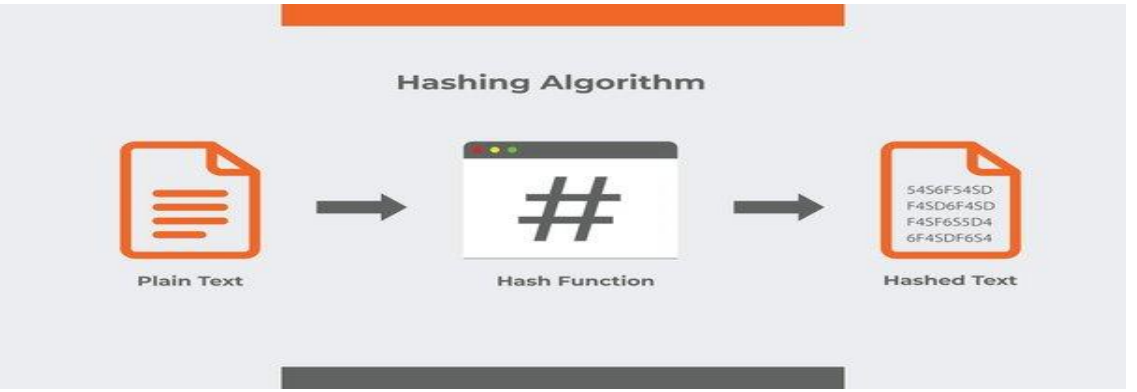
Mathematics for Computer Science
CSE 401
Hash Function/ Hashing/ Modulo Hash

Dr. Shah Murtaza Rashid Al Masud
Department of CSE
University of Asia Pacific

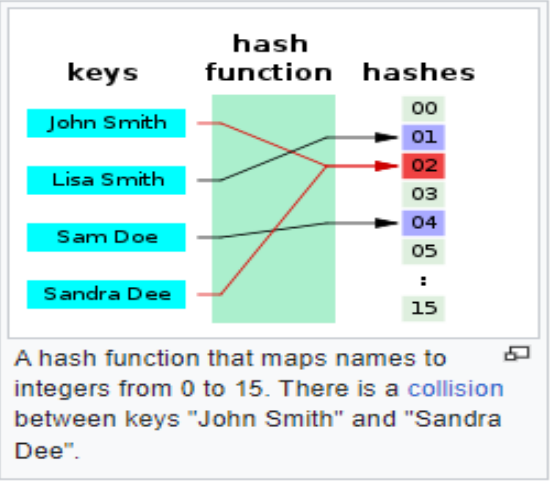


hashing (verb from French ‘hache’ = hatchet, & Old High German ‘happja’ = scythe)

A **hash function** is any function that can be used to map data of arbitrary size to fixed-size values to uniquely identify a specific object from a group of similar object.

We can refer to the function input as *message* or simply as input.

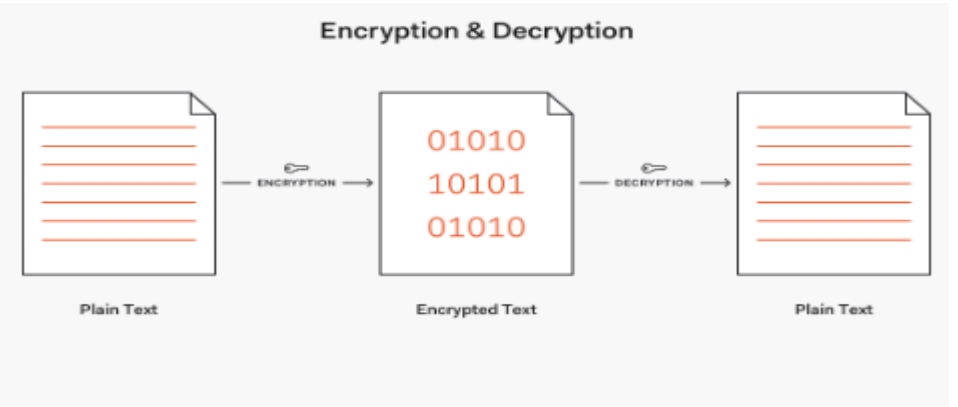
The values returned by a hash function are called *hash values*, *hash codes*, *digests*, or simply *hashes*. The values are usually used to index a fixed-size table called a *hash table*.

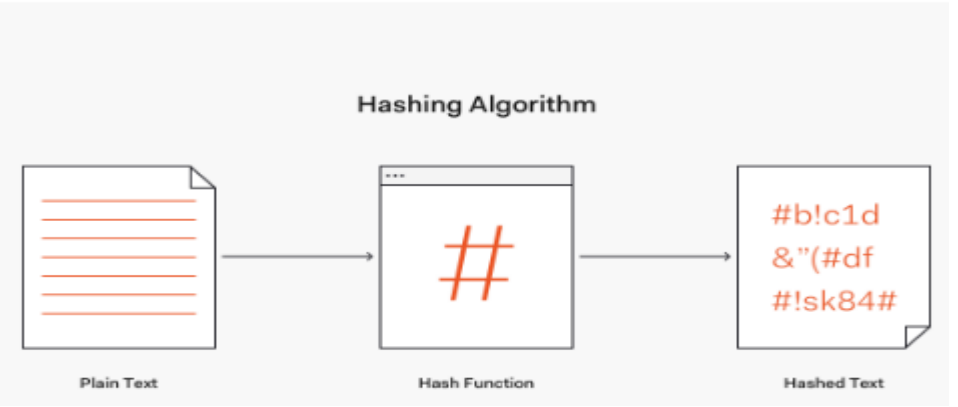


A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

By dictionary definition, hashing refers to "chopping something into small pieces" to make it look like a "confused mess". That definition closely applies to what hashing represents in computing.

It's easy and practical to compute the hash, but "difficult or impossible to re-generate the original input if only the hash value is known.“
In contrast to encryption, hashing is a one-way mechanism. The data that is hashed cannot be practically "unhashed".





Hash Functions

Consider a function $h(k)$ that maps the universe U of keys (specific to the hash table, keys could be integers, strings, etc. depending on the hash table) to some index 0 to m. We call this function a **hash function**. When inserting, searching, or deleting a key k, the hash table hashes k and looks at the $h(k)$th slot to add, look for, or remove the key.

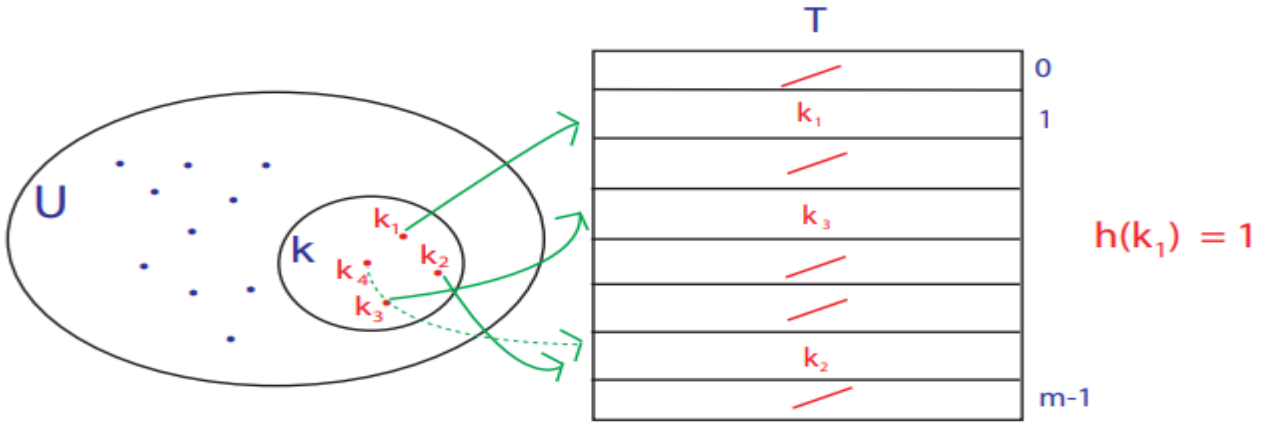


Figure 2: Mapping keys to a table

Different types of hashing algorithms

MD5 (Message-Digest Algorithm 5)

SHA-1 (Secure Hash Algorithm 1)

SHA-256 (Secure Hash Algorithm 256-bit)

SHA-3 (Secure Hash Algorithm 3)

Hashing

- A **hash function** tells us where to place an item in array called a **hash table**.
  - This method is known as **hashing**.

- Hash function **maps a search key into an integer** between 0 and n – 1 (or m-1).
  - We can have different hash functions.
  - Hash function depends on key type (int, string, ...)
    - E.g., **h(x) = x mod n**, where x is an integer (key) and n=size of hash table

Hash Methods:

1. **Division method**
2. **Multiplication method**

Hash Method

1. **Division method**
2. **Multiplication method**

```
h(k) = k mod n
```

Here, h(k) is the hash value obtained by dividing the key value k by size of hash table n using the remainder. It is best that n is a prime number as that makes sure the keys are distributed with more uniformity.

An example of the Division Method is as follows –

```
k=1276
n=10
h(1276) = 1276 mod 10
= 6
```

The hash value obtained is 6

Hash Method

1. **Division method**
2. **Multiplication method**

The hash function used for the multiplication method is –

```
h(k) = floor( n( kA mod 1 ) )
```

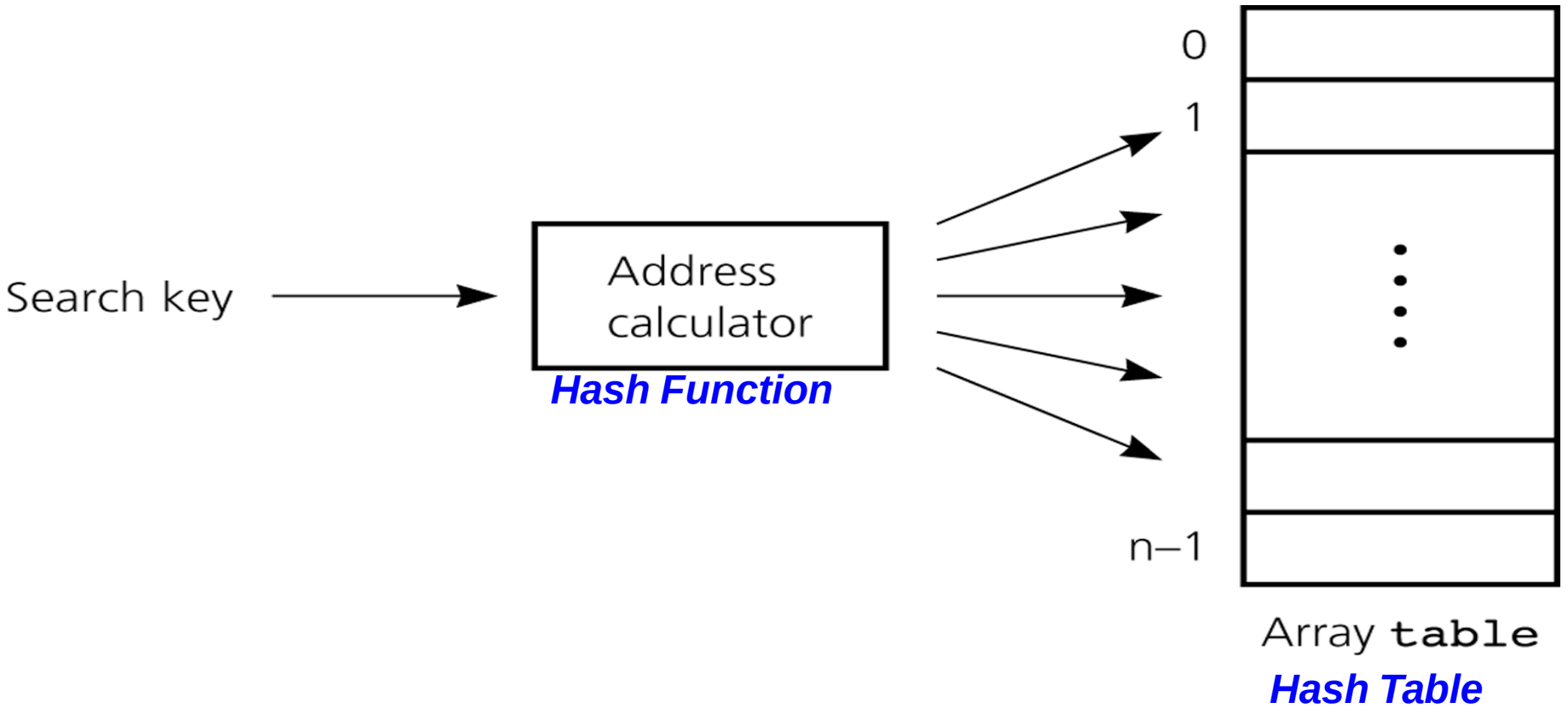
Here, k is the key and A can be any constant value between 0 and 1. Both k and A are multiplied and their fractional part is separated. This is then multiplied with n to get the hash value.

An example of the Multiplication Method is as follows –

```
k=123
n=100
A=0.618033
h(123) = 100 (123 * 0.618033 mod 1)
= 100 (76.018059 mod 1)
= 100 (0.018059)
= 1
```

The hash value obtained is 1

Hash Function -- Address Calculator



Hash Tables

- In hash tables, we have
  - **An array** (index ranges 0 … n – 1) and
    - Each array location is called a ***bucket***
  - **An address calculator** (***hash function***), which maps a search key into an array index between 0 … n – 1

Basic Operations

Following are the basic primary operations of a hash table.

Search − Searches an element in a hash table.

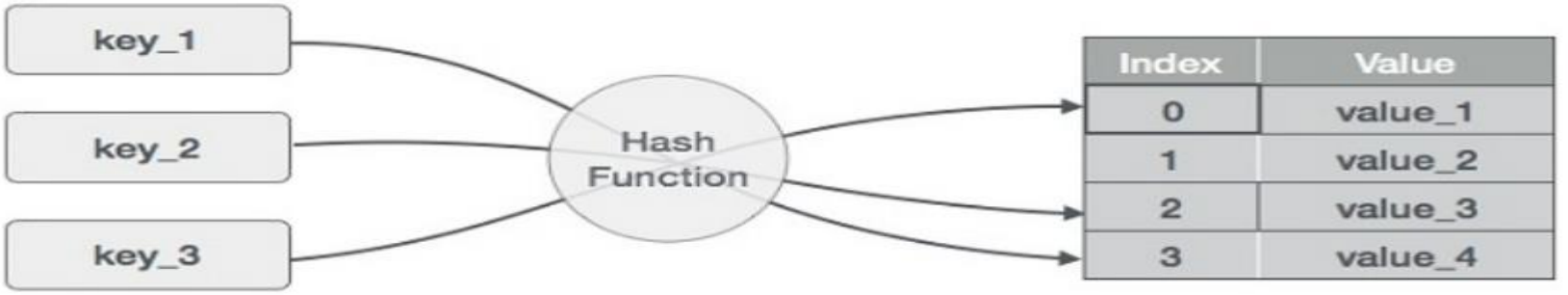
Insert − inserts an element in a hash table.

delete − Deletes an element from a hash table.

Hashing example

h(x) = x mod n, where x is an integer (key) and n=size of hash table

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an **example** of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Hashing example

h(x) = x mod n, where x is an integer (key) and n=size of hash table

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Hash Functions and Collisions

- We can design different hash functions.
- But a **good hash function** should
  - be easy and fast to compute
  - place items uniformly (evenly) throughout the hash table.
- A **perfect hash function** maps each search key into a unique location of the hash table.
  - A perfect hash function is possible if we know all search keys in advance.
  - In practice we do not know all search keys, and thus, a hash function can map more than one key into the same location.
- **Collisions** occur when a hash function maps more than one item into the same array location.
  - We have to resolve the collisions using a certain mechanism.

Clustering : When data is concentrated in one record, it is called clustering.

Hash Functions and Collisions

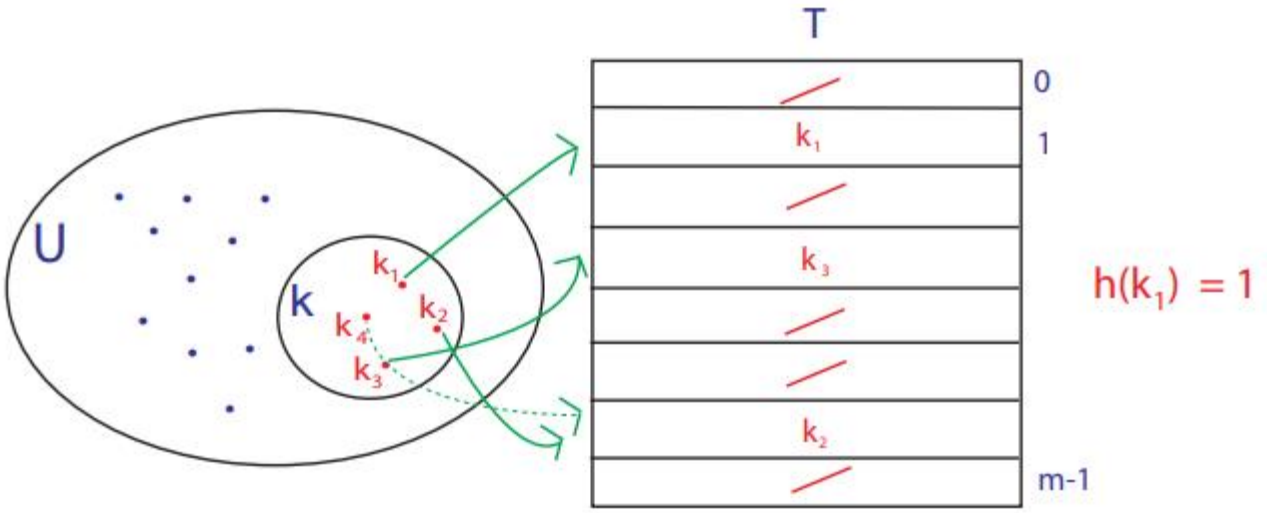


Figure 2: Mapping keys to a table

- two keys $k_i, k_j \in K$ collide if $h(k_i) = h(k_j)$

How do we deal with collisions?

Collisions Solution: Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Linear probing resolves collisions by simply checking the next slot, i.e. if a collision occurred in slot j, the next slot to check would be slot $j + 1$. More formally, linear probing uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

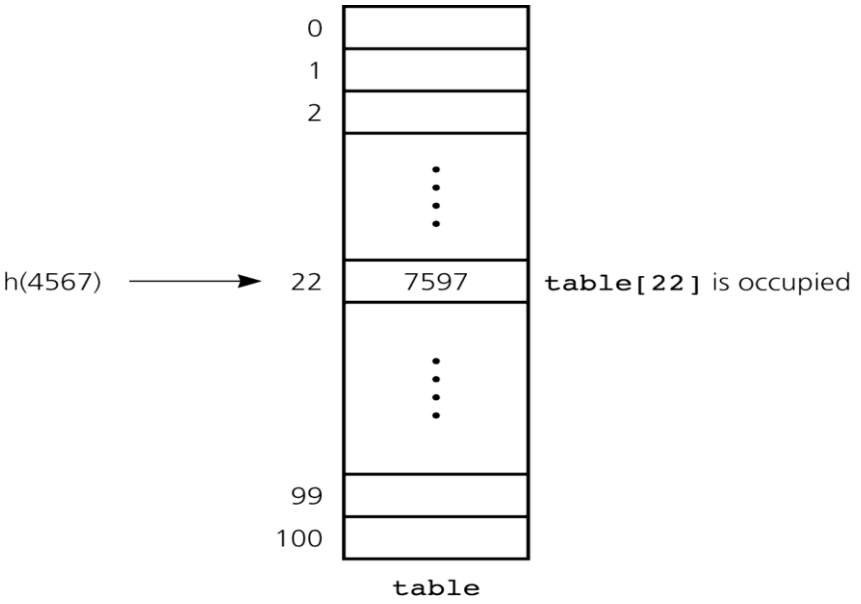
Where $h'(k)$ is the hash function we try first. If $h(k, 0)$ results in a collision, we increment i until we find an empty slot.

Collisions Solution: Linear Probing

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Recap: Linear Probing

- **linear probing**: search table sequentially starting from the original hash location.
  - Check next location, if location is occupied.
  - Wrap around from last to first table location



Recap: Linear Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **h(x) = x mod 11**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
      - 13 mod 11 = 2 ➔ 2+1=3
      - 25 mod 11 = 3 ➔ 3+1=4
      - 24 mod 11 = 2 ➔ 2+1, 2+2, 2+3=5
      - 10 mod 11 = 10
      - 9 mod 11 = 9 ➔ 9+1, 9+2 mod 11 =0

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

Collisions Solutions: Quadratic Probing

- **Quadratic probing:** Instead of linearly traversing through the hash table slots in the case of collisions, quadratic probing introduces more spacing between the slots we try in case of a collision, which reduces the clustering effect seen in linear probing.
- Almost eliminates clustering problem
- Approach:
  - Start from the original hash location **h´(k) (*where* h´(k) = k $mod\ m$)**
  - If location is occupied, check locations **h´(k)+i²**

(where ***h (k,i)*** is the next index for the key **k. i** is the collision number. For **first collision** i=1, For **second collision** i=2, For **third collision**, i=3, and so on. This is calculated until a space is assigned to the key element in the hash table.**)**

  - Wrap around table, if necessary.

Hash function:

h*(*k*, *i*) = (*h*´(*k*) + *i*²) mod *m ***(where*** h´(k) = k $mod\ m$***)***

Quadratic Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **h(x) = x mod 11**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 ➔ $2+1^2=3$
    - 25 mod 11 = 3 ➔ $3+1^2=4$
    - 24 mod 11 = 2 ➔ $2+1^2$, $2+2^2=6$
    - 10 mod 11 = 10
    - 9 mod 11 = 9 ➔ $9+1^2$, $9+2^2$ mod 11, $9+3^2$ mod 11 =7

0	
1	
2	2
3	13
4	25
5	
6	**24**
7	**9**
8	30
9	20
10	10

Collisions Solutions: Double Hashing

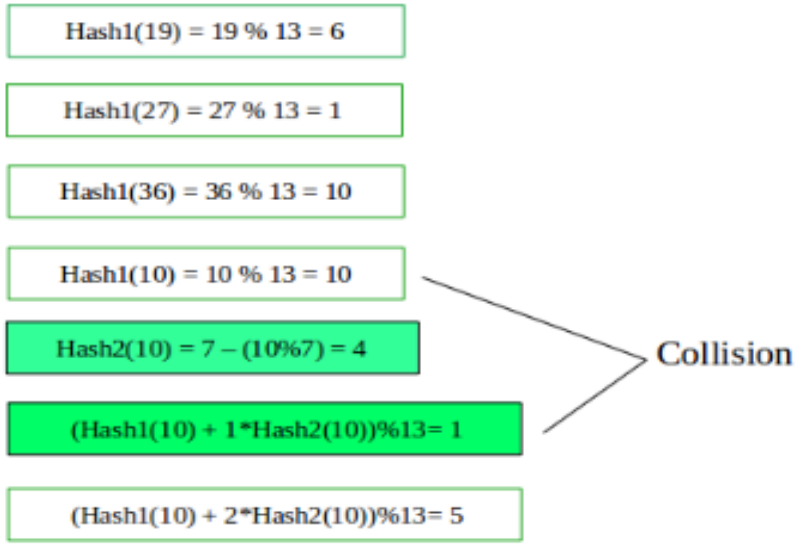
Double hashing resolves collisions by using another hash function to determine which slot to try next:

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

With double hashing, both the initial probe slot and the method to try other slots depend on the key k, which further reduces the clustering effect seen in linear and quadratic probing.

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)



Collisions Solutions: Double Hashing

- **Double hashing** also reduces clustering.
- **Idea**: increment using a **second hash function h_2**. Should satisfy:

$$h_2(\text{key}) \neq 0$$

$$h_2 \neq h_1$$

- Probes following locations until it finds an unoccupied place

$h_1(\text{key})$

$h_1(\text{key}) + h_2(\text{key})$

$h_1(\text{key}) + 2*h_2(\text{key})$,

...

Double Hashing -- Example 1

- Example:
  - Table Size is 11 (0..10)
  - Hash Function:

$$h_1(x) = x \bmod 11$$

$$h_2(x) = 7 - (x \bmod 7)$$

  - Insert keys: 58, 14, 91
    - 58 mod 11 = 3
      - 14 mod 11 = 3 ➔ 3+7=10
      - 91 mod 11 = 3 ➔ 3+7, 3+2*7 mod 11=6

0	
1	
2	
3	58
4	
5	
6	91
7	
8	
9	
10	14

Double Hashing -- Example 2

- Example:
  - Table Size is 11 (0..10)
  - Hash Function:

$$h_1(x) = x \bmod 11$$

$$h_2(x) = 1 + (x \bmod t)$$

, where t = *tablesize-1* (*t = 10 here*)

  - Insert keys: 58, 14, 91, 69, 80, 102, 25, 113, 124
    - 58 mod 11 = 3
      - 14 mod 11 = 3 ➔ 3+(1+4)=8 mod 11=8
      - 91 mod 11 = 3 ➔ 3+(1+1)= 5 mod 11=5
      - 69 mod 11 = 3 ➔ 3+(1+9)= 13 mod 11=2
      - 80 mod 11 = 3 ➔ 3+(1+0)= 4 mod 11=4
      - 102 mod 11 = 3 ➔ 3+(1+2)= 6 mod 11=6
      - 25 mod 11 = 3 ➔ 3+(1+5)= 9 mod 11=9
      - 113 mod 11 = 3 ➔ 3+(1+3)= 7 mod 11=7
      - 124 mod 11 = 3 ➔ 3+(1+4)= 8 (full) + (1+4) mod 11=2 (full)+ (1+4) mod 11 = 7 (full) + (1+4) mod 11 = 1 —> 3+4*h_2(124) mod 11

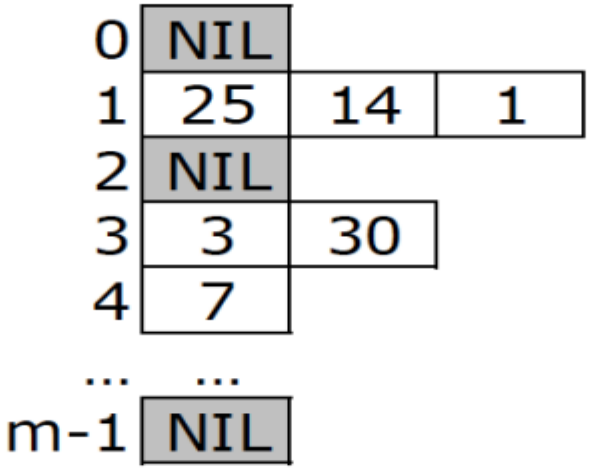
0	
1	124
2	69
3	58
4	80
5	91
6	102
7	113
8	14
9	25
10	

Collision Solutions: Chaining

In the chaining method of resolution, hash table slot j contains a linked list of every key whose hash value is j. The hash table operations now look like

- `insert(k)` - insert k into the linked list at slot $h(k)$
- `search(k)` - search for k in the linked list at slot $h(k)$ by iterating through the list
- `remove(k)` - search for k in the linked list at slot $h(k)$ and then remove it from the list

With chaining, if a key collides with another key, it gets inserted into the same linked list in the slot they hash into.



Collision Solutions: Chaining

Linked list of colliding elements in each slot of table

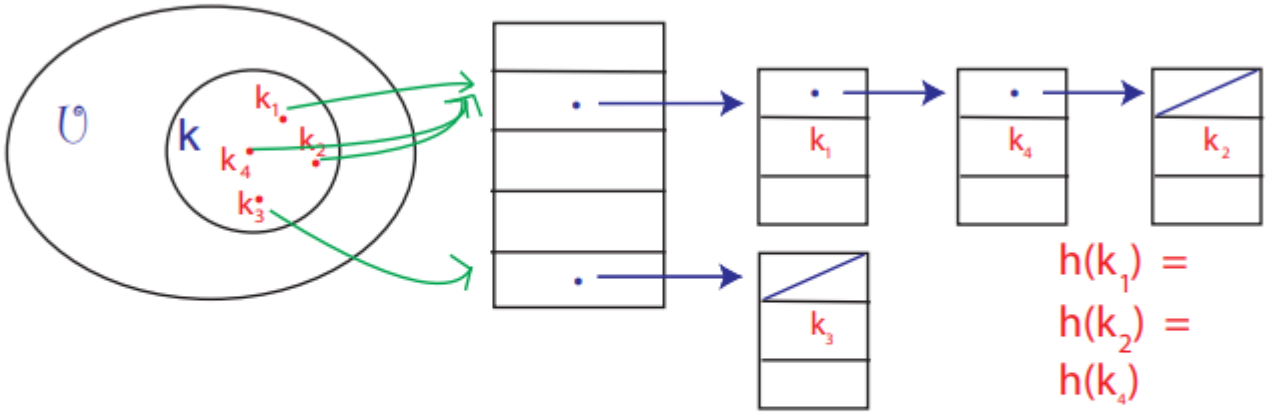


Figure 3: Chaining in a Hash Table

- Search must go through *whole* list T[h(key)]

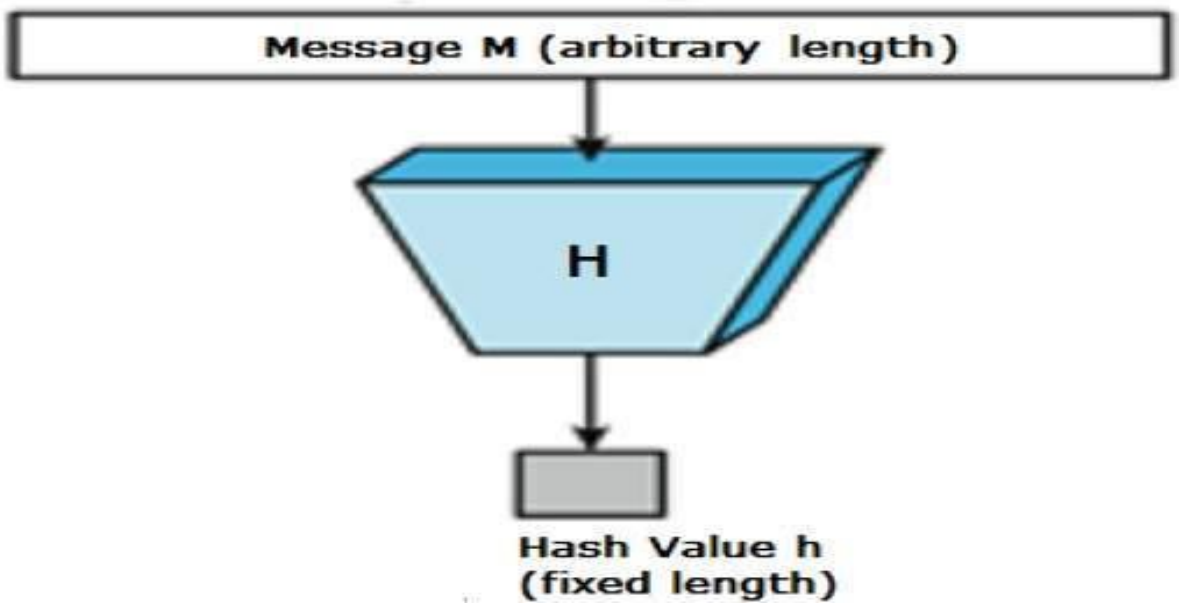
Hashing/Hash Functions

Hash functions are extremely useful and appear in almost all information security applications.

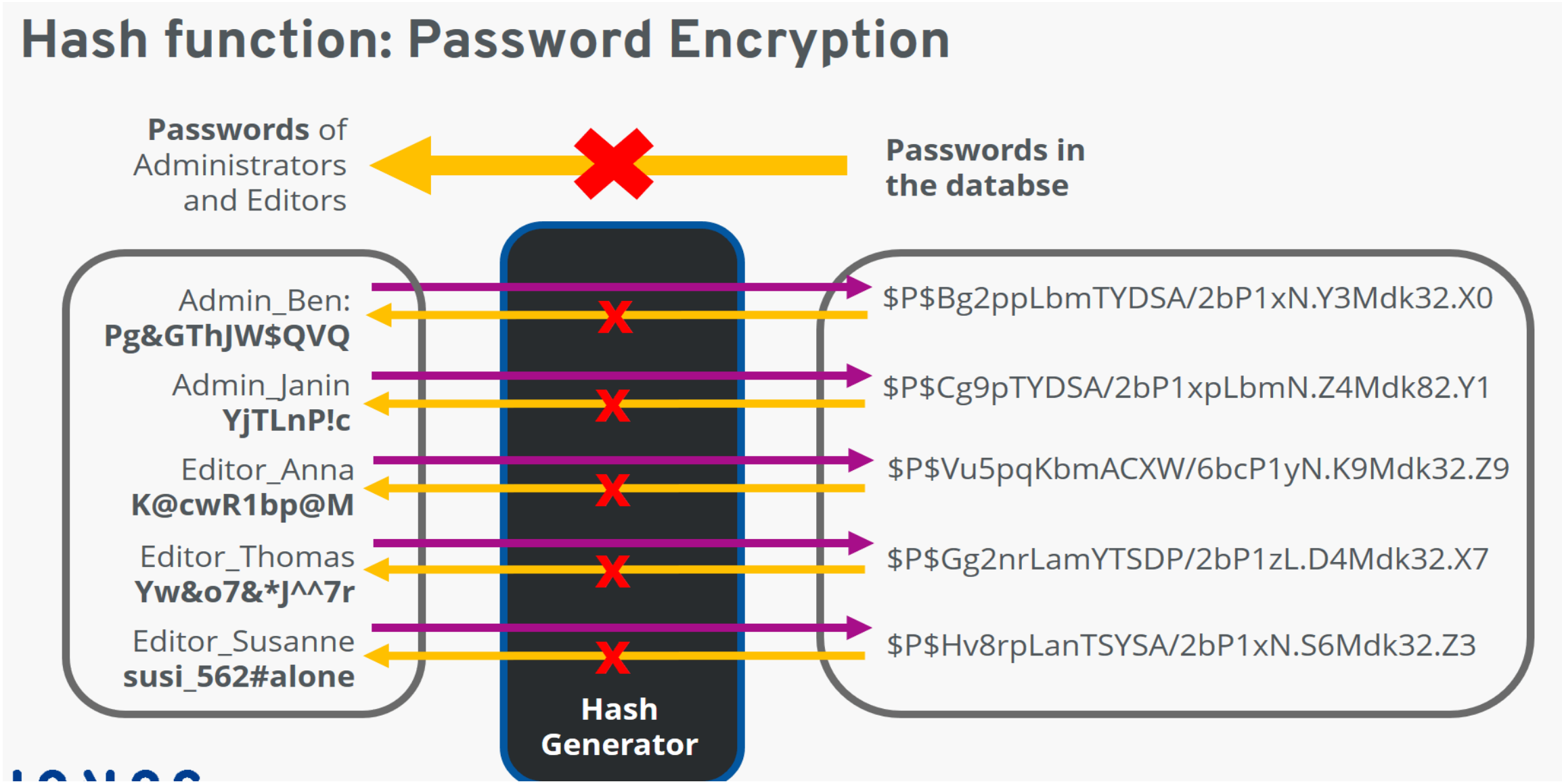
A hash function is a mathematical function that converts a numerical input value into another compressed numerical value. The input to the hash function is of arbitrary length (variable, any length) but output is always of fixed length.

Values returned by a hash function are called **message digest** or simply **hash values**. The following picture illustrated hash function −

A **hash function** converts strings of different length into fixed-length strings known as hash values or digests. You can use hashing to scramble passwords into strings of authorized characters for example. The output values cannot be inverted to produce the original input.



Hashing/Hash Functions



Hashing/Hash Functions

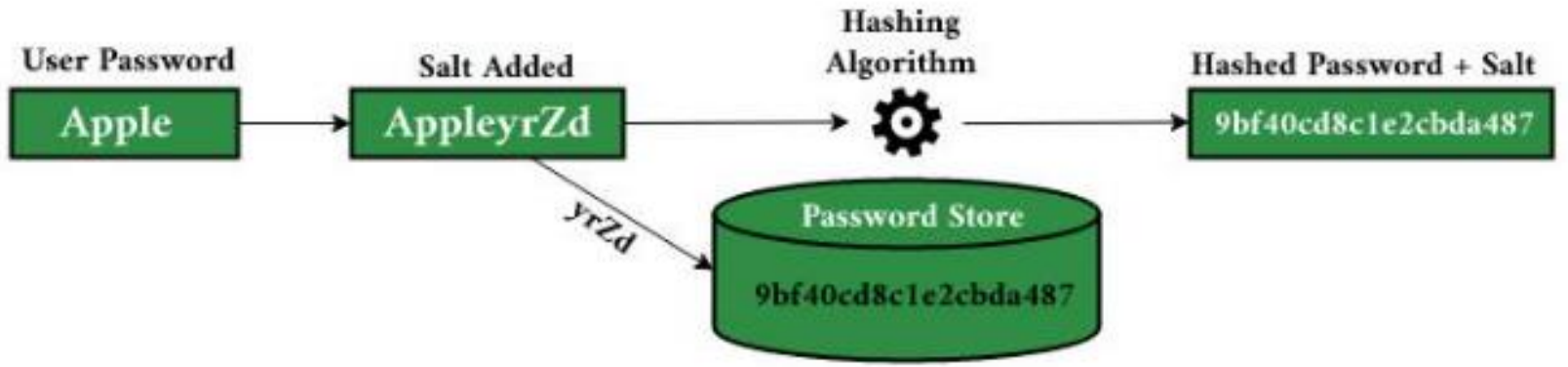
Hash functions are designed so that they have the following **properties**:

One-way

Once a hash value has been generated, it must be **impossible to convert it back** into the original data. For instance, in the example above, there must be no way of converting “PHv8rpLanTSYSA/2bP1xN.S6Mdk32.Z3” back into “susi_562#alone”.

Salted Secure Hash Algorithm

Salted secured hash algorithm helps protect password hashes against dictionary attacks by introducing additional randomness. Password hash salting is when random data – a salt – is used as an additional input to a hash function that hashes a password.

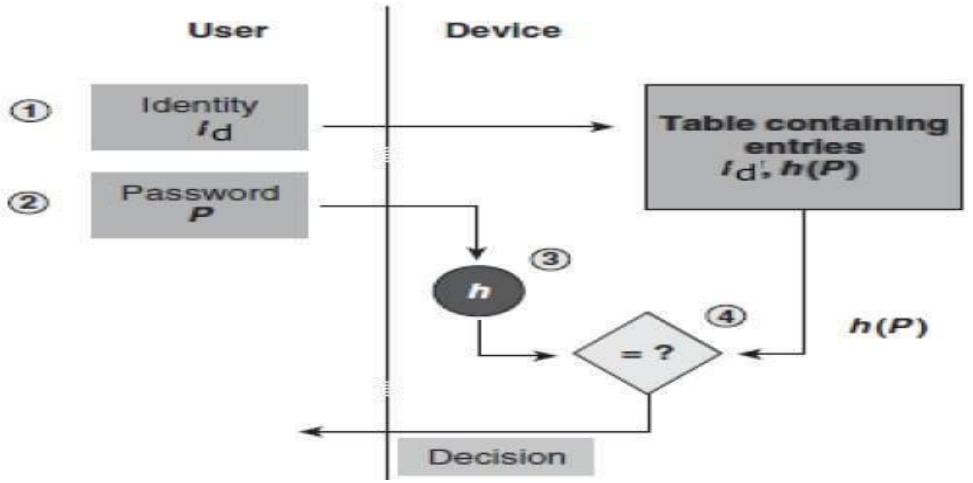


Password hashing

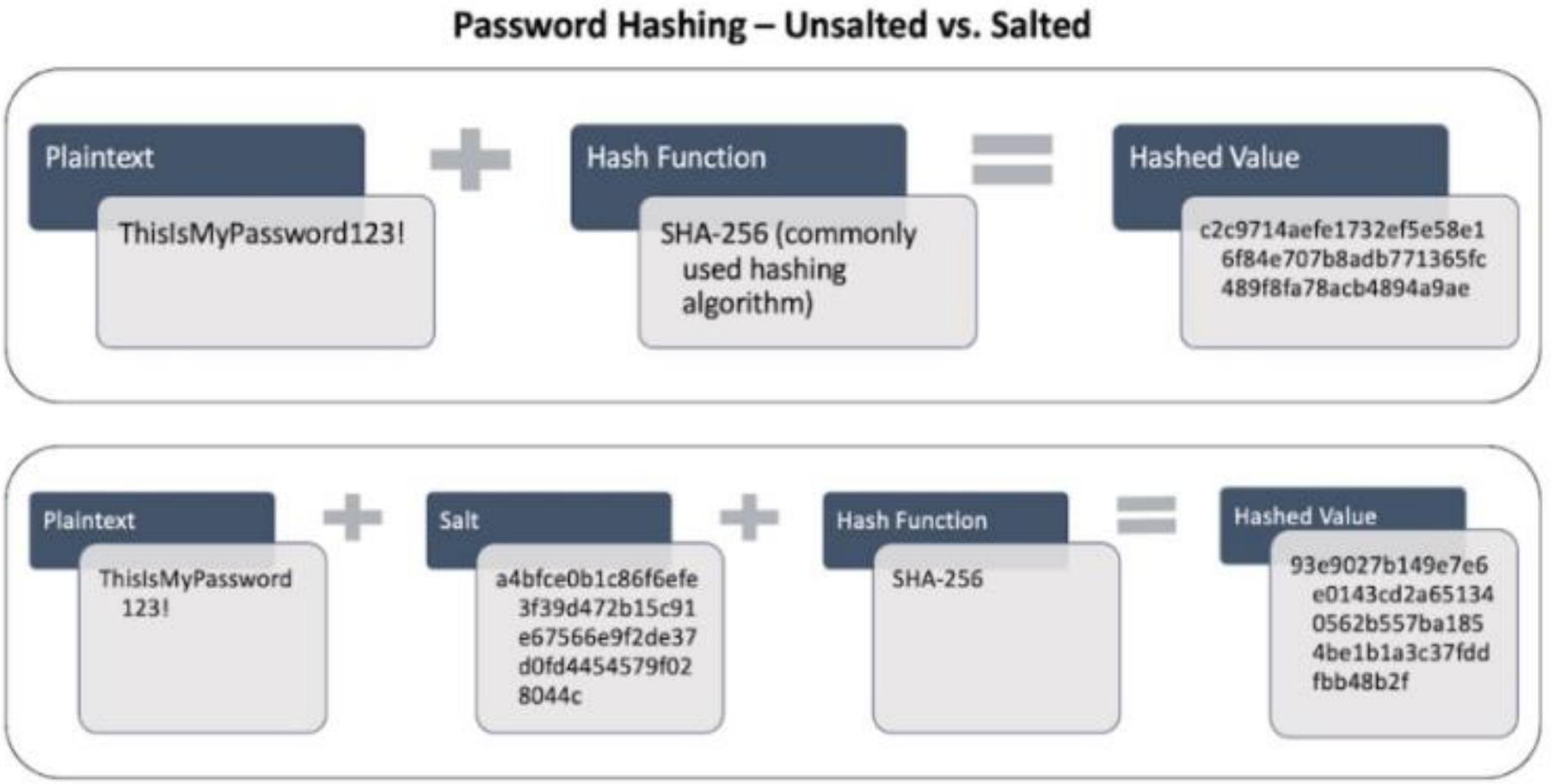
Password hashing is a method of protecting passwords by converting them into a series of random characters, also known as a hash.

This process is different from encryption, which is used to conceal information and can be reversed.

Password hashes, on the other hand, are designed to be irreversible, meaning that even if a hacker gains access to the hash, they cannot determine the original password.



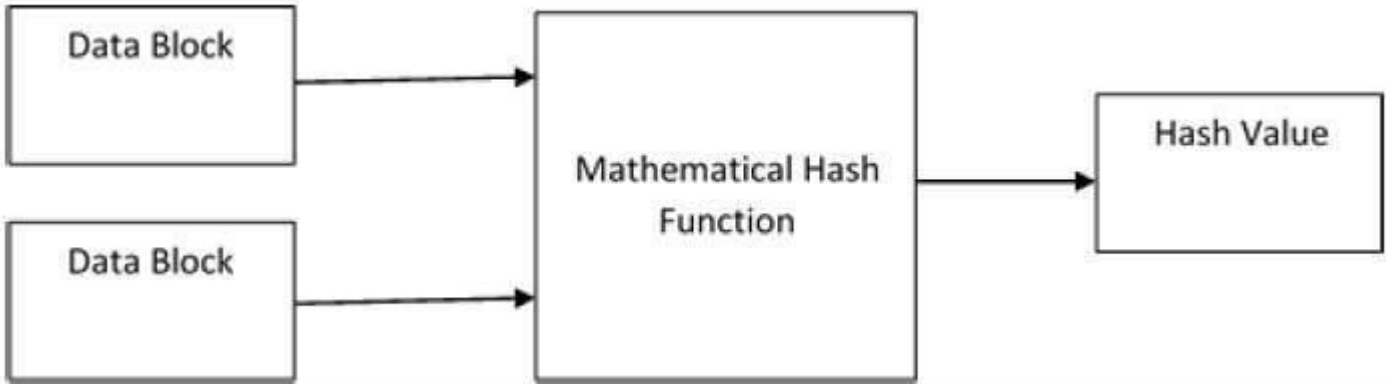
Password hashing



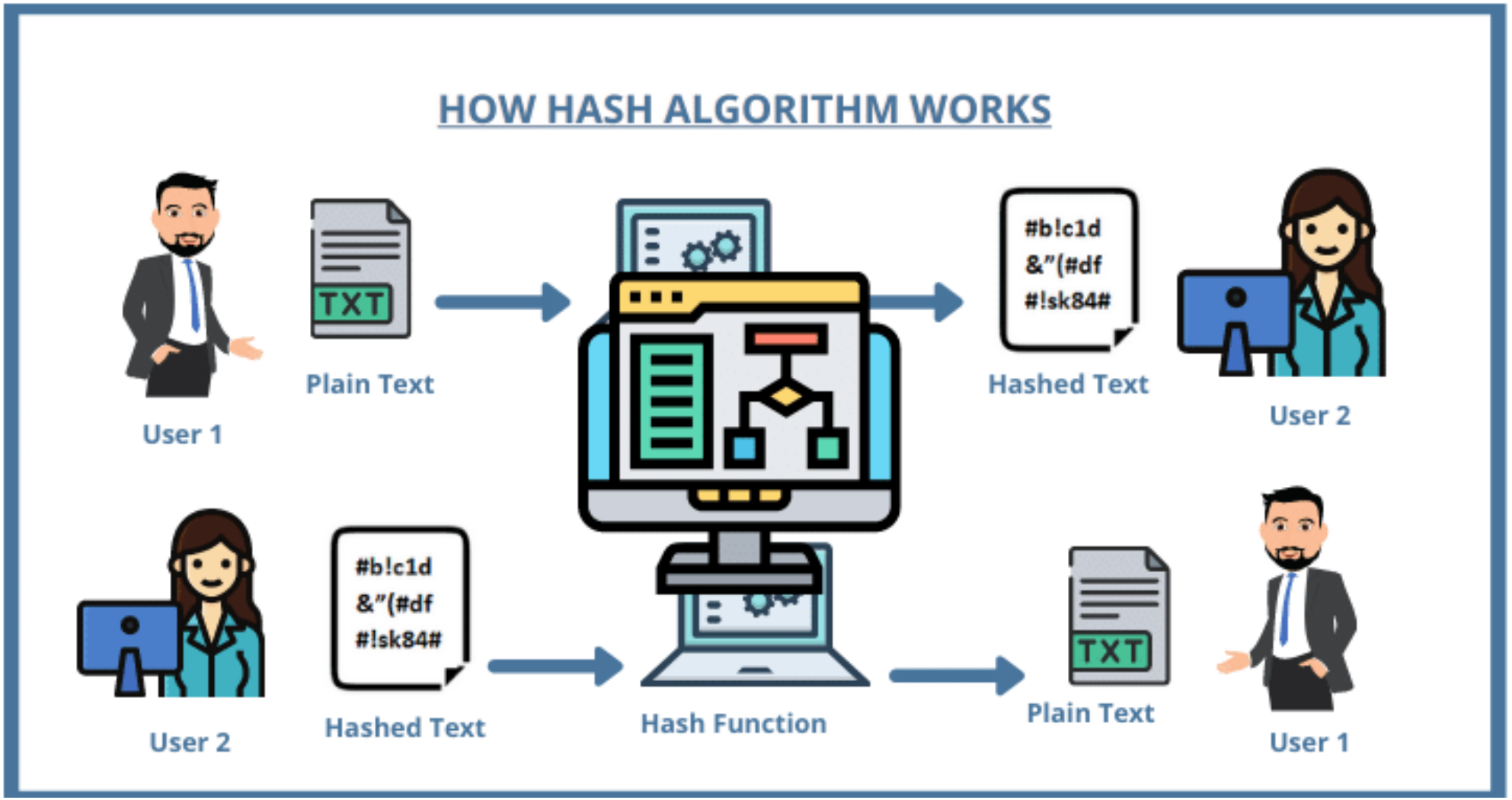
Cryptographic Hash Function (CHF)

A cryptographic hash function (CHF) is an equation used to verify the validity of data.

It has many applications, notably in information security (e.g. user authentication).

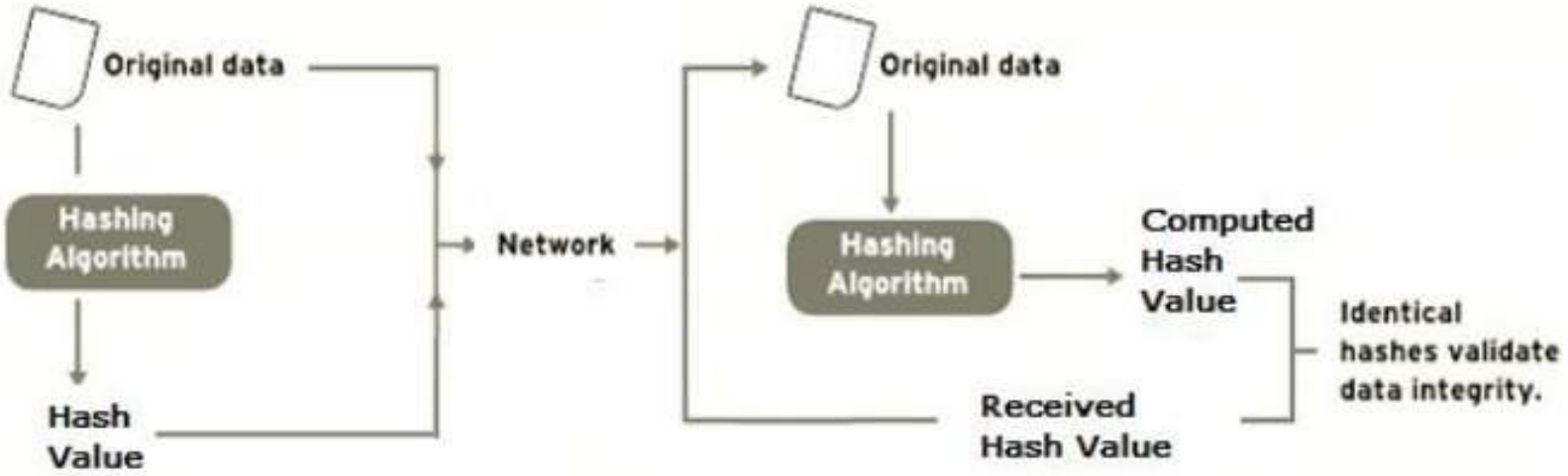


Cryptographic Hashing Algorithms

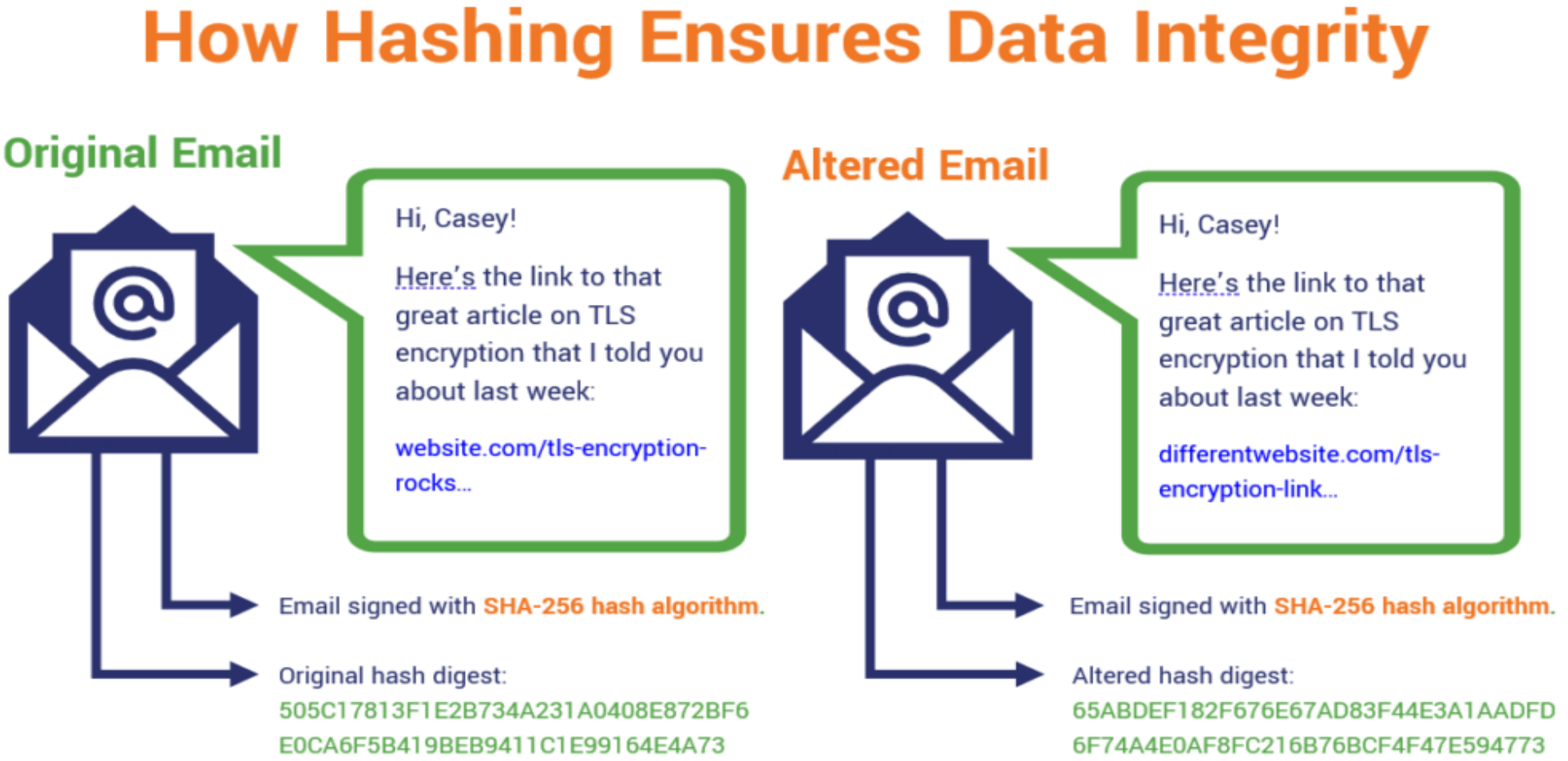


Data Integrity Check

Data integrity check is a most common application of the hash functions. It is used to generate the checksums on data files. This application provides assurance to the user about correctness of the data.

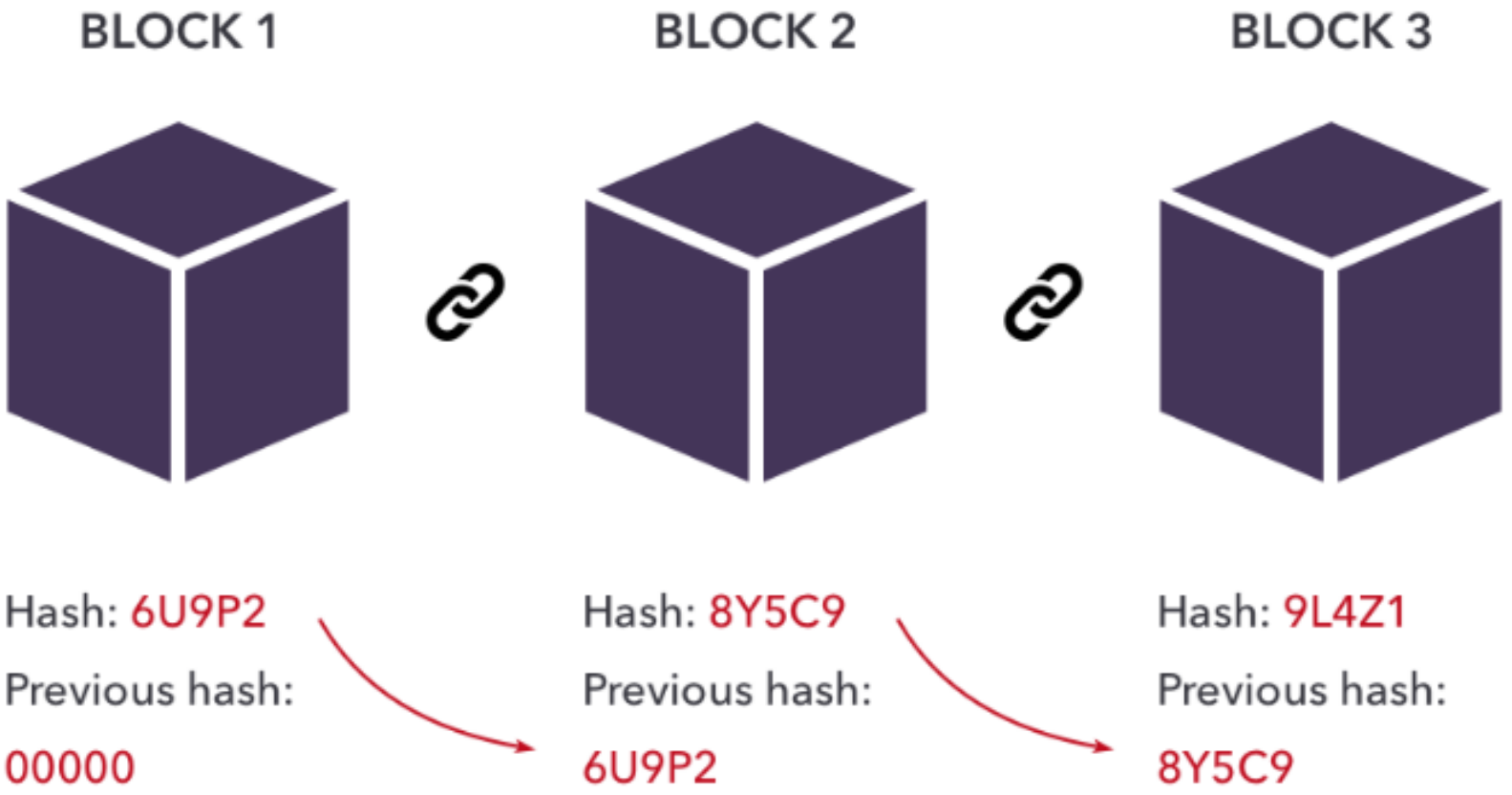


Data Integrity Check



The example above is of a digitally signed email that's been manipulated in transit via a MitM attack. The hash digest changes completely when any of the email content gets modified after being digitally signed, signaling that it can't be trusted.

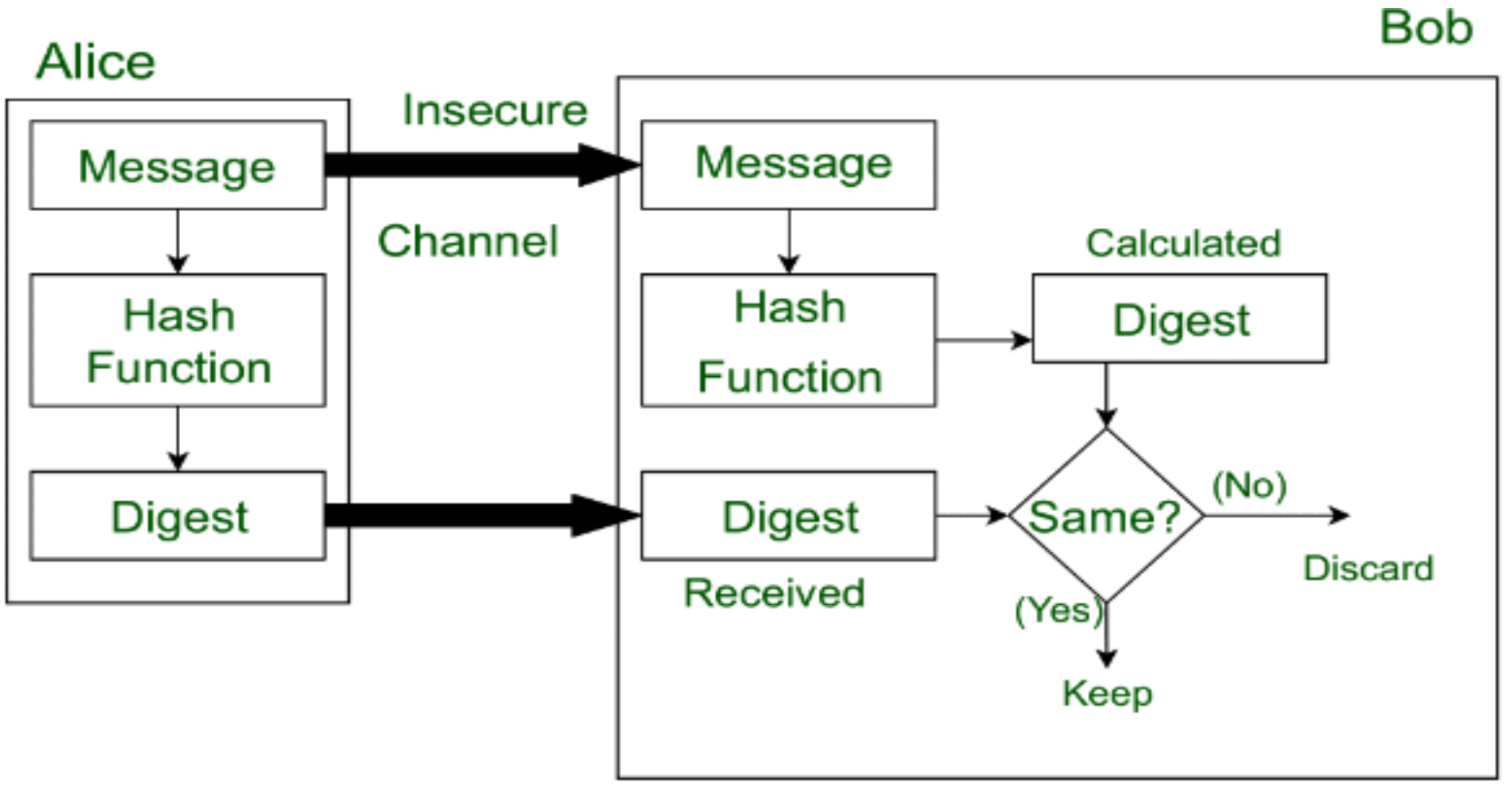
Hashing in Blockchain



Code signing certificate

https://sectigostore.com/blog/hash-function-in-cryptography-how-does-it-work/

Message Digest in Information security



https://www.geeksforgeeks.org/message-digest-in-information-security/

Reference: Concrete mathematics by Ronald, Knuth