

Developing an Application for Researching the RSA Algorithm Behavior on a Multithread Platform

Nina Sinyagina¹, Gergana Kalpachka¹, Velko Todorov², Ventsislav Kalpachki³

¹South-West University "Neofit Rilski", 66 Ivan Mihailov Str., Blagoevgrad, Bulgaria

²Sofia University "St. Kl. Ohridski", 15 Tsar Osvoboditel Blvd., Sofia, Bulgaria

³Technical University of Sofia, 8 Kiment Ohridski Blvd., Sofia, Bulgaria

Abstract – The article is focused on issues concerning the design of all needed software for researching the speed of the RSA encryption algorithm executed on a multithreaded platform. The base structure of the application is described in detail as well as the testing plan and algorithm. Shown is a method to handle and control all threads during the processes of encryption and decryption. The final results are visualized graphically through diagrams.

Keywords – multithreading, cryptographic algorithms, asymmetrical encryption algorithm RSA.

1. Introduction

The article is a continuation on a previous research by the same authors on multithreaded RSA algorithms and their conceptual and architectural model [1].

The current article is focused more on the practical application of the problem: creating an application for analyzing the multi-core RSA algorithms in terms of speed. To achieve its goal, the application needs to meet the following criteria [2], [3]:

- use of accurate time measuring means;
- correct execution and functionality of the RSA encryption and decryption algorithms;

DOI: 10.18421/TEM94-07

<https://doi.org/10.18421/TEM94-07>

Corresponding author: Gergana Kalpachka,
South-West University "Neofit Rilski", 66 Ivan Mihailov
Str., Blagoevgrad, Bulgaria.


Email: kalpachka@swu.bg

Received: 10 August 2020.

Revised: 12 October 2020.

Accepted: 18 October 2020.

Published: 27 November 2020.

 © 2020 Nina Sinyagina et al.; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.

The article is published with Open Access at www.temjournal.com

- automation of the test procedures;
- good visual representation of the end results [4].

The application user interface needs to be as simple as possible, not taking too much computational resources [5], [6]. The test machine needs not to be overloaded or working on other tasks during the test, except the necessary operating system processes. This ensures correct and accurate test results, with small margin for error [7], [8].

The test user interactions with the application should be limited to only two: when passing the input parameters and when receiving the output test results. For better visualization of the final results, a software for graphical representation of data can be used to project the test output [9].

Since the application is for test purposes only, all input parameters can be set as global parameters. For convenience they are reduced to minimum, while they are still able to define all types of procedures required to correctly measure the end results. In order to achieve this, additional blocks need to be added for automatic changes in the input data like generating new RSA key pairs [10], [11], generation of data for encryption with different length, control over the used threads by the system during the test execution, etc.

2. Application Architecture

The programming language used for creating the application is Java. In Java, the different objects are defined as *classes*. The classes are a combination of *variables* (object attributes) and *functions* (object behavior) connected in a logical structure. Combining classes produces *packages*. Packages are used to group classes in one functional unit, which allows for easier integration and migration into other projects. The code needs to be properly split into classes and packages, which allows easier readability of all functions.

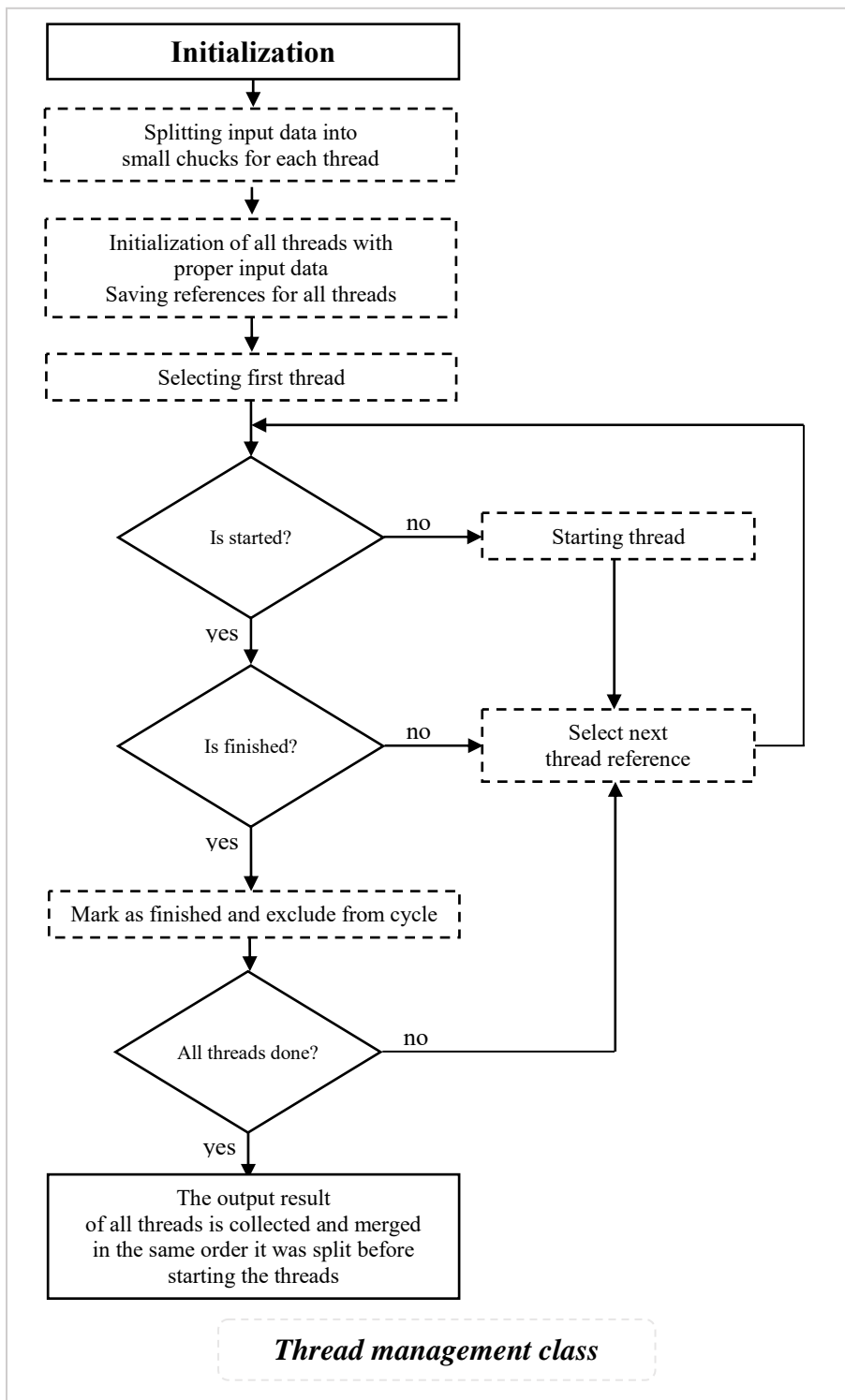


Figure 1. Structure of the encryption and decryption processes management

The test application made out of two packages:

- *multithread_rsa*: consists of all functions for the execution of the multithreaded RSA algorithm's encryption and decryption processes. The package is independent and can be exported and used separately as an independent task. It consists of the classes for encryption and decryption, both inheriting the thread class (needed for thread management) and a class for RSA key pair generation. These are the minimum requirements needed to execute the required task.
- *rsa_speed_test*: consists of all functions for test task management and time measurement. The package is an example implementation of the *multithread_rsa* package. It consists of the following objects: class for time measurement and methods for implementation of the *multithread_rsa* package.

Classes for Encryption and Decryption

The classes *RSA_encrypt_thread* and *RSA_decrypt_thread* (from the *multithread_rsa* package) are used to encrypt and decrypt the input data. Both inherited the thread class, which allows them to be executed multiple times in parallel. The parameters for each of the class are as follows:

- *id*: the start order of each thread needs to be noted and then used to correctly merge the output data in order;
- *encryption/decryption keys*: RSA keys are required to run the encryption and decryption algorithm, the public key is required for encryption and the private key for decryption;
- *input data for encryption/decryption*: the information each thread is required to convert (encrypt or decrypt), for this test the type of the variable is set to *string*, to be more flexible and easier to visualize;
- *output data*: every thread needs to collect the output data until it is needed for collection;
- *finish flag*: a flag indication when the job for the current task is done. This is required for easier thread management and data collection.

All parameters are defined as *private* for the class and cannot be directly accessed by other classes. This lowers the risk of invalid data and computational errors during the processes.

The public methods of the encryption and decryption processes are:

- *initialization*: start of thread and setting initial parameters (*id*, encryption keys and data to work with);
- *“run()” method*: the *run()* method from the thread class needs to be defined for every child class to use the parallelism. In this scenario it will just point to the private method *encrypt()* or *decrypt()*;
- *return of output data*: result of the operation of the thread (encryption or decryption). In a similar way to the input, the output is also type *string*, to be easier to visualize;
- *return of thread id*: a simple function, outputting the *id* of the current thread, required when collecting and arranging data from all threads;
- *status check*: method to validate if the current thread has finished its task and availability of its output.

The only private methods in the encrypt and decrypt classes are the corresponding functions called by the *run()* method and *task done* method – used to update the finished flag and signalize that the thread work is done.

3. Structure of the Thread Management

The module for managing the multiple RSA thread tasks consists of two functions: one for controlling the encryption block and one controlling the decryption block. Both of them follow the same steps to execute the required tasks (Fig. 1.).

The first step is to initialize the functions. This includes setting the data for encryption/decryption, passing the public or private keys required and the number of threads that will work on the job.

After the initialization the input information is split into smaller chunks depending on the number of threads to be used. The goal is to split the data in a way so all threads have relatively similar work time.

Then each thread is being separately initialized by passing all required data to it, including the calculated chunks of data to be encrypted/decrypted. Reference to every thread is saved, so it can be used later on in the management process.

All thread managing operations have the following stages: thread starting, status monitoring, data collection. All stages act as a state machine that cycles through all threads until all of them are done.

The first stage is to check the process start. If a thread is not running, it will be started and this action will be marked. If the thread is running, then the state machine checks its current status (*finish flag*). If the thread has done its task and raised the finished flag, the state machine marks it as ready.

Once all tasks are marked as ready the data collection can take place. The data needs to be saved in the same order as the input was split. This is essential to properly assemble the result.

4. Structure of the Test Method

The main settings for the test method are defined with its input parameters: maximum number of threads to use, RSA key length, size of string to be used for encryption and decryption and number of test repeats (used to minimize the measurement error).

Figure 2. shows the structure of the test method.

The first step from the test method is to create a random string with the given size. The string will be used to be encrypted and then decrypted back. After that the RSA key pair is generated based on the preset key length.

The next stage is the cycle for executing the encryption and decryption procedures. The cycle is broken once the preset test count is reached. After all tests are done, the average time is calculated and printed out.

After that, the number of maximum threads to use is reduced by one and all tests are redone and their average result is measured again. This is repeated while the number of threads to use is one or more.

5. Test Planning

The research aims to be a base for comparison of the encryption and decryption processes speed of data with different sizes on a multithreaded platform.

To achieve correct and accurate results, the following criteria need to be met:

- *Multiple execution of each test:* average results lower the risk of errors generated by a lot of

random events that can occur during testing and resource usage spikes caused by operating system processes.

- *Idle system during testing:* no other applications should be running on the system while the tests are taking place, so there is a minimum change in the available system resources during all tests. Only the necessary system processes are allowed to run.

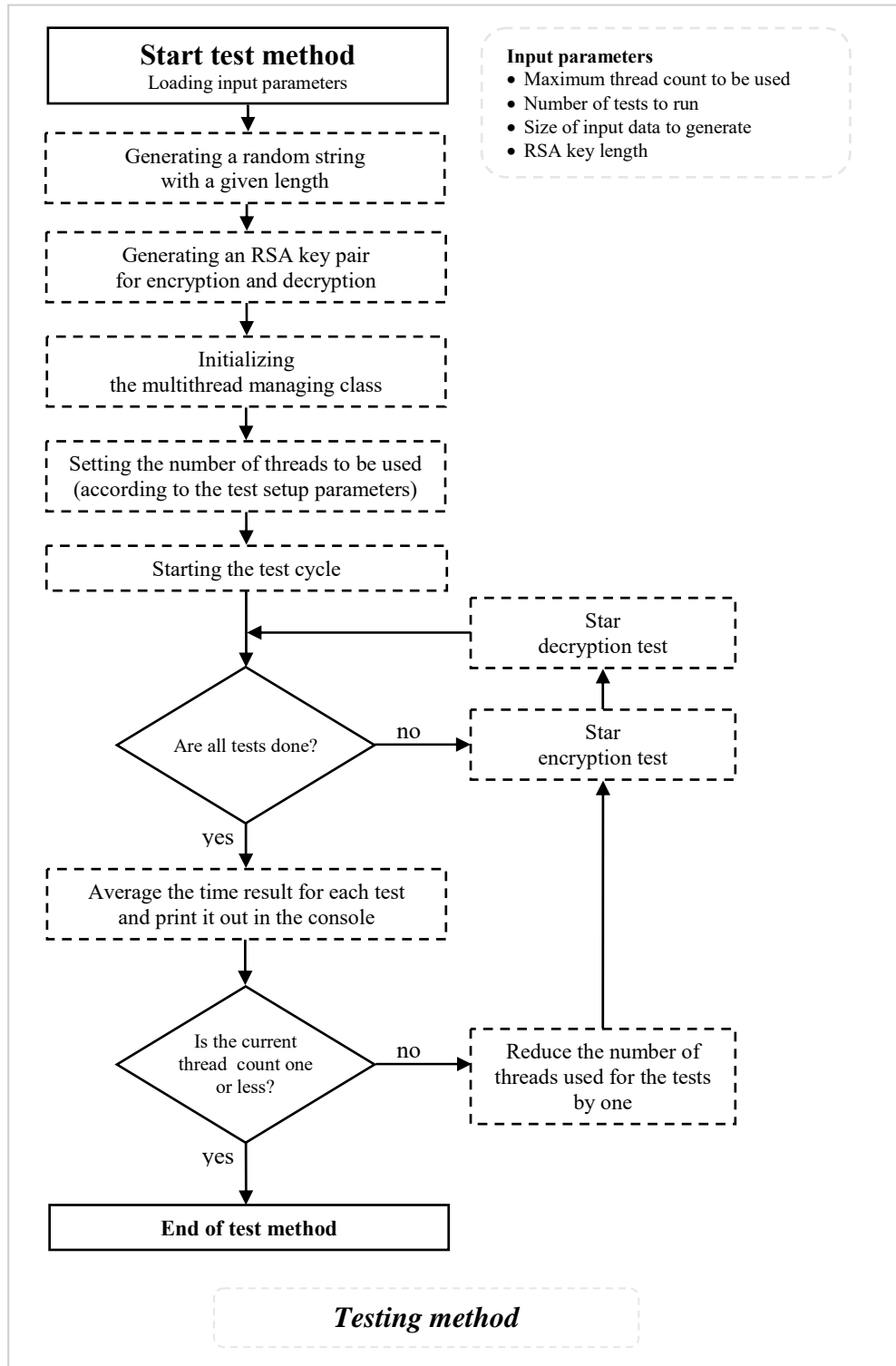


Figure 2. Structure of the test method

- *Comparing tasks with identical input parameters:* when comparing test results, it is essential to use the same data to encrypt and decrypt. The size of the input data is only modified when changing the length of the input data to run a different comparison test.
- *Display result in specific format:* the output test information needs to be presented in an understandable format and needs to be easy to export for visual representation of the data.

The tasks of the test module are: generating an RSA key pair, generating information for encryption/decryption, starting all defined tests and outputting result information.

The following test scenarios/criteria will be evaluated:

- *Speed when using different number of threads:* the main purpose of the research. Requires multiple tests with the same input parameters executed on different number of threads.
- *Speed of encryption/decryption of data with different sizes:* a sub-goal of the research is to find the optimal number of threads to be used for input data with different sizes.
- *Speed when using Intel’s Hyper Threading technology:* the idea is to check if Hyper Threading helps to speed up the RSA encryption/decryption processes.

All tests are using 1024 bit RSA keys.

According to these tests, the main parameters to base the comparison will be:

- number of parallel processes;
- size of input data to encrypt/decrypt;
- time for execution.

6. Results

The tests are run with combinations of the following parameters:

- different sizes of information to encrypt/decrypt (10, 100, 1 000, 10 000, 50 000, 100 000 and 1 000 000 symbols);
- different number of threads used – minimum 1, maximum 8 (4 of these are virtual threads using Intel’s Hyper threading technology);
- each test is run 100 times and only the average result is taken into account.

This makes a total of 56 000 tests.

After running all tests, the result data is saved and visualized into graphs (Fig. 3.–6.).

7. Conclusions

As anticipated, the results show that when using a larger size of input data, the time taken for the encryption and decryption processes is less when using a bigger number of parallel processes.

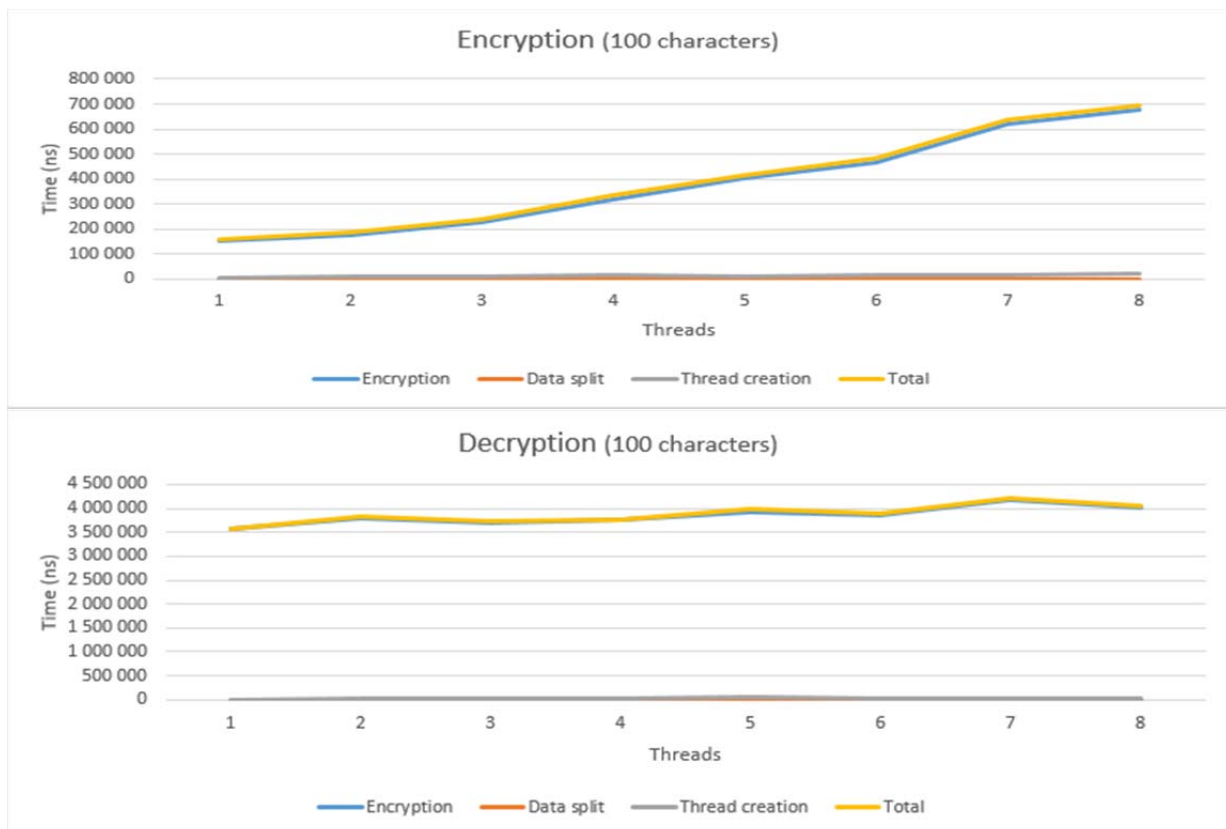


Figure 3. Time to encrypt and decrypt text with size of 100 characters

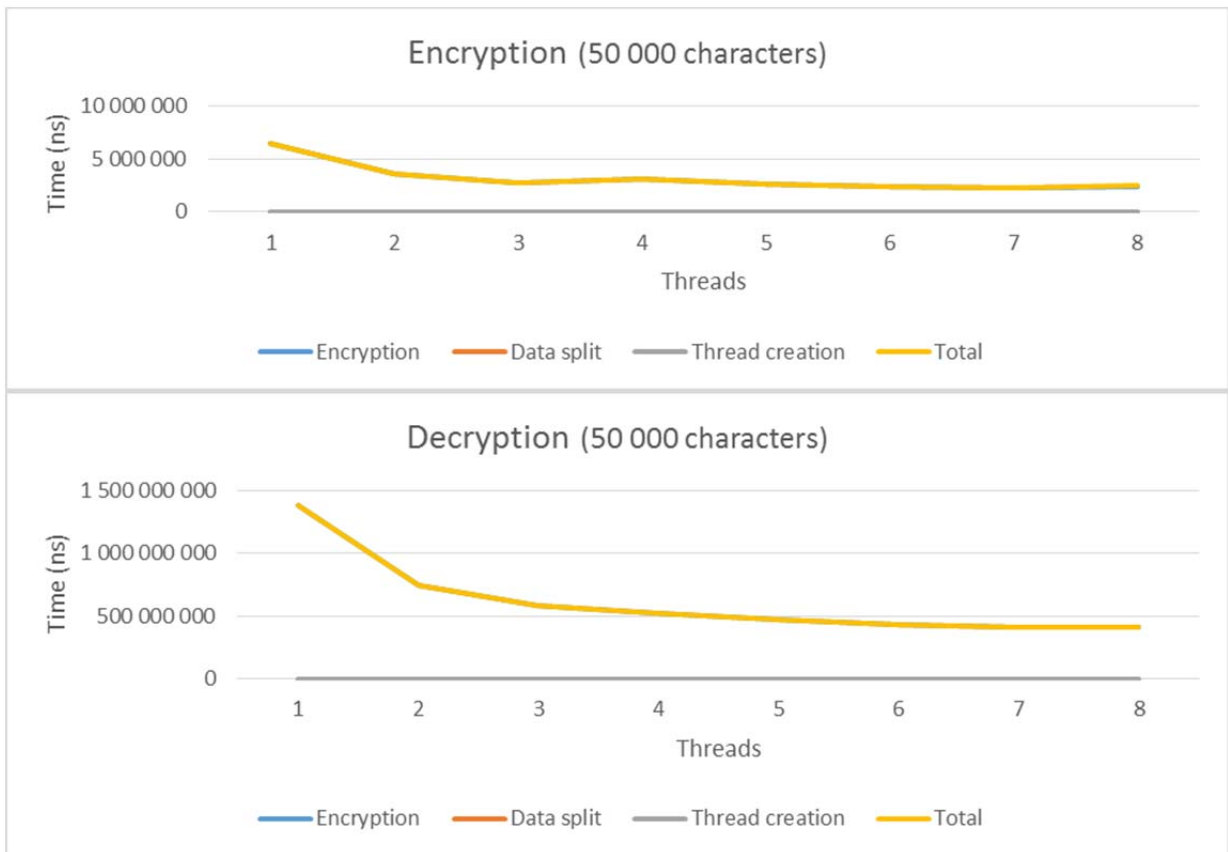


Figure 4. Time to encrypt and decrypt text with size of 50 000 characters

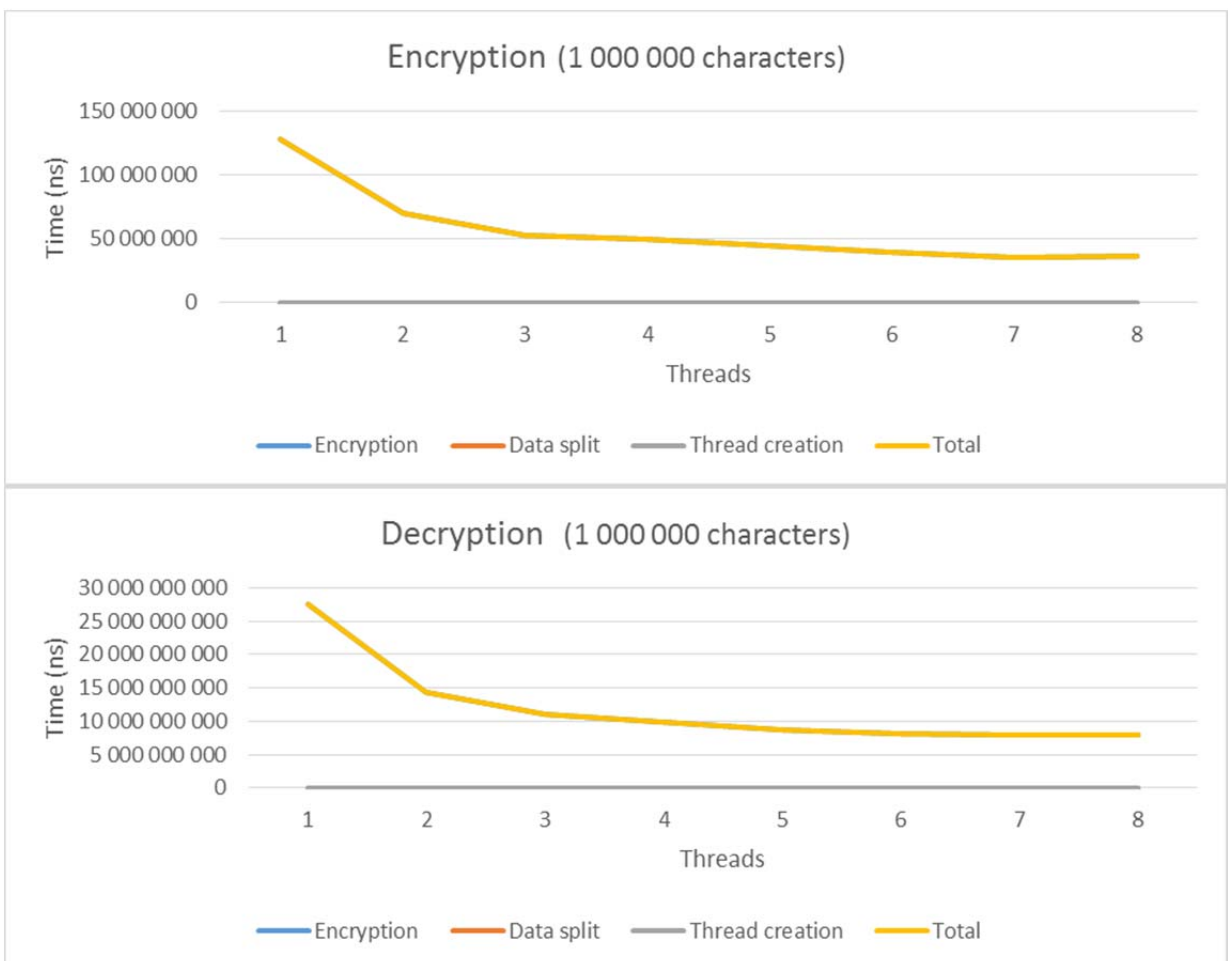


Figure 5. Time to encrypt and decrypt text with size of 1 000 000 characters

According to the output data, the decryption time is much higher than the encryption time, which is expected given that the decryption calculations are much more complicated. Because of this, the data is better visualized when split based on the process type: encryption or decryption. Still the relation of using multithread technology has similar effect on both processes.

Most of the tests show almost 50 % time reduction when using two threads instead of one. Comparing 1 thread with 3 threads gives almost 60 % time reduction and 4 threads – 65 %. Intel’s Hyperthread technology saves additional 5 % time when used on big input data (when using 5-8 threads).

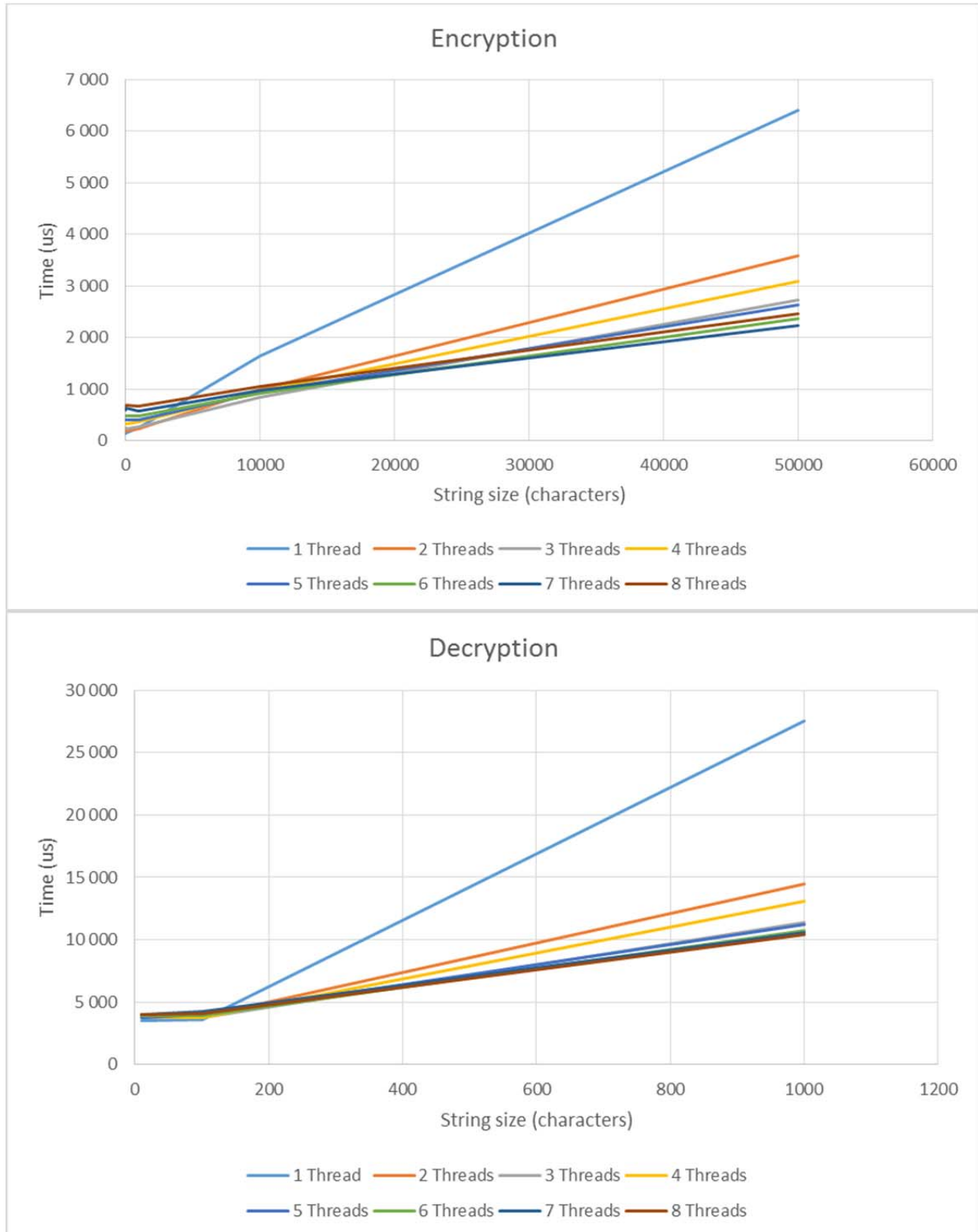


Figure 6. Time for encryption/decryption using different number of threads

In terms of input data size, it is confirmed that multithread has negative impact on data smaller than 100 characters during decryption and 1 000 characters during encryption. With a large number of characters to encrypt, all results clearly show better performance on multiple threads.

The optimal thread count, depending on the data size that needs to be encrypted can be split into three categories:

- Under 1 000 characters – multithreading will not speed up the process significantly, using 1 thread is optimal.
- Between 1 000 and 50 000 characters – the effects of Intel’s Hyperthread technology do not have effect on the results. Real cores provide significant speed improvement, especially up to 3 threads.
- Above 50 000 characters – even Intel’s Hyperthread technology impacts the performance with a few percent.

The source code for all conducted tests and the corresponding parts for RSA encryption and decryption algorithms are specifically designed to be easy to use and simple to port in new projects. The code structure is simple and logical, so it is easy to read and comprehend.

Along with the advancement of technologies, there will always be new methods for faster, simpler and easier execution of complex calculations, required by the RSA algorithm. The described platform in this research can be used as an initial base for any future development in the field.

References

- [1]. Sinyagina, N., Todorov, V., & Kalpachka, G. (2020). Implementation of cryptographic algorithms via multithreading. *Bulgarian Chemical Communications*, (52)A, 220-224.
- [2]. Stokes, J. (2002). *Introduction to Multithreading*. Super-threading and Hyper threading Ars Technica.
- [3]. Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., & Upton, M. (2002). Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1).
- [4]. Fadhil, H. M., & Younis, M. I. (2015). *A Multithreading Implementation of RSA Algorithm on Multicore and GPU: Parallel Processing*. LAP LAMBERT Academic Publishing.
- [5]. Casey, S. (2011). How to determine the effectiveness of hyper-threading technology with an application. *Intel Technology Journal*, 6(1), 11.
- [6]. Fadhil, H. M., & Younis, M. I. (2014). Parallelizing RSA algorithm on multicore CPU and GPU. *International Journal of Computer Applications*, 87(6).
- [7]. Haili, H. K., & Basir, N. (2009). RSA Decryption Techniques and the underlying Mathematical concepts. *International Journal of Cryptology Research, Malaysia*, 1(2), 165-177.
- [8]. Nisha, S., & Farik, M. (2017). RSA Public Key Cryptography Algorithm–A Review. *International journal of scientific & technology research*, 6(7), 187-191.
- [9]. Hruska, J. (2012). *Maximized performance: comparing the effects of hyper-threading, software updates*. ExtremeTech.
- [10]. Jonsson, J., & Kaliski, B. (2003). *Public-key cryptography standards (PKCS)# 1: RSA cryptography specifications version 2.1* (pp. 1-68). RFC 3447, February.
- [11]. Moriarty, K., Kaliski, B., Jonsson, J., & Rusch, A. (2016). *PKCS #1: RSA cryptography specifications version 2.2*, RFC 8017, no. 10.