BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FILE TRANSFER OVER A COVERT ICMP CHANNEL

PŘENOS SOUBORŮ PŘES SKRYTÝ ICMP KANÁL

AUTHOR **MICHAL REPČÍK**
AUTOR PRÁCE

BRNO 2025

REPČÍK, Michal. *File Transfer over a Covert ICMP Channel.* Brno, 2025. . Brno University of Technology, Faculty of Information Technology.

Contents

Chapter 1

Overview

This documentation describes a client/server application that transfers encrypted files using ICMP/ICMPv6 Echo-Request/Response packets.

Features

- AES encryption (key derived from the user's login).
- File chunking to fit within the standard MTU (1500 bytes).
- Reliable transfer ensured by a custom protocol.
- Support for both IPv4 and IPv6.
- Implemented in C++ with OpenSSL and libpcap [7].

System Requirements

- GNU/Linux environment (tested on Debian and WSL2 with GCC 12.5.0 and GNU Make 4.3).
- Root privileges (required for raw packet capture and injection).
- OpenSSL and libpcap libraries installed.
- C++17 or higher (uses `std::variant` and `std::filesystem`).
- `GNU Make` 3.8 or higher (required for advanced wildcards and pattern rules in the default Makefile).

Chapter 2

Usage

This chapter explains how to build and run the application, including installation requirements, client commands, server commands, and usage examples.

2.1 Installation

1. Download the project or clone the git repository:

   ```
   git clone https://github.com/rm-a0/isa-encrypted-icmp
   cd isa-encrypted-icmp
   ```

2. Ensure all dependencies are installed:

   - C++17 or higher (compiler support for `std::variant` and `std::filesystem`).
   - OpenSSL development libraries.
   - libpcap development libraries.
   - `GNU Make` **3.8 or higher** (required for advanced wildcards and pattern rules in the default Makefile).

3. Build the project (run from the project root):

   ```
   make
   ```

4. The executable `secret` will be available in the project root directory after a successful build.

2.2 Exectuion

Syntax

```
secret -r <file> -s <ip|hostname> [-l]
```

Parameter Description

- `-r <file>` : specify the file to transfer; can be relative or absolute path (e.g., `-r ../test.file`).
- `-s <ip|hostname>` : IP address or hostname of the target machine to which the file will be sent.
- `-l` : if this flag is set, the program runs in *server* mode (listen) — the program listens on the network, receives incoming ICMP packets, reconstructs and decrypts the file.

Notes

- Raw packet operations require superuser privileges — run the program with `sudo` (e.g., `sudo ./secret ...`).
- The `-s` parameter accepts both IPv4 and IPv6 addresses as well as hostnames; the program automatically switches to ICMPv6 when using an IPv6 address.
- File paths can be relative or absolute; relative paths are resolved from the current working directory.
- If a file with the same name already exists on the server, it will be overwritten.

Examples

Starting the server (listening mode)

```
# run the application as a server
sudo ./secret -l
```

Sending a file (client mode)

```
# send test.txt to the machine at 192.168.1.10
sudo ./secret -r test.txt -s 192.168.1.10

# send a binary file to a hostname
sudo ./secret -r ../video.mp4 -s server.example.com
```

Chapter 3

Implementation Details

This chapter provides an in-depth look at the implementation of the `secret` application, including its code structure, key classes and namespaces, custom protocol, error handling, and design choices.

3.1 Code Structure

The application is implemented in C++17 and organized into modular source files, each handling a specific aspect of functionality. The code adheres to good practices with consistent formatting, meaningful variable names, and doxygen style comments in each header file. The source files are:

- `main.cpp`: Entry point, parses arguments, and initializes client or server mode.
- `arg_parser.cpp`: Handles command-line argument parsing and validation.
- `file_handler.cpp`: Manages file reading, writing, and path processing.
- `encoder.cpp`: Implements AES-256-CBC encryption and decryption using OpenSSL [6].
- `chunker.cpp`: Splits files into chunks and reassembles them.
- `protocol.cpp`: Defines the custom protocol for packet structure and serialization.
- `icmp_connection.cpp`: Manages ICMP/ICMPv6 socket creation and packet sending [3, 4].
- `net_utils.cpp`: Provides utility functions for address resolution and checksum calculation [2, 8, 4, 3].
- `client.cpp`: Implements client logic for file encryption, chunking, and transmission.
- `server.cpp`: Implements server logic for packet capture, processing, and file reconstruction [7].

3.2 Custom Protocol

The application uses a custom protocol to ensure reliable file transfer over ICMP/ICMPv6 [5, 1]. The protocol defines two packet types: `METADATA` and `DATA`, encapsulated within ICMP Echo Request/Reply packets.

Packet Structure

Each packet has the following fields:

- **Magic Number** (4 bytes): A fixed value (`0xDEADBEEF`) to identify valid packets.
- **Version** (1 byte): Protocol version, set to 1.
- **Packet Type** (1 byte): `0` for `METADATA`, `1` for `DATA`.
- **Sequence Number** (4 bytes): Ensures correct packet ordering, starting at 0 for `METADATA` and incrementing for each `DATA` packet.
- **Client ID** (8 bytes): A randomly generated identifier to distinguish packets from different clients.
- **Payload**: Either `Metadata` or `Data`, serialized as follows:
  - `Metadata` contains:
    - Filename length (1 byte)
    - Filename (variable length)
    - File size (4 bytes)
    - Total chunks (4 bytes)
    - AES initialization vector (16 bytes)
  - `Data` contains:
    - Chunk number (4 bytes)
    - Chunk payload (up to 1472 bytes).

Transfer Process

1. The client sends a `METADATA` packet with sequence number 0, containing the filename, encrypted file size, total number of chunks, and AES IV.
2. The client sends `DATA` packets with incrementing sequence numbers, each containing a chunk of the encrypted file.
3. The server captures packets using libpcap, filters for ICMP/ICMPv6, and stores them in a thread-safe queue.
4. A consumer thread processes packets, grouping them by client ID and sequence number. Once all chunks are received (verified using `totalChunks`), the server reassembles and decrypts the file.

Reliability

The protocol assumes no packet loss, as specified in the task. Sequence numbers ensure correct packet ordering, and the client ID separates packets from multiple clients. Network byte order (`htonl`, `ntohl`, `htobe64`, `be64toh`) is used for portability across architectures.

3.3 Design Choices and Assumptions

- **Hardcoded Login**: The encryption key is derived from the hardcoded login `xrepcim00` using SHA-256, as the task specifies using the user's login but does not require dynamic input.
- **File Overwriting**: If a file with the same name exists on the server, it is overwritten to simplify implementation, as the task does not specify handling naming conflicts.
- **Filename in Metadata**: The filename sent in the `METADATA` packet is the same as the name of the file being sent, extracted from the file path using `file_handler::getNameFromPath`.
- **Encryption Before Chunking**: The entire file is encrypted with AES-256-CBC using a single initialization vector (IV) before being split into chunks, ensuring consistent encryption across all chunks [6].
- **No Packet Loss**: The protocol assumes no packet loss, omitting retransmission mechanisms, as allowed by the task.
- **IPv4/IPv6 Detection**: `net_utils` automatically detects IPv4 or IPv6 addresses using `inet_pton` and `getaddrinfo` for robust resolution [3, 4].

Chapter 4

Additional Functionality and Limitations

4.1 Multiple Client Support

The server uses a thread-safe design to process packets from multiple clients:

- Each client generates a unique 64-bit `clientId` using `std::mt19937_64`, included in every packet.
- The server maintains a map (`clientPackets`) that groups packets by `clientId`, ensuring packets from different clients are processed independently.
- A consumer thread, protected by a mutex and condition variable, processes packets from a queue, allowing concurrent packet capture and processing.
- Once all chunks for a client (determined by `totalChunks` in the `METADATA` packet) are received, the server reassembles and decrypts the file, then clears the client's data to free memory.

This design ensures scalability and prevents interference between concurrent file transfers.

4.2 Limitations

The application has the following limitations, particularly in real-world scenarios:

- **Large File Transfers and Packet Loss**: The protocol assumes no packet loss, as specified by the task. In real-world scenarios where packet loss occurs, large file transfers may fail to complete, resulting in no file being created on the server, as the server requires all chunks (verified by `totalChunks`) to reconstruct the file.
- **Memory Usage**: Since the entire file is encrypted and decrypted as a whole (using a single IV), the server must hold the reassembled encrypted file in memory before decryption. Additionally, the `clientPackets` map stores all packets for each client in memory until the transfer is complete. This can lead to significant memory usage, especially when multiple clients send large files concurrently, potentially exhausting the server's RAM.

- **Client ID Collisions**: Although the 64-bit `clientId` generated by `std::mt19937_64` minimizes the risk, there is a theoretical possibility of two clients generating the same ID (due to the birthday paradox). This could cause packets from different clients to be misattributed, leading to corrupted or mixed file transfers. The large 64-bit space makes this unlikely but not impossible.

Chapter 5

Testing

The application was tested to ensure correct functionality and compatibility with the reference environment.

5.1 Manual Testing

- **Client-Server Transfer**: Tested on Debian and WSL2 by sending text and binary files (e.g., `test.txt`, `video.mp4`) to localhost and remote IPs (IPv4 and IPv6). Files were successfully transferred, decrypted, and saved with correct contents, using the filename from the `METADATA` packet.
- **Error Handling**: Invalid inputs (e.g., missing `-r`, non-existent files, invalid IPs) were tested, confirming appropriate error messages on `stderr`.
- **Multiple Clients**: Multiple clients were run simultaneously, sending files to the server, which correctly reconstructed each file using `clientId`.

5.2 Wireshark Testing

The application was tested using Wireshark to verify ICMP/ICMPv6 packet structure and content [7]. Captured packets confirmed correct `METADATA` and `DATA` packet formats, proper sequence numbering, and inclusion of the `0xDEADBEEF` magic number.

5.3 Reference Environment

The application was compiled on `merlin.fit.vutbr.cz` using GCC 12.5.0, OpenSSL 3.0.17, and GNU Make 4.3, ensuring portability and compliance with the task's reference environment. The application was tested on WSL2 and Debian GNU/Linux.

Bibliography

[1] Conta, A.; Deering, S. and Gupta, M. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification* online. Internet Engineering Task Force, March 2006. Available at: https://datatracker.ietf.org/doc/html/rfc4443. [cit. 2025-09-30].

[2] Hall, B. J. *Beej's Guide to Network Programming: IP Addresses, versions 4 and 6* online. 2025. Available at: https://beej.us/guide/bgnet/html/#ip-addresses-versions-4-and-6. [cit. 2025-09-30].

[3] Pages, L. M. *Ip(7) - Linux man page* online. 2025. Available at: https://man7.org/linux/man-pages/man7/ip.7.html. [cit. 2025-09-30].

[4] Pages, L. M. *Ipv6(7) - Linux man page* online. 2025. Available at: https://man7.org/linux/man-pages/man7/ipv6.7.html. [cit. 2025-09-30].

[5] Postel, J. *Internet Control Message Protocol* online. Internet Engineering Task Force, September 1981. Available at: https://datatracker.ietf.org/doc/html/rfc792. [cit. 2025-09-30].

[6] Project, O. *EVP Symmetric Encryption and Decryption* online. 2023. Available at: https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption. [cit. 2025-09-30].

[7] Project, T. *Libpcap Documentation* online. 2025. Available at: https://www.tcpdump.org/pcap.html. [cit. 2025-09-30].

[8] Repcik, M. *IPK L4 Scanner: utils.cpp* online. GitHub, 2025. Available at: https://github.com/rm-a0/ipk-l4-scanner/blob/main/src/utils.cpp. [cit. 2025-09-30].