Yggdrasil: Secure File Transfer System Midterm Report

Mert Bozkir
mert.bozkirr@gmail.com

April 26, 2025

Abstract

This report details the midterm progress of the Yggdrasil project, a secure file transfer system developed in Python. The primary goal is to create a robust application ensuring confidentiality, authentication, and integrity during network file exchange, while also incorporating low-level network protocol understanding through manual IP header manipulation. Key features implemented to date include a TCP client-server framework, RSA-based client authentication via challenge-response, secure AES key exchange using RSA encryption, and AES-GCM encryption for file data transmission. The system also includes basic packetization, checksum validation for encrypted payloads, and initial ACK/NACK handling. Docker support is provided for ease of deployment and testing. This report outlines the system architecture, details the implementation of current features, discusses challenges faced, and maps out the future work required to meet all project objectives.

1 Introduction

Secure data transmission is paramount in modern networking. The Yggdrasil project aims to address this need by developing a secure file transfer system from the ground up. This system is designed not only to provide practical security features but also to serve as an educational tool, requiring exploration into lower-level network operations.

The core problem Yggdrasil solves is the secure transfer of files between two endpoints over an potentially insecure network. Key objectives, derived from the project requirements outlined in `docs/.cursorrules` and `docs/stages.md`, include:

- **Functionality:** Reliable file sending and receiving over TCP sockets.
- **Security:** Implementing robust client authentication (RSA), data confidentiality (AES-GCM), and data integrity (SHA-256).
- **Low-Level Networking:** Manual manipulation and processing of IP headers (flags, TTL, checksum, fragmentation) using Scapy.
- **Performance Analysis:** Measuring and analyzing metrics like latency, bandwidth, and packet loss under various conditions.
- **Security Analysis:** Validating security measures against simulated attacks like MITM and packet injection.

This midterm report details the progress made towards these objectives, focusing on the implemented architecture, core transfer mechanisms, security features, and Dockerization. It also discusses the current system status, challenges encountered, and the planned steps for future development.

2 System Design and Architecture

Yggdrasil employs a standard client-server architecture communicating over TCP sockets. Python (version 3.11+) was chosen as the primary development language due to its extensive libraries for networking and cryptography.

The system is divided into a client (`src/client.py`) responsible for initiating connections and sending files, and a server (`src/server.py`) responsible for listening for connections, authenticating clients, and receiving files. Common configurations and utility functions are shared via `src/config.py` and `src/utils.py`, respectively.

Key technologies and libraries used include:

- **Python Standard Library:** Primarily the `socket`, `struct`, `os`, `hashlib`, and `time` modules.
- **Cryptography:** The `cryptography` library (version >= 44.0.2) is used for all cryptographic operations, including RSA key generation, signing/verification, encryption/decryption, AES key generation, and AES-GCM authenticated encryption/decryption.
- **Scapy:** Planned for use in later stages for low-level packet manipulation, as indicated by its inclusion in `pyproject.toml` and presence of scripts like `scripts/send_manipulated_packet.py`.
- **uv:** Used as the Python package installer and virtual environment manager, specified in `pyproject.toml`.
- **Docker:** Utilized for containerization, with configuration defined in the `Dockerfile`.

Project dependencies are managed using `uv` via the `pyproject.toml` file.

2.1 Packet Structure

A custom packet header is defined in `src/config.py` to manage the file transfer process. The header format (`HEADER_FORMAT = "!II16sB"`) translates to:

- Sequence Number (32-bit unsigned integer, network byte order): Identifies the packet order.
- Data Length (32-bit unsigned integer, network byte order): Specifies the length of the subsequent payload data.
- Checksum (16 bytes): MD5 hash of the (potentially encrypted) payload data, used for basic corruption detection (`utils.calculate_checksum`).
- Flags (8-bit unsigned integer): Indicate packet type or status, such as `FLAG_FIN` (final packet), `FLAG_ACK` (acknowledgment), or `FLAG_NACK` (negative acknowledgment).

The total header size (`HEADER_SIZE`) is calculated based on this format. Packets are constructed using `utils.create_packet` and parsed using `utils.unpack_header`. Note that the checksum is currently calculated over the *encrypted* payload before transmission.

3 Implementation Details

This section details the core components implemented so far.

3.1 Core File Transfer

The basic file transfer relies on standard TCP socket programming.

- **Connection:** The client establishes a TCP connection to the server at the host and port specified in `config.py`. The server listens for and accepts incoming connections. Basic socket timeouts (`SOCKET_TIMEOUT`) are implemented.
- **Framing:** Before sending variable-length data like cryptographic elements (signatures, keys) or filenames/hashes, the data's length is packed into a 4-byte network-order integer and sent first, followed by the data itself. The receiver uses this length prefix to know how many bytes to expect. This is handled by `utils.send_data_with_length` and `utils.receive_data_with_length`.
- **File Handling:** The client reads the input file in chunks defined by `CHUNK_SIZE`. Each chunk is encrypted and encapsulated in a data packet. The server receives these packets, decrypts them, and writes the data sequentially to an output file in the `OUTPUT_DIR`.
- **Termination:** The client sends a final packet with the `FLAG_FIN` flag set after sending all file data. The server acknowledges this FIN packet.
- **ACK/NACK:** Basic ACK/NACK handling is implemented. The server sends ACKs for successfully received and processed packets. The client currently handles ACKs to advance its conceptual sliding window (`utils.SlidingWindow`). NACKs are sent by the server on checksum failures or decryption errors. The client includes logic to retransmit packets upon receiving a NACK or experiencing a timeout (`client.handle_ack`, `client.run_client` loop). A simple retransmission limit (`MAX_RETRIES`) is used.

3.2 Security Mechanisms

Security is implemented in stages: authentication, key exchange, and data encryption. Keys are generated using `scripts/generate_keys.py` and loaded using `utils.load_private_key` and `utils.load_public_key`.

- **Authentication:** A challenge-response mechanism using RSA signatures ensures client identity.
    1. Server generates a random challenge (`os.urandom(AUTH_CHALLENGE_SIZE)`) and sends it to the client.
    2. Client signs the challenge using its private RSA key (`utils.rsa_sign` with PSS padding).
    3. Client sends the signature back to the server.
    4. Server verifies the signature against the original challenge using the client's public key (`utils.rsa_verify` with PSS padding). If valid, authentication succeeds.
- **AES Key Exchange:** A shared secret (AES key) for bulk data encryption is established securely.
    1. Client generates a random AES key (`utils.generate_aes_key`, size `AES_KEY_SIZE`).
    2. Client encrypts the AES key using the server's public RSA key (`utils.rsa_encrypt` with OAEP padding).
    3. Client sends the encrypted AES key to the server.
    4. Server decrypts the AES key using its private RSA key (`utils.rsa_decrypt` with OAEP padding).
    5. Server sends confirmation (`AES_KEY_OK`) back to the client.
- **Data Encryption:** File chunks are encrypted before transmission using AES in GCM mode, providing both confidentiality and authenticity.
    1. For each chunk, the client generates a unique nonce (`os.urandom(AES_NONCE_SIZE)`).
    2. The client encrypts the chunk using the established AES key and the nonce (`utils.aes_encrypt`). AES-GCM produces ciphertext and an authentication tag.
    3. The nonce is prepended to the ciphertext+tag before being sent as the packet payload.
    4. The server receives the payload, extracts the nonce, and decrypts the ciphertext using the AES key and nonce (`utils.aes_decrypt`). AES-GCM automatically verifies the authentication tag during decryption; an invalid tag raises an exception (`InvalidTag`), which triggers a NACK.
- **Integrity (Initial):** Before transfer, the client calculates the SHA-256 hash of the entire file (`utils.calculate_file_hash`) and sends it to the server along with the filename. After receiving the FIN packet and assembling the file, the server calculates the hash of the received file and compares it to the expected hash. The result (pass/fail) is logged.

3.3 Dockerization

A `Dockerfile` is provided to containerize the application. It uses a Python 3.11 base image provided by Astral (specifically 'ghcr.io/astral-sh/uv:python3.11-bookworm-slim'), installs system dependencies (`libpcap-dev` for Scapy), installs project dependencies using `uv pip install -system`

., and copies the source code (`src/`), scripts (`scripts/`), and keys (`keys/`) into the container image. Instructions for building the image and running the client and server containers (including network configuration for macOS using Docker Desktop or Colima) are detailed in the `README.md`. This simplifies setup and ensures a consistent runtime environment.

4 Current Status and Results

As of this midterm report, the following core functionalities are implemented and operational:

- Successful TCP client-server connection establishment.
- RSA key generation and loading.
- RSA-based challenge-response authentication.
- Secure AES key exchange via RSA encryption.
- Transmission of filename and SHA-256 file hash.
- AES-GCM encryption and decryption of file chunks during transfer.
- Basic packet framing with custom headers (sequence number, length, checksum, flags).
- MD5 checksum calculation and verification on the encrypted payload.
- Basic ACK/NACK handling for flow control and error notification (checksum/decryption failure).
- Basic retransmission mechanism based on timeouts and NACKs.
- Server-side file reassembly and final integrity check using the received SHA-256 hash.
- Docker containerization for both client and server.

Preliminary tests involving transferring text files locally between the client and server, both run directly and via Docker containers (using 'host.docker.internal' or the Colima IP), have been successful, demonstrating the functionality of the implemented security layers and basic transfer protocol.

5 Future Work

Significant development is planned to complete the project's objectives. The next phases of work will focus on the following key areas:

- **Robust Flow Control:** Implementing a more sophisticated sliding window protocol (`utils.SlidingWindow` exists but needs full integration and testing) to handle network congestion and optimize throughput.
- **Low-Level IP Manipulation:** Integrating Scapy (using logic from `scripts/send_manipulated_packet.py` and `scripts/sniff_packets.py`) to manually modify IP headers (TTL, flags, fragmentation) and validate checksums at the IP layer.

- **Network Performance Measurement:** Conducting systematic evaluations using tools like `iPerf` and `tc` to measure latency, bandwidth, and the impact of simulated packet loss, comparing performance across different network conditions (e.g., Wi-Fi vs. wired).
- **Security Analysis and Validation:** Performing packet captures with Wireshark to verify encryption effectiveness in transit and simulating attacks (MITM, packet injection) to test the resilience of the implemented security measures.
- **Comprehensive Testing:** Developing thorough unit tests for critical components (e.g., security functions, packet handling) and integration tests to ensure reliable end-to-end system operation under various conditions.
- **Documentation Refinement:** Enhancing code comments and finalizing a comprehensive project report detailing the system design, implementation specifics, performance results, and security analysis outcomes.
- **Final Review and Polish:** Conducting a final review of all project components (code, tests, documentation) to ensure all requirements are met before submission.

6 Conclusion

The Yggdrasil project has made substantial progress towards creating a secure file transfer system. The core architecture is established, and critical security features including RSA-based authentication, secure AES key exchange, and AES-GCM data encryption are functional. Basic file transfer, packet handling, and Dockerization have also been successfully implemented. The focus moving forward will be on enhancing the robustness of the transfer protocol (flow control, error handling), implementing low-level IP header manipulation, and conducting thorough performance and security analyses as outlined in the project requirements. The project is currently on track to meet its objectives for the final submission.