RSA Encryption to Enhance Embedded Data Security: Application in Audio Steganography

Muhammad Adha Ridwan 13523098
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13523098@std.stei.itb.ac.id

Abstract—**This paper presents the development and implementation of a secure audio steganography system combining Least Significant Bit (LSB) embedding and RSA encryption. The methodology begins by compressing the secret audio file to optimize its size, followed by splitting the compressed data into chunks and encrypting each chunk using RSA-2048 with Optimal Asymmetric Encryption Padding (OAEP). The encrypted data is then embedded into the least significant bits of the carrier audio file, ensuring minimal perceptual distortion. The stego file is subsequently extracted, decrypted, and reconstructed into its original format, demonstrating the system's robustness and accuracy.**

Keywords—**Audio steganography, RSA encryption, Least Significant Bit (LSB), data security.**

I. Introduction

In the contemporary digital landscape, the secure transmission of sensitive information has become increasingly crucial. As cyber threats continue to evolve, traditional data protection mechanisms may no longer provide adequate security. Audio steganography, combined with robust cryptographic techniques like RSA encryption, offers a promising solution for enhanced data security in embedded systems.

Audio steganography, the art of concealing information within audio files, has gained significant attention due to its potential for covert communication. While steganography alone provides security through obscurity, its integration with RSA encryption creates a formidable two-layer security mechanism. This hybrid approach addresses both data hiding and cryptographic security, making it particularly valuable for embedded systems where resources are constrained and security requirements are stringent.

The RSA algorithm, named after Rivest, Shamir, and Adleman, remains one of the most widely used public-key cryptosystems. Its mathematical foundation, based on the computational complexity of prime factorization, provides a robust framework for secure data encryption. When applied to audio steganography, RSA encryption ensures that even if the presence of hidden data is detected, the encrypted content remains protected from unauthorized access.

The integration of cryptographic algorithms with steganographic techniques presents a robust solution to enhance data security. Among various cryptographic approaches, RSA (Rivest-Shamir-Adleman) encryption has proven to be particularly effective due to its asymmetric nature and strong security foundations. The combination of RSA encryption with audio steganography creates a dual-layer security mechanism: the data is first encrypted before being concealed within the audio carrier, significantly increasing the complexity for potential attackers.

The application of RSA encryption in audio steganography involves several critical challenges. These include managing the increased payload size of encrypted data, maintaining the imperceptibility of the steganographic process, and ensuring efficient implementation within resource-constrained embedded systems. Additionally, the need to preserve the audio quality while accommodating the encrypted payload requires careful consideration of embedding algorithms and parameters.

This paper presents a comprehensive framework for implementing RSA encryption in audio steganography. The proposed approach addresses the aforementioned challenges through an RSA implementation, an adaptive embedding scheme that maintains audio quality, and a robust key management system for secure deployment.

II. Theoretical Foundation

A. Number Theory

Number theory constitutes a mathematical discipline dedicated to investigating the properties and relationships of natural numbers, encompassing topics such as divisibility, prime decomposition, and integer solutions to equations.

These theoretical foundations have significant applications. This paper explores their applications in cryptography, particularly the RSA algorithm, which relies on the computational difficulty of factoring large numbers into their prime components to ensure secure data transmission.

The concepts that are applied in RSA algorithm are the following:

1. Prime Number
   Let p be a positive integer where $p > 1$, p is said to be a prime number if and only if it has exactly two distinct factors, 1 and p.
2. Modular Arithmetic
   Modular arithmetic is a system of arithmetic for an integer, the numbers circled around the system when exceeding certain value, this value is called a modulus.
3. Euler’s Totient Function
   Euler’s totient function ($\phi(n)$) counts the number of integers less than n that are coprime to n.

4. Greatest Common Divisor
   The GCD of two numbers is the largest number that divides both without remainder. Pair of numbers are called coprime if their GCD is equal to 1.
5. Modulus Inverse
   The modulus inverse of a number *e* under modulus *m* is a number d such that $e \cdot d \equiv 1 \pmod{m}$.

B. RSA Encryption

RSA (Rivest-Shamir-Adleman) is a public key cryptosystem where it generates two key, public key and private key. Public key is shared with all user hence others know about this key. It is used for encrypting data, ensuring sensitive information can be securely transmitted, meanwhile the private key is kept only for user or authenticated receiver only. It is used for decrypting the encrypted text back to original meaning.

RSA's security lies in the difficulty of factoring large numbers composite number, a computationally intensive problem. RSA generates key based on the product of two large prime numbers. This makes the public key (often denoted as *n)* and an encryption exponent (denoted as *e),* while the private key involves a more complex process to find the decryption exponent (denoted as *d*) derived using modular arithmetic and Euler's totient function.

The key features of RSA encryption are following:

1. Asymmetric Encryption
   RSA uses two different keys for encryption and decryption process, unlike symmetric encryption, where the same key is being used.
2. Secure Communication
   RSA allow a secure communication of exchanging information without the needing the parties to share a secret key beforehand.
3. Digital Signatures
   RSA is also used for authentication and data integrity verification through digital signatures.

Here how the encryption algorithm works:

1. Key Generation
   The user generates two large primes, typically more than 200 digits, then calculates *n* as their product, *n* is shared with others.
2. Public Key
   A public key *e* is calculated by finding a prime that relatively prime with φ(n) or we can denote as $\mathrm{GCD}(\varphi(n), e) = 1$.
3. Private Key
   A private key *d* is calculated by finding *d* that satisfy $e \cdot d \equiv 1 (\mathrm{mod}\ \varphi(n))$. *d* kept secret for decrypting the data. φ(n) is a number that is relatively prime with *p* and *q* and is kept hidden.
4. Encryption
   The user / sender uses the public key (*n* and *e*) to compute a ciphertext by raising the message m to the power of *e* and dividing by *n.*
5. Decryption
   The receiver uses the private key (*d*) to compute the plaintext by raising the ciphertext to the power of *d* and dividing by n.

RSA algorithm is one of the most powerful cryptography algorithms to date, since there is no efficient way to compute the factor of n, n is a very large prime number.

C. Audio Steganography

Audio steganography is a method of embedding secret information within an audio signal in such a way that the changes are not detectable within human auditory system. Unlike traditional cryptography, which aim to make data unreadable to unauthorized parties, steganography focuses on concealing the very existence and the trace of the data. This makes steganography an attractive approach for secure information exchange, communication, and copyright protection such as watermark.

The Least Significant Bit (LSB) is one the simplest and most widely used techniques for audio steganography. It operates by modifying the least significant bits of the audio samples to encode the secret information. Since the secret data is placed in the least perceptible often ignorable portion of the audio data, the changes are inaudible to the human ear. For an audio sample S_i, the modified audio-sample S_i' can be represented as:

$$S_i' = S_i - (S_i \bmod 2) + b_i$$

From formula above, b_i represent the bits of the secret messages to be embedded. The LSB method ensure minimal distortion of the original audio signal while embedding the data. The process involves splitting the secret message into a binary stream and embedding each bit sequentially into the least significant bits of the audio samples. During extraction, the binary stream is reconstructed by reading the least significant bits of the audio-samples.

The main advantage of the LSB method is its simplicity and high embedding capacity, allowing significant amounts of data to be hidden within an audio file. However, it has limitations, such as vulnerability to lossy compression and noise. Any alteration to the audio file, such as re-encoding or adding background noise, can compromise the hidden data. In practical applications, the LSB method is used in scenarios requiring high embedding capacity and low computational complexity. For instance, it is commonly employed for embedding sensitive information in digital audio files for secure communication or watermarking.

Despite its simplicity, the LSB method exemplifies the core principles of audio steganography by balancing concealment and minimal distortion, making it a practical choice for various applications.

D. Audio Encryption Using RSA

RSA audio encryption transforms sound files into secure digital data by first converting the audio into numbers that represent the sound waves. These numbers are then broken down into smaller chunks that work well with RSA encryption. Each chunk gets encrypted using a public key, creating protected data that can only be unlocked with a matching private key.

The encryption process starts with converting the audio file into a discrete digital sample. Each sample or block of samples is then encrypted using the RSA algorithm.

For an example, let's take a look at this example:

Assuming RSA parameters are $e = 65537$, $n = 11413$, $d = 1573$, for example, a block of original samples

Table 1. Example of the samples.

Original Samples									
123	456	789	101	202	303	404	505	606	707

Using RSA, each sample *m* is encrypted using the formula:

$$c = m^e \bmod n$$

Calculations examples are shown below:

1. $m = 123, c = 123^{65537} \bmod 11413 = 123$
2. $m = 456, c = 456^{65537} \bmod 11413 = 4121$
3. $m = 789, c = 789^{65537} \bmod 11413 = 8161$
4. $m = 101, c = 101^{65537} \bmod 11413 = 5209$
5. $m = 202. c = 202^{65537} \bmod 11413 = 10436$
6. $m = 303, c = 303^{65537} \bmod 11413 = 2249$
7. $m = 404, c = 404^{65537} \bmod 11413 = 6741$
8. $m = 505, c = 505^{65537} \bmod 11413 = 3289$
9. $m = 606, c = 606^{65537} \bmod 11413 = 9291$
10. $m = 707, c = 707^{65537} \bmod 11413 = 5775$

From the result of encryption above we can form the tables of block of samples after encryption.

Table 2. Encrypted Example Samples.

Encrypted Samples									
123	4121	8161	5209	10436	2249	6741	3289	9291	5775

RSA encryption transforms each sample into a unique ciphertext, making the data unreadable without the corresponding private key Even though in this example appear a value that seems unchanged due to modular arithmetic, the RSA encryption ensures security and cannot be reversed without the private key.

III. Proposed Scheme

A. Encryption and Embedding Scheme

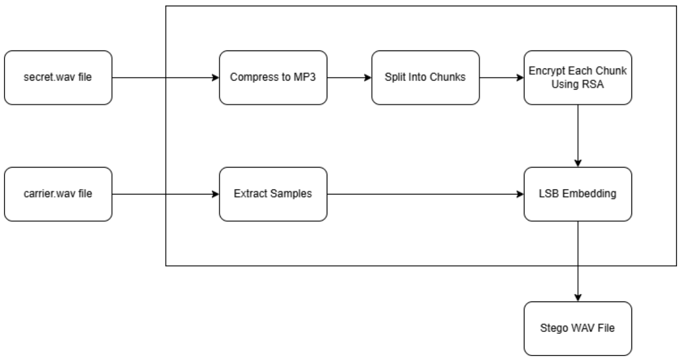


Figure 1. Proposed encryption and embedding scheme

The proposed encryption and embedding methodology begin by acquiring the secret and carrier WAV files. The secret audio is first compressed into an MP3 format to reduce its size, optimizing the available space within the carrier audio for embedding. This compression step helps balance efficiency and quality, ensuring that the carrier file can hold the secret data without significant distortion. The compressed audio is then divided into multiple chunks or blocks, each prepared for encryption.

To secure the secret audio, the chunks are encrypted using RSA-2048 with Optimal Asymmetric Encryption Padding (OAEP). This encryption ensures that the secret data remains confidential and tamper-proof, as decryption is only possible with the corresponding private key. Simultaneously, the carrier audio samples are extracted to serve as the embedding medium. The Least Significant Bit (LSB) steganography technique is used, embedding the encrypted chunks into the carrier's LSBs, as modifications in this region are imperceptible to the human ear.

Finally, the modified carrier audio, now containing the encrypted secret data, is saved as a stego WAV file. This file combines the original carrier audio with the hidden, securely encrypted secret, maintaining the audio's quality while ensuring the secret data is safely embedded and ready for secure transmission or storage.

B. Extraction and Decryption Scheme

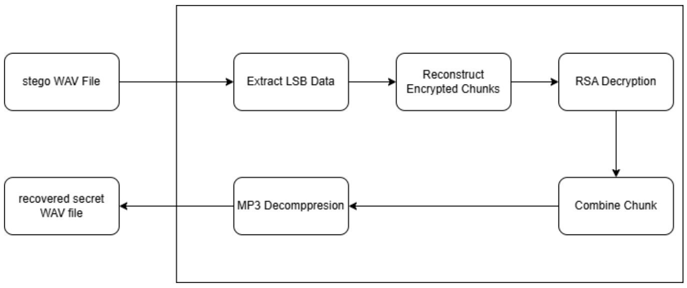


Figure 2. Proposed extraction and decryption scheme

The proposed extraction and decryption methodology begins with retrieving the stego audio file containing the embedded secret data. The Least Significant Bits (LSBs) of the carrier audio are extracted, providing the hidden data that will undergo reconstruction. During this process, the encrypted chunks embedded in the LSBs are reassembled to prepare them for decryption.

Next, each reconstructed chunk is decrypted using RSA-2048 with the private key d. This step ensures the recovery of the original secret data that had been securely encrypted during the embedding phase. Once all the chunks are decrypted, they are combined to recreate the original audio data in its compressed MP3 format.

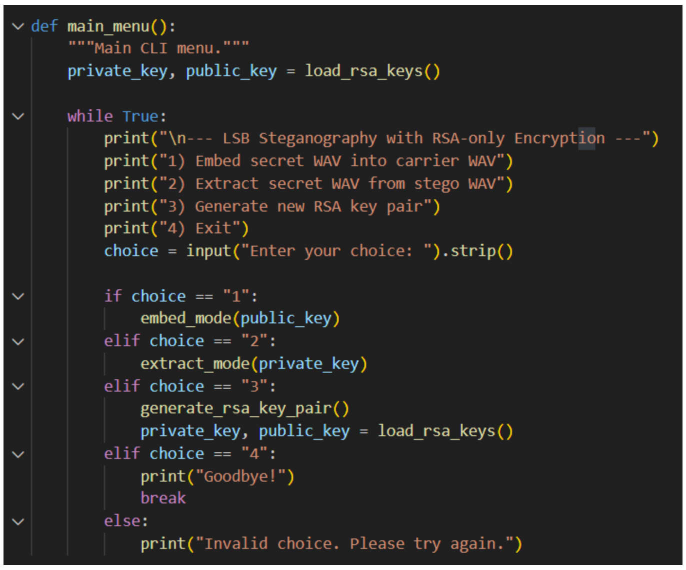
Finally, the reconstructed MP3 data undergoes decompression to restore it to its original WAV audio format. This step completes the process, delivering the secret audio file in its original, usable state while maintaining the integrity of the extracted and decrypted data.

IV. Implementation

This program is developed using the Python programming language, chosen for its versatility and robust ecosystem of libraries that simplify handling audio files and encryption tasks. The *os* module is utilized for managing file paths and system operations, while the *wave* module facilitates reading and writing WAV audio files. To handle audio data in chunks, the *struct* module is employed, enabling efficient manipulation of binary data.

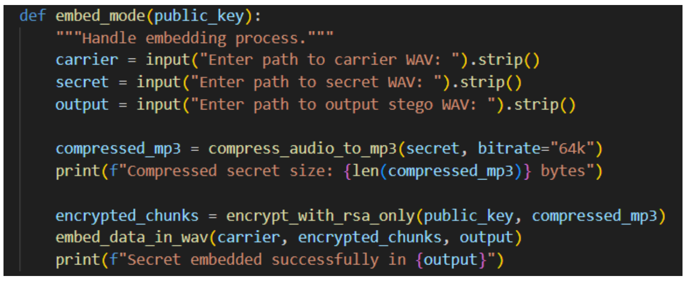
Additionally, the *pydub* module is leveraged for compressing and decompressing audio files, making it easier to manage file sizes during the embedding and extraction processes. The *io* module is used for handling in-memory byte streams, ensuring smooth data flow between operations. Lastly, the *cryptography* library provides the tools necessary for implementing RSA encryption and decryption, ensuring the secure handling of sensitive audio data throughout the program.

A. Main Program



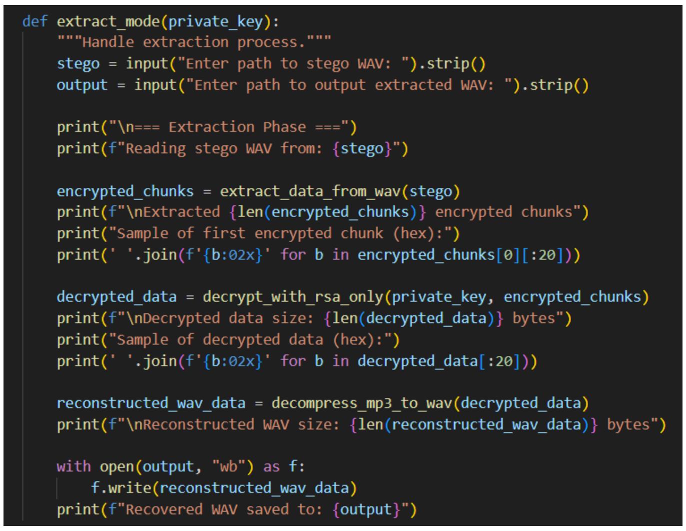
```python
def main_menu():
    """Main CLI menu."""
    private_key, public_key = load_rsa_keys()

    while True:
        print("\n--- LSB Steganography with RSA-only Encryption ---")
        print("1) Embed secret WAV into carrier WAV")
        print("2) Extract secret WAV from stego WAV")
        print("3) Generate new RSA key pair")
        print("4) Exit")
        choice = input("Enter your choice: ").strip()

        if choice == "1":
            embed_mode(public_key)
        elif choice == "2":
            extract_mode(private_key)
        elif choice == "3":
            generate_rsa_key_pair()
            private_key, public_key = load_rsa_keys()
        elif choice == "4":
            print("Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")
```

Figure 3. Main menu source code



```python
def embed_mode(public_key):
    """Handle embedding process."""
    carrier = input("Enter path to carrier WAV: ").strip()
    secret = input("Enter path to secret WAV: ").strip()
    output = input("Enter path to output stego WAV: ").strip()

    compressed_mp3 = compress_audio_to_mp3(secret, bitrate="64k")
    print(f"Compressed secret size: {len(compressed_mp3)} bytes")

    encrypted_chunks = encrypt_with_rsa_only(public_key, compressed_mp3)
    embed_data_in_wav(carrier, encrypted_chunks, output)
    print(f"Secret embedded successfully in {output}")
```

Figure 4. Embed Menu

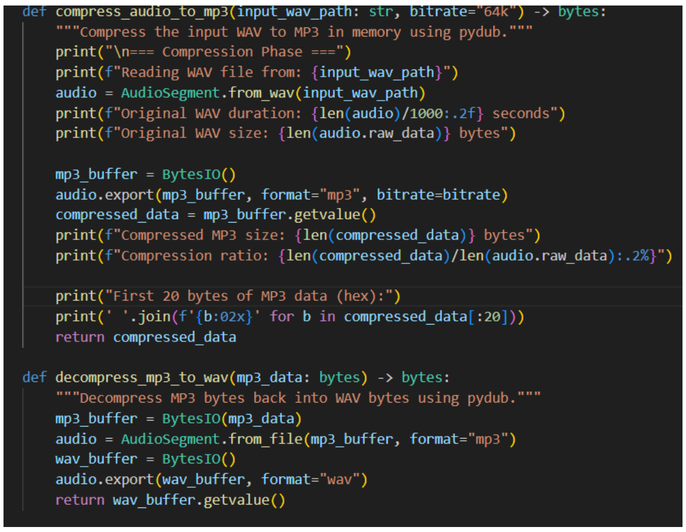


```python
def extract_mode(private_key):
    """Handle extraction process."""
    stego = input("Enter path to stego WAV: ").strip()
    output = input("Enter path to output extracted WAV: ").strip()

    print("\n=== Extraction Phase ===")
    print(f"Reading stego WAV from: {stego}")

    encrypted_chunks = extract_data_from_wav(stego)
    print(f"\nExtracted {len(encrypted_chunks)} encrypted chunks")
    print("Sample of first encrypted chunk (hex):")
    print(' '.join(f'{b:02x}' for b in encrypted_chunks[0][:20]))

    decrypted_data = decrypt_with_rsa_only(private_key, encrypted_chunks)
    print(f"\nDecrypted data size: {len(decrypted_data)} bytes")
    print("Sample of decrypted data (hex):")
    print(' '.join(f'{b:02x}' for b in decrypted_data[:20]))

    reconstructed_wav_data = decompress_mp3_to_wav(decrypted_data)
    print(f"\nReconstructed WAV size: {len(reconstructed_wav_data)} bytes")

    with open(output, "wb") as f:
        f.write(reconstructed_wav_data)
    print(f"Recovered WAV saved to: {output}")
```

Figure 5. Extract Menu

In this section, all the function are implemented in accordance with each function. This program features a CLI Menu based on the input and control flow desired by the user.

B. Compression Utilities

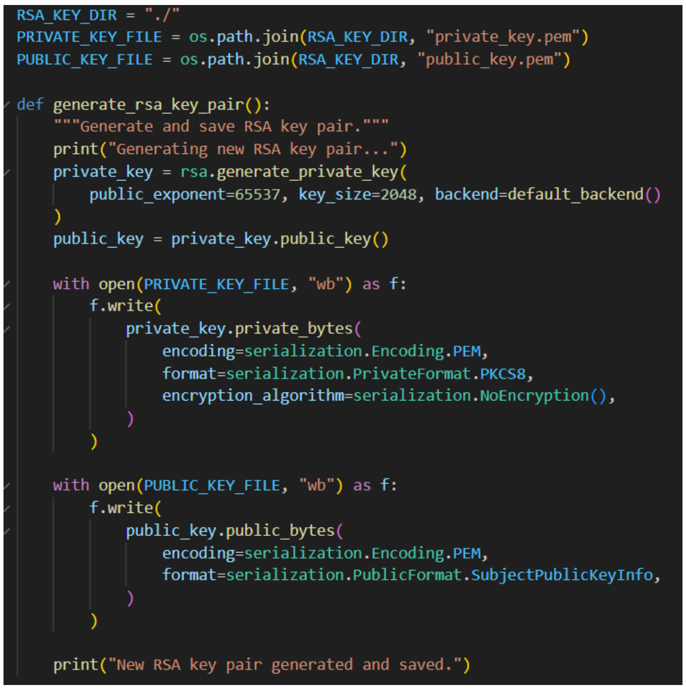


```python
def compress_audio_to_mp3(input_wav_path: str, bitrate="64k") -> bytes:
    """Compress the input WAV to MP3 in memory using pydub."""
    print("\n=== Compression Phase ===")
    print(f"Reading WAV file from: {input_wav_path}")
    audio = AudioSegment.from_wav(input_wav_path)
    print(f"Original WAV duration: {len(audio)/1000:.2f} seconds")
    print(f"Original WAV size: {len(audio.raw_data)} bytes")

    mp3_buffer = BytesIO()
    audio.export(mp3_buffer, format="mp3", bitrate=bitrate)
    compressed_data = mp3_buffer.getvalue()
    print(f"Compressed MP3 size: {len(compressed_data)} bytes")
    print(f"Compression ratio: {len(compressed_data)/len(audio.raw_data):.2%}")

    print("First 20 bytes of MP3 data (hex):")
    print(' '.join(f'{b:02x}' for b in compressed_data[:20]))
    return compressed_data

def decompress_mp3_to_wav(mp3_data: bytes) -> bytes:
    """Decompress MP3 bytes back into WAV bytes using pydub."""
    mp3_buffer = BytesIO(mp3_data)
    audio = AudioSegment.from_file(mp3_buffer, format="mp3")
    wav_buffer = BytesIO()
    audio.export(wav_buffer, format="wav")
    return wav_buffer.getvalue()
```

Figure 6. Compression Utilities

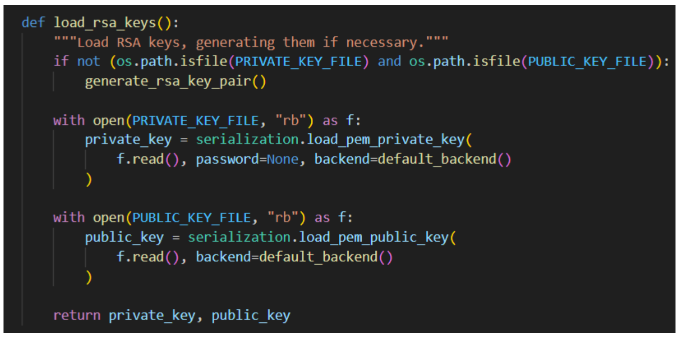
This section includes two functions that handle the compression and decompression of the audio file. The compression function is used to optimize the size of the secret.wav file, making the embedding process more efficient by reducing its file size. On the other hand, the decompression function is applied after decrypting and reconstructing the combined chunks. Since the reconstructed audio is in MP3 format rather than WAV, decompression is necessary to convert it back into the original WAV format.

C. RSA Key Manager



```python
RSA_KEY_DIR = "./"
PRIVATE_KEY_FILE = os.path.join(RSA_KEY_DIR, "private_key.pem")
PUBLIC_KEY_FILE = os.path.join(RSA_KEY_DIR, "public_key.pem")

def generate_rsa_key_pair():
    """Generate and save RSA key pair."""
    print("Generating new RSA key pair...")
    private_key = rsa.generate_private_key(
        public_exponent=65537, key_size=2048, backend=default_backend()
    )
    public_key = private_key.public_key()

    with open(PRIVATE_KEY_FILE, "wb") as f:
        f.write(
            private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.PKCS8,
                encryption_algorithm=serialization.NoEncryption(),
            )
        )

    with open(PUBLIC_KEY_FILE, "wb") as f:
        f.write(
            public_key.public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo,
            )
        )

    print("New RSA key pair generated and saved.")
```

Figure 7. RSA key pair generator

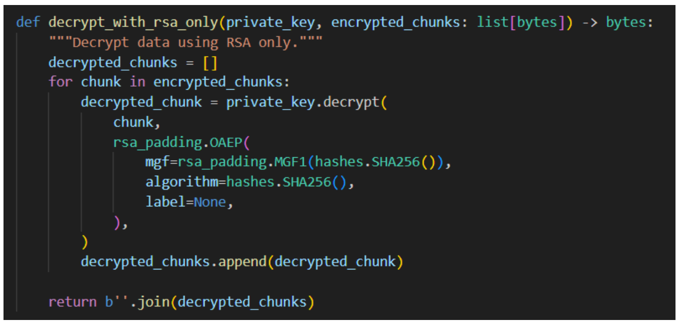


```python
def load_rsa_keys():
    """Load RSA keys, generating them if necessary."""
    if not (os.path.isfile(PRIVATE_KEY_FILE) and os.path.isfile(PUBLIC_KEY_FILE)):
        generate_rsa_key_pair()

    with open(PRIVATE_KEY_FILE, "rb") as f:
        private_key = serialization.load_pem_private_key(
            f.read(), password=None, backend=default_backend()
        )

    with open(PUBLIC_KEY_FILE, "rb") as f:
        public_key = serialization.load_pem_public_key(
            f.read(), backend=default_backend()
        )

    return private_key, public_key
```

Figure 8. RSA key loader

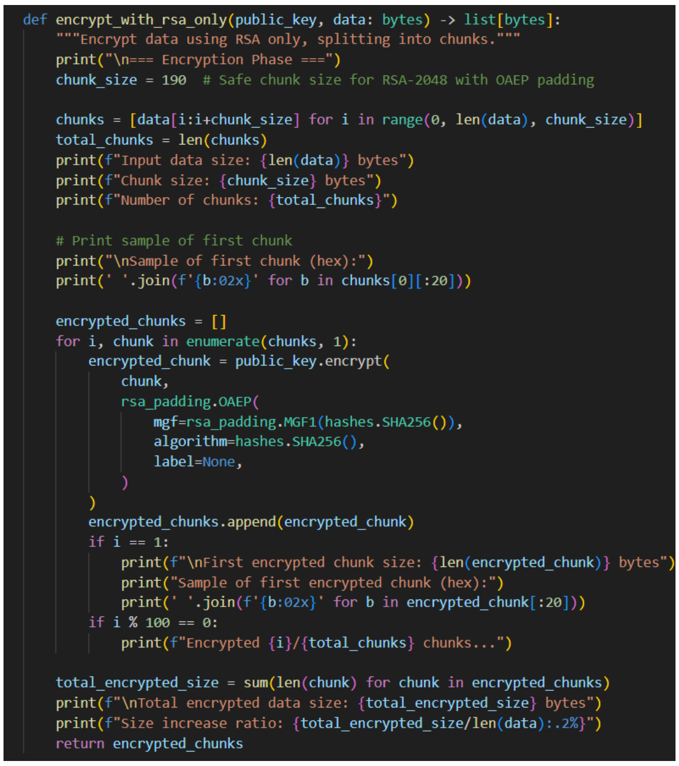
This section includes two functions that act as the RSA key manager. The first function generates an RSA key pair, creating both the public and private keys. The second function is responsible for loading the key pair from local storage, ensuring the keys are accessible for encryption and decryption operations.

D. RSA Audio Encryption-Decryption



```python
def decrypt_with_rsa_only(private_key, encrypted_chunks: list[bytes]) -> bytes:
    """Decrypt data using RSA only."""
    decrypted_chunks = []
    for chunk in encrypted_chunks:
        decrypted_chunk = private_key.decrypt(
            chunk,
            rsa_padding.OAEP(
                mgf=rsa_padding.MGF1(hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None,
            ),
        )
        decrypted_chunks.append(decrypted_chunk)

    return b''.join(decrypted_chunks)
```

Figure 9. RSA Audio Decryptor

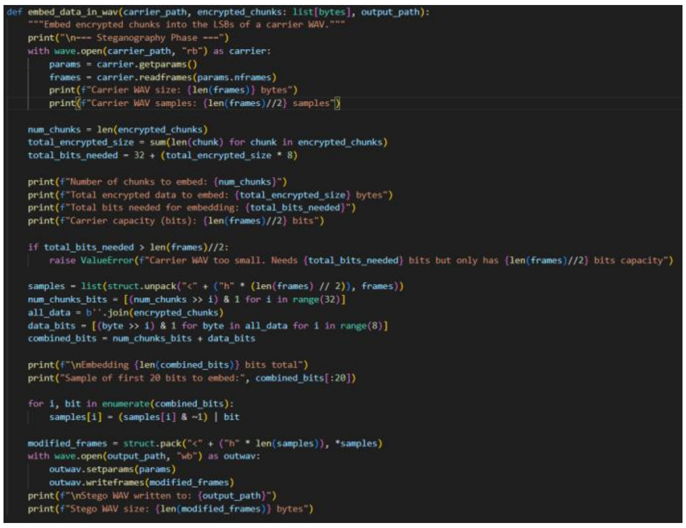


```python
def encrypt_with_rsa_only(public_key, data: bytes) -> list[bytes]:
    """Encrypt data using RSA only, splitting into chunks."""
    print("\n=== Encryption Phase ===")
    chunk_size = 190  # Safe chunk size for RSA-2048 with OAEP padding

    chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
    total_chunks = len(chunks)
    print(f"Input data size: {len(data)} bytes")
    print(f"Chunk size: {chunk_size} bytes")
    print(f"Number of chunks: {total_chunks}")

    # Print sample of first chunk
    print("\nSample of first chunk (hex):")
    print(' '.join(f'{b:02x}' for b in chunks[0][:20]))

    encrypted_chunks = []
    for i, chunk in enumerate(chunks, 1):
        encrypted_chunk = public_key.encrypt(
            chunk,
            rsa_padding.OAEP(
                mgf=rsa_padding.MGF1(hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None,
            )
        )
        encrypted_chunks.append(encrypted_chunk)
        if i == 1:
            print(f"\nFirst encrypted chunk size: {len(encrypted_chunk)} bytes")
            print("Sample of first encrypted chunk (hex):")
            print(' '.join(f'{b:02x}' for b in encrypted_chunk[:20]))
        if i % 100 == 0:
            print(f"Encrypted {i}/{total_chunks} chunks...")

    total_encrypted_size = sum(len(chunk) for chunk in encrypted_chunks)
    print(f"\nTotal encrypted data size: {total_encrypted_size} bytes")
    print(f"Size increase ratio: {total_encrypted_size/len(data):.2%}")
    return encrypted_chunks
```

Figure 10. RSA Audio Encryptor

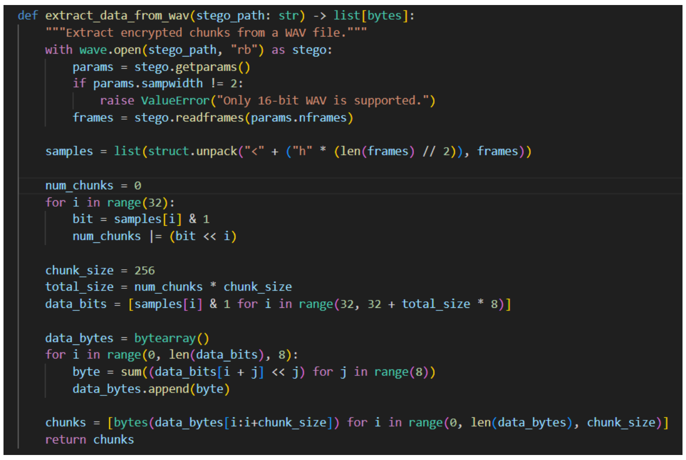
This section includes two functions dedicated to handling the encryption and decryption processes using RSA. The first function serves as the RSA encryptor, responsible for securely encrypting data in chunks using the public key and ensuring confidentiality during transmission or storage. The second function acts as the RSA decryptor, using the private key to decrypt the encrypted chunks and reconstruct the original data. Together, these functions ensure a seamless and secure encryption-decryption workflow, adhering to RSA standards for secure communication.

E. LSB Steganography Utilities



```python
def embed_data_in_wav(carrier_path, encrypted_chunks: list[bytes], output_path):
    """Embed encrypted chunks into the LSBs of a carrier WAV."""
    print("\n--- Steganography Phase ---")
    with wave.open(carrier_path, "rb") as carrier:
        params = carrier.getparams()
        frames = carrier.readframes(params.nframes)
        print(f"Carrier WAV size: {len(frames)} bytes")
        print(f"Carrier WAV samples: {len(frames)//2} samples")

    num_chunks = len(encrypted_chunks)
    total_encrypted_size = sum(len(chunk) for chunk in encrypted_chunks)
    total_bits_needed = 32 + (total_encrypted_size * 8)

    print(f"Number of chunks to embed: {num_chunks}")
    print(f"Total encrypted data to embed: {total_encrypted_size} bytes")
    print(f"Total bits needed for embedding: {total_bits_needed}")
    print(f"Carrier capacity (bits): {len(frames)//2} bits")

    if total_bits_needed > len(frames)//2:
        raise ValueError(f"Carrier WAV too small. Needs {total_bits_needed} bits but only has {len(frames)//2} bits capacity")

    samples = list(struct.unpack("<" + ("h" * (len(frames) // 2)), frames))
    num_chunks_bits = [(num_chunks >> i) & 1 for i in range(32)]
    all_data = b''.join(encrypted_chunks)
    data_bits = [(byte >> i) & 1 for byte in all_data for i in range(8)]
    combined_bits = num_chunks_bits + data_bits

    print(f"\nEmbedding {len(combined_bits)} bits total")
    print("Sample of first 20 bits to embed:", combined_bits[:20])

    for i, bit in enumerate(combined_bits):
        samples[i] = (samples[i] & ~1) | bit

    modified_frames = struct.pack("<" + ("h" * len(samples)), *samples)
    with wave.open(output_path, "wb") as outwav:
        outwav.setparams(params)
        outwav.writeframes(modified_frames)
    print(f"\nStego WAV written to: {output_path}")
    print(f"Stego WAV size: {len(modified_frames)} bytes")
```

Figure 11. LSB Embedder



```python
def extract_data_from_wav(stego_path: str) -> list[bytes]:
    """Extract encrypted chunks from a WAV file."""
    with wave.open(stego_path, "rb") as stego:
        params = stego.getparams()
        if params.sampwidth != 2:
            raise ValueError("Only 16-bit WAV is supported.")
        frames = stego.readframes(params.nframes)

    samples = list(struct.unpack("<" + ("h" * (len(frames) // 2)), frames))

    num_chunks = 0
    for i in range(32):
        bit = samples[i] & 1
        num_chunks |= (bit << i)

    chunk_size = 256
    total_size = num_chunks * chunk_size
    data_bits = [samples[i] & 1 for i in range(32, 32 + total_size * 8)]

    data_bytes = bytearray()
    for i in range(0, len(data_bits), 8):
        byte = sum((data_bits[i + j] << j) for j in range(8))
        data_bytes.append(byte)

    chunks = [bytes(data_bytes[i:i+chunk_size]) for i in range(0, len(data_bytes), chunk_size)]
    return chunks
```

Figure 12. LSB Extractor

This section includes two functions designed for managing data embedding and extraction using the Least Significant Bit (LSB) technique. The first function, the LSB embedder, embeds the encrypted data chunks into the least significant bits of the carrier audio samples, ensuring that the modifications are imperceptible to human hearing. The second function, the LSB extractor, retrieves the embedded data from the carrier's LSBs, reconstructing the encrypted chunks for subsequent decryption. These functions work together to ensure secure and efficient data hiding and retrieval while maintaining the quality of the carrier audio.

V. TESTING AND ANALYSIS

The program could be executed by simply running the python file or clicking the run button on Visual Studio Code run button, make sure the required python and its libraries are installed.

In this simulation run we will be using two audio WAV files, carrier.wav and secret.wav

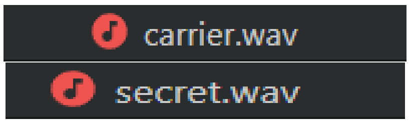
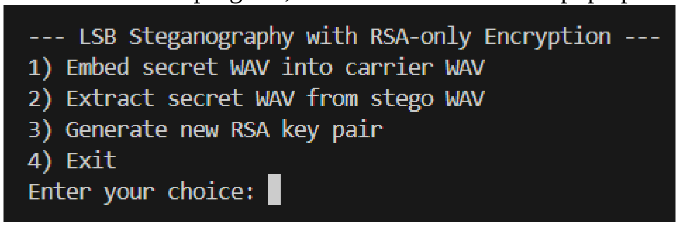


Figure 13. Simulation File

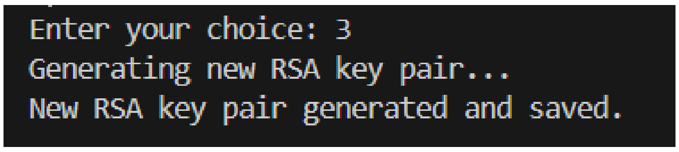
When we run the program, the main CLI menu will pop up



```
--- LSB Steganography with RSA-only Encryption ---
1) Embed secret WAV into carrier WAV
2) Extract secret WAV from stego WAV
3) Generate new RSA key pair
4) Exit
Enter your choice:
```

Figure 14. Main Menu CLI

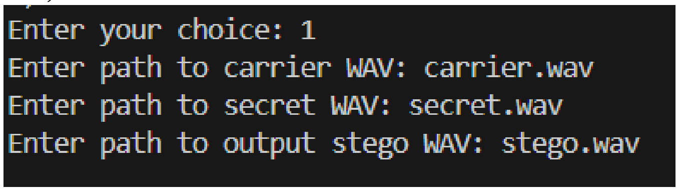
Inputting 3 will generate RSA key-pair to do encrypting and decrypting.



```
Enter your choice: 3
Generating new RSA key pair...
New RSA key pair generated and saved.
```

Figure 15. RSA key pair generated

Then we can start encrypting and embedding secret.wav into carrier.wav, inputting 2 followed up by the path of secret, carrier, and result file

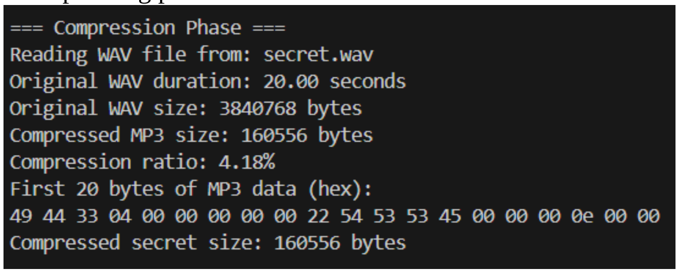


```
Enter your choice: 1
Enter path to carrier WAV: carrier.wav
Enter path to secret WAV: secret.wav
Enter path to output stego WAV: stego.wav
```

Figure 16. Inputting file path

Then it will go into three phases: compressing, encryption, and last embedding.

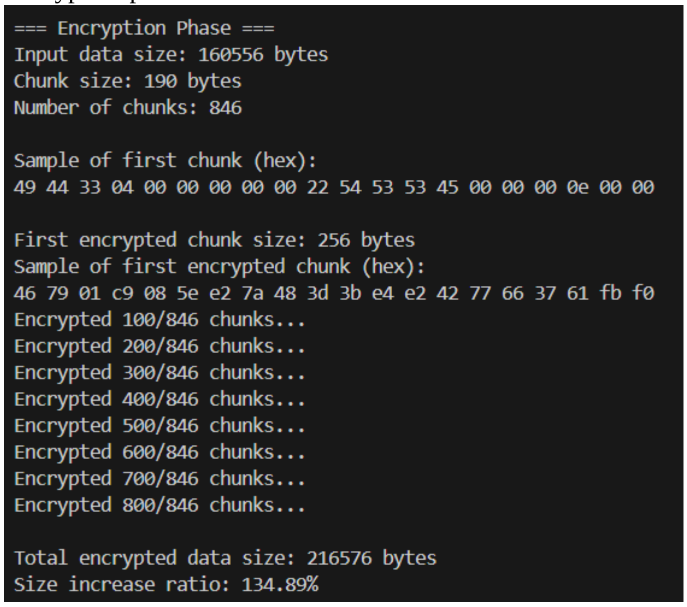
Compressing phase:



```
=== Compression Phase ===
Reading WAV file from: secret.wav
Original WAV duration: 20.00 seconds
Original WAV size: 3840768 bytes
Compressed MP3 size: 160556 bytes
Compression ratio: 4.18%
First 20 bytes of MP3 data (hex):
49 44 33 04 00 00 00 00 00 22 54 53 53 45 00 00 00 0e 00 00
Compressed secret size: 160556 bytes
```

Figure 17. Compressing phase

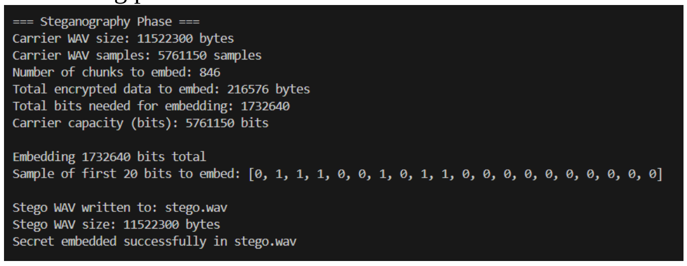
Encryption phase:



```
=== Encryption Phase ===
Input data size: 160556 bytes
Chunk size: 190 bytes
Number of chunks: 846

Sample of first chunk (hex):
49 44 33 04 00 00 00 00 00 22 54 53 53 45 00 00 00 0e 00 00

First encrypted chunk size: 256 bytes
Sample of first encrypted chunk (hex):
46 79 01 c9 08 5e e2 7a 48 3d 3b e4 e2 42 77 66 37 61 fb f0
Encrypted 100/846 chunks...
Encrypted 200/846 chunks...
Encrypted 300/846 chunks...
Encrypted 400/846 chunks...
Encrypted 500/846 chunks...
Encrypted 600/846 chunks...
Encrypted 700/846 chunks...
Encrypted 800/846 chunks...

Total encrypted data size: 216576 bytes
Size increase ratio: 134.89%
```

Figure 18. Encryption phase

Embedding phase:



```
=== Steganography Phase ===
Carrier WAV size: 11522300 bytes
Carrier WAV samples: 5761150 samples
Number of chunks to embed: 846
Total encrypted data to embed: 216576 bytes
Total bits needed for embedding: 1732640
Carrier capacity (bits): 5761150 bits

Embedding 1732640 bits total
Sample of first 20 bits to embed: [0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Stego WAV written to: stego.wav
Stego WAV size: 11522300 bytes
Secret embedded successfully in stego.wav
```

Figure 19. Embedding phase

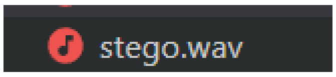
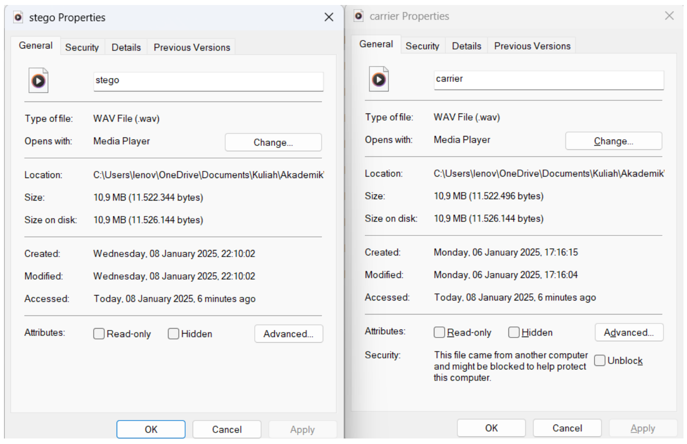


Figure 20. Embedding Result



stego Properties

General | Security | Details | Previous Versions

stego

Type of file: WAV File (.wav)
Opens with: Media Player

Location: C:\Users\lenov\OneDrive\Documents\Kuliah\Akademik
Size: 10,9 MB (11.522.344 bytes)
Size on disk: 10,9 MB (11.526.144 bytes)

Created: Wednesday, 08 January 2025, 22:10:02
Modified: Wednesday, 08 January 2025, 22:10:02
Accessed: Today, 08 January 2025, 6 minutes ago

Attributes: ☐ Read-only ☐ Hidden

carrier Properties

General | Security | Details | Previous Versions

carrier

Type of file: WAV File (.wav)
Opens with: Media Player

Location: C:\Users\lenov\OneDrive\Documents\Kuliah\Akademik
Size: 10,9 MB (11.522.496 bytes)
Size on disk: 10,9 MB (11.526.144 bytes)

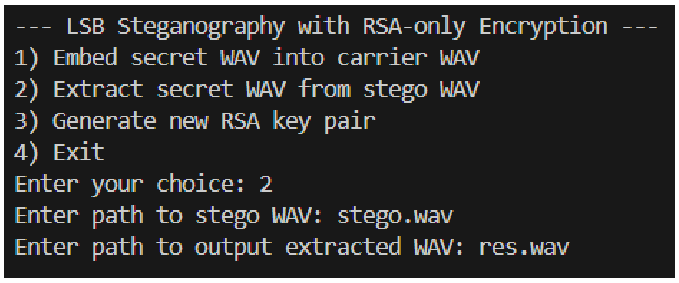
Created: Monday, 06 January 2025, 17:16:15
Modified: Monday, 06 January 2025, 17:16:04
Accessed: Today, 08 January 2025, 6 minutes ago

Attributes: ☐ Read-only ☐ Hidden
Security: This file came from another computer and might be blocked to help protect this computer. ☐ Unblock

Figure 21. Size after Embedding

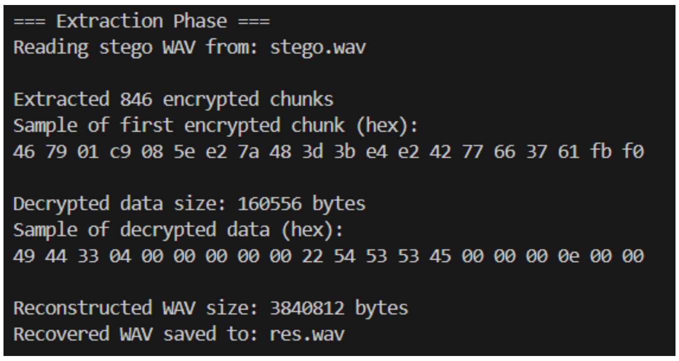
From Figure 21, we can see the size of the file didn’t change or the change is minor that it can be neglected, this makes the hidden file harder to detect.

Now we will try to extract the hidden data or the secret file that we just embedded.



```
--- LSB Steganography with RSA-only Encryption ---
1) Embed secret WAV into carrier WAV
2) Extract secret WAV from stego WAV
3) Generate new RSA key pair
4) Exit
Enter your choice: 2
Enter path to stego WAV: stego.wav
Enter path to output extracted WAV: res.wav
```

Figure 22. Extract menu



```
=== Extraction Phase ===
Reading stego WAV from: stego.wav

Extracted 846 encrypted chunks
Sample of first encrypted chunk (hex):
46 79 01 c9 08 5e e2 7a 48 3d 3b e4 e2 42 77 66 37 61 fb f0

Decrypted data size: 160556 bytes
Sample of decrypted data (hex):
49 44 33 04 00 00 00 00 00 22 54 53 53 45 00 00 00 0e 00 00

Reconstructed WAV size: 3840812 bytes
Recovered WAV saved to: res.wav
```

Figure 23. Extraction phase

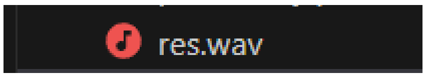


Figure 24. Extracted secret WAV file

For the fully and better demonstration, the link to video demonstration is in the attachment below.

VI. Conclusion

The system successfully implements LSB steganography combined with RSA encryption, achieving secure embedding and retrieval of audio data without compromising the quality of the carrier or the secret audio. The process of compressing the secret WAV file into MP3 reduced its size significantly, enabling efficient embedding, while RSA encryption ensured the confidentiality of the secret audio data. The embedded data was successfully extracted, decrypted, and reconstructed into its original WAV format, demonstrating the robustness and accuracy of the implemented methodology.

Despite its effectiveness, the system has room for improvement. The compression step, while efficient, relies on lossy MP3 encoding, which may lead to minor quality loss in the secret audio during reconstruction. Additionally, the size increases due to RSA encryption, with a ratio of 134.89%, could be optimized by exploring hybrid encryption methods, such as combining RSA with symmetric encryption for larger data sets. The current embedding process is dependent on the carrier WAV's capacity, which might limit its scalability for larger secret files.

The program also show weakness, because relying on RSA, encrypting a large data sets results in increased computational overhead and embedding size.

Future development could focus on optimizing the encryption algorithm and increasing the embedding efficiency to be able to handle large files without sacrificing performance.

VII. Acknowledgments

The completion of this paper could not have been possible without the aid of all IF1220 lecturers, especially Sir Ir. Rila Mandala, M.Eng., Ph.D. who has taught the K02 for the discrete mathematics. The writer has learned a hefty amount of information in the process of developing this paper.

References

[1] Dujella, A. (2021). *Number theory*. Zagreb, Croatia: Školska knjiga.

[2] Bansal, N. (2020). RSA encryption and decryption system. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 6(5).* https://doi.org/10.32628/CSEIT206520

[3] https://www.academia.edu/5285700/Prime_Numbers, accessed in 6 January 2025

[4] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/15-Teori-Bilangan-Bagian1-2024.pdf, accessed in 6 January 2025

[5] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/16-Teori-Bilangan-Bagian2-2024.pdf, accesed in 6 January 2025

[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/17-Teori-Bilangan-Bagian3-2024.pdf, accessed in 6 January 2025

Attachments

1. Video Makalah : https://linktr.ee/adharid1
2. LinkGithub : https://github.com/adharidwan/Makalah-IF1220

Statements

Hereby, I declare this paper I have written with my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 8 Januari 2025

Adha

Muhammad Adha Ridwan 13523098