

Extension of RSA algorithm

Abstract

In my first paper on the expansion to Dirichlet's theorem, the "Exemption Rule" allowed for us to identify which terms in an arithmetic progression described by $an+b$, where $\gcd(a,b)=1$, would be prime. In the RSA algorithm, we require semi-prime numbers, which are products of two prime numbers p and q , where p and q can be either distinct or identical. The exemption rule allows us to generate n -values that would lead $an+b$ to be of the form $P(2k+1)$, a product of two odd numbers, and the exemption rule ensures that the two numbers are prime. To further enhance the security of the algorithm, at the end of the process, we can implement dynamic mapping functions which adds another layer of complexity, and with machine learning capabilities, create a modern enigma machine, in which the configurations are changing periodically.

Introduction

Cryptography is the process of concealing a message so that only the intended recipient can read it, while anyone else attempting to decipher it would encounter random letters, numbers, and symbols. In the modern era, cryptography is integral to various aspects of our lives, from securing transactions with bank cards to sending private messages to loved ones.

Among the many cryptographic tools available, the RSA (Rivest-Shamir-Adleman) algorithm is the most widely used public-key algorithm. RSA's security is based on the computational difficulty of factoring large semi-prime numbers, which are products of two prime numbers. In mathematics, problems involving prime numbers are notoriously challenging, especially those requiring efficient factorization of large numbers. Enhancing the robustness of the RSA algorithm is crucial to maintaining the security of our digital lifestyle.

In number theory, Dirichlet's theorem states that for any arithmetic progression $an+b$, where $\gcd(a,b)=1$, there are infinitely many prime numbers. Building on this theorem, the Exemption Rule allows us to identify which terms in the arithmetic progression are prime and which are composite. The composite terms can be expressed as $an+b=P(2k+1)$, where P is a prime number, and $2k+1$ is an odd number for all positive integers k . Applying the Exemption Rule to the sequence $2k+1$ ensures that it is also prime. Therefore, the composite term is a product of two prime numbers, forming the semi-prime numbers needed for the RSA algorithm.

The primary goal of this research is to enhance the efficiency and security of the RSA algorithm. By utilising the Exemption Rule, we aim to simplify the prime generation process. Additionally, we introduce dynamic mapping functions at the final stage of RSA encryption, adding a layer of complexity. These functions, combined with machine learning capabilities, create a modern enigma machine with periodically changing configurations, significantly bolstering cryptographic security.

Our contributions include the novel application of the Exemption Rule for prime generation in RSA and the implementation of dynamic mapping functions to enhance security. The results of this research have the potential to significantly improve cryptographic practices, providing a robust and adaptable solution to emerging security threats.

Methodology

Exemption Rule: The Exemption rule states that let the positive integer n in the AP of $an+b$ be:

$\frac{2Pk}{a} + \frac{P-b}{a}$, for an odd prime P and any positive integer k .

Derivation of Exemption Rule:

The derivation of Exemption Rule is basically reverse engineering method where we set $an+b$ equal $P(2k+1)$, and make n the subject of the statement.

$$\begin{aligned} an+b &= P(2k+1) \\ an+b &= 2Pk+P \\ an &= (2Pk)+(P-b) \\ n &= \frac{2Pk}{a} + \frac{P-b}{a} \end{aligned}$$

NOTE: For some Dirchlet's Theorem APs, they do not contain all the odd prime numbers, i.e, $4n+1$, however, we can still apply the Exemption Rule, if we have prior knowledge on what are the missing prime numbers in the AP. On the other hand, APs like $2n+1$ and $2n+3$ contain all the odd prime numbers, hence, we do not have any of the odd prime numbers, and generate the prime numbers progressively, along with their Exemption Rule.

We can now explore how the exemption rule can be applied to the RSA algorithm;

It begins with the calculation of the semi-prime $X = P \times Q$, where P and Q are primes. To compute X , there are two ways to go about using our exemption rule;

- a) First way is to have the AP $an+b$ and generate its exemption rule. Use the exemption rule to find an exempted n -value. This n -value would lead $an+b$ to be of the form $P(2k+1)$, since $2k+1$ is another AP we can use the exemption rule with, we would generate its exemption rule, and find an unexempted n -value, so $2k+1$ is another prime Q .

- b) Second way is to use one AP $an+b$, but we use its exemption rule to find two unexempted n -values, n_1 and n_2 , such that $P=an_1+b$ and $Q=an_2+b$.

Depending on the code written, one method may be more efficient than the other. Note that $an+b$ is an AP described by Dirichlet's theorem.

Afterwards, we can follow the remaining steps of the RSA algorithm such as public and private key generations, key distribution and encryption of plaintext M to a padded plaintext m , which then by Euler's totient function gives the ciphertext c . More information on RSA algorithm can be found [here](#). At the end of the process, we will get a ciphertext c , a numerical value, that we would use in the dynamic mapping functions of our algorithm.

Dynamic Mapping functions

Once we have attained the ciphertext c , we can map it to an alphabet using various functions. One such mapping would be assigning values to the alphabets based on their position and modular arithmetic.

Since there are 26 alphabets in English, we would work the mapping of numbers to letters using mod26. So,

$$\begin{aligned} A &\equiv 1 \pmod{26} \\ B &\equiv 2 \pmod{26} \\ &\vdots \\ Y &\equiv 25 \pmod{26} \\ Z &\equiv 26 \equiv 0 \pmod{26} \end{aligned}$$

Since the process so far has already been quite heavy in computation, we can keep the mapping of each number simple.

There are a total of 10 digits in base 10: $\{0,1,2,3,4,5,6,7,8,9\}$.

So, to keep the mapping simple, we just need to use 10 alphabets to map each digit of a number: $\{Z,A,B,C,D,E,F,G,H,I\}$.

We can call the elements in the numerical set γ and the elements in the alphabet set θ .

With linear mapping like $\gamma=\theta$ as our basic function, we have the following inputs and outputs;

| γ (ciphertext c) | θ (alphabetical ciphertext c) |
|----------------------------|-----------------------------------------|
| 0 | Z |

| | |
|-------|-------|
| 12 | AB |
| 304 | CZD |
| 99 | II |
| 40023 | DZZBC |

After the initial mapping we can create other functions like

$$\theta = \gamma^2:$$

| γ | γ^2 | θ |
|----------|------------|----------|
| 12 | 144 | ADD |
| 0 | 0 | Z |
| 000 | 000 | ZZZ |
| 132 | 17424 | AGDBD |

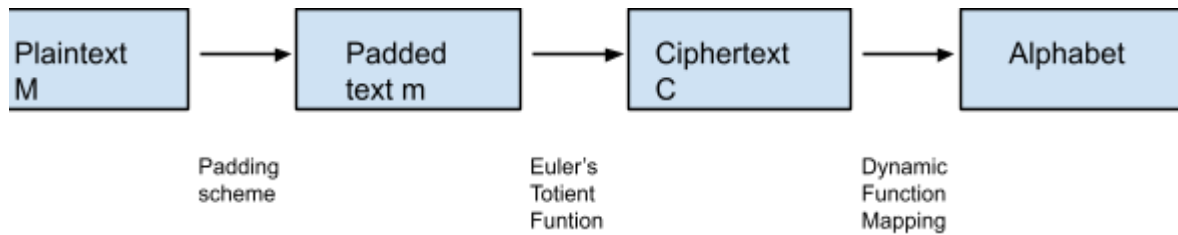
As such we can create more functions, which we then implement into our encryption model.

What makes the feature dynamic is that, just like how the enigma machine was re-configured everyday during world war 2, we can periodically change what function is being used in the encryption process.

As for the decryption process, we can write the inverse of the function that is being used to get back the original ciphertext, which can then be decrypted by one's private key to get the original padded plaintext, and eventually the actual plaintext.

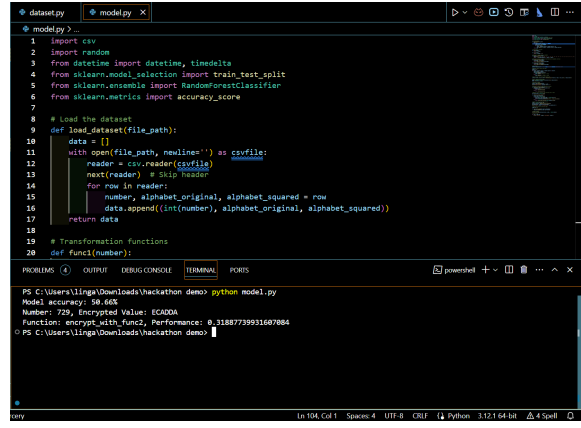
We can implement machine learning/AI modelling for the dynamic mapping functions so that the system is able to identify which function is more efficient for encryption, and also change the periodically changing pattern to random changes (but the model is able to accurately identify which function, thus, inverse function needed to be used). Especially during cyberattacks on the system, the code is able to prevent the attackers from gaining the method to crack the encryption by changing the function used.

In conclusion, already being computationally difficult, this enhanced RSA algorithm is also more complex to crack due to the additional encryption scheme with the dynamic mapping functions model.



Results

| Code | Analysis |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Semiprime.py: Uses the Exemption Rule method to generate prime numbers in a given time period, and uses the prime numbers to generate a list of semi-prime numbers in semiprime.txt</p> | <p>For demo test: time limit was set to 10s.</p> <p>Number of primes: 9890 Time taken to generate semiprimes: 35.19s Total number of semiprimes: 48910995</p> <pre> Found prime: 185421 Found prime: 185423 Found prime: 185451 Found prime: 185457 Found prime: 185472 Found prime: 185483 Found prime: 185511 Total primes generated: 9890 Time taken to generate semiprimes: 35.19s Total semiprimes generated: 48910995 Semiprimes written to 'semiprime.txt' </pre> |
| <p>dataset.py: Used the two example functions for the generation of 1,000,000 data, ranging from 0 to 999,999, and created a csv file and map these numbers to their respective letters for each function.</p> <p>model.py: Uses the dataset from dataset.py to train an encryption model that would periodically change which function is being used, and print the result.</p> | <p>Average accuracy of the model: 50%</p> <p>Note: The code is at its early stages, and with further optimizations, we can achieve higher accuracy.</p> <pre> # dataset.py def select_best_function(number): prediction = trained_model.predict([feature_vector])[0] return encrypt_with_func1 if prediction == 1 else encrypt_with_func2 # Adaptive function selector that alternates every 2 hours last_switch_time = datetime.now() def adaptive_function_selector(): global last_switch_time current_time = datetime.now() number = random.randint(0, 9999) # or any range appropriate to your data if current_time - last_switch_time > timedelta(hours=2): last_switch_time = current_time return random.choice([encrypt_with_func1, encrypt_with_func2]) else: return select_best_function(number) # Example of using the selected function number = random.randint(0, 999) selected_function = adaptive_function_selector() # model.py Model accuracy: 50.03% Number: 937, Encrypted Value: HGOIFI Function: encrypt_with_func2, Performance: 0.05219957596383196 PS C:\Users\llinga\Downloads\hackathon demo> python model.py Model accuracy: 50.05% Number: 978, Encrypted Value: ZIFQD Function: encrypt_with_func2, Performance: 0.885670288667317 PS C:\Users\llinga\Downloads\hackathon demo> python model.py Model accuracy: 49.94% Number: 525, Encrypted Value: HEEFBE Function: encrypt_with_func2, Performance: 0.2832167889836836 PS C:\Users\llinga\Downloads\hackathon demo> </pre> |

| | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| |  <pre> 1 import csv 2 import random 3 from datetime import datetime, timedelta 4 from sklearn.model_selection import train_test_split 5 from sklearn.ensemble import RandomForestClassifier 6 from sklearn.metrics import accuracy_score 7 8 # Load the dataset 9 def load_dataset(file_path): 10 data = [] 11 with open(file_path, newline='') as csvfile: 12 reader = csv.reader(csvfile) 13 next(reader) # Skip header 14 for row in reader: 15 number, alphabet_original, alphabet_squared = row 16 data.append((int(number), alphabet_original, alphabet_squared)) 17 return data 18 19 # Transformation functions 20 def func1(number): </pre> <p>Model accuracy: 0.465 Number: 729, Encrypted Value: ECADA Function: encrypt_with_func2, Performance: 0.31887739931607884</p> |
| <p>main.py: The main component of the project. This code binds the other three codes together for a functioning enhanced RSA algorithm with dynamic mapping function, and an algorithm to generate semiprimes.</p> | <p>-</p> |

The codes can be reviewed here: <https://github.com/RedBOyC/Enhanced-RSA/tree/main>