Security Assessment Report

Doroteya Stoyanova
Student ID: 52095187
Course: CS4028: Security
Institution: University of Aberdeen

October 22, 2023

Contents

Hashing approaches

1 Introduction

Hashing, vital in cybersecurity, turns data into a fixed-length string of bytes, typically a sequence of numbers and letters. Enhanced with salting, it is even more secure. This assignment compares Rust and Python in hashing and dictionary tasks, highlighting the influence of language choice on performance. Python was the language of choice for the initial three tasks, shedding light on its capabilities. Yet, regarding the fourth task, Rust was chosen, enabling a direct performance comparison with Python within a professional context.

2 Findings and Analysis

2.1 Brute Force Hashing

The only guaranteed method to find the original input from a hash is to try every possible input until you get a matching hash. This method is what a brute force attack does. For our attack, we have utilised three libraries to enhance the efficiency of our code: **itertools** leverages iterator-based computation for non-redundant combinations; **hashlib** quickly computes hashes, and **string** offers diverse character sets. The primary task was to decode four hashes, each four characters. We have assumed that the hashes consist of lowercase letters and single-digit integers. The code comprises of two functions:

1. `sha512_hash(password)`: Accepts a password and returns its SHA-512 hash for it. [**?**]

Listing 1: SHA-512 Hashing

```
1 def sha512_hash(password):
2     """Return the SHA-512 hash of the given password."""
3     return hashlib.sha512(password.encode('utf-8')).hexdigest()
```

2. The `brute_force_decode(hashes, max_length)`: Utilizes two nested loops for decoding password hashes. The outer loop navigates through potential password lengths up to `max_length`. Simultaneously, the inner loop crafts combinations of characters for each given length. The parameter `max_length` designates the boundary for the character count of the hashed word. Discovered matches are catalogued in the `found_passwords` dictionary. The function terminates either after full hash decoding or upon full combination exploration. After that, the deciphered passwords and their hashes are printed.

Listing 2: Brute-Force

```
1 def brute_force_decode(hashes, max_length=4):
2     """Brute force algorithm encoding all possible combinations of
    ↪ letters and numbers up to the given length."""
3     characters = string.ascii_lowercase + string.digits
```

```
    # an empty dictionary to store the found passwords
    found_passwords = {}
    # Loop through each possible password length up to the maximum length, plus 1 because range() is exclusive
    for length in range(1, max_length + 1):
        # Loop through all possible combinations of letters and numbers of the given length
        for combination in itertools.product(characters, repeat=length):
            # Join the characters together to form the password
            password = ''.join(combination)
            # Hash the password
            hashed_password = sha512_hash(password)

            # Check if the hashed password is in the list of hashes
            if hashed_password in hashes and hashed_password not in found_passwords:
                # If it is, add it to the dictionary of found passwords
                found_passwords[hashed_password] = password

                # Exit if all hashes have been found
                if len(found_passwords) == len(hashes):
                    return found_passwords

    return found_passwords
```

In a computational assessment, the brute force algorithm demonstrated notable efficiency for four-character inputs, executing in a mere 83.2 milliseconds.

Listing 3: Time for Brute Force

```
CPU times: user 82.2 ms, sys: 2.06 ms, total: 84.2 ms
Wall time: 83.2 ms
```

2.2 Dictionary Hashing

Dictionary attacks are another common technique employed to decipher password hashes. We were equipped with a **'PasswordDictionary.txt'** file in the second task. We sought to extract passwords from the file based on the provided hashes. The main procedure is stored in the function **'find_passwords_from_hashes'**. The process takes in a dictionary file and a list of hashes. It reads the file line by line, turning each password into a SHA512 hash. [2] These hashes are stored in a 'hash_dict,' where the hash is the key, and the password is its value. For each input hash, the function checks if it's in the dictionary. If there is a match, it adds the password to a list; if not, it adds a 'None' placeholder. In the end, it returns this list of passwords or placeholders.

Listing 4: Dictionary Attack

```
def find_passwords_from_hashes(dictionary_file, hashes):
    # Create a dictionary of hashed passwords from the dictionary file
    with open(dictionary_file, 'r') as f:
```

```
4
5           hash_dict = {sha512_hash(password.strip()): password.strip() for
    ↪ password in f}
6
7       # Return list of passwords corresponding to the hashes or None if not
    ↪ found
8       return [hash_dict.get(h) for h in hashes]
```

Upon execution, the code runs for 16.4 ms.

Listing 5: Dictionary Attack

```
1 CPU times: user 14.8 ms, sys: 1.93 ms, total: 16.7 ms
2 Wall time: 16.4 ms
```

2.3 Dictionary with Salt

A salt is a random sequence of characters. Its inclusion complicates dictionary attacks since each word must be paired with every potential salt, increasing computational demands. In our case, each hash has its unique salt. This means we combine each dictionary word with the salt and then use the SHA-512 method to turn it into a hash. Python's hashlib helps with this.

Listing 6: Dictionary Attack with Salts

```
1 def sha512_hash_with_salt(string, salt):
2     """Hashes a string combined with salt using SHA-512 and returns the
    ↪ hexadecimal representation."""
3     # Encode the string and salt as bytes and hash using SHA-512 algorithm
4     return hashlib.sha512((string + salt).encode()).hexdigest()
```

To make the code more modular,we have created a function that loads words from the file into a list.

Listing 7: Loading from file

```
1 def load_dictionary_from_file(filename):
2     """Load words from a given file into a list."""
3     with open(filename, 'r') as file:
4         return [line.strip() for line in file]
```

The primary function attempts to decipher a list of salted password hashes using the same dictionary file. It initializes an empty list to store results. For each salted hash [8] and its associated salt, the function iterates over the dictionary words, hashes each word with the salt, and compares it to the given salted hash. If a match is found, the word, hash, and salt are recorded, and the loop breaks for that hash. If no match is found after checking all dictionary words, we have indicated it is not found. The function then returns the list of found passwords.

Listing 8: Salt Dictionary Attack

```
1 def crack_salted_passwords(dictionary, salted_hashes):
2     # Create an empty list to store the cracked passwords
3     cracked_passwords = []
```

```
    # Loop through each group of salted hashes
    for salted_hash_group in salted_hashes:
        for salted_hash, salt in salted_hash_group:
            found = False
            # Loop through each word in the dictionary
            for word in dictionary:
                # Hash the word with the salt
                if sha512_hash_with_salt(word, salt) == salted_hash:
                    # If the hash matches, append the word to the list
                    cracked_passwords.append((word, salted_hash, salt))
                    print(f"Hash: {salted_hash}\nSalt: {salt}\nPassword: {word}\n{'-'*150}")
                    found = True
                    break

            # If no match is found for this hash, append an indicator (optional)
            if not found:
                cracked_passwords.append((None, salted_hash, salt))
                print(f"Hash: {salted_hash}\nSalt: {salt}\nPassword: NOT FOUND\n{'-'*120}")

    return cracked_passwords
```

Adding salts extends the execution time in dictionary attacks, as observed below.

Listing 9: Dictionary Attack with Salts Time

```
CPU times: user 78.6 ms, sys: 4.6 ms, total: 83.2 ms
Wall time: 87.5 ms
```

3 Miniproject: Security Comparison

In our mini-project, our objective was to assess the comparative efficiency of Rust in contrast to Python. Rust, a compiled language, inherently possesses superior control over system resources and excels in memory management. Our working hypothesis posits that Rust will outperform Python in speed and performance.

3.1 Python vs Rust

- **The Goal**: We set two primary objectives for our project:
  - **Comparison** Reimplement all three Rust tasks and evaluate their performance against Python.
  - **Speed Test**Evaluating execution speed for generating strings of various lengths in both Python and Rust for the brute force algorithm.
- **The Methods**: At the outset, we needed to familiarize ourselves with Rust. [6]This meant understanding its syntax, principles and best practices. A lot of our foundational knowledge came from official documentation and hands-on practice. To ensure a fair

comparison, we made the Python and Rust functions as similar as possible regarding their functionality.

3.2 Deciphering Passwords: Brute Force in Rust

1. `fn sha512_hash(password: &str) -> String`: this function in Rust operates identically to its Python counterpart. [?] It takes a password as input and computes its SHA-512 hash.

Listing 10: SHA-512 Hashing in Rust

```rust
// Function to compute the SHA-512 hash of a given password.
fn sha512_hash(password: &str) -> String {
    // Initialize the SHA-512 hasher.
    let mut hasher = Sha512::new();

    // Update the hasher with the password.
    hasher.update(password);

    // Finalize the hash computation.
    let result = hasher.finalize();

    // Return the hash as a hexadecimal string.
    format!("{:x}", result)
}

```

2. `pub fn brute_force_decode(`
   `hashes: &[&str],`
   `max_length: usize)`
   `-> HashMap<String, String>:`
   The Rust algorithm mirrors the Python version, adapting to Rust's syntax, data types, and libraries. In the Rust adaptation, we retained the core logic while explicitly defining data types (e.g., String and HashMap). Rust's syntax uses fn for functions, for loops, and let for variables. We employed the itertools crate for combination generation, aligning with Rust's conventions while preserving the algorithm's functionality. Multi-cartesian dot products, such as the multi_cartesian_product(), streamline the generation of combinations from multiple sets or iterators, eliminating the need for more nested loops and facilitating tasks like password generation more efficiently.

Listing 11: Brute Force Rust

```rust
pub fn brute_force_decode(hashes: &[&str], max_length: usize) ->
    ↪ HashMap<String, String> {
    let characters = "abcdefghijklmnopqrstuvwxyz0123456789";
    // Initialize an empty HashMap to store the found passwords
    ↪ because empty HashMaps are
    let mut found_passwords: HashMap<String, String> = HashMap::new()
    ↪ ;

    // Outer loop iterates over the specified maximum password length
    ↪ .
```

```
    'outer: for length in 1..=max_length {
        // For each length, generate all possible combinations of passwords.
        for password_chars in (0..length).map(|_| characters.chars()).multi_cartesian_product() {
            // Convert the character combination into a String.
            let password: String = password_chars.into_iter().collect();
            // Compute the SHA-512 hash of the generated password.
            let hash = sha512_hash(&password);
            // Check if the computed hash exists in the provided list of hashes.
            If hashes.contains(&hash.as_str()) {
                // If a match is found, store the password in the HashMap.
                found_passwords.insert(hash, password);
                // If all hashes have been found, break out of the loops.
                if found_passwords.len() == hashes.len() {
                    break 'outer;
                }
            }
        }
    }
    // Return the HashMap containing the decoded passwords.
    found_passwords
}
```

3.3 Deciphering Passwords: Dictionary Attacks in Rust

Standard Dictionary Attack in Rust:
The Rust function `dictionary_attack(dictionary_file: &str, hashes: &[&str])` performs a classic dictionary attack. It reads potential passwords from a dictionary file, hashes each using the `sha512_hash` function, and stores matching hashes in a `HashMap`. While it incorporates Rust-specific constructs like `filter_map`, its core functionality mirrors the Python version.

Listing 12: Standard Dictionary Attack

```
// This function reads a file containing a list of passwords and returns a HashMap containing the SHA-512 hashes of the passwords.
pub fn dictionary_attack(dictionary_file: &str, hashes: &[&str]) -> HashMap<String, String> {
    // Initialize an empty HashMap to store the found passwords.
    let mut hash_dict = HashMap::new();
    if let Ok(lines) = read_lines(dictionary_file) {
        for line in lines {
            // For each line in the file, compute the SHA-512 hash of the password.
            if let Ok(password) = line {
```

```
                // Store the hash and password in the HashMap.
                let hash = sha512_hash(&password);
                hash_dict.insert(hash, password);
            }
        }
    }
    // Filter the provided list of hashes only to include the ones in the dictionary.

    hashes.iter()
        .filter_map(|&hash| hash_dict.get(hash).map(|password| (hash.to_string(), password.to_string())))
        .collect()
}


```

Salted Hash Dictionary Attack in Rust:
The function `crack_salted_passwords(dictionary: &[String], salted_hashes: &[(String, String)])` caters to salted hashes. Salting appends a random string to a password before the hashing process, bolstering security. This function hashes each dictionary password with the salt using the `sha512_hash_with_salt` method. Corresponding hashes signify decoded passwords, subsequently stored in the `HashMap`. The function then returns this assortment of decoded salted passwords.

```
// This function attempts to crack a list of salted hashes using a provided dictionary.
pub fn crack_salted_passwords(dictionary: &[String], salted_hashes: &[(String, String)]) -> HashMap<String, String> {
    let mut cracked_passwords: HashMap<String, String> = HashMap::new();

    // Iterate over the provided list of salted hashes.
    for (hash, salt) in salted_hashes.iter() {
        // Iterate over the provided dictionary.
        for word in dictionary {
            // Compute the SHA-512 hash of the current dictionary word and salt.
            if sha512_hash_with_salt(word, salt) == *hash {
                cracked_passwords.insert(hash.clone(), word.clone());
                // Print the cracked password and break out of the loop.
                println!("Hash: {}\nSalt: {}\nPassword: {}\n{}", hash, salt, word, "-".repeat(60));
                break;
            }
```

```
            }
        }

        cracked_passwords
}
```

3.4 Benchmarking

To evaluate the performance of the assigned task functions in Rust, we have employed Criterion's bench function. This statistics tool executes the program 100 times to obtain precise measurements. Criterion [7] provides comprehensive insights, including average execution time and standard deviation. We have established a bench directory where the my_bench.rs function is located. We have incorporated the three tasks within the file. To kickstart the benchmarking process, navigate to the appropriate directory using the cd command and invoke the cargo bench. Upon completion, Criterion generates a comprehensive report which can be conveniently accessed and reviewed for insights.

We could not identify a direct counterpart to Rust's Criterion for benchmarking for Python. However, we conducted equivalent performance tests by leveraging **'matplotlib'** [5] for visualization and **'timeit'** within Jupyter. Using matplotlib, we visualized a performance comparison between Python and Rust across the tasks. The bar chart represents execution times for each job and indicates that Rust consistently outperforms Python.

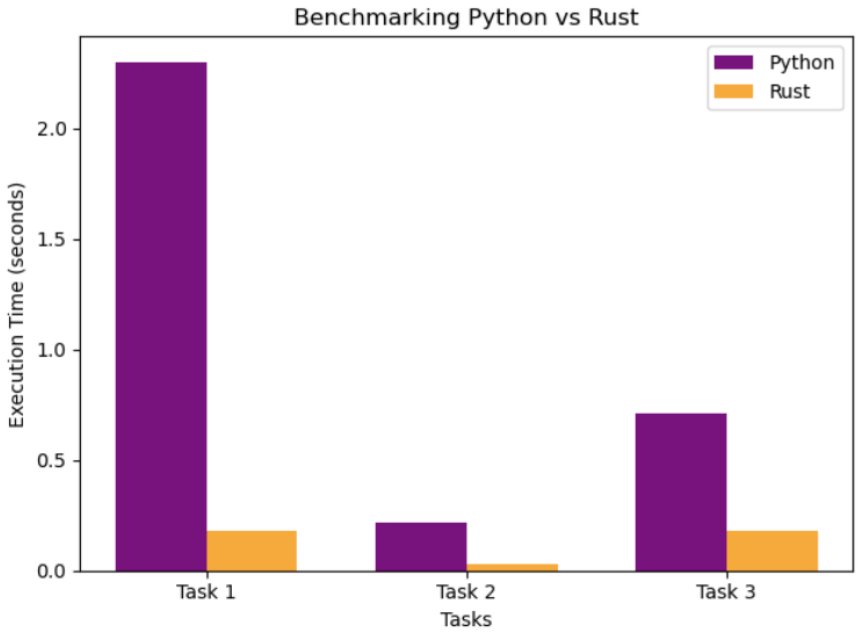

Figure 1: Rust is three times faster

Task	Python Execution Time (seconds)	Rust Execution Time (seconds)
Task 1	0.85	0.18
Task 2	0.22	0.03
Task 3	0.71	0.18
Overall Average	0.59	0.13

Table 1: Execution Times for Python and Rust by Task

Metric	Value
Python Overall Average Time (seconds)	0.59
Rust Overall Average Time (seconds)	0.13
Percentage Rust is Faster	78%

Table 2: Overall Average Execution Times and Speedup Percentage

We expanded our brute force testing in Python and Rust for a more comprehensive analysis, iterating through character lengths ranging from 1 to 5 and running it ten times.

```
def generate_random_password(length):
    characters = string.ascii_lowercase + string.digits
    return ''.join(random.choice(characters) for _ in range(length))
...[More in the Jupyter Notebook]
def time_brute_force_for_length(hash_val, target_length, runs=10):
    times = []
    for _ in range(runs):
        start_time = time.time()
        _ = brute_force_decode_for_length(hash_val, target_length)
        elapsed_time = time.time() - start_time
        times.append(elapsed_time)
    avg_time = statistics.mean(times)
    std_time = statistics.stdev(times)
    return avg_time, std_time
```

This provided deeper insights into performance differences across varying complexities. After we received our results for both programming languages, we saved them in CSV files. The table below illustrates the results. As observed, Rust is considerably faster than Python

Length	**Rust Time (s)**	**Python Time (s)**	**Difference %**
1	0.000006	0.000062	90.32%
2	0.000204	0.001257	83.76%
3	0.006563	0.034218	80.81%
4	0.437300	0.855632	48.90%
5	9.294659	37.880536	75.47%

Table 3: Average Performance Comparison: Python vs Rust for Randomly Generated Passwords

and is, once again, a considerably stronger competitor.

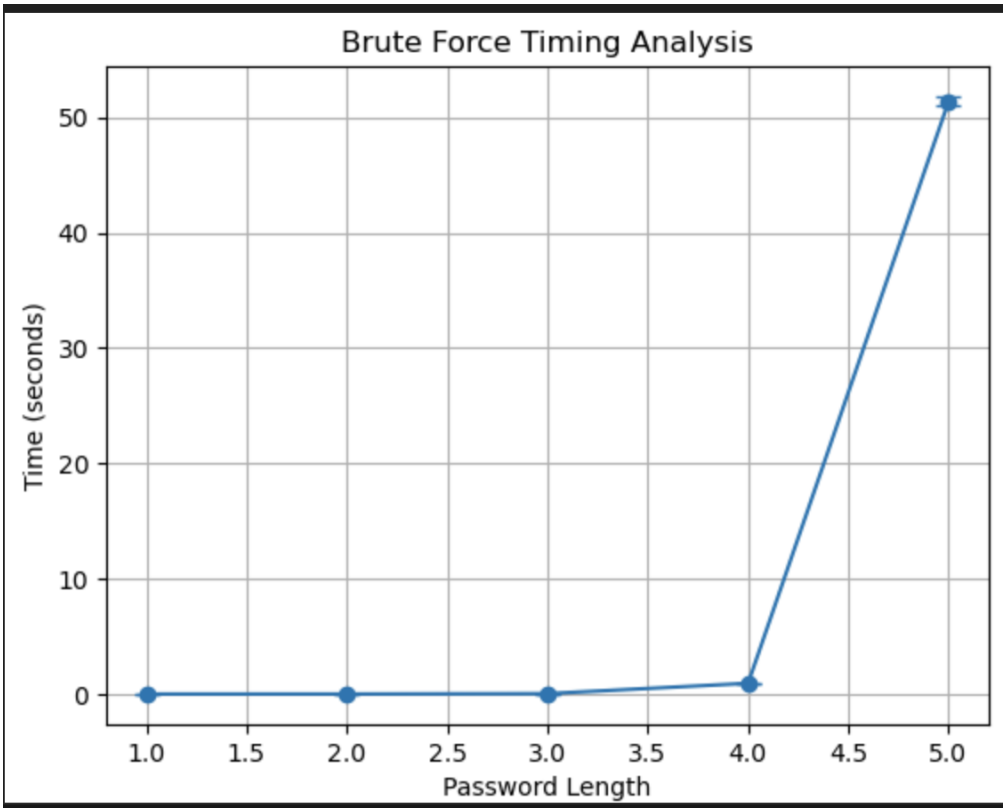

Figure 2: Brute Force analysis in Python

3.5 Conclusions

In comparing Rust and Python, Rust consistently showcased superior performance across all the tasks. While Python is revered for its ease of use and readability, Rust emerges as a powerhouse in tasks demanding higher computational efficiency. This highlights the significance of choosing the appropriate tool for cryptography and proves our hypotheses. Though Rust's performance is commendable, it's essential to acknowledge the trade-offs, such as its steeper learning curve. However, for performance-critical applications, the optimization potential of Rust can lead to efficiency.

3.6 Learning Outcome

Through this analysis, we acquired proficiency in a new programming language. While our code implementation showed promise, there's room for improvement, particularly through modular coding for better organization and maintainability. This experience also expanded our knowledge of libraries, honed our data visualization skills, and deepened our understanding of benchmarking across programming languages. It emphasized the importance of choosing the right tool for specific tasks. Notably, we discovered Rust's efficient memory allocation with vectors.

References

[1] "sha2" Rust crate documentation.,2023
https://docs.rs/sha2/latest/sha2/

[2] Shanto Roy, "Password Cracking through Dictionary Attack in Python," 2023.
https://shantoroy.com/security/password-cracking-through-dictionary-attack-in-python/

[3] Python Software Foundation, "hashlib - Secure hashes and message digests," 2023.
https://docs.python.org/3/library/hashlib.html

[4] The Rust Project Developers, "Rust Itertools - MultiProduct," 2023.
https://docs.rs/itertools/latest/itertools/structs/struct.MultiProduct.html

[5] LearnPython.com, "Calculating the Average in Matplotlib," 2023.
https://learnpython.com/blog/average-in-matplotlib/

[6] The Rust Programming Language, "Learn Rust," 2023.
https://www.rust-lang.org/learn

[7] Learn Criterion,2023
https://docs.rs/criterion/latest/criterion/

[8] Work with salts,2023
https://stackoverflow.com/questions/9594125/salt-and-hash-a-password-in-python