Parallelized RSA Algorithm: An Analysis with Performance Evaluation using OpenMP Library in High Performance Computing Environment

Md. Ahsan Ayub*, Zishan Ahmed Onik†, Steven Smith‡
Department of Computer Science
Tennessee Technological University
Cookeville, USA
{mayub42*, zonik42†, smsmith23‡}@students.tntech.edu

***Abstract*—RSA algorithm is an asymmetric encryption algorithm used to maintain confidentiality and integrity of data as it is transported across networks. As time has gone on, security and confidentiality has grown in importance leading to more data requiring encryption. Parallelization has become an important aspect in improving the speed and efficiency of processing for encryption algorithms. Improvements in parallel implementations of the RSA algorithm lead to better security and efficiency for parallel systems utilizing the algorithm. In this study, we present a comprehensive survey of methods proposed by researchers for parallelization of the RSA algorithm from 1978 till date. This survey aims to provide a deeper understanding of the possible avenues that can be considered to obtain better performance of the RSA algorithm in parallel environments. To demonstrate the improvements, this paper presents a parallel CPU-based implementation of the RSA algorithm using the OpenMP library. This implementation focuses on parallelizing the exponentiation operation of the algorithm. To provide a robust analysis, the study makes use of a High Performance Computing environment to illustrate results for different scenarios in terms of parallel processing units. Through experimental analysis, the implementation is shown to have greatly improved execution times when compared against serial implementation.**

***Index Terms*—Asymmetric Key Encryption, Cryptography, Parallel Computing, Public Key Encryption, RSA**

I. INTRODUCTION

Encrypting and decrypting data at a higher speed has become an important issue in almost every application that employs cryptographic techniques. It is not only important to ensure no breach of confidentiality and integrity of the communication, but also significant to obtain improved speed in encryption and decryption of the data [7]. The prime motivation of designing and implementing cryptographic algorithms is to make the encrypted information unreadable without the key. In addition, the message / data passing between sender and receiver will remain secret from the outside world as well as its data integrity and origin integrity can be checked during transmission.

Based on the keys used, cryptographic algorithms can be classified as symmetric or asymmetric encryption. On a higher level, symmetric encryption is designed to perform encryption with one private key shared between the sender and receiver. This key is used to encrypt plaintext by the sender and decrypt the cipher text by the receiver. The main concern is passing the key through a secure channel. The process usually completes at high speed with a low storage overhead [19]. This process is specified in IEEE 802.15.4 standard. On the other hand, asymmetric encryption is designed to carry out its tasks with two separate keys: a public key, which is derived from a mathematical function and a private key, which is mathematically linked to the public key. The public key is used to encrypt plaintext or to verify a digital signature; whereas the private key is used to decrypt cipher text or to create a digital signature. Decrypting the ciphertext using a public key will show a different message. Although sharing of the key is not required for this technique, the overall process is computationally intensive. A few notable goals of asymmetric key cryptography are [28] -

- The process of encrypting and decrypting a message will be computationally feasible.
- Deriving the private key from the public key of a particular user should be computationally infeasible.
- Deriving the private key from a chosen plaintext should also be computationally infeasible.

A classic example of asymmetric encryption is the RSA algorithm, which was invented by Ronald Rivest, Adi Shamir, and Leonard Adleman in 1978 [32]. The algorithm is designed to ensure communication remains private after the sender encrypts the message with two keys: sender's private key and receiver's public key. The public and private key pair is mathematically related and generated together so that the message can be encrypted by the sender, and the recipient can decrypt the message. It ensures a third party who is able to see the message will not understand the communication between two parties. Thus, the communication can be made private even when the underlying channel is insecure.

Due to the benefits of the RSA algorithm, the necessity of faster implementations will continue to grow. In our study, we explore the avenues of parallelization of the RSA algorithm in a high performance computing environment. Thus, the main contributions of our paper are -

- A comprehensive survey of the types of work that have already been proposed as well as explored by researchers and industry professionals in this domain.

- Serial and parallel implementations of the RSA algorithm. The parallel implementation utilizes the OpenMP library in a high performance computing environment.
- In the spirit of open science, our developed tool has made open source for everyone's use and available online at *https://github.com/AhsanAyub/RSA_Parallelization*.

The rest of the paper is organized as follows: Section 2 provides an overview of the RSA algorithm. Section 3 explains our experimental methods to implement both sequential and parallelized approach of this public key encryption technique. Section 4 presents the results of our research. Section 5 consists of a comprehensive review of related work in this domain which can be considered as a foundation for further research, followed by discussion in section 6. Section 7 summarizes the paper, its contributions, and future work to improve upon this research.

II. RSA Algorithm

The RSA algorithm is a public key cryptography technique where two keys (private and public) are used to both encipher and decipher the message, M. In order to encrypt M, the sender uses a pair of public encryption keys (e, n), where both e and n are positive integers and $e < n$. Decrypting the message, M requires a decryption key pair (d, n), where d is private key. To elaborate further, the sender enciphers the message with his / her own private key (to ensure integrity) first and then the receiver's public key (to confirm confidentiality). Conversely, the receiver deciphers the message with his / her own private key (which verifies confidentiality) and then the sender's public key (to confirm both origin and message integrity). Thus, the RSA algorithm employs double encipherment on the sender's part and double decipherment on the receiver's part. The encryption, E and the decryption, D processes are more formally defined as [32] -

$$C = E(M) = (M^{d_{\text{(sender)}}} \mod n)^{e_{\text{(receiver)}}} \mod n \quad (1)$$

$$M = D(C) = (C^{d_{\text{(receiver)}}} \mod n)^{e_{\text{(sender)}}} \mod n \quad (2)$$

where C is the ciphertext and M is the message (or plaintext) as well as both are integers in the form of 0 to $n - 1$. It is assumed that the encipherment process is done by the sender in the equation 1 while the decipherment process is done by the receiver in the equation 2.

By following the steps mentioned in the Algorithm 1, both the sender and receiver will be able to derive the public key pair (e, n) separately, which are public knowledge and private key d, that is known only to the owner of the private key. Then, both the sender and the receiver will be able to encrypt and decrypt the message from the equation 1 and 2 respectively.

Selecting two random large prime numbers p and q depends upon the users, but the larger the numbers are, the more difficult it becomes to crack the encryption. The values have to be large enough so that deriving the value of n can become computationally infeasible for every possible values of p and q.

Algorithm 1 Mathematical Operations in the RSA Algorithm

```
Input: $p$, $q$
Output: $n$, $e$, $d$
  1) Choose two large prime numbers, $p$ and $q$
  2) Compute: $n = p \times q$
  3) Compute: Euler totient function [29], $\phi(n) = (p-1) \times (q-1)$
  4) Choose $1 < e < n$ such that e is relatively prime to $\phi(n)$
  5) Compute: $d = e^{-1} \bmod \phi(n)$
```

Rivest et al. [1978] suggested the use of 100-digit (decimal) values for p or q. In addition, the users have to choose the value of k as well; however, the its value will be less than $2log_2(n)$ [32]. If these conditions are properly met, cracking the RSA algorithm has been proven to be extremely difficult through cryptanalysis [21].

III. Experiment Methodology

The experimental setting of this study has been conducted by implementing the RSA algorithm sequentially first, followed by finding the avenue where it could be further improved on with the aid of parallelization in a high performance computing environment. We base our analysis on encrypting a sequence of characters which is passed to our RSA encryption implementation. As per the previous suggestions, we specify the values of p and q as large prime numbers, which gives us a larger computed value for n. Thus, the computation of the key pair (e, d) according to Algorithm 1 becomes easier. The double encryption and decryption method is ensured by following Equation 1 and Equation 2 respectively for the communication of the sender to the receiver. However, it should be noted that the underlying medium of communication and key sharing protocol are out of scope for our study.

The exponentiation operation takes notable amount of time as the values of e (the public key) and / or d (the private key) are big enough to make the encryption process computationally intensive. Table I is a sample output case for a classic sequential approach of the RSA encryption algorithm that illustrates how large the private key, d would be to perform the exponentiation operation for decryption process in this case.

Due to the fact that exponentiation operations increase computational time of the encryption and decryption process of the RSA algorithm, the parallelization of these operations seem significant to aid faster computation in terms of encipherment and decipherment of the message. We design our approach to parallelize RSA algorithm with a focus on the exponentiation operations (Equation 1 and Equation 2). Specifically, we aim to achieve improvement in performance by partitioning the exponentiation task among different independent processing elements. For example, the integer ASCII conversation for the plaintext, "A" is 65. Let us assume, the private key is 20. We can divide the exponentiation task, 65^{20} among multiple threads. If we partition the task among 4 threads, each thread will compute 65^4. Then, the results from each thread will

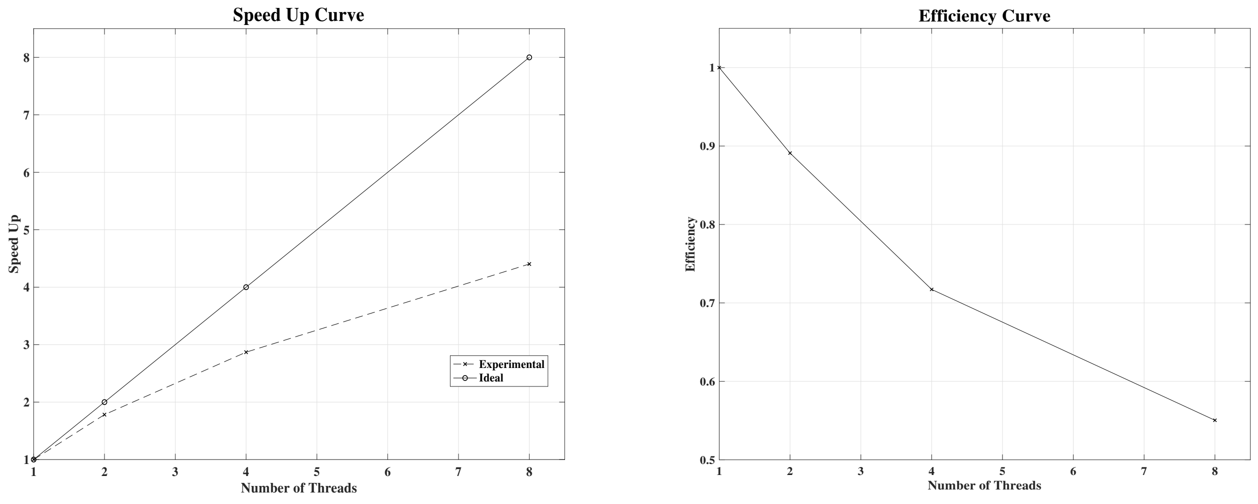

Fig. 1. Speed up curve and efficiency curve of our approach for the parallelized RSA encryption algorithm

TABLE I
AN EXAMPLE OUTPUT CASE FOR RSA ALGORITHM

p
121310724392112718973236715316124404284724276337014109256
345493123019643734208561932419736532241686654101705736136
5214171711713797974299334871062829803541
q
120275242554787488859562207937345121287333878036820754336
538999839551798509887978998691469008091316111533468170508
32096022160146366346391812470987105415233
n (derived from $n = p \times q$)
145906768007583323230186939349070635292401872375357164399
581871019873438799005358938369571402670149802121818086292
467422828157022922076746906543401224889672472407926969987
100581290103199317858753663710862357656510507883714297115
633427889114635351027120327651665184117268598379886721118
37205085526346618740053
$\boldsymbol{\phi(n)}$ (derived from $\phi(n) = (p-1) \times (q-1)$)
145906768007583323230186939349070635292401872375357164399
581871019873438799005358938369571402670149802121818086292
467422828157022922076746906543401224889648313811232279966
317301397777852365301547848273478871297222058587457152891
606459269718119268971163555070802643999529549644116811947
516513938184296683521280
e (selected from $1 < e < n$ such that e is relatively prime to $\phi(n)$)
65537
d (derived from $d = e^{-1} \bmod \phi(n)$)
894894250092744443682285459217730939196695860658842574454
978544564876748396298183909349419732628796167979706089172
836798754993315741611138540888132754881105882471930775825
272784379065040156806234235500672400424666656542323835029
222154936232894721388664458187891279461234078077257026266
44091036502372545139713

need to be multiplied to compute the final exponentiation value. In our study, we demonstrate success in reducing the computational time of the program as we increase the number of threads / processing elements. It is worthwhile to note that, this is the only avenue we observed in the RSA algorithm that is most suitable for parallelization.

We have built our parallelized RSA encryption algorithm tool using the OpenMP library [4] and run our experiments in the high performance computing cluster or environment of Tennessee Technological University [5] in order to test and evaluate our analysis. To streamline and perform encryption as well as decryption of any given plaintext passed in our tool, we have used GNU Multiple Precision Arithmetic Library (GMP) [3]. We measure two performance metrics to illustrate our experimental findings: Speed Up (as shown in Equation 3) and Efficiency (as shown in Equation 4). This enables us to compare how effective the parallelized approach is against the sequential approach.

$$\textit{Speed Up},\ S = \frac{T_S}{T_P} \tag{3}$$

$$\textit{Efficiency},\ E = \frac{S}{p} \tag{4}$$

where T_S is the run time of the sequence implementation, T_P is the run time of the parallel implementation, and p is the number of processors. These equations will be referred to when computing the speed up and efficiency of each parallel execution of the RSA algorithm.

IV. RESULTS

The experimental findings of our research are discussed with an input of a random string-based plaintext which is first converted to the numerical ASCII value of each letter and then certain computations are performed for every letter as described in the methodology section. It is to note that the variant of input strings do not impact the speed up and efficiency for our developed tool; only the execution time will be different. We analyze our results for a large integer input of "528733642850100297336123456789018."

The OpenMP library has allowed us to split the parallelized implementation of RSA algorithm into several processing elements. We observe the execution time of the program, performing both successful encryption and decryption of the input, three times for different thread counts, such as, 1, 2, 4, and 8, in a single physical CPU machine or node. Next, we compute the average run time of each execution which is used to derive the values of the speed up and efficiency. We present Table II to illustrate our findings for each thread count in terms of run time (in milliseconds), speed up, and efficiency.

TABLE II
EXECUTION CASES OF THE RSA ALGORITHM

Threads	1st Run Time	2nd Run Time	3rd Run Time	Avg. Run Time	Speed Up	Efficiency
1	4.5	4.7	4.6	4.6	1.00	1.00
2	2.7	2.6	2.5	2.6	1.782	0.891
4	1.7	1.5	1.6	1.6	2.8688	0.7172
8	1.0	1.1	0.9	1.0	4.4033	0.5504

Due to the fact that not every portion of the RSA algorithm can be parallelized, the speed up values are less than the ideal case. However, the execution time for each case decreases as the number of the processors increase. We present Fig. 1 to depict the speed up and efficiency curve for each execution case of our experiment. This shows that the RSA algorithm is not by nature a classic parallel solution like the Matrix Multiplication problem [41]. As the number of processors are increased, the experimental speed up values seem to be moderately increased for our study. Similarly, the efficiency decreases as the number of processors increase.

However, the execution times for each case show that it is possible to improve the performance of the RSA encryption algorithm as we tend to get lower execution time by increasing the number of processors. We have been able to bring down the execution of RSA algorithm from 4.6 seconds with a single thread to 1 second with 8 threads. This proves a faster implementation of this algorithm is achieved and worth exploring further. Additionally, this shows how our approach effectively reduces the execution time through parallelization of the RSA algorithm.

V. Related Work

Pearson [30] examined portions of the RSA algorithm that would benefit from parallelization. His paper focused on parallelization of the encryption and decryption process by allowing each parallel element to handle a portion of the processing, increasing the overall throughput of the algorithm. Specifically, he found that the squaring operation. The squaring operation in the equation 1 and 2 could be performed simultaneously with other sequential processes such as multiplication. Other researchers found that several multiplication operations in the algorithm could be implemented with asymptotically faster multiplication algorithms such as FTT and Karatsuba [22] leading to improvements in efficiency. Imbert and Bajard [6] focused on implementation of an efficiency Montgomery Multiplication Algorithm based on the Reside Number System (RNS). This enabled them to obtain faster arithmetic operations in parallel. They created 2 types of RSA implementations to illustrate their approach. Additionally, their results were compared against those achieved by Kawamura et al. [20] and Posch and Posch [31] in terms of the number of elementary modular multiplications needed for their implementations of Montgomery multiplication. Li et al [24] proposed a variant to the traditional RSA implementation they called EAMRSA or Encrypt Assistant Multi-Prime RSA. This was based on a combination of the Multi-Prime RSA [12], [9] and RSA-S2 [27] proposed by previous researchers. This model is made up of three main components: key generation, encryption, and decryption. Key generation generate the private and public key pair with four parameters including a security-specific parameter. The encryption process and decryption process differ as the encryption process is done in two steps while the decryption process is performed with the Chinese Remainder Theorem [14]. This approach has also considered parallelization in terms of data and task decomposition. The data composition is performed with the private and public key as well as the plaintext and ciphertext. Task decomposition focuses on splitting the encryption and decryption processes across the parallel processing elements. Their results were compared against various sequential RSA implementations in terms of speed up.

Chiou [11] addressed the problem of modular exponentiation of the RSA algorithm. In his studay, he outlined a modular exponentiation approach based on the parallel binary method. This approach led to more efficiency in the encryption and decryption processes, leading to a roughly 33% process speed increase. He concluded that the technique could be merged with other parallelization techniques for larger improvements to processing speed. Wu et al. proposed a parallel exponentiation algorithm which was said to be faster than the efficient Savas-Tenca-Koc algorithm [34]. In the paper, they described 2 binary methods for the exponentiation operations (e.g. equation 1 and equation 2) in the RSA algorithm. These were the right to left binary method and the left to right binary method [37], [8]. Additionally, they detailed exponentiation techniques proposed by other researchers [18], [35], [33] and compared their performance to their newly proposed approach based on speed up, with results ranging from 1.06 to 2.75. They were also able to decrease the number of multiplication and squaring operations I norder to improve the efficiency of the exponentiation evaluation. A novel exponentiation technique was proposed by Sepahvandi et al. [36] in 2009 that performed the squaring and multiplication operations of the RSA algorithm in parallel leading to a 50% increase in speed. In their paper, they have shown the traditional method used for the modular exponentiation operation in RSA has been performed either with the binary method or a traditional arithmetic-based method [43], [38], [10], [37]. Their approach managed to split these operations up into parallel components that could be computed simultaneously, resulting in a minimal number of sequential steps needed to derive the ciphertext and plaintext. Damrudi and Ithnin created a method to improve the parallelization of the RSA algorithm through a tree based approach called TRSA [13].

This approach utilizes a binary tree based parallel processing architecture to split exponentiation operations into a number of nodes. They compared their results against Sepahvandi et al. [2009] [36], Montgomery [39], Bielecki and Burak [2007] [7], and CRT [23], demonstrating improvements in terms of speed up and reduced computational complexity. The speed up results were measured by comparing the serial version of the RSA algorithm against their parallel implementation utilizing the OpenMP library [4].

Other researchers sought special hardware-based solutions for improved RSA parallel processing speed. Liu et al. [25] proposed the use a systolic, array-based VLSI enabled architecture as the RSA crypto processor for the Montgomery Multiplication Algorithm as well as the square and multiplication portions of the algorithm. This architecture was designed as a two-stage access scheme along with a single backup scheme in order to eliminate the fanout bottleneck. They claimed thus architecture would be more efficient than past architecture implementations. In 2004, Tang et al. proposed the user of a Field Programmable Gate Array (FPGA), a reconfigurable hardware device for use in a semi-systolic implementation of the modular exponentiation unit as part of a parallel implementation of the RSA algorithm [40]. This hardware design addressed the time intensive issue of modular exponentiation operations in RSA. Their approach to serial-parallel multiplication utilized the left to right binary method to improve efficiency. The RSA processor itself was implemented with Very High-Speed Integrated Circuit Hardware Description Language (VHDL) using automatic Place-and-Route (PAR).

Researchers have also proposed implementations of RSA made to take advantage of the efficiency of multi-core Graphical Processing Units (GPUs) for parallel tasks. Fan et al. [16] presented an approach to RSA parallelization using JCUDA [42] and Hadoop [1]. Their method separated the algorithm into two levels: tread level, which is associated with the CUDA [2] framework and computer level. The researchers compared their experimental findings vs. CPU run times across 10 threads, 16 threads, 100 threads, and 500 threads with different plaintext character lengths. Their results showed a significant increase in speed up compared to CPU-based parallel implementations of the RSA algorithm. Fadhil and Younis [15] also focused their research on comparing GPU-based and CPU-based parallel RSA implementations. They were able to obtain a speed up factor of 23 for the GPU-based implementation vs the multi-thread CPUs speed up factor of 6 when compared against the same serial CPU implementation of the RSA algorithm.

VI. Discussion

The RSA algorithm is a public key encryption scheme that ensures better security than symmetric encryption algorithms; for instance, DES [26]. The algorithm is heavily used in the industry and adapted for various communication schemes. Due to its ever lasting popularity, different avenues for improving the performance of the RSA algorithm through parallelization have been discussed in this paper. Additionally, one of these avenues has been analyzed and implemented to illustrate the improved performance of this approach, strengthening the quality of the study.

The experiments were run in the High Performance Computing environment of Tennessee Technological University [5]. The high performance cluster provides all the necessary functionality to execute parallel programs using the OpenMP library, which was used to parallelize the RSA algorithm. Upon analysis, certain operations cannot be parallized in the RSA algorithm due to the dependency of one operation on another. The steps depicted in the Algorithm 1 are dependent upon previous actions in the sequence. However, the output values of the algorithm for a given input stream do not need to be recomputed. Thus, this gives us a portion of the program which will always remain sequential.

One of the limitations of this parallelized approach is that the program does not address parallelizing and / or optimizing the modular operations in Equation 1 and Equation 2. It is analyzed, this operation is also time intensive and worth implementing for parallel processing. One method to minimize this computational complexity would be incorporating the Chinese Remainder Theorem (CRT) [14]. Another limitation of the approach is that the study does not consider the underlying protocol of the RSA algorithm on which the keys will be shared. These limitations deviate the study from a real-life scenario of RSA encryption. However, the limitations do not affect the parallel performance and model of this encryption scheme which is the essence of this study.

VII. Conclusion

In our study, we present a comprehensive literature survey of parallelization of the RSA algorithm, a public key encryption scheme, from 1978 till date. The survey represents several avenues on which the algorithm could be parallelized as well as techniques or methods that allow sequential processes to run simultaneously on different processing elements, be it on Central Processing Unit (CPU) and / or Graphics Processing Unit (GPU). The survey should give researchers a foundation to understand the types of research available in this field of study and help develop strategies to further improve on parallelization of the RSA algorithm. Additionally, our research presents a parallel implementation of the RSA algorithm focused on one of the possible discussed avenues, which is the exponentiation operations. We analyze the exponentiation operations by partitioning them into individual processing elements in order to compute the encryption and decryption mathematical operations faster. This gives us improved performance in terms of execution time of the program compared with the serial implementation leading to high speed up and efficiency. The detailed comparisons and experimental findings are described in the Results section.

Our research also includes some limitations that are mentioned in the Discussion section which could be considered as possible future work. For example, our work could possibly be improved through analysis and evaluation of an implementation utilizing another high performance message passing computing library, such as, Message Passing Interface (MPI) [17]. Additionally, our focus was on developing CPU-based parallel processing tool. Focusing on GPU-based implementation, however, would be an another avenue for future work.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments and feedback. We would also like to extend our gratitude to Dr. Ambareen Siraj and Dr. Sheikh K. Ghafoor for the guidance and direction we have received throughout the research. The work reported in this paper has been fully supported by Cybersecurity Education, Research & Outreach Center (CEROC) at Tennessee Technological University.

References

[1] Apache hadoop — open-source software for reliable, scalable, distributed computing. https://hadoop.apache.org/. Accessed: April 8, 2019.

[2] Cuda by nvidia. https://developer.nvidia.com/cuda-zone. Accessed: April 5, 2019.

[3] The gnu multiple precision arithmetic library. https://gmplib.org. Accessed: April 6, 2019.

[4] Openmp — enabling hpc since 1997. https://www.openmp.org/. Accessed: April 5, 2019.

[5] Tennessee tech university's impulse high performance computing cluster. https://www.hpc.tntech.edu. Accessed: October 22, 2019.

[6] J-C Bajard and Laurent Imbert. A full rns implementation of rsa. *IEEE Transactions on Computers*, 53(6):769–774, 2004.

[7] Wńodzimierz Bielecki and Dariusz Burak. Parallelization method of encryption algorithms. In *Advances in Information Processing and Protection*, pages 191–204. Springer, 2007.

[8] Thomas Blum and Christof Paar. High-radix montgomery modular exponentiation on reconfigurable hardware. *IEEE transactions on computers*, 50(7):759–764, 2001.

[9] Dan Boneh and Hovav Shacham. Fast variants of rsa. *CryptoBytes*, 5(1):1–9, 2002.

[10] Ernest F Brickell. A survey of hardware implementations of rsa. In *Conference on the Theory and Application of Cryptology*, pages 368–370. Springer, 1989.

[11] Che Wun Chiou. Parallel implementation of the rsa public-key cryptosystem. *International Journal of Computer Mathematics*, 48(3-4):153–155, 1993.

[12] Thomas Collins, Dale Hopkins, Susan Langford, and Michael Sabin. Public key cryptographic apparatus and method, December 8 1998. US Patent 5,848,159.

[13] Masumeh Damrudi and Norafida Ithnin. Parallel rsa encryption based on tree architecture. *Journal of the Chinese Institute of Engineers*, 36(5):658–666, 2013.

[14] Pei Dingyi, Salomaa Arto, and Ding Cunsheng. *Chinese remainder theorem: applications in computing, coding, cryptography*. World Scientific, 1996.

[15] Heba Mohammed Fadhil and Mohammed Issam Younis. Parallelizing rsa algorithm on multicore cpu and gpu. *International Journal of Computer Applications*, 87(6), 2014.

[16] Wenjun Fan, Xudong Chen, and Xuefeng Li. Parallelization of rsa algorithm based on compute unified device architecture. In *2010 Ninth International Conference on Grid and Cloud Computing*, pages 174–178. IEEE, 2010.

[17] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[18] Shay Gueron and Or Zuk. Applications of the montgomery exponent. In *International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II*, volume 1, pages 620–625. IEEE, 2005.

[19] Dan Hu. Using rsa and aes for file encryption. https://www.codeproject.com/Tips/834977/Using-RSA-and-AES-for-File-Encryption, note=(Accessed on April 27, 2019), 2014.

[20] Shinichi Kawamura, Masanobu Koike, Fumihiko Sano, and Atsushi Shimbo. Cox-rower architecture for fast parallel montgomery multiplication. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 523–538. Springer, 2000.

[21] Sharon S Keller. The 186-4 rsa validation system (rsa2vs).

[22] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1997.

[23] Cetin Kaya Koc. High-speed rsa implementation version 2.0. *RSA Security*, 1994.

[24] Yunfei Li, Qing Liu, and Tong Li. Design and implementation of an improved rsa algorithm. In *2010 International Conference on E-Health Networking Digital Ecosystems and Technologies (EDT)*, volume 1, pages 390–393. IEEE, 2010.

[25] Qiang Liu, Fangzhen Ma, Dong Tong, and Xu Cheng. A regular parallel rsa processor. In *The 2004 47th Midwest Symposium on Circuits and Systems, 2004. MWSCAS'04.*, volume 3, pages iii–467. IEEE, 2004.

[26] Prerna Mahajan and Abhishek Sachdeva. A study of encryption algorithms aes, des and rsa for security. *Global Journal of Computer Science and Technology*, 2013.

[27] Tsutomu Matsumoto, Koki Kato, and Hideki Imai. Speeding up secret computations with insecure auxiliary devices. In *Conference on the Theory and Application of Cryptography*, pages 497–506. Springer, 1988.

[28] Bishop Matt et al. *Introduction to computer security*. Pearson Education India, 2006.

[29] Ivan Niven, Herbert S Zuckerman, and Hugh L Montgomery. *An introduction to the theory of numbers*. John Wiley & Sons, 2013.

[30] David Pearson. A parallel implementation of rsa. *Cornell University (July 1996)*, 1996.

[31] Karl C Posch and Reinhard Posch. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):449–454, 1995.

[32] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[33] Yasuyuki Sakai and Kouichi Sakurai. Simple power analysis on fast modular reduction with generalized mersenne prime for elliptic curve cryptosystems. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 89(1):231–237, 2006.

[34] Erkay Savas, Alexandre F Tenca, Çetin Kaya Koç, et al. A scalable and unified multiplier architecture for finite fields gf (p) and gf (2ˆ m). In *CHES*, pages 277–292, 2000.

[35] Katja Schmidt-Samoa, Olivier Semay, and Tsuyoshi Takagi. Analysis of fractional window recoding methods and their application to elliptic curve cryptosystems. *IEEE Transactions on Computers*, 55(1):48–57, 2006.

[36] S Sepahvandi, M Hosseinzadeh, K Navi, and A Jalali. An improved exponentiation algorithm for rsa cryptosystem. In *2009 International Conference on Research Challenges in Computer Science*, pages 128–132. IEEE, 2009.

[37] Mark Shand and Jean Vuillemin. Fast implementations of rsa cryptography. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pages 252–259. IEEE, 1993.

[38] Ajay C Shantilal. A faster hardware implementation of rsa algorithm. *Oregon State University, Corvallis, Oregon*, 97331, 1993.

[39] Nigel Paul Smart et al. *Cryptography: an introduction*, volume 3. McGraw-Hill New York, 2003.

[40] SH Tang, KS Tsui, and Philip Heng Wai Leong. Modular exponentiation using parallel multipliers. In *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT)(IEEE Cat. No. 03EX798)*, pages 52–59. IEEE, 2003.

[41] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[42] Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *European Conference on Parallel Processing*, pages 887–899. Springer, 2009.

[43] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sang-Jae Moon. Rsa speedup with chinese remainder theorem immune against hardware fault cryptanalysis. *IEEE Transactions on computers*, 52(4):461–472, 2003.