CRYPTOGRAPGHY ASSIGNMENT

IT Number	Name	Contribution
IT22893116	J M D U Umayanga	Implementation and Initial Testing
IT22005526	M H Rathnayake	Implementation and Initial Testing
IT22346490	M T H K Samarasinghe	Testing and Performance Analysis
IT22199508	Athapaththu A M M I P	Algorithm Selection and Research
IT22298508	Wanasinghe W M K R	Algorithm Selection and Research

Overview of AES (Advanced Encryption Standard) and RSA (Rivest–Shamir–Adleman)

AES is a contemporary, highly effective standard for encryption used in various sectors, ranging from secure communication to data storage. Covered weaknesses consist of side-channel attacks and insufficient key management. In terms of performance, the overhead is negligible even when dealing with enormous datasets making it ideal for high-performance settings.

RSA is one of the well-known public-key encryption techniques that is often employed in secure communication and signing documents electronically.
Weaknesses: Download issues on small key sizes, sophisticated attacks such as timing attacks.
In particular, RSA's performance may further limit its applicability, especially for large key sizes or large amounts of data even though RSA is very efficient when there is a need for secure key exchange or digital signature.

AES (Advanced Encryption Standard)

- **History**

  Joan Daemen and Vincent Rijmen, two Belgian cryptographers, work with this. It came out successful in a five-year long competition of 7 final contenders among which was even the strong algorithm RC6 and was chosen by the National Institute of Standards and Technology in the year 2001.

- **Design Principles**

  AES is a block cipher that has a fixed block size of 128 bits and supports key sizes of 128, 192, and 256 bits. the algorithm is designed as a substitution-permutation network, utilizing multiple rounds of processing depending on the size of the key.

- **Common Use Cases**

  It is extensively employed in services like VPNs, secure communication programs, and disk manager applications like BitLocker. It also serves as the benchmark in both government agencies and corporate organizations for the storage as well as the transmission of information in a safe manner.

- **Vulnerabilities**

  The system sounds good on paper, however, its practical execution could pose risks in operations against side-channel attacks, where an adversary relies on ancillary information emitted during the encryption process, for example timing attacks. Likewise, inept management of cryptographic keys may put at risk information that has been encrypted.

- **Performance**

AES is impressively swift in both hardware and software implementation, thus making it highly ideal for performance-critical applications. While there is an increase in computational load as the key size increases. However, it remains fairly efficient.

RSA (Rivest–Shamir–Adleman)

- **History**

  Introduced in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman. It was one of the first public-key cryptosystems, and it remains highly influential in cryptography.

- **Design Principles**

  The RSA algorithm is rooted in the principles of number theory, specifically the task of factorizing enormous numbers. It employs a couple of keys, a public key for encoding a message, and a corresponding private key for restoring the original content. The conventional sizes of RSA keys also vary but range from 1024 bits to 4096 bits.

- **Common Use Cases**

  There are numerous areas where RSA is used most popularly in the use of digital certificates like SSL/TLS, Secure email communication which involves PGP, digital signatures aimed at confirming their authenticity.

- **Vulnerabilities**

  RSA can equally be susceptible to brute-force attacks as long due to the fact that the key sizes are small enough especially for instance 1024-bit RSA which is now regarded as being insecure. One must also pay attention during key generation in order to avoid the generation of weak keys.

  Timing attacks and other side channel attacks may also pose a danger given that RSA has improper implementation.

- **Performance**

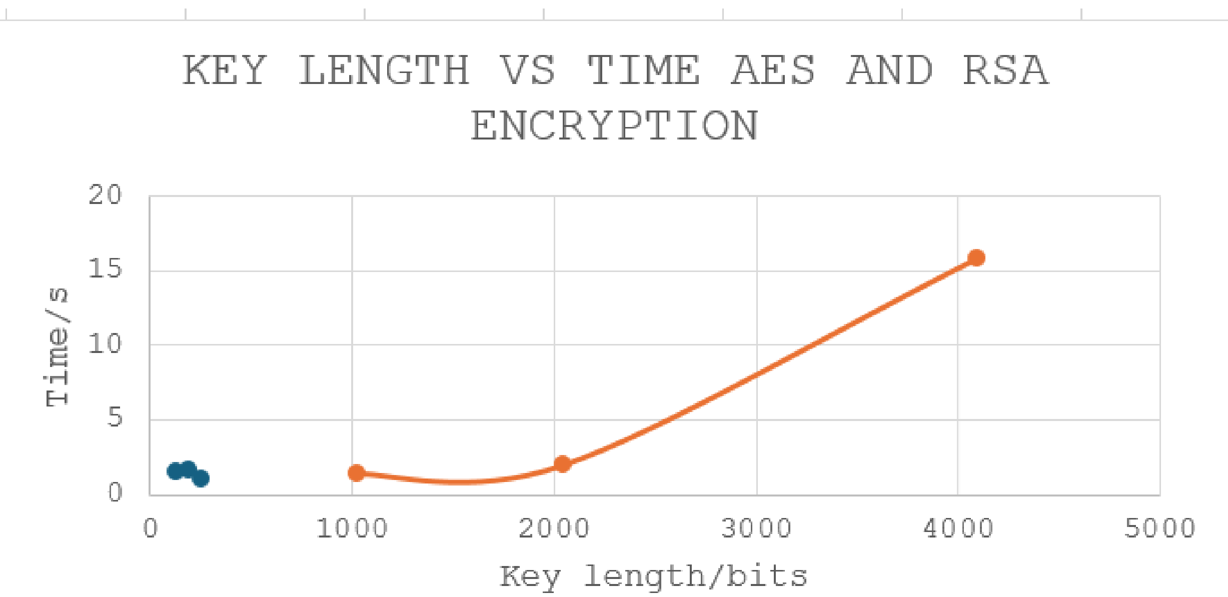
  RSA, in particular, is quite demanding computational wise, especially when it comes to encrypting or decrypting larger information. The performance hit is a lot, a lot more as opposed to symmetric key algorithms like AES. Though, it does well in encrypting smaller data like session keys.

NOTE: FOLLOWING DATA COMPARISON IS BASED ON THE PLAIN TEXT *"test"*.

Blue Line – AES
Red Line - RSA

KEY LENGTH VS TIME AES AND RSA ENCRYPTION

KEY LENGTH (bits)	AES ENCRYPTION TIME (S)	RSA ENCRYPTION TIME (S)
128	1.52	-
192	1.69	-
256	1.09	-
1024	-	1.38
2048	-	1.98
4096	-	15.81



Comparison

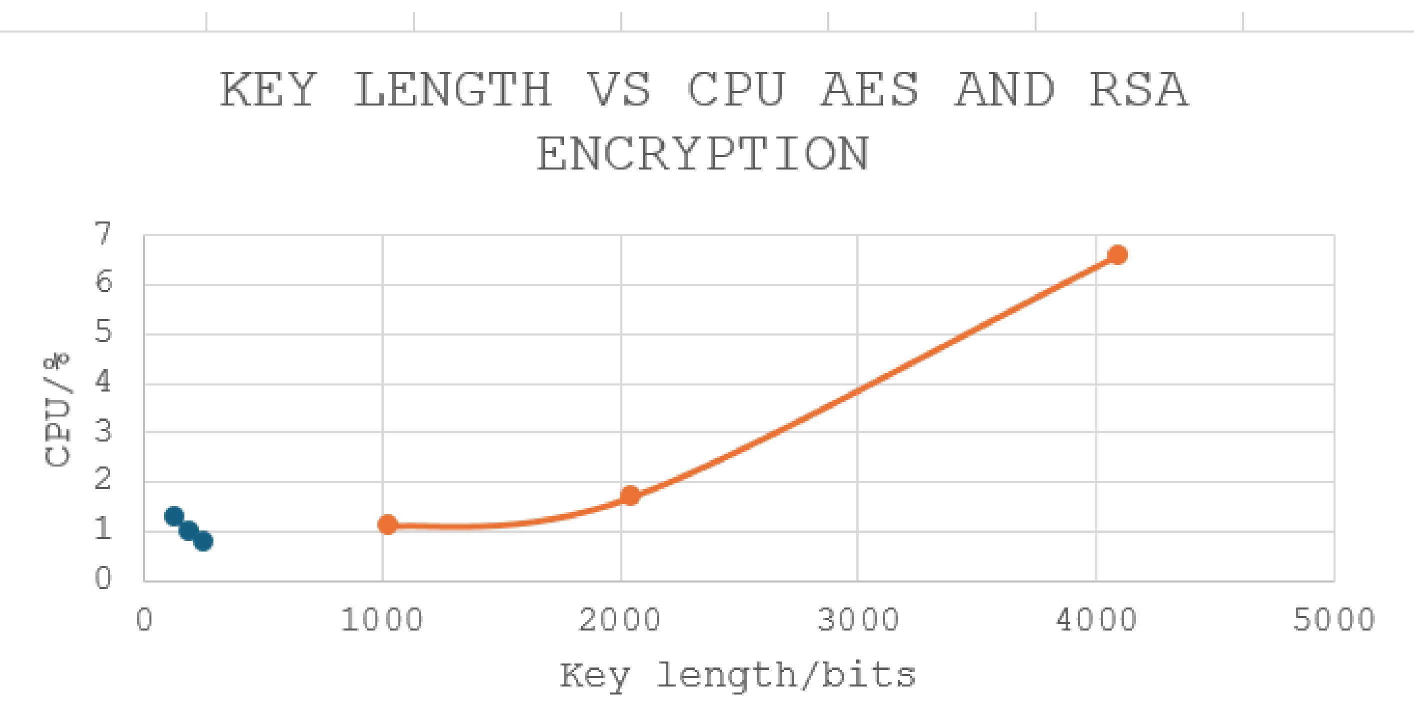
AES Encryption: In case of AES, it is not true that the increase of encryption time is directly proportional to the key length. Quite unexpectedly, the time required for the 256-bit key (which is the longest) was the least (1.09 seconds) while for the 192-bit key, the time taken was maximum (1.69 seconds).

RSA Encryption: RSA encryption times increase significantly with increase in key length. For instance, the time taken to encrypt a 1024-bit key (1.38 seconds) and that of a 2048-bit key (1.98 seconds) varies only slightly. However, with a key length of 4096 bits, the time taken for encryption increases sharply to 15.81 seconds.

This signifies that the RSA encryption algorithm is more vulnerable to the increase in the time taken for different key lengths than that of AES encryption which more consistently varies with time for different key lengths.

KEY LENGTH VS CPU AES AND RSA ENCRYPTION

KEY LENGTH (bits)	AES ENCRYPTION CPU(%)	RSA ENCRYPTION CPU (%)
128	1.3	-
192	1	-
256	0.8	-
1024	-	1.1
2048	-	1.7
4096	-	6.6



Comparison

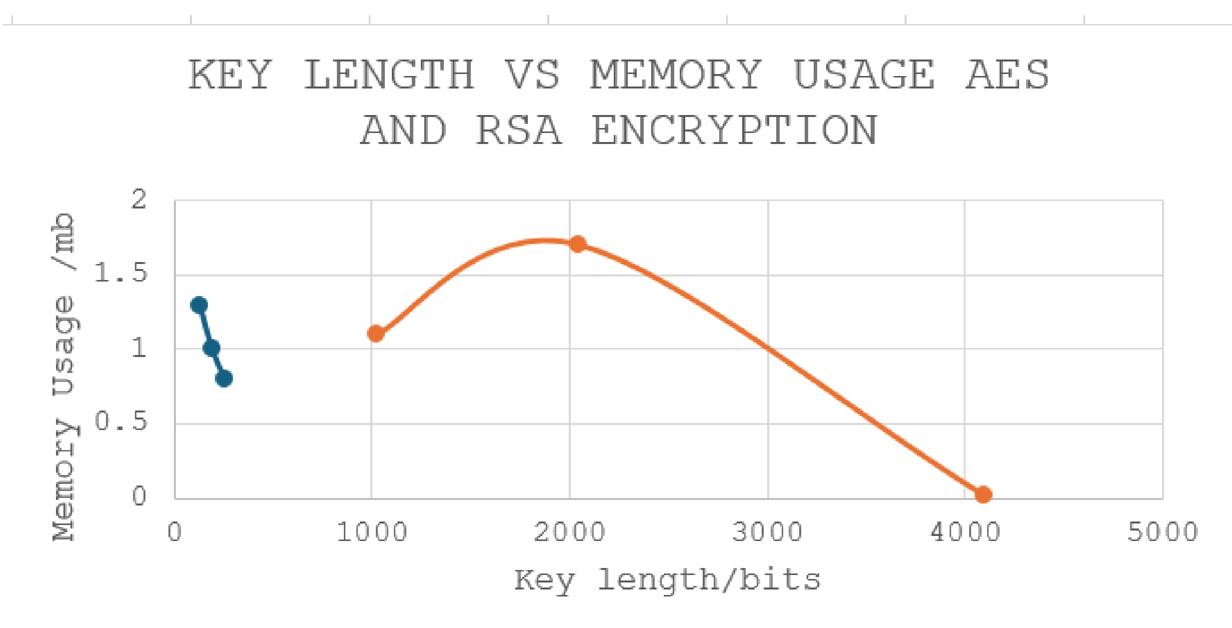
AES Encryption: The CPU usage for the AES encryption/decryption algorithm depends on the key length. For example, with a 128-bit key length, the CPU usage is 1.3%, and this percentage decreases to 0.8% when the key length is 256 bits, suggesting that longer key lengths put a little less processing power on the CPU's usage.

RSA Encryption: As the length of the key increases the use of CPU for RSA also increases greatly. At 1024 bit key, the CPU usage is 1.1% but this increases to 6.6% at 4096-bit key which illustrates that RSA encryption is more taxing on the CPU in particular for bigger key lengths.

It seems that the AES encryption standard does not take much CPU power as the key length is increased whereas the RSA cryptosystem, does take more CPU power as the key length increases, most especially at 4096 bits. This shows that the RSA algorithm is more CUDA intensive than the AES algorithm especially for long keys.

KEY LENGTH VS MEMORY USAGE AES AND RSA ENCRYPTION

KEY LENGTH (bits)	AES ENCRYPTION MEMORY USAGE (mb)	RSA ENCRYPTION MEMORY USAGE (mb)
128	1.12	-
192	0.04	-
256	0.02	-
1024	-	0.07
2048	-	0.01
4096	-	0.02



Comparison

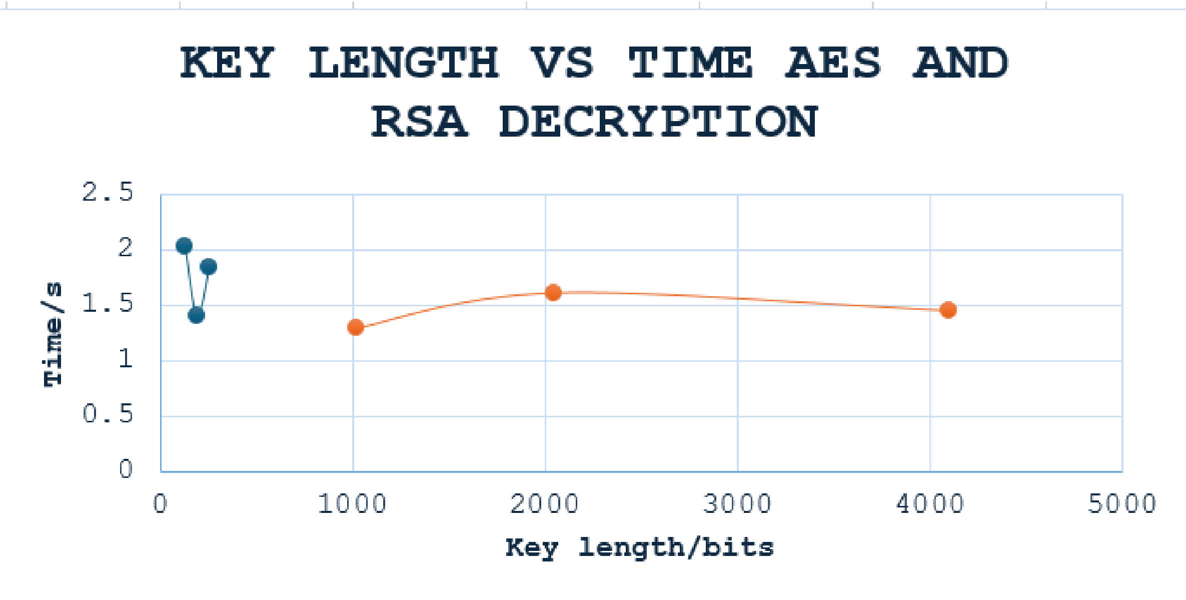
AES Encryption: The amount of memory required by AES plummets significantly with the increase of the key size. For instance, the AES 128-bit key uses 1.12 MB of memory as compared to 2.96 MB and 1.20 MB of 192-bit and 256-bit keys respectively. This implies that longer key lengths in AES tend to be more efficient in this respect.

RSA Encryption: When it comes to RSA's overall memory capacity, it may be considered low for any of the key sizes; however, it does not show any clear increase or decrease with the key size. The 1024-bit key requires 0.07MB whereas the 2048-bit counterpart needs only 0.01MB. Surprisingly, the 4096-bit key needs only 0.02MB implying that in the case of RSA, the memory requirement does not scale linearly with the key size.

The amount of memory consumed by the AES encryption algorithm goes down with an increase in the length of the key, where it can be observed that the 128 bits key consumes much more memory compared to longer keys. On the contrary, it is noted that the RSA encryption algorithm uses very little memory regardless of the key size and there is no readily apparent trend with the changing key sizes. This implies that one could argue that while AES may demand more memory at shorter key lengths, both algorithms are fairly effective with regards to the memory consumed.

KEY LENGTH VS TIME AES AND RSA DECRYPTION

KEY LENGTH (bits)	AES DECRYPTION TIME (s)	RSA DECRYPTION TIME (s)
128	2.03	-
192	1.41	-
256	1.85	-
1024	-	1.29
2048	-	1.61
4096	-	1.45



Comparison

AES Decryption: With regard to AES, decryption timings cannot be said to show consistent trends of increase or decrease as key length is varied. The 192-bit key had the shortest decryption time of 1.41 seconds, while the longest decryption time was recorded at 2.03 seconds for the 128-bit key. The 256-bit key required 1.85 seconds, which is intermediate to the two other key lengths.

RSA Decryption: When it comes to the RSA decryption times, they display a small variation with respect to the key length though it does not show a dramatic increase as in the case of RSA encryption. The 1024-bit key took 1.29 seconds while 2048-bit key took 1.61 seconds. However, somewhat unexpectedly, the 4096-bit key decryption time came out to be 1.45 seconds which is less than the 2048-bit key decryption time.

When it comes to AES decryption, there is no obvious relationship between the length of the key and the time it takes to decrypt data. In contrast with RSA, however, the time taken to decrypt a message with more than a modest increase in key length does experience a slight increase, although the 4096-bit key does better than anticipated. Hence, it follows that both AES and RSA are efficient irrespective of key size, but for some reason, the time taken to decrypt a message using RSA does not appear to increase linearly with key lengths as is the case with encryption.

The Screenshots of the Code

main.py

C: > Users > Uditha J > OneDrive > Desktop > sonnxxxxxx > main.py > SimpleCryptoApp > create_widgets

```python
import tkinter as tk
from tkinter import messagebox
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.PublicKey import RSA
from Crypto.Util.Padding import pad, unpad
from base64 import b64encode, b64decode
import hashlib
import psutil
from time import perf_counter
import os

class SimpleCryptoApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Encrypto")

        # Initialize RSA keys
        self.rsa_key = None
        self.public_key = None
        self.aes_key = None  # AES key is generated automatically

        # Create GUI components
        self.create_widgets()

    def create_widgets(self):
        # Dropdown to select the algorithm (AES or RSA)
        tk.Label(self.root, text="Select Algorithm:").pack(pady=5)
        self.algo_var = tk.StringVar(value="AES")
        self.algo_dropdown = tk.OptionMenu(self.root, self.algo_var, "AES", "RSA", command=self.on_algo_change)
        self.algo_dropdown.pack(pady=5)

        # Dropdown for key size
        tk.Label(self.root, text="Select Key Size:").pack(pady=5)
        self.key_size_var = tk.StringVar(value="128")
        self.key_size_dropdown = tk.OptionMenu(self.root, self.key_size_var, "128", "192", "256")
        self.key_size_dropdown.pack(pady=5)

```

```python
        # Text area for plaintext input
        self.text_input = tk.Text(self.root, height=5, width=40)
        self.text_input.pack(pady=10)

        # Label and text area for displaying the encrypted message
        tk.Label(self.root, text="Encrypted Message:").pack(pady=5)
        self.encrypted_output = tk.Text(self.root, height=5, width=40)
        self.encrypted_output.pack(pady=5)

        # Buttons for encryption and decryption
        tk.Button(self.root, text="Encrypt", command=self.encrypt_message).pack(pady=5)
        tk.Button(self.root, text="Decrypt", command=self.decrypt_message).pack(pady=5)

    def on_algo_change(self, value):
        """Adjust UI based on selected algorithm."""
        if value == "RSA":
            self.key_size_var.set("2048")
            self.key_size_dropdown["menu"].delete(0, "end")
            self.key_size_dropdown["menu"].add_command(label="1024", command=tk._setit(self.key_size_var, "1024"))
            self.key_size_dropdown["menu"].add_command(label="2048", command=tk._setit(self.key_size_var, "2048"))
            self.key_size_dropdown["menu"].add_command(label="4096", command=tk._setit(self.key_size_var, "4096"))
        else:  # AES is selected
            self.key_size_var.set("128")
            self.key_size_dropdown["menu"].delete(0, "end")
            self.key_size_dropdown["menu"].add_command(label="128", command=tk._setit(self.key_size_var, "128"))
            self.key_size_dropdown["menu"].add_command(label="192", command=tk._setit(self.key_size_var, "192"))
            self.key_size_dropdown["menu"].add_command(label="256", command=tk._setit(self.key_size_var, "256"))

    def get_cpu_usage(self):
        return psutil.cpu_percent(interval=None)  # Get current CPU usage percentage

```

```python

    def get_memory_usage(self):
        process = psutil.Process()
        memory_info = process.memory_info()
        return memory_info.rss / (1024 * 1024)  # Convert bytes to MB

    def generate_rsa_key(self, size):
        """Generate a new RSA key pair of the specified size."""
        self.rsa_key = RSA.generate(size)
        self.public_key = self.rsa_key.publickey()

    def generate_aes_key(self, key_size):
        """Generate a new AES key of the specified size in bits."""
        return os.urandom(key_size // 8)  # Generate a random key of size (key_size / 8) bytes

    def aes_encrypt(self, key_size, plaintext):
        """AES encryption function."""
        self.aes_key = self.generate_aes_key(key_size)  # Automatically generate the AES key

        cipher = AES.new(self.aes_key, AES.MODE_CBC)  # AES in CBC mode
        ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block_size))  # Encrypt with padding
        return b64encode(cipher.iv + ciphertext).decode()

    def aes_decrypt(self, key_size, ciphertext):
        """AES decryption function."""
        b64_ciphertext = b64decode(ciphertext)
        cipher = AES.new(self.aes_key, AES.MODE_CBC, b64_ciphertext[:16])  # AES with the same key
        plaintext = unpad(cipher.decrypt(b64_ciphertext[16:]), AES.block_size)  # Decrypt and unpad
        return plaintext.decode()

    def rsa_encrypt(self, plaintext):
        """RSA encryption function."""
        cipher = PKCS1_OAEP.new(self.public_key)
        return b64encode(cipher.encrypt(plaintext.encode())).decode()

```

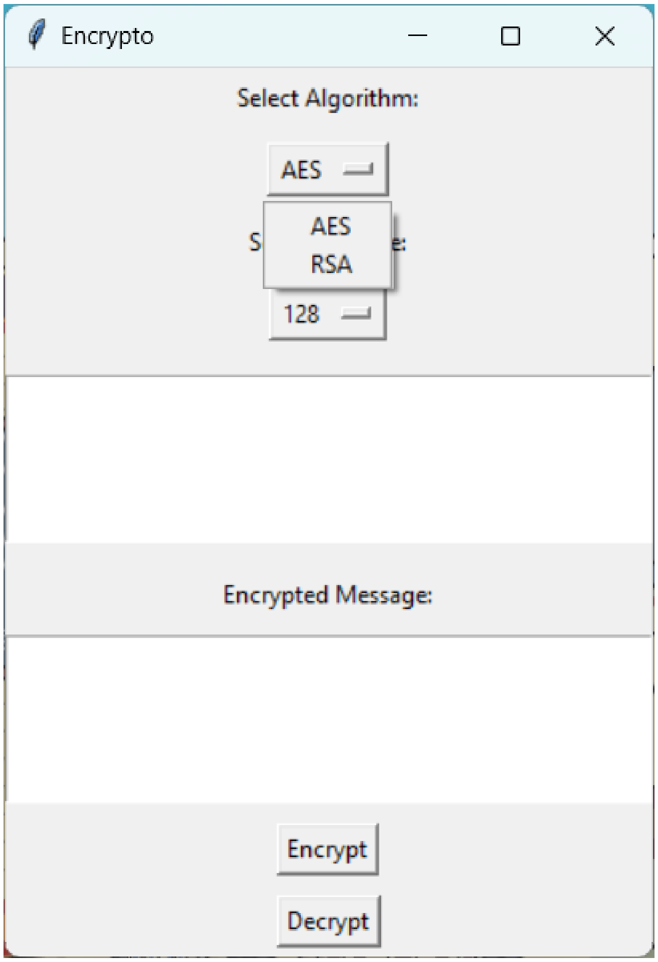
```python

    def rsa_decrypt(self, ciphertext):
        """RSA decryption function."""
        cipher = PKCS1_OAEP.new(self.rsa_key)
        return cipher.decrypt(b64decode(ciphertext)).decode()

    def encrypt_message(self):
        plaintext = self.text_input.get("1.0", tk.END).strip()
        algorithm = self.algo_var.get()
        key_size = int(self.key_size_var.get())

        # Measure time, CPU, and memory usage before operation
        start_time = perf_counter()
        start_cpu = self.get_cpu_usage()
        start_memory = self.get_memory_usage()

        if algorithm == "AES":
            try:
                encrypted_message = self.aes_encrypt(key_size, plaintext)
                self.encrypted_output.delete("1.0", tk.END)
                self.encrypted_output.insert(tk.END, encrypted_message)
                messagebox.showinfo("AES Encryption", f"AES Key (auto-generated): {self.aes_key.hex()}")
            except Exception as e:
                messagebox.showerror("Error", f"Error in AES encryption: {e}")
        else:
            # Generate RSA key based on selected key size
            self.generate_rsa_key(key_size)
            try:
                encrypted_message = self.rsa_encrypt(plaintext)
                self.encrypted_output.delete("1.0", tk.END)
                self.encrypted_output.insert(tk.END, encrypted_message)
                messagebox.showinfo("RSA Encryption", f"RSA Key (auto-generated, {key_size}-bit)")
            except Exception as e:
                messagebox.showerror("Error", f"Error in RSA encryption: {e}")

```

```python

        # Measure time, CPU, and memory usage after operation
        end_time = perf_counter()
        end_cpu = self.get_cpu_usage()
        end_memory = self.get_memory_usage()

        # Calculate time taken, CPU usage difference, and memory usage difference
        time_taken = end_time - start_time
        cpu_usage_diff = end_cpu - start_cpu
        memory_usage_diff = end_memory - start_memory

        messagebox.showinfo("Performance", f"Time Taken: {time_taken:.6f} seconds\n"
                                           f"CPU Usage: {cpu_usage_diff}%\n"
                                           f"Memory Usage Change: {memory_usage_diff:.2f} MB")

    def decrypt_message(self):
        ciphertext = self.encrypted_output.get("1.0", tk.END).strip()
        algorithm = self.algo_var.get()
        key_size = int(self.key_size_var.get())

        # Measure time, CPU, and memory usage before operation
        start_time = perf_counter()
        start_cpu = self.get_cpu_usage()
        start_memory = self.get_memory_usage()

```

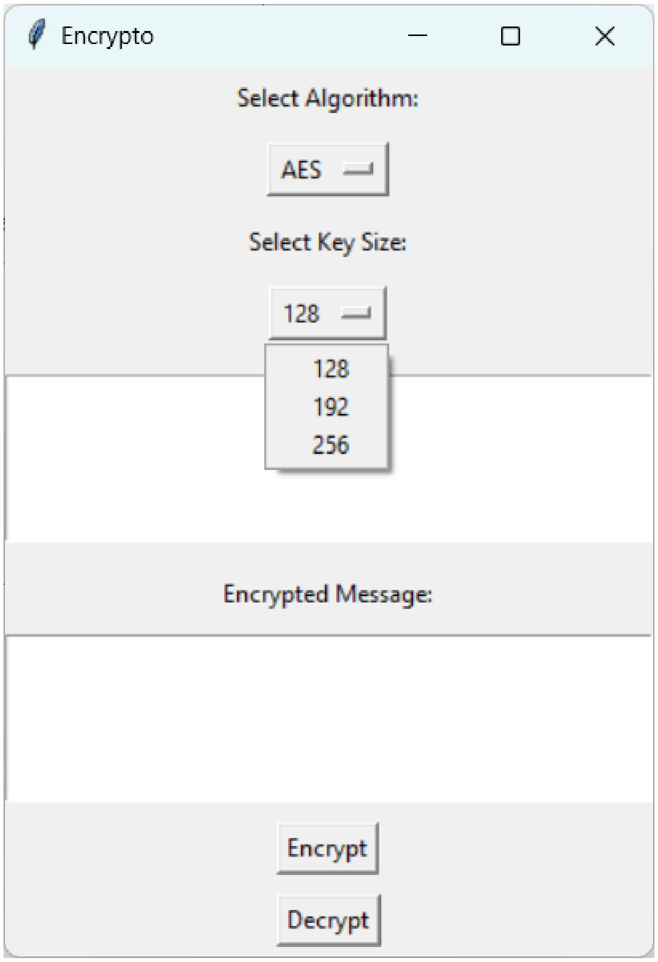
```python

        if algorithm == "AES":
            try:
                decrypted_message = self.aes_decrypt(key_size, ciphertext)
                messagebox.showinfo("Decrypted Message", decrypted_message)
            except Exception as e:
                messagebox.showerror("Error", f"Error in AES decryption: {e}")
        else:
            try:
                decrypted_message = self.rsa_decrypt(ciphertext)
                messagebox.showinfo("Decrypted Message", decrypted_message)
            except Exception as e:
                messagebox.showerror("Error", f"Error in RSA decryption: {e}")

        # Measure time, CPU, and memory usage after operation
        end_time = perf_counter()
        end_cpu = self.get_cpu_usage()
        end_memory = self.get_memory_usage()

        # Calculate time taken, CPU usage difference, and memory usage difference
        time_taken = end_time - start_time
        cpu_usage_diff = end_cpu - start_cpu
        memory_usage_diff = end_memory - start_memory

        messagebox.showinfo("Performance", f"Time Taken: {time_taken:.6f} seconds\n"
                                           f"CPU Usage: {cpu_usage_diff}%\n"
                                           f"Memory Usage Change: {memory_usage_diff:.2f} MB")

if __name__ == "__main__":
    root = tk.Tk()  # Create the main window
    app = SimpleCryptoApp(root)  # Instantiate the app
    root.mainloop()  # Start the GUI event loop

```

User-Interface

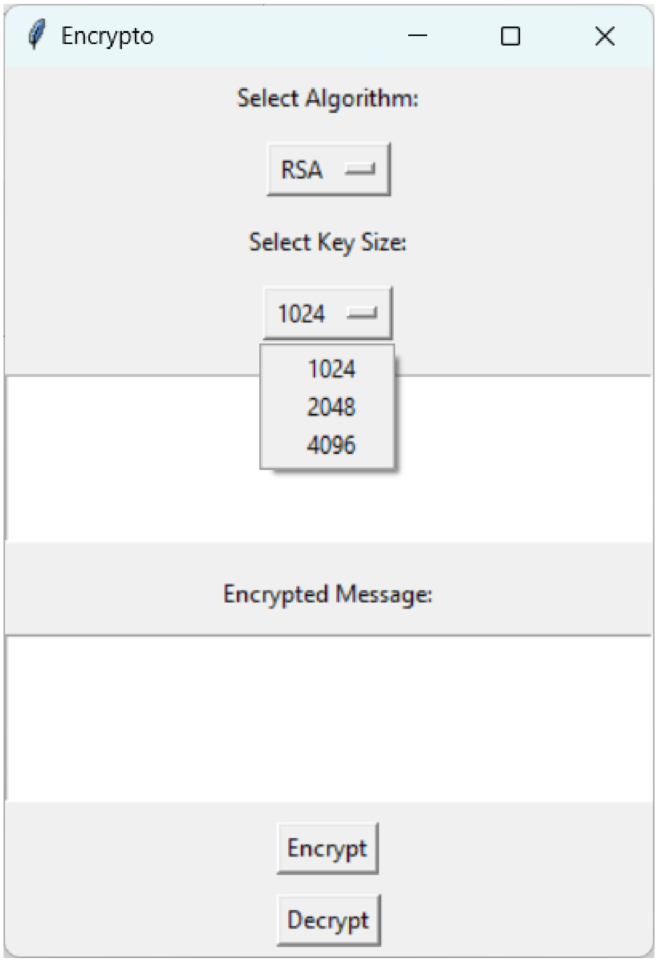
- The two types of algorithms; AES and RSA as follow:



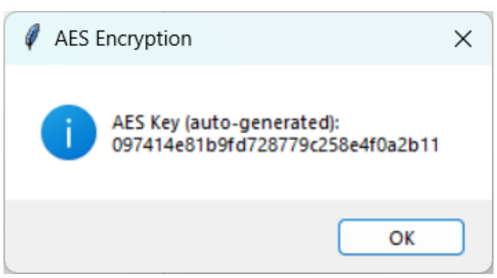
- **AES key sizes:**

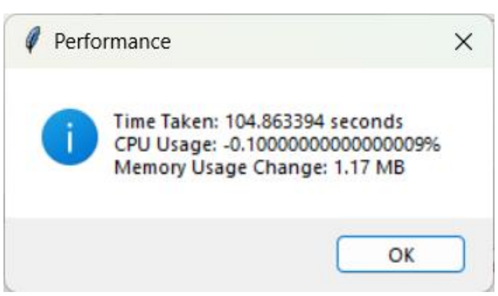


- **RSA key sizes:**

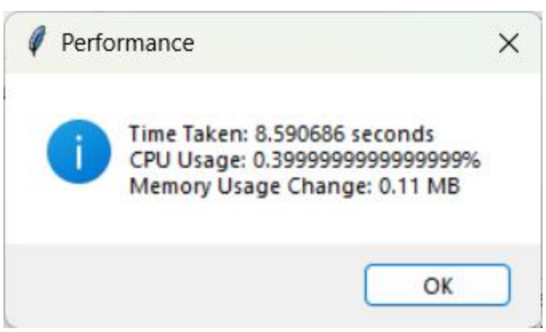


- **AES Encryption:** Encrypted Message, AES Key generated, Time Taken, CPU Usage and Memory Usage Change are shown below;

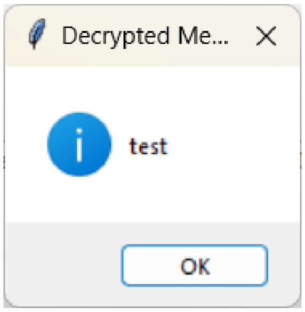


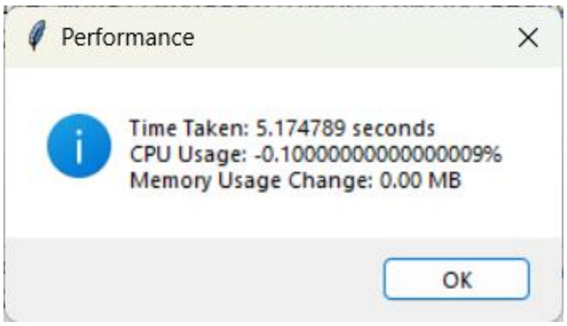


- **RSA Encryption:** Encrypted Message, Time Taken, CPU Usage and Memory Usage Change are shown below;



- **AES Decryption:** Decrypted Message, Time Taken, CPU Usage and Memory Usage Change are shown below;





- **RSA Decryption:** Decrypted Message, Time Taken, CPU Usage and Memory Usage Change are shown below;