Password Cracking Research and Technical Overview

Securing Passwords

Hashing vs Encryption

Hashing is a one-way function whereas encryption is a two-way function; as a result of encryption allowing for the original plaintext to be found, the storage becomes less secure and should "only be used in edge cases where it is necessary to obtain the original plaintext password" according to OWASP. Instead, hashing a password is preferable as it makes it more secure; it is impossible to decrypt a hashed password but it is possible to crack it instead by using brute force - or educated brute force - methods.

Improving the Security of the Hashing Process

Salting

Since the hashing process is deterministic, which means that for every given input, there is a particular and predictable output, then duplicate passwords would generate the same hashed output; if a list of hashed passwords is exfiltrated from a database, cracking one hashed string could expose the passwords for multiple individuals.

A salt is a distinct (unique for each password), randomly generated string that is added to a password before it is hashed. This salt is stored in plaintext alongside the hashed value, such that when a user logs in for example, the system retrieves the salt first, adds it and then hashes it to check against the stored hash. Note that the salt does not need to be cryptographically secure since it is stored in plaintext and is not kept a secret.

A salt, therefore, has two key benefits:

- Two users with the same password would have different hashes since they have different salts; cracking one password does not immediately reveal the other.
- Salting greatly defends against rainbow table attacks; since rainbow tables precompute the hashes, you would need a rainbow table for each salt value, thereby making the storage cost very expensive and unrealistic.

Peppering

A pepper is a secret string that, following best practice, is stored in Hardware Security Modules (HSMs), which are hardware dedicated to protect secrets such as crypto keys. Peppers act as a second salt, and rather than being unique like a salt, is instead shared between all passwords.

From a rainbow table perspective, where a salt requires a table unique to each user or password, a pepper requires a table unique to each database. And, unlike the salt, the pepper is kept a secret, making a rainbow table attack even less feasible.

Work Factors

The work factor is a measure of the computational price required by an attacker to crack a password. To be more specific, according to OWASP, the factor measures the number of times that the password is rehashed, where there are 2^{work} iterations.

This work factor is usually stored in the database as plaintext and helps to slow down the hashing process required by a password cracker. Whilst this slows down bad actors in the case of an exfiltration, it also reduces the program of the actual program itself and can be exploited by bad actors in a DDoS attack.

A commonly used approach is to change the work factor at each user authentication to match the current system capabilities. The hash is updated with the new work factor and allows passwords to be stored more securely as the available hardware improves over time.

Cryptographically Secure Pseudo-Random Number Generators (CSPRNG)

A pseudo-random number generator (PRNG) is a deterministic algorithm that generates 'random' numbers initialised using a seed state, also known as a seed. The 'pseudo' nature of the RNG is due to the fact that is in not possible to generate truly random numbers using a computer, which is deterministic hardware.

While PRNGs are, to the naked eye, random for the most part, certain design decisions could result in the produced numbers more predictable. In contrast, CSPRNGs have three distinguishing characteristics (in addition to being more computationally expensive):

- High entropy (randomness or disorder)
- Uniqueness of generated strings
- Zero correlation

Despite salts not requiring cryptographic security, given the uniqueness of strings generated, CSPRNGs are often an easy solution for distinct salts. Below are some examples of CSPRNGs for some languages:

Language	Unsafe functions	CSPRNGs
C	random(), rand()	getrandom(2)
Python	random()	secrets()
Java	Math.random(), StrictMath.random(), java.util.Random, java.util.concurrent.ThreadLocalRandom, java.util.SplittableRandom	java.security.SecureRandom, java.util.UUID.randomUUID
Node.js	Math.random()	crypto.randomBytes(), crypto.randomInt(), crypto.randomUUID()

What is Password Cracking?

Password cracking is the process of retrieving passwords from data transmitted or stored inside digital systems. However, since hash functions are considered irreversible, password cracking relies upon guesses to find the original password (or an equivalent that produces an identical hash). However, simply brute forcing a password is not optimal for most passwords, which is why modern programs take advantage of patterns in human psychology to crack passwords incredibly quickly. Some examples of these methods include dictionary and combinator attacks. Furthermore, whilst often left as a last resort for particularly resilient passwords, brute forcing has become far more feasible due to the high hash rates of modern top-end GPUs.

Password Cracking Algorithms

Historical Method - Rainbow Tables

Historically, the cost of CPU time was greater than that of storage, which led to attackers pre-computing hashes and storing them in a relatively storage-efficient structure. In contrast to a naïve hash table, a rainbow table is produced by first hashing a string, reducing the resulting hash, rehashing the reduced hash and then sending it through another reduction function that is distinct to the first. This process is repeated over and over to produce a very long chain of a starting string and a final hash.

Once a rainbow table is produced, it can be used to quickly look up a plaintext string that produces a particular hash; the hash is first checked against all the hashes at the end of each chain. If none match, the hash is reduced and hashed again; if it matches with one of the ends of a chain, it implies that the given hash was the second last hash in the chain. This process is repeated over and over until the chain containing the hash is found. Finally, once the chain is determined, the program hashes and reduces along the chain until the string immediately prior to the given hash is found - that string would be the password.

Whilst this was very effective historically, the ever decreasing cost of computing hashes proportional to storage cost and the introduction of salts and peppers have made brute-force (and educated brute-force) attacks far more preferable.

Dictionary Attack

A very simple attack which hashes and compares all passwords from a given wordlist. Often, these wordlists are chosen based on long lists of common passwords, such as the rockyou.txt wordlist.

Mask Attack

Whilst a dictionary attack brute forces all passwords from a given wordlist, a mask attack hashes and tests all passwords from a given keyspace whilst applying certain 'masks' over the keyspace. These 'masks' are chosen by considering the common processes by which individuals design their passwords, such as how it is common for the first character of a password to be capitalised but rare for the second or third characters to be.

Combinator Attack

Creates a new dictionary/wordlist by appending words from one dictionary to another. A hybrid attack can also be produced by using wordlists produced by different attack methods, such as a dictionary attack wordlist on one side, and a mask attack wordlist on another.

Technical Component - A Password Cracker

Initial Goal

To create a password cracker (like HashCat) using some of the researched methods above.

Writing an API for Hashing Strings

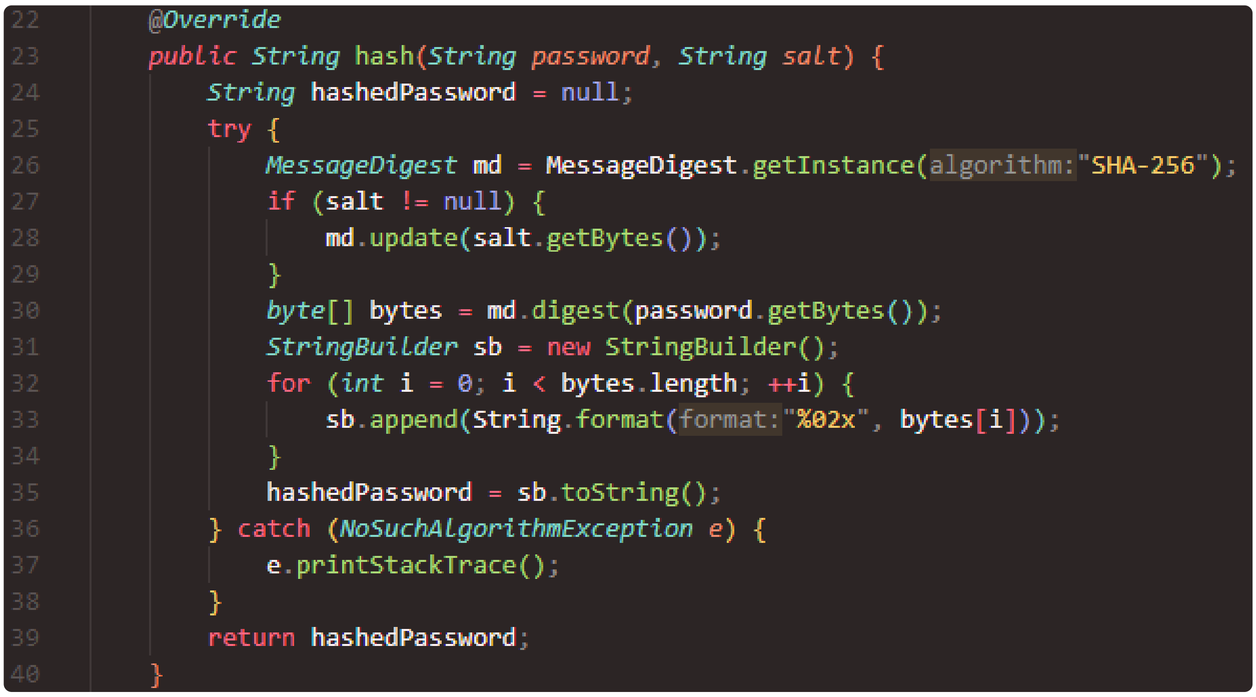
I decided to create the implementation using Java as this was the programming language that I had last used. With Java in mind, I did a little research into what built-in classes could be used to do some simple hashing of passwords and have found the

`MessageDigest` class that was required to have SHA-256 support for all versions of Java.

With this in mind, I created the `Hasher` interface with five methods in mind:

- `randomHashN()` $\rightarrow$ generates and hashes a random password of size n and salt, returning the password, the salt and the hash as a `Triplet`. This will mainly be used for testing the performance of cracking passwords of varying lengths.
- `randomHash()` $\rightarrow$ generates and hashes a random password and salt, returning the password, the salt and the hash as a `Triplet`.
- `hashRandomSalt(String)` $\rightarrow$ hashes a given password with a random salt, returning the salt and the hash as a `Pair`.
- `hash(String, String)` $\rightarrow$ hashes a given password with a given salt. This will be used within the password cracking algorithms.
- `hash(String)` $\rightarrow$ hashes a given password with no salt.

For simplicity's sake, I decided to only implement this interface with the SHA-256 hashing algorithm, thereby allowing me to create the SHA256Hash class. With this tutorial and the Java documentation helping to contextualise the roles of each method within the `MessageDigest` class, I was able to implement the methods listed above.



```java
    @Override
    public String hash(String password, String salt) {
        String hashedPassword = null;
        try {
            MessageDigest md = MessageDigest.getInstance(algorithm:"SHA-256");
            if (salt != null) {
                md.update(salt.getBytes());
            }
            byte[] bytes = md.digest(password.getBytes());
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < bytes.length; ++i) {
                sb.append(String.format(format:"%02x", bytes[i]));
            }
            hashedPassword = sb.toString();
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        return hashedPassword;
    }
```

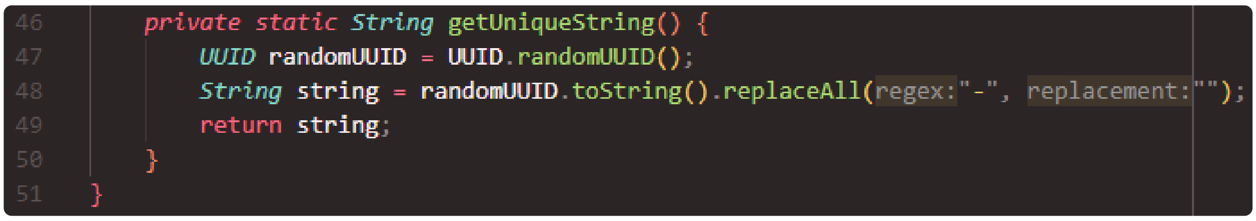
According to Oracle's Java documentation, message digest algorithms "are secure one-way hash functions that take arbitrary-sized data and output a fixed-length hash value". Here, I chose the SHA-256 hashing algorithm using the `getInstance()` method within the `MessageDigest` class.

The `update()` method allows the `MessageDigest` to be updated with some given bytes. Here, a salt may be optionally used and the `String` is converted into an array of bytes. The `digest()` method then allows for a given password to (once it has been converted into an array of bytes) be hashed by the chosen algorithm (SHA-256).

The resulting bytes after the hashing has been carried out are then constructed into a hexadecimal string using the `StringBuilder` class. Notably, on line 33, my method deviates significantly from the suggested classes and methods recommended by the tutorial; a simpler section of code is used to the same effect. Instead of

```
sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1))
```

my code uses the `format()` method from the `String` class to append each byte in hexadecimal with at least two characters. Whilst the `toString()` method has been shown to be more performant than the `format()` method, given the demonstrating nature of this project, it is more important to have easier-to-understand code rather than super-performant code.



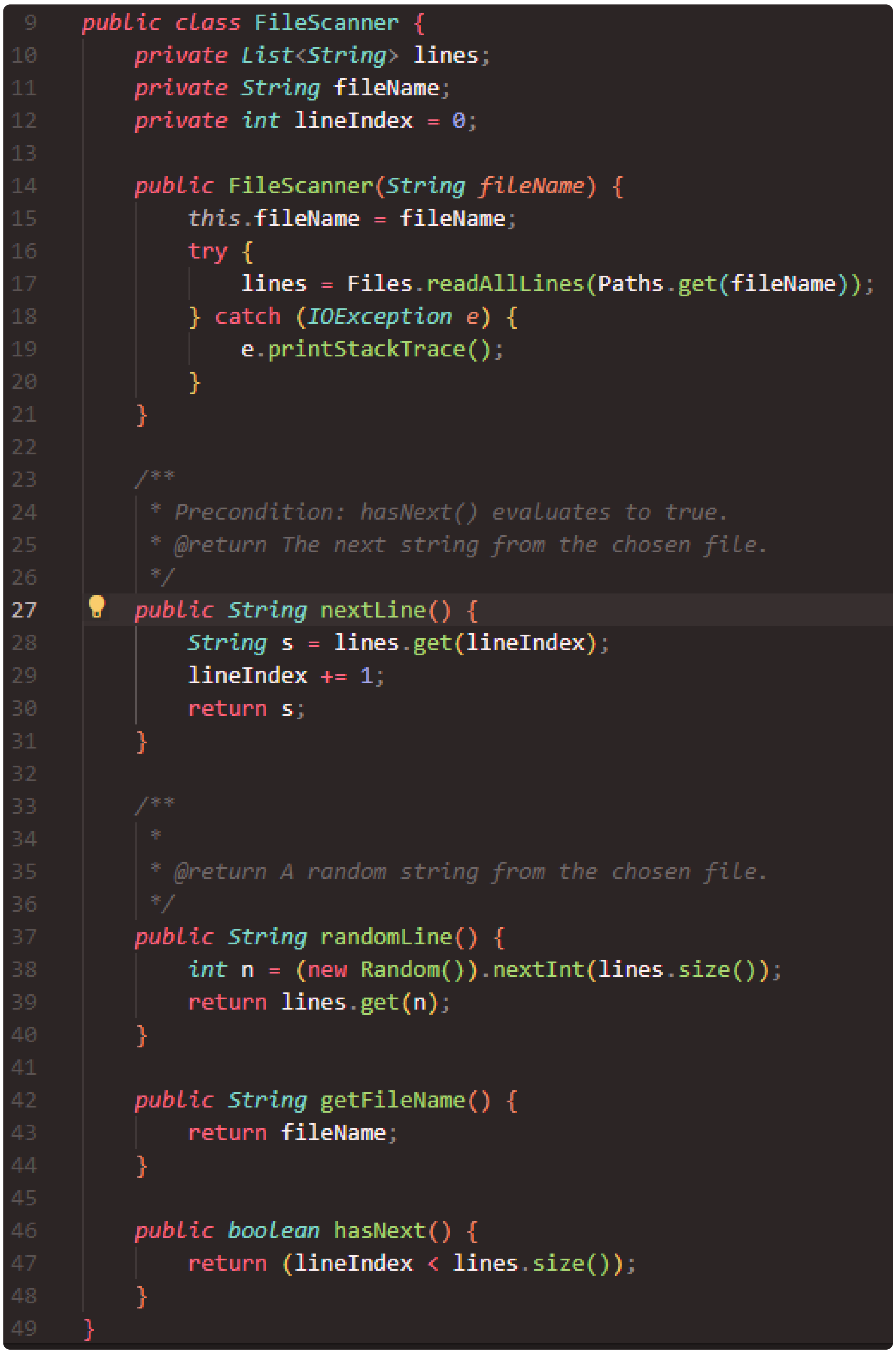
```
    private static String getUniqueString() {
        UUID randomUUID = UUID.randomUUID();
        String string = randomUUID.toString().replaceAll(regex:"-", replacement:"");
        return string;
    }
}
```

The generation of a random password or salt in the above methods is handed by the private method `getUniqueString`. This method calls the `UUID.randomUUID()` method that, as mentioned earlier and described by the JavaDoc, "the UUID is generated using a cryptographically strong pseudo random number generator".

Since the only requirement for a salt is that it is unique, which is easily satisfied by a CSPRNG, then this method is suitable for both password and salt generation.

Writing the Password Cracking Algorithms

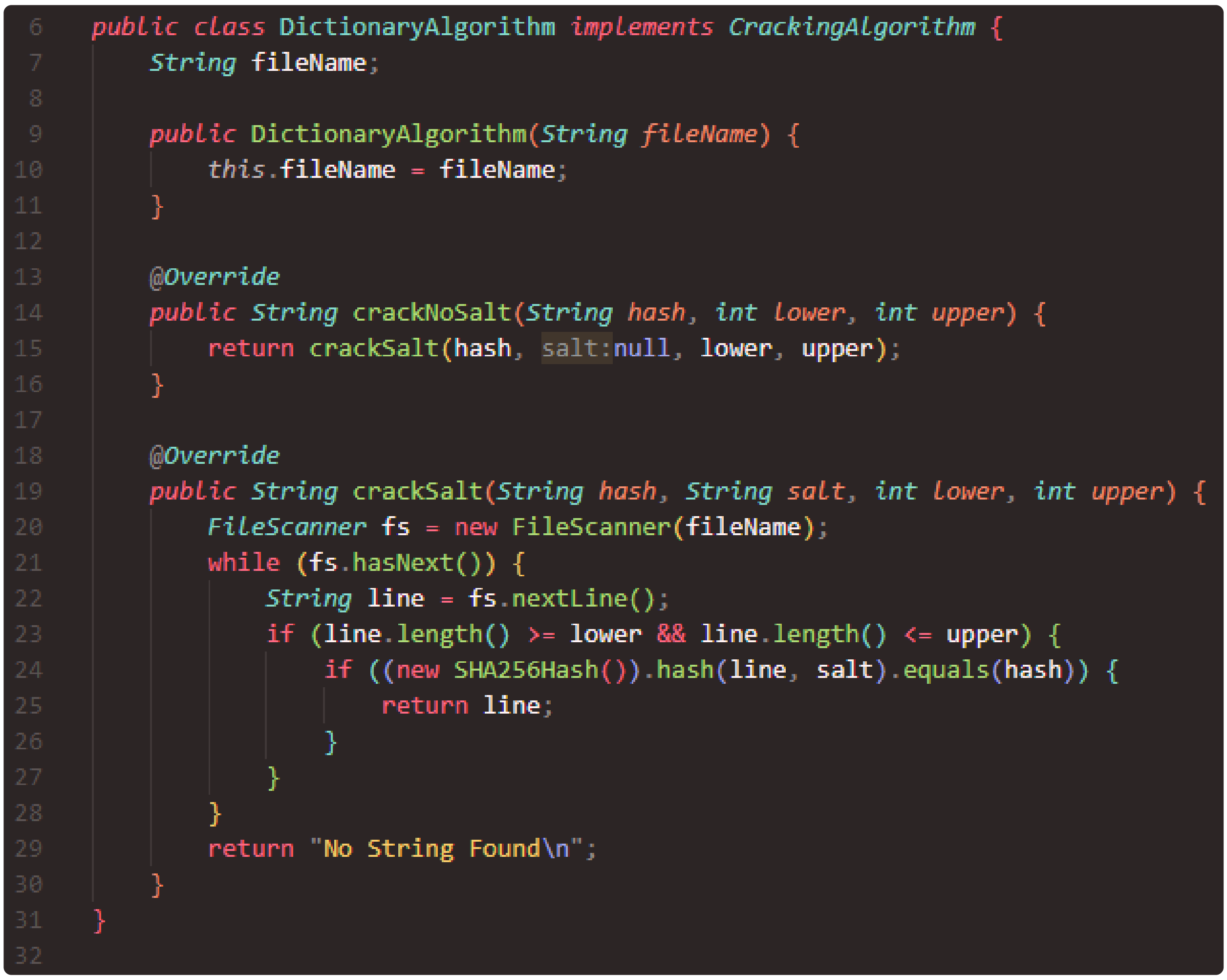
I began by writing the code for the brute form algorithm as it seemed to be the easiest to implement. The code for the class can be seen below.



```java
public class FileScanner {
    private List<String> lines;
    private String fileName;
    private int lineIndex = 0;

    public FileScanner(String fileName) {
        this.fileName = fileName;
        try {
            lines = Files.readAllLines(Paths.get(fileName));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Precondition: hasNext() evaluates to true.
     * @return The next string from the chosen file.
     */
    public String nextLine() {
        String s = lines.get(lineIndex);
        lineIndex += 1;
        return s;
    }

    /**
     *
     * @return A random string from the chosen file.
     */
    public String randomLine() {
        int n = (new Random()).nextInt(lines.size());
        return lines.get(n);
    }

    public String getFileName() {
        return fileName;
    }

    public boolean hasNext() {
        return (lineIndex < lines.size());
    }
}
```

As can be seen above, the final version of the class used a more modern implementation with the `Files` and `Paths` classes. Initially, I attempted to use the older `FileReader` class that was used in some exercises of a previous Java course, however, I made little progress and the code was bug-ridden. A substantial amount of time later spent trying to debug the program and browsing the forums, I found the classes used above.

Using the `FileScanner` class, I then implemented the dictionary attack algorithm below.



```java
public class DictionaryAlgorithm implements CrackingAlgorithm {
    String fileName;

    public DictionaryAlgorithm(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public String crackNoSalt(String hash, int lower, int upper) {
        return crackSalt(hash, salt:null, lower, upper);
    }

    @Override
    public String crackSalt(String hash, String salt, int lower, int upper) {
        FileScanner fs = new FileScanner(fileName);
        while (fs.hasNext()) {
            String line = fs.nextLine();
            if (line.length() >= lower && line.length() <= upper) {
                if ((new SHA256Hash()).hash(line, salt).equals(hash)) {
                    return line;
                }
            }
        }
        return "No String Found\n";
    }
}
```

Change of Plans

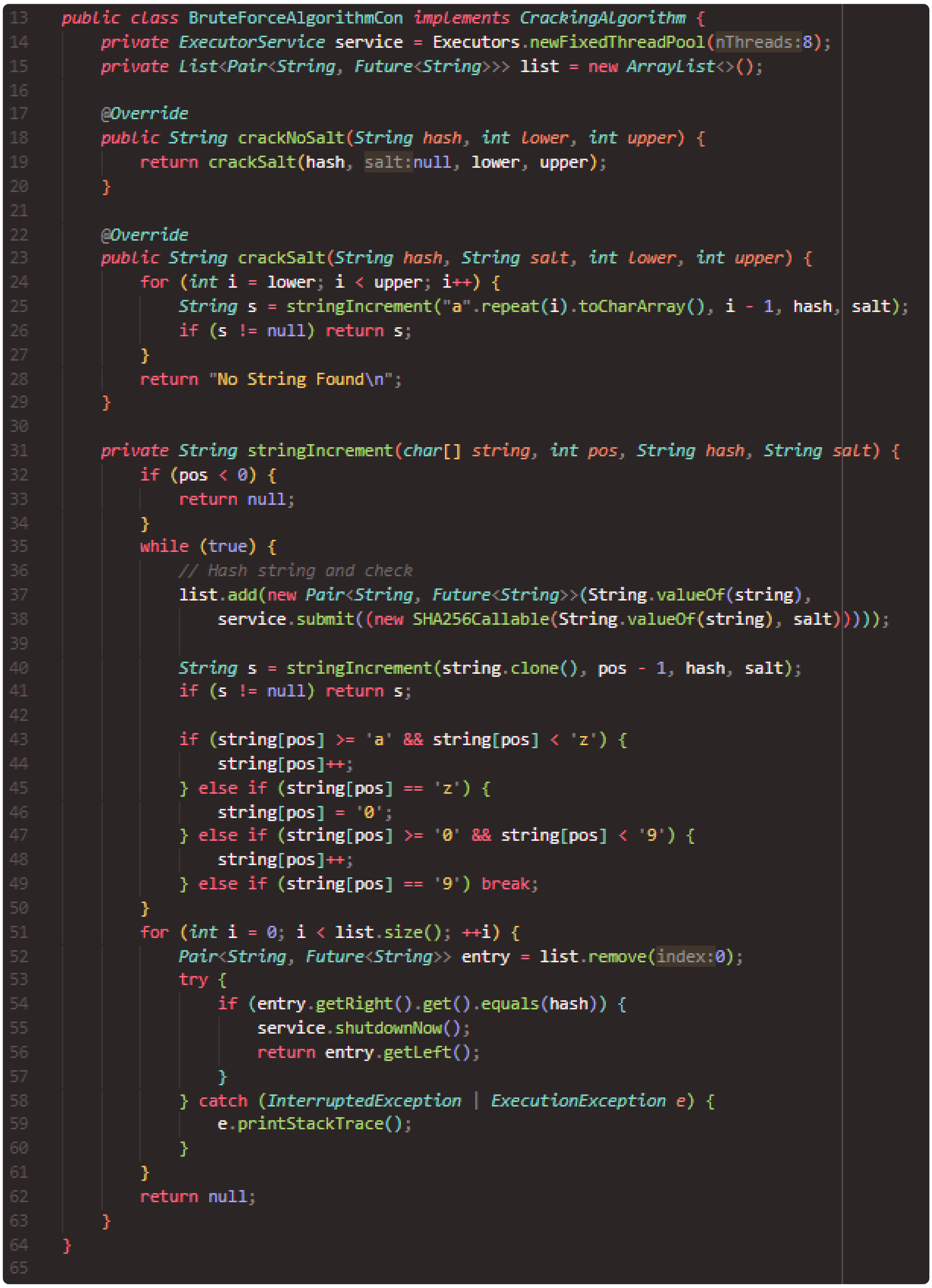
Whilst I initially intended to implement more of the researched algorithms, I ran into a glaring issue - the performance of the code was far too lacking compared to what I had read about! With this in mind, rather than developing more rudimentary and non-performant code, I decided to shift my focus to improving the speed at which the code may execute. In particular, I decided to optimise the brute force method as it is the only method that can crack any password - whilst many individuals may use simpler passwords that involve common words, more security-inclined individuals may use randomly generated strings instead which cannot be cracked by algorithms such as a dictionary attack or a mask attack.

Figuring Out Concurrency

When considering how to improve the speed of my brute-force attack, I looked into the benefits of multi-threading and concurrent programming. Notably, I attempted to use some code similar to content learnt in previous Java courses that suggested the use of the `Thread` class and `Runnable` objects. However, I ran into issues to do with the retrieving of an object's value once the thread has finished executing, given the fact that the `run()` method does not return anything.

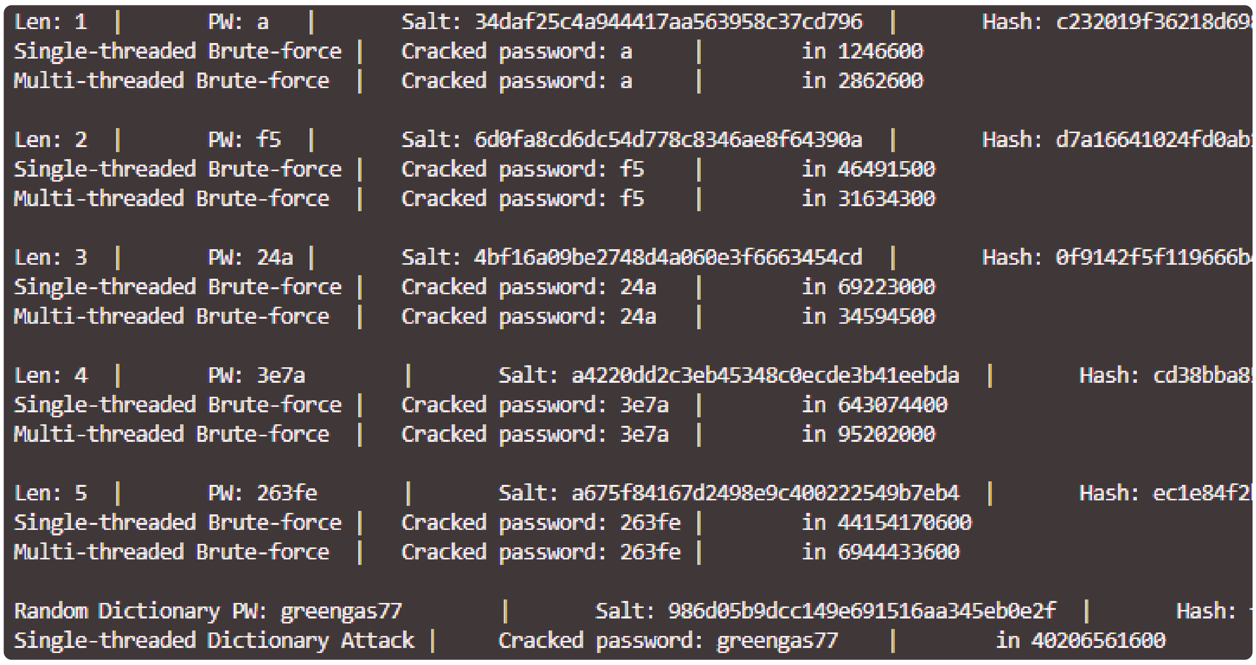
After some substantial research, I discovered the `ExecutorService` and `Callable` classes that allowed for the use of the `Future<T>` class. To summarise briefly, these classes behave in much the same way as ordinary threads, however, `Callable` objects contain a `call()` method that returns a `Future<T>` object in contrast to the `void run()` method.

This resulted in the following code in a new class `BruteForceAlgorithmCon`; having both the concurrent and non-concurrent algorithms would allow us to compare the differences in performance.



```java
public class BruteForceAlgorithmCon implements CrackingAlgorithm {
    private ExecutorService service = Executors.newFixedThreadPool(nThreads:8);
    private List<Pair<String, Future<String>>> list = new ArrayList<>();

    @Override
    public String crackNoSalt(String hash, int lower, int upper) {
        return crackSalt(hash, salt:null, lower, upper);
    }

    @Override
    public String crackSalt(String hash, String salt, int lower, int upper) {
        for (int i = lower; i < upper; i++) {
            String s = stringIncrement("a".repeat(i).toCharArray(), i - 1, hash, salt);
            if (s != null) return s;
        }
        return "No String Found\n";
    }

    private String stringIncrement(char[] string, int pos, String hash, String salt) {
        if (pos < 0) {
            return null;
        }
        while (true) {
            // Hash string and check
            list.add(new Pair<String, Future<String>>(String.valueOf(string),
                service.submit((new SHA256Callable(String.valueOf(string), salt)))));

            String s = stringIncrement(string.clone(), pos - 1, hash, salt);
            if (s != null) return s;

            if (string[pos] >= 'a' && string[pos] < 'z') {
                string[pos]++;
            } else if (string[pos] == 'z') {
                string[pos] = '0';
            } else if (string[pos] >= '0' && string[pos] < '9') {
                string[pos]++;
            } else if (string[pos] == '9') break;
        }
        for (int i = 0; i < list.size(); ++i) {
            Pair<String, Future<String>> entry = list.remove(index:0);
            try {
                if (entry.getRight().get().equals(hash)) {
                    service.shutdownNow();
                    return entry.getLeft();
                }
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
}
```

Testing the Performance of Concurrent Code

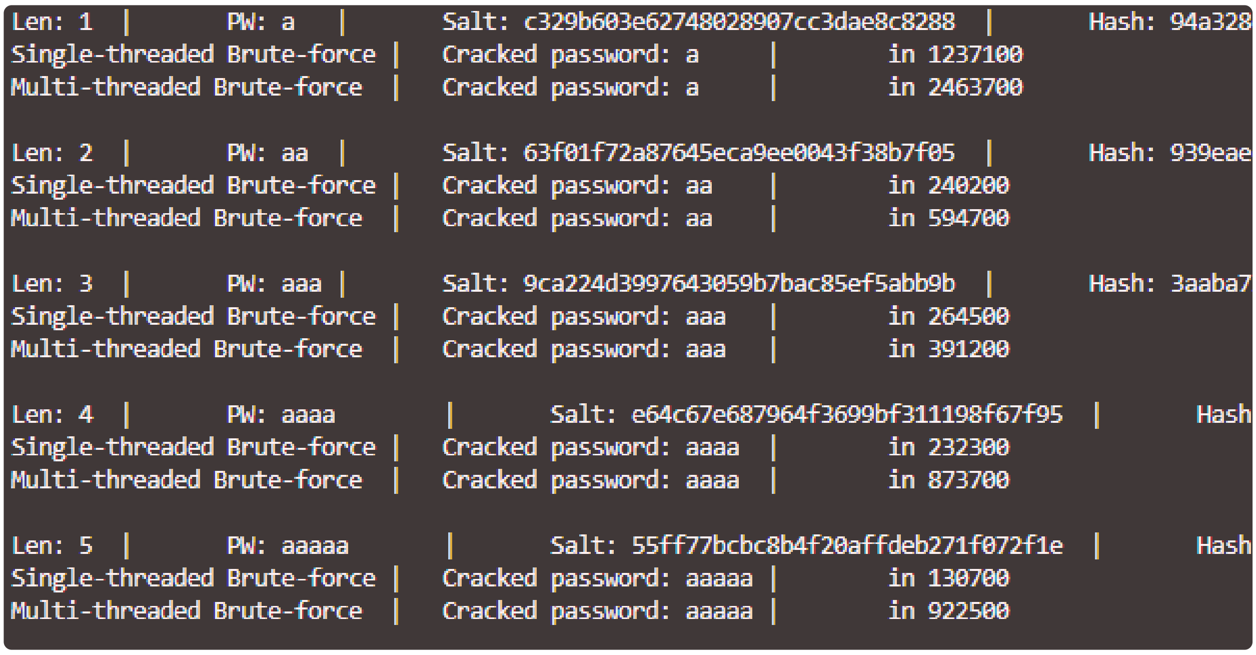


```
Len: 1  |       PW: a   |       Salt: 34daf25c4a944417aa563958c37cd796  |       Hash: c232019f36218d69
Single-threaded Brute-force |   Cracked password: a     |       in 1246600
Multi-threaded Brute-force  |   Cracked password: a     |       in 2862600

Len: 2  |       PW: f5  |       Salt: 6d0fa8cd6dc54d778c8346ae8f64390a  |       Hash: d7a16641024fd0ab
Single-threaded Brute-force |   Cracked password: f5    |       in 46491500
Multi-threaded Brute-force  |   Cracked password: f5    |       in 31634300

Len: 3  |       PW: 24a |       Salt: 4bf16a09be2748d4a060e3f6663454cd  |       Hash: 0f9142f5f119666b
Single-threaded Brute-force |   Cracked password: 24a   |       in 69223000
Multi-threaded Brute-force  |   Cracked password: 24a   |       in 34594500

Len: 4  |       PW: 3e7a        |       Salt: a4220dd2c3eb45348c0ecde3b41eebda  |       Hash: cd38bba8
Single-threaded Brute-force |   Cracked password: 3e7a  |       in 643074400
Multi-threaded Brute-force  |   Cracked password: 3e7a  |       in 95202000

Len: 5  |       PW: 263fe       |       Salt: a675f84167d2498e9c400222549b7eb4  |       Hash: ec1e84f2
Single-threaded Brute-force |   Cracked password: 263fe |       in 44154170600
Multi-threaded Brute-force  |   Cracked password: 263fe |       in 6944433600

Random Dictionary PW: greengas77        |       Salt: 986d05b9dcc149e691516aa345eb0e2f  |       Hash:
Single-threaded Dictionary Attack |     Cracked password: greengas77    |       in 40206561600
```

For each section of the above image, the first line lists the password and salt randomly generated for the test as well as their resulting hash. The following two lines then list the cracked password (used to confirm the algorithm has correctly identified it), but more importantly, each line also lists the time required to carry out each algorithm in nanoseconds.

For randomly generated passwords of length 2 to 5, it can be shown that the multi-threaded performance is better than the single-threaded performance. Notably, the advantage of multi-threading trends upwards for larger passwords. Interestingly however, the single-threaded performance is more performant for passwords of length 1, at least for this particular run. This may be attributed to not only the size of the password, but more importantly, to the actual password's value itself.

The password randomly selected for the length 1 test was a single character `a`, which, when observing the algorithm, is in fact the first string tested. Given the slight overhead required to begin each thread/`Callable`, it makes sense for the single-threaded performance to be better for very specific passwords (that is, if the password is the very first one checked). To confirm this hypothesis, I then entered passwords composed of a certain number of repeated `a`'s into the brute-force algorithm as well as selected the lower bound to be the exact length of the password.



```
Len: 1  |       PW: a   |       Salt: c329b603e62748028907cc3dae8c8288  |       Hash: 94a328
Single-threaded Brute-force |   Cracked password: a     |        in 1237100
Multi-threaded Brute-force  |   Cracked password: a     |        in 2463700

Len: 2  |       PW: aa  |       Salt: 63f01f72a87645eca9ee0043f38b7f05  |       Hash: 939eae
Single-threaded Brute-force |   Cracked password: aa    |        in 240200
Multi-threaded Brute-force  |   Cracked password: aa    |        in 594700

Len: 3  |       PW: aaa |       Salt: 9ca224d3997643059b7bac85ef5abb9b  |       Hash: 3aaba7
Single-threaded Brute-force |   Cracked password: aaa   |        in 264500
Multi-threaded Brute-force  |   Cracked password: aaa   |        in 391200

Len: 4  |       PW: aaaa        |       Salt: e64c67e687964f3699bf311198f67f95  |       Hash
Single-threaded Brute-force |   Cracked password: aaaa  |        in 232300
Multi-threaded Brute-force  |   Cracked password: aaaa  |        in 873700

Len: 5  |       PW: aaaaa       |       Salt: 55ff77bcbc8b4f20affdeb271f072f1e  |       Hash
Single-threaded Brute-force |   Cracked password: aaaaa |        in 130700
Multi-threaded Brute-force  |   Cracked password: aaaaa |        in 922500
```

As shown in the results above, my hypothesis was validated with the single-threaded algorithm consistently outperforming the multi-threaded algorithm by a factor of 2 to 5.

Reflecting on the Project

Looking back on the work that I had done, I found that I had overestimated my limited and messy programming skills, which resulted in a bloated scope that I could not complete. Notably, I had to restrict my scope to simply writing a somewhat performant brute force algorithm and a dictionary algorithm; I did not implement the other researched algorithms nor did I create a command line interface for the program.

A large part of why I struggled to satisfy my initial scope was an overemphasis on reading broadly on the topics rather than skimming through some relevant articles or papers. This allowed me to gain a better appreciation for the techniques as well as grasp a more holistic image of the password cracking scene. However, much of this information did not actually describe or help in implementing and optimising the technical side of the algorithms.

Furthermore, much of the remaining time was spent on debugging the concurrent code for the brute-force algorithm; with little to no forum posts addressing the specific errors or bugs that I encountered, a couple of hours ended up being spent debugging the code via trial and error. Where some mistakes were simple, others required completely restructuring the code as in the case of transitioning from using Runnable objects to Callable ones instead.

I had also planned to implement GPU processing (to further speed up the program significantly) by spending a couple hours over the suggested 30 hours, however, given the unexpectedly longer debugging process, I was already well over the 30 hour timeframe and I decided that I should begin on writing up my report as well as the other parts of my deliverables.

If I were given the opportunity to start the project over again, I’d place less emphasis on the broad theory behind each algorithm (and in particular, the extensive cross-referencing), and instead focus on learning how to best implement the technical aspects of each algorithm. With slightly better time-management and scheduling, the password cracker would have had much more similar performance compared to programs such as HashCat.

Overall, the project was enjoyable and I still learnt a lot! Not only did I gain a significant appreciation for an important part of cybersecurity, I also ended up becoming a lot more comfortable with concurrent programming, which can be helpful in many future projects.