A Cryptosystems Demonstration Using AES and RSA

Austin Howard
Dept. of Electrical and Computer Engineering
Southern Methodist University
Dallas, United States
howardad@smu.edu

CS 7349 – Data and Network Security
Section 400
Spring 2025
Dr. Chen Wang

***Abstract*— Cryptography plays a central role in modern digital security, with symmetric and asymmetric encryption systems forming the backbone of how information is protected. This project presents a working implementation of the Advanced Encryption Standard (AES) and the RSA public-key cryptosystem, both developed entirely in Python without external cryptographic libraries. The AES module performs 128-bit block encryption using custom implementations of SubBytes, ShiftRows, MixColumns, and key expansion logic. The RSA module covers the full asymmetric workflow, from key generation using predefined primes to encryption, decryption, and a brute-force attack designed to demonstrate the risks of weak key sizes. All components were built from scratch in accordance with course constraints, with output designed for reproducibility and clarity. The project effectively demonstrates how these systems operate programatically while highlighting their practical strengths and limitations.**

Keywords—AES, RSA, symmetric encryption, asymmetric encryption, cryptography, brute-force attack

I. Introduction

Cryptography is the backbone of secure digital communication today. Whether it's protecting sensitive health data, authenticating online banking, or simply securing a password, encryption plays a key role in keeping information protected. Most systems we use rely on either symmetric or asymmetric encryption. This project dives into one of each: the Advanced Encryption Standard (AES) and the RSA public-key cryptosystem.

A. Project Objectives

The project was designed with two separate but related implementations in mind. For the symmetric portion involving AES, the focus was on manually building out AES—no libraries allowed. AES-128 was selected because of its widespread use and structured round-based logic. For the asymmetric side, the RSA implementation required four distinct parts: a prime-checking tool, a key generator using specific prime indices, an encrypt/decrypt test using the message "rsa," and a brute-force simulation of how an attacker might break a poorly secured key.

B. Scope of Implementation

Both systems were developed from scratch using only Python's standard library. AES includes full block cipher functionality with test strings used to verify encryption and decryption. RSA uses small primes (an intentional decision to avoid slowdown) to make the attack demonstration viable in a reasonable time frame. Keys are generated using the 10th and 19th primes between 1000 and 10000. The RSA attack script simulates how an adversary might recover the private key d using only the public key and ciphertext. All outputs are clearly printed to console and marked for verification using [MATCH] and [MISMATCH] tags.

II. AES: Symmetric Encryption

The Advanced Encryption Standard (AES) is a symmetric block cipher formalized by NIST in 2001. AES operates on 128-bit blocks of plaintext and supports key sizes of 128, 192, or 256 bits [1]. The AES-128 variant—used in this implementation—performs ten rounds of transformation per block. Each round includes SubBytes, ShiftRows, MixColumns, and AddRoundKey operations [2]. Key expansion is handled before the first round, with each round receiving a unique subkey derived from the original.

A. Implementation

The AES system is structured into modular files, allowing for easy substitution, debugging, or integration. AES-128 was implemented manually using Python and consists of two core modules. The aes_cipher.py file contains the core encryption and decryption logic, including the S-box and inverse S-box tables, Galois Field multiplication routines, and round transformation logic. Key expansion is implemented directly in accordance with the AES standard using the Rcon array, RotWord, and SubWord functions.

```
for round_num in range(1, 10):
    state = sub_bytes(state)
    state = shift_rows(state)
    state = mix_columns(state)
    state = add_round_key(state,
                          round_keys[round_num])
```

Figure 1. Main AES encryption round transformation structure excluding the final round

Supporting functions for formatting, padding, and block processing are contained within aes_utils.py.

B. Demonstration

The aes_encrypt.py script accepts plaintext input and divides it into padded 16-byte blocks. From there, it encrypts each block using a static 128-bit key and writes the ciphertext to a binary file. Decryption is handled by aes_decrypt.py, which reads the binary file and applies the AES decryption routine to recover the original plaintext. All transformations are printed to the console in hex format to show the intermediate and final results.

C. Testing

The test_aes.py script automates round-trip encryption and decryption testing. Multiple test strings are passed through the AES pipeline, and the script verifies that the decrypted output matches the original input. For each result, the program prints a [MATCH] or [MISMATCH] tag, allowing for quick verification. Testing confirmed that the implementation behaves correctly across a variety of sample inputs.

III. RSA: Asymmetric Encryption

The RSA cryptosystem is a form of asymmetric encryption that uses two keys: one for encryption (the public key) and one for decryption (the private key) [3]. It is based on the mathematical difficulty of factoring large composite numbers and is commonly used for securing data transmissions, establishing trust in networked systems, and facilitating digital signatures. RSA encryption transforms plaintext into ciphertext using modular exponentiation, while decryption relies on knowledge of the private key and the underlying prime factors.

A. Key Generation

Key generation is handled in generate_keys.py. The script begins by identifying the 10th and 19th prime numbers between 1000 and 10000, which are then assigned as p and q. These are multiplied together to form the modulus n. From this, Euler's totient function φ(n) is calculated as $\varphi(n) = (p - 1)(q - 1)$. A common exponent, e = 65537, is used. The private exponent d is then derived as the modular inverse of e with respect to φ(n) [4]. The script outputs all relevant values for visibility: p, q, n, φ(n), the public key (e, n), and the private key (d, p, q).

B. Demonstration

The rsa_test.py script performs a full encrypt/decrypt demonstration. The input message is hardcoded as "rsa" and is converted to a list of integers in the range 0–25 using a simple letter-to-number mapping (e.g., a = 0). Each value is encrypted using the formula $c \equiv m^e \bmod n$ and then decrypted using $m \equiv c^d \bmod n$. Once decrypted, the numeric values are mapped back to characters and compared to the original message. If the message is recovered successfully, a [MATCH] tag is printed to the console.

C. Attack Simulation

The rsa_attack.py script simulates an attacker attempting to recover the private key using a brute-force approach. Without access to φ(n), the attacker cannot compute the modular inverse of e directly. Instead, the script performs an exhaustive search for d by iterating through possible values and testing whether each candidate successfully decrypts the known ciphertext back to the original message. Because the RSA key uses small primes by design, the brute-force attack completes quickly and successfully identifies the correct value of d. The script prints both the recovered key and the elapsed time, illustrating the vulnerability of weak RSA parameters.

```
for guess_d in range(2, n):
    decrypted = [
        pow(c, guess_d, n)
        for c in ciphertext
    ]
    if decrypted == expected_plaintext:
        recovered_d = guess_d
        break
```

Figure 2. Exhaustive brute-force search for the private exponent d using only public key inputs

IV. Evaluation

Both the AES and RSA modules performed as intended during testing. The AES implementation was validated using multiple test strings passed through the full encryption and decryption pipeline. For each case, the system produced the original plaintext after decryption, with [MATCH] tags printed to confirm success. The AES test script demonstrated that the transformation rounds and key expansion logic behaved consistently across varying input.

The RSA demonstration followed a similar approach. A known plaintext message was encrypted using the generated public key and decrypted using the corresponding private key. The final result matched the original input, confirming the correctness of the RSA key generation, encryption, and decryption logic.

A. Brute-Force Simulation Results

The RSA attack module successfully recovered the private key d using an exhaustive search. This was only possible because small primes were intentionally used to reduce the size of n, making brute-force attempts computationally feasible within a fraction of a second (as tested on the system the project was coded on). The script displayed the recovered key and total time taken. The simulation highlights how simplified RSA implementations lack protections like mutual authentication and challenge-response [5].

B. Compliance with Constraints

All functionality was implemented from scratch using only Python's standard library. No third-party cryptographic libraries or external modules were used. Prime number checking, modular inverse computation, key expansion, and Galois Field arithmetic were coded manually. Each part of the project maps directly to the assignment's structural requirements, with testing outputs designed to visibly demonstrate correctness and integrity.

V. Conclusion

This project served as a practical demonstration of two foundational cryptographic systems: the Advanced

Encryption Standard (AES) and the RSA public-key cryptosystem. Both were implemented from scratch using only Python's standard library, without the use of external cryptographic packages or other APIs (such as RSA's own Python Library). In each case, the goal was not just to replicate expected encryption behavior but to understand how these systems function internally.

The AES module included a full implementation of 128-bit block encryption, with support for key expansion and round-based transformation logic. Testing confirmed that the cipher correctly encrypted and decrypted multiple inputs and that the internal operations behaved as expected. The RSA module covered all stages of the asymmetric workflow, from prime generation and key creation to encryption, decryption, and a working brute-force attack against a weak keypair. This attack simulation highlighted RSA's reliance on key size as a safeguard against an attacker's ability to discover private keys.

Completing this project afforded a greater understanding of the underlying mathematical and logical foundations in cryptography, from modular arithmetic to Galois Field operations. This was facilitated through both project requirements and its emphasis on reproducible output. The final result is a working, well-structured codebase that meets the project's requirements and illustrates the strengths and vulnerabilities of each cryptographic system.

VI. References

[1] National Institute of Standards and Technology, "FIPS PUB 197: Advanced Encryption Standard (AES)," Nov. 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[2] C. Wang, "Lecture 5 – Symmetric Encryption," Slide 65, CS 5/7349 Data and Network Security, Southern Methodist University, Spring 2025. [Lecture notes; accessed via Canvas]

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[4] C. Wang, "Lecture 6 – Public Key Cryptography," Slide 25, CS 5/7349 Data and Network Security, Southern Methodist University, Spring 2025. [Lecture notes; accessed via Canvas]

[5] C. Wang, "Lecture 8 – Authentication," Slide 15, CS 5/7349 Data and Network Security, Southern Methodist University, Spring 2025. [Lecture notes; accessed via Canvas]