Analyzing the Time Complexity of RSA Encryption Cracking Using the Brute Force Method

Ikhwan Al Hakim - 13522147[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522147@std.stei.itb.ac.id

***Abstract*—This research paper investigates the time complexity of brute force attacks on RSA encryption, a pivotal public-key cryptographic algorithm widely used for securing digital communication. Focusing on the difficulty of factoring the product of large prime numbers, the study employs mathematical models and simulations to analyze the impact of key bit size on the feasibility of brute force decryption. The examination includes various key sizes, ranging from 2 bit to 20 bit.**

***Keywords*— Brute force, Prime numbers, RSA encryption, Time complexity.**

I. INTRODUCTION

RSA encryption is widely regarded as one of the most secure encryption algorithms in use today. Developed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977, RSA (Rivest-Shamir-Adleman) relies on the mathematical complexity of factoring the product of two large prime numbers, making it resistant to attacks by classical computers. The security of RSA encryption is based on the assumption that factoring the product of two large primes is a computationally infeasible task. The algorithm plays a crucial role in securing digital communication, including online transactions, data transmission, and secure communication protocols. Its strength lies in the difficulty of deducing the private key from the public key, ensuring a robust level of security for sensitive information.

While RSA encryption has long been considered highly secure, recent advancements in computing have raised concerns about its long-term resilience. The algorithm's strength relies on the difficulty of factoring the product of two large prime numbers, a task that becomes increasingly computationally intensive as the size of the primes grows. While it is currently deemed practically impossible for classical computers to efficiently factor the product within a reasonable timeframe, the rise of quantum computing poses a potential threat. Quantum computers, leveraging principles of quantum mechanics, have the potential to exponentially speed up certain calculations, including factoring large numbers. This development could, in theory, undermine the security of RSA encryption by efficiently solving the underlying mathematical problem.

For that reason, this paper is made to calculate what is the exact time complexity to brute force an RSA encryption. Brute force is a straightforward and exhaustive approach to problem-solving that relies on systematically trying all possible solutions until the correct one is found. It is a general problem-solving technique that can be applied to a wide range of problems, particularly in the fields of computer science, cryptography, and mathematics.

In the context of algorithms and computer science, a brute force algorithm typically involves checking all possible combinations or solutions to a problem without using any optimization or heuristics to narrow down the search space. While this method is simple and guarantees a solution, it can be highly inefficient, especially for problems with large solution spaces, as it requires evaluating a vast number of possibilities.

II. THEORETICAL BASIS

A. RSA Encryption

RSA (Rivest-Shamir-Adleman) is a widely used public-key cryptosystem that enables secure communication over an insecure channel. It was invented by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977 and remains one of the most highly secure encryption algorithms.

The RSA algorithm involves two keys: a public key and a private key. These keys are mathematically related but computationally infeasible to derive from one another. The public key can be freely distributed, while the private key must be kept secret.

These are the processes to generate an RSA key:

1. Choose two large prime numbers, namely p and q.
2. Compute the multiplication between them, let $n = p \times q$.
3. Calculate $\phi(n) = (p-1) \times (q-1)$, where ϕ is the Euler's totient function.
4. Choose a number e that is coprime with $\phi(n)$.
5. Compute a number d such that $de \equiv 1 \pmod{\phi(n)}$.

The public key is e and the private key is d.

B. Time Complexity

Time complexity is a key parameter used to evaluate the computational efficiency of algorithms. It allows researchers and practitioners to make informed decisions about algorithm selection based on the expected behavior of the algorithm as the input size increases. The notation employed for time complexity, commonly known as Big O notation, provides a concise representation of an algorithm's worst-case time complexity. By expressing the upper bound of an algorithm's running time in terms of a mathematical function, Big O notation facilitates a high-level understanding of an algorithm's scalability.

Time complexity is formally defined as the function $T(n)$, representing the maximum amount of computational time required by an algorithm for an input of size n. In Big O notation, denoted as $O(f(n))$, the focus is on characterizing the upper bound of $T(n)$ as n approaches infinity. The notation $O(f(n))$ implies that the running time of the algorithm does not grow faster than a constant multiple of $f(n)$ for sufficiently large n.

Below are the example of Big O notation and their respective graph mapping:

1. Constant Time ($O(1)$): The running time of the algorithm remains constant regardless of the input size. Simple operations like accessing an element in an array fall into this category.
2. Linear Time ($O(n)$): The running time of the algorithm grows linearly with the input size. Example of this category is iterating through an entire element of an array or a list.
3. Logarithmic Time ($O(\log n)$): The running time of the algorithm grows logarithmically with the input size.
4. Linearithmic Time ($O(n \log n)$): This time complexity is common in some efficient sorting algorithms like Merge Sort and Heap Sort.
5. Quadratic Time ($O(n^2)$): The running time is proportional to the square of the input size. This is often seen in nested loops.
6. Exponential Time ($O(k^n)$): The running time grows exponentially with the input size. Algorithms with this complexity are generally impractical for large inputs.
7. Factorial Time ($O(n!)$): This time complexity is relatively uncommon in practical algorithms due to its rapid growth rate. In most cases, factorial time complexity is associated with naive or brute-force algorithms that exhaustively generate all possible permutations or combinations of a set.

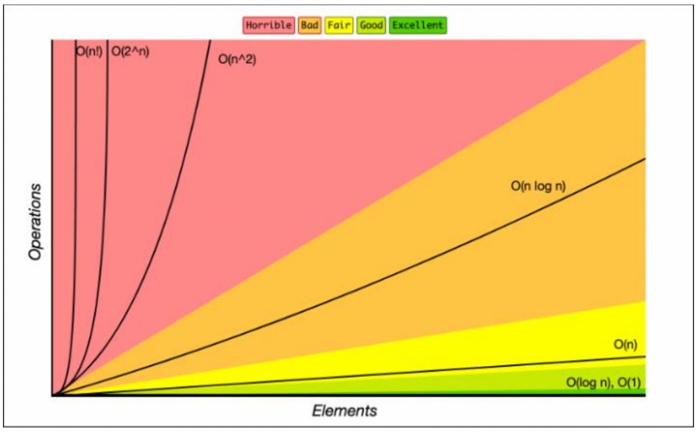


Figure 1. Time Complexity Graph
Source: https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/

C. *Brute Force*

Brute force is a deterministic and exhaustive technique utilized across diverse domains to systematically explore and evaluate all possible options until the correct solution is discovered. One prominent application of brute force is in the realm of cybersecurity, particularly in password cracking attempts. In this context, attackers employ brute force to systematically test every conceivable password combination in an effort to gain unauthorized access to a system. The success of such attacks hinges on factors like the strength of passwords and the computational resources available to the attacker. While brute force might be a less sophisticated approach compared to more intricate hacking techniques, it can prove effective, especially when dealing with weak or easily guessable passwords.

Cryptography also frequently encounters brute force, particularly in decryption attempts. In cryptographic systems, a brute force attack involves systematically trying all possible decryption keys until the correct one is found. The efficacy of this method depends on the encryption algorithm's complexity and the length of the encryption key. Robust encryption algorithms with longer keys significantly raise the difficulty level for successfully executing a brute force attack, rendering them more resilient against unauthorized decryption attempts. Cryptographers continually strive to develop encryption methods that withstand brute force attacks, emphasizing the importance of key length and algorithmic intricacy in ensuring data security.

While brute force methods offer a straightforward and conceptually simple approach to problem-solving, they are not always the most efficient or practical solutions. In certain scenarios, the solution space may be vast, leading to a time-consuming and resource-intensive process. In computer science, brute force algorithms may be employed to solve problems by systematically considering all potential solutions. However, more optimized algorithms, leveraging specific characteristics of the problem, are often favored for their efficiency, particularly in large-scale or complex problem domains. Thus, while brute force remains a viable strategy in certain contexts, its limitations underscore the importance of developing and implementing more sophisticated methods to tackle intricate challenges effectively.

III. Methodology

Before the author explain any of the method used in this research, it should be noted that the experiment is done in the following hardware specifications:

- Processor: AMD Ryzen 7 5800HS 3.2GHz
- RAM: 16 GB

A. Generating the Prime Numbers

First, there is the need to define the bit size of the prime numbers. Because the experiment is done in a small environment and a not-so-good hardware, the author only able to generate the prime numbers in the size of 2, 4, 8, 10, 12, 14, 16, 18, 20, and 22 bit. The author have tried the size beyond that and it just doesn't run at all because this prime number generator have the time complexity of $O(2^n)$. But in this research, this fact will be ignored because we only do the observation on the RSA key cracking.

The author have made the code to generate prime numbers in various bit sizes. Below is the implementation of the code:

```c
#define boolean unsigned char
#define true 1
#define false 0

#include <stdio.h>
#include <math.h>
#include “dinlist.h”

unsigned long long int power(unsigned long long int x, unsigned long long int y) {
    unsigned long long int temp;
    if (y == 0)
        return 1;
    temp = power(x, y / 2);
    if (y % 2 == 0)
        return temp * temp;
    else
        return x * temp * temp;
}

int main() {
    unsigned long long int maxPrime;
    boolean prime;
    dinList primes;
    primes.list = malloc(sizeof(unsigned long long int));
    primes.size = 0;

    printf(“Enter the prime number bit size: “);
    scanf(“%lld”, &maxPrime);
    while (maxPrime >= 64) {
        printf(“Please enter below 64 bit (it goes beyond C’s capability)\n”);
        printf(“Enter the prime number bit size: “);
        scanf(“%lld”, &maxPrime);
    }
    maxPrime = power(2, maxPrime);

    for (int i = 2; i < maxPrime; i++) {
        prime = true;
        if (i > 2) {
            for (int j = 2; j < i; j++) {
                if (i % j == 0) {
                    prime = false;
                    break;
                }
            }
        }
        if (prime) {
            append(&primes, i);
        }
    }

    FILE *fptr;
    fptr = fopen(“prime.txt”, “w”);
    for (int i = 0; i < primes.size; i++) {
        fprintf(fptr, “%lld “, primes.list[i]);
    }
    fclose(fptr);
}
```

Firstly, the code will ask the user about the max bit size of the prime. After entering the size, then the code will generate prime numbers from the smallest prime up to the highest prime in that bit size range. Note that the author limits the size in the 64 bit mark because the maximum size of an *unsigned long long* integers in *C* is $2^{64} - 1$. After that, the code will store them in a dynamically-arranged list and write them into a textfile called *prime.txt*. Below is the header file and the implementation file of the dynamically-arranged list:

```c
#ifndef DINLIST_H
#define DINLIST_H

#include <stdio.h>
#include <stdlib.h>

typedef struct dinList {
  long long int size;
  long long int* list;
} dinList;

void append(dinList *arr, long long int value);

#endif
```

```c
#include "dinlist.h"

void append(dinList *arr, long long int value) {
  long long int *new_ptr = realloc(arr->list, sizeof *(arr->list) * (arr->size + 1u));
  if (new_ptr == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit (EXIT_FAILURE);
  }
  arr->list = new_ptr;
  arr->list[arr->size] = value;
  arr->size++;
}
```

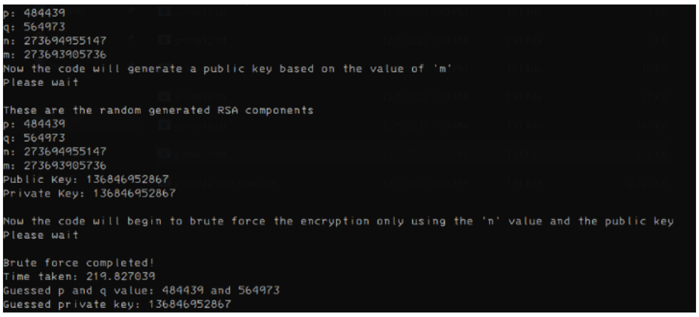
B. Generating and Predicting the RSA Key

Now that we already have the prime numbers, the next step is to generate the RSA key. This process follow the exact steps that is mentioned before. The author have already convert those steps into a code and below is the implementation of it:

```c
#define boolean unsigned char
#define true 1
#define false 0

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "dinlist.h"

int ctoi(char letter) {
    if (letter == '0') {
        return 0;
    }
    else if (letter == '1') {
        return 1;
    }
    else if (letter == '2') {
        return 2;
    }
    else if (letter == '3') {
        return 3;
    }
    else if (letter == '4') {
        return 4;
    }
    else if (letter == '5') {
        return 5;
    }
    else if (letter == '6') {
        return 6;
    }
    else if (letter == '7') {
        return 7;
    }
    else if (letter == '8') {
        return 8;
    }
    else if (letter == '9') {
        return 9;
    }
}

unsigned long long int gcdExtended(unsigned long long int a, unsigned long long int b, unsigned
long long int* x, unsigned long long int* y) {
    if (a == 0) {
        *x = 0, *y = 1;
        return b;
    }
    unsigned long long int x1, y1;
    unsigned long long int gcd = gcdExtended(b % a, a, &x1, &y1);
    *x = y1 - (b / a) * x1;
    *y = x1;
    return gcd;
}

unsigned long long int modInverse(unsigned long long int ra, unsigned long long int rb) {
    srand(time(NULL));
    unsigned long long int rc, sa = 1, sb = 0, sc, i = 0;
    if (rb > 1) do {
            rc = ra % rb;
            sc = sa - (ra / rb) * sb;
            sa = sb, sb = sc;
            ra = rb, rb = rc;
        } while (++i, rc);
    sa *= (i *= ra == 1) != 0;
    sa += (i & 1) * sb;
    return sa;
}

boolean coprime(unsigned long long int num1, unsigned long long int num2) {
    unsigned long long int min, count;
    boolean flag = true;
    min = num1 < num2 ? num1 : num2;
    for(count = 2; count <= min; count++) {
        if(num1 % count == 0 && num2 % count == 0) {
            flag = false;
            break;
        }
    }
    return flag;
}

unsigned long long int find_coprime(unsigned long long int m) {
    for (unsigned long long int a = m/2; a > 2; a--) {
        if (coprime(a, m)) {
            return a;
        }
    }
}

int main() {
    FILE *fptr;
    dinList primes;
    primes.list = malloc(sizeof(unsigned long long int));
    primes.size = 0;
    char ch;
    unsigned long long int temp = 0;
    fptr = fopen("prime.txt", "r");
    do {
        ch = fgetc(fptr);
        if (ch != ' ') {
            temp = temp*10 + ctoi(ch);
        }
        else {
            append(&primes, temp);
            temp = 0;
        }
    } while (ch != EOF);
    fclose(fptr);

    srand(time(NULL));
    unsigned long long int p = primes.list[rand() % primes.size];
    unsigned long long int q = primes.list[rand() % primes.size];
    unsigned long long int n = (unsigned long long int) (p * q);
    unsigned long long int m = (unsigned long long int) ((p - 1) * (q - 1));
    printf("p: %lld\n", p);
    printf("q: %lld\n", q);
    printf("n: %lld\n", n);
    printf("m: %lld\n", m);
    printf("Now the code will generate a public key based on the value of \'m\'\n");
    printf("Please wait\n\n");

    unsigned long long int publicKey = find_coprime(m);

    unsigned long long int privateKey = modInverse(publicKey, m);

    printf("These are the random generated RSA components\n");
    printf("p: %lld\n", p);
    printf("q: %lld\n", q);
    printf("n: %lld\n", n);
    printf("m: %lld\n", m);
    printf("Public Key: %lld\n", publicKey);
    printf("Private Key: %lld\n\n", privateKey);

    printf("Now the code will begin to brute force the encryption only using the \'n\' value
and the public key\n");
    printf("Please wait\n\n");

    boolean found = false;
    unsigned long long int i = 0, j;
    unsigned long long int privateKeyGuess;

    clock_t start = clock();
    while (i < primes.size && !found) {
        j = 0;
        while (j < primes.size && !found) {
            if ((unsigned long long int) (primes.list[i] * primes.list[j]) == n) {
                unsigned long long int mGuess = (unsigned long long int) ((primes.list[i] -
1) * (primes.list[j] - 1));
                if (find_coprime(mGuess) == publicKey) {
                    privateKeyGuess = modInverse(publicKey, mGuess);
                    found = true;
                }
            }
            j++;
        }
        i++;
    }
    clock_t stop = clock();
    double timeTaken = (double)(stop - start) / CLOCKS_PER_SEC;

    i--;
    j--;

    printf("Brute force completed!\n");
    printf("Time taken: %f\n", timeTaken);
    printf("Guessed p and q value: %lld and %lld\n", primes.list[i], primes.list[j]);
    printf("Guessed private key: %lld\n", privateKeyGuess);
}
```

First, the code reads the *prime.txt* that is generated before using the previous code and convert them into integers with the *ctoi* function. Then the code will store them into a dynamically-arranged list. After that, two random numbers from 0 upto the length of the list will be generated and the code will use those two numbers as indices to read from the list of prime numbers and store the result in variables, namely *p* and *q*. After that, the code will compute the multiplication of those two numbers and store it in the *n* variable. Then, it will calculate the Euler's totient function of *n* and store it in the *m* variable. Next, the code will search for a number that is coprime with *m* and declare it as the public key and lastly, it will find a number that when multiplied with the public key, the result will be congruent to 1 modulo *m* and declare it as the private key.

Now that we have the randomly generated RSA components, it is time to continue to the last process and that is predicting the private key only using the *n* value and the public key. In this process, the author use the list of prime numbers that is generated before and iterate through them until the right combination of two prime numbers is found. Then it will display the guessed private key and the time taken to guess it as shown below in the following figure:



```
p: 484439
q: 564973
n: 273694955147
m: 273693905736
Now the code will generate a public key based on the value of 'm'
Please wait

These are the random generated RSA components
p: 484439
q: 564973
n: 273694955147
m: 273693905736
Public Key: 136846952867
Private Key: 136846952867

Now the code will begin to brute force the encryption only using the 'n' value and the public key
Please wait

Brute force completed!
Time taken: 219.827039
Guessed p and q value: 484439 and 564973
Guessed private key: 136846952867
```

Figure 2. The result of guessing RSA private key on 20 bit sized prime numbers
Source: author's documentation

IV. RESULT

The author will only calculate the time complexity of the RSA cracking in the size of 2, 4, 8, 10, 12, 14, 16, 18, and 20 bit. To increase the accuracy of the outcome, five measurement were taken for each size then the author compare the average time taken. This is the mapping of the output in the form of table and graph:

Table 1. Time taken to guess RSA private key with brute force for each prime number size

Size (bit)	Time taken (second)					Average (second)
	First	Second	Third	Fourth	Fifth	
2	2×10^{-6}	3×10^{-6}	2×10^{-6}	3×10^{-6}	2×10^{-6}	2.4×10^{-6}
4	3×10^{-6}	2×10^{-6}	3×10^{-6}	2×10^{-6}	2×10^{-6}	2.4×10^{-6}
6	4×10^{-6}	3×10^{-6}	6×10^{-6}	6×10^{-6}	6×10^{-6}	5×10^{-6}
8	3.3×10^{-5}	2.1×10^{-5}	4.7×10^{-5}	2.3×10^{-5}	3.4×10^{-5}	3.16×10^{-5}
10	8.43×10^{-4}	3.15×10^{-4}	2.64×10^{-4}	2.78×10^{-4}	8.68×10^{-4}	5.14×10^{-4}
12	9.32×10^{-3}	7.76×10^{-3}	1.1×10^{-2}	7.09×10^{-3}	1.13×10^{-2}	9.29×10^{-3}
14	7.22×10^{-2}	6.06×10^{-2}	7.82×10^{-2}	1.4×10^{-1}	8.95×10^{-2}	8.81×10^{-2}
16	2.02	1.7	7.52×10^{-1}	1.64	1.44	1.51
18	14.36	17.72	29.59	21.59	44.76	25.6
20	219.83	307.86	437.94	223.53	311.04	300.04

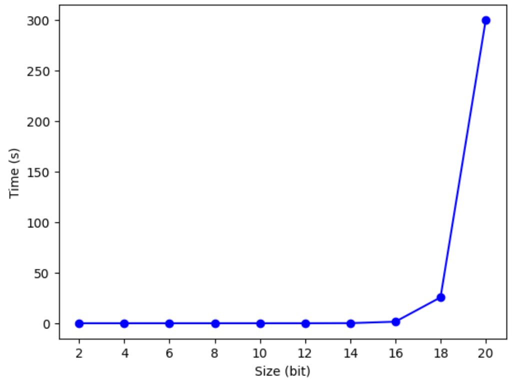


Figure 3. Graph between the maximum size of the number in bit and the time taken to guess the private key with brute force
Source: author's documentation

If the observation is done directly from the table and the graph, there isn't much information to extract because apparently *C* language is not really good at capturing the time taken to compute something really quick. Which is why the time taken for the smaller bit do not differ from each other. However on the bigger side of the bit, the time taken changes significantly. Then after 20 bit, the time it takes is longer than 6 hour because the author have tried leave the code running overnight and the private key was still not found in the next morning.

Because the information from the table and the graph isn't enough to conclude the time complexity, the inspection will be done on the code. If we look at the code, there are two loops in the section where the code start to guess the two large prime numbers.

```
    clock_t start = clock();
    while (i < primes.size && !found) {
        j = 0;
        while (j < primes.size && !found) {
            if ((unsigned long long int) (primes.list[i] * primes.list[j]) == n) {
                unsigned long long int mGuess = (unsigned long long int) ((primes.list[i] -
1) * (primes.list[j] - 1));
                if (find_coprime(mGuess) == publicKey) {
                    privateKeyGuess = modInverse(publicKey, mGuess);
                    found = true;
                }
            }
            j++;
        }
        i++;
    }
    clock_t stop = clock();
    double timeTaken = (double)(stop - start) / CLOCKS_PER_SEC;
```

Those two loops is used to iterate the list of prime numbers and multiply them until the right combination of prime number is found. So in this section, the time complexity is $O(n^2)$ where$n$ is the amount of prime numbers on that particular bit size. After that, if the right two prime numbers is already found, it will calculate the Euler's totient function of the multiplication between those two numebrs and it will continue to search for the coprime of it using the *find_coprime* function and that is another $O(n^2)$. Lastly, if the coprime is finally found, it will compute the inverse modulo of the coprime with the extended Euclidean algorithm with the time complexity of $O(\log n)$.

Now that every section is already covered, it is time to compute the total time complexity with the following equation:

$$T(n) = n^2 \times n^2 + n \log n$$
$$T(n) = n^4 + n \log n = O(n^4)$$

We have found out that the time complexity of cracking an RSA private key is $O(n^4)$, where n is the amount of prime numbers in that bit range. However, we want the time complexity with respect to the bit size of the number. So we have to calculate how many prime number there are with respect to the bit size.

Table 2. The amount of prime numbers according to their bit size

Size (bit)	Amount of prime numbers
2	2
4	6
6	18
8	54
10	172
12	564
14	1900
16	6542
18	23000
20	82025

Based on the table, a conclusion can be made that for each two bit increase in size, the amount of prime numbers will increase approximately three times as before. With that being said, we can write the connection between n, the amount of prime numbers in the given bit size, and b, the bit size.

$$n(b) \approx 2 \times 3^b$$

Next, plug the value of $n(b)$ into $O(n^4)$.

$$O(n^4) = O((2 \times 3^b)^4) = O(1296^b)$$

So there it is, our final answer. The time complexity it takes to crack an RSA private key using brute force method is $O(k^n)$ where k is a constant and n is the maximum bit size of the number.

V. Conclusion

From the research that we did, we got the exact time complexity of guessing RSA private key with brute force and that is a stupendous $O(k^n)$ with n being the maximum bit size of the prime numbers. However, the brute force method that is used in this research is a pure brute force without any optimization. So $O(k^n)$ is the highest time complexity possible and could get lower if the the cracking is done with better performance enhancement.

VI. Suggestion

The author's suggestion for future researchers that want to continue this topic is to do this research in a more high-end hardware so that the code can calculate the time taken for prime numbers with the size of more that 20 bit because this is a really performance-taxing observation.

VII. Acknowledgment

The author extends heartfelt gratitude to Allah, the Most Merciful and Compassionate, for providing the strength, wisdom, and perseverance to undertake this scholarly journey. Acknowledgment is due to the esteemed lecturer, Dr. Ir. Rinaldi Munir, M.T. and Monterico Adrian, S.T., M.T., whose invaluable guidance, insightful feedback, and unwavering support have profoundly shaped the quality of this work. The author is profoundly grateful to their family for the unwavering love and encouragement that served as pillars of strength, and to friends, whose inspiration and camaraderie enriched the entire process. The author recognizes and appreciates the collective contributions, big and small, from everyone involved, and hopes that this work, guided by divine grace, contributes positively to the broader realm of knowledge.

References

[1] Munir, Rinaldi. (2020). "Teori Bilangan (Bagian 3)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/16-Teori-Bilangan-Bagian3-2023.pdf (accessed on 29th November 2023)

[2] C. Sun, "Comparative Study of RSA Encryption and Quantum Encryption," in *Theoretical and Natural Science*, vol. 2, no. 1, pp. 121–125, Feb. 2023. [Online]. Available: https://tns.ewapublishing.org/media/1d174a4cfbe645db8ed7cd81d1948ed3_A8qx0UX.pdf (accessed on 30th November 2023)

[3] T. Nguyen, "Privacy preserving using extended euclidean – algorithm applied to RSA," in *Tạp chí Khoa học Đại học Sư phạm Thành phố Hồ Chí Minh*, vol. 1, no. 2, pp. 53-63, Apr. 2023. [Online]. Available: https://sj.hpu2.edu.vn/index.php/journal/article/download/185/124 (accessed on 30th November 2023)

[4] M. M. Sudershan, S. V. Pydakula, and A. Priya, "File Encryption and Decryption using AES and RSA Algorithm," in *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 10, no. 2, pp. 11-14, Mar. 2023. [Online]. Available:

https://ijsrcseit.com/paper/CSEIT2390130.pdf (accessed on 2nd December 2023)

ATTACHMENT

Source code used in this research and the demo video: https://github.com/Nerggg/RSA-Crack-Simulation

STATEMENT

I, the individual signing below, affirm that the content presented in this document is an original creation authored by me. It is not a derivative work, translation of another document, or a product of plagiarism.

Bandung, 9th December 2023

Ikhwan Al Hakim, 13522147