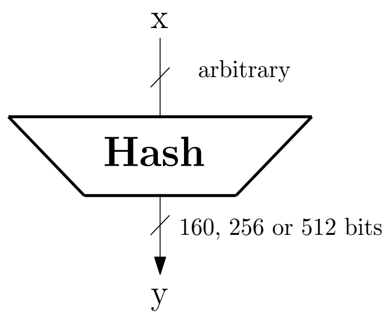
Crypto 8

Note: this material is not intended to replace the live lecture for students.

Contents

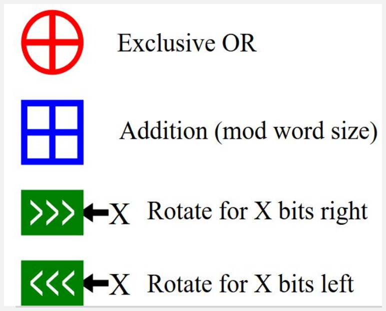


8.1 Hash : ARX

Many modern block ciphers and hashes are ARX algorithmstheir round function involves only three operations: modular addition, rotation with fixed rotation amounts, and XOR (ARX). Examples include ChaCha20, Speck, XXTEA, and BLAKE. Many authors draw an ARX network, a kind of data flow diagram, to illustrate such a round function.

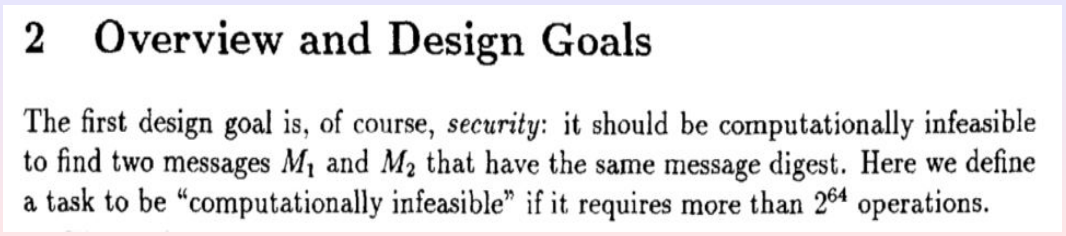
These ARX operations are popular because they are relatively fast and cheap in hardware and software, and also because they run in constant time, and are therefore immune to timing attacks. The rotational cryptanalysis technique attempts to attack such round functions.

ARX



Critics: https://keccak.team/2017/not_arx.html

8.1.1 MD4



2 Overview and Design Goals

The first design goal is, of course, *security*: it should be computationally infeasible to find two messages M_1 and M_2 that have the same message digest. Here we define a task to be "computationally infeasible" if it requires more than 2^{64} operations.

from [Rivest91, Section 2]

The MD4 output digest has 128 bits.
MD4 is based in Merkle-Damgård idea.
There is a padding: a single "1" bit is appended to the message, and then enough zero bits are appended so that the length in bits of the padded message becomes congruent to 448 modulo 512. Then the 64 bit representation of the pre-padded message is appended. If the length of the original message needs more than 64 bits then only the 64 lower-order bits are used.

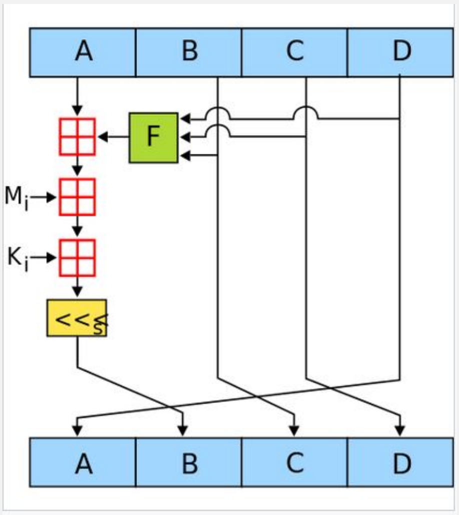
The blocks are of 512 bits divided in 16 words (32 bits).
The main register (called 4-word buffer by Rivest) has 4 words A B C D , see bellow figure.

Another design goal of MD4 was to favor little-endian architectures.

NOTE 8.1.1

In 2011 MD4 was considered historic (obsolete) `https://tools.ietf.org/html/rfc6150`: "In Section 6, this document discusses attacks against MD4 that indicate use of MD4 is no longer appropriate when collision resistance is required. Section 6 also discusses attacks against MD4's pre-image and second pre-image resistance. Additionally, attacks against MD4 used in message authentication with a shared secret (i.e., HMAC-MD4) are discussed."

Figure 8.1.2: MD4 Compression



One MD4 operation : MD4 consists of 48 of these operations, grouped in three rounds of 16 operations. *F* is a nonlinear function; one function is used in each round. M_i denotes a 32-bit block of the message input, and K_i denotes a 32-bit constant, different for each operation.

Exercise 8.1.3

What is the IV vector of MD4 ?

Exercise 8.1.4

Let M and N as follows

$M =$ 839c7a4d7a92cb5678a5d5b9eea5a7573c8a74deb366c3dc20a083b69f5d2a3bb3719dc69891e9f95e809fd7e8b23ba6318edd45e51fe39708bf9427e9c3e8b9

$N =$ 839c7a4d7a92cbd678a5d529eea5a7573c8a74deb366c3dc20a083b69f5d2a3bb3719dc69891e9f95e809fd7e8b23ba6318edc45e51fe39708bf9427e9c3e8b9

1) Check that $M \neq N$ and find all the differences. Hint: To check that $M \neq N$ hash them with Keccak-256 online.

2) Show that $\mathbf{MD4}(M) = \mathbf{MD4}(N)$.

8.1.2 SHA-1

8.1.5 SHA-1

Secure Hash Algorithm 1 (**SHA**-1), designed by NSA output a digest of 160 bit (20 byte).

SHA-1 is used by *Revision control systems* as Git to identify modifications of files.
Here an example of **SHA**-1 hash:

$$\textbf{SHA}\text{-}1(\text{ciao}) = 1e4e888ac66f8dd41e00c5a7ac36a32a9950d271$$

The input $x = \text{ciao}$ of 4 byte 'ciao' is converted by using ASCII:

$$x = 01100011011010010110000101101111$$

whilst the digest $'1e4e\cdots'$ is written in hexadecimal:

$$1e4e\cdots = 0001111001001110\cdots$$

We have broken **SHA-1** in practice.

8.1.3 The SHA-2 Family

Is a set of cryptographic hash functions designed by the United States National Security Agency (NSA).

They are built using the Merkle-Damgård structure together with a compression function obtained by Davis-Meyer.

The SHA-2 family consists of six hash functions with digests (hash values) that are 224, 256, 384 or 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256.

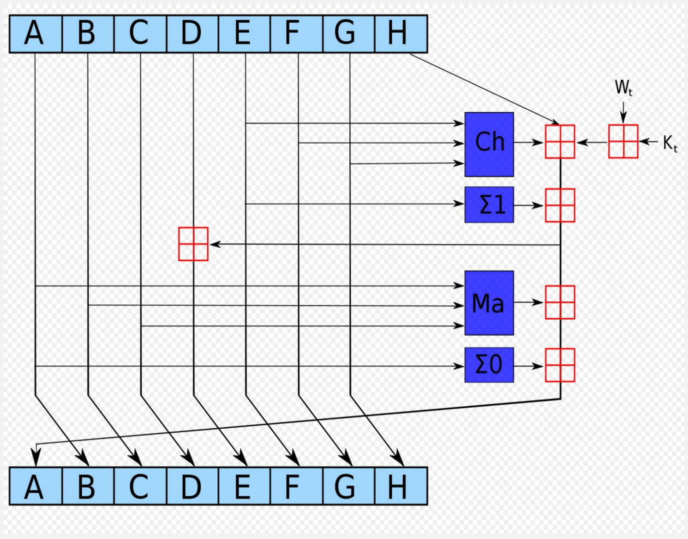
Here a description of SHA-256:
The message M is padded and parsed in blocks of 512 bit: $M^{(1)}M^{(2)}\cdots M^{(N)}$.

The blocks are processed one at time: beginning with a fixed initial hash value $H^{(0)}$:

$$H^{(i)} = H^{(i-1)} + \mathsf{Enc}_{M^{(i)}}(H^{(i-1)})$$

$H^{(N)}$ is the hash of M.

Figure 8.1.6: The round function used in $C_M(H)$



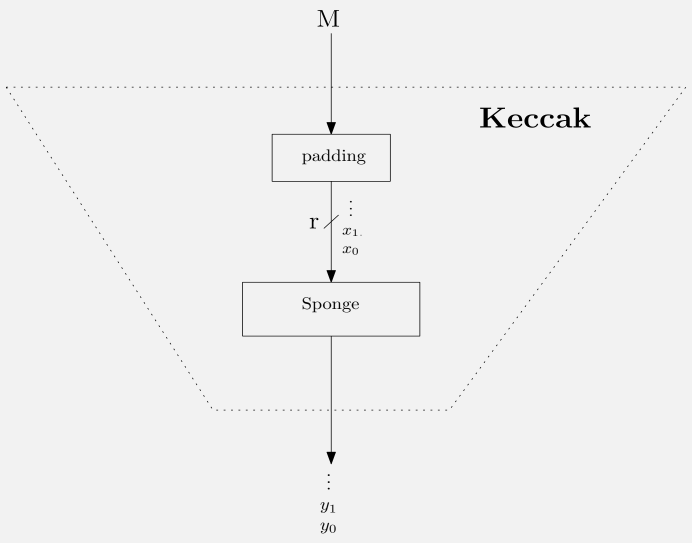
A B C D E F G H are words of 32 bits. Each 512 block $M^{(i)}$ is regarded as a key of a block-cipher Enc which cipher plaintext $H^{(i-1)}$. There 64 rounds, K_t are fixed constants (depending on the round) and W_t is part of the key-schedule of Enc.

8.2 Hash : Permutation

8.2.1 Keccak & the SHA-3 family

Keccak

Here it is:

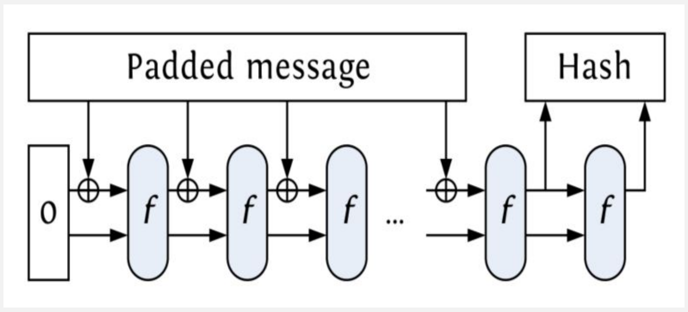


The padding is $x := M||\text{pad}[r](|M|)$ i.e.:

$$x = M||1\,0^*\,1$$

where 0^* denotes the minimum sequence of bits such that $|x|$ is multiple of r. Then x is parsed as $x = \cdots\,||x_1||x_0$ with $|x_i| = r$.

Here the sponge:



https://emn178.github.io/online-tools/sha3_512.html

8.2.2 The SHA-3 Family

NOTE 8.2.1

November 2, 2007: NIST's call for **SHA**-3. October 2, 2012: NIST chose Keccak as the base of **SHA**-3.
The standard FIPS PUB 202 gives the **SHA**-3 family:

The SHA-3 family consists of four cryptographic hash functions, called SHA3-224, SHA3-256, SHA3-384, and SHA3-512, and two extendable-output functions (XOFs), called SHAKE128 and SHAKE256.

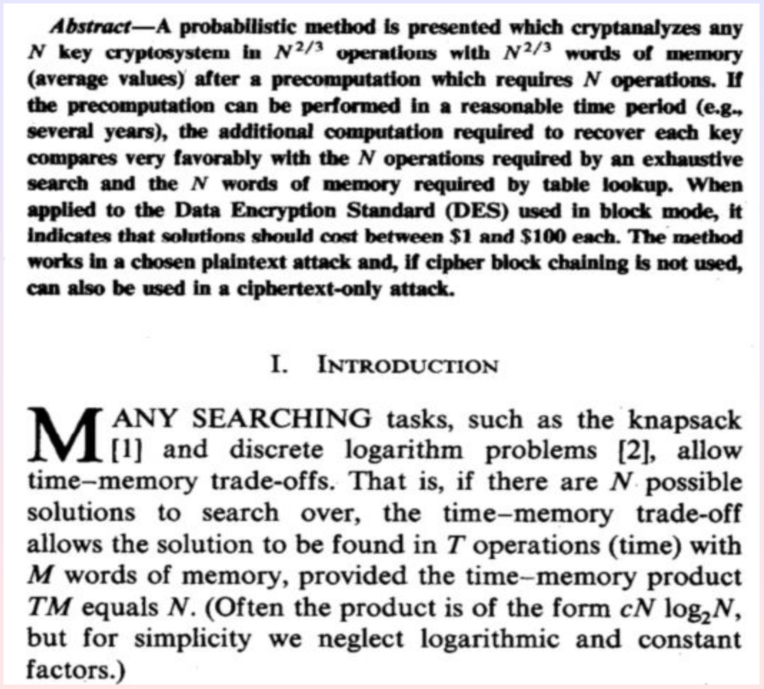
SHA-3 picks $b = 1600$. The rate r in **SHA**-3 could be $1152, 1088, 832, 576$ which determines the "security level" $\frac{c}{2}$. This corresponds to **SHA**-3-512,**SHA**-3-384,**SHA**-3-256,**SHA**-3-224.

The numbers $512, 384, 256$ are 224 the output bit dimension (i.e. digest length) of the four Hash **SHA**-3: they are the "least significative bits" of y_0, i.e. there is just one squeeze and just part of y_0 is used!.

The digest dimension of SHAKE128 e SHAKE256 is selected by the user. Here the numbers 128 e 256 refer to the "security strengths". This is very useful for digital signatures where the digest is signed with a **PKC** whose input is 2000 o 3000 bits like **RSA**.

There is also a difference in the padding (tra Keccak e **SHA**-3): **SHA**-3 do $x := M||0\,1\,1\,0^*\,1$. One reason of difference is to make different the digests of Keccak and of the standard. https://crypto.stackexchange.com/questions/60875/why-was-padding-changed-when-keccak-became-sha3

8.3 Hellman's and Rainbow tables



Abstract—A probabilistic method is presented which cryptanalyzes any N key cryptosystem in $N^{2/3}$ operations with $N^{2/3}$ words of memory (average values) after a precomputation which requires N operations. If the precomputation can be performed in a reasonable time period (e.g., several years), the additional computation required to recover each key compares very favorably with the N operations required by an exhaustive search and the N words of memory required by table lookup. When applied to the Data Encryption Standard (DES) used in block mode, it indicates that solutions should cost between \$1 and \$100 each. The method works in a chosen plaintext attack and, if cipher block chaining is not used, can also be used in a ciphertext-only attack.

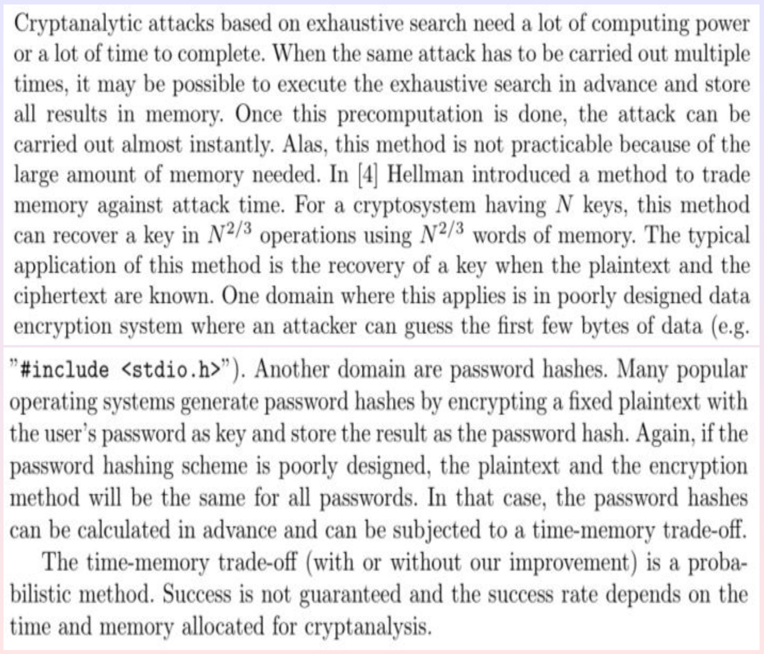
I. Introduction

MANY SEARCHING tasks, such as the knapsack [1] and discrete logarithm problems [2], allow time–memory trade-offs. That is, if there are N possible solutions to search over, the time–memory trade-off allows the solution to be found in T operations (time) with M words of memory, provided the time–memory product TM equals N. (Often the product is of the form $cN \log_2 N$, but for simplicity we neglect logarithmic and constant factors.)

From Hellman's paper [H80]

Look up table

The cryptanalyst enciphers some fixed plaintext P_0 under each $N = |\mathcal{K}|$ keys. This produce a table $(\mathsf{Enc}_k(P_0), k)$ which is sorted by $\mathsf{Enc}_k(P_0)$ and stored.
When a user chooses a new key k, he is forced (CPA-attack) to provide $\mathsf{Enc}_k(P_0)$ to the cryptanalyst. Since the look up table is sorted the cryptanalyst running time to find k is $O(\log(N))$.



Cryptanalytic attacks based on exhaustive search need a lot of computing power or a lot of time to complete. When the same attack has to be carried out multiple times, it may be possible to execute the exhaustive search in advance and store all results in memory. Once this precomputation is done, the attack can be carried out almost instantly. Alas, this method is not practicable because of the large amount of memory needed. In [4] Hellman introduced a method to trade memory against attack time. For a cryptosystem having N keys, this method can recover a key in $N^{2/3}$ operations using $N^{2/3}$ words of memory. The typical application of this method is the recovery of a key when the plaintext and the ciphertext are known. One domain where this applies is in poorly designed data encryption system where an attacker can guess the first few bytes of data (e.g. "`#include <stdio.h>`"). Another domain are password hashes. Many popular operating systems generate password hashes by encrypting a fixed plaintext with the user's password as key and store the result as the password hash. Again, if the password hashing scheme is poorly designed, the plaintext and the encryption method will be the same for all passwords. In that case, the password hashes can be calculated in advance and can be subjected to a time-memory trade-off.

The time-memory trade-off (with or without our improvement) is a probabilistic method. Success is not guaranteed and the success rate depends on the time and memory allocated for cryptanalysis.

From Oechslin's paper [Oe03]

Martin Hellman
Phillipe Oechslin

Hellman's and Rainbow tables

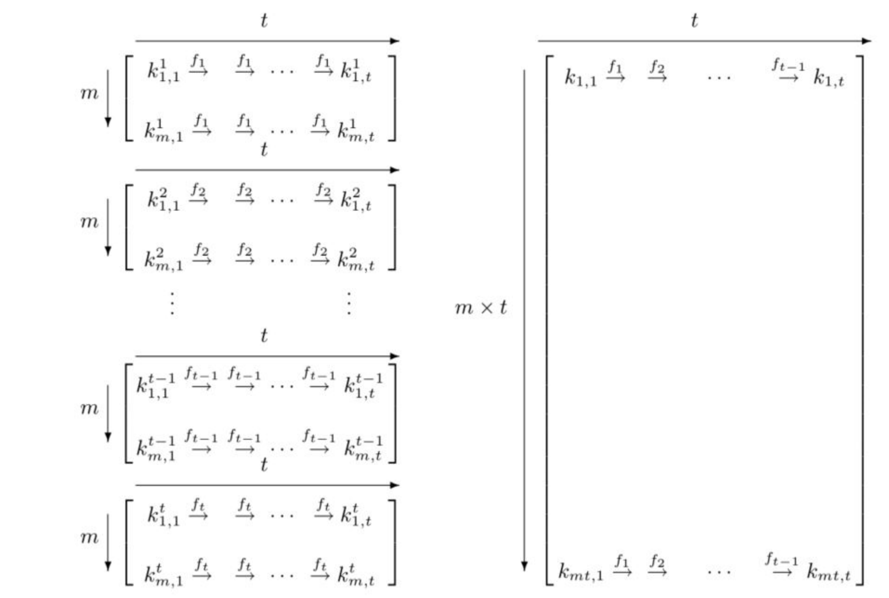


Fig. 2. t classic tables of size $m \times t$ on the left and one rainbow table of size $mt \times t$ on the right. In both cases merges can occur within a group of mt keys and a collision can occur with the remaining $m(t-1)$ keys. It takes half as many operations to look up a key in a rainbow table than in t classic tables.

Here are Hellman's two main ideas:

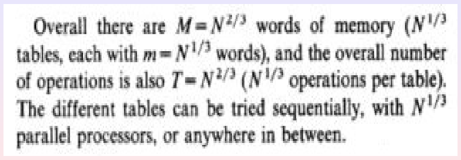
First one: In one classic Hellman's table m keys $k_1, \cdots, k_m$ are chosen uniform and random from the key space $\mathcal{K}$. To reduce memory just the first and last column are stored before been sorted by the last column.

Now assume that someone chooses a key k and the cryptanalyst intercepts $Y_1 = f(\mathsf{k})$.
So he checks if Y_1 is in the table. If yes then he knows the key is in the next to the last column of a given arrow hence he recover the key with $O(t)$ operations.
Otherwise he compute $Y_2 = f(Y_1)$ and checks if it is the last column.
Notice that to look up a key t search operations are necessary. If all $m \times t$ elements are different and random the success probability is $\frac{mt}{N}$.

Second one: t classical tables are generated using random "reductions" R. Namely, t random bijections R are chosen and $f_R := R \circ f$ is used as above keeping in mind that once intercepted $f(\mathsf{k})$ we have to check against $Y_1 = R(f(\mathsf{k}))$.
Now we have success probability $\frac{mt^2}{N}$. Notice that to look up a key $t^2 = t \times t$ search operations are necessary.

Exercise 8.3.1

Explain the number $N^{2/3}$ in Hellman's claim.


Overall there are $M=N^{2/3}$ words of memory ($N^{1/3}$ tables, each with $m=N^{1/3}$ words), and the overall number of operations is also $T=N^{2/3}$ ($N^{1/3}$ operations per table). The different tables can be tried sequentially, with $N^{1/3}$ parallel processors, or anywhere in between.

From Hellman's paper [H80]

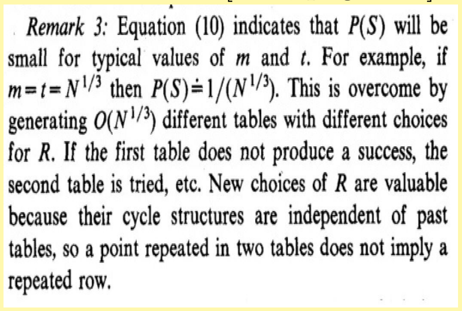
The rainbow table is generated by using also t reductions R as showed in figure above. Notice that to look up a key $\frac{t^2}{2}$ search operations are necessary.

Exercise 8.3.2

Why Rainbow's tables works better that Hellman's classic ones? Hint: collision, merge and operations to look up a key.

NOTE 8.3.3

Why is there an advantage if different R (reductions) functions are used line by line? Here an answer from [H80, page 403]:

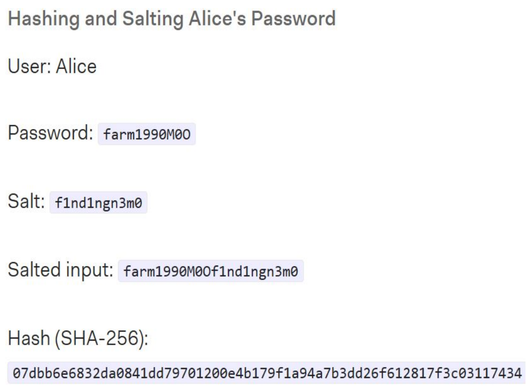

Remark 3: Equation (10) indicates that $P(S)$ will be small for typical values of m and t. For example, if $m=t=N^{1/3}$ then $P(S)\doteq 1/(N^{1/3})$. This is overcome by generating $O(N^{1/3})$ different tables with different choices for R. If the first table does not produce a success, the second table is tried, etc. New choices of R are valuable because their cycle structures are independent of past tables, so a point repeated in two tables does not imply a repeated row.

8.3.1 Salt

One of the earlier applications of cryptographic hash functions was the storage of passwords for user authentication in computer systems. With this method, a password is hashed after its input and is compared to the stored (hashed) reference password. People realized early that it is sufficient to only store the hashed versions of the passwords.

[Paar10, Chapter 11, page 315]

A *salt* is a random value appended to the password before hashing. Together with the hash, the value of the salt is stored in the database:



Hashing and Salting Alice's Password

User: Alice

Password: `farm1990M0O`

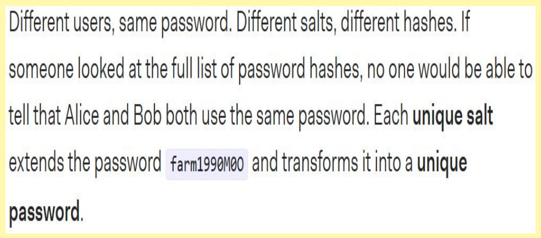
Salt: `f1nd1ngn3m0`

Salted input: `farm1990M0Of1nd1ngn3m0`

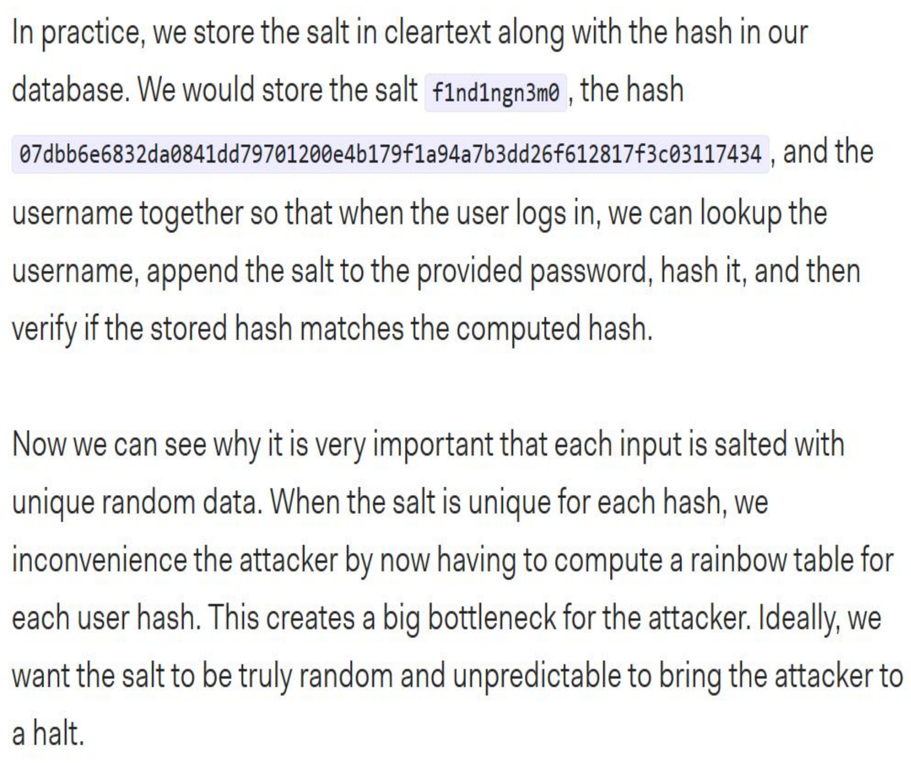
Hash (SHA-256):

`07dbb6e6832da0841dd79701200e4b179f1a94a7b3dd26f612817f3c03117434`

NOTE 8.3.4



Different users, same password. Different salts, different hashes. If someone looked at the full list of password hashes, no one would be able to tell that Alice and Bob both use the same password. Each **unique salt** extends the password `farm1990M0O` and transforms it into a **unique password**.



In practice, we store the salt in cleartext along with the hash in our database. We would store the salt `f1nd1ngn3m0`, the hash `07dbb6e6832da0841dd79701200e4b179f1a94a7b3dd26f612817f3c03117434`, and the username together so that when the user logs in, we can lookup the username, append the salt to the provided password, hash it, and then verify if the stored hash matches the computed hash.

Now we can see why it is very important that each input is salted with unique random data. When the salt is unique for each hash, we inconvenience the attacker by now having to compute a rainbow table for each user hash. This creates a big bottleneck for the attacker. Ideally, we want the salt to be truly random and unpredictable to bring the attacker to a halt.

How much salt ?

According to the NIST the salt shall be at least 128 bits, page 6 here: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf

But in other blogs 32 bytes are recommended:

https://www.mcafee.com/blogs/enterprise/cloud-security/what-is-a-salt-and-how-does-it-make-password-hashing-more-secure

https://cyberhoot.com/cybrary/password-salting/

As Bruce Schneier wrote in [Schneier15, page 53]:
Salt isn't a panacea; increasing the number of salt bits won't solve everything.

8.4 Bibliography

Books I used to prepare this note:

[Aumasson18] Jean-Philippe Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption*, No Starch Press, 2018.

[KatLin15] Jonathan Katz; Yehuda Lindell, *Introduction to Modern Cryptography* Second Edition, Chapman & Hall/CRC, Taylor & Francis Group, 2015.

[Paar10] Paar, Christof, Pelzl, Jan, *Understanding Cryptography, A Textbook for Students and Practitioners*, Springer-Verlag, 2010.

[Schneier15] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C* ,Wiley; 20th Anniversary edition,2015.

Here a list of papers:

[BeRo95] Bellare, Mihir and Rogaway, Phillip; *Random Oracles are Practical: A Paradigm for Designing Efficient Protocols*, Proceedings of the First Annual Conference on Computer and Communications Security https://cseweb.ucsd.edu/~mihir/papers/ro.pdf

[BDPA11] Bertoni,G.;Daemen,J.;Peeters,M. and Van Assche, G.: *The Keccak reference*, https://keccak.team/files/Keccak-reference-3.0.pdf

[BDPA11b] Bertoni,G.;Daemen,J.;Peeters,M. and Van Assche, G.: *Cryptographic sponge functions*, https://keccak.team/files/CSF-0.1.pdf

[DH76] Diffie, W.; Hellman, M. *New directions in cryptography*, (1976). IEEE Transactions on Information Theory. 22 (6): 644-654. https://ee.stanford.edu/~hellman/publications/36.pdf

[H80] Hellman, M. *A Cryptanalytic Time-Memory Trade-Off*, (1980). IEEE Transactions on Information Theory, vol. IT-26, no.4, July 1980: 401-406.

[Oe03] Oechslin, P. ; *Making a Faster Cryptanalytic Time-Memory Trade-Off*, Advances in Cryptology - CRYPTO 2003. LNCS. 2729. pp. 617630; (2003). https://lasec.epfl.ch/pub/lasec/doc/Oech03.pdf

[Rivest91] Rivest R.L. ; *The MD4 Message Digest Algorithm*, Menezes A.J., Vanstone S.A. (eds) Advances in Cryptology-CRYPTO 90. CRYPTO 1990. Lecture Notes in Computer Science, vol 537. Springer, Berlin, Heidelberg (1991). https://link.springer.com/content/pdf/10.1007%2F3-540-38424-3_22.pdf

and some interesting links:

http://professor.unisinos.br/linds/teoinfo/Keccak.pdf

https://csrc.nist.gov/csrc/media/projects/hash-functions/documents/keccak-slides-at-nist.pdf

http://passwords12.at.ifi.uio.no/Joan_Daemen_Passwords12.pdf

https://link.springer.com/content/pdf/10.1007/3-540-38424-3_22.pdf

https://emn178.github.io/online-tools/keccak_256.html

http://passwords12.at.ifi.uio.no/Joan_Daemen_Passwords12.pdf

https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passw