COM-402 Password Cracking and PAKE Implementation Homework Solution

Exercise 1: Password Cracking

Password cracking is kind of the *Hollywood* way of hacking. But contrary to what Horacio taught us, this is not always suitable. For example, against a web login, this is a very bad idea, as each try is monitored, takes a couple of seconds, and may result in you being blocked.

Though, there are certain situations when this is doable. In particular, when you have at hand encrypted passwords. But why would they be encrypted? And how?

When you deal with users and passwords, it's very *very* bad practice to store plaintext passwords. What you do is store the password as a hash and when the user logs in, you compute the hash of its input and compare it to the stored hash. You can also add some subtleties, such as salt or time-dependent hashes. But the idea is always the same.

Now, suppose your database gets hacked and dumped. Everyone on the internet has access to your database. Now, hackers have a list of hashes, that match usernames. This is, in itself, useless as the hashes are non-reversible and can't be used to log in. That's when password cracking shines: you have at hand a bunch of hashes, and your goal is to reverse them to find the original passwords.

Note: For all password cracking, it's always a good idea not to try and find one hash, but to compute a hash and compare it to all the unsolved hashes. This will save a lot of time, rather than doing the whole process multiple times.

Part 1a: Brute force attack

Understanding the task

Bruteforce attacks are the simplest form of password cracking. Basically, you try all the combinations one after the other. Given that you do it well, this method is guaranteed to reverse the hash... some day. Because the complexity and thus length of the task is directly proportional to the length of the password, and the size of the character set. More precisely: say you try all combinations from a set of 36 characters (lowercase letters and all numbers), with a password length of 5, you'll have to try $36^5 = 60466176$ different passwords.

For example, a hacker knowing that all passwords consist of fewer than 5 characters with only lowercase letters and numbers will laugh. But having upper and lower cases, numbers, some special characters and a 16-length password, you're in for about 10^{29}. Practically unbreakable.

Here we'll try and hack $36^4 + 36^5 + 36^6 = 2238928128$ passwords. This is a relatively imposing number, but a laptop on a CPU can try all of them within a few hours.

The methods

There are several methods for you to try and hack the hashes. In a real scenario, you'd want to rely on existing software. The most well-known and used are John the Ripper and Hashcat. They support many hash types, are optimized to run as fast as possible and can even leverage your GPU(s).

For the sake of comprehension, here we will use a custom python script. Python is great, but not necessarily in this situation, because it's a quite heavier language, compared to say C or C++. Nonetheless, this is far simpler to demo on that.

Code

```python
import itertools
import hashlib
import time
```

```python
# all the possible characters
charset = "abcdefghijklmnopqrstuvwxyz1234567890"

# the list of all the hashes
all_hashes = set([
    "7c58133ee543d78a9fce240ba7a273f37511bfe6835c04e3edf66f308e9bc6e5",
    "37a2b469df9fc4d31f35f26ddc1168fe03f2361e329d92f4f2ef04af09741fb9",
    "19dbaf86488ec08ba7a824b33571ce427e318d14fc84d3d764bd21ecb29c34ca",
    "06240d77c297bb8bd727d5538a9121039911467c8bb871a935c84a5cfe8291e4",
    "f5cd3218d18978d6e5ef95dd8c2088b7cde533c217cfef4850dd4b6fa0deef72",
    "dd9ad1f17965325e4e5de2656152e8a5fce92b1c175947b485833cde0c824d64",
    "845e7c74bc1b5532fe05a1e682b9781e273498af73f401a099d324fa99121c99",
    "a6fb7de5b5e11b29bc232c5b5cd3044ca4b70f2cf421dc02b5798a7f68fc0523",
    "1035f3e1491315d6eaf53f7e9fecf3b81e00139df2720ae361868c609815039c",
    "10dccbaff60f7c6c0217692ad978b52bf036caf81bfcd90bfc9c0552181da85a"])

begin = time.time()

for length in range(4, 7):
    # define all possible combinations of a certain length
    for comb in itertools.product(charset, repeat=length):
        # change "('a', 'a', 'a', 'a')" to 'aaaa'
        password = "".join(comb)
        # compute the hash
        h = hashlib.sha256(password.encode())
        digest = h.hexdigest()

        if digest in all_hashes:
            # if one hash match, print and remove (to save some time)
            print("{} === {}".format(digest, password))
            all_hashes.remove(digest)
        else:
            # print your progress
            print(password, end="\r")

end = time.time()
print("Time for naïve: {:.3f}".format(end-begin))
```

This code runs within a few hours on a laptop CPU. Not great, not terrible.

We obtain the following corresponding passwords:

```
7c5813... === xex167
37a2b4... === xontbc
19dbaf... === szpn9
06240d... === feh9ay
f5cd32... === 7rimq7
dd9ad1... === gi02n
845e7c... === j67c
a6fb7d... === bgfvf
1035f3... === 2vdxm
10dccb... === 26i4id
```

As you probably realize, this kind of attack can't be reasonably used at a large scale. This is the last resort method, that can't be deployed for large passwords, or too complicated passwords.

Leveraging your multi-core machine

Though, this task is embarrassingly parallel. If you had a thousand threads, you can easily split your task among them, and your computation would be roughly a thousand times faster. Good libraries such as Hashcat or John actually do that, and can use your CPU at its maximum capacity. This is also how GPUs are leveraged: they contain thousands of "weak cores". Individually they are not great for heavy computing, but for such task they are perfect.

Here is an example of a parallelized code that uses your CPU resources a bit better:

```python
import hashlib
import multiprocessing
import time
import itertools

def block_func(combination):
    """Take one combination, compute its hash, and return if a match is found"""
    password = "".join(combination)
    h = hashlib.sha256(password.encode())
    digest = h.hexdigest()
    if digest in all_hashes:
        all_hashes.remove(digest)
        print("{} === {}".format(digest, password))
        return digest

# all the possible characters
charset = "abcdefghijklmnopqrstuvwxyz1234567890"

begin = time.time()

# create a pool of processes
pool = multiprocessing.Pool(processes=16)
total_matches = []
for length in range(4, 7):
    # create the set of passwords for this length
    combinations_generator = itertools.product(charset, repeat=length)

    # compute number of possible passwords
    total_combinations = len(charset)**length

    #read the generator lazily and map combinations to the function
    for i, x in enumerate(pool.imap_unordered(block_func, combinations_generator, 1000)):
        if x is not None:
            total_matches.append(x)
        if i % 100 == 0:
            # display your progress
            print("{:.3f}".format(100*i/total_combinations), end="\r")
```

```
end = time.time()
print("Time for naïve: {:.3f}".format(end-begin))
```

Code explanation

The above code is quite simple, but not perfect. It uses `imap_unordered`, which will do the following: - Read the generator lazily by chunks of 1000 (you don't want to have *all* the combinations at once in memory); - Then the 16 workers generated earlier will pick the combinations one after the other and apply each combination to the defined function; - Because it's unordered, if a worker finishes early, he doesn't have to wait until his predecessor is done, because the output can be unordered.

This leverages the computing power of your CPU better, but is still not perfect. The main defect is that it creates a lot of computing overhead. Each "block" (a combination) is quite fast, so a worker is often starting and finishing. It would be better to give to each worker a chunk of combinations, and they each spend some time iterating within them. But for now, good enough.

Part 1b: Dictionary attack with rules

As you saw in the previous part, exhaustive attacks while very powerful are not a viable solution for many passwords. But (un)fortunately, humans are not random machines, and strongly dislike passwords of the type `dn8S89?/8+F\f'hk7|è`, while they can simply remember `password123`. Even in 2021, those are still awfully used.

How it works

The idea behind dictionary attacks is just that: humans are more likely to generate their passwords using words from a dictionary. So if you have your hand on a dictionary, you can try all the words in it. For all intents and purposes, a dictionary is a list of unique words, likely to be used by humans.

Let's start with a simple dictionary. Take the Oxford dictionary, it contains about 171k words. Not too bad, finding all the hashes may take a couple of seconds on a laptop CPU. You may find a few passwords with that.

But what if someone uses `Password`? Well, that's the next step. Now that you have your list of words, it's better to try some tweaks. Try some capital letters, change letters (`o -> O -> 0`, `A -> 4`). The next step is to combine words. Concatenate 2,3,4 or 10 words, try combinations.

And what about `motdepasse`? Yet another step. Your dictionary may not be exhaustive, and you only cover English speakers. Well, download a French, German, or Finnish dictionary, and add them to your initial one.

And what about using your mother's name, Martha, in the password? Yes, possible. Retrieve a list of human names. And also city names. And celebrities. And events. And slurs. And everything. Oh, and don't forget non-alphanumerical characters (`!,ç%&+"...`)

It's exploding!!

By doing all these steps, you realize you now have a database of several billion or more words to try. Are you really in a better spot compared with brute force? Actually yes. Because now you have the same space size (say, 10^{12} passwords), but with brute force that only covered the passwords less than 5 characters, on a small character set. Now your huge dictionary covers `P4ssw0rd123!` which is more likely to be used rather than `4D/x.`. With brute force, you'd never reach that password.

Hackers spend years enhancing their dictionaries. If you're smart, you also sort these words in a manner that allows smart selection. For example, you're hacking a German music platform. Chances are, your Finnish and fishing-related dictionaries are not useful there. If you target a specific person or entity, you also must have a "personal" dictionary (their pet's name, date of birth, significant other's name,...).

Having a good dictionary is tedious, but may really help you when cracking passwords.

Which dictionary

On the recommended website, you can find all sorts of dictionaries. Some are plain "dictionaries" (list of US cities for example), while there are also a bunch of lists of passwords. You'll find a more complete one on crackstation (either the full 15 GiB one with all words in Wikipedia and words from project Gutenberg, or the smaller, 650 MiB one, with only dumps of passwords). All the hashes we gave you come from the *rockyou.txt* dictionary. It's a very complete and famous leak, containing a lot of unencrypted passwords, making it a very interesting tool.

For the sake of completeness, we will assume this is not the case, and will explore multiple dictionaries.

Parallelism

As before, this task is embarrassingly parallel. Each entry in your dictionary is independent of the others (this is not exactly true, but good enough for us). Thus, to gain a lot of time, you can re-use some code from before and leverage your cores.

Commutativity

On your first try, you may try the following: take all your modifications, and create all the combinations (4 modifications, that gives 16 different outcomes, including the "no-modify" one). While this is absolutely OK, there may be one or more passwords that will remain unbreakable. The reason for that is commutativity. The first modification possible (change capital letters) will be affected by the other transformations.

Take `window`, the example given. If you first *title-ize* it, then replace the numbers, you'll get `W1nd0w`. If you first replace the `is`, then *title-ize*, then replace the os, you'll get `W1Nd0w`. And first replacing the numbers, then *title-ize*, you obtain `W1Nd0W`. These three will yield different hashes.

It is thus important to take all combinations, with respect to the order. That gives 65 possible combinations of these modifications (again, including the empty modification). Don't worry though, if you code it smartly, many combinations are commutative, and this rarely matters. After filtering, you'll have to compute only a handful of versions of the password, usually between 1 and 20.

The size of files

All the files from *skullsecurity* are relatively small, and fit in memory. This is not always the case (for example, the 15 GiB file from *crackstation*). For this reason, you must think about it and work with the file open.

The code (finally)

Generating the combinations One good way to generate all combinations of modifications is to take advantage of the fact that Python deals with functions as objects. This allows to create an array of the unique modifications, and then create combinations from that. Here, we define named functions for the sake of completeness, but as the functions are almost trivial, lambda functions could have done the trick.

Also, we use hints, available from python 3.5, that make the development a lot easier.

```python
def modif1(p: str):
    return p.title()

def modif2(p: str):
    return p.replace("e", "3")

def modif3(p: str):
    return p.replace("o", "0")

def modif4(p: str):
    return p.replace("i", "1")

all_modifs_combinations = set()
all_modifs = [modif1, modif2, modif3, modif4]
for length in range(1, len(all_modifs)+1):
    for comb in itertools.permutations(all_modifs, length):
        all_modifs_combinations.add(comb)
```

Creating the combinations

```python
# let p be a given entry from the dictionary
all_versions = set([p])
for comb in all_modifs_combinations: # go through each set of modifications
    p_temp = p
    for modificator in comb: # apply them fo the base password iteratively
        p_temp = modificator(p_temp)
    all_versions.add(p_temp) # all_versions is a set, so no duplicate will remain
```

Hashing

```python
for version in all_versions:
    hash = hashlib.sha256(version.encode()).hexdigest()
    if hash in all_hashes:
        print("{} === {} (from {})".format(hash, version, p))
        all_hashes.remove(hash)
        return p, hash
```

Complete, with parallelism

```python
import hashlib
import itertools
import multiprocessing

all_hashes = set([
    "...",
    "...",
    "..."
])


def modif1(p: str):
    return p.title()
def modif2(p: str):
    return p.replace("e", "3")
def modif3(p: str):
    return p.replace("o", "0")
def modif4(p: str):
    return p.replace("i", "1")

def modif_and_hash(p):
    """Try all modifications and hashes from one given base password."""
    if len(all_hashes) == 0:
        return
    p = p.replace("\n", "") # ensure there are no \n at the end
    # create the alternative from the base passwords
    all_versions = set([p]) # include the base password
    for comb in all_modifs_combinations:
        p_temp = p
        for modificator in comb:
            p_temp = modificator(p_temp)
        all_versions.add(p_temp)
    # Hash and compare each of them
    for version in all_versions:
        hash = hashlib.sha256(version.encode()).hexdigest()
        if hash in all_hashes:
            print("{} === {} (from {})".format(hash, version, p))
            all_hashes.remove(hash)
            return p, hash


# define the name of all dictionaries, along with their encoding
dictionaries = [
    ("500-worst-passwords.txt", "utf-8"),
    ("alypaa.txt", "utf-8"),
    ("cain.txt", "utf-8"),
    ("carders.cc.txt", "latin-1"),
    ("conficker.txt", "utf-8"),
    ("english.txt", "utf-8"),
    ("elitehacker.txt", "utf-8"),
    ("facebook-pastebay.txt", "utf-8"),
    ("facebook-phished.txt", "latin-1"),
    ("faithwriters.txt", "utf-8"),
    ("file-locations.txt", "utf-8"),
    ("fuzzing-strings.txt", "utf-8"),
    ("german.txt", "latin-1"),
    ("hak5.txt", "utf-8"),
    ("honeynet.txt", "latin-1"),
    ("hotmail.txt", "utf-8"),
    ("john.txt", "utf-8"),
    ("phpbb.txt", "latin-1"),
    ("phpmyadmin-locations.txt", "latin-1"),
    ("singles.org.txt", "utf-8"),
    ("tuscl.txt", "latin-1"),
    ("twitter-banned.txt", "utf-8"),
    ("us_cities.txt", "utf-8"),
    ("web-extensions.txt", "utf-8"),
    ("web-mutations.txt", "utf-8"),
    ("rockyou.txt", "latin-1"),
    ("crackstation.txt", "latin-1")
]
all_modifs_combinations = set()
all_modifs = [modif1, modif2, modif3, modif4]
for length in range(1, len(all_modifs)+1):
    for comb in itertools.permutations(all_modifs, length):
        all_modifs_combinations.add(comb)

# iterate through all dictionaries
for fname, encoding in dictionaries:
    print("Opening file {}".format(fname))
    file = open("Dictionaries/{}".format(fname), encoding=encoding)
    pool = multiprocessing.Pool(8) # define a pool of 8 workers
    results = []

    # read the file by chunks of 10k rows at a time, then feed one
    # rows one after the other to a worker
    for r in pool.imap_unordered(modif_and_hash, file, 10000):
        if r is not None:
            results.append(r)
    file.close()
```

Passwords

- `2e41f7133f... === alban1an123`
- `7987d2f5f9... === Dalbc00s`
- `076f8c265a... === Oscar00`
- `b1ea522fd2... === captpimp`
- `3992b888e7... === B3Llucc1`
- `326e90c0d2... === delmar00`
- `2693983012... === 6033503`
- `4fbee71939... === 802561`
- `55c5a78379... === 1M0nj1tas`
- `5106610b8a... === 19387`

Final thoughts

As you can observe, this method allows to test not more or less, but different passwords. If you were absolutely positive the password was random (e.g. generated by a password manager), then this method is useless. But against passwords of most people, this is a very powerful attack. No wonder hackers take years to complete their own dictionaries! Also, just as before, it would be possible to generate in advance a list of all hashes, and create a lookup table (or a rainbow table, courtesy of Philippe Oeschlin, teacher of fall 2020). Next exercise will show you one protection against that.

Part 1c: Hashes get salty

Why use salt?

The previous attack, while powerful in theory, is often countered. If you push forward the reasoning there once you have your list of words, you can just compute a lookup table of all the hashes of all those passwords, and use it. You just have to do the work once, and then de-hashing is $\mathcal{O}(1)$! Good for us, no good for others. But if every password you store has a "unique" salt at the end, this "do the work once" must be done once per salt. Supposing, as mentioned, a 2 hexadecimal characters salt, this increases your work by a factor 256! The good thing is that you don't even need to keep the salt secret (well, it's always better). If the password is salted, chances are there is no lookup table for those hashes.

WPA2 Trivia

With WPA2 Personal, routers salt the password with the SSID of the network. But some big routers always came with the same default SSID that people never bothered changing (*linksys*, *asus*, *orange*,...). Some are so common that people started collecting them. Once you have the most common, you can start creating lookup tables and rainbow tables, specific for each of these common SSIDs. For this reason, amongst others, modern routers now come with a default SSID appended by some pseudo-random value (like *linksys-SJF3O-F82ND-F8F3K*).

How to attack

The idea is very similar to the previous exercise. Now, instead of applying some functions to your password, you just try to append each salt and try it. Because we provide you with all the salts used, you only have to generate 10 versions for each password you consider, and hash each of them. But if we hadn't, or if the salt was partially dynamic, we would need to try all the salts for all the passwords. For example, passlib (a standard authentication library for python) lets you specify a 0-16 alphanumerical characters salt or will generate one at random each time. With such method, you can't rely on rainbow tables at large scales.

The code

We mostly reuse the code from before.

Salt and hash

```python
def salt_and_hash(p):
    """Take one password, and hash it using all the possible salts."""
    salts = ["b9", "be", "bc", "72", "9f", "17", "94", "7f", "2e", "24"]
    p = p.replace("\n", "") # remove possible trailing \n
    for s in salts:
        salted = p+s
        hash = hashlib.sha256(salted.encode()).hexdigest()
        if hash in all_hashes_c:
            print("{} === {} (salt {})".format(hash, p, s))
            return p, hash
```

Parallelizing

```python
dictionaries = [
    ...
    ("rockyou.txt", "latin1"),
    ("crackstation.txt", "latin-1")
]
# try multiple files
for fname, encoding in dictionaries:
    print("Opening file {}".format(fname))
    file = open("Dictionaries/{}".format(fname), encoding=encoding)
    pool = multiprocessing.Pool(8) # 8 concurrent workers
    results = []
    # open the file 10k rows at a time, and feed them to "salt_and_hash"
    for r in pool.imap_unordered(salt_and_hash, file, 10000):
        # store your results, for later use
        if r is not None:
            results.append(r)
    file.close()
```

Passwords

- `962642e330bd === albanian123` (salt `b9`)'
- `8eef79d547f7 === thesims9` (salt `be`)
- `e71067887d50 === jasnic1` (salt `bc`)
- `889a22781ef9 === atychi1` (salt `72`)
- `6a16f9c6d954 === solkingfran` (salt `9f`)
- `2317603823a0 === kapono` (salt `17`)
- `c6c51f8a7319 === kaylahh1` (salt `94`)
- `c01304fc3665 === kennethix` (salt `7f`)
- `cff39d9be689 === steele99` (salt `2e`)
- `662ab7be194c === born03101991#` (salt `24`)

Closing thoughts

As you saw, this is quite straightforward. But remember that in the wild, you may not know the list of potential salts, and may have to explore a huge list *for each password*. If you were to know that the hashes were always 2 hexa characters, this is 256 times harder than a normal dictionary attack.

Exercise 2: Use Rainbow Tables

Hashing

The first thing to do is define a hashing function. As instructed, we use SHA256. You are probably familiar with the process, by now:

```
import hashlib

def hashing(plaintext: str):
    return hashlib.sha256(plaintext.encode()).hexdigest()
```

Reduction functions

This is a more tricky part. Given a cipher (a hash), we have to find multiple functions that will map (deterministically) to the "password space". There are many good functions out there. This is actually important for a good rainbow table, because the better those functions are, the fewer collisions you will have. As instructed, we build a function that transforms the hash into a base 16 int, then turns that into a "base 36" number, thus allowing us to cover the whole character set.

```python
def reduction(h: str, i: int):
    # change the hash to an int (base 16), and add number of column
    corresp_int = int(h, 16) + i
    chars = string.ascii_lowercase + string.digits
    chars_len = len(chars)
    pwd = ""
    # convert to the new base, char by char, until the size of the password
    # reaches desired length.
    while len(pwd) < SIZE_PASSWORDS:
        pwd = pwd + chars[corresp_int % chars_len]
        corresp_int = corresp_int // chars_len
    return pwd
```

It is important to note that, while only a single `reduction` function is defined, it actually serves the purpose of multiple different functions in practice (one per column), as the column number is added to the int derived from the hash.

First string of rows

Once you have the first element of the row, the rest is computed deterministically (given the hashing and reduction functions). But this first element ought to be unique. Fortunately, we have a unique element assigned to each row: its index. We pass it (as a hexadecimal number) to the reduction function, to obtain our first and unique element.

```python
init_plaintext = reduction(hex(row_num), 0)
```

The strings generated that way will be `baaaaaaa`, `caaaaaaa`,... For additional randomness, we could replace the 0 with a random integer and, depending on the size of the table, check for collisions.

Computing the table

This is the initial step of any attack. If your table is well done, it's worth spending a lot of time creating it, as it will make attacks more likely to succeed. The general process is as follows (pseudo-code):

```
for n in NUMBER_ROWS:
    s = random_string
    for reduc in ordered_reduction_functions:
        h = hash(s)
        s = reduc(h)
```

Then, store your initial and last strings (plaintext) in a lookup table.

```python
def build():
    end_to_start_matching = dict()
    for row_num in range(NUM_ROWS):
        init_plaintext = reduction(hex(row_num), 0)
        plaintext = init_plaintext
        for i in range(NUM_REDUCTIONS):
            h = hashing(plaintext)
            plaintext = reduction(h, i)
        end_to_start_matching[plaintext] = init_plaintext
```

At each iteration, you compute NUM_REDUCTIONS passwords but only keep 2: the initial string, and the final one. And because each reduction function is different, the risk of collision is minimal.

Serializing

Once you have computed your table, simply store it as a pickle file.

```
import pickle as pkl

with open(filename, "wb") as f: #mode "wb" is important
    pkl.dump(end_to_start_matching, f)
```

On the other hand, deserializing is very similar:

```
with open(filename, "rb") as f:
    end_to_start_matching = pkl.load(f)
```

Second phase: attack

The attack is done in two steps: first, compute hashing and reductions, as explained in the handout, until you have a match between the computed hash and one of the “final hashes” computed earlier; If you have a match, recompute the whole row until you find your hash.

```
def crack_hash(target_hash):
    end_to_start_matching = #load pickle
    for i in range(NUM_REDUCTIONS-1, -1, -1):
        # try from all starting columns, starting from the end
        # (so i = N-1, N-2, N-3,...,0)
        h = target_hash
        for j in range(i, NUM_REDUCTIONS):
            # range j between i, i+1, i+2,...,N
            plaintext = reduction(h, j)
            h = hashing(plaintext)
        if plaintext in end_to_start_matching:
            # you found a match: go to phase 2, recompute the whole row
            # may be a false alarm.
            preimage = find_preimage(target_hash,
                                    end_to_start_matching[plaintext])
            if preimage is not None:
                # if not a false alarm, return it.
                return target_hash, preimage

def find_preimage(target_hash: str, start_text: str):
    # phase 2: recompute the whole row until you find your target hash
    # (or the end of the row, false alarm)
    plaintext = start_text
    for i in range(NUM_REDUCTIONS):
        h = hashing(plaintext)
        if h == target_hash:
            return plaintext
        plaintext = reduction(h, i)
    #if hash not found in chain, return
    return None
```

Simply wrap that around a for loop to crack a list of hashes.

Parallelization

Building a table is usually rather costly. Attacking a hash is rather fast (though depends on the shape of the table), but attacking multiple times becomes long. Fortunately, each row of the table can compute independently, and each hash can be attacked independently. For these reasons, we rely on parallelism to accelerate the process. We won't detail the code here, as it goes beyond the scope of the exercise, but you are encouraged to read and understand the alternate version of the sample solution, which uses Python's multiprocessing library to handle multiple threads and speed up the process. The idea is always the same: create a function that takes part of 1 "unit" (one row or one hash), then make use of the library to handle its execution through different threads. Using a pool of workers and `imap_unordered` (as we do not usually care for the order of rows or hashes) makes the implementation a lot faster.

You can find a different version of the solution, parallelized, in the attached `ex2_parallel.py`.

Matching passwords

You can find them in `dump_matches.txt`.

Theoretical questions

1. Q: *How do you compute the success rate of your table?*
   A: Either empirically or theoretically. Empirically, look at the fraction of hashes you managed to reverse. Theoretically, consider the size of the password space (36^8 in our case), and the *maximum* number of passwords covered (#rows x #columns). Though, this supposes no or little colliding branches, which is not always the case. Bigger tables lead to more overlaps, and a higher chance of duplicates. This also supposes everything is uniformly distributed, which usually isn't. A finer success rate can be found in the appendix of the original paper
2. Q: *What are the advantages of making the table "fatter" (more columns)? And "taller" (more rows)?*
   A: The advantage of a "fat" table is that it takes less space. Because you "discard" the values in between, you'll have to store less. But in turn, looking up a hash becomes costly. At the extreme, you only have one row, arbitrarily long. You then only store 2 plaintexts, but now have to recompute everything at each lookup. On the other hand, a "tall" table makes lookup faster, but you have to store more. Looking again at the extreme, you'd have a table where you only apply 1 hashing and 1 reduction. Lookup becomes extremely fast (you only have to do one reduction and look it up), but then you've basically built a (bad) lookup table. Ultimately, it is necessary to carefully consider the trade-off between memory and speed, also based on the target and the available resources.
3. Q: *How do height and width relate to the size (on disk) of the table, and the lookup time?*
   A: The lookup time is *t(t-1)/2* for a table with *t* columns. This is because you first try to compute one reduction and one hash, then two reductions and two hashes,

then three reductions and three hashes,... thus the result. The size on disk is directly proportional to *m* (the number of rows): You will keep *2m* elements from the password space.

4. Q: *The state-of-the-art cracking uses multiple tables. What is the point?*
   A: Even though rainbow tables are more efficient than Hellman's tables, with a drastically reduced number of collisions, it may still happen. Suppose you have a table of size *m x t*, that you want to expand by a factor *l*. Making a table of size *m x lt*, your lookup will skyrocket roughly $(lt)^2$, compared to lt^2 for *l* distinct tables. If we made it taller (*lm x t*), the risk of collision would increase.
5. Q: *What would be the (estimated) size of a table to have a >95% success rate on this password space?*
   A: Following the fine-grained analysis of the original paper, and after some numerical computations, we find that a table of size `60B x 90K` would have a rate of ~95.5%. We included a utility in the solution Python file, to compute the success rate according to some parameters.
6. Q: *Can you parallelize the processes of creating the table and hacking?*
   A: Yes, parallelization is indeed possible, and actually crucial because of the usually high computational cost of the process. As explained above, each row can be computed independently, so does each hash. If longs were to be particularly long, we could even parallelize the cracking *within* one hash.
7. Q: *How would you modify the reduction functions to match exactly the Zürich Insurance requirements? How would that affect your success rate?*
   A: The first character would be taken from a different space (a-zA-Z), and all others would have the same (larger) charset. This would create a password space of $52 \times 62^7 \approx 1.83 \times 10^{14}$. If we used the same table size as above (`60B x 90K`), the success rate would drop to 0.27%.

Exercise 3: PAKE Implementation

This exercise is a bit more complex than the previous ones. You really only have one procedure to implement, but the juggling with types, and the wrestling with websockets make it more challenging. Let's start from the beginning.

The structure

As we are working with asynchronous websockets, we can't just put everything as a script and run it. We must tell python that we expect asynchronous outputs. For that, we will use the following skeleton:

```
import websockets
import asyncio
import ...

# CONSTANTS, as provided
EMAIL = "your.email@epfl.ch"
PASSWORD = "correct horse battery staple"
N = int("EEAF0AB9ADB38DD69C33F80AFA8FC5E86072618775FF3C0B9EA2314C9C256576D674DF7496EA81D3383B
g = 2
NUMBER_LENGTH = 32

def some_standard_function():
  #normal functions
```

```
async def websocket_function(websocket, param):
  # asynchronous functions that interact with the websocket

async def srp():
  # the core of the procedure


asyncio.get_event_loop().run_until_complete(srp())
```

Breaking down the process

As you can guess, we can break down our core procedure in a bunch of *send* and *receive*. Everything will be done within one opening of a websocket. We can then start our procedure with the following:

```
uri = "ws://127.0.0.1:5000"
async with websockets.connect(uri) as websocket:
    # then work here
```

The first step of the protocol is to send the email. This one isn't too hard:

```
await websocket.send(EMAIL)
```

Then, you need to receive the salt. It arrives in hexadecimal form, and you want to convert it,

```
salt_hex = await websocket.recv()
salt_int = int(salt_hex, 16)
salt_bin = format(salt_int, "x").encode()
# alternatively: salt_bin = salt_hex.encode()
```

Next you want to generate a random `a`, then compute `A` using `g, a, N`.

```
import os
a = int.from_bytes(os.urandom(NUMBER_LENGTH), "big")
A = pow(g, a, N)
A_hex = format(A, "x").encode()
await websocket.send(A_hex)
```

Note that the os library can create random bytes, encoded in big-endian, and we need to convert it tho int, thus the `int.from_bytes(..., "big")`

Then, we await the `B`, quite straightforward:

```
B_hex = await websocket.recv()
B = int(B_hex, 16)
```

We get to the hashing library. This one requires to use, as mentioned, binary-encoded hexadecimal representations of the numbers. Note that we could use any binary-encoded representation (bytes, int, hex,...); it doesn't matter as long as we are consistent with ourselves and the server.

We create an empty hash, hash A and B, and retrieve the result (*digest*):

```
import hashlib
h = hashlib.sha256()
h.update(format(A, "x").encode)
h.update(format(B, "x").encode)

u_hex = h.hexdigest()
u = int(u_hex, 16)
```

Now, we have a nested hash. We could compute it in one go, but it's simpler to consider this as an *inner* hash, and an *outer* hash. We first compute the inner.

```
# H(U || ":" || PASSWORD)
inner = hashlib.sha256()
inner.update(EMAIL.encode())
inner.update(b":")
inner.update(PASSWORD.encode())
```

And the outer hash, that requires the inner digest; you then get `x` as the digest of the outer hash:

```
outer = hashlib.sha256()
outer.update(salt_bin)
outer.update(inner.hexdigest().encode())

x = int(outer.hexdigest(), 16)
```

Some math now: we now have `g,x,N,a,u`, we can then compute the secret!

```
#S = (B - g^x)^(a + u * x) % N
S = pow(B - pow(g, x, N), (a + u * x), N)
```

Note that we can't compute `g^x` directly, at the risk of an overflow. We can simply modulo it to `N`, as the result is identical (hence the `pow(g,x,N)`). And we now have the shared secret!

Last step (necessary for verification purpose, but optional for a real-life scenario) is to verify we have the same secret as the server. For that, we have to compute `H(A || B || S)`, and send it. This is straightforward, and very similar as what we did before:

```
# create the hash
h = hashlib.sha256()
h.update(itob(A))
h.update(itob(B))
```

```
h.update(itob(S))

# send the hex digest
await websocket.send(h.hexdigest().encode())

# receive confirmation (or not) that the secret is the same.
resp = await websocket.recv()
print("Response: {}".format(resp))
```