CSC281: Discrete Math for Computer Science

King Saud University
Computer Science Department Second Semester 1441/1442

February 25, 2020

Hashing Functions

You have to work in group of 3 students. Deadline: **Part (1)**: Monday 16^{th} of March 2020 (week 9) **Part (2)**: Monday 30^{th} of March 2020 (week 11).

As the majority of users will re-use their passwords among different applications, it is important to store passwords in a way that that prevents them from being obtained by an attacker, even if the application or database is compromised.
One way to protect user passwords is to store the hashed passwords instead of storing them in a clear text.
A hash function maps an arbitrary size data from large domain to a small fixed size codomain. Unlike encryption the hash functions are one-way functions (not invertible).

1 Ideally Hash function:

. Functions are not bijective: thus not invertible

. Uniformity: mapping should equally distribute (hash) elements in the domain over elements in the codomain

. Deterministic, meaning that the same message always results in the same hash

. Efficient to compute

. Infeasible to generate a message that yields a given hash value

. Infeasible to find two different messages with the same hash value

. A small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value (avalanche effect)

2 Applications:

. Hash table-based data structures (maps, caches, etc.)

. Message Authentication Codes (MAC): message digests

. Verifying the Data Integrity: Checksums

- Alice computes the hash of a message/file/data
- Alice sends the message and hash to Bob
- Bob computes the hash of the received message
- If the same: high confidence that the message was authentic and not tampered with
- If different: file integrity was compromised (or the hash)
- Oscar can still modify both and tamper with the message/file (but it should be difficult)

. Hashing passwords: to avoid storing passwords in clear text

- A users provides a password
- The password is hashed and the resulting hash value is stored
- Authentication is repeated by asking for the password, hash it, and compare to the stored hash
- The actually password is never stored and only ever exists in memory

3 Attacks:

3.1 Preimage attack:

Given: A hash function $h : X \to Y$,and $y \in Y$
Find: $x \in X$ such that $h(x) = y$
- Effective for password cracking (you have the hash, you need to find the password or provide an alternative, equivalent password)

3.2 Second Preimage attack:

Given: A hash function $h : X \to Y$ and $x \in X$
Find: $\acute{x} \in X$ such that $x \neq \acute{x}$ and $h(x) = h(\acute{x})$

3.3 Collision attack:

Given: A hash function $h : X \to Y$
Find: $x, \acute{x} \in X$ such that $h(x) = h(\acute{x})$
- Find two messages that have the same hash
- Compromises digital signatures: you can get Alice to sign document A and then use the signature on document B if $h(A) = h(B)$
- Brute force yield to $2^{co-domain/2}$ possibilities
- Birthday attack (Birthday paradox)

4 Part (1):

The user will provide the pair of (user-name, password) and your program should find the hash of password then, store user name and hashed password in a file.
Implement a simple hash function that sum up the ASCII of the password's symbols mod 39. Both
e.g.user name is **Maha** and password is **abc**, your program will store the following line :

Maha 21

where 21 is obtained from this calculation: abc $(97 + 98 + 99) mod\ 39 = 21$
You have to implement the following methods:

- map() Create a new, empty map. It returns an empty map collection.
- calculateHash(password) Take the plain password as input then, return the hashed password.
- register(user-name, password) Add the user name along with a hashed password to the map.
- retrieve() Extracts the content from the file to a suitable data structure.
- login(user-name, password) return true if the match exist false otherwise

In your report include the following:

a) List the problem/s in this hash implementation

b) Design an algorithm that is able to perform the mentioned attacks if they are possible.

c) How many trails to find a collision -if it possible- (provide the final number along with your calculations). Hint: use Generalized Pigeonhole Principle.

5 Part (2):

To avoid the vulnerabilities in part (1), implement a new hash function that operate as follow:
First symbol hash: $H(x) = (ASCII + 1)\ mod\ 39$
$H(x) = (ASCII^{previous-symbol-ASCII} + position)\ mod\ 39$
e.g. abc. $H(a) = (97 + 1)\ mod\ 39, H(b) = (98^{97} + 2)\ mod\ 39$, and $H(d) = (99^{98} + 3)\ mod\ 39$

You have to implement the following methods

- map() Create a new, empty map. It returns an empty map collection.
- findPower(ASCII, previous symbol ASCII, p) to calculate the power. (Use Fermats Little Theorem).

- calculateHash(password) Take the plain password as input then, return the hashed password.

- register(user-name, password) Add the user name along with a hashed password to the map.

- retrieve() Extracts the content from the file to a suitable data structure.

- login(user-name, password) return true if the match exist false otherwise

In your report include the following:

a) List the problem/s in this hash implementation, and in which point part 2 is better

b) Design a code that able to perform the collision attack.

c) How many possible trails to find a collision -if it possible- (provide the final number along with your calculations). Hint: use Generalized Pigeonhole Principle.

6 You have to submit:

Well commented code implementing the required functionality.
A report answering each part's questions

7 Grading

Criteria	Points
Submission (code + report)	1
Compilation	1
Comments	0.5
Test1-part (1)	2.75
Part (1) answers	1
Test2-part (2)	2.75
Part (2) answers	1
Total	10