3 Applied Cryptography – RSA Encryption and Secure Communication

Delve into applied cryptography by implementing a secure communication system based on RSA public-key encryption. The goal is to create a working software prototype that can generate cryptographic keys, encrypt and decrypt messages, and possibly integrate symmetric encryption for efficient data transfer. The core of the project is the RSA cryptosystem, which relies on fundamental number-theoretic algorithms. RSA is a widely used public-key scheme for secure data transmission, based on the mathematical fact that while it is easy to multiply two large primes, it is extremely difficult to factor their product. Implement RSA key generation (including prime number generation and computing modular inverses), encryption and decryption functions, and then use these to secure a simple message exchange or data storage system. This project is mathematically rigorous: it will require understanding modular arithmetic, prime factorization, and the properties of Euler's totient function. Teams must ensure the correctness of their implementation (for example, verifying that encryption and decryption are inverses under the generated keys) and consider efficiency (large integer arithmetic and optimization for large primes). Optionally, the project can incorporate a symmetric cipher (such as AES) to encrypt bulk data, with RSA used to exchange the symmetric key - mimicking real-world secure communication protocols. The final system should demonstrate the entire pipeline of secure communication: key generation, encryption, transmission (simulated), and decryption, all implemented by the team (preferably in C++).

3.1 Required Components and Algorithms:

1. RSA Key Generation: Implement the RSA key generation algorithm, ensuring the following criteria are met:

    - Prime numbers are generated on the order of at least 16 bits for demonstration, and preferably 1024 bits or higher for actual security
    - A probabilistic primality test (such as Miller-Rabin) or a library routine is used to test primality.
    - The output keys should be a public key and private key
    - Ensure to handle big integers for all these computations (use C++ multiple-precision libraries such as boost::multiprecision::cppint if needed, or implement basic big integer support).

2. RSA Encryption/Decryption: Implement the RSA cryptographic functions (this will require efficient modular exponentiation – use fast exponentiation by squaring to handle large exponents).

3. Include proper padding or format for messages if dealing with text strings (for simplicity, you may choose a straightforward encoding like converting

bytes to integers without complex padding schemes, but should be aware of security implications).

4. Symmetric Encryption Integration (Optional): To simulate a full secure communication session, implement a symmetric-key cipher for the message payload:

    - Use a well-known algorithm like AES (Advanced Encryption Standard) for encrypting the actual message data under a random symmetric key. AES is a standard symmetric cipher widely used to secure data. You can implement a simplified version of AES or use a provided library for AES if low-level implementation is too time-consuming, but you must understand and explain its working.
    - Encrypt the symmetric key itself with RSA (this is how hybrid encryption is done: RSA secures the small key, AES secures the bulk data). The receiver will use RSA to decrypt the symmetric key, then use that key to decrypt the message.

5. Secure Communication Protocol: Develop a simple protocol or application scenario to demonstrate the system:

    - For example, a command-line program where one user can generate a key pair, publish the public key, and another user can use it to encrypt a message. The encrypted message is then transmitted (simulated via a file or console output) and the recipient uses their private key to decrypt it.
    - Alternatively, implement a client-server simulation where the server holds a public/private key pair and clients send encrypted messages to it.

6. Ensure the protocol includes steps for key exchange (or public key distribution), encryption of data, and decryption. Although network programming is not required (the exchange can be simulated by function calls or file writing), the roles of sender and receiver should be clearly defined in the code structure.

7. Validation and Testing: Provide extensive testing for each component:

    - Test the correctness of the extended Euclidean algorithm.
    - Test RSA encryption/decryption on known small examples (where p, q, e, d are manually computed) to validate that decrypting an encrypted message returns the original.
    - If possible, test the system with various message sizes and contents (alphabetic, binary data) to ensure reliability. Also test the scenario with the optional symmetric cipher to verify that the hybrid encryption works (the final plaintext after full decryption matches the original).

8. Analyze the performance: for example, measure how encryption/decryption time grows with key size or message size, and ensure it is manageable for demonstration purposes.

9. Security Considerations: Though this is primarily an implementation project, students should be aware of basic security considerations. Include in the documentation a brief discussion on:

   - Why RSA is secure (related to difficulty of prime factorization) and any assumptions (e.g., the keys generated are large enough to prevent brute force).
   - The importance of using padding schemes in real applications (though not mandatory to implement PKCS1 padding here, acknowledging it is good practice).
   - The difference between public-key and symmetric-key encryption and why both are used (highlight the efficiency of symmetric ciphers like AES for bulk data.
   - Any limitations of your implementation (for instance, if the system is not resistant to certain attacks due to simplified padding or smaller key sizes used in testing).

3.2 Final Deliverables:

1. Implemented Software: A fully working cryptographic program (preferably in C++). This should be provided as source code in the Git repository, and optionally as a compiled binary for convenience. The program's functionality must include:

   (a) RSA key generation (outputting public/private key pairs).

   (b) RSA encryption and decryption (with appropriate input/output format for messages).

   (c) If included, symmetric encryption/decryption and the combined usage with RSA for key exchange.

   (d) The software should be usable via a simple interface (command-line interface is acceptable), where a user can follow steps to generate keys, encrypt a message, and decrypt a message.

2. Git Repository (if Git is used): The repository containing all source code, with a clear structure (separating modules for RSA, symmetric cipher, etc.). The commit history should indicate the development timeline and contributions. Include a README.md with instructions on how to build and run the program, as well as sample usage. Any external libraries or resources should be acknowledged and included (or installation instructions given).

3. Technical Documentation: A comprehensive report (ca. 10 pages) describing the project. Key sections should include:

   (a) Introduction: Overview of RSA and cryptographic objectives of the project, including the problem statement in context (e.g., "securely transmitting a message over an insecure channel using our implemented cryptosystem").

   (b) Mathematical Background: Explanation of the mathematical foundations of RSA (prime numbers, modular arithmetic, one-way functions) in a way suitable for a second-year CS student. Cite the fundamental fact that RSA's security comes from factoring difficulty. If AES or another cipher was used, briefly explain symmetric vs. asymmetric encryption

   (c) Design and Implementation: Description of how the key generation was implemented (how primes are chosen/tested), how encryption and decryption are implemented (including any optimizations like fast modular exponentiation). Include pseudocode or flowcharts for the main algorithms. If a symmetric cipher is implemented, outline its operation (for AES, discuss the round structure at a high level).

   (d) Results: Present the results of testing the system. This includes examples of encryption/decryption (show a plaintext, the ciphertext produced, and the plaintext recovered). Also discuss performance results: for instance, how long key generation took for certain key sizes, how encryption time grows with message size, etc. If any limitations were encountered (e.g., inability to use extremely large keys due to time constraints), note them here.

   (e) Security Analysis: A short discussion on the security of the implemented system. Emphasize that while the project's RSA keys might not be large enough for real security (if that's the case), the implementation is conceptually correct. Mention any potential vulnerabilities (e.g., lack of proper random padding, which in real systems could lead to attacks). This section shows awareness of real-world cryptographic practices and where the project stands in relation.

   (f) Conclusion: Summarize what was learned and any future work (for example, how one could extend the project to a fuller system, use larger keys, or implement additional features like digital signatures).

4. User Guide: Include a brief user manual, either as a section in the report or a separate document, that explains how to run the program. This should have step-by-step instructions for generating keys and performing encryption/decryption, so the graders or instructors can easily verify the functionality. Include examples with actual command inputs and outputs for clarity.

5. Presentation/Demo (optional): Prepare the necessary materials (slides, and a live demo script). The presentation should focus on the problem

addressed, the approach taken, and a highlight of results (e.g., showing that your system successfully encrypted and decrypted a sample message).

3.3 Language and Tools Requirements:

- Programming Language: Implementation should be in C++ (preferred) or another low-level language like C. High-level languages like Python are not encouraged for the cryptographic computations, because managing big integers and performance-intensive tasks in Python would be too slow and abstracted. In C++, you may use libraries like boost::multiprecision for big integers or GMP (GNU Multiple Precision Arithmetic Library) if needed, but the core logic of RSA should be written and understood by the team (do not use any library's built-in RSA functions).

- Precision and Arithmetic: Since RSA deals with large integers, ensure your environment can handle integers larger than the standard 64-bit types. The use of cppint from Boost or Bigint libraries is acceptable. All random number generation for primes should use a cryptographically secure random source if possible (e.g., std::randomdevice or relevant library functions).

- Development Practices: You are encouraged to utilize Git for version control. Given the security-critical nature of cryptographic code, use Git to track all changes carefully—this also aids in code reviews. Teams should conduct mutual code reviews to catch logical errors (which can be common in complex math code). Unit tests for mathematical functions (gcd, mod inverse, primality test) should be written and run regularly.

- Testing Tools: It may be useful to incorporate existing test vectors for RSA (to ensure your implementation matches known outputs for small primes). Use debugging tools to step through key generation and encryption steps if results are not as expected. Memory checking tools (like Valgrind) are important to ensure no leaks or buffer overruns, especially when dealing with manual arrays or byte buffers for encryption.

- Documentation and Formatting: All code should be written in a clear, maintainable style. Given the formal nature of this project, proper inline documentation (comments explaining non-obvious sections of code, especially mathematical computations) is required. The final write-up should be prepared in a professional format, suitable for inclusion in a course syllabus or report compendium. Use of LaTeX for typesetting mathematical formulas in the report is encouraged (but not required) to maintain precision and clarity in notation.