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

Part 2:

5.5 Authenticated encryption

5.6 Time-memory tradeoff for dictionary attack (Optional)

5.7 Birthday attack variant

5.8 Other interesting cryptography topics

5.9 Summary of cryptography

5.5 Authenticated Encryption

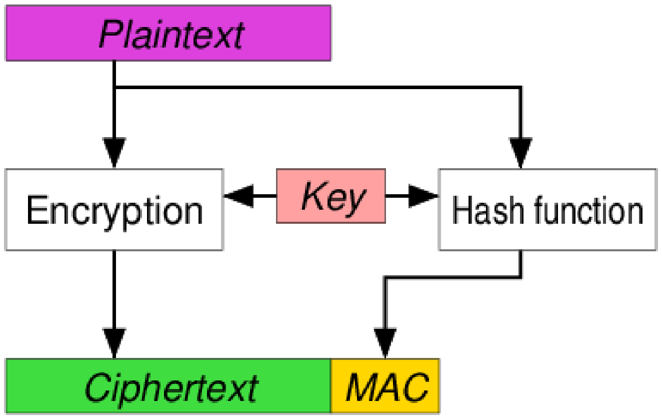
What is Authenticated Encryption

- ***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 ***(C, T)***

Illustration with 1 key
(Source: Wikipedia)

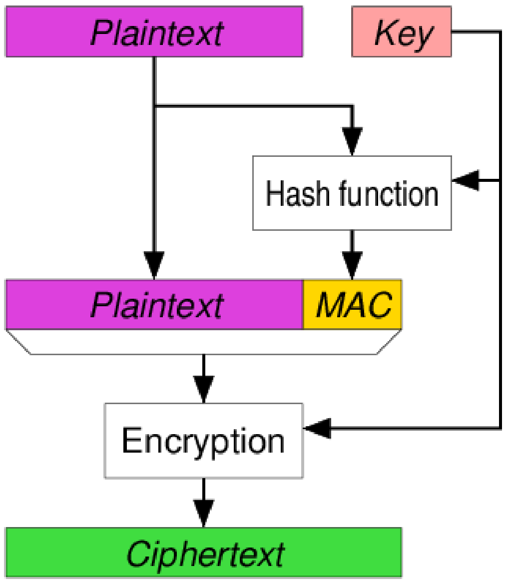


- It is used in **SSH** (with a strong MAC like HMAC-SHA-256)
- Issue: T may not be random looking, and could leak information

MAC-then-Encrypt (MtE)

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

Illustration with 1 key
(Source: Wikipedia)

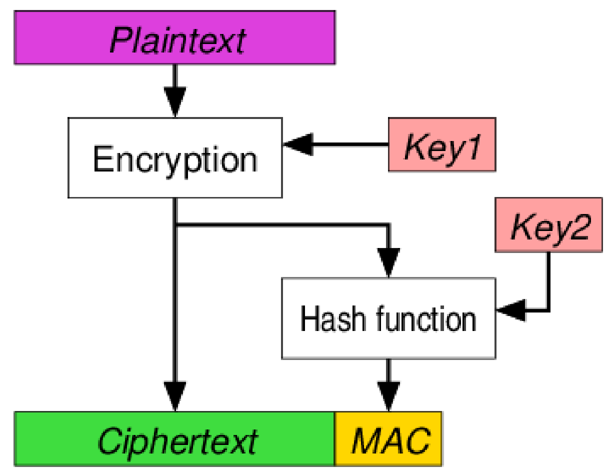


- It is used in **SSL** and **TLS** up to version 1.2:
  the latest TLS v1.3 uses authenticated ciphers, e.g. AES-GCM
- Issue: a decryption is still needed on a corrupted message

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 **($\boldsymbol{C}$, $\boldsymbol{T}$)**

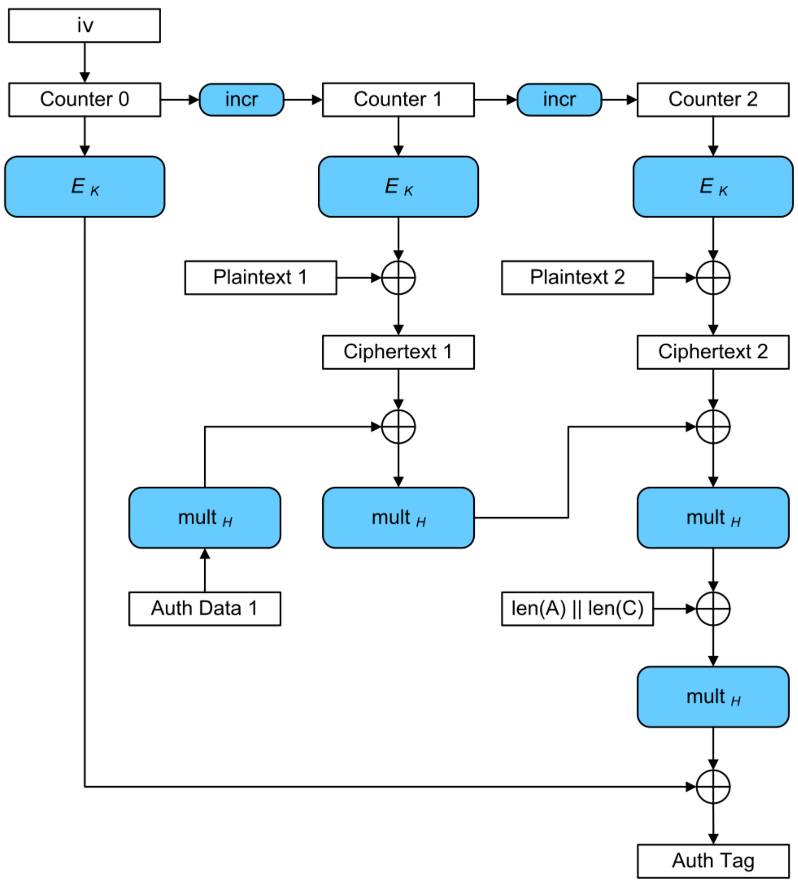
Illustration with 1 key
(Source: Wikipedia)



- 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

Optional

5.6 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 (TMT)

- 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:

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

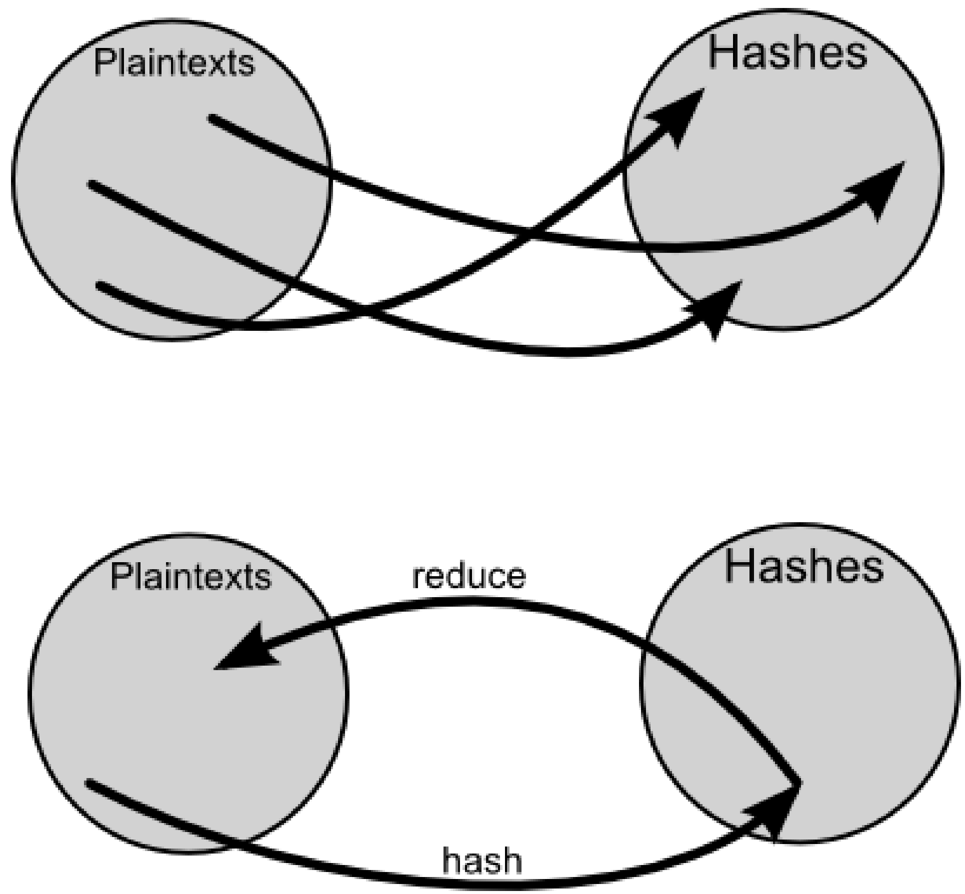
- Note that $R()$ is clearly ***not*** an inverse of $H()$
- Here is a **pre-computed hash chain**, which starts from a randomly-chosen word w_0 in ***D***

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

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 $\boldsymbol{(w_0, y_2)}$ in the data-structure ***T***
  5. Repeat the process with other randomly-chosen starting points

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

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

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

- 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 $\mathbf{w_2}$, 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)

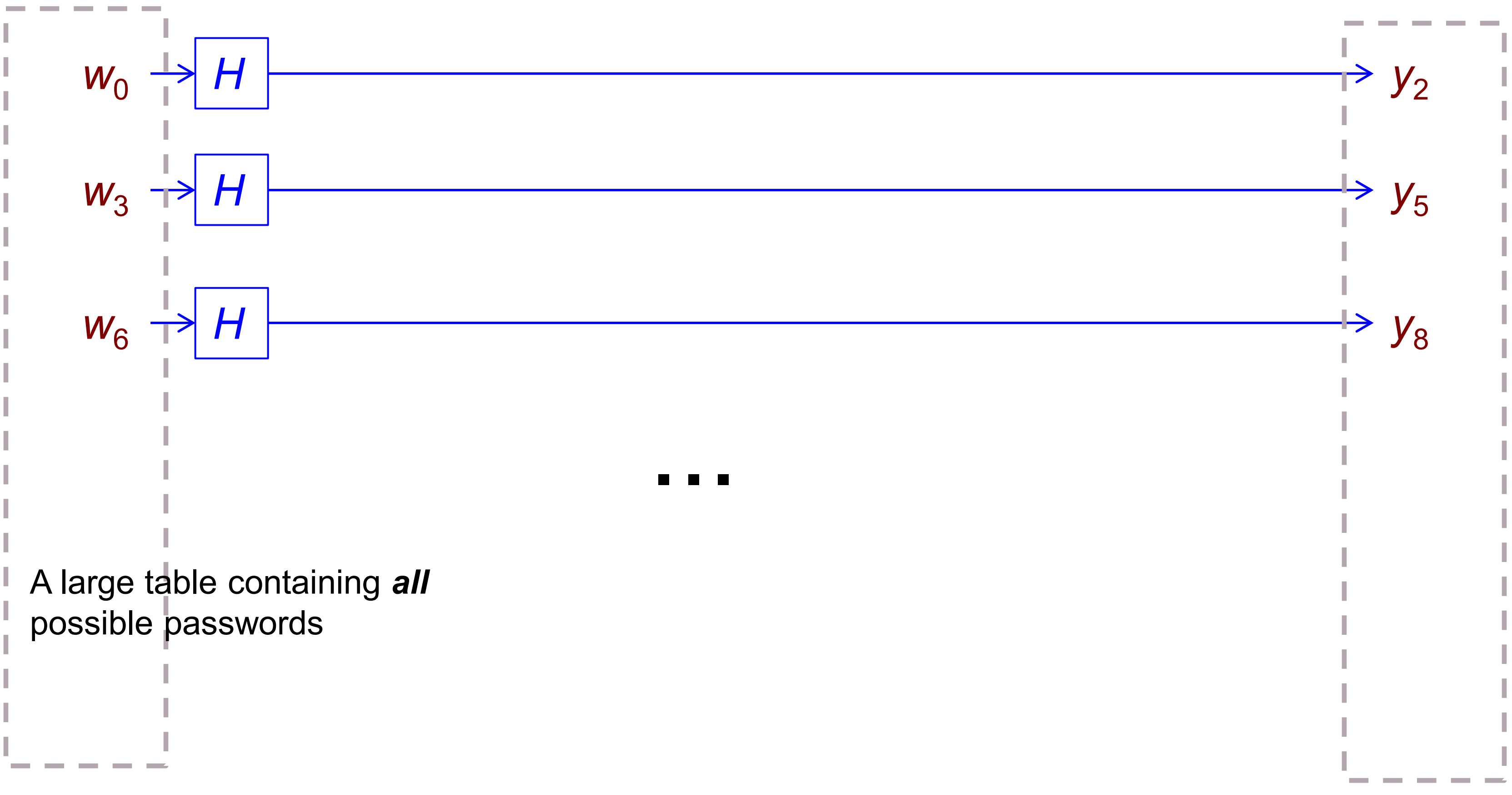
$$w_0 \rightarrow y_0 = H(w_0) \rightarrow w_1 = R(y_0) \rightarrow y_1 = H(w_1) \rightarrow w_2 = R(y_1) \rightarrow y_2 = H(w_2)$$

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

- Suppose y is ***not*** in $\boldsymbol{T}$:
  - Compute $y' = H(R(y))$
  - Search the data-structure for y'
  - Suppose y' is in $\boldsymbol{T}$:
    - 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 $\boldsymbol{T}$, compute $y'' = H(R(y'))$ and repeat this query process

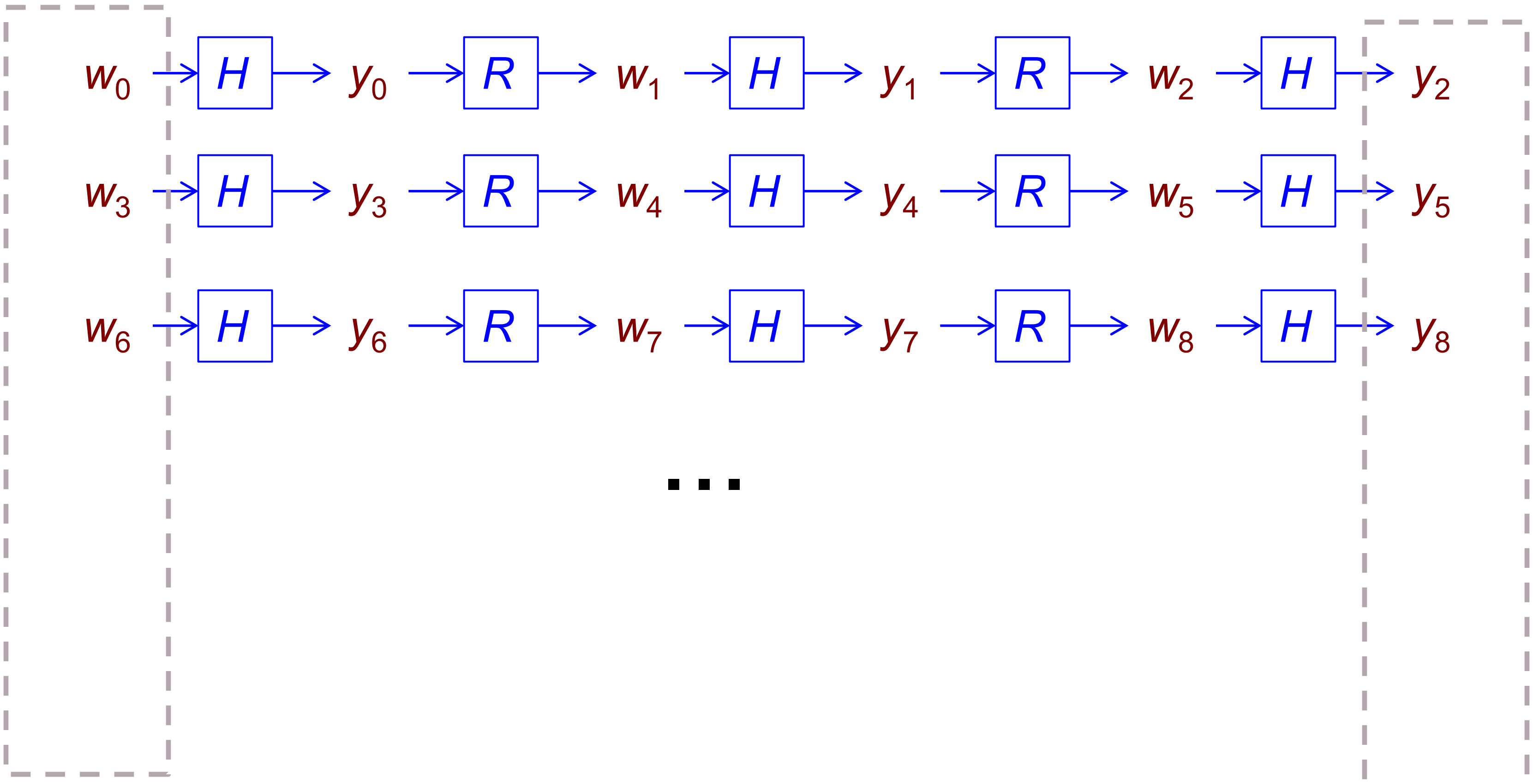
Without Hash Chains (As in Tutorial 1)

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

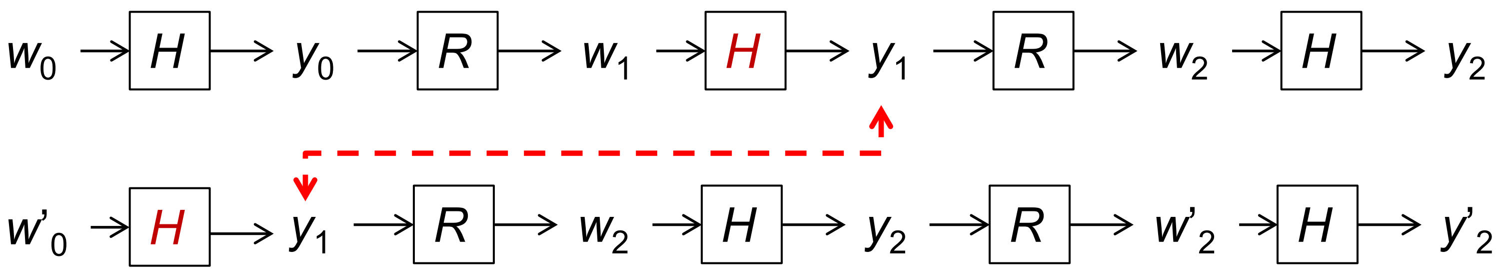
- Let's **compare** the required space & time of querying stored hash chains with the naive full table method:
  - **Space**: A reduction of space **by a factor of 3** (why?)
  - **Time per query**: the number of hash operations increases by a factor of 3, but also require 2 reduce operations (why?)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

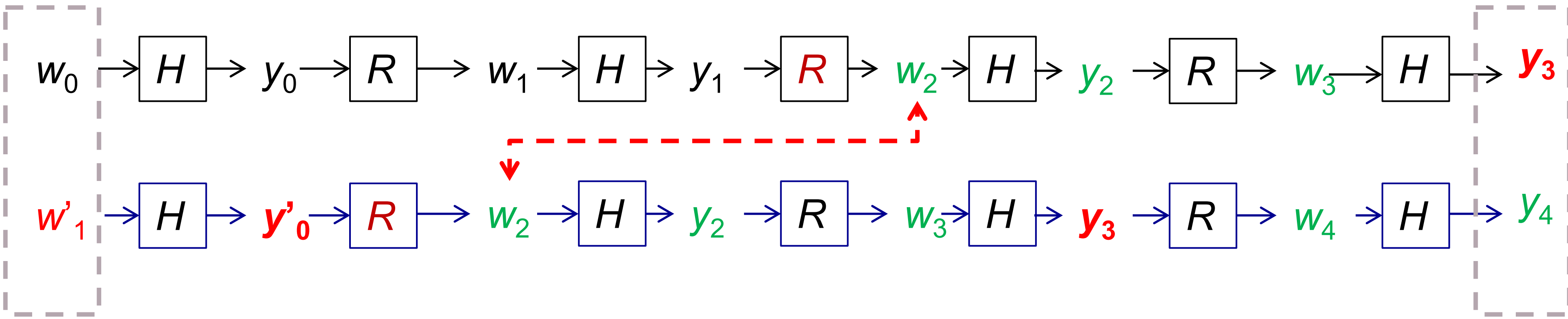
- The following **collision** is due to *H*():



- 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

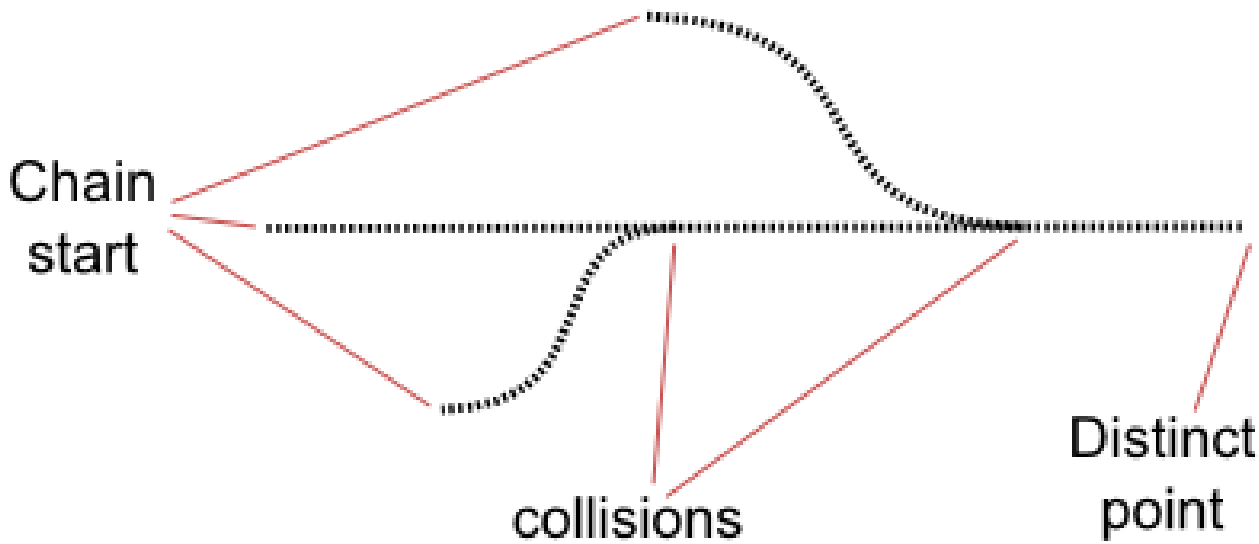


When given y_0', the algorithm is unable to find w'_1 in the first chain since:
The querying algorithm initially performs these steps:
(1) lookup y_0', (2) lookup y_2 (3) lookup y_3 (4) search the chain starting from w_0

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



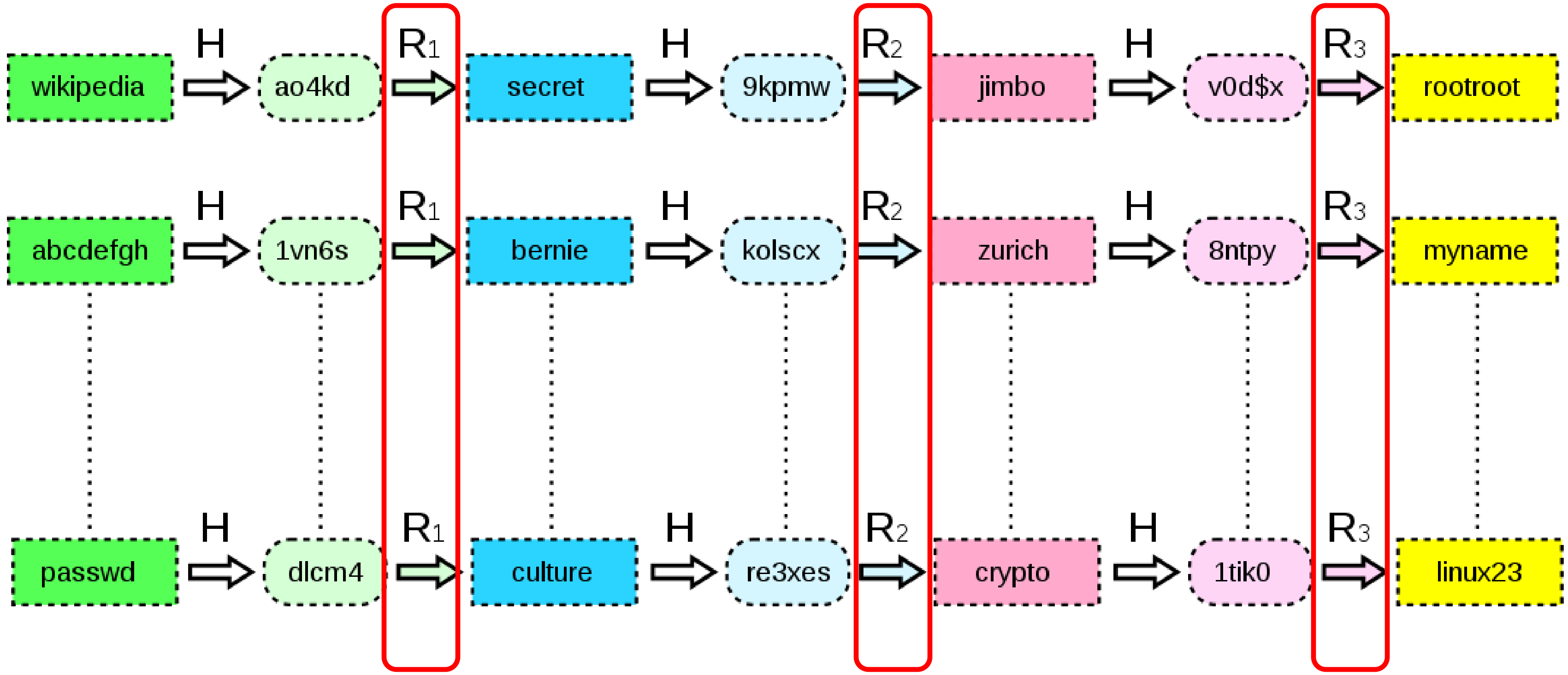
Source: http://kestas.kuliukas.com/RainbowTables/

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

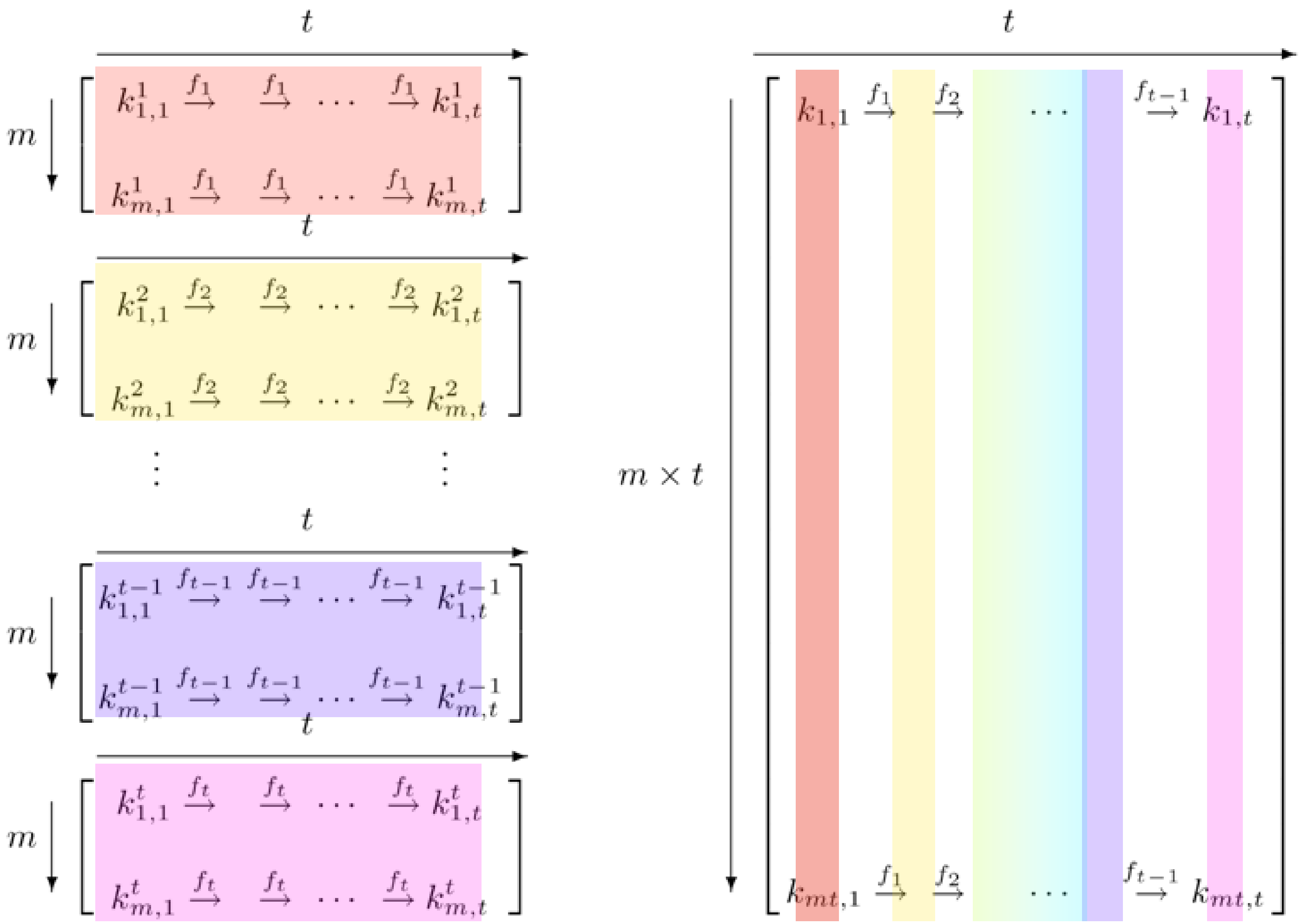
- To find out more, see:
  - http://en.wikipedia.org/wiki/Rainbow_table
  - http://kestas.kuliukas.com/RainbowTables/
- The original research paper:
  P. Oechslin, *Making a Faster Cryptanalytical Time-Memory Trade-Off*, CRYPTO 2003
  http://lasec.epfl.ch/~oechslin/publications/crypto03.pdf

Improved Variant: Rainbow Table



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

Improved Variant: Rainbow Table

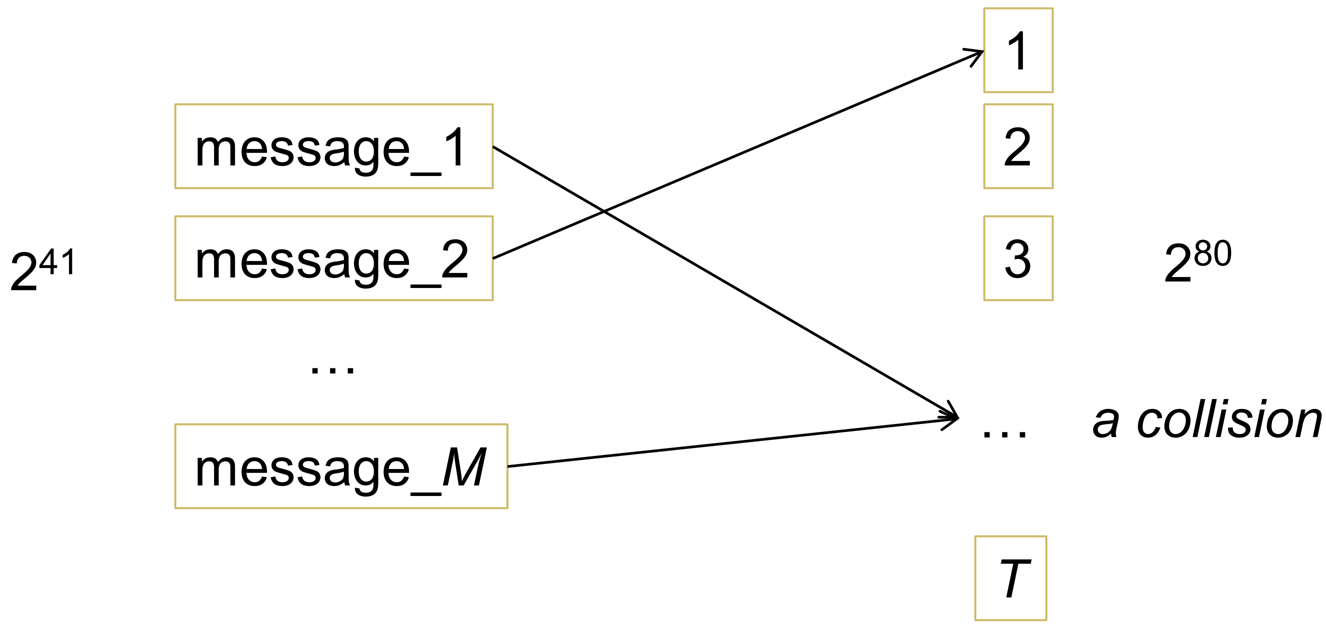


Source: Wikipedia

5.7 Birthday Attack Variant

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 $\boldsymbol{k}$ distinct elements, where each element is a $\boldsymbol{n}$-bit binary string
- Now, let us independently and randomly select $\boldsymbol{m}$ n-bit binary strings
- 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 of the generated m strings are *different*
- *How can we visualize this setting and formula?*

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**:
  - It enables its user to replace a ciphertext ***C*** **= E(*K*, *M*)** with another ciphertext ***C'*** **= 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!

5.9 Summary of Cryptography

Cryptography: Summary

- We have covered various cryptography topics in this module
- The **main objectives**:
  - Learn how cryptographic schemes and primitives work
  - Learn how to use them correctly
  - Learn how to reason about their security
- What cryptography **provides**?
  - It provides many useful primitives
  - It serves as the basis for many security mechanisms
- However, cryptography:
  - Is **not** the solution to all security problems:
    software vulnerabilities, social engineering attacks, etc.
  - Needs to be implemented and deployed **securely/properly**
  - Is not something you should invent/design yourself

Importance of Crypto?

Source: Wikipedia