticity
- **Authenticated encryption** process: $AE(K_{AB}, M) = (C, T)$
- **Decryption** process: $AD(K_{AB}, C, T) = M$ **only if *T* is valid**
- Different **variants/approaches**:
  - Encrypt-and-MAC (E&M)
  - MAC-then-Encrypt
  - Encrypt-then-MAC
  - Specialized authenticated cipher

Encrypt-and-MAC (E&M)

- The sender computes the ciphertext C and tag T ***separately***
- It performs **encryption**, e.g. using 2 keys $K_{1_{AB}}$ and $K_{2_{AB}}$ as follows:
  - $C = E(K_{1_{AB}}, M)$
  - $T = \mathrm{MAC}(K_{2_{AB}}, M)$
- It finally sends $\boldsymbol{(C, T)}$

Illustration with 1 key
(Source: Wikipedia)

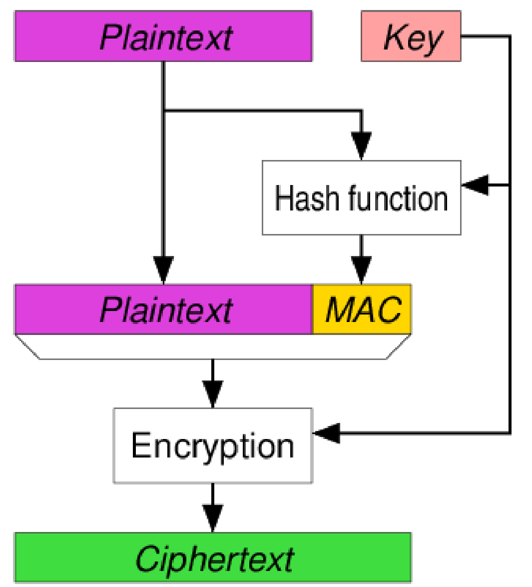


- It is used in **SSH** (with a strong MAC like HMAC-SHA-256)
- Issue: T may not be random looking, and could leak information

MAC-then-Encrypt (MtE)

- The sender first computes the **tag** $\boldsymbol{T} = \mathrm{MAC}(K_{2_{AB}}, M)$
- It then **generates** the ciphertext $\boldsymbol{C} = E(K_{1_{AB}}, M \,||\, T\,)$
- It finally sends $\boldsymbol{C}$

Illustration with 1 key
(Source: Wikipedia)

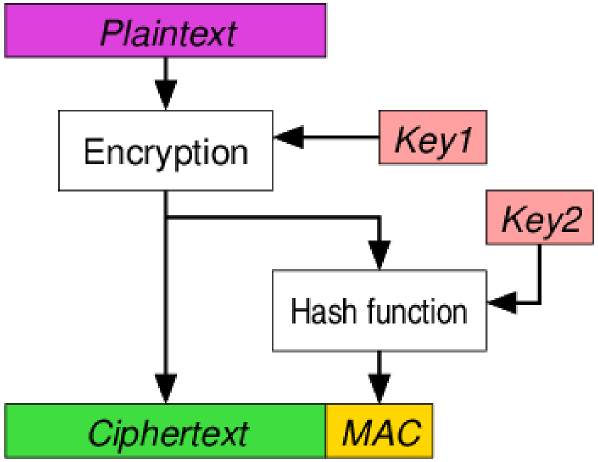


- It is used in **SSL** and **TLS** up to version 1.2:
  the latest TLS v1.3 uses authenticated ciphers, e.g. AES-GCM
- Issue: a decryption is still needed on a corrupted message

Encrypt-then-MAC (EtM)

- The sender first **generates** the ciphertext $\boldsymbol{C} = E(K_{1_{AB}}, M)$
- It then computes the **tag** $\boldsymbol{T} = \mathrm{MAC}(K_{2_{AB}}, C)$
- It finally sends **($\boldsymbol{C}$, $\boldsymbol{T}$)**

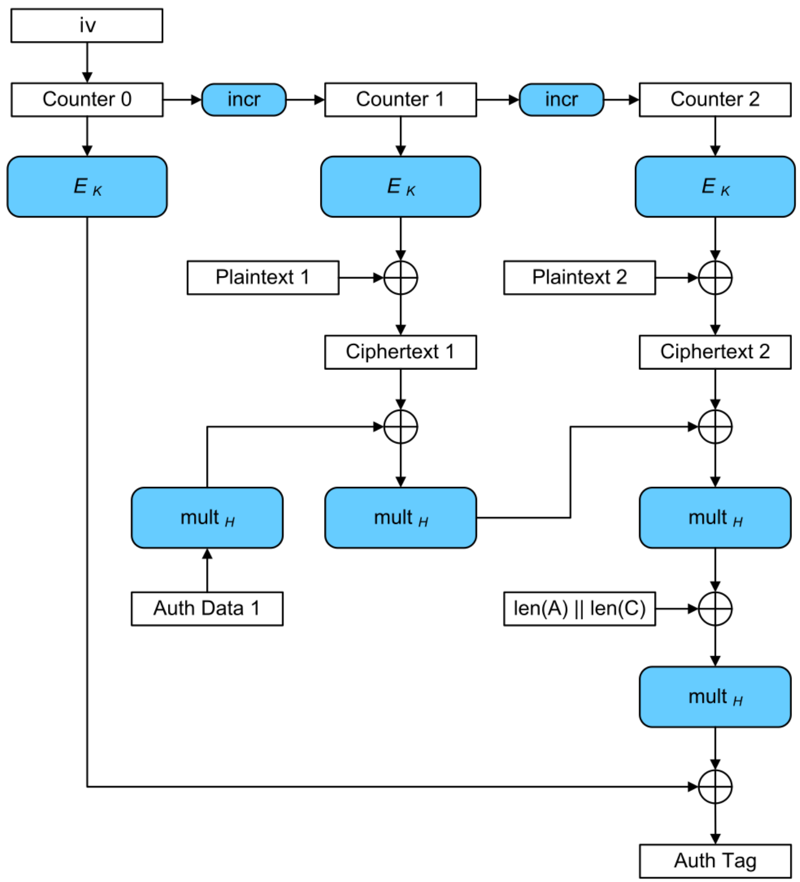
Illustration with 2 keys
(Source: Wikipedia)



- It is used in **IPsec**
- Feature: a decryption is **not** performed on a corrupted message

Authenticated Cipher

- It returns an authentication tag **together with** the ciphertext
- An example is **AES-GCM** (AES in the Galois counter mode):
  - One authenticated cipher in TLS version 1.3
  - The most widely used authenticated cipher

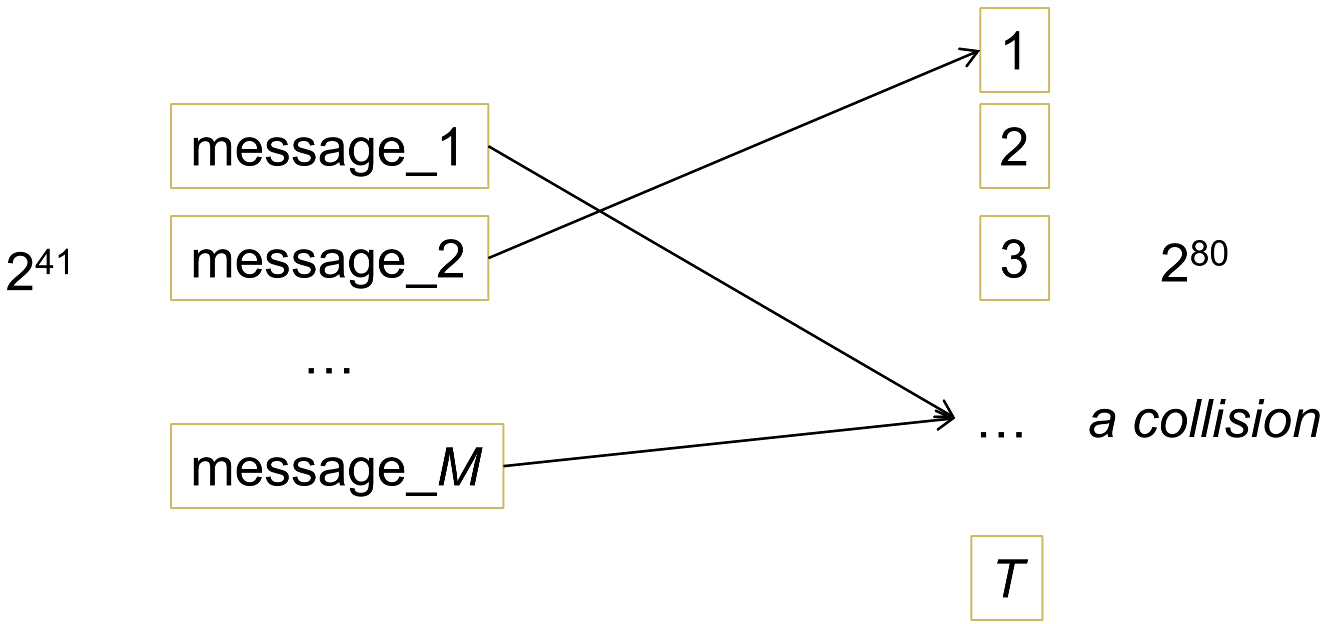


Source: Wikipedia

5.6 Birthday Attack Variant

(The Standard) Birthday Attack on Hash

- Suppose the digest of a hash is 80 bits: $T = 2^{80}$
- Now, an attacker wants to find a collision
- If the attacker randomly generates 2^{41} messages ($M=2^{41}$), then $M > 1.17\, T^{0.5}$
- Hence, with probability more than 0.5, among the 2^{41} messages, ***two of them*** *give the same digest*!



In general, the probability that a collision occurs ≈ $1 - \exp(-M^2/(2T))$

A Variant of Birthday Attack

- A **variant** of birthday attack is shown below
- Let *S* be a set of ***k*** distinct elements,
  where each element is a ***n***-bit binary string
- Now, let us independently and randomly select ***m***
  n-bit binary strings: you call this set *M*, with |M|<|S|
- It can be shown that, the probability that at least one of the randomly chosen strings is in *S* is (larger than):

$$1 - 2.7^{-km2^{-n}}$$

- Notice that the set *S* and the set *M* are *different*
- *How can we visualize this setting?*

5.7 Remarks on Security Reduction

Comparing the Security of Two Systems/Schemes

- Tutorial 4 (Q8 on "Security requirements of a cryptographic hash function") illustrates the notion of ***security reduction***
- In a security reduction, there are two security requirements: e.g. *A* (one way) and *B* (collision-resistant)
- In the analysis of reduction, we want to prove the following **3 equivalent statements**

For any system *S*,

there exists an attack on *S* that breaks requirement *A*	$\Rightarrow$	there exists an attack on *S* that breaks requirement *B*

For any system *S*,

there does not exist attack on *S* that breaks requirement *A*	$\Leftarrow$	there does not exist attack on *S* that breaks requirement *B*

For any system *S*,

the system *S* achieves requirement *A*	$\Leftarrow$	the system *S* achieves requirement *B*

Comparing the Security of Two Systems/Schemes

From our Tutorial 4's question:

- S = hash function $h()$
- Requirement A = one way
- Requirement B = collision-resistant
- Objective: to prove the following **3 equivalent statements**

For any hash function $h()$,		
There exists an attack that invert $h()$	$\Rightarrow$	There exists an attack that finds collision of $h()$

For any hash function $h()$,		
There does not exist an efficient way to invert $h()$	$\Leftarrow$	There does not exist an efficient way to find a collision

For any hash function $h()$,		
$h()$ is one-way	$\Leftarrow$	$h()$ is collision resistant

Existence of One-Way Function and Other Claims

- *Question: Does a one-way function really exist? How about SHA-3?*
- In this module, we always use the term “*we believe*”, “*there is no known attacks*”, “*finding something is computationally difficult*”, such as:
  - We believe that SHA-3 is collision resistant
  - There is no known efficient attack on AES
  - Finding the private key from public key is difficult if factorization is difficult
- Why not commit ourselves and give a definite statement like below?
  - SHA-3 is collision resistant!!!
  - It is difficult to find the private key from public key!!!!
- Security of many crypto primitives **relies on** the ***assumption*** that one-way function exists:
  - Unfortunately, there is ***no known*** mathematical proof on the existence of one-way functions
  - Nevertheless, **many people believe** that indeed one-way function exists
- So, instead of claiming “*SHA-3 is collision resistant*”, a mathematically rigorous claim is “*SHA-3 is **believed to be** collision resistant*”

5.8 Other Interesting Cryptography Topics

Interesting Types of Encryption

- **Format-preserving encryption:**
  - A basic cipher **does not** care if, for instance, a plaintext is an image
  - The ciphertext is **not** a viewable image
  - A format-preserving encryption solves this issue:
    ciphertexts have ***the same format*** as the plaintexts
  - Other possible target **plaintext types**:
    IP address, ZIP code, credit card numbers
- **Fully-homomorphic encryption**:
  - Preserves the mathematical structure between plaintext & ciphertext spaces
  - Enables its user to replace a ciphertext $\boldsymbol{C = \mathrm{E}_K(M)}$ with another ciphertext $\boldsymbol{C' = \mathrm{E}_K(F(M))}$, where $F()$ is as a function of M, **without ever decrypting** the initial ciphertext C
  - Example: M is a text document, $F()$ is a modification of part of the text
  - It's very useful for a **cloud provider**:
    it doesn't know the plaintext/data, but **can change** the data as requested by the data owner (on the owner's behalf)
  - It is still very **slow**: a basic operation needs an unacceptably long time!

Optional

5.9 Time-Memory Tradeoff for Dictionary Attack

Reference:

See the "*Precomputed hash chains*" Section of: http://en.wikipedia.org/wiki/Rainbow_table

The above Wiki page describe "Rainbow table", which is an improved variant of time-memory tradeoff. The original basic variant is described in the section "Precomputed hash chain".

"Inverting" a Hash Digest in Real-Time

- Suppose a hash *H*() is **collision resistant**: it is also one-way
- Thus, given a digest *y*, it is **difficult** to find a *x* s.t. $H(x)=y$

- Suppose we know that *x* is chosen from a **relatively small set** of *dictionary* *D*
- For illustration, assume *x* is a randomly & uniformly chosen **50-bit** message

- Now, even if *H*() is one-way, given the digest *y*, it is still feasible to find a *x* in *D* s.t. $H(x)=y$
- One method of "inverting": **exhaustively search** 2^{50} messages in *D*

- Although feasible, this would take **days** of computing time
- As the attacker, we want to **speed up** the inverting process to support a "real-time" attack

Speeding up Using a Large Table

- Supposed we are allowed to perform a **pre-processing**
- Let's view the 50 bits message as an **integer**

- One naive method is to build a dataset with 2^{50} elements:

  $$(H(x), x) \quad \text{for } x = 0, 1, 2, \ldots, 2^{50}-1$$

  and store these elements in a data structure ***T*** that supports a **fast lookup** (e.g., a **hash table** that facilitates a constant-time lookup)

- Now, given a digest y, we can **query** the data structure and readily find the associated x

- **Issue**: such table is too large: 2^{50} entries = 2^{10} "Tera" entries
- **Solution**: the time-memory tradeoff is a technique that "trades off" time for memory, e.g. a **lower lookup time** for a **higher storage**

Time-Memory Tradeoff (TMTO)

- The main idea: use a precomputed ***hash-chain***
- (Note: the term "hash-chain" appears in different context and refer to different techniques)

- Define a ***reduce function*** R() that maps a digest y to a word w in the dictionary ***D***
- For illustration, if ***D*** consists of all 50-bit messages, and each digest is 320 bits, then a possible reduce function simply **keeps the first 50 bits** of input:

$$R(b_0 b_1 \ldots b_{320}) = b_0 b_1 \ldots b_{49}$$

- Note that $R()$ is clearly ***not*** an inverse of $H()$

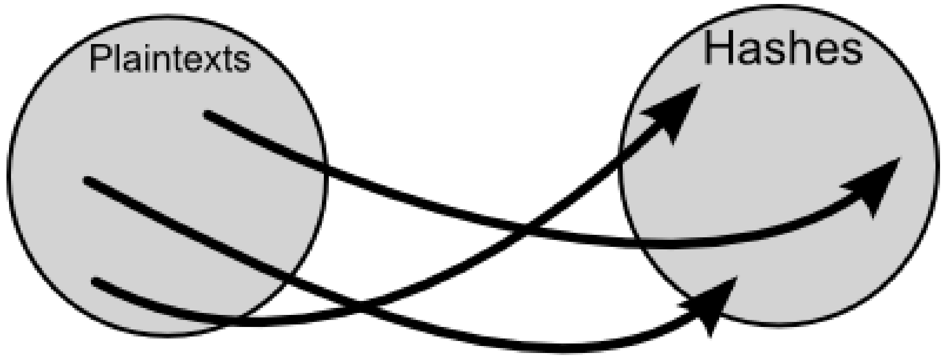
- Here is a **pre-computed hash chain**, which starts from a randomly-chosen word w_0 in ***D***

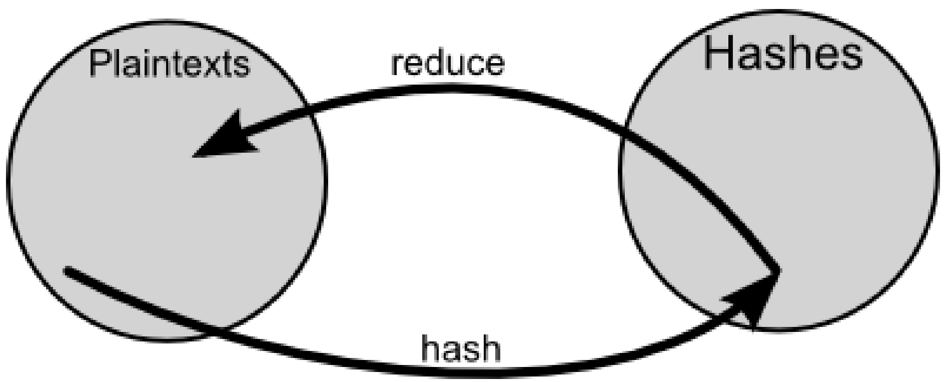
$$w_0 \rightarrow y_0 = H(w_0) \rightarrow w_1 = R(y_0) \rightarrow y_1 = H(w_1) \rightarrow w_2 = R(y_1) \rightarrow y_2 = H(w_2)$$

E.g.:

"hello" → **A0C0…20** → "qwert1" → **03F0…50** → "Pikachu" → **77FF…3A**

Hash *H*() and Reduce *R*(): Illustration





Source: http://kestas.kuliukas.com/RainbowTables/

Building Pre-Computed Hash Chain Dataset

$$w_0 \rightarrow y_0 = H(w_0) \rightarrow w_1 = R(y_0) \rightarrow y_1 = H(w_1) \rightarrow w_2 = R(y_1) \rightarrow y_2 = H(w_2)$$

$$w_0 \rightarrow \boxed{H} \rightarrow y_0 \rightarrow \boxed{R} \rightarrow w_1 \rightarrow \boxed{H} \rightarrow y_1 \rightarrow \boxed{R} \rightarrow w_2 \rightarrow \boxed{H} \rightarrow y_2$$

- The ***dataset-building steps*** are as follows:
  1. Select a randomly-chosen word $\boldsymbol{w_0}$ in *D*
  2. Create the **hash chain** for w_0 as shown above
  3. Call w_0 the **starting-point**, and y_2 the **ending-point**
  4. Store the pair **($\boldsymbol{w_0}$, $\boldsymbol{y_2}$)** in the data-structure ***T***
  5. Repeat the process with other randomly-chosen starting points

Querying Pre-Computed Hash Chain Dataset (1/2)

$$w_0 \rightarrow y_0 = H(w_0) \rightarrow w_1 = R(y_0) \rightarrow y_1 = H(w_1) \rightarrow w_2 = R(y_1) \rightarrow y_2 = H(w_2)$$

$$\boldsymbol{w_0} \rightarrow \boxed{H} \rightarrow y_0 \rightarrow \boxed{R} \rightarrow w_1 \rightarrow \boxed{H} \rightarrow y_1 \rightarrow \boxed{R} \rightarrow \boldsymbol{w_2} \rightarrow \boxed{H} \rightarrow \boldsymbol{y_2}$$

- Given a digest y, first search for y in the data-structure $\boldsymbol{T}$
- Suppose y is in $\boldsymbol{T}$:
  - That is, y is one of the ending-points stored
  - Let's assume that w_0 is the corresponding **starting point**: hence $y = y_2$ in the above chain
  - A pre-image of y is $\boldsymbol{w_2}$, but we don't know w_2 at this point
  - Nevertheless, the fact that y_2 is the ending-point implies that w_2 is **within** the chain starting from w_0
  - We can construct the chain $w_0, y_0, w_1, y_1, w_2, y_2$
  - When the process hits y_2, we have found the required w_2

Querying Pre-Computed Hash Chain Dataset (2/2)

$$w_0 \rightarrow y_0 = H(w_0) \rightarrow w_1 = R(y_0) \rightarrow y_1 = H(w_1) \rightarrow w_2 = R(y_1) \rightarrow y_2 = H(w_2)$$

$$\boldsymbol{w_0} \rightarrow \boxed{H} \rightarrow y_0 \rightarrow \boxed{R} \rightarrow \boldsymbol{w_1} \rightarrow \boxed{H} \rightarrow \boldsymbol{y_1} \rightarrow \boxed{R} \rightarrow w_2 \rightarrow \boxed{H} \rightarrow \boldsymbol{y_2}$$

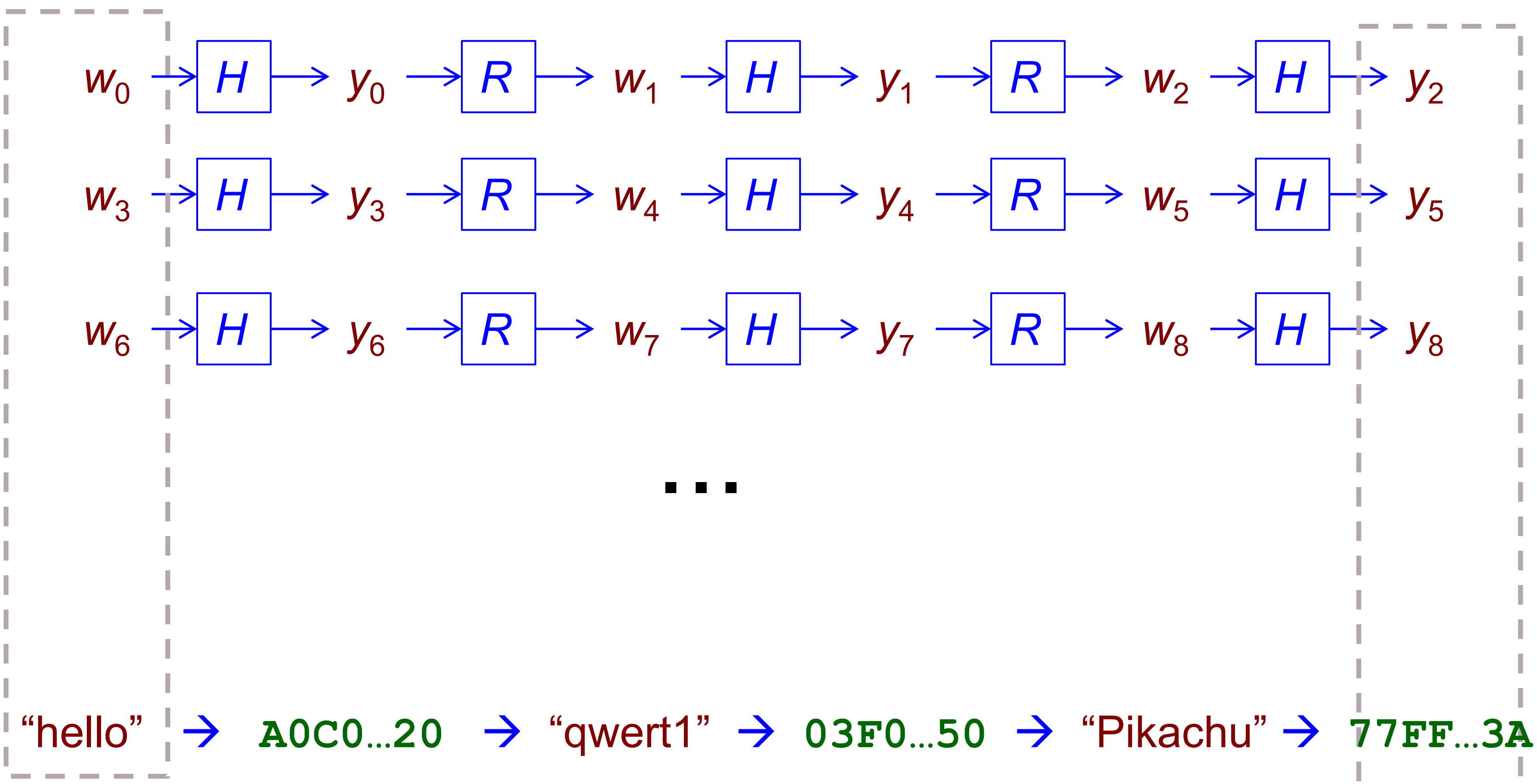
- Suppose y is ***not*** in ***T***:
  - Compute $y' = H(R(y))$
  - Search the data-structure for y'
  - Suppose y' is in ***T***:
    - Let's assume that the starting-point is w_0 (hence $y' = y_2$)
    - With high chances (it's not certain due to possible collisions), $y = y_1$
    - So, a pre-image of y is w_1, i.e. $H(w_1)=y$
    - At this point, we don't know w_1
    - Constructing the chain from w_0, and see if w_1 can be found, otherwise skip *(due to a collision issue described next)*
  - If y' is not in ***T***, compute $y'' = H(R(y'))$ and repeat this query process

Without Hash Chains (As in Tutorial 1)

Given an input *y*, find *w* such that *H*(*w*) = *y*
by building a **full** lookup table

With Hash Chains (for Tutorial 1)

Given an input y, find w such that $H(w) = y$
by storing **hash-chains**



Analysis of Time and Space Required

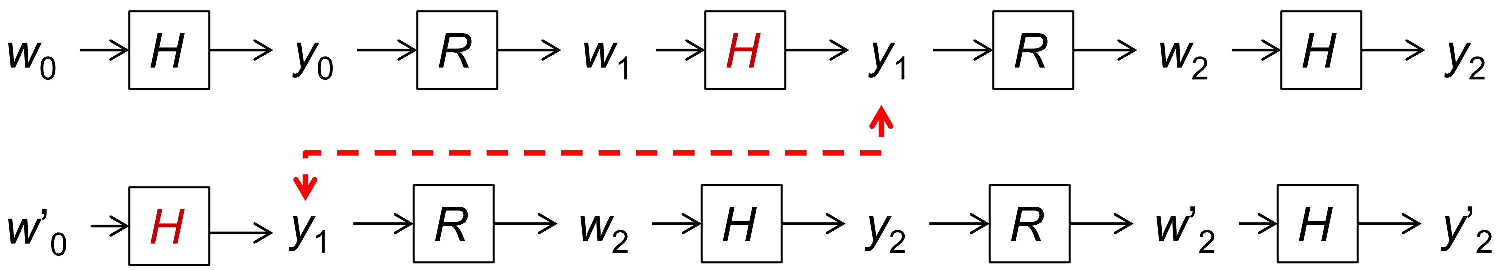
- Let's **compare** the required space & time of querying stored hash chains with the naive full table method:
  - **Space**: A reduction of space **by a factor of 3** (why?)
  - **Time per query**: the number of hash operations increases by a factor of 3, but also require 2 reduce operations (why?)
  - **Accuracy:** The chains can contain repetitions (***why?***)

- A **general rule** (where *k* is a parameter):
  we can choose the length of the chain so that the **reduction of space** is a factor of *k*, with the **increase of search time penalty** by a factor of *k*

- In our **50-bit** example:
  - The total number of entries in the full "virtual table" is 2^{50}
    (as we shall see later, these 2^{50} entries are not unique)
  - Suppose we choose $\boldsymbol{k=2^{15}}$
  - The hash-chain storage is reduced to 2^{35} entries; whereas the query time increases to 2^{15} hash operations, which can still be computed in real-time

Remark: Collisions due to Hash Function (Rare/Unlikely)

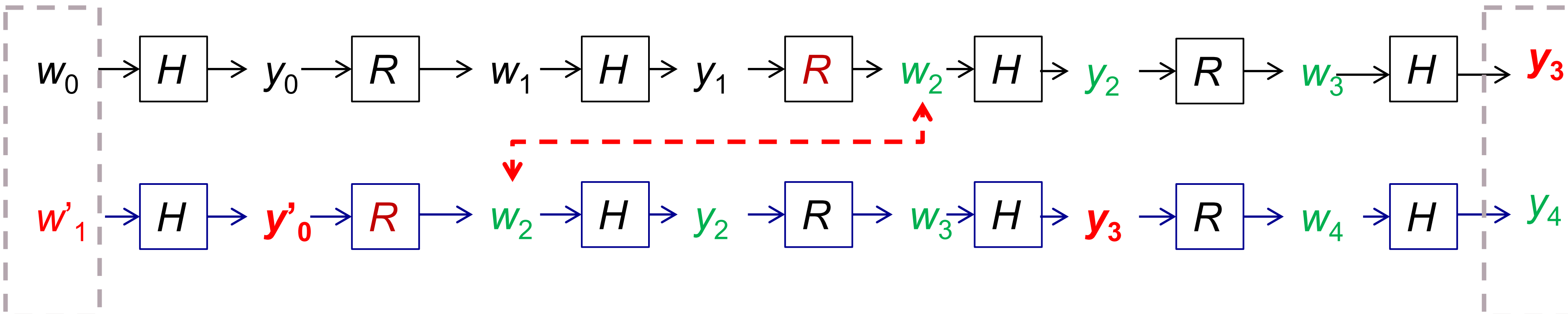
- The following **collision** is due to *H*():



- It is extremely **unlikely**, since we assume that *H*() is "secure", i.e. a strong hash function
- This type of collision can thus be **omitted** in our design consideration

Remark: Collisions due to Reduce Function (Frequent/Likely)

- Collisions due to the **reduce function** may happen **frequently**, i.e. two different digests being mapped to the same word

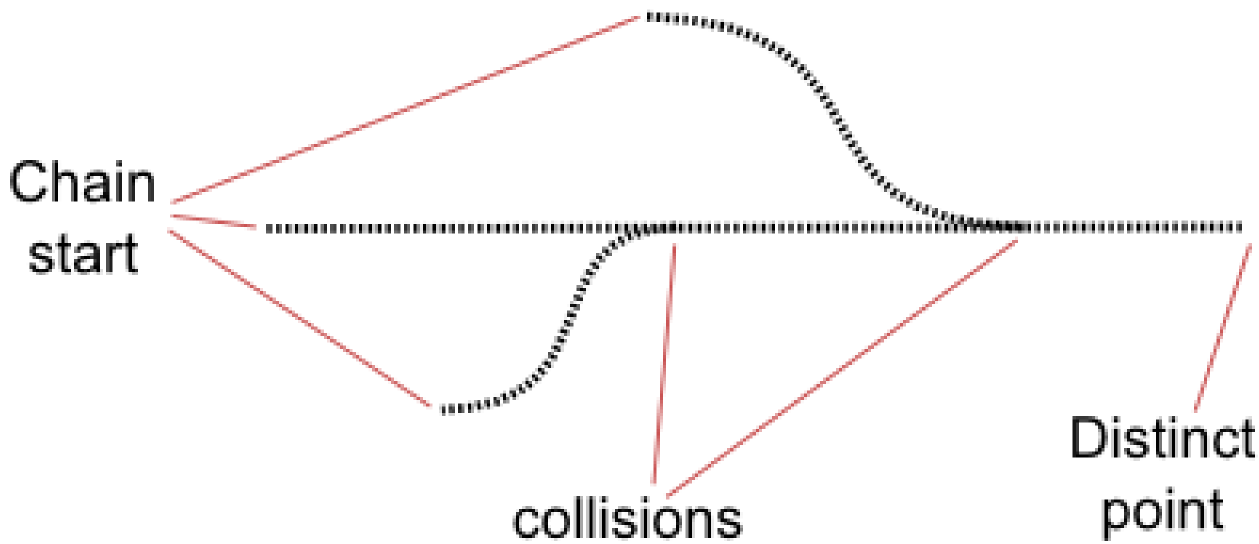


When given y_0', the algorithm is unable to find w'_1 in the first chain since:
The querying algorithm initially performs these steps:
(1) lookup y_0', (2) lookup y_2 (3) lookup y_3 (4) search the chain starting from w_0

This causes two issues:

- Inefficiency in **storage**: part of the chain is **duplicated**, e.g. w_2 and w_3 are stored twice
- Inefficiency in **search**: it leads to searches in the **wrong chains**, before hitting the right chain, e.g. for querying y_0', the lookup process would transverse **both chains**

Collisions and Hash-Chain Merges: Illustrated



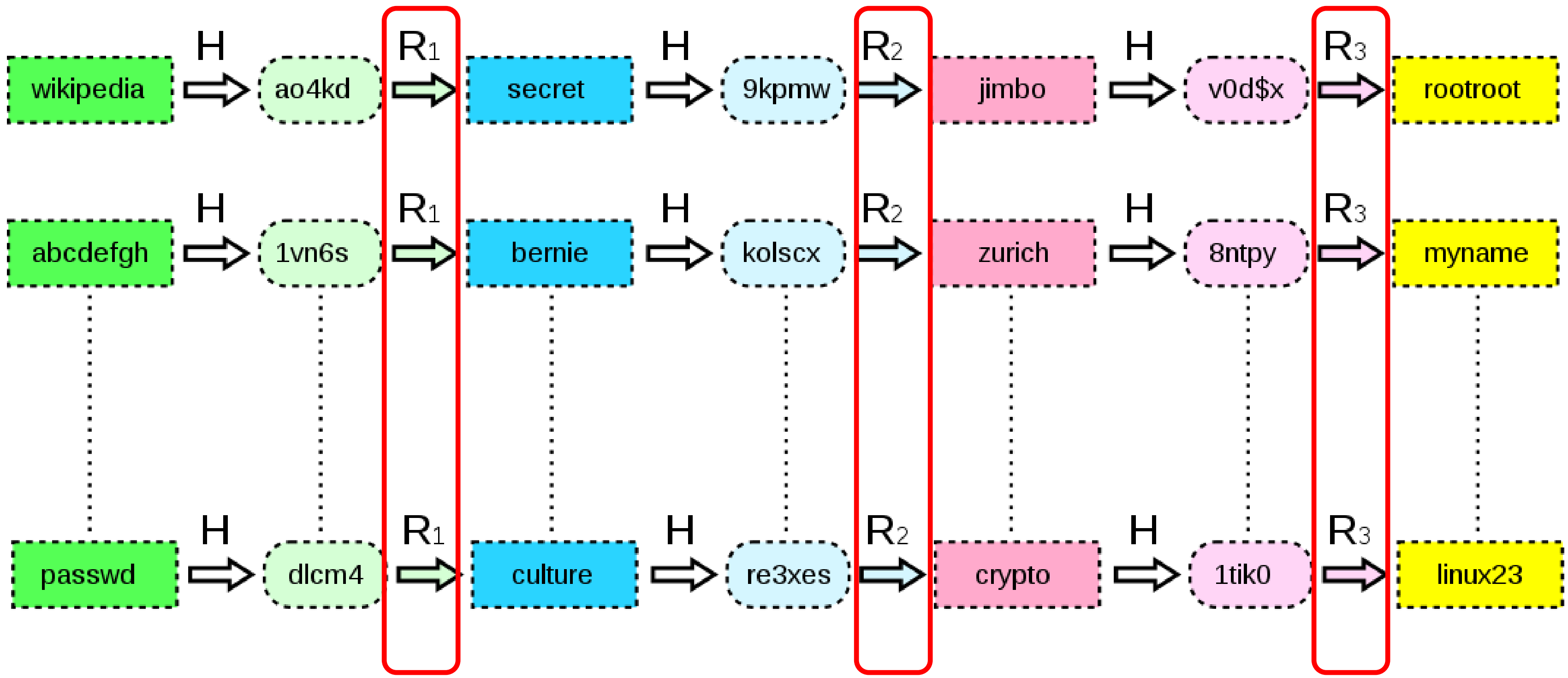
Source: http://kestas.kuliukas.com/RainbowTables/

Improved Variant: Rainbow Table

- ***Rainbow table*** gives a simple but **effective method** to address the collision issue in time-memory trade-off (the method is simple, but its analysis is quite complex)
- Rainbow table utilizes ***multiple*** **"reduce" functions**
- Details are not included in this module

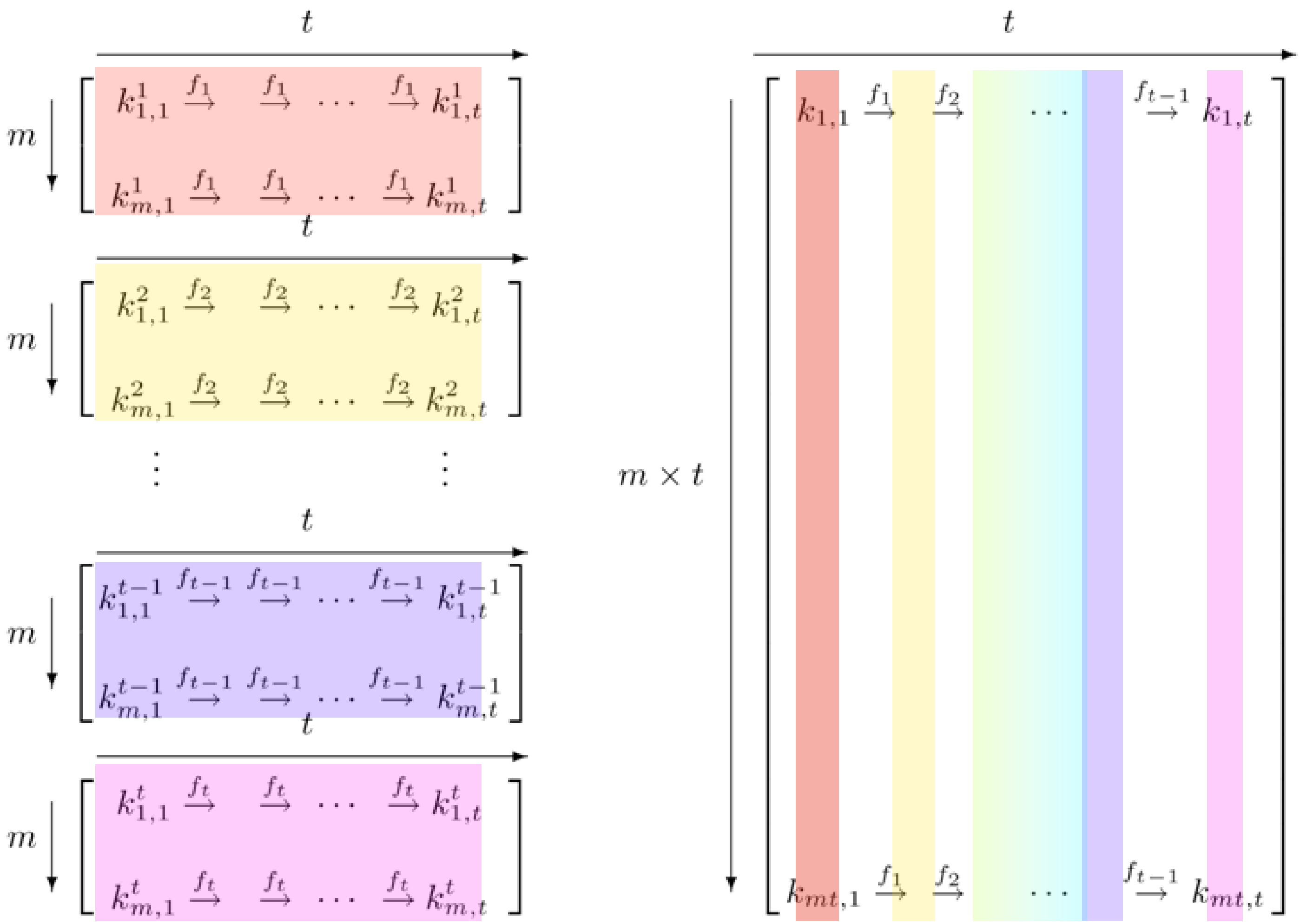
- To find out more, see:
  - http://en.wikipedia.org/wiki/Rainbow_table
  - http://kestas.kuliukas.com/RainbowTables/
- The original research paper:
  P. Oechslin, *Making a Faster Cryptanalytical Time-Memory Trade-Off*, CRYPTO 2003
  http://lasec.epfl.ch/~oechslin/publications/crypto03.pdf

Improved Variant: Rainbow Table



Source: https://cyberhoot.com/cybrary/rainbow-tables/

Improved Variant: Rainbow Table



Source: Wikipedia

5.10 Summary of Cryptography

Cryptography: Summary & Takeaways

- The **main objectives**:
  - To learn how cryptographic schemes and primitives work
  - To learn how to **use them correctly**
  - To learn how to reason about their security
- What cryptography **provides**?
  - It provides many useful primitives
  - It serves as the basis for many security mechanisms
  - It is possible to build a **secure channel** (confidentiality, integrity, authenticity) over an insecure underlying communication channel
- However, cryptography:
  - Is **not** the solution to all security problems: network security issues (to be discussed next week), software vulnerabilities, social engineering attacks, etc.
  - Needs to be implemented and deployed **securely/properly**
  - Is not something you should invent/design yourself

Importance of Cryptography? It was *Once* a Munition!

Source: Wikipedia