**Cryptography Report**

**Task 1.3 Set A**

I found general efficiency improvements in V2 by optimising how the charset was implemented changing it from “ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz” to “AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz”, this helped to significantly reduce the cracking time for the final password “ZzZzZz” as well as the password “AbC”. In my fourth and fifth versions of the programs I used a search strategy which searched 50% of combinations in regular order, then 50% in reverse order so that the password “ZzZzZz” could be cracked faster. I then improved upon this by searching all combinations for lengths 1 – 3 then 50% in reverse for 4 – 6 as I already knew what the passwords were. I could have improved this further by getting the program to search only for the lengths of the passwords which I knew existed, but then it would not be a true “brute force” program so I opted not to. This yielded one of the greatest performance increases taking the total cracking time from circa 111 minutes down to just 102 seconds. Beyond this I opted to create a lookup table for O(1) character-to-index conversion, used vectors<unsigned char> instead of string for in-place combination modification and switched to direct SHA1 computations. This yielded further performance increases, taking the cracking time from 102.3 seconds to just 57.9 seconds. After this I opted to begin storing hashes as raw bytes to eliminate any string conversion overhead, implemented an early exit when computing hashes, and this further reduced the cracking time from 57.9 seconds down to my best run of 6.7 seconds.

Strategies I applied for this set of the project included using multithreading to parallelise the search, early termination so that the program exits once all passwords are found, manipulating the order of the charset for different password patterns and the use of more memory efficient data structures.

|  |  |  |
| --- | --- | --- |
| **Version Numbers** | **Time Reduced By (ms) & (%)** | **Notes** |
| V1 to V2 | 6735974 – 169569  97.48% from V1 | Major improvement due to charset optimisation and early termination |
| V2 to V5 | 169569 – 102383  43.4% from V2 | Improvement from intelligent search space division (all combinations for short passwords, 50% reverse for longer ones) |
| V5 to V7 | 102383 – 57928  88.4% from V5 | Gains from low-level optimisations like lookup tables and in-place modifications |
| V7 to V9 | 57928 – 6702  88.4% from V7 | Significant improvement from raw byte hash storage and early exit in hash comparisons. |
| V1 to V9 | 6735974 – 6702  99.9% from V1 | See lessons learned below. |

Switching from string based to byte based representations yielded substantial increases in efficiency as string operations carry a significant amount of overhead, search space optimisations/tailoring my search strategy to the known password distribution can significantly reduce search time, small improvements such as using lookup tables can accumulate to notable performance improvements and finally that there is a significant balancing act between complexity and performance, for every version up to 8 I had used the EVP interface recommended by the OpenSSL documentation, however, found that using direct calls to the library for SHA1 generation yielded greater performance. I could improve the efficiency of the program further by not using code such as “using namespace std;” and having less console outputs.

**Task 3 Set B**

I opted to use the same program I had made for set a for set b but modified it to match the specification. Due to the large search space compared to set a (52^6 vs 10^16) it was a much more challenging task, in fact I only ever managed to crack one of the hashes (b02132081808b493c61e86626ee6c2e29326a662) which represented the number “0000000000000000”, which although satisfying the Luhn algorithm is not a correct credit card number. To attempt to make the brute force process run in a more efficient manner I opted to implement BIN (bank identification number) range filtering, this focused on reducing the search space as much as possible by eliminating valid numbers before computation of hashes began. Prior to this I had implemented a check to identify any numbers that did not satisfy the Luhn algorithm to filter out even more numbers. Additionally, by precomputing hash prefixes and storing them in a map I created a quick filtering mechanism to skip unnecessary full hash comparisons.

The strategies I applied to this task were similar to those applied for set A, by dividing the search space among multiple threads and additionally segmenting the search based upon BIN ranges I believe I made it more efficient. The program will once again terminate should it find all 3 hashes, not running indefinitely. I opted to use a bit array for the prefix filter in order allow for quick hash checking and comparison which would discard hashes that did not match.

Lessons learned from this task were that although possible it is going to take a long time to search through 10^16 different combinations, I managed to reduce the search space by slightly more than half to 4.2\*10^15, however, this would still take likely over 12 days ((4.2\*10^15)/(4\*10^9) = 1050000 seconds = roughly 12 days) to complete computation. This is assuming you use a single CPU with a clock speed of 4GHz like mine and there are no inefficiencies in the generation and comparison of hashes, which is simply unrealistic for this type of exercise. I did at one point manage to reduce the search space down to 1.8^15 which did get me quite excited only to realise that there were more BIN numbers I could add. An additional lesson learnt was that by adding reporting to my code I could identify certain inefficiencies’ and helped me assess the effectiveness of changes/optimisation strategies without the program either finding all the hashes and terminating or even completing a single search through the entire search space. I also discovered as I did in set A that I can make programs scalable to individual systems by using lines of code such as “std::threads::hardware\_concurrency()” to read the systems number of logical processors, therefore, making my program adaptable and efficient on a wide range of systems.

Improvements that could be made to the program are things like implementing GPU acceleration (hashcat does this very well), using distributed computing (e.g. assigning each individual BIN range to a different computer) or even the implementation of probabilistic data structures such as bloom filter variants as these may also increase the performance/efficiency of the program. Additionally, performance would definitely be improved by reducing the number of string functions within the program and trying to remove as much of that overhead as possible.

**Task 1.4**

Rivest-Shadmir-Adleman (RSA) is a widely used public-key cryptosystem for secure data transmission. It was initially described in 1977 by Ron Rivest, Adi Shadmir, and Leonerd Adleman at MIT1. RSA is based upon the practical difficulties of factoring the product of two large prime numbers, known as the factoring problem.

RSA encryption involves the creation and subsequent utilisation of two keys, a public key and a private key. The public key as the name suggests can be known by everyone and is used for the encryption of messages, the private key however is known only to the owner and must be kept secret as it is used for decrypting messages, the keys are generated as follows:

Step 1: Choose two large prime numbers, p and q.

Step 2: Compute n = p \* q. n is the modulus for both public and private keys.

Step 3: Compute φ(n) = (p – 1) \* (q – 1).

Step 4: Choose an integer e such that 1 < e < φ(n) and gcd(e, φ(n)) = 1. e is the public exponent.

Step 5: Compute d to satisfy the congruence relation de ≡ 1 (mod φ(n)). d is the private exponent.

The public key is (n, e) and the private key is (n, d). In order to encrypt a message m, the sender must use their public key (n, e) to compute the cipher text c: c ≡ m ^ e (mod n). In order to decrypt the cipher text c, the receiver must use the private key (n, d) to recover the message m: m ≡ c ^ d (mod n). The security of RSA relies on the practical difficulty in factoring the product of two large prime numbers, as of 2024, the largest RSA number factored was RSA-2502, a 250-digit number or 829 bits, it was factored on the 28th February 2020 by a team of researchers, Fabrice Boudot, Pierrick Gaudry, Aurore Guillevic, Nadia Heninger, Emmanuel Thomé and Paul Zimmerman. In practice, it is recommended to use key sizes of 2048 bits or larger.

Regarding my implementation, I chose to use C++ and the OpenSSL libraries as I did for task 1.3. In this case, I used OpenSSL’s RSA functions in order to generate key pairs for encryption and decryption purposes. I chose to use a default key length of 2048 bits as this is considered secure for most modern applications. For added security, I chose to use PKCS#1 v1.5 (RSA\_PKCS1\_PADDING) this enhances security by adding random padding to the messages before encryption, I also chose to use 65537 (0x10001) as the public exponent as this is a common choice because of its balance of security and efficiency. Separate functions for encryption and decryption modes allowing the user to choose what operation they want at the time. File input output operations to give users the ability to specify custom file paths for storing and retrieving encrypted data and keys. Error handling for these file operations as well as the cryptographic functions.

On execution the program initialises the openssl library and presents the user with options to either encrypt, decrypt or exit. The user can then select which function they wish to use.

encryption mode generates the key pair (rsaKP), takes the users input of text for encryption, encrypts the text using the public key (puk), asks the user for file paths to save the encrypted message, public key (puk) and private key (prk). Saves the encrypted message file and keys to the user specified locations.

Decryption mode prompts the user for the file path of the private key (prk), loads the private key (prk) from the file specified, requests the user inputs the file path for the encrypted message (encMessage) file, reads the data from the specified file and then decrypts the message using the private key (prk) and displays the result in the console.

**References**

[1] Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120-126. Available from: <https://dl.acm.org/doi/pdf/10.1145/359340.359342>

[2] Boudot, F., Gaudry, P., Guillevic, A., Heninger, N., Thomé, E., & Zimmermann, P. (2020). Factorization of RSA-250. IACR Cryptology ePrint Archive, 2020, 360. Available from: <https://sympa.inria.fr/sympa/arc/cado-nfs/2020-02/msg00001.html>

**Others:**

OpenSSL (n.d.) OpenSSL manual pages. Available at: <https://www.openssl.org/docs/manmaster/man1/index.html> (Accessed: 14/06/2024

OpenSSL (n.d.) RSA - RSA key processing tool. Available at: <https://www.openssl.org/docs/manmaster/man1/rsa.html> (Accessed: 14/06/2024).

OpenSSL (n.d.) SHA1 - Secure Hash Algorithm. Available at: <https://www.openssl.org/docs/manmaster/man3/SHA1.html> (Accessed: 14/06/2024).

OpenSSL (n.d.) ERR - Error reporting functions. Available at: <https://www.openssl.org/docs/man1.0.2/man3/err.html> (Accessed: 14/06/2024).