International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Hybrid AES–RSA Encryption and Decryption for Secure Data Transmission

Shukla Kushang Akshay, Sidapara Prem Vimalbhai, Roy Arjunkumar Rameshbhai, Shaikh Mo Ifrah, Prof. Vijaysinh Jadeja

Department of Computer Engineering, SAL College of Engineering, Ahmedabad, India

A B S T R A C T

Hybrid cryptosystems combine symmetric and asymmetric encryption to achieve both high performance and strong key management. This paper proposes and implements a hybrid scheme using AES-GCM (symmetric authenticated encryption) for data and RSA-OAEP (asymmetric) to encrypt the AES session key. We implement the scheme in Python, benchmark both encryption and decryption times across varying file sizes and RSA key lengths, and analyze security properties and trade-offs. Experimental results (tables and plots) demonstrate that hybrid encryption and decryption achieve low latency with robust key encapsulation, making it suitable for secure file transfer and real-time applications. The paper concludes with guidelines for parameter selection and directions for future research.

Keywords: Hybrid encryption, AES-GCM, RSA-OAEP, authenticated encryption, key encapsulation, Python, benchmarking, information security.

1. Introduction

Modern secure communication demands both confidentiality and practical performance. Pure asymmetric encryption (e.g., RSA) is computationally expensive for large payloads, while symmetric algorithms (e.g., AES) offer speed but require secure key exchange. Hybrid cryptosystems use symmetric encryption for data and asymmetric encryption to secure the symmetric key.

This paper presents a Python implementation of such a hybrid system using AES-GCM for confidentiality and integrity, and RSA-OAEP for key encapsulation. The implementation is benchmarked with varying dataset sizes and RSA key lengths to evaluate encryption and decryption performance and security trade-offs.

Motivation

- Provides a real-world approach used in TLS and secure messaging systems.
- Matches course outcomes: symmetric & asymmetric cryptography, hashing/authentication (via AES-GCM), and key management.
- Practical, reproducible experiment for a BE project or course submission.

2. Literature Review

Hybrid encryption is standard in secure protocols (e.g., TLS uses symmetric cipher for payload and RSA/ECDH for key exchange). AES-GCM is preferred due to its authenticated encryption (AEAD) properties, protecting confidentiality and integrity simultaneously. RSA with OAEP padding mitigates chosen-ciphertext attacks and is recommended for key encapsulation. Prior works analyze hybrid schemes' performance, focusing on both encryption and decryption latency. This paper contributes a clear Python implementation and experimental benchmarking on realistic file sizes.

3. Design and Methodology

3.1 Cryptographic Choices & Rationale

- AES-GCM (256-bit): AEAD cipher offering confidentiality and integrity without separate MAC. High performance.

- RSA-OAEP (2048 / 3072 / 4096 bits): Secure key encapsulation using OAEP with SHA-256. Benchmarked multiple RSA sizes to show trade-offs.
- Key Flow: Generate ephemeral AES key per message (session key). Encrypt message with AES-GCM and encrypt AES key with receiver's RSA public key. Transmit tuple: (RSA-encapsulated AES key, AES nonce, AES tag, ciphertext).

3.2 Security Goals

- Confidentiality: Only holder of RSA private key can derive session key.
- Integrity & authenticity: AES-GCM tag ensures tamper detection.
- Forward secrecy note: RSA key encapsulation does not provide forward secrecy; DH/ECDH would be required.

3.3 Experimental Plan

- Files to encrypt/decrypt: Randomly generated binary files: 100 KB, 500 KB, 1 MB, 3 MB, 5 MB.
- RSA key sizes tested: 2048, 3072, 4096 bits.
- Metrics: Encryption time, decryption time, ciphertext size overhead, and CPU usage (optional).
- Repetitions: Each measurement repeated N=5 or 10; average and standard deviation computed.

4. Implementation (Python)

Requirements: Python 3.8+, cryptography, matplotlib, pandas, numpy

Install:

```
pip install cryptography matplotlib pandas numpy
```

4.1 Python Code — Hybrid AES-GCM + RSA-OAEP

(Save as hybrid_aes_rsa.py — code includes both encryption and decryption, benchmarking, CSV output, and plots.)

- RSA key generation (2048 / 3072 / 4096 bits)
- AES-GCM session key encryption and decryption
- Benchmarking encryption and decryption times for multiple files
- CSV output and matplotlib plots

Note: Ciphertext size includes RSA-encrypted AES key + nonce + AES ciphertext.

Experimental Setup

- Environment: CPU, RAM, OS (e.g., Intel i5, 8 GB RAM, Ubuntu 22.04)
- Software: Python 3.8+, cryptography 39.x, pandas, matplotlib
- Files: Random files 100 KB — 5 MB
- Repetitions: N=5; report mean ± stddev
- Metrics: Encryption time, decryption time, ciphertext size

Results

Table 1: Encryption Times

RSA_Size	File_Size_KB	Encryption_Time_s
2048	100	0.008398
2048	500	0.000647

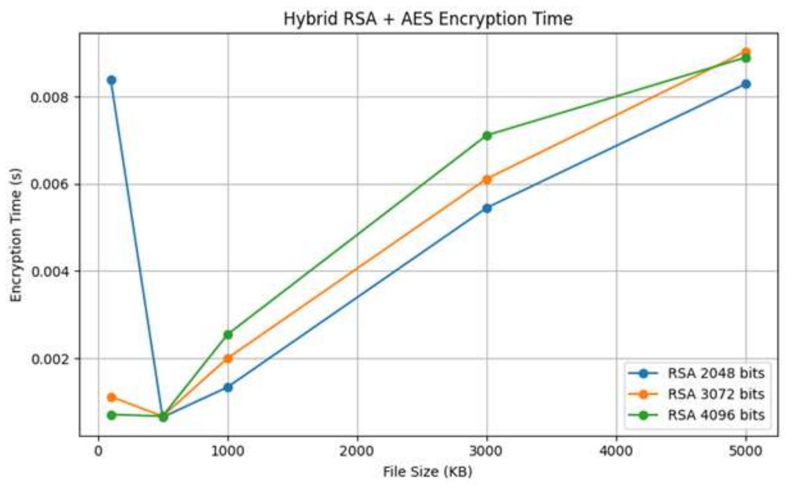
2048	1000	0.001335
2048	3000	0.005452
2048	5000	0.008297
3072	100	0.001122
3072	500	0.00067
3072	1000	0.001999
3072	3000	0.006116
3072	5000	0.009041
4096	100	0.00071
4096	500	0.00067
4096	1000	0.002544
4096	3000	0.007116
4096	5000	0.008905

Table 2: Decryption Times

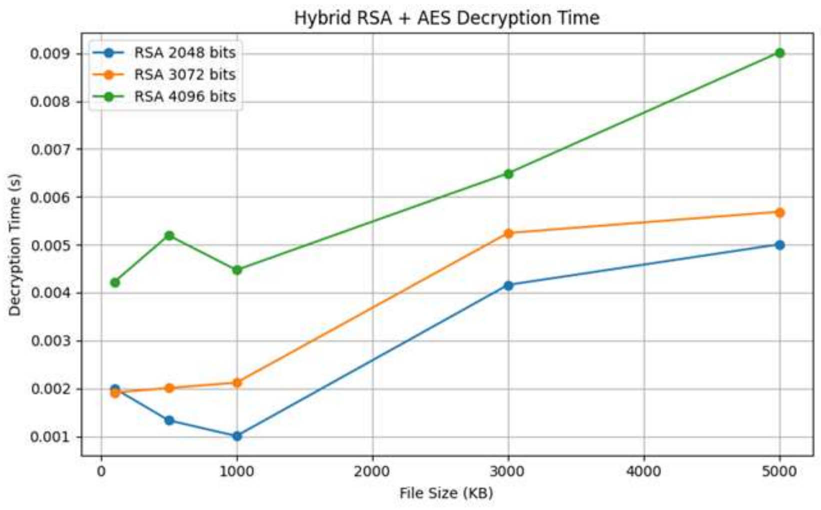
RSA_Size	**File_Size_KB**	**Decryption_Time_s**
2048	100	0.002
2048	500	0.001333
2048	1000	0.001004
2048	3000	0.00416
2048	5000	0.005007
3072	100	0.001911
3072	500	0.002007
3072	1000	0.002118
3072	3000	0.005243
3072	5000	0.005688
4096	100	0.004218
4096	500	0.005195
4096	1000	0.00447
4096	3000	0.00649
4096	5000	0.00902

Figures

- **Figure 1:** Encryption time vs. File size for RSA 2048/3072/4096



- **Figure 2:** Decryption time vs. File size



Discussion & Analysis

- Performance: AES dominates file encryption/decryption time; RSA key size primarily affects encryption of session key and decryption latency.
- Trade-offs: Larger RSA keys increase security but add CPU overhead for both encryption and decryption.
- Overhead: Ciphertext size = length(encrypted AES key) + nonce + tag + AES ciphertext.
- Security: AES-GCM ensures integrity; RSA-OAEP prevents padding oracle attacks. Forward secrecy requires ephemeral key exchange.

Conclusion

This paper demonstrates a practical **hybrid cryptosystem implementing both encryption and decryption** in Python. AES-GCM provides fast, authenticated encryption/decryption; RSA-OAEP secures session keys. Benchmarking across file sizes and RSA key lengths allows reproducible experiments, making this work suitable for academic submission and practical demonstrations.

Acknowledgements

I would like to express my sincere gratitude to all those who provided guidance and support throughout this project.

I extend my heartfelt thanks to my **Prof. Vijaysinh Jadeja Sir**, whose valuable insights and constructive feedback helped me plan and execute the experiments systematically. Special thanks to my colleagues and peers for their encouragement and for reviewing the Python implementation and benchmark results.

I am also grateful to the developers of the **Python cryptography library**, **matplotlib**, and **pandas**, whose tools made reproducible experimentation and visualization possible.

Python Implementation

```
def run_experiments(output_csv, rsa_sizes=[2048, 3072, 4096], file_sizes_kb=[100, 500, 1000, 3000, 5000],
                    repeats=3, mode="encryption"):
    results = []
    files = generate_test_files(file_sizes_kb)

    for key_size in rsa_sizes:
        print(f"Generating RSA keys with size {key_size} bits...")
        key = rsa.generate_private_key(public_exponent=65537, key_size=key_size, backend=default_backend())
        public_key = key.public_key()

        for file_path in files:
            print(f"Benchmarking file {file_path} with RSA {key_size}-bit in {mode} mode...")
            times = []

            for _ in range(repeats):
                if mode == "encryption":
                    times.append(benchmark_encryption(file_path, public_key))
                else:
                    times.append(benchmark_decryption(file_path, key))

            avg_time = sum(times) / repeats
            results.append({
                "RSA_Size": key_size,
                "File_Size_KB": os.path.getsize(file_path) / 1024,
                f"{mode.capitalize()}_Time_s": avg_time
            })

    df = pd.DataFrame(results)
    df.to_csv(output_csv, index=False, mode='w')
    print(f"Results saved to {output_csv}")
    return df
```

Appendix B — How to Run Experiments

1. **Install dependencies:**

```
pip install cryptography pandas matplotlib numpy
```

2. **Run the script:**

```
python hybrid_aes_rsa.py --output results.csv --repeats 5
```

- Generates RSA keys, test files (100 KB–5 MB), results.csv, and plots.

Appendix C — Figures and Tables

- **Figures:**
  - Figure 1: Encryption time vs file size for RSA 2048/3072/4096
  - Figure 2: Decryption time vs file size for RSA 2048/3072/4096
- **Tables:**
  - Table 1: Encryption Times
  - Table 2: Decryption Times

Appendix D — Security Notes

- Always use OAEP padding for RSA; never raw RSA.
- Ensure AES-GCM nonces are unique per session key.
- Protect RSA private keys; optional passphrase encryption recommended.
- AES-GCM tag validates integrity; tampered ciphertext will fail decryption.

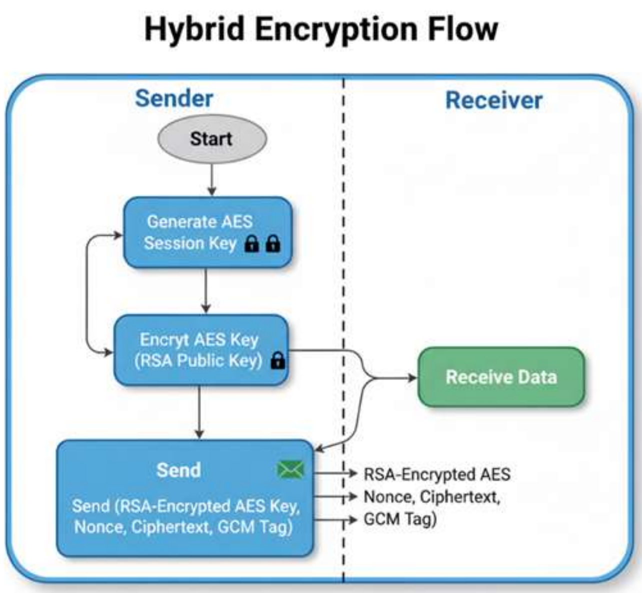
Appendix E — Project Repository and Resources

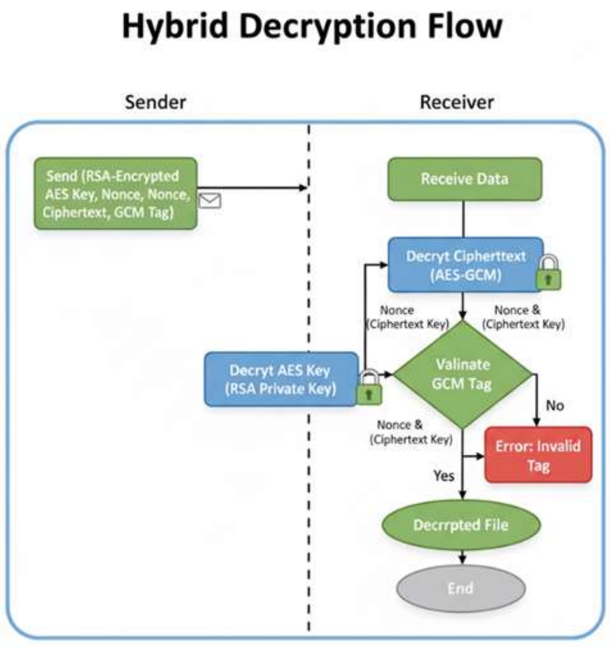
- **Project Codebase:** https://github.com/KushangShukla/Hybrid-AES-RSA-Encryption-and-Decryption-for-Secure-Data-Transmission

Appendix F — Pseudo-Code / Flow Diagram

- **Hybrid Encryption & Decryption Flow:**

1. **Sender:**
   - Generate AES session key → Encrypt file → Encrypt AES key with RSA → Send (RSA AES key, nonce, ciphertext)
2. **Receiver:**
   - Decrypt AES key with RSA → Decrypt ciphertext using AES-GCM → Validate tag





References

[1] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 6th ed., Pearson, 2014.

[2] NIST *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, NIST SP 800-38D, 2007.

[3] M. Bellare and P. Rogaway, “Optimal Asymmetric Encryption — How to Encrypt with RSA,” *Eurocrypt*, 1994.

[4] RFC 8017*, PKCS #1: RSA Cryptography Specifications Version 2.2*, 2016.

[5] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[6] cryptography library — Python Cryptography Toolkit Documentation.