LTCE

SOFTWARE ENGINEERING LABORATORY

Experiment 2

Software Requirements Specification (SRS) Document

IEEE Standard Format

Group Members:

1. **Raj Dubey** - Roll No: 11
2. **Yash Maurya** - Roll No: 25
3. **Piyush Kanojiya** - Roll No: 23

Subject: Software Engineering

College: Lokmanya Tilak College Of Engineering (LTCE)

University: Mumbai University

Academic Year: 2025-26

Date: August 11, 2025

AIM AND OBJECTIVES

🎯 Primary Aim:

To develop a comprehensive Software Requirements Specification (SRS) document following IEEE standard format for the Packet Sniffer case study.

📋 Specific Objectives:

- Create IEEE 830-1998 compliant SRS document
- Define functional and non-functional requirements
- Specify system interfaces and constraints
- Document use cases and system behavior

SOFTWARE REQUIREMENTS SPECIFICATION

Software Requirements Specification

for

Packet Sniffer Tool

Version 1.0 approved

Prepared by

Raj Dubey (11)

Piyush Kanojiya (23)

Yash Maurya (25)

Table of Contents

Beyond the core functional and nonfunctional requirements:

The packet sniffer must adhere to several additional constraints that govern its legal, technical, and international usage. One key requirement is internationalization. The user interface text must be abstracted into a translation-ready format, allowing for easy conversion into other languages without changing source code.

Legal considerations must be embedded into the software. Developers must ensure the tool complies with local and international cybersecurity and data monitoring laws. It must not contain any backdoors, packet injection routines, or exploit scripts. Any attempt to use the tool beyond its intended purpose should be traceable and clearly disallowed.

From a reuse perspective, the software architecture should be modular, enabling its core

packet capture and protocol analysis components to be extracted and integrated into other network tools. These components should follow open standards and support API documentation for external use.

Lastly, compatibility with both IPv4 and IPv6 networks must be guaranteed. This means capture filters, parsers, and logging mechanisms must correctly recognize and log modern network packets, including fragmented IPv6 segments. Testing must confirm accurate behavior in both stacks.

1. Introduction

1.1 Purpose

The purpose of this Software Requirements Specification (SRS) document is to define the complete functional and non-functional requirements for the Packet Sniffer Tool project. This tool is designed to capture, analyze, and log network packets in real-time to aid in network diagnostics, security analysis, and protocol learning. It will be used primarily by students, educators, developers, and network engineers seeking a user-friendly, high-performance, and modular network monitoring application.

This document outlines the scope of the software, specifies the features to be implemented, describes the system environment, and defines how the software will interact with hardware and other applications. It aims to provide a comprehensive reference for both developers and stakeholders, ensuring all parties have a shared understanding of what the software will achieve. The Packet Sniffer Tool described here will be developed following the Incremental Process Model to accommodate modular delivery and early testing.

The SRS serves as a binding agreement between the client (in this academic case, the course instructor or project mentor) and the development team (students or developers). It will be used throughout the project lifecycle as a roadmap to measure implementation progress, validate functionality, and ensure all required components are delivered successfully.

1.2 Document Conventions

This document adheres to a structured format where each major section and subsection is numerically labeled. Requirements are denoted using identifiers such as REQ-1.1, REQ-2.2, etc., to ensure easy traceability during design, implementation, and testing phases. Functional requirements are listed in section 4 and are categorized by system features.

Text formatting conventions include:

- **Bold** for section titles and identifiers.
- *Italics* for special terms or technical jargon.
- Monospaced font for code references or commands (e.g., `tcpdump`).

All diagrams, tables, and bullet lists follow consistent indentation and style for readability. Priority levels are indicated using keywords High, Medium, or Low, and are discussed within their corresponding feature sections. Where information is incomplete, placeholders marked TBD (To Be Determined) are used and collected in Appendix C.

1.3 Intended Audience and Reading Suggestions

This document is intended for multiple audiences who are directly or indirectly involved in the Packet Sniffer Tool's development, deployment, or usage. The primary readers include:

- **Developers:** to understand system features and implementation goals.
- **Testers:** to derive test cases from the requirements.
- **Instructors/Supervisors:** to validate scope and academic progress.
- **End-users (e.g., students or network learners):** to understand the application's features and constraints.

Readers unfamiliar with the system should begin with Section 1 (Introduction) to understand the motivation and scope, then proceed to Section 2 for a high-level overview. Developers may skip directly to Sections 3 and 4, which contain technical interface specifications and feature breakdowns. Non-functional and legal constraints are covered in Section 5 onward.

Diagrams and appendices provide additional clarity and should be reviewed during design and validation stages.

1.4 Product Scope

The Packet Sniffer Tool is a software application designed to provide visibility into real-time network traffic. Its primary goals are to capture packets traversing a network interface, decode common protocols (e.g., TCP, UDP, ICMP), apply filtering rules, and display/log the relevant data in an accessible format. It offers a mix of command-line and graphical interfaces depending on the development phase.

The tool is targeted for academic environments where students can learn about packet structures, protocol behavior, and basic network security. It also has utility in basic enterprise network diagnostics. The design will support modular extension, such as adding support for new protocols, output formats, or visualization tools.

Key benefits of the project include: enhancing student understanding of low-level networking, providing a simple alternative to complex tools like Wireshark, and enabling portable, open-source

network diagnostics. It aligns with institutional goals of promoting hands-on, experiential learning in software and systems engineering.

1.5 References

This document draws upon several key resources that inform its structure and content:

- **Karl Wiegers' SRS Template (1999):** The foundational structure of this document, with permission for academic use.
- **Scapy Documentation (https://scapy.net/):** Reference for Python packet manipulation and capture APIs.
- **Wireshark User Guide:** Concepts related to packet capture, filtering, and analysis workflows.
- **IEEE Std 830-1998:** IEEE Recommended Practice for Software Requirements Specifications.
- **Mumbai University Software Engineering Curriculum (2025-26):** Project-based learning outcomes and deliverables.

Each of these references supports the project's technical implementation, design philosophy, or academic alignment. URLs and version numbers are recorded where possible to enable verification and access by readers and reviewers.

2. Overall Description

2.1 Product Perspective

The Packet Sniffer Tool is envisioned as a standalone application designed for educational, diagnostic, and exploratory network analysis. It is not an extension of any existing software suite, though it follows familiar paradigms used by tools like Wireshark or tcpdump. Its architecture is modular and layered, separating packet capture, decoding, filtering, logging, and presentation components. This ensures that enhancements, fixes, or experiments in one module do not affect others.

As a self-contained tool, it interacts directly with the host machine's network interfaces via system-level packet capture APIs, but does not depend on external databases, cloud services, or enterprise backends. The user initiates and controls all operations, meaning the tool functions independently without persistent internet connectivity. This supports its portability and safe usage within isolated or testbed environments.

In a typical system, the tool will reside alongside other utilities used for software development or

network monitoring, but does not require integration or interoperation. However, its logs and output formats are deliberately compatible with other tools such as Excel, ELK stack, and Wireshark, allowing users to post-process captured data externally if needed. A high-level block diagram would show inputs from the OS's network layer feeding into the tool's capture engine, flowing through protocol parsing and filters, then into both GUI display and file-based logging subsystems.

2.2 Product Functions

At its core, the Packet Sniffer Tool offers a suite of functions focused on capturing, analyzing, and managing network traffic. Each of these functionalities is developed as an increment in the chosen Incremental Process Model to support modularity and gradual enhancement.

The primary functions include:

- **Real-time packet capture:** Intercepting raw Ethernet frames or IP packets from a selected network interface.
- **Protocol analysis:** Parsing headers for TCP, UDP, ICMP, and displaying key fields like IP addresses, ports, flags, and checksums.
- **Filtering system:** Enabling user-defined filters based on IP, protocol, port, and packet size to focus on relevant traffic.
- **Data logging:** Persistently storing packet information in structured file formats (CSV/JSON) with timestamping and log rotation.
- **Graphical User Interface (GUI):** Visualizing captured packets, filter options, and system status in a responsive interface.

Each function contributes to a layered workflow, progressing from data acquisition to interactive analysis and export. These are designed to work asynchronously to ensure real-time performance without blocking the UI or causing data loss.

2.3 User Classes and Characteristics

This tool is intended to serve a range of user classes, each with distinct needs and technical skill levels:

- **Undergraduate Students (Primary Users):** Typically enrolled in software or computer networks courses, these users require a simplified, intuitive interface to understand basic packet structures and traffic behavior. They may have limited command-line experience, so the GUI must abstract complexity while retaining control.
- **Instructors and Teaching Assistants:** These users require repeatable and reliable output to demonstrate concepts, generate assignment datasets, and validate student submissions. They may use the CLI for automation and appreciate features like filtering and export customization.
- **Advanced Users (Network Enthusiasts or Developers):** These users may explore performance benchmarking, protocol debugging, or tool extension. They will benefit from the open-source nature, documentation, and modularity of the system, often diving deeper into

logs and configurations.

Each class varies in their frequency of use, expected outputs, and error tolerance. The interface and feature access will be consistent across users, but documentation will address needs of each group explicitly.

2.4 Operating Environment

The tool must function on standard personal computing hardware running either Microsoft Windows (Windows 10 or later) or popular Linux distributions (such as Ubuntu 20.04+ or Fedora). The minimum system requirements include:

- A dual-core CPU
- At least 4 GB RAM
- Access to a functional network interface card (NIC)

The software will be developed using Python 3.10 or later, and will rely on third-party libraries such as `scapy` for low-level packet handling, `tkinter` or `PyQt` for GUI design, and `logging` for file operations. The host OS must allow raw socket access, which typically requires elevated permissions.

Dependencies will be bundled or documented clearly for installation. On Windows, `Npcap` must be installed in WinPcap compatibility mode. On Linux, `libpcap` and proper permissions for raw network access must be ensured. Virtual environments will be used to isolate Python dependencies.

2.5 Design and Implementation Constraints

Due to its development as part of an academic curriculum, the Packet Sniffer Tool must be completed within one semester, meaning strict time and resource constraints apply. The codebase must remain simple and maintainable, making Python a suitable language due to its readability, extensive libraries, and rapid development cycle.

Additional constraints include:

- **Technology constraint:** Only open-source tools and libraries may be used, ruling out commercial components.
- **Platform constraint:** Must support both Windows and Linux environments.
- **Security constraint:** Must not perform or suggest packet manipulation, only capture.
- **Compliance constraint:** All development must follow university academic integrity policies.

Design choices must reflect these boundaries. For instance, multithreading will be implemented using Python's standard `threading` module rather than advanced concurrency models. Cross-platform compatibility will be prioritized, avoiding OS-specific functions unless alternatives exist.

2.6 User Documentation

The software will include a set of user-facing documentation to support onboarding, training, and problem resolution. This includes:

- **User Manual:** A PDF or HTML file explaining installation, basic operations, and GUI components with screenshots.
- **Quick Start Guide:** A short walkthrough that enables users to run the sniffer and view sample results within five minutes.
- **Contextual Help:** Tooltips and hover guides embedded into the GUI.
- **Troubleshooting Appendix:** Common error messages, causes, and fixes.

All documentation will use plain language and follow a modular format to allow easy referencing. Advanced users may consult a separate API reference or developer notes for extending the tool's functionality. Localization of documentation may be added in later releases.

2.7 Assumptions and Dependencies

Several assumptions are made in this SRS which, if changed, may impact the feasibility or performance of the project:

- It is assumed that the user will have the necessary permissions (admin/root) to access low-level packet capture interfaces.
- The host machine will have internet access for initial library installations.
- Users are aware of legal and ethical considerations and will use the tool responsibly.

Dependencies include the presence of specific Python libraries (`scapy`, `tkinter` or `PyQt`, `pandas`) and system-level capture drivers (`Npcap` on Windows or `libpcap` on Linux). These components must be properly installed and accessible. The application depends on accurate system time for timestamping and assumes that network interface cards are functioning and exposed to the OS-level APIs.

A change in any of these assumptions—e.g., restricted permissions or missing dependencies—could affect how the application behaves. Therefore, configuration checks and fallback logic should be included during initialization.

3. External Interface Requirements

3.1 User Interfaces

The user interface for the Packet Sniffer Tool is a critical component that bridges raw data with user comprehension. It must be intuitive, responsive, and provide actionable insights in real time. The tool will begin as a command-line interface (CLI) during early development stages to quickly

prototype functionality, and evolve into a graphical user interface (GUI) using frameworks like Tkinter or PyQt by the final increment.

The GUI will consist of several structured regions:

- A toolbar containing controls such as Start, Stop, Save Log, and Settings
- A live packet table showing timestamp, source/destination IP, protocol, port, and packet size
- A filter panel allowing users to define rules for capturing specific traffic based on IP, port, protocol, or custom expressions
- A logging panel or status bar displaying real-time status messages and errors

GUI responsiveness is paramount, especially when processing large traffic volumes. It must support sorting, filtering, and highlighting rows. Standard UI/UX guidelines must be followed, such as accessibility (color contrast, keyboard navigation), system-native icons, and minimal user clicks. Keyboard shortcuts (e.g., Ctrl+S for Save) and tooltips should also be included. Screen layout must adapt across standard screen resolutions.

A separate GUI specification may define mockups, color palettes, and widget behavior in more detail. In the CLI mode, a user menu will provide similar features with typed commands and console logs. Both UI layers must follow consistency in feature access and terminology.

3.2 Hardware Interfaces

The hardware interface requirement focuses on how the software will interact with physical components of the host machine, primarily the network interface card (NIC). This card acts as a conduit through which all incoming and outgoing packets flow. The software must access this NIC via platform-specific APIs such as WinPcap/Npcap on Windows or libpcap on Linux.

During initialization, the application must detect all active NICs and present them to the user for selection. This includes physical adapters (Ethernet, Wi-Fi) and virtual interfaces (loopback, VPN adapters). The NIC name, description, and MAC address should be displayed to aid identification. Only one interface may be active at a time, and changes must trigger re-initialization of the capture engine.

The software must handle different NIC types gracefully. For example, Wi-Fi adapters may drop broadcast packets in monitor mode, while loopback interfaces may require special privileges on Linux. Device enumeration must use system calls with fallbacks for compatibility.

No direct control or configuration of hardware (e.g., toggling interfaces or altering routing) is permitted by the tool. It is a passive sniffer and must operate without altering hardware state or interfering with normal operation.

3.3 Software Interfaces

The Packet Sniffer Tool must interface with a variety of software components that either provide

low-level access or enable higher-level functionality. These software dependencies must be clearly defined to ensure compatibility, ease of installation, and proper operation.

Key components include:

- **Operating System APIs:** On Windows, interaction with Npcap will occur via WinPcap-compatible APIs. On Linux, libpcap functions will be invoked.
- **Python Libraries:** The tool will rely on `scapy` for crafting and decoding packets, `logging` for file-based logs, `tkinter` or `PyQt` for GUI elements, and `threading` for concurrency.
- **Data Storage:** Logs will be written in structured CSV or JSON files, which can be opened using third-party applications like Excel, Logstash, or pandas.

These interfaces must be loosely coupled to allow future replacement. For example, the logging module should be replaceable without affecting the capture engine. Proper error checking and exception handling should wrap all interface points to detect failures like permission issues, missing libraries, or unsupported OS versions.

APIs must be documented internally, and version compatibility must be tracked. The application should warn users if required libraries are missing or outdated. Dependencies must be declared in `requirements.txt` or equivalent.

3.4 Communications Interfaces

The packet sniffer is inherently a listening tool—it passively observes and analyzes data flowing through the network stack. It does not initiate any communications on its own. However, it must be capable of understanding and correctly interpreting various communication protocols to extract useful insights.

Supported protocols include Ethernet, ARP, IP, TCP, UDP, ICMP, and DNS. The software must detect these based on header signatures and extract relevant fields such as source/destination addresses, port numbers, flags, and payload data. Higher-level decoding (e.g., HTTP headers) may be supported later.

All interpreted packets must retain their metadata such as timestamp, interface ID, and direction (inbound/outbound if discernible). For security reasons, the application must never forward or rebroadcast captured data. Users may export data through controlled logging formats only.

The application must support reading and parsing of communication streams using standard formats. Packet formatting should follow industry specifications, and decoding errors should be logged without crashing the application. Future versions may support exporting logs in PCAP format for interoperability.

4. System Features

4.1 Core Packet Capture Engine

4.1.1 Description and Priority

The core packet capture engine is the foundational feature of the Packet Sniffer Tool. It is responsible for interfacing with the system's network interfaces to capture all packets passing through the selected adapter. This engine will run in a dedicated thread or background process to ensure uninterrupted and real-time acquisition of traffic, regardless of the volume or type.

Priority: High. Without this feature, the tool cannot function as a packet sniffer. It forms the technical basis upon which all other components—protocol analysis, filtering, logging, GUI—rely. The implementation must be reliable, resource-efficient, and capable of running continuously without crashing or consuming excessive CPU and memory.

The capture engine must provide raw packet access in a structured format for further parsing. It must also include logic to discard malformed or unsupported packets gracefully. In later increments, enhancements such as buffer resizing, timestamp calibration, and optional capture filters may be integrated directly at this level.

4.1.2 Stimulus/Response Sequences

When the user selects a network interface and clicks the 'Start Capture' button, the system initiates a background process to begin listening on that interface. As packets are received, the system immediately queues them for parsing and display.

The user may stop the capture at any time by clicking 'Stop', which halts the capture thread and flushes buffers. In CLI mode, typed commands like `start eth0` or `stop` trigger equivalent actions. Status updates are shown on the UI or console in real-time.

4.1.3 Functional Requirements

Requirement ID	Description	Priority
REQ-1.1	The system shall detect and list all active network interfaces at startup	High
REQ-1.2	The system shall allow the user to select one network interface for monitoring	High

REQ-1.3	The system shall begin packet capture in less than 3 seconds after user initiation	High
REQ-1.4	The capture engine shall support capturing at least 1000 packets per second with zero packet loss under normal system load	High
REQ-1.5	The system shall timestamp each packet with system time accurate to milliseconds	High

4.2 Protocol Analysis Engine

4.2.1 Description and Priority

The protocol analysis engine is the component responsible for parsing each captured packet and decoding its protocol-specific headers. It enables the software to interpret raw bytes into human-readable values such as source IP, destination port, TCP flags, and so on. This parsing is essential for both visualization and filtering.

Priority: High. Without proper protocol decoding, the user cannot make sense of captured traffic. The engine must support a wide range of standard protocols including Ethernet, ARP, IPv4, IPv6, TCP, UDP, ICMP, and optionally DNS and HTTP headers. Parsing must be performed efficiently to avoid UI lag or buffer overflow.

The engine must be modular and extensible, so that support for new protocols can be added easily without affecting the core logic. For example, HTTP parsing can be layered on top of TCP. Parsing failures (e.g., corrupted headers) must be logged and bypassed, not cause system failures.

4.2.2 Stimulus/Response Sequences

As each packet is captured, it is passed to the protocol analysis engine. The engine determines the packet type by inspecting the header fields and then decodes protocol-specific data. The decoded information is stored in structured objects or dictionaries and handed to the GUI or logger.

The user sees the parsed fields rendered in the packet viewer. For example, for a TCP packet, the source and destination ports, sequence number, and flags will be extracted and shown.

4.2.3 Functional Requirements

Requirement ID	Description	Priority

REQ-2.1	The system shall decode and display headers for Ethernet, ARP, IP, TCP, UDP, and ICMP protocols	High
REQ-2.2	The parser shall extract and display source IP, destination IP, ports, protocol type, and packet size	High
REQ-2.3	The system shall highlight packets with invalid checksums or malformed headers	Medium
REQ-2.4	The protocol parser shall operate asynchronously from capture to avoid blocking	High
REQ-2.5	The system shall allow future support for additional protocols like HTTP or DNS without breaking core functionality	Medium

4.3 Filtering and Search System

4.3.1 Description and Priority

The filtering system allows users to focus on specific network traffic by applying rules based on various packet attributes. This is essential for educational use where students need to isolate particular protocols, IP addresses, or communication patterns for analysis.

Priority: Medium. While not essential for basic packet capture, filtering greatly enhances usability and learning outcomes by reducing information overload.

4.3.2 Functional Requirements

Requirement ID	Description	Priority
REQ-3.1	The system shall allow filtering by source and destination IP addresses	Medium
REQ-3.2	The system shall allow filtering by protocol type (TCP, UDP, ICMP)	Medium
REQ-3.3	The system shall allow filtering by port numbers	Medium

REQ-3.4	The system shall provide a search function to find specific packets in captured data	Low

4.4 Data Export and Logging

4.4.1 Description and Priority

Data export functionality enables users to save captured packet information for offline analysis, reporting, or integration with other tools. This feature supports the educational mission by allowing students to analyze data outside the capture session.

Priority: Medium. Essential for academic use where analysis and reporting are required.

4.4.2 Functional Requirements

Requirement ID	Description	Priority
REQ-4.1	The system shall export captured data to CSV format	Medium
REQ-4.2	The system shall export captured data to JSON format	Medium
REQ-4.3	The system shall include timestamps in all exported data	High
REQ-4.4	The system shall provide optional PCAP format export for Wireshark compatibility	Low

5. Other Nonfunctional Requirements

5.1 Performance Requirements

The packet sniffer tool must operate efficiently under various network load conditions, ensuring that it is capable of capturing and analyzing packets in real-time without introducing performance bottlenecks or resource exhaustion. At a baseline, the software should be able to process at least 1000 packets per second, regardless of the protocol mix or packet size variations. This requirement

ensures that the application is usable in high-traffic academic or small enterprise environments. Developers must consider buffering strategies, memory management, and threading techniques to meet this goal.

Real-time display of packet data must occur with a GUI update latency of no more than 200 milliseconds per packet. This target ensures that the user experiences minimal delay between when a packet is captured and when it is shown on-screen. Implementing efficient data queuing and UI thread synchronization is critical to achieving this responsiveness. The packet capture engine must perform with deterministic behavior under various stress levels to maintain consistency in testing and deployment.

The application must initialize and begin capturing within 3 seconds of launch on systems that meet the minimum hardware specifications. Fast startup is essential for usability and contributes to the impression of software reliability. Developers should defer non-essential processes to background threads and avoid blocking calls during initialization. Logging and configuration loading should be optimized to prevent delays.

Stress testing must verify system stability when subjected to bursts of traffic, large packet payloads, and malformed data. Performance metrics must be logged for analysis, including memory usage, CPU consumption, packet processing rate, and GUI responsiveness. This data will guide optimization efforts and ensure the application performs predictably and efficiently under real-world conditions.

5.2 Safety Requirements

While the packet sniffer tool is not categorized as a safety-critical system, its misuse or malfunction can result in unintended consequences that may compromise user systems, networks, or organizational security. For example, unfiltered or excessive packet capture may overwhelm system memory, resulting in system crashes or degraded performance. Additionally, improper handling of data parsing or storage may lead to unresponsive behavior or data loss, particularly on systems with constrained resources. Therefore, safety considerations must be incorporated throughout the software's lifecycle.

To safeguard against these risks, the software must employ robust error handling and resource management mechanisms. This includes preemptively checking for memory and buffer limits before initiating packet capture, ensuring that infinite loops or malformed packets do not compromise stability. During unexpected network behavior—such as malformed headers or denial-of-service attempts—the software must gracefully discard invalid packets and log the anomaly for review, without affecting the integrity of ongoing captures.

Furthermore, safeguards must prevent misuse of the software for unethical or illegal surveillance. The application must display a clear disclaimer warning users about legal responsibilities and the scope of ethical usage. The user must explicitly acknowledge this disclaimer before initiating any network sniffing activity. The software should not include any features that enable the transmission of captured data to third parties without user consent, nor should it include packet injection or

manipulation functions.

Although no specific safety certification is mandated for this educational tool, developers must adhere to general software safety best practices, such as fail-safe defaults, watchdog timers during critical operations, and avoiding privilege escalation vulnerabilities. The software's architecture should isolate core functionalities to contain failure domains, ensuring that a crash in one module (e.g., filtering) does not affect others (e.g., logging).

5.3 Security Requirements

The security of the packet sniffer tool is paramount, particularly because it interfaces directly with low-level network traffic, which may include sensitive or private information. One of the fundamental requirements is that the tool must not send, forward, or transmit any captured packets or data logs to external servers unless the user explicitly configures and approves such an action. By default, the application must operate in a secure, standalone mode that never attempts unsolicited network communication.

Authentication is also a critical concern. Only users with administrative or root privileges should be able to initiate packet capturing operations. The application must verify user permissions at startup and gracefully inform unauthorized users of their limitations. For extended usage in multi-user environments, optional user authentication with session logging could be implemented to maintain audit trails. Each session log should include the username, time of operation, and filters used.

Another important consideration is data encryption. While raw packets are stored in plain formats like CSV or JSON for accessibility, an optional encryption mode should be provided for users who wish to secure their log files. For example, the user could choose to export logs using AES-256 encryption with a passphrase, making them suitable for transmission in more sensitive environments.

The application must protect user privacy by anonymizing or masking sensitive packet fields upon user request. For instance, IP addresses, MAC addresses, or DNS query data can be obfuscated during capture or export. All privacy-related options must be clearly documented, and their activation should be logged. Furthermore, the software must comply with privacy regulations like the General Data Protection Regulation (GDPR) by ensuring data minimization and providing options to delete or avoid storing user-related traffic data.

5.4 Software Quality Attributes

Delivering high-quality software for a packet sniffer tool requires a commitment to multiple dimensions of software quality. First and foremost is usability. Since this tool will be used by both beginners and advanced users, the interface must be intuitive, well-documented, and responsive. Tooltips, inline help messages, and proper error feedback mechanisms are essential. The GUI must adhere to common design principles like consistency, visibility of system status, and user control.

Reliability is another cornerstone. The software should be able to run continuously for extended

durations (at least 2 hours) without crashing, leaking memory, or causing system instability. Stress testing under heavy network loads must be part of the validation plan. Reliability also includes graceful degradation—if the logging module fails, the GUI and capture engine must continue functioning, and vice versa.

Portability and maintainability are essential for long-term usefulness. The software should be easily portable across Windows and Linux systems by minimizing OS-specific code and using cross-platform libraries. Maintainability should be supported through modular design, code commenting, version control, and automated testing. Dependencies must be explicitly declared and managed.

Other attributes such as testability, extensibility, and reusability are also important. Each component should be independently testable with unit tests achieving >80% coverage. Developers must be able to extend functionality—for example, to add a new protocol parser—without rewriting core modules. Code reuse should be encouraged through the design of general-purpose packet decoding libraries.

5.5 Business Rules

The packet sniffer must operate under clearly defined constraints that guide its behavior and usage based on user roles, legal boundaries, and contextual permissions. A fundamental business rule is that only system administrators or privileged users can initiate packet capture. This prevents misuse by unauthorized users and aligns with OS-level security standards on packet inspection.

The application must enforce scope-limited capture capabilities. For example, capturing packets from loopback interfaces or internal subnets may be allowed without restriction, but capturing external traffic should require an additional confirmation. This ensures users are fully aware of the data being monitored and prevents accidental eavesdropping of confidential information.

Furthermore, any filters or export settings that could expose private user data must be explicitly configured and logged. Users should not be able to activate stealth mode captures or store logs in hidden directories. Transparency in operation is a key rule.

A mandatory acceptance of terms and conditions (EULA) should be required before first use, with periodic reminders or confirmations for ethical usage. The EULA must state that the software is for educational or authorized network diagnostics only, and any other use is the sole responsibility of the user. Violations of these rules should result in software self-termination or limited functionality.

6. Other Requirements

Beyond the core functional and nonfunctional requirements, the packet sniffer must adhere to

several additional constraints that govern its legal, technical, and international usage:

- **Internationalization:** All UI text must be stored as strings to allow future translation into other languages without changing source code.
- **Legal:** Software must comply with local cybersecurity laws and must not include tools for offensive network actions. The tool must include clear disclaimers about responsible usage.
- **Reusability:** Core packet capture and protocol decoding modules should be implemented as reusable libraries that can be integrated into other network tools.
- **Compatibility:** Must work on both IPv4 and IPv6 networks, with capture filters, parsers, and logging mechanisms correctly recognizing modern network packets.
- **Documentation:** Comprehensive user manual, API documentation, and troubleshooting guides must be provided.
- **Testing:** Unit test coverage of at least 80% for core modules, integration testing for end-to-end workflows, and performance testing under various load conditions.

Appendix A: Glossary

Term	Definition
Packet	A unit of data transmitted over a network
Protocol	A set of rules for formatting and transmitting data
TCP/UDP	Transport layer protocols for reliable and unreliable data transmission
IP Address	Identifier for a device on a network (IPv4 or IPv6)
GUI	Graphical User Interface
Scapy	A Python tool for network packet manipulation
Npcap/libpcap	Libraries for packet capture on Windows and Linux respectively
NIC	Network Interface Card

PCAP	Packet Capture file format for storing network traffic

Appendix B: Analysis Models

Use Case Diagram

Describes interactions between user and packet sniffer, including:

- User selects network interface
- User starts/stops packet capture
- User applies filters to captured traffic
- User exports captured data
- System captures and displays packets

Data Flow Diagram

Illustrates how packets flow through capture, decode, filter, and log subsystems:

- Network Interface → Capture Engine
- Capture Engine → Protocol Parser
- Protocol Parser → Filter Engine
- Filter Engine → GUI Display & Data Logger

Sequence Diagram

Details timing of capture-to-display pipeline:

- Initialization sequence for interface detection
- Capture start sequence with thread management
- Packet processing sequence from capture to display
- Export sequence for data saving

Appendix C: To Be Determined List

TBD Item	Description	Target Resolution
TBD-1	Final list of protocols to be supported in protocol analysis (e.g., HTTP/FTP)	Design Phase
TBD-2	Exact GUI layout and widget hierarchy	UI Design Phase
TBD-3	User authentication scheme if deployed in production	Security Review
TBD-4	Optional encrypted logging support implementation	Advanced Features Phase
TBD-5	Performance under high-throughput (10,000+ packets/sec)	Performance Testing Phase

REFERENCES

1. IEEE Std 830-1998, "IEEE Recommended Practice for Software Requirements Specifications"
2. Sommerville, Ian. "Software Engineering". 10th Edition, Pearson Education, 2015
3. Pressman, Roger S. "Software Engineering: A Practitioner's Approach". 8th Edition, McGraw-Hill, 2014
4. Stevens, W. Richard. "TCP/IP Illustrated, Volume 1: The Protocols". 2nd Edition, Addison-Wesley, 2011
5. Kurose, James F. "Computer Networking: A Top-Down Approach". 7th Edition, Pearson, 2016

📋 **END OF DOCUMENT** 📋

Software Engineering Laboratory - Experiment 2

SRS Document - IEEE Format

LTCE | Mumbai University | Academic Year 2025-26