ABERDEEN 2040

Network Security Technology

Hash functions and message authentication codes

September 2025

Outline of lecture

1. Properties of hash functions.
2. Attacks on hashes.
3. Construction of hash algorithms.
4. Real hash algorithms.
5. Keyed hash functions.
6. Message authentication codes.

Hash functions in general

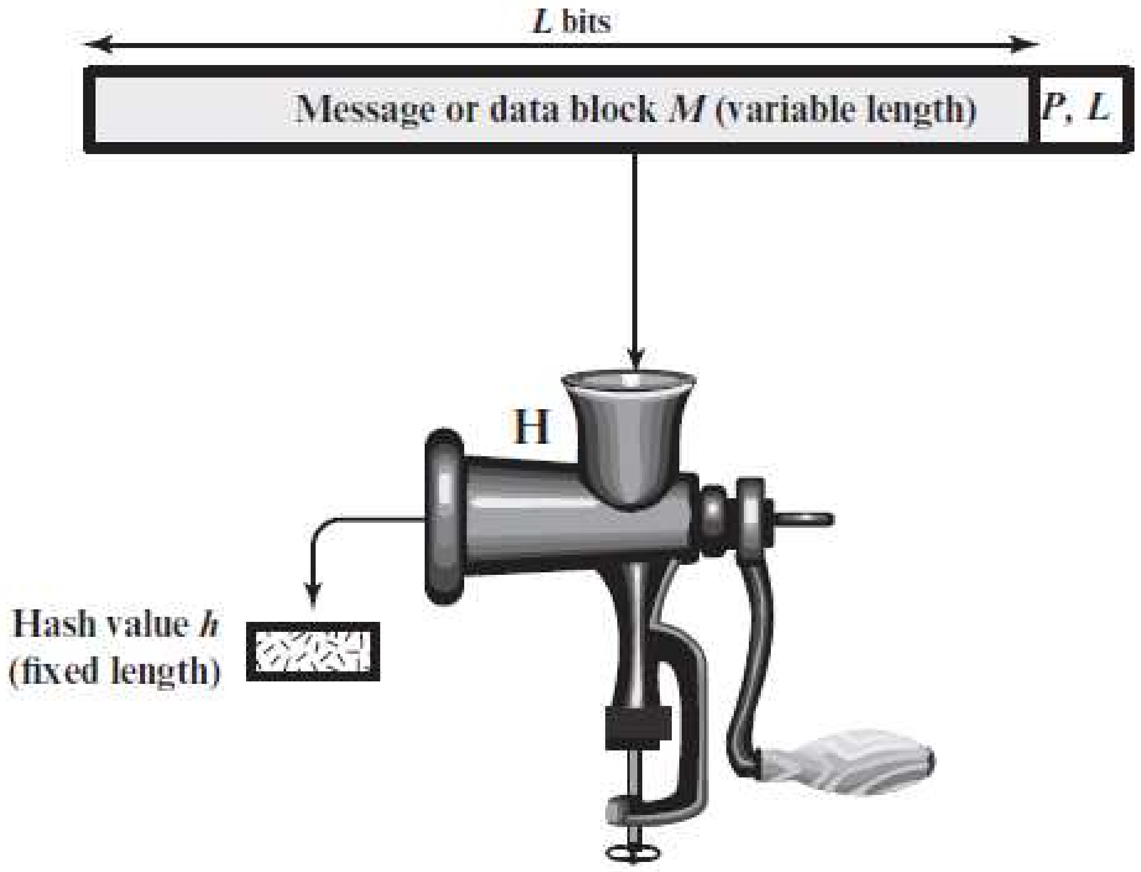
1. Used in CS for a long time, not just security.
2. A good hash function *h* on bit strings has several properties:
   1. The results of applying the function to a large set of inputs will produce outputs that are **evenly distributed** and **apparently random**.
   2. Ease of computation: given *x*, it is easy to compute the hash value *h*(*x*)
   3. Compression: *h* takes inputs of any (variable) length, and maps to outputs of some fixed length *L*, i.e, $h : \{0,1\}^* \rightarrow \{0,1\}^L$
3. In general terms, the objective of a hash function is to provide data integrity.
4. A change to any bit or bits in *M* results, with high probability, in a change to the hash value.

Applications of hash functions

1. They are building blocks for many cryptographic protocols and security systems.
2. The various applications determine the properties that are required of the hashes.
3. For example, integrity checks:
    1. Widely use to provide some assurance of the integrity of data.
    2. If *x* supplied and a hash value *v* supplied in some secure way, one can check *h*(*x*) = *v* and be confident that *x* is as desired.
    3. If a hash function has no key, then anyone can perform the verification.

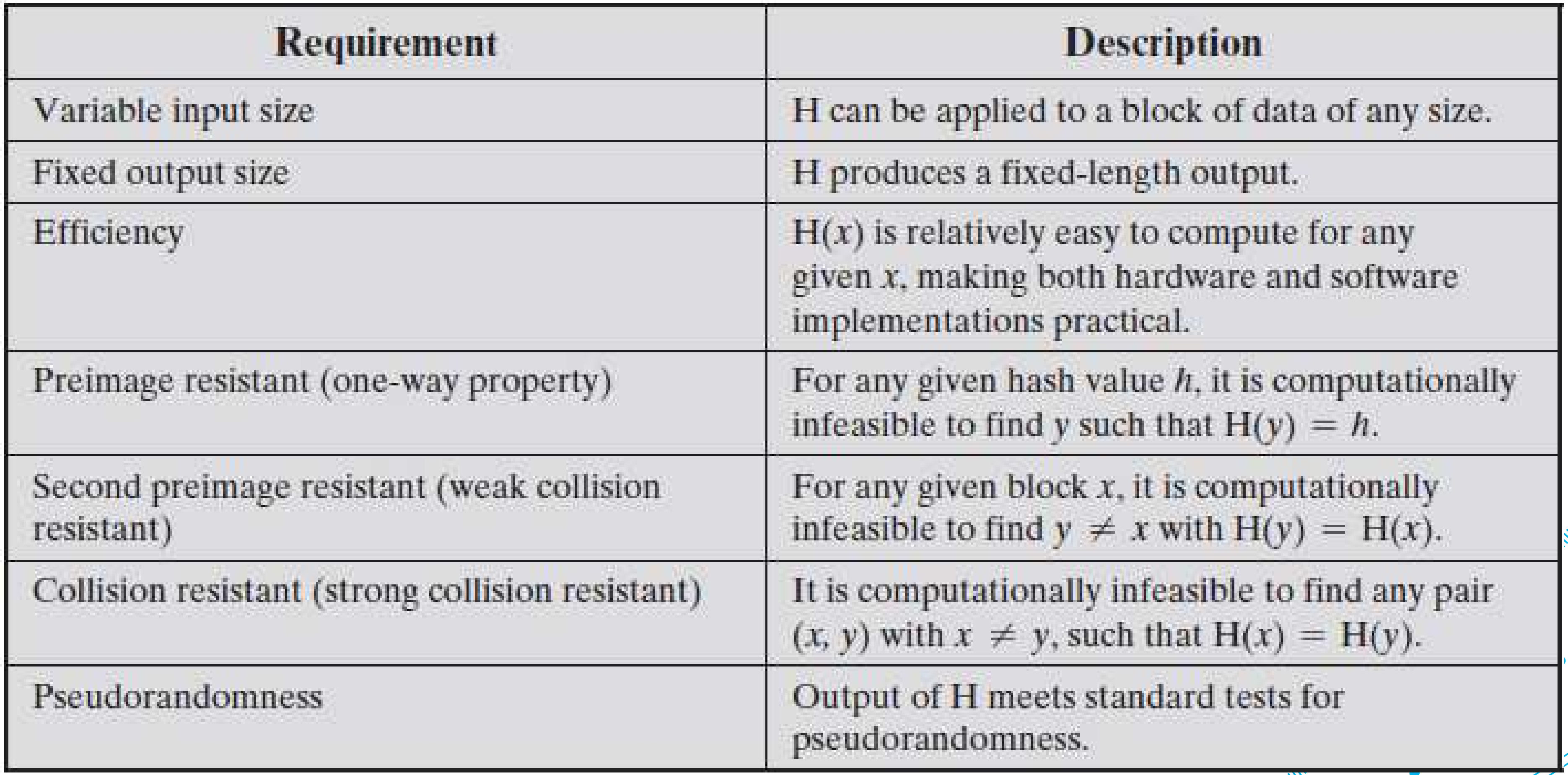
General operation of a cryptographic hash function

1. The input is padded out to an integer multiple of some fixed length (e.g., 1024 bits).
2. The padding includes the value of the length of the original message in bits.
3. The length field is a security measure to increase the difficulty for an attacker to produce an alternative message with the same hash value.


L bits
Message or data block M (variable length)
P, L
H
Hash value h (fixed length)

P, L = padding plus length field

Properties of hash functions



Requirement	Description
Variable input size	H can be applied to a block of data of any size.
Fixed output size	H produces a fixed-length output.
Efficiency	$H(x)$ is relatively easy to compute for any given x, making both hardware and software implementations practical.
Preimage resistant (one-way property)	For any given hash value h, it is computationally infeasible to find y such that $H(y) = h$.
Second preimage resistant (weak collision resistant)	For any given block x, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$.
Collision resistant (strong collision resistant)	It is computationally infeasible to find any pair (x, y) with $x \neq y$, such that $H(x) = H(y)$.
Pseudorandomness	Output of H meets standard tests for pseudorandomness.

Pre-image resistant property

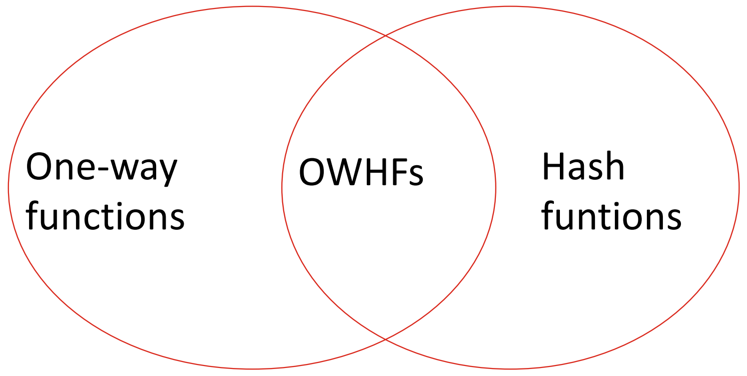
1. For a given hash value *y*, any input *x* such that *h(x) = y* is called a **pre-image** of *y*.
2. A hash function is **pre-image resistant** if, for any given hash value, it is hard to find (compute) an input that yields it, given only the hash value.
3. A one-way property: it is easy to generate a code given a message, but virtually impossible to generate a message given a code.
4. If the hash function is not one way, an attacker can easily discover the secret value.

One-way functions

1. A function *h: X* → *Y* is **one-way** if it:
   1. There is an efficient algorithm for computing the value *h(x)* for any given *x* in *X*, and
   2. It is preimage resistant: there is no efficient algorithm that finds a preimage for *y in Y*. That is to say, we don't know how to find *x* such that *h(x) = y*.
2. Notes
   1. Sometimes don't want it to be too efficient.
   2. The `*no*' in the last point is a bit subtle. It means something like `*no known*'. Sometimes it means `*no known, and there is reason to believe there is none, given what people currently think about computational complexity*'.

One-way hash functions

1. A **one-way hash function (OWHF)** is a hash function that is preimage resistant.



Applications of one-way hash functions

1. Integrity checks on files and programs:
    1. Distribute a file, *x*, and publish its **hash value** (**message digest**, **checksum**), $h(x)$.
    2. If *x* is modified to some *x'*, then it will (for a good hash function) give a new $h(x') \neq h(x)$.
    3. So, the value of the hash will not match.
    4. The algorithm computing *h* is public.
    5. No secret key is required.
    6. In this usage, hashes are also known as modification detection codes, message integrity codes.
    7. Can think of hash value as being like a digital fingerprint.

collisions

1. A **collision** is a pair of inputs *x* and *x′* such that *h(x) = h(x′)*.
2. Apply the *pigeonhole principle* to hashes:
    1. There are more strings of variable length than there are strings of the given fixed length *L*.
    2. Each string gets a (unique) hash value.
    3. Thus, some input strings must get the same hash value.
    4. There must be some collisions.

Collisions for good hashes

1. Hash value should be a kind of fingerprint of the original (pre-image) value.
2. It should be *unlikely* that there is any other given input value hashes to the same value.
    1. Given *x* and *x'*, it should be very unlikely that *h(x)* = *h(x')*.
    2. *L* needs to be big enough to allow for this.
3. There should not be too many collisions. If it had absolutely no collisions, then it would not really be hashing.
4. A hash function does not absolutely guarantee that a particular hash value came from a particular input; it can only give a very high-degree of confidence.

Collision-resistance and pre-image resistance

1. A hash is **collision-resistant** if it is hard to find any two inputs with the same hash value:
   1. They must exist, but they are hard to search for by computational and other cryptological means.
   2. The (awful) terminology **collision-free** is also common.
2. Recall: a hash is **pre-image resistant** if, for any given hash value, it is hard to find an input that yields it, given only the hash value**.**
3. Note the difference between computational problems:
   1. finding <u>any two</u> *x* and *x′* that collide, and
   2. finding an *x′* that collides with a <u>give</u>n *x (*with hash value *h(x)).*

Second pre-image resistant property

1. It guarantees that it is infeasible to find an alternative message with the same hash value as a given message.
2. This prevents forgery when an encrypted hash code is used.
3. If this property were not true, an attacker would be capable of the following sequence:
    1. First, observe or intercept a message plus its encrypted hash code;
    2. second, generate an unencrypted hash code from the message;
    3. third, generate an alternate message with the same hash code.

2nd pre-image resistance

1. Sometimes both the input and its hash value are known to a potential attacker:
    1. Attacker knows both *x* and its hash value *h(x)*.
    2. See Yuval's attack on signatures below.
2. It still needs to be difficult for the attacker to find another input with the same hash value.
3. A hash is called **2nd pre-image resistant** if it is hard to find $x' \neq x$ such that $h(x') = h(x)$, given any x and $h(x)$.

Recap: Computational problems

1. **Pre-image Resistant**: It should be hard to find a message with a given hash value.
2. **Collision Resistant**: it should be hard to find two messages with the same hash value.
3. **2nd Pre-image Resistant**: it should be hard to find another message with the same hash value.

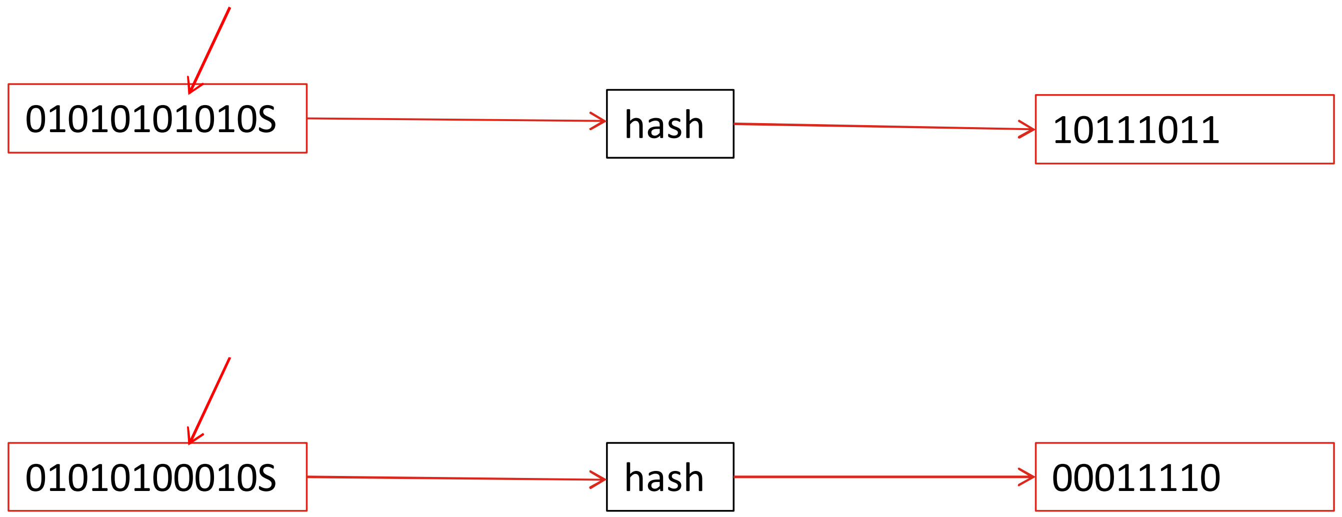
Relationships

1. Collision resistant implies 2nd preimage resistant.
2. A hash that is not preimage resistant is unlikely to be collision resistant (although this is not impossible).
3. A hash that is not preimage resistant is unlikely to be 2nd preimage resistant.

Avalanche

1. **Avalanche effect**: a small change to the input value (usually) changes the output value a lot.
2. Moreover, in a good hash function, a single bit change will lead to approximately half of the output values changing, on average.
    1. Our toy XOR example above is not a good hash function for security purposes.
    2. Ideally, an attacker should have subjective probability of approximately $1/2^L$ for any particular output (of length L) with a previously untested input.
3. Idea: if this wasn't the case, an attacker could use this information to tune a search for collisions.

Avalanche in a 8-bit hash (digest)



1. S is some fixed binary string. One digit of input changes. About half of the output bits change.

Colliding birthdays

1. The **birthday paradox**. Think about the three questions:
   1. How likely is it that a randomly chosen other person has the same birthday as you?
   2. How likely is it that there is another person in this class who has the same birthday as you?
   3. How likely is it that there are any two people in the class who have the same birthday?

Colliding birthdays: Q1

1. How likely is it that a randomly chosen other person has the same birthday as you?
2. We need assumptions:
    1. \`random’ above means \`uniformly at random’
    2. People are equally likely to have been born on any day of the year.
3. A: probability of collision = 1/365 ≈ 0.003.
4. Note: probability of <u>no</u> collision = 364/365 = 0.997.

Colliding birthdays: Q2

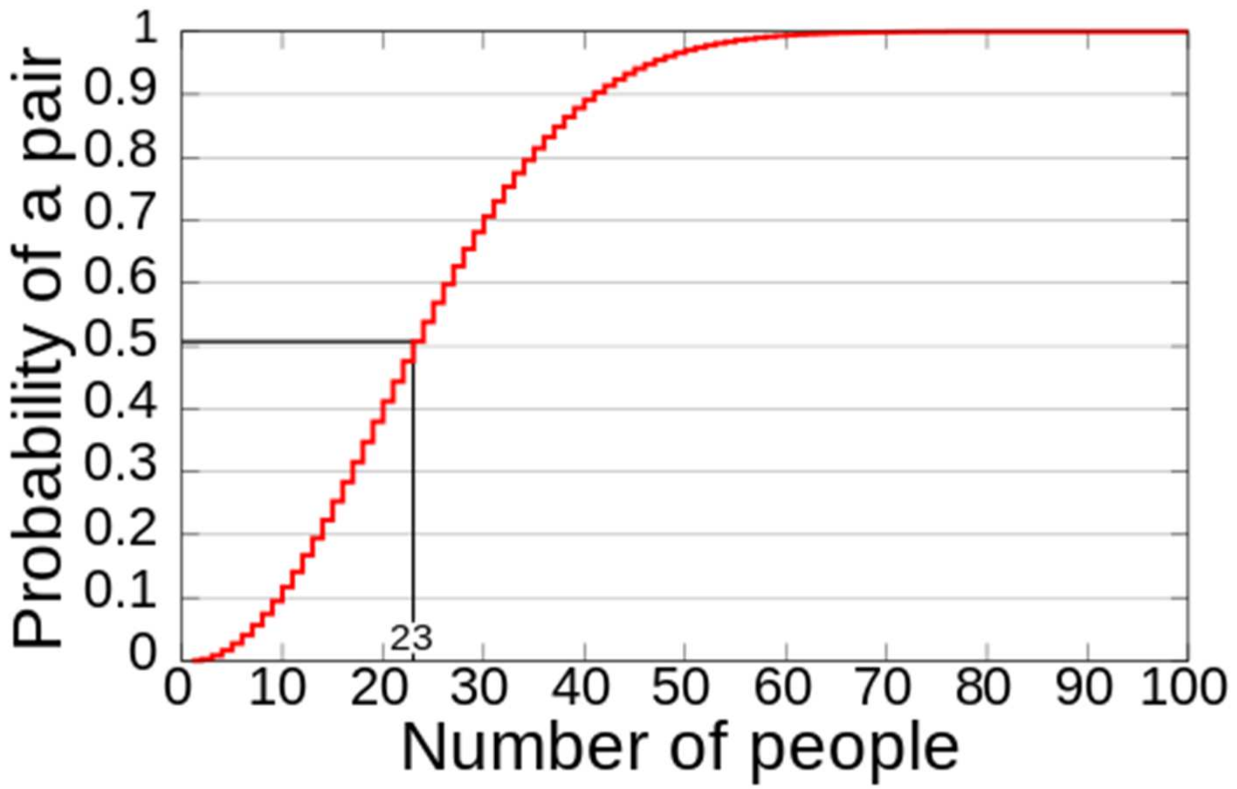
1. How likely is it that there is another person in this class who has the same birthday as you?
2. N people in the room.
    3. Below, N=23, for example
3. There are N – 1 people with whom you might share a birthday:
    1. There are 22 possible people with whom you might share a birthday.
    2. Assume people's birthdays are independent.
    3. Gives a $(364/365)^{22} \approx 0.94$ probability of no collision for somebody else with you.
    4. So only about 0.06 probability of collision.

Colliding birthdays: Q3

1. How likely is it that there are any two people in the class who have the same birthday?
2. There are N × (N – 1) / 2 pairs of people in the class:
    1. N = 23 gives 253 pairs.
    2. Probability of <u>no</u> collision = (1 - 1/365) x (1 - 2/365) x …. (1 – (23-1)/365) .
    3. Probability of collision ≈ 0.51.

Birthday paradox

Figure by Rajkiran g - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=10784025



Birthdays: sampling

1. Drawing random elements with replacement from a set of *N* elements, a repeat is likely after approximately √N = $N^{1/2}$ selections.
2. If N = 2^L elements (the size of our space of hash values), then a repeat is likely after approximately $(N^{1/2})^L = 2^{L/2}$ selections.

Birthday attack

1. Usable against any un-keyed hash function.
2. Based on birthday paradox.
3. Running time $O(2^{L/2})$, where L is the number of output bits in the hash.
4. Good hash functions will:
    1. Have sufficiently large L.
    2. Be such that finding a collision will not be much better than $2^{L/2}$.
    3. And sometimes meet more requirements.

Yuval's birthday attack example

1. An attack on use of hash for integrity.
2. 64-bit hash, so L = 64.
3. If machine hashes a million messages per second, finding a collision for a given hash value takes about 600,000 years. Finding a pair takes about an hour.
4. Mallory has two contracts, one that pays him £1000, the other £100,000. The first is to be signed with the hash. He makes $2^{L/2} = 2^{32}$ minor variations in each (e.g. spaces/control chars) and finds a pair with the same hash. Later claims second document was signed, not first. (Source: Schneier, Applied Crypto).

Hash length

1. 64-bit (output) hashes too small to survive Birthday attack.
2. Most real hashes have at least 128-bit outputs:
   1. This is starting to look too small, $2^{128/2} = 2^{64}$.
3. Many modern ones have at least 160 bits.
   1. For some purposes even 160 bits is not considered enough, $2^{160/2} = 2^{80}$.

Efficiency

1. Input, *x*, to hash, *h*, can be of variable length.
2. Essentially, anybody should be able to compute *h*(*x*) if they are given *x* (of reasonable size). So, *h* needs to sufficiently efficient.
3. However, we don’t want them to be able to search the space of inputs for collisions in any reasonable time.
4. This will depend on the size of the search space, the number of collisions, the unpredictability of hash function outputs, the efficiency of computation of hashes.

Efficiency (cont’d)

1. Sometimes hash functions are chosen not to be too efficient.
2. Toy example:
    1. Table of (insufficiently salted) password hashes stored.
    2. If some attacker gains access to this, they may run a dictionary attack.
        1. They have a list of the most likely passwords, and they compute the hash for each.
        2. They then search the table for *any* match.
    3. May be slowed down if hash computation is slow.

Fast hash functions and blocks

1. The output is a certain number of bits, say N, in length:
    1. When people speak about an N-bit hash algorithm, this is usually what they mean.
2. Most fast hash functions have a **block size**:
    1. An input x gets chopped into **blocks** $x_1, \dots , x_m$ each of the same length, say k.
    2. Input might have to be padded so that input is some multiple of k. Say the length is mk.

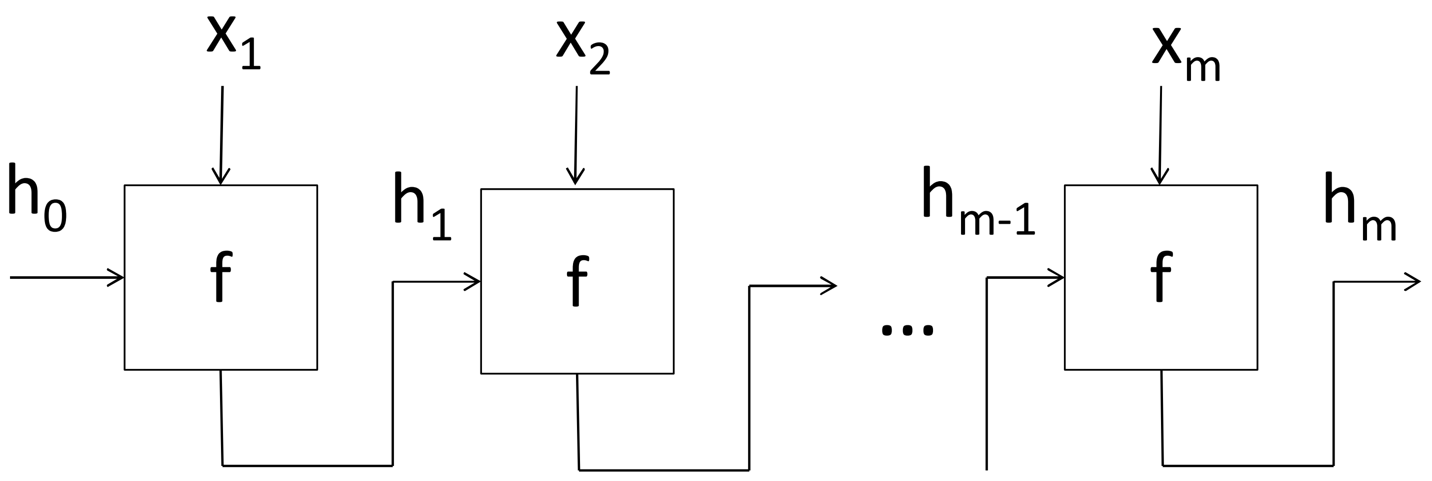
Compression functions

1. These are a common basic unit for building hash functions.
2. They take an input of a fixed length (the block length plus the length of the largest hash value) and produce a value.
3. A compression function *f* takes two inputs, an *n*-bit input called a chaining variable and a *b*-bit block and produces an *n*-bit output.
4. Here, $b < n$, hence the term compression.

Iterated hash functions

1. This allows us to handle messages that consist of arbitrarily many blocks, $x_1 \ldots x_m$.
2. The hash is generated by repeatedly applying the compression function.
3. *h_0*, a given initial value/initialization vector.
4. $h_i = f(x_i :: h_{i-1})$, for $1 \leq i \leq m$, where *::* is concatenation.
5. Remember that *m* was the number of input blocks.

Iterated hash function (cont'd)



Additional steps, e.g., careful padding are usually taken to make this more secure.

Recent history of hash function algorithms

1. MD4:
    1. Appeared in 1990. Used by old P2P algorithms.
    2. Input split into 512-bit blocks; hashing done in a sequence of rounds; each round is a very complex `nonlinear’ function of its inputs.
    3. Broken, and replaced by MD5.
2. MD5:
    1. 128-bit (16 byte) hash value (32 hex digits).
    2. Widely used, not collision-resistant, not suitable for SSL certificates and digital sigs. They can (and have been) faked.
    3. "*should be considered cryptographically broken and unsuitable for further use*” (CMU Soft. Eng. Institute).

SHA

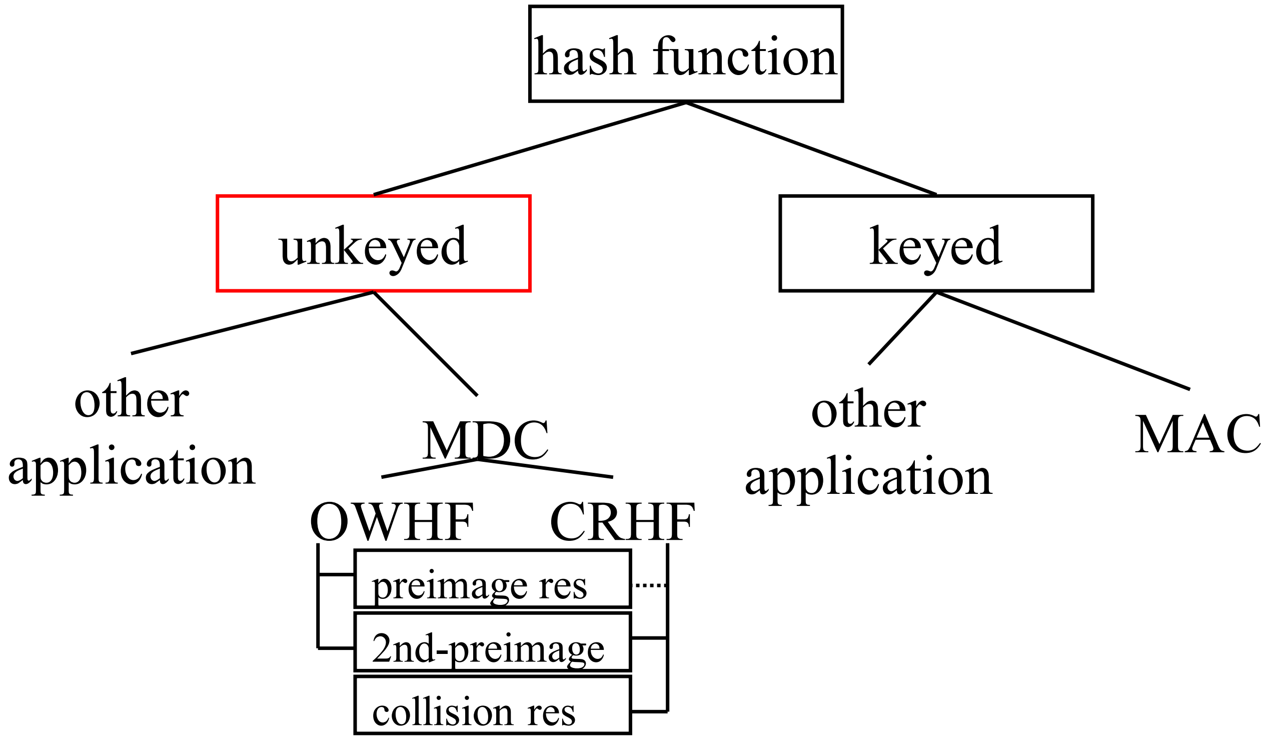
1. **SHA-1:**
    1. Also replaced MD4.
    2. 160-bit hashes. Designed by the NSA to be part of the Digital Signature Algorithm (DSA). Cryptographic weaknesses discovered in SHA-1.
    3. No longer approved for most cryptographic uses after 2010.
2. **SHA-2**: Two similar hash functions. Designed by the NSA.
    1. Evolutions of SHA-1.
    2. DSA now uses SHA-2.
3. **SHA-3**: Chosen in 2012 after a public competition among non-NSA designers.
    1. New. Different methods to SHA-1 and SHA-2.
4. These things are being broken and replaced worryingly fast.

Applications

1. SSL/TLS, PGP, SSH, S/MIME, and IPsec use SHA-1 or MD5.
    1. But forged X.509 certificates for SSL/TLS connections have been created by finding collisions.
2. Digital Signature Algorithm (DSA) has moved from SHA-1 to SHA-2.
3. Git uses SHA-1, but not for security, just for general integrity.

Hash function variants

1. Hash functions come in keyed and un-keyed variants. We start with the latter.

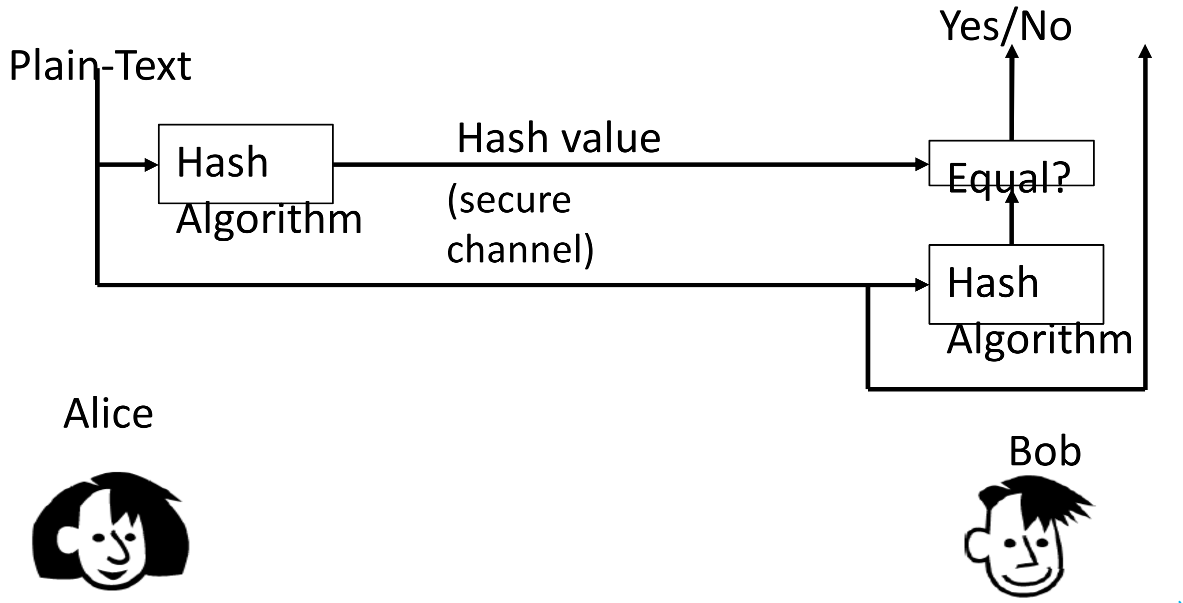


Names for unkeyed cryptographic hash functions

1. **(Cryptographically secure) hash function**.
2. **Compression function:** often a specific part of a hash function. Not to be confused with compression functions used elsewhere in computing.
3. The names **Message integrity check (MIC)** and **manipulation detection code (MDC)** hint at many of their typical applications.
4. The outputs (and sometimes the functions themselves) are referred to as **message digests**, **(digital) fingerprints**, **cryptographic checksum**s.

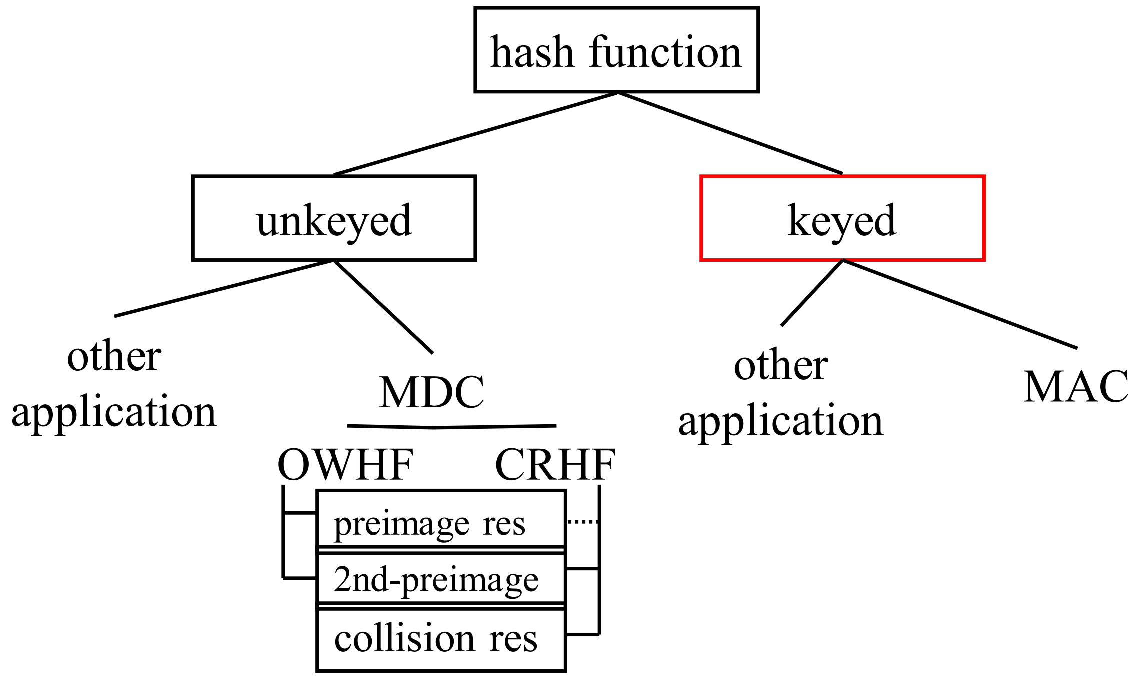
Un-keyed hash

1. *Unkeyed* cryptographic hash functions are **a sub-type of the cryptographic hash functions**. They take input of variable length and convert it into a fixed-length output, and the length depends on the type of the function used.



Keyed hash functions

1. An algorithm that uses a cryptographic key AND a cryptographic hash function to produce a message authentication code that is keyed and hashed.



Message authentication codes

1. A.k.a. **data authentication code (DAC)**.
2. Provide assurance about source and integrity of a message (**data origin authentication).**
3. Code is computed using a keyed hash function.

Message authentication codes (cont'd)

1. Hash function now also requires a key for its operation.
2. The key is a secret. Not known publicly.
3. Only someone with the secret key can verify the crypto
    1. Gives data origin authentication provided the key is kept secret.
4. Hash value is a function of the pre-image and the key (somehow concatenated).

Applications of MACs

1. **Message Authentication Codes:**
    1. Provide assurance about the source and integrity of a message (data origin authentication)
    2. Often done with **keyed hash functions**.
        1. Let *h* be one of these.
    3. Two inputs: message *x*, plus a secret key *k*.
    4. Write output as *h(k, x)*.
    5. Idea: receiver shares secret, k, with sender (only).

Applications of MACs (cont'd)

1. **Message Authentication Codes (cont'd):**

h needs to have the following **computational resistance property**: even if adversary can get hold of a set of pairs

$$<x, h(k, x)>$$

of messages and their hash values (using *h* and *k*), she can't use it compute *h(k, x')* for any other message *x'*.

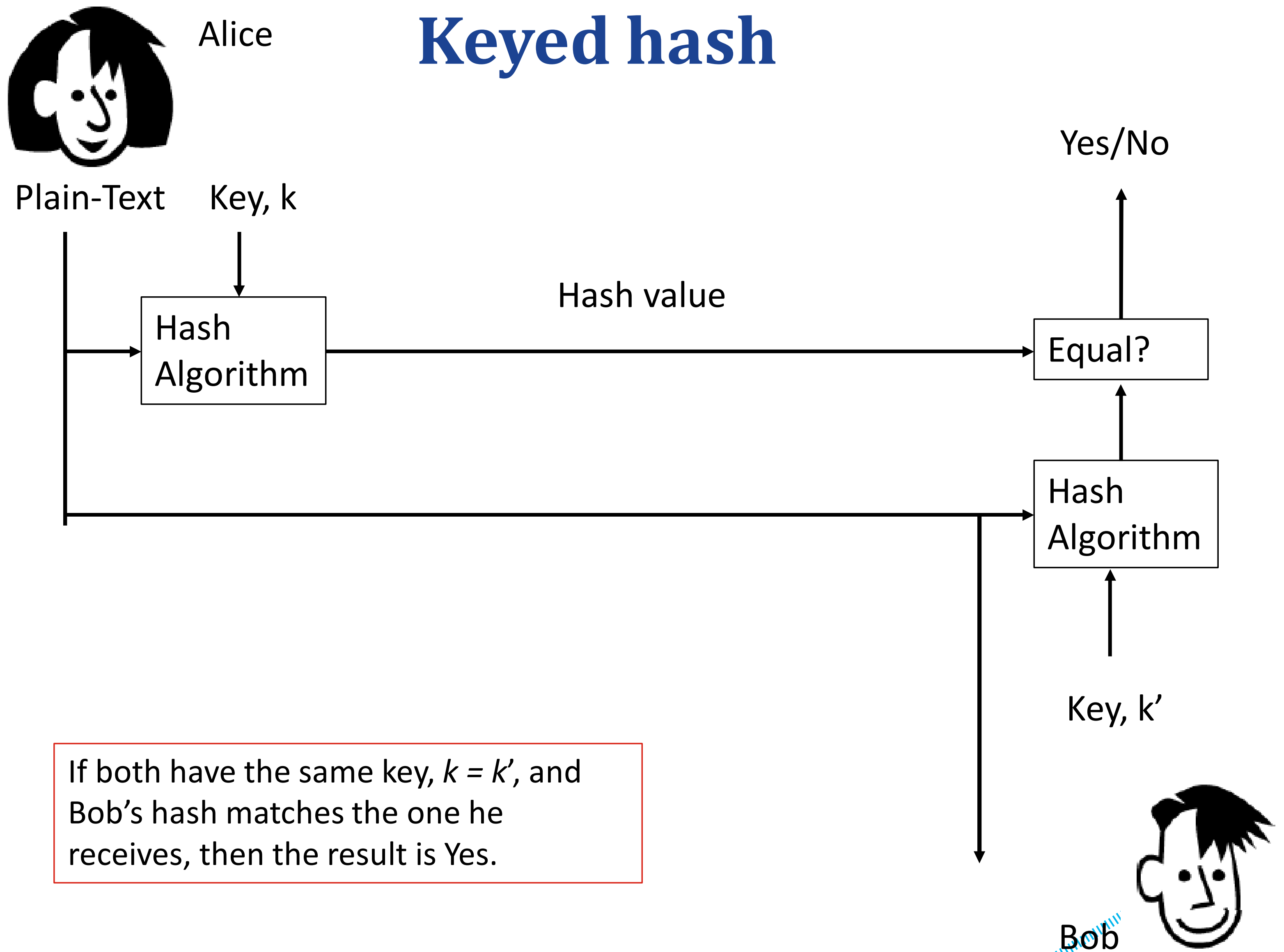
Key derivations functions

1. Now considered poor practice to use “pseudorandom function” (PRF) algorithms like SHA directly for many hashing applications as they are too fast.
2. Recommended to use **key derivation functions** (such as PBKDF2 or scrypt).
3. Iterate the PRF very many times.
4. scrypt also uses enough memory to make massive parallelization attack difficult.

Use of the key

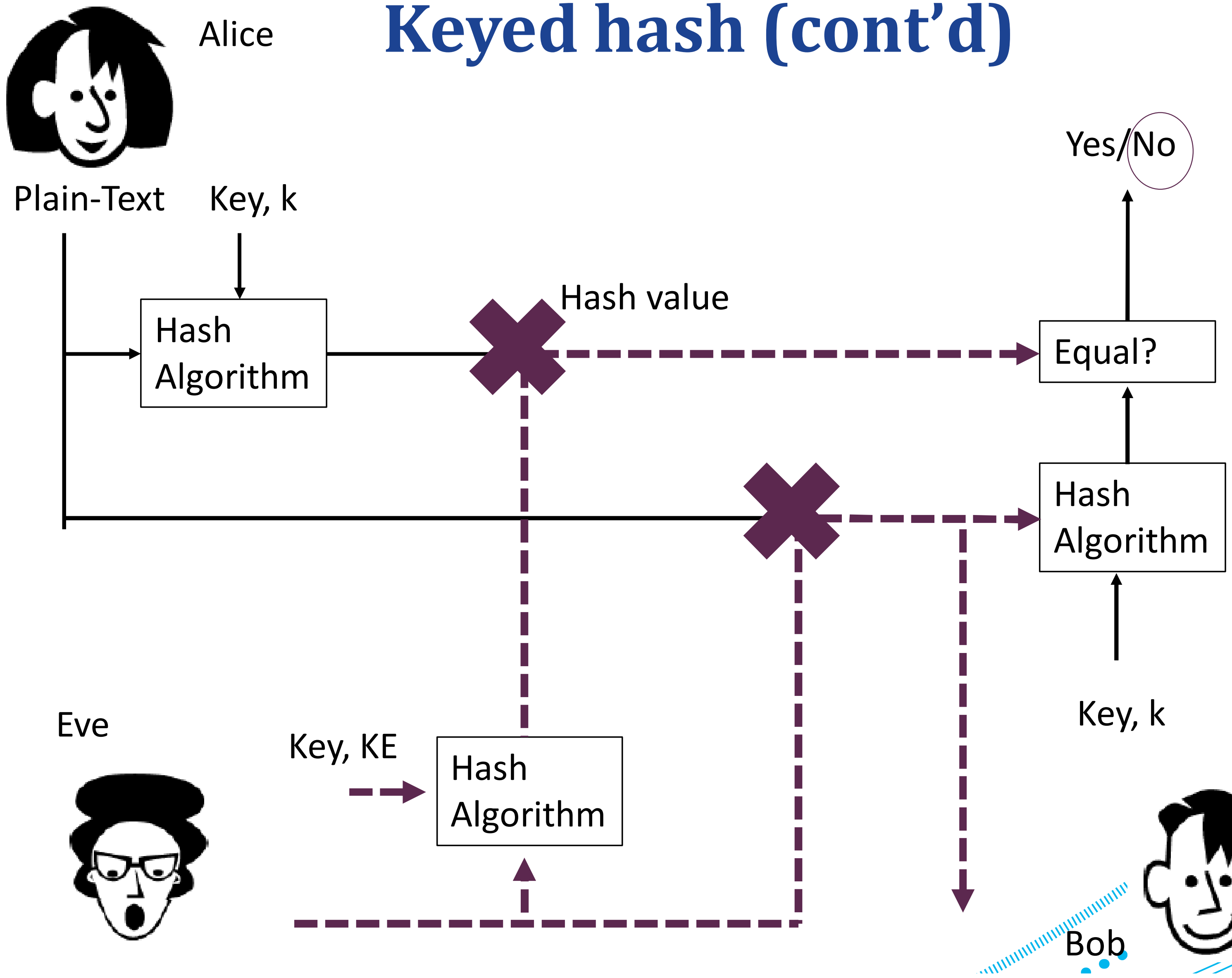
1. To authenticate a message, the receiver and sender must share the secret key.
2. A third party cannot validate the code without the key.

Keyed hash



If both have the same key, *k* = *k*', and Bob's hash matches the one he receives, then the result is Yes.

Keyed hash (cont'd)



Digital signatures

1. These are related to message authentication codes, but they rely on public key cryptography, so that the exact same key information does not need to be shared by Alice and Bob.
2. We’ll look at these in a later lecture.

HMAC construction

1. One standard way to construct a MAC from an un-keyed hash *h* is to take

   $$\mathrm{HMAC}_k(x) = h((k :: p_1) :: h((k :: p_2) :: x))$$

   for key k and message x, where p_1 and p_2 are strings(called *padding*) and :: is concatenation.
2. The padding extends *k* to the block length used in *h*.

Further applications

1. Authentication:
    1. You want to verify that someone, say *A*, has a copy *x* of a particular file (for which you also have a copy, *x'*). You don't want them to actually send *x* itself, perhaps because of its size or because they want to keep it a secret and the channel is insecure.
    2. You can just ask *A* to send the hash value, *h(x)* to you.
    3. You can calculate *h(x')*.
    4. If *h(x)* = *h(x')*, then you have a degree of confidence that *A* really does have genuine access to the file…
        1. Or they know someone who knows someone who […] does.

Further applications (cont'd)

1. Storage of authentication data:
    1. Allows us to avoid storing the full (secret) authentication data.
    2. Only have to store *complementary data:*
        1. e.g. password hashes in password files.

Other requirements

1. One might want to use hashes for other purposes.
2. Example 1:
    1. Bob takes the fact that *h(x)* and *x* (supplied by Alice) match and then passes *h(x)* to Charlie, telling him Alice's input *x* can be trusted if it matches h(x).
    2. Problem: Maybe Alice can compute *x* and *x'* with *h*(*x*) = *h*(*x'*).
3. Example 2:
    1. Bob uses *h*(*x*) from Alice as a proof by him of authenticity of *x*.
    2. Problem: given *h(x)* and *x*, maybe Bob can compute *x'* such that *h(x)* = *h(x').*

ABERDEEN 2040