Security

Continuous Assessment: Hash-cracking

Document version date: 15th September **2024**

Introduction

This assessment asks you to write and document your own hash-cracking algorithms.

This supports your understanding of cryptographic primitives that appear in the course, and vulnerabilities that can arise through usage and implementation.

This is an open-ended exercise. Better grades will be earned for work that impresses the marker(s).

A Note on Background

We may not have covered in lectures some of the concepts in the assessment long before the deadline, and certainly not before you want to start work on it. We often go through these in the lectures on authentication as well as the lectures on hashing. As a consequence, you will likely need to do some background reading on the following:

- Brute-force attacks against hashed passwords
- Dictionary attacks against hashed passwords
- Salting of passwords.

These topics are now well-known and quite standard. Wikipedia is a reasonable starting point, but there are plenty of other resources.

The Tasks

Before you begin, some notes:

(1) Assume ASCII printable characters throughout.
(2) Hashes are assumed to have been created using hashlib.sha512() of the hashlib module [1]. Be sure to read the description of the `update' method carefully.
(3) In your designs, you should assume that computing hashes is more expensive than searching lists. Real modern applications of hash algorithms for password hashing ensure that the hashing is sufficiently slow in order to obstruct attacks.

Task 1: Brute-force Cracking (10%)

Implement in Python an algorithm that searches for hashed passwords by *brute force*. The input should be a list of hashes. The output should be the corresponding list of passwords. You can assume that the password characters `a' to 'z' (lower-case) and '0' to '9'.

Hint: Follow a shortlex order [2] to ensure that you cover every possibility.

Demonstrate by finding the passwords that generate the list of four hashes:

```
['f14aae6a0e050b74e4b7b9a5b2ef1a60ceccbbca39b132ae3e8bf88d3a946c6d8687f3266
fd2b626419d8b67dcf1d8d7c0fe72d4919d9bd05efbd37070cfb41a',
'e85e639da67767984cebd6347092df661ed79e1ad21e402f8e7de01fdedb5b0f165cbb30a2
0948f1ba3f94fe33de5d5377e7f6c7bb47d017e6dab6a217d6cc24',
'4e2589ee5a155a86ac912a5d34755f0e3a7d1f595914373da638c20fecd7256ea1647069a2
bb48ac421111a875d7f4294c7236292590302497f84f19e7227d80',
'afd66cdf7114eae7bd91da3ae49b73b866299ae545a44677d72e09692cdee3b79a022d8dce
c99948359e5f8b01b161cd6cfc7bd966c5becf1dff6abd21634f4b']
```

These hashes have been chosen so that in this case all passwords should be found quite quickly - minutes at worst, but much, much faster is possible.

Task 2: Dictionary Cracking (15%)

Implement in Python an algorithm that searches for hashed passwords using a *dictionary attack*. The input should be a dictionary and a list of (unsalted) hashes. The output should be the corresponding list of passwords. Use the password dictionary from reference 3 below. Your solution for this task should avoid re-computing the same hash twice. You should assume that there are no repeated words in the dictionary and no repeated hashes in the input list of hashes.

Demonstrate by finding the passwords that generate the list of ten hashes:

```
['31a3423d8f8d93b92baffd753608697ebb695e4fca4610ad7e08d3d0eb7f69d75cb16d61c
af7cead0546b9be4e4346c56758e94fc5efe8b437c44ad460628c70',
'9381163828feb9072d232e02a1ee684a141fa9cddcf81c619e16f1dbbf6818c2edcc7ce2dc
053eec3918f05d0946dd5386cbd50f790876449ae589c5b5f82762',
'a02f6423e725206b0ece283a6d59c85e71c4c5a9788351a24b1ebb18dcd8021ab854409130
a3ac941fa35d1334672e36ed312a43462f4c91ca2822dd5762bd2b',
'834bd9315cb4711f052a5cc25641e947fc2b3ee94c89d90ed37da2d92b0ae0a33f8f7479c2
a57a32feabdde1853e10c2573b673552d25b26943aefc3a0d05699',
'0ae72941b22a8733ca300161619ba9f8314ccf85f4bad1df0dc488fdd15d220b2dba3154dc
8c78c577979abd514bf7949ddfece61d37614fbae7819710cae7ab',
'6768082bcb1ad00f831b4f0653c7e70d9cbc0f60df9f7d16a5f2da0886b3ce92b4cc458fbf
03fea094e663cb397a76622de41305debbbb203dbcedff23a10d8a',
'0f17b11e84964b8df96c36e8aaa68bfa5655d3adf3bf7b4dc162a6aa0f7514f32903b3ceb5
3d223e74946052c233c466fc0f2cc18c8bf08aa5d0139f58157350',
'cf4f5338c0f2ccd3b7728d205bc52f0e2f607388ba361839bd6894c6fb8e267beb5b5bfe13
b6e8cc5ab04c58b5619968615265141cc6a8a9cd5fd8cc48d837ec',
'1830a3dfe79e29d30441f8d736e2be7dbc3aa912f11abbffb91810efeef1f60426c31b6d66
6eadd83bbba2cc650d8f9a6393310b84e2ef02efa9fe161bf8f41d',
'3b46175f10fdb54c7941eca89cc813ddd8feb611ed3b331093a3948e3ab0c3b141ff6a7920
f9a068ab0bf02d7ddaf2a52ef62d8fb3a6719cf25ec6f0061da791'
]
```

All the passwords in this example appear in the dictionary.

Task 3: Dictionary Cracking with Salts (15%)

Implement a dictionary search for salted hashes. The input should consist of the dictionary and a list of pairs, where each pair consists of a salted hash and the corresponding salt (in the usual way). The output should be a list of passwords. Below is the input of 10 pairs to crack:

```
[('63328352350c9bd9611497d97fef965bda1d94ca15cc47d5053e164f4066f546828eee45
1cb5edd6f2bba1ea0a82278d0aa76c7003c79082d3a31b8c9bc1f58b',
  'dbc3ab99'),

('86ed9024514f1e475378f395556d4d1c2bdb681617157e1d4c7d18fb1b992d0921684263d
03dc4506783649ea49bc3c9c7acf020939f1b0daf44adbea6072be6',
  'fa46510a'),

('16ac21a470fb5164b69fc9e4c5482e447f04f67227102107ff778ed76577b560f62a586a1
59ce826780e7749eadd083876b89de3506a95f51521774fff91497e',
  '9e8dc114'),

('13ef55f6fdfc540bdedcfafb41d9fe5038a6c52736e5b421ea6caf47ba03025e8d4f83573
147bc06f769f8aeba0abd0053ca2348ee2924ffa769e393afb7f8b5',
  'c202aebb'),

('9602a9e9531bfb9e386c1565ee733a312bda7fd52b8acd0e51e2a0a13cce0f43551dfb3fe
2fc5464d436491a832a23136c48f80b3ea00b7bfb29fedad86fc37a',
  'd831c568'),

('799ed233b218c9073e8aa57f3dad50fbf2156b77436f9dd341615e128bb2cb31f2d4c0f7f
8367d7cdeacc7f6e46bd53be9f7773204127e14020854d2a63c6c18',
  '86d01e25'),

('7586ee7271f8ac620af8c00b60f2f4175529ce355d8f51b270128e8ad868b78af852a5017
4218a03135b5fc319c20fcdc38aa96cd10c6e974f909433c3e559aa',
  'a3582e40'),

('8522d4954fae2a9ad9155025ebc6f2ccd97e540942379fd8f291f1a022e5fa683acd19cb8
cde9bd891763a2837a4ceffc5e89d1a99b5c45ea458a60cb7510a73',
  '6f966981'),

('6f5ad32136a430850add25317336847005e72a7cfe4e90ce9d86b89d87196ff6566322d11
c13675906883c8072a66ebe87226e2bc834ea523adbbc88d2463ab3',
  '894c88a4'),

('21a60bdd58abc97b1c3084ea8c89aeaef97d682c543ff6edd540040af20b5db228fbce66f
ac962bdb2b2492f40dd977a944f1c25bc8243a4061dfeeb02ab721e',
  '4c8f1a45')
]
```

Again, all the passwords appear in the dictionary, and you should be able to find them all.

Task 4: Extensions (Miniproject, 60%)

Implement (and analyse, as appropriate) something more challenging. Examples, but you can do something else:

- Dictionary attack with “word-mangling”
- Add modes to handle special cases [4]

- Distributed cracking on a virtualised network with MPI
- Re-implement in C and compare performance, with graphs and statistics
- Use rainbow tables [5] (it is likely to be tricky to get this to work)
- GPU implementation
- Implement more serious key derivation as described for pbkdf2_hmac in the hashlib documentation [1] and references 6 and 7.

This task is intentionally open-ended to encourage deeper exploration, with higher grades awarded for solutions that show substantial effort, complexity, and creativity. The more thought and innovation applied, the greater the potential *reward*. Marks will be based on the quality of work, considering factors such as completeness, design decisions, and the overall execution, rather than through a deduction-based approach.

Ask (preferably in a tutorial class) if you are unsure if your idea is appropriate.

Avoid anything that involves experimenting with human users. There is insufficient time and there is too much bureaucracy around it.

Presentation of Your Work

You **must** submit both *runnable code* and a *readable report*.

The code

The code must be highly readable. Plenty of useful comments and good organization are expected. Good algorithm design underpinning the implementation is expected.

The report

The report must be very well presented. It must convey the necessary information to describe your work. It must be easy for the markers to follow what you are trying to communicate.

For each of the Tasks write a section of the report. Each section should have a header.

For Tasks 1-3 include a very concise description of *at most* 200 words on the algorithm design (rationale, how it works) underpinning your implementation.

For Task 4, include a concise description of *at most* 1000 words describing: (1) the goals, (2) the methods, (3) the conclusions, (4) what you learned.

For all parts, include the full code listing in an appendix. This will not be counted in the word count. Inclusion of pseudocode would be welcome too. Tables and graphics relating to performance can also be placed in the appendix and will not count towards the word count. Give figures, tables, listings captions and number them so that you can refer to them in the text and so that we can know what you are talking about.

You **must** give a list of references so that you can cite any work that you have built upon.

Submission and Marking

Marking will follow University policies, processes, and procedures.

The deadline is advertised on the course page. Take note of it and do not forget to submit on time. The standard penalties for late submission will apply.

You will be given a single CGS grade for the assessment. You will also be given textual feedback. Feedback is necessarily brief given the very short window we use for return of marks and feedback.

In deciding how interesting and well-executed an attempt at Task 4 is, the marker(s) will ask themself/themselves the following questions:

- [Communication] How well communicated was the work?
- [Knowledge and understanding] How much knowledge and understanding was demonstrated?
- [Skill demonstrated] How complex was what was done? How well was it done?
- [Work done] How much work was done?

Marking will align with the University CGS descriptors: A (excellent), B (very good), C (good), D (pass), E (weak), F (poor), G (very poor). The questions above will be used to do this.

Indicative (all submissions will have different strengths and weaknesses) characteristics of CGS bands in this context are in Table 1:

	Task 1	Task 2	Task 3	Task 4	Communication
A	Very well-executed	Very well-executed	Very well-executed	Very interesting, very well-executed	Strong
B	Well-executed	Well-executed	Well-executed	Interesting and executed well	Strong/good
C	Complete and competent	Complete and competent	Complete and competent	Good attempt	Good
D	Complete	Complete	Incomplete or lacking competence	Omitted or limited attempt	Acceptable
E and below	Incomplete	Incomplete	Incomplete	Omitted	Poor

Table 1: Indicative CGS Characteristics for this CA

Quality of execution and competence here include the quality of explanation and communication as well as the quality of the underlying work done.

Academic Integrity in this Context

The work should be your own **individual** work. It is **not** a group project. Submissions from different students are required to be different. **You <u>must</u> appropriately cite all outside sources used. This specifically includes code. This includes code that is open source. Use of automatically generated code is not allowed.** It is the case that there are online projects that overlap with this exercise. You can use them as a reference (remembering to cite them) if you are stuck with something, but it might be best if you tried to find your own direction first. Do not copy them completely, call the job done, and think that this will be sufficient to earn a good grade. Ask if you do not know what is allowed and what is not.

References

[1] *Hash Algorithms*, Python 3.10.7 documentation, Python Software Foundation , https://docs.python.org/3/library/hashlib.html#hash-algorithms (Accessed 13th September 2022)

[2] Wikipedia Contributors, *Shortlex Order*, Wikipedia, https://en.wikipedia.org/wiki/Shortlex_order (Accessed 13th September 2022)

[3] Peter Staev, Password dictionary, Github, https://gist.github.com/PeterStaev/e707c22307537faeca7bb0893fdc18b7 (Accessed 13th September 2022)

[4] *John the Ripper's cracking modes*, Openwall, https://www.openwall.com/john/doc/MODES.shtml (Accessed 13th September 2022)

[5] Wikipedia Contributors, *Rainbow table*, Wikipedia, https://en.wikipedia.org/wiki/Rainbow_table (Accessed 14th September 2022)

[6] Sönmez Turan, Meltem; Barker, Elaine; Burr, William; Chen, Lily. *"Recommendation for Password-Based Key Derivation Part 1: Storage Applications" (PDF)*. NIST. SP 800-132. *Available at https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf (Accessed 14th September 2022)*

[7] Wikipedia Contributors, *PBKDF2*, Wikipedia, https://en.wikipedia.org/wiki/PBKDF2 (Accessed 14th September 2022)