FIRMAI: AI-POWERED IoT FIRMWARE VULNERABILITY ANALYZER

PROJECT REPORT

Submitted by

ABHIRAM G NAIR   PRC22CSOT001

NAVOMI TITUS   PRC22CSOT019

SPANDANA NAIR   PRC22CSOT025

VIGNESH S KUMAR   PRC22CSOT028

To

APJ Abdul Kalam Technological University

in partial fulfillment of the requirements for the award of

B.Tech Degree in

Computer Science and Engineering

(Internet of Things, Cyber Security including Blockchain Technology)

Department of Internet of Things & Cyber Security

Providence College of Engineering, Chengannur

March 2025

CERTIFICATE

Certified that this report entitled **"FIRMAI: AI-Powered IoT Firmware Vulnerability Analyzer"** is the report of project completed by the following students during **2024–2025** in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science and Engineering (Internet of Things, Cyber Security including Blockchain Technology).

ABHIRAM G NAIR	PRC22CSOT001
NAVOMI TITUS	PRC22CSOT019
SPANDANA NAIR	PRC22CSOT025
VIGNESH S KUMAR	PRC22CSOT028

Ms. Praseetha S Nair   (Project Supervisor)

Assistant Professor

Department of Internet of Things & Cyber Security (CSOT)

Ms. Salitha M.K   (Project Co-Supervisor)

Assistant Professor

Department of Internet of Things & Cyber Security (CSOT)

Ms. Salitha M.K   (Project Co-Ordinator)

Assistant Professor & HoD - CSOT

Department of Internet of Things & Cyber Security (CSOT)

Providence College of Engineering, Chengannur

DECLARATION

We, hereby declare that, this project report entitled **'FIRMAI: AI-Powered IoT Firmware Vulnerability Analyzer'** is the bonafide work of ours carried out under the supervision of **Ms. Praseetha S Nair**, Assistant Professor, Department of Internet of Things & Cyber Security. We declare that, to the best of our knowledge, the work reported herein does not form part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion to any other candidate. The content of this report is not being presented by any other student to this or any other University for the award of a degree.

ABHIRAM G NAIR	PRC22CSOT001
NAVOMI TITUS	PRC22CSOT019
SPANDANA NAIR	PRC22CSOT025
VIGNESH S KUMAR	PRC22CSOT028

Ms. Praseetha S Nair (Project Supervisor)

Assistant Professor

Department of Internet of Things & Cyber Security (CSOT)

Ms. Salitha M.K (HoD - CSOT)

Assistant Professor & HoD - CSOT

Department of Internet of Things & Cyber Security (CSOT)

Providence College of Engineering, Chengannur

Date: 31/03/2025

ACKNOWLEDGEMENTS

ABSTRACT

This project proposes FIRMAI, an AI-powered IoT firmware vulnerability analyzer...

Contents

List of Figures

List of Tables

LIST OF ABBREVIATIONS

AI   Artificial Intelligence

IoT   Internet of Things

FIRMAI Firmware AI

Chapter 1

INTRODUCTION

1.1 Background

The Internet of Things has fundamentally transformed technology ecosystems by connecting billions of devices worldwide. Industry projections indicate IoT devices will exceed 30 billion by 2025, generating trillions in economic value. However, this exponential growth accompanies corresponding increases in security vulnerabilities and cyber threats. IoT devices present unique security challenges distinguishing them from traditional computing systems, demanding specialized security solutions.

IoT devices operate under severe resource constraints with minimal processing power, memory, and energy resources. Many devices have only kilobytes of RAM and low-power microcontrollers, making comprehensive security solutions like antivirus software impractical. This necessitates lightweight security mechanisms balancing protection with performance. Unlike traditional x86-dominant computing, IoT employs diverse processor architectures including ARM, MIPS, PowerPC, and specialized microcontrollers, each with different instruction sets and hardware features. This heterogeneity complicates security analysis as tools developed for one architecture may not transfer to others.

Firmware vulnerabilities represent critical attack surfaces. Firmware controls hardware components and implements core functionality at low levels. Vulnerabilities including buffer overflows, command injection flaws, hardcoded credentials, and insecure cryptographic implementations can persist for years without patches. Network-level threats exploit protocols lacking security features. The Mirai botnet attack of 2016 demonstrated devastating potential, leveraging over 1.2 million infected devices for massive DDoS attacks. Limited visibility prevents comprehensive assessment as administrators lack device discovery mechanisms. IoT-targeted malware has evolved in sophistication with families like Mirai, Gafgyt, and Hajime exploiting vulnerabilities and weak credentials.

Recent advances in artificial intelligence and machine learning offer new security possibilities. Deep learning techniques, particularly CNNs, demonstrate remarkable effectiveness in pattern recognition and classification. Automated firmware analysis frameworks enable vulnerability discovery through emulation in virtualized environments. Our project addresses the

critical need for comprehensive integrated security solutions protecting IoT ecosystems across multiple technology stack layers through the FIRMAI platform.

1.2   Existing System

1.2.1   Firmware Analysis Tools

Traditional firmware analysis relies on static analysis tools examining binaries without execution. Tools like Binwalk extract filesystem contents and identify embedded files but cannot reveal runtime behaviors or network interactions. Many vulnerabilities only manifest during execution when specific code paths trigger or components interact unexpectedly. Dynamic analysis frameworks attempted automated firmware emulation but achieved low success rates around 16% for diverse firmware collections due to rigid assumptions about hardware configurations. Existing frameworks primarily target specific architectures like MIPS, struggling with ARM, x86, and PowerPC devices. Network configuration challenges prevent establishing functional connectivity in emulated environments. Manual intervention requirements undermine automation objectives when failures occur, requiring security analysts to debug low-level boot processes and hardware emulation.

1.2.2   Network Scanning Solutions

Generic network scanners like Nmap [7] provide powerful reconnaissance but lack IoT-specific capabilities. IoT devices employ non-standard configurations, minimal services, and restrictive firewalls that evade conventional scanning. Commercial IoT discovery platforms offer enhanced identification but require substantial investment and impose vendor lock-in with limited customization. Basic open-source scanners provide lightweight alternatives but typically lack sophisticated vendor identification, parallel processing for performance, and integration capabilities with other security tools. Single-threaded implementations restrict applicability in large-scale environments requiring efficient scanning of thousands of devices.

1.2.3   Malware Detection Approaches

Traditional signature-based antivirus identifies malware by comparing against known threat databases. This approach fundamentally cannot detect novel variants, polymorphic threats, or zero-day exploits. Resource constraints prevent deploying full-featured antivirus on many IoT devices. Network-based IDS solutions like Snort analyze traffic for suspicious patterns but require continuous updates and generate significant false positives. They lack sophistication for identifying subtle behavioral anomalies. Early machine learning attempts using KNN, SVM, and Decision Trees showed promise but suffered from feature engineering requirements, limited

pattern recognition in high-dimensional data, and poor scalability. Traditional models struggle with complex hierarchical patterns that deep learning excels at capturing automatically.

1.2.4 Fragmentation Problems

The fundamental limitation lies in fragmented approaches addressing individual security aspects in isolation. Organizations deploy separate tools for firmware analysis, network visibility, and malware detection, creating integration gaps with incompatible data formats, resource duplication consuming redundant computational resources, incomplete coverage missing threats spanning multiple stack layers, and delayed response from manual correlation. This fragmentation motivated developing FIRMAI as an integrated platform eliminating these limitations through unified architecture combining all three security domains.

1.3 Problem Statement

To design and develop an integrated AI-powered security framework for comprehensive IoT firmware vulnerability analysis, network device discovery, and real-time malware detection utilizing advanced machine learning techniques.

The rapid proliferation of IoT devices across residential, commercial, and industrial environments has created an expansive attack surface that traditional security solutions struggle to protect effectively. Current firmware analysis frameworks achieve only 16% emulation success rates, meaning the vast majority of firmware images cannot be analyzed dynamically. This fundamental limitation forces reliance on less effective static analysis or expensive manual reverse engineering. Organizations lack adequate network visibility into heterogeneous IoT ecosystems where devices use diverse protocols and minimal service footprints.

Traditional signature-based malware detection fails against evolving threats employing polymorphism and novel attack vectors. Even machine learning approaches suffer from manual feature engineering requirements and inability to generalize beyond training data. Current security tools operate in isolation creating integration gaps preventing holistic assessments. Security professionals lack unified platforms simultaneously analyzing firmware vulnerabilities, mapping network topologies, and detecting malicious activities.

The resource-constrained nature of IoT devices necessitates lightweight security mechanisms operating efficiently without degrading performance. Traditional heavyweight solutions prove impractical with limited processing power, memory, and energy resources. Deployment complexity requires extensive manual configuration and specialized expertise. These challenges are compounded by diverse processor architectures, difficulty obtaining physical device access, rapidly evolving malware, lack of standardized security practices, and extended operational lifecycles where devices remain deployed long after vendor support ends. Our integrated framework addresses these interconnected challenges through comprehensive automation and

unified architecture.

1.4 Objectives

1.4.1 Primary Objectives

Develop Integrated IoT Security Platform: Design and implement a unified framework seamlessly integrating firmware analysis, network scanning, and malware detection capabilities. The platform eliminates fragmentation characterizing existing solutions by providing single interface for comprehensive security assessment. Data flows efficiently between modules enabling correlation of findings across security domains to identify complex multi-vector threats that individual tools would miss. The integration provides end-to-end visibility from firmware-level vulnerabilities to network traffic anomalies.

Implement High-Efficiency Firmware Emulation: Develop automated firmware emulation engine capable of extracting and emulating diverse IoT firmware images across multiple processor architectures. The engine implements advanced arbitration techniques systematically addressing common emulation failures through intelligent workarounds rather than attempting perfect hardware replication. Target emulation success rates exceed 75% across diverse firmware collections from major IoT device vendors. The system enables dynamic vulnerability analysis without requiring physical hardware access, dramatically reducing analysis costs and time.

Create Comprehensive Network Discovery: Build automated network scanning capabilities identifying active IoT devices and extracting device metadata with minimal performance overhead. The scanner implements ARP-based scanning to bypass firewall restrictions, TCP port scanning to identify running services, MAC address vendor lookup for device identification, and multi-threaded processing for high-performance scanning of large subnets. The system generates comprehensive device inventories providing security teams complete visibility into IoT deployments.

Deploy AI-Powered Malware Detection: Implement deep learning-based malware classification using CNNs capable of identifying and categorizing IoT malware with accuracy exceeding 95%. We develop preprocessing pipelines extracting relevant features from network traffic, design and train CNN architectures with dropout regularization and batch normalization, implement inference pipelines for real-time classification, and enable continuous model improvement through retraining on updated datasets. The system detects both known malware families and novel variants through learned behavioral patterns.

1.4.2   Secondary Objectives

Ensure System Scalability: Design architecture supporting efficient analysis of large-scale IoT environments with hundreds or thousands of devices. Implementation includes parallel processing for concurrent firmware emulation, multi-threading for high-performance network scanning, GPU acceleration for deep learning inference, containerization with Docker for resource isolation and scalability, and distributed deployment options supporting horizontal scaling. The system maintains acceptable performance as deployment size grows.

Provide Integration Capabilities: Develop RESTful APIs enabling integration with existing SIEM platforms, security orchestration tools, and ticketing systems. Implementation includes standardized data formats for interoperability, webhook notifications for real-time alerts, and comprehensive logging for audit trails. Organizations incorporate our system into existing security workflows seamlessly without major infrastructure changes.

Build Robust Testing Framework: Conduct rigorous testing using real-world IoT firmware images from major vendors, live network environments with diverse device types, and labeled malware datasets for model validation. Implementation includes unit testing for individual components, integration testing for module interactions, performance benchmarking against defined metrics, and security testing ensuring the system itself doesn't introduce vulnerabilities.

Develop Comprehensive Documentation: Create extensive documentation covering system architecture and design decisions, installation and deployment procedures, configuration options and parameters, API reference with code examples, troubleshooting guides for common issues, and security best practices for operation. Documentation enables both end users to deploy effectively and researchers to understand and extend our work.

1.5   Scope

1.5.1   In-Scope Capabilities

Firmware Analysis and Emulation: Our system supports automated extraction, analysis, and emulation of firmware images from major IoT device categories including wireless routers, IP cameras, network-attached storage devices, and smart home hubs. We support firmware in common formats including binary images, compressed archives, and manufacturer-specific formats. Supported processor architectures include MIPS (both endianness), ARM (multiple versions), x86 (32-bit and 64-bit), and PowerPC. The firmware emulation component leverages QEMU virtualization with customized kernel images optimized for IoT firmware execution.

Network Device Discovery: Automated scanning capabilities identify active devices within specified IP address ranges supporting both IPv4 and basic IPv6. The system employs ARP requests at Layer 2 bypassing firewall configurations, TCP SYN scans for port accessibility, service banner grabbing for version identification, and ICMP pings as supplementary discovery.

We extract comprehensive metadata including IP addresses, MAC addresses, open TCP ports, service banners, and manufacturer information from MAC address databases.

Malware Detection: Deep learning models trained on labeled datasets (such as IoT-23 [8]) enable identification and classification of malicious network traffic. We support detection of major IoT malware families including Mirai variants, Gafgyt/BASHLITE, Hajime, VPNFilter, and emerging families as training data becomes available. The system performs real-time traffic analysis through packet capture, feature extraction, preprocessing and normalization, CNN-based classification with confidence scoring, and alerting when malicious activity is detected.

System Integration: A unified management interface coordinates firmware analysis, network scanning, and malware detection workflows. The system provides web-based dashboard for visualization, RESTful APIs for programmatic access, command-line interfaces for scripting automation, and webhook notifications for external system integration. Automated reporting generates comprehensive security assessments combining findings from all system components in multiple formats including HTML, PDF, and JSON.

1.5.2 Out-of-Scope Limitations

Physical Hardware Analysis: The system does not perform physical security assessments including hardware-level debugging with JTAG, invasive techniques requiring device disassembly, side-channel attacks, or chip-level security evaluation. These capabilities require specialized equipment and expertise beyond our software-focused approach.

Comprehensive Protocol Analysis: While analyzing network traffic at packet level, deep inspection of application-layer protocols beyond common standards is limited. We support HTTP/HTTPS, MQTT, CoAP, FTP, Telnet, and SSH, but proprietary or undocumented protocols may not be fully analyzed.

Active Exploitation: The system identifies vulnerabilities through dynamic analysis but does not include capabilities for actively exploiting discovered weaknesses beyond proof-of-concept demonstrations. Full exploit development and weaponization are explicitly excluded.

Automated Remediation: The system identifies vulnerabilities and security issues but does not automatically remediate them. We do not implement automatic firmware patching, device configuration changes, or malware removal to avoid unintended consequences.

1.5.3 Target Users and Deployments

The FIRMAI system targets security researchers investigating IoT vulnerabilities, network administrators managing organizational IoT deployments, product security teams assessing device security, academic institutions teaching cybersecurity concepts, and managed security service providers offering assessment services. Deployment scenarios include corporate networks, research laboratories, educational environments, and development testbeds. The system operates on standalone workstations, dedicated servers, or cloud infrastructure depending on scale and

performance requirements.

Chapter 2

LITERATURE REVIEW

2.1 IoT Firmware Analysis and Emulation

Research in automated firmware analysis evolved significantly over the past decade as security implications of firmware vulnerabilities became apparent. Early approaches relied primarily on static analysis examining firmware binaries without execution. While static analysis provides valuable insights into code structure, it fundamentally cannot capture runtime behaviors, inter-process communications, or network interactions revealing security flaws. This limitation motivated development of dynamic analysis frameworks based on firmware emulation.

Chen et al. [3] introduced Firmadyne in 2016 as a pioneering framework for automated large-scale firmware emulation. Firmadyne automatically extracts filesystem contents, identifies embedded kernels, configures QEMU-based virtual machines, sets up network interfaces, and launches emulated instances. The framework demonstrated feasibility of automated firmware analysis at scale, revealing widespread vulnerabilities including backdoor accounts, web interface vulnerabilities, insecure network services, and inadequate input validation. However, Firmadyne achieved only 16.28% emulation success across diverse collections due to rigid hardware and network configuration assumptions.

Subsequent research at KAIST's SysSec Lab [6] developed advanced arbitration techniques improving emulation success. Rather than attempting perfect hardware replication, arbitrated emulation prioritizes creating functional environments sufficient for security analysis. Researchers identified five critical arbitration categories: boot process arbitration handling initialization failures, network configuration arbitration establishing connectivity, NVRAM arbitration emulating persistent storage, kernel compatibility arbitration managing version mismatches, and environment arbitration addressing miscellaneous setup issues. Implementation of systematic arbitration achieved 79.36% emulation success across 1,124 firmware images—nearly five-fold improvement over previous approaches. The arbitrated approach also enabled discovery of 12 new zero-day vulnerabilities affecting 23 devices.

2.2 Network Scanning and Device Discovery

Network scanning encompasses techniques for discovering active hosts, identifying services, and fingerprinting systems. For IoT security, comprehensive network visibility is essential as administrators cannot secure devices they don't know exist. Traditional scanning tools like Nmap [7] provide powerful capabilities but lack IoT-specific features.

The Address Resolution Protocol (ARP) operates at Layer 2 of the OSI model providing mechanism to map IP addresses to hardware MAC addresses. ARP-based scanning offers significant advantages for IoT device discovery. When hosts need to communicate with IP addresses on local networks, they broadcast ARP requests. Network scanners exploit this by sending ARP requests to all subnet addresses and recording responses. ARP scanning bypasses firewall rules typically operating at Layer 3 and above. Devices configured to ignore ICMP pings still respond to ARP requests.

Research demonstrated Python-based ARP scanning implementations using Scapy library for packet crafting. Scapy provides low-level network access enabling construction of custom ARP packets, broadcast sending, response capture and parsing, and IP/MAC address extraction. Implementation considerations include multi-threading for concurrent address scanning, appropriate timeout values balancing speed and reliability, retry mechanisms handling packet loss, and rate limiting avoiding network infrastructure overload.

TCP port scanning reveals services devices are running. Several techniques exist with different tradeoffs. TCP SYN scanning sends SYN packets observing responses without completing handshakes—faster and stealthier than full connections. Service version detection connects to open ports analyzing banners for software identification. Research demonstrated operating system fingerprinting through port pattern analysis as different systems have characteristic open port configurations. For IoT devices specifically, non-standard ports and minimal service footprints present challenges requiring broader port range testing.

2.3 Machine Learning for Malware Detection

Malware detection evolved through several technology generations. First-generation signature-based detection matched known malware signatures but completely failed for novel threats. Second-generation heuristic analysis examined code behavior patterns detecting suspicious activities without specific signatures, though generating high false positives. Third-generation traditional machine learning applying KNN, SVM, Decision Trees, and Random Forests improved over heuristics but required extensive manual feature engineering and struggled with high-dimensional data. Fourth-generation deep learning automatically learns hierarchical feature representations from raw data, eliminating manual engineering while identifying subtle patterns across millions of parameters.

Research into deep learning for malware detection explored various architectures. Convolutional Neural Networks originally developed for image classification excel at identifying spatial patterns. Researchers adapted CNNs to malware detection by representing malware as images with binary code visualized as pixels or network traffic converted to image format. CNNs automatically learn convolutional filters detecting characteristic patterns at multiple scales. Khan et al. [5] demonstrated deep boosted CNN for IoT malware detection converting network packets into grayscale images, achieving 98.6% detection accuracy through ensemble approaches.

Recurrent Neural Networks process sequential data maintaining hidden states capturing temporal dependencies. For malware detection, RNNs analyze sequences of system calls, API calls, or network packets. Long Short-Term Memory variants address vanishing gradient problems enabling learning of long-range dependencies. Al Abbas et al. [1] applied RNNs with NLP techniques to IoT malware detection treating binary code as text sequences, achieving 99.3% accuracy using word embeddings for assembly instructions and RNNs capturing execution flow patterns.

Hybrid architectures combin multiple network types leverage complementary strengths. Jeyalakshmi & Nallaperumal [4] introduced HCAGAN-DBN combining GANs and DBNs. Their GAN generated synthetic training samples to address class imbalance while the DBN did hierarchical feature learning, achieving 99.83% accuracy. Cross-architecture malware detection addresses IoT's diverse processor architectures. Chaganti et al. [2] developed architecture-agnostic feature extraction capturing semantic behaviors regardless of instruction set, achieving 100% malware identification accuracy and 98% family classification across architectures.

2.4   Integrated Security Frameworks

While significant research addresses individual IoT security components, fewer works examine integrated approaches combining multiple domains. Security Information and Event Management platforms aggregate security data from multiple sources enabling correlation and analysis. Key SIEM principles include data normalization into common schemas, real-time stream processing for continuous analysis, correlation rules identifying patterns across events, and automated response executing predefined playbooks.

Security Orchestration, Automation and Response platforms extend SIEM with enhanced automation. SOAR characteristics include workflow automation defining tasks as directed graphs, integration capabilities providing standardized APIs for diverse tools, playbook libraries for common security scenarios, and human-in-the-loop support for critical decisions requiring approval. Modern distributed systems increasingly adopt containerization and microservices architectures. Container technologies like Docker provide application isolation, consistent environments, efficient resource utilization, and rapid deployment capabilities. Microservices architecture breaking systems into loosely-coupled services provides modularity,

independent scalability, resilience, and technology diversity.

Our literature review reveals substantial progress in individual domains while highlighting critical need for integrated solutions. Key findings informed our system design: arbitrated emulation prioritizing functional environments over perfect hardware replication achieves practical success rates; ARP-based scanning provides superior IoT device discovery bypassing firewall restrictions; deep learning with CNNs achieves state-of-the-art accuracy generalizing to novel malware variants; and integrated platforms combining multiple security capabilities provide superior threat visibility compared to fragmented toolsets. By synthesizing insights from these research areas, we developed FIRMAI as comprehensive integrated IoT security platform addressing critical gaps in existing solutions.

Chapter 3

SYSTEM ANALYSIS

3.1 Functional Requirements

3.1.1 Firmware Image Processing (FR-1)

The system shall extract, analyze, and prepare firmware images for emulation. Input includes firmware binary files in common formats (.bin,.img,.zip,.tar.gz). Processing extracts filesystem contents using binwalk and complementary tools, identifies embedded Linux kernel version and architecture, locates configuration files and executables, analyzes filesystem structure determining device type and manufacturer, and generates metadata describing firmware characteristics. Output includes extracted filesystem directory preserving original hierarchy, metadata JSON file containing kernel version and architecture details, and analysis report summarizing firmware contents. This high-priority requirement provides foundational capability required for firmware emulation. Validation confirms successful extraction from firmware images representing major device categories and processor architectures.

3.1.2 Automated Firmware Emulation (FR-2)

The system shall emulate extracted firmware in QEMU-based virtual environments using advanced arbitration techniques. Input includes extracted firmware filesystem, identified kernel image, device-specific configuration parameters, and emulation preferences. Processing configures QEMU virtual machines with appropriate architecture and resources, selects matching kernel images, applies boot process arbitration for non-standard initialization, configures network interfaces using arbitration techniques, implements NVRAM emulation for persistent storage, monitors emulation health, and applies systematic interventions when failures occur. Output includes running emulated firmware instance accessible via network, console logs documenting boot process, NVRAM contents showing configuration, and status reports with diagnostic information. This high-priority core capability enables dynamic firmware analysis. Validation achieves emulation success exceeding 75% across diverse collections with network accessibility verification.

3.1.3 Vulnerability Analysis Testing (FR-3)

The system shall perform automated vulnerability testing against emulated firmware instances. Input includes running emulated instances, applicable CVE lists, fuzzing templates, and exploitation modules. Processing enumerates exposed services, queries vulnerability databases for known CVEs affecting identified versions, executes exploit modules attempting vulnerability triggers, performs fuzzing of web interfaces and services, monitors for crashes and authentication bypasses, and documents successful exploits with proof-of-concept. Output includes comprehensive vulnerability assessment reports with CVE identifiers and severity ratings, successful exploit demonstrations with evidence, remediation recommendations, and crash dumps when exploitation causes failures. High priority demonstrates practical security impact of identified vulnerabilities.

3.1.4 Network Device Discovery (FR-4)

The system shall identify active devices within specified network ranges using ARP and TCP scanning. Input includes IP address ranges in CIDR notation, optional port range specifications, and scan intensity parameters. Processing validates CIDR notation determining target ranges, generates IP address lists, sends ARP requests across subnets, collects responses containing IP/MAC pairs, performs TCP SYN scans on commonly used ports, attempts full connections for banner grabbing, correlates ARP and TCP results by IP address, and handles network errors gracefully. Output includes device inventory listing IP addresses, MAC addresses, response timestamps, open TCP ports per device, service banners identifying software versions, and reachability metrics. High priority provides essential network visibility. Validation confirms accurate discovery of all test network devices with Class C subnet completion within 5 minutes.

3.1.5 Device Identification (FR-5)

The system shall identify device vendors and types based on MAC addresses and service fingerprints. Input includes MAC addresses from scans, open port lists, service banners, and optional additional context. Processing extracts OUI from MAC addresses, queries local OUI databases determining manufacturers, analyzes open port patterns against device signatures, parses service banners for vendor strings and versions, applies heuristic rules combining multiple indicators, and assigns confidence scores to classifications. Output includes enhanced inventory with manufacturer names, device types with confidence scores, firmware versions from banners, and model numbers when identifiable. Medium priority enhances inventory quality beyond basic scanning.

3.1.6   Traffic Capture and Preprocessing (FR-6)

The system shall capture network traffic and preprocess into formats suitable for deep learning models. Input includes network interface specifications, capture duration parameters, and BPF filters for selective capture. Processing initializes packet capture using libpcap, captures matching packets, extracts relevant features including packet sizes and protocol distributions, aggregates packet-level features into flow-level statistics, normalizes feature values, encodes categorical variables numerically, and handles missing data. Output includes preprocessed feature matrices in NumPy format with rows as samples and columns as features, metadata describing feature definitions and scaling parameters, and optionally raw PCAP files for reference. High priority essential for malware detection capability.

3.1.7   Malware Classification (FR-7)

The system shall classify network traffic as malicious or benign using trained CNN models. Input includes preprocessed feature matrices from traffic capture. Processing loads pre-trained model weights, validates input dimensions match expectations, performs forward propagation through convolutional and dense layers, applies batch normalization, generates output activations, applies softmax producing probability distributions, selects highest probability classes as predictions, and applies confidence thresholds rejecting low-confidence predictions. Output includes classification results indicating benign or malicious, specific malware families when malicious, confidence scores, and optionally detailed feature importance analysis. High-priority core malware detection capability. Validation achieves classification accuracy exceeding 95% on test sets with both precision and recall above 90% per malware family.

3.1.8   Model Training (FR-8)

The system shall support training and fine-tuning of deep learning models on custom datasets. Input includes labeled datasets with ground truth classifications, training hyperparameters, and optional pre-trained weights for transfer learning. Processing loads and validates datasets ensuring sufficient samples per class, splits into training/validation/test sets ensuring balance, applies data augmentation if beneficial, initializes or loads model weights, implements training loops with batch gradient descent calculating losses and gradients, applies optimization algorithms updating weights, implements dropout regularization during training, normalizes activations with batch normalization, monitors metrics after each epoch, implements early stopping preventing overfitting, and selects best weights based on validation performance. Output includes trained model weights, training history plots, comprehensive performance metrics, and feature importance analysis. Medium priority enables customization beyond pre-trained models.

3.1.9 Integrated Dashboard (FR-9)

The system shall provide unified web-based interface displaying results from all modules. Input includes user authentication credentials, query parameters for filtering, and visualization preferences. Processing authenticates users verifying authorization, queries databases for firmware/network/malware results, aggregates data correlating by device identifiers, generates visualizations including topology graphs and vulnerability heat maps, formats reports in HTML/PDF/JSON formats, and applies access controls ensuring users see only authorized data. Output includes interactive web dashboard with real-time updates, drill-down capabilities, filtering and search functionality, and downloadable reports. Medium priority enhances usability though system can operate via command-line interfaces.

3.1.10 Alert Generation (FR-10)

The system shall generate alerts when critical vulnerabilities or malware infections are detected. Input includes configured alert thresholds and severity levels, notification preferences, and suppression rules preventing flooding. Processing monitors analysis results real-time for high-severity findings including critical vulnerabilities with CVSS $\geq$ 7.0, malware detections with confidence $\geq$ 90%, unusual network patterns, and firmware with known backdoors, evaluates findings against configured rules, applies suppression avoiding duplicates, formats alert messages with relevant details including affected device identification and remediation recommendations, and delivers via configured channels. Output includes timely notifications via email with formatted messages, HTTP POST requests to webhooks with JSON payloads, dashboard popup notifications, and optionally SMS for critical alerts. Medium priority improves response time.

3.2 Non-Functional Requirements

3.2.1 Performance and Scalability (NFR-1)

The system shall support parallel processing maintaining acceptable performance as workload increases. For firmware emulation, process 50+ images per hour using 8-core processors, support concurrent emulation of 10+ instances, and scale linearly adding processing nodes. For network scanning, complete Class C subnet scans within 5 minutes, support Class B subnets within 2 hours, and handle concurrent scan requests without interference. For malware classification, process 100+ samples per minute on CPU, process 1000+ samples per minute with GPU acceleration, and maintain inference latency below 100ms per sample on GPU. Security assessments of large IoT deployments require processing many firmware images and scanning extensive networks, making performance and scalability essential.

3.2.2 Reliability and Fault Tolerance (NFR-2)

The system shall handle errors gracefully maintaining availability and data integrity when components experience issues. Implementation includes comprehensive exception handling for all external interactions, detailed error logging with stack traces and system state, automatic retry with exponential backoff for transient failures, isolation of firmware emulation failures preventing system-wide impacts through containerization, health checks for all services with automatic restart of failed components, and database consistency through transactions and proper rollback. Measurement targets system uptime > 99% during continuous operation excluding planned maintenance, error recovery success rate > 95% for transient failures, no data loss or corruption when components fail, and mean time to recovery < 5 minutes for component failures.

3.2.3 Usability and Accessibility (NFR-3)

The system shall provide intuitive interfaces accessible to users with varying technical expertise from security experts to network administrators. Specifications include web dashboard requiring no specialized training for basic operations, clear informative error messages with suggested remediation, command-line interfaces supporting comprehensive help documentation, interactive tutorials guiding new users through common workflows, consistent terminology and interface conventions, and responsive design supporting desktop and tablet form factors. Measurement targets new users completing basic scanning workflows within 15 minutes of first use, user satisfaction ratings > 4.0/5.0 in surveys, error messages enabling users to resolve common issues without support, and dashboard navigation intuitive to 90%+ of users on first attempt.

3.2.4 Security and Privacy (NFR-4)

The system shall protect sensitive security data preventing unauthorized access ensuring the security tool itself doesn't introduce vulnerabilities. Implementation includes role-based access control with Admin/Analyst/Viewer roles, AES-256 encryption for sensitive data at rest including passwords and vulnerability details, TLS 1.3 for all network communications, secure session management with secure httpOnly cookies and JWT tokens with appropriate expiration, audit logging of all administrative actions, OWASP Top 10 guideline adherence, input validation and sanitization preventing injection attacks, and principle of least privilege throughout design. A compromised security tool provides attackers valuable reconnaissance data. Validation includes automated vulnerability scanning, manual penetration testing, and code review focusing on security with no high or critical vulnerabilities.

3.2.5 Maintainability and Extensibility (NFR-5)

The system architecture shall facilitate maintenance, updates, and feature additions enabling long-term evolution without major rewrites. Specifications include modular architecture with well-defined interfaces via RESTful APIs and message queues, comprehensive code documentation following industry standards with docstrings and inline comments for complex logic, unit test coverage > 80% for critical modules, integration tests covering major end-to-end workflows, configuration externalized to files separate from code in JSON/YAML/environment variables, version control with git using feature branches and pull requests, and continuous integration pipelines running tests on every commit. Measurement targets code maintainability index > 70, average time for minor feature additions < 1 week, and regression bug rate < 2% per release.

3.2.6 Portability and Deployment (NFR-6)

The system shall support deployment across diverse computing environments from developer workstations to cloud platforms. Specifications include Docker containerization for consistent deployment regardless of host environment, support for major Linux distributions including Ubuntu 20.04+, Debian 11/12, CentOS 8+, and Rocky Linux 8+, cloud-compatible architecture supporting AWS, Azure, and Google Cloud Platform, automated deployment scripts and comprehensive installation documentation, clearly documented resource requirements with minimum and recommended specifications, and support for x86_64 and ARM64 architectures where practical. Measurement targets successful deployment on all specified distributions and cloud platforms, installation time < 30 minutes following documentation, and zero manual configuration for basic deployment with sensible defaults.

3.2.7 Documentation and Training (NFR-7)

The system shall include comprehensive documentation supporting installation, configuration, operation, and troubleshooting. User documentation includes installation guides with step-by-step instructions for different environments, user manuals covering all features with screenshots and examples, configuration references documenting all parameters, troubleshooting guides for common issues with solutions, and FAQs addressing frequent questions. Developer documentation includes architecture documentation with diagrams explaining system design, API references with endpoint descriptions and examples, extension guides showing how to add new modules, code documentation via docstrings for all public interfaces, and contribution guidelines for external developers. Training materials include tutorial videos demonstrating key workflows, sample datasets for experimentation, case studies showing real-world applications, and presentation slides for classroom training. Measurement targets documentation completeness covering all features, documentation accuracy with tested procedures working as described, and user ability to complete tasks using documentation only with >90% success rate.

3.3 Feasibility Analysis

3.3.1 Technical Feasibility

Published research demonstrates technical feasibility of automated firmware emulation with arbitrated approaches achieving 79.36% success across diverse collections. The framework's implementation details are openly published enabling replication. Our team possesses necessary technical skills including Linux system internals knowledge, virtualization experience with QEMU/KVM, Python proficiency for automation, and networking understanding. Required technologies are mature and stable: QEMU provides robust full-system emulation, binwalk reliably extracts filesystems, standard Linux utilities handle filesystem manipulation, and Docker enables containerized deployment. Assessment: Technically Feasible—proven technology with clear implementation path.

Network scanning using ARP and TCP techniques is well-established with numerous successful implementations. Python libraries provide necessary functionality through Scapy for packet manipulation, socket for TCP connections, concurrent.futures for multi-threading, and established MAC vendor databases. Our team has networking experience including OSI model understanding, network programming experience, and security tools familiarity. The technologies are mature with ARP standard since 1982 and TCP since 1981. Assessment: Technically Feasible—straightforward implementation using proven techniques.

Deep learning frameworks provide robust foundations for CNN-based malware classifiers. TensorFlow/Keras offer high-level APIs simplifying model development, GPU acceleration through CUDA/cuDNN, and extensive documentation with community support. Published research demonstrates accuracy exceeding 95% on labeled datasets validating approaches. Our team has machine learning knowledge including neural networks understanding, Python experience with NumPy/Pandas, and model training familiarity. Required datasets are available through IoT-23 from Stratosphere Laboratory and other public malware datasets. Assessment: Technically Feasible—proven deep learning approaches with accessible frameworks.

Modern software engineering practices enable integration of independent modules. RESTful APIs provide standardized inter-module communication, message queues enable asynchronous processing, shared databases facilitate data exchange, and Docker containerization simplifies deployment. Our team has software engineering experience including API design, database schema design, web application development, and system integration experience. Integration patterns are well-established with microservices as industry standard. Assessment: Technically Feasible—standard integration patterns with proven technologies. Overall Technical Feasibility Conclusion: All major components have proven implementations, integration patterns are well-understood, required technologies are mature and stable, our team has necessary skills, and no fundamental technical barriers exist. Verdict: TECHNICALLY FEASIBLE.

3.3.2 Operational Feasibility

Target users including security researchers, network administrators, and security analysts possess sufficient technical background to operate command-line and web interfaces effectively. They are already familiar with security tools and concepts. Web-based interfaces reduce learning curves compared to command-line-only tools while comprehensive documentation and tutorials support onboarding. Our value proposition addresses genuine user needs including lack of comprehensive IoT security tools, frustration with fragmented existing solutions, and demand for automated security assessment capabilities. Early feedback from potential users indicates strong interest.

Docker containerization dramatically simplifies installation, updates, and troubleshooting compared to traditional deployment. Standard Linux system administration skills suffice for deployment and maintenance. Infrastructure requirements including servers and databases are familiar to IT departments. Cloud deployment options eliminate on-premises infrastructure requirements for organizations preferring cloud solutions. Most organizations already have Linux servers or cloud accounts, PostgreSQL/MongoDB familiarity as common databases, and Docker experience increasingly standard.

RESTful APIs enable integration with existing SIEM platforms, ticketing systems like Jira and ServiceNow, and security orchestration tools. Standard data formats including JSON and CSV facilitate data exchange. Webhook notifications push alerts to external systems. Organizations can incorporate our system into existing security operations workflows without major disruption. Common integration scenarios include exporting findings to SIEM for correlation, creating tickets automatically when high-severity findings are detected, and triggering automated response playbooks via SOAR platforms.

Comprehensive documentation reduces training requirements. Web-based interfaces are intuitive for users with basic technical literacy. Command-line interfaces follow common conventions familiar to Linux users. Tutorial videos and examples provide self-service training resources while community forums enable peer support. Expected training timeline includes 1–2 hours for basic operation, 1–2 days for advanced features and customization, and ongoing learning through experimentation. Overall Operational Feasibility Conclusion: System design aligns with user capabilities and expectations, deployment and maintenance require only standard widely available skills, integration capabilities support incorporation into existing workflows, and training requirements are reasonable given target user background. Verdict: OPERATIONALLY FEASIBLE.

3.3.3 Economic Feasibility

For academic context, software development cost is ₹0 as student labor is not salaried. Development hardware costs ₹120,000-₹180,000 for workstation with suitable specifications. Cloud infrastructure for development/testing is ₹0-₹10,000/month optional for testing scenarios using

free tiers initially. Personnel costs are ₹0 in academic context. Total development cost in academic context is ₹120,000-₹180,000 for hardware only.

Deployment costs vary by scenario. On-premises deployment requires hardware costing ₹150,000-₹400,000 for servers, ₹10,000-₹50,000 for network equipment, and ₹20,000-₹50,000 for UPS and peripherals, totaling ₹180,000-₹500,000 one-time with ₹20,000-₹50,000 annual maintenance. Cloud deployment costs ₹20,000-₹200,000/month depending on scale with ₹0 software costs as open-source software runs on cloud infrastructure. Hybrid deployment combines on-premises for sensitive data and cloud for scalability costing ₹100,000-₹300,000 initial plus ₹30,000-₹80,000/month ongoing.

Return on investment through cost avoidance includes preventing security breaches averaging ₹50 lakhs-₹1 crore for SMBs with even preventing one small breach justifying system cost. Saved analysis time worth ₹50,000-₹100,000/month in analyst salaries translates to ₹200,000-₹400,000/year in analyst time savings through manual firmware analysis automation saving 40–80 hours/month and network scanning saving 20–40 hours/month. Automated vulnerability discovery provides 10–100x cheaper pre-deployment identification versus post-deployment patching with product recalls for security issues costing ₹10 lakhs-₹1 crore+ prevented through early discovery.

Conservative ROI calculation with ₹180,000 on-premises initial cost plus ₹50,000 annual maintenance versus ₹15 lakhs annual benefit from 30% breach probability avoidance ($0.3 \times ₹50$ lakhs = ₹15 lakhs) plus ₹3 lakhs saved analyst time totals ₹18 lakhs annual benefit. Year 1 ROI is (₹18 lakhs - ₹50,000 - ₹1.8 lakhs) / ₹2.3 lakhs $\times$ 100 = 680%. Year 2+ ROI is (₹18 lakhs - ₹50,000) / ₹50,000 $\times$ 100 = 3500%.

Comparison to commercial alternatives costing ₹5–20 lakhs/year in licenses, ₹2–10 lakhs implementation, and 15–20% annual support shows our system 3-year cost of ₹3.3 lakhs on-premises or ₹18 lakhs cloud (medium) versus commercial 3-year cost of ₹20 lakhs-₹80 lakhs+, providing cost savings of ₹16.7 lakhs-₹76.7 lakhs over 3 years. Overall Economic Feasibility Conclusion: Open-source foundation minimizes software costs, hardware costs are modest and typical for enterprise IT, development costs are primarily hardware in academic context, deployment costs vary but are manageable, ROI is strongly positive even with conservative estimates, and cost savings versus commercial alternatives are substantial. Verdict: ECONOMICALLY FEASIBLE.

3.4 Hardware Requirements

3.4.1 Development Environment Specifications

Minimum specifications include Intel Core i5–8400 or AMD Ryzen 5 2600 with 6 cores at 2.8 GHz base clock, 16 GB DDR4 RAM minimum for comfortable development with multiple VMs and containers, 256 GB SSD for operating system and development tools plus 1 TB HDD

for firmware image storage and datasets, integrated graphics sufficient though dedicated GPU recommended for deep learning, Gigabit Ethernet adapter at 1000 Mbps, and Ubuntu 20.04 LTS or later. These minimum specifications enable basic development activities including running development tools and IDEs, testing firmware emulation of single instances, training small-scale machine learning models, and running unit and integration tests, though parallel processing and large-scale testing will be limited.

Recommended specifications include Intel Core i7–11700K or AMD Ryzen 7 5800X with 8 cores at 3.6 GHz base and 4.9 GHz boost, 32 GB DDR4 RAM at 3200 MHz or faster, 512 GB NVMe SSD PCIe 3.0 or 4.0 for OS and development tools plus 2 TB HDD 7200 RPM for firmware images and datasets plus optional 500 GB external SSD for backups, NVIDIA GeForce RTX 3060 with 12 GB VRAM or better for GPU-accelerated deep learning, dual Gigabit Ethernet adapters for management and scanning isolated test networks, 27-inch 1440p monitor for improved productivity, mechanical keyboard and precision mouse, UPS for power protection, and Ubuntu 22.04 LTS. Recommended specifications enable efficient development with concurrent emulation of multiple firmware instances, rapid deep learning model training with GPU acceleration, smooth operation of resource-intensive IDEs, parallel testing workflows, and comfortable multitasking.

3.4.2   Production Deployment Hardware

Small-scale deployment suitable for organizations with 1–100 devices requires Intel Xeon E-2236 with 6 cores at 3.4 GHz or AMD EPYC 7232P with 8 cores at 3.2 GHz, 32 GB ECC RAM, 512 GB NVMe SSD plus 2 TB HDD or SSD, dual 10 Gigabit Ethernet adapters, optional 550W redundant power supply, 1U or 2U rackmount server or tower workstation form factor, estimated cost ₹150,000-₹250,000, and capacity to analyze 50–100 firmware images, scan networks up to 500 devices, and process 1000–5000 malware samples per day.

Medium-scale deployment suitable for organizations with 100–1000 devices requires dual Intel Xeon Silver 4314 with 32 cores total at 2.4 GHz or AMD EPYC 7452 with 32 cores at 2.35 GHz, 128 GB ECC RAM with 8×16GB modules for dual-channel, 1 TB NVMe SSD plus 8 TB RAID 10 array plus optional SAN or NAS for extended storage, optional but recommended NVIDIA Tesla T4 or A40 for high-throughput malware classification, dual 25 Gigabit Ethernet adapters with failover, dual redundant 750W power supplies, 2U rackmount server form factor, estimated cost ₹400,000-₹700,000, and capacity to analyze 500–1000 firmware images per day, scan networks with thousands of devices, and process 50,000+ malware samples per day with GPU.

Large-scale deployment suitable for enterprises with 1000+ devices uses distributed cluster architecture. Master node requires dual Intel Xeon Gold 6230 with 40 cores total, 256 GB ECC RAM, 2 TB NVMe SSD, and dual 100 Gigabit Ethernet. Worker nodes (3–10 nodes) require dual Intel Xeon Silver or AMD EPYC processors, 128 GB ECC RAM per node, 1 TB

NVMe SSD per node, and 25–100 Gigabit Ethernet. GPU nodes (1–3 nodes for ML inference) require Intel Xeon or AMD EPYC processors, 256 GB RAM, 4× NVIDIA A100 (40GB or 80GB) per node, 2 TB NVMe SSD, and 100 Gigabit InfiniBand or Ethernet. Enterprise SAN or distributed storage provides 50+ TB capacity. Estimated cost is ₹2,500,000-₹8,000,000 with capacity to analyze thousands of firmware images daily, scan enterprise networks with tens of thousands of devices, process millions of malware samples per day, and support multiple concurrent security analysts.

3.5   Software Requirements

3.5.1   Core System Software

Primary operating system is Ubuntu 22.04 LTS providing 5 years of security updates, extensive package repositories, strong community support, excellent Docker compatibility, and widespread industry adoption. Alternatives include Ubuntu 20.04 LTS, Debian 11/12, Rocky Linux 8/9, and CentOS Stream 8/9. Python environment requires Python 3.8, 3.9, 3.10, or 3.11 with pip 21.0+ for package installation, virtualenv or conda for environment isolation, and setuptools and wheel for package building. Containerization uses Docker CE 20.10+ and Docker Compose 1.29+ for multi-container orchestration with optional Kubernetes 1.24+ and Helm 3.x for large-scale distributed deployment.

3.5.2   Firmware Emulation Software

QEMU 5.0+ provides full system emulation with qemu-system-mips, qemu-system-arm, qemu-system-x86, and qemu-system-ppc packages supporting MIPS, ARM, x86, and PowerPC architectures. Firmware extraction tools include binwalk 2.3.3+ for firmware analysis and extraction, jefferson 0.4+ for JFFS2 filesystem extraction, sasquatch 1.0+ for SquashFS extraction with non-standard compression, firmware-mod-kit for additional manipulation, and unstuff, unrar, 7zip for archive extraction. System tools include busybox for minimal Unix utilities, netcat-openbsd for network connections and debugging, dnsmasq for lightweight DNS/DHCP, bridge-utils for network bridge configuration, and uml-utilities for user-mode Linux. Python libraries include python-magic 0.4+ for file type identification and configparser for configuration file parsing. PostgreSQL 12–15 stores metadata with psycopg2 2.9+ as PostgreSQL adapter for Python.

3.5.3   Network Scanner Software

Python core scanning libraries include scapy 2.4.5+ for packet manipulation and network scanning, python-nmap 0.7+ as Python wrapper for Nmap, and impacket 0.10+ for network protocol implementations. Network utilities include netifaces 0.11+ for network interface

enumeration, ipaddress built-in for IP address manipulation, and netaddr 0.8+ for advanced IP handling. Device identification uses mac-vendor-lookup 0.1+ for MAC to vendor mapping and requests 2.28+ as HTTP client for API calls. Parallelization employs concurrent.futures built-in for thread/process pools and asyncio built-in for asynchronous I/O. Data processing uses pandas 1.4+ for data manipulation and numpy 1.22+ for numerical computing. System tools include optional nmap 7.80+, arp-scan 1.9+ as alternative ARP scanner, and masscan 1.3+ as optional high-speed port scanner. MongoDB 5.0–7.0 stores flexible document data with pymongo 4.0+ as MongoDB driver for Python.

3.5.4 Malware Classifier Software

Primary deep learning framework is TensorFlow 2.10–2.13 with Keras API or alternatively PyTorch 1.13–2.1. Data science libraries include numpy 1.23+ for numerical arrays, pandas 1.5+ for data manipulation, scikit-learn 1.2+ for preprocessing and metrics, and scipy 1.10+ for scientific computing. Visualization uses matplotlib 3.6+ for plotting, seaborn 0.12+ for statistical visualization, and plotly 5.13+ for interactive visualizations. Optional GPU acceleration requires CUDA Toolkit 11.8 or 12.0, cuDNN 8.6 or 8.7, and TensorRT 8.5+ for optimized inference. Data handling uses h5py 3.8+ for HDF5 file format, pickle built-in for serialization, and joblib 1.2+ for efficient serialization. Model management optionally uses mlflow 2.1+ for experiment tracking and tensorboard 2.11+ for training visualization. Traffic capture uses pyshark 0.6+ as Python wrapper for tshark/WiShark, pcapy 0.11+ as interface to libpcap, and dpkt 1.9+ for packet creation and parsing.

3.5.5 System Integration Software

Web framework uses Flask 2.2+ or FastAPI 0.95+ for REST API implementation with flask-cors or fastapi-cors for Cross-Origin Resource Sharing and flask-jwt-extended or python-jose for JWT authentication. WSGI server uses gunicorn 20.1+ or uWSGI 2.0+ for production-grade serving. Web server uses nginx 1.22+ or Apache 2.4+ for reverse proxy and load balancing. Optional frontend uses Node.js 18 LTS, React 18+ or Vue.js 3+ framework, D3.js 7+ or Vis.js 9+ for visualizations, and Axios or Fetch API as HTTP client. Data storage uses Redis 7.0+ for caching and queuing with redis-py 4.5+ client, MinIO for S3-compatible object storage, or cloud alternatives AWS S3, Google Cloud Storage, Azure Blob Storage. Optional messaging uses RabbitMQ 3.11+ or Apache Kafka 3.4+ with celery 5.2+ for distributed task queuing. API documentation uses Swagger/OpenAPI 3.0 specification with Redoc or Swagger UI for interactive documentation.

3.6 Life Cycle Used

3.6.1 Incremental Software Development Model

The incremental model divides the project into multiple iterations delivering functional capability subsets. Each increment builds upon previous iterations adding features while refining existing functionality based on testing feedback. This approach aligns with our three-subsystem structure: firmware emulation, network scanning, and malware classification. Early functionality delivery provides usable capabilities enabling stakeholder evaluation and feedback before complete system delivery. Risk mitigation identifies technical challenges early by implementing core functionality first. Flexibility allows requirements evolution between increments based on feedback and lessons learned. Parallel development enables team members working on separate increments concurrently after establishing architectural foundations. The model provides iterative enhancement, stakeholder engagement, and systematic development suitable for our modular architecture.

3.6.2 Machine Learning Development Lifecycle

For malware classification, we follow specialized Machine Learning Development Lifecycle addressing unique ML requirements. The MDLC includes problem definition and scoping precisely defining classification goals and metrics, data collection and ingestion acquiring labeled IoT-23 dataset [8], exploratory data analysis understanding distributions and quality issues, data preprocessing and feature engineering transforming raw traffic into ML-ready features, model selection and architecture design choosing CNN approach and defining network structure, model training implementing training loops with proper validation, model evaluation calculating comprehensive performance metrics, model optimization and tuning improving performance through hyperparameter adjustment, model deployment exporting and serving models via API, and model monitoring and maintenance tracking performance and planning retraining. This structured approach ensures data quality emphasis, iterative refinement, deployment readiness, and reproducibility essential for effective ML development.

3.6.3 Project Increments Schedule

Increment 1 (Weeks 1–8) establishes firmware emulation foundation including development environment setup, firmware extraction implementation, basic QEMU emulation, initial boot arbitration, PostgreSQL database schema, and achieving basic emulation for simple images. Increment 2 (Weeks 6–11) implements network scanner including ARP-based discovery, TCP port scanner with multi-threading, MAC vendor lookup integration, MongoDB schema, command-line interface, and result visualization. Increment 3 (Weeks 9–16) develops deep learning classifier including IoT-23 dataset acquisition and preprocessing, CNN architecture design,

training pipeline implementation, initial model training achieving >95% accuracy, inference pipeline development, and model serving API creation. Increment 4 (Weeks 14–20) achieves system integration including unified data model design, RESTful API implementation for all modules, web dashboard development, workflow orchestration, alert generation system, and comprehensive reporting functionality. Increment 5 (Weeks 18–24) focuses on performance optimization including firmware emulation optimization, network scanning performance improvements, model inference speed optimization, Docker containerization, and comprehensive documentation creation. Increment 6 (Weeks 22–28) conducts testing and validation including unit testing, integration testing, performance benchmarking, security testing, and real-world validation with bug fixes. Increment 7 (Weeks 26–30) completes final documentation and presentation including complete project report, presentation materials, demonstration scenarios, packaged deliverables, and final quality assurance.

3.7 Software Cost Estimation

3.7.1 COCOMO Model Application

We employ Constructive Cost Model (COCOMO) for algorithmic cost estimation. COCOMO relates development effort to code size through empirical equations from analysis of hundreds of projects. Basic COCOMO uses equations: Effort (E) = a $\times$ (KLOC)^b person-months, Development Time (D) = c $\times$ (E)^d months, and Personnel (P) = E / D. Our project classification is Semi-Detached due to medium complexity, mixed team experience, some innovative requirements, security-sensitive but not life-critical nature, and moderate technical constraints. For Semi-Detached projects, coefficients are a = 3.0, b = 1.12, c = 2.5, d = 0.35.

3.7.2 Code Size and Cost Calculation

Estimated code size totals 25.1 KLOC including firmware emulation module (4,500 LOC), network scanner module (3,200 LOC), malware classifier module (4,400 LOC), system integration (4,200 LOC), web dashboard (5,000 LOC), and testing and documentation (3,800 LOC). Cost calculation yields Effort E = 3.0 $\times$ (25.1)$^{1.12}$ = 106.26 person-months, Development Time D = 2.5 $\times$ (106.26)$^{0.35}$ = 12.7 months, and Personnel P = 106.26 / 12.7 = 8.37 people. For our 4-person team, adjusted development time is 106.26 / 4 = 26.6 person-months or approximately 6.65 months per person, though realistic project duration is 10–12 months accounting for sequential dependencies, learning curves, integration overhead, and academic commitments.

In academic context, software development cost is ₹0 as student labor is not salaried, making total project cost equal to hardware cost only at ₹174,700. For industry context assuming ₹55,000/month average salary, total development cost is 106.26 $\times$ ₹55,000 = ₹58,44,300, and total project cost including hardware is ₹58,44,300 + ₹174,700 = ₹60,19,000. Our estimates

align with industry benchmarks as security software typically ranges 20–50 KLOC and our 25.1 KLOC falls within this range. The 10–12 month timeline aligns with typical two-semester academic projects.

3.8 Hardware Cost Estimation

Development hardware costs include development workstation with Intel i7–11700K, 32GB RAM, 512GB NVMe SSD, RTX 3060 12GB costing ₹120,000; 2TB 7200 RPM HDD for firmware images costing ₹4,500; managed Gigabit 8-port switch costing ₹3,500; two Raspberry Pi 4 (8GB) units for test IoT network costing ₹14,000; ten Cat6 Ethernet cables costing ₹1,500; USB-to-Serial FTDI adapter for debugging costing ₹800; two surge-protected power strips costing ₹2,400; 500GB USB 3.1 SSD for backups costing ₹5,500; 27" 1440p IPS display costing ₹18,000; and mechanical keyboard plus precision mouse costing ₹4,500. Total hardware cost is ₹174,700. Component justification includes high-performance workstation essential for running multiple QEMU VMs, training deep learning models efficiently, compiling code, and operating development tools with RTX 3060 GPU providing 10–50x speedup versus CPU. Cost-effective HDD provides sufficient capacity for firmware image collections. Managed switch enables creating isolated test networks, VLAN configuration, and traffic monitoring. Raspberry Pi units simulate real IoT devices. Additional quality peripherals improve productivity during extended development sessions.

3.9 Total Product Cost Estimation

3.9.1 Academic and Commercial Context Costs

For academic context, total project cost equals hardware cost only at ₹174,700 (one lakh seventy-four thousand seven hundred rupees only) since software development cost is ₹0 for student work. For industry/commercial context, total project cost is ₹60,19,000 (sixty lakhs nineteen thousand rupees only) including ₹58,44,300 software development plus ₹174,700 hardware.

3.9.2 Cost-Benefit Analysis

Tangible annual benefits include prevented security breaches worth ₹15–50 lakhs per year assuming 30–50% breach probability without system at average ₹50 lakhs-₹1 crore breach cost, saved labor costs worth ₹2.5–5 lakhs per year from 40–80 hours monthly time savings at ₹50,000–100,000 monthly analyst salary, automated vulnerability discovery providing 10–100x cheaper pre-deployment identification versus post-deployment fixes, and avoided regulatory fines worth ₹2–20 lakhs per year with 20–40% probability reduction. Total annual tangible benefits conservatively estimate ₹20–80 lakhs.

ROI calculation for Year 1 academic context with ₹20 lakhs benefits and ₹1.75 lakhs costs yields ROI = ((₹20,00,000 - ₹1,75,000) / ₹1,75,000) × 100 = 1043%. Commercial context Year 1 with ₹20 lakhs benefits and ₹60 lakhs costs yields ROI = -67% (negative due to development costs), but Year 2 with ₹20 lakhs benefits and ₹0.5 lakhs maintenance yields ROI = 3900%. Break-even occurs approximately at 3 years with conservative estimates, earlier with realistic ₹40–80 lakhs annual benefits.

3.9.3 Comparison to Commercial Alternatives

Commercial IoT security platforms cost ₹5–20 lakhs per year in enterprise licenses, ₹2–10 lakhs for professional services implementation, and ₹0.75–4 lakhs per year for annual support (15–20% of license), totaling ₹18–72 lakhs over 3 years. Our system 3-year cost is ₹1.75 lakhs academic or ₹61.5 lakhs commercial versus commercial alternatives, providing cost savings of ₹16.25–70.25 lakhs in academic context. Additional advantages include unlimited customization with full source code access, no vendor lock-in maintaining control, complete methodology transparency, community support enabling contributions, and intellectual property ownership.

3.10 Project Scheduling using Gantt Chart

3.10.1 Phase Timeline Overview

Phase 1 Planning and Setup (Weeks 1–3) includes literature review, requirements analysis, environment setup, hardware procurement, team role assignment, and project plan finalization. Phase 2 Firmware Emulation (Weeks 3–10) implements firmware extraction, QEMU emulation setup, arbitration techniques, PostgreSQL integration, and testing. Phase 3 Network Scanner (Weeks 6–13) develops ARP scanner, TCP port scanner, vendor identification, multi-threading optimization, MongoDB integration, and validation. Phase 4 Malware Classifier (Weeks 9–18) handles dataset acquisition, preprocessing pipeline, CNN architecture design, model training and validation, inference pipeline, model serving API, and evaluation. Phase 5 System Integration (Weeks 14–22) creates REST API design, database schema integration, web dashboard development, workflow orchestration, alert generation, report generation, and integration testing. Phase 6 Testing and Validation (Weeks 18–26) includes unit test development, integration testing, performance benchmarking, security testing, bug fixes, and real-world validation. Phase 7 Documentation and Deployment (Weeks 22–28) produces user documentation, developer documentation, Docker containerization, deployment guide, and tutorial creation. Phase 8 Final Report and Presentation (Weeks 26–30) completes project report writing, presentation preparation, demonstration setup, final review and corrections, and final submission.

3.10.2 Critical Path and Milestones

Critical path flows Phase 1 → Phase 2 → Phase 4 → Phase 5 → Phase 6 → Phase 8 with 30-week duration. Phase 2 firmware emulation must complete before full integration. Phase 4 malware classifier has longest duration and cannot be significantly shortened. Phase 5 integration depends on all three core modules. Phase 6 testing must complete before finalization. Phase 8 report depends on all previous phases. Any critical path delays directly impact completion date. Key milestones include M1 (Week 5) firmware extraction working for 10+ test images, M2 (Week 8) basic emulation successful, M3 (Week 10) arbitration achieving >70% success, M4 (Week 8) ARP scanner discovering all test devices, M5 (Week 11) TCP scanner identifying ports correctly, M6 (Week 13) Class C scan within 5 minutes, M7 (Week 12) dataset preprocessed and ready, M8 (Week 16) model achieving >90% validation accuracy, M9 (Week 18) inference API processing 100+ samples/minute, M10 (Week 16) APIs for all modules functional, M11 (Week 20) dashboard displaying integrated data, M12 (Week 22) end-to-end workflow executing, M13 (Week 21) unit tests passing for all modules, M14 (Week 24) performance targets achieved, M15 (Week 26) no critical bugs remaining, M16 (Week 25) Docker deployment successful, M17 (Week 28) documentation complete, M18 (Week 29) report draft complete, and M19 (Week 30) final submission and presentation.

3.10.3 Risk Management Strategy

Technical risks include low firmware emulation success rate with medium probability and high impact, mitigated by allocating buffer time for additional arbitration development with contingency to reduce firmware variety if success remains below 60%. Insufficient training data quality has low probability and high impact, mitigated by early dataset acquisition and exploring multiple sources with contingency using transfer learning and data augmentation. Integration complexity has medium probability and medium impact, mitigated by designing clear module interfaces early and incremental integration testing with contingency simplifying integration by focusing on core functionality. Schedule risks include module development delays with medium probability and medium impact, mitigated by parallel development and front-loading critical activities with contingency reallocating resources. Hardware procurement delays have low probability and low impact, mitigated by immediate ordering upon approval with contingency using cloud resources temporarily. Resource risks include team member unavailability with medium probability and medium impact, mitigated by cross-training and comprehensive documentation with contingency redistributing tasks. Contingency buffer incorporates 2–3 weeks distributed as Phase 2 buffer (1 week at Week 10), Phase 4 buffer (1 week at Week 17), Phase 6 buffer (1 week at Week 25), and final buffer (1 week at Week 29) absorbing minor delays without impacting completion.

Chapter 4

METHODOLOGY

4.1 Proposed System

4.1.1 System Architecture Overview

Our FIRMAI system employs modular microservices-inspired architecture with three primary subsystems operating in coordination. The layered three-tier architectural pattern separates concerns enabling independent development and scaling. Tier 1 Presentation Layer provides web dashboard with interactive visualization, REST API with JSON format supporting JWT authentication and versioned endpoints, and command-line interface supporting scripting with comprehensive help. Tier 2 Business Logic Layer implements firmware analysis orchestrator managing emulation lifecycle, network scanning coordinator controlling operations across address spaces, malware classification pipeline processing traffic through ML models, and integration engine correlating findings across subsystems generating unified reports. Tier 3 Data Layer uses PostgreSQL for firmware metadata with structured schema, MongoDB for semi-structured scan results and malware events, Redis for job queuing and caching, and object storage like MinIO/S3 for firmware images and model weights.

4.1.2 Firmware Emulation Subsystem Design

The firmware emulation subsystem automatically extracts, emulates, and analyzes IoT firmware with minimal intervention. Firmware ingestion service handles image upload accepting files via web or file path, validates integrity, extracts metadata, assigns unique identifiers, stores in object storage, and creates database records. Extraction engine executes binwalk identifying embedded filesystems, recursively extracts nested archives, identifies compressed formats, locates kernel images, determines processor architecture, and generates filesystem tree representation. Emulation orchestrator selects appropriate QEMU emulator based on architecture, configures virtual hardware, selects matching kernel, sets up virtual networking, applies arbitration techniques, launches QEMU, monitors boot process, and detects failures applying interventions. Arbitration engine implements boot arbitration modifying kernel command-line parameters and injecting

boot scripts, network arbitration creating interfaces with expected names and forcing configurations, NVRAM arbitration emulating storage as persistent files, kernel arbitration maintaining pre-compiled kernel library, and environment arbitration creating device nodes and establishing expected paths. Vulnerability scanner enumerates exposed services, fingerprints versions, queries vulnerability databases, executes exploit modules, performs fuzzing, and documents successful exploits. Technology stack uses Python 3.10 for orchestration, binwalk 2.3+ for extraction, QEMU 5.2+ for emulation, PostgreSQL 14 for metadata, and Docker for containerized isolation.

4.1.3 Network Scanning Subsystem Design

The network scanning subsystem discovers IoT devices, identifies characteristics, and maps network topologies efficiently. Scan controller manages operations by validating CIDR input, generating IP address lists, distributing workload across worker threads, aggregating parallel scan results, handling timeouts and retries, and storing results in MongoDB. ARP scanner implements Layer 2 discovery constructing ARP request packets using Scapy, sending broadcast requests, capturing responses, extracting IP/MAC pairs, recording timestamps, and handling rate limiting. TCP scanner performs Layer 4 scanning implementing SYN scanning for speed, testing configurable port ranges, handling timeouts appropriately, identifying port states, performing banner grabbing, and recording version information. Device identifier determines types and vendors by extracting OUI from MAC addresses, querying vendor databases updated from IEEE, analyzing port patterns against device signatures, parsing service banners, applying heuristic rules, and assigning confidence scores. Result aggregator combines findings correlating ARP and TCP results by IP, merging vendor information with details, generating device inventory, calculating topology relationships, and exporting in multiple formats. Technology stack uses Python 3.10, Scapy 2.5+ for packets, python-nmap for enhanced scanning, mac-vendor-lookup for identification, MongoDB 6.0 for storage, and concurrent.futures for multi-threading.

4.1.4 Malware Classification Subsystem Design

The malware classification subsystem uses deep learning to identify and categorize IoT malware with high accuracy. Traffic capture module captures network traffic interfacing with libpcap or pyshark, supporting configurable BPF filters, implementing circular buffers for continuous capture, providing start/stop controls, and exporting PCAP format. Preprocessing pipeline extracts features including packet size statistics, protocol distributions, port frequencies, inter-arrival times, TCP flags, and payload byte frequency, then normalizes using standardization for normal distributions and min-max scaling for skewed features, encodes categorical variables using one-hot or label encoding, balances classes using SMOTE or undersampling, and splits into train/validation/test sets ensuring balance. CNN model architecture includes input layer accepting preprocessed features, feature learning layers with Dense 512/256/128/64 neurons

using ReLU activation with batch normalization and dropout (40%/40%/30%), and output layer with softmax activation producing class probabilities. Training configuration uses Adam optimizer with 0.001 learning rate, categorical cross-entropy loss, batch size 64, 50 epochs with early stopping patience 10, and 20% validation split. Inference engine loads trained weights on startup, accepts preprocessed features, performs batch inference, applies softmax, returns predictions with confidence scores, and logs classifications. Model serving API provides REST endpoint accepting JSON traffic features with asynchronous processing for high throughput. Technology stack uses TensorFlow 2.12 with Keras, NumPy 1.24 and Pandas 2.0, Scikit-learn 1.2, FastAPI for serving, and CUDA 11.8 + cuDNN 8.6 for GPU acceleration.

4.1.5 System Integration Architecture

The integration layer unifies three subsystems into cohesive platform. API gateway provides single entry point routing requests to appropriate subsystems, implements authentication and authorization, enforces rate limiting, aggregates responses, handles versioning, and provides error responses. Message queue enables asynchronous processing using Redis-based job queue with Celery framework for firmware analysis jobs, network scan requests, and classification tasks with results published to subscribers. Data correlation engine links findings across subsystems correlating by device identifiers including MAC, IP, and firmware hash, joins firmware vulnerabilities with network results, associates malware detections with affected devices, calculates aggregate risk scores, and identifies attack patterns. Alert generator produces timely notifications monitoring for high-severity findings including CVSS $\geq$ 7.0 vulnerabilities and high-confidence malware detections, evaluates against configurable rules, applies suppression preventing duplicates, formats notifications with relevant context, and delivers via multiple channels including email, webhooks, and dashboard. Report generator creates comprehensive assessments aggregating findings from all subsystems, generates executive summaries, produces detailed technical sections, creates visualizations, exports in multiple formats, and supports scheduled and on-demand generation. Web dashboard provides unified interface as modern single-page application with React/Vue.js, real-time WebSocket updates, interactive D3.js visualizations, responsive design, role-based access control, and comprehensive search and filtering.

4.2 Advantages of Proposed System

4.2.1 Comprehensive Security Coverage

Our system addresses threats across multiple IoT stack layers simultaneously. Firmware layer analysis identifies vulnerabilities in device firmware code, misconfigurations, hardcoded credentials and backdoors, insecure cryptographic implementations, and buffer overflows. Network

layer discovery provides complete network visibility, port scanning reveals exposed services, traffic analysis detects malicious communications, and topology mapping identifies unusual relationships. Application layer classification identifies known and novel malware threats, behavioral analysis detects suspicious patterns, family classification aids incident response, and real-time detection enables rapid response. Cross-layer correlation reveals complex attack patterns as the integration engine automatically connects findings across layers. Traditional separate tools identify findings independently requiring manual analyst correlation, while our automatic correlation dramatically reduces time-to-detection and response enabling proactive threat mitigation.

4.2.2   Automation and Scalability

FIRMAI design maximizes automation eliminating tedious manual tasks. Manual firmware reverse engineering requires days to weeks per device while our automated emulation processes 50+ images per hour. Manual device inventory in large networks is impractical while our scanner completes Class C subnets in under 5 minutes. Traditional malware analysis requires specialized expertise and significant time per sample while our trained model classifies 1000+ samples per minute with GPU. Automation enables comprehensive assessment rather than small subset sampling, dramatically improving security posture. Parallel processing throughout the system includes Docker containerization enabling concurrent firmware emulation with isolation, multi-threaded network scanning distributing workload, batch malware inference processing multiple samples simultaneously with GPU providing 10–50x speedup, and multiple GPUs for higher throughput. This parallel architecture scales from single-server deployments to distributed clusters handling enterprise-scale workloads.

4.2.3   Advanced AI Capabilities

Our deep learning model automatically learns complex malware patterns. Automatic feature learning uses CNN discovering relevant features through training without manual engineering while multi-layer architecture learns hierarchical representations from low-level packet features to high-level behavioral patterns. Generalization enables detecting novel malware variants exhibiting similar behaviors with transfer learning adapting to new families using limited labeled examples. Adaptation through periodic retraining keeps models current with evolving threats while new families can be added without architectural changes. High accuracy achieves >99% on IoT-23 test set [8] with >95% precision and recall per family and low false positive rate <1% reducing alert fatigue. Real-world validation on actual malware samples confirms effectiveness with performance competitive or superior to published research. Confidence scoring provides calibrated probabilities allowing low-confidence predictions flagged for manual review.

4.2.4   Cost-Effectiveness

Open-source foundation provides substantial advantages. Zero licensing costs use freely available components including QEMU, Python, TensorFlow, PostgreSQL, MongoDB, and Docker. No vendor lock-in with complete source code access prevents dependency on single vendor while organizations maintain full control. Community support provides bug fixes and enhancements with extensive documentation reducing learning curves. Commercial platforms cost ₹5–20 lakhs annually in licensing plus implementation and support fees which our system eliminates. Commodity hardware runs on standard servers without specialized equipment including consumer-grade workstation (₹1.2 lakhs) for development, standard rackmount servers (₹1.5–7 lakhs) for production, and flexible cloud deployment eliminating upfront expenditure. Preventive savings deliver ROI through breach prevention averaging ₹50 lakhs-₹1 crore for SMBs with even one prevented incident justifying investment, early vulnerability detection being 10–100x cheaper than post-deployment patching, and labor efficiency saving 100+ analyst hours monthly worth ₹2–5 lakhs annually.

4.2.5   Flexibility and Extensibility

Modular architecture with well-defined interfaces enables extending capabilities including additional scanners for Bluetooth, Zigbee, or custom protocols, alternative ML models like RNN or transformer architectures, organization-specific analysis modules, and custom export formats and visualizations. Configurable workflows allow users customizing system behavior through scanning parameters adjusting port ranges and timeouts, emulation settings configuring resource limits, classification thresholds tuning confidence levels, and alert rules defining custom generation logic. Integration-friendly standard interfaces include REST APIs enabling integration with any HTTP-supporting system, standard formats JSON/CSV/PDF working with existing tools, webhooks pushing notifications to external systems, SIEM integration exporting to Splunk/ELK/QRadar, and SOAR integration triggering automated response playbooks. Educational and research value provides hands-on learning with firmware analysis exposing low-level concepts, network security with packet manipulation experience, machine learning with practical application, and system integration with distributed systems experience. Reproducible research through open source complete code, methodology documentation enabling replication, test data facilitating validation, and performance metrics enabling comparison contributes to broader security ecosystem.

4.3   System Components and Integration

4.3.1   Firmware Emulation Implementation

Firmware extraction implementation uses binwalk as primary tool with FirmwareExtractor class handling extraction running binwalk with signature scanning and extraction, handling squashfs with sasquatch for custom compression, extracting JFFS2 filesystems, identifying architecture by analyzing kernel images, and locating kernels in extraction directory. QEMU emulation orchestration through FirmwareEmulator class dynamically conffigures based on firmware characteristics, selects appropriate pre-compiled kernel matching version and configuration, builds QEMU command line specifying machine type, kernel, rootfs drive, kernel append parameters, network configuration with TAP interfaces, and memory allocation, starts emulation launching QEMU process and monitoring boot for success indicators or failure detection. Arbitration technique implementation uses strategy pattern with ArbitrationStrategy base class and specific strategies including BootArbitration modifying kernel cmdline forcing shell as init, NetworkArbitration forcing interface creation with generated network scripts, and NVRAMArbitration emulating storage creating NVRAM defaults as persistent files. Implementation uses Python with subprocess for process management, file manipulation for filesystem operations, and systematic intervention application when failures occur.

4.3.2   Network Scanner Implementation

ARP scanner implementation uses Scapy for low-level packet control with ARPScanner class performing scans creating ARP request packets for specified CIDR ranges, broadcasting with Ethernet destination ff:ff:ff:ff:ff:ff, sending and receiving with configurable timeout, and parsing responses extracting IP psrc and MAC hwsrc addresses returning results list with IP, MAC, and timestamp. TCP port scanner implementation uses multi-threading for performance with TCP-Scanner class managing concurrent scans using ThreadPoolExecutor distributing port scanning across threads, scanning individual ports using socket with connect_ex testing connectivity, collecting open ports for each target, and grabbing banners connecting to open ports receiving initial response data. Device identification implementation with DeviceIdentifier class performs vendor lookup using mac-vendor-lookup querying updated OUI database, handles unknown vendors gracefully, classifies device types analyzing port signatures matching routers by {80, 443, 22, 23} ports with vendor keywords, IP cameras by {554, 8000, 8080} with camera vendor keywords, IoT hubs by {1883, 8883, 5683} MQTT/CoAP ports, and returns device type with confidence score.

4.3.3   Malware Classifier Implementation

Preprocessing pipeline implementation with TrafficPreprocessor class extracts features from PCAP files using rdpcap reading packets and extracting size, protocol, ports, TCP flags, inter-arrival times, aggregating to flow-level statistics, preprocessing feature matrices handling missing values with mean imputation, encoding categorical variables using LabelEncoder, scaling numerical features with StandardScaler, and handling class imbalance using SMOTE over-sampling. CNN model implementation uses TensorFlow/Keras with MalwareClassifier class building architecture with Input layer accepting num_features dimensions, Dense layers with 512/256/128/64 neurons using ReLU activation, BatchNormalization after each Dense layer, Dropout regularization with 0.4/0.4/0.3/0.0 rates, and output Dense layer with softmax activation for num_classes, compiling with Adam optimizer at 0.001 learning rate and categorical crossentropy loss, training with EarlyStopping callback monitoring val_loss with patience 10, ModelCheckpoint saving best weights, and TensorBoard logging, performing inference with predict generating probability distributions and argmax selecting highest probability classes with max extracting confidence scores.

4.3.4   System Integration Implementation

REST API implementation uses FastAPI creating unified interface with firmware analysis endpoints for upload accepting UploadFile creating firmware_id and enqueuing analysis, status checking querying database for firmware_id status, and results retrieval returning complete analysis findings. Network scanning endpoints initiate scans accepting ScanRequest with CIDR range and type, enqueuing scan job returning scan_id, and retrieving results querying scan_results database. Malware classification endpoints classify traffic accepting ClassificationRequest with base64 PCAP data, decoding and extracting features, performing inference, and returning classification with confidence. Integrated endpoints provide dashboard summary aggregating firmware/network/malware statistics with calculated risk scores. Celery task queue implementation uses Redis broker with analyze_firmware_task performing extraction using FirmwareExtractor, emulation using FirmwareEmulator, vulnerability scanning using VulnerabilityScanner, storing results in database, and returning status with firmware_id. scan_network_task performs ARP scanning using ARPScanner, TCP scanning using TCP-Scanner, device identification using DeviceIdentifier, compiling device list with all attributes, storing scan results, and returning status with scan_id. Docker Compose configuration defines multi-container deployment with postgres service for firmware metadata, mongodb for scan results, redis for caching and queuing, minio for object storage, api service running FastAPI, worker service running Celery, and dashboard service running React/Vue frontend, using volumes for data persistence and environment variables for configuration.

Chapter 5

SYSTEM IMPLEMENTATION

5.1 Introduction

This chapter explains the complete implementation of the proposed system FIRMAI: AI-Powered IoT Firmware Vulnerability Analyzer. It describes how the system architecture designed in the previous chapters is translated into a working software solution. The implementation focuses on integrating firmware analysis, network scanning, and AI-based malware detection into a single unified platform. Special emphasis is given to modular design, automation, scalability, and security.

The system is implemented using open-source tools and modern software frameworks to ensure flexibility, reliability, and ease of deployment. Each module is developed independently and later integrated through well-defined interfaces to provide end-to-end IoT security analysis.

5.2 Overall System Implementation Architecture

The FIRMAI system follows a modular and service-oriented architecture, where each functional component operates as an independent module. These modules communicate using RESTful APIs and asynchronous message queues, ensuring loose coupling and high scalability.

The implementation is divided into four logical layers:

- **Presentation Layer** – Provides web-based dashboard and API interfaces for user interaction.

- **Application Logic Layer** – Implements firmware analysis, network scanning, vulnerability detection, and malware classification logic.

- **Data Management Layer** – Handles structured and unstructured data storage.

- **Infrastructure Layer** – Manages containerization, networking, and deployment services.

5.3   Firmware Acquisition and Processing Module

The firmware acquisition module is responsible for collecting IoT firmware images for analysis. Firmware images can be obtained through multiple methods such as manual upload, direct extraction from IoT hardware devices, or interception of Over-The-Air (OTA) update traffic using a transparent proxy.

Once a firmware image is collected, it is processed using Binwalk, which identifies embedded filesystems, compressed archives, and executable binaries. The extraction process recursively unpacks nested components and reconstructs the complete filesystem structure.

Important metadata such as processor architecture, kernel version, filesystem type, and vendor-specific details are extracted and stored in the database. This information is essential for selecting appropriate emulation environments and vulnerability testing strategies.

5.4   Firmware Emulation Implementation

Firmware emulation is a core component of FIRMAI and is implemented using QEMU, a widely used open-source hardware emulator. Based on the detected architecture (ARM, MIPS, x86, etc.), the system automatically selects the appropriate QEMU emulator.

A customized Linux kernel is loaded along with the extracted firmware filesystem. Since many IoT firmware images use non-standard boot mechanisms, an emulation arbitration mechanism is implemented. This includes:

- Modifying kernel boot parameters

- Emulating NVRAM using file-based storage

- Creating expected device nodes

- Dynamically configuring network interfaces

5.5   Vulnerability Analysis Engine Implementation

After successful firmware emulation, the vulnerability analysis engine performs automated security assessments. The engine first enumerates all exposed services and identifies their versions. These versions are correlated with known vulnerabilities using public CVE databases.

In addition to signature-based detection, dynamic testing techniques such as fuzzing are applied to web interfaces and network services. Input parameters are mutated to trigger unexpected behavior such as crashes, authentication bypasses, or command injection. All detected vulnerabilities are classified based on severity and documented with evidence. The system generates detailed vulnerability reports including CVE IDs, risk scores, affected services, and recommended mitigation steps.

5.6 Network Scanning and Device Discovery Module

The network scanning module identifies IoT devices connected to the target network. ARP-based scanning is used for fast device discovery at the data link layer, enabling detection even in environments with restrictive firewall rules.

Once devices are discovered, multi-threaded TCP scanning is performed to identify open ports and running services. Banner grabbing techniques are used to extract service information such as software names and versions.

Each discovered device is cataloged with details including IP address, MAC address, open ports, service banners, and response time. This information is later correlated with firmware vulnerabilities and malware detection results.

5.7 Device Identification and Fingerprinting

Device identification is performed by analyzing MAC address OUIs, open port patterns, and service banners. The system maps MAC address prefixes to vendor information using standard vendor databases.

Service fingerprints and port combinations are matched against known IoT device signatures to determine device type and manufacturer. Confidence scores are assigned to each identification result, ensuring accuracy and transparency.

5.8 Malware Detection and AI Model Implementation

The malware detection module uses Artificial Intelligence to classify network traffic as benign or malicious. Network packets are captured using libpcap and processed into flow-level features such as packet size distribution, protocol usage, port frequency, and inter-arrival times.

A Convolutional Neural Network (CNN) is implemented using TensorFlow and Keras. The model is trained on labeled IoT malware datasets, enabling it to learn behavioral patterns of various malware families.

During real-time operation, captured traffic is passed through the trained model, which outputs classification results along with confidence scores. High-confidence malicious detections trigger alerts and are logged for further analysis.

5.9 System Integration and Data Correlation

All modules in FIRMAI are integrated using RESTful APIs and asynchronous message queues. Each module publishes its results to a central data repository.

A correlation engine links findings from different modules using common identifiers

such as IP address, MAC address, and firmware hash. This enables the system to associate firmware vulnerabilities with live network behavior and detected malware activity.

The integrated view allows security analysts to understand attack paths and identify high-risk devices efficiently.

5.10   Dashboard and User Interface Implementation

The user interface is implemented as a web-based dashboard that provides real-time visibility into system operations. The dashboard displays:

- Discovered IoT devices

- Firmware vulnerability details

- Malware detection alerts

- Risk scores and analysis reports

Role-based access control ensures that only authorized users can access sensitive information. Users can generate and download reports in multiple formats for documentation and compliance purposes.

5.11   Deployment and Containerization

The entire FIRMAI system is containerized using Docker, ensuring consistent deployment across different environments. Docker Compose is used to orchestrate multiple services such as backend APIs, databases, AI model servers, and the dashboard.

Containerization simplifies installation, scaling, and maintenance. The system can be deployed on local servers, institutional labs, or cloud platforms with minimal configuration changes.

5.12   Security Considerations

Security is enforced throughout the implementation. All communications between system components are secured using HTTPS. Sensitive data such as credentials and vulnerability information are encrypted at rest.

Authentication and authorization mechanisms prevent unauthorized access, and audit logs are maintained for all critical actions. Regular security testing ensures that the system itself does not introduce new vulnerabilities.

Chapter 6

TESTING AND RESULTS

6.1   Introduction

This chapter discusses the testing strategies used to evaluate the performance, correctness, and reliability of the FIRMAI: AI-Powered IoT Firmware Vulnerability Analyzer. Since the system integrates firmware emulation, network scanning, and artificial intelligence–based malware detection, it is essential to validate each module independently and also as an integrated system. Testing ensures that the system meets its functional and non-functional requirements and performs accurately in real-world IoT environments.

Different levels of testing were carried out including unit testing, integration testing, system testing, and performance testing. In addition, validation of the AI-based malware detection model was performed using standard evaluation metrics.

6.2   Testing Strategy

The testing of FIRMAI was carried out using a structured approach to ensure full coverage of all modules. The following testing methods were used:

- Unit Testing

- Integration Testing

- System Testing

- Performance Testing

- Security Testing

- AI Model Validation

6.3   Unit Testing

Unit testing was performed to verify the functionality of individual modules in isolation.

6.3.1 Firmware Processing Module

The firmware extraction module was tested using multiple firmware formats such as.bin,.img, and compressed archives. Binwalk was verified for correct extraction of filesystem structures, kernel images, and configuration files. The output was checked for accuracy and completeness.

6.3.2 Network Scanning Module

The ARP and TCP scanning functions were tested using known test networks containing routers, cameras, and IoT boards. The module was verified for correct IP detection, MAC address extraction, and port scanning.

6.3.3 Malware Detection Module

The feature extraction and CNN classification functions were tested independently to verify that valid inputs produced expected outputs. Error handling for missing or malformed data was also validated.

6.4 Integration Testing

Integration testing was carried out to ensure that all modules worked correctly when combined. Firmware emulation results were passed to the vulnerability scanner to verify service detection. Network scanning results were integrated with the malware detection engine to correlate infected devices. The dashboard was tested to ensure it correctly displayed aggregated results from all subsystems.

All modules were found to interact successfully without data loss or inconsistency.

6.5 System Testing

System testing evaluated the complete FIRMAI platform in a real-world-like environment. The system was deployed using Docker and tested on a local network containing IoT devices such as routers and ESP32 boards.

The system successfully:

- Detected all active devices

- Emulated firmware images

- Identified vulnerable services

- Detected malicious traffic

6.6   Security Testing

Security testing was performed to ensure the system itself was secure. The following were verified:

- API endpoints were protected using authentication

- Data was encrypted

- Unauthorized access was blocked

- Input validation prevented injection attacks

Chapter 7

CONCLUSIONS

This project successfully developed FIRMAI, an integrated AI-powered security framework for comprehensive IoT vulnerability analysis combining firmware emulation, network scanning, and intelligent malware detection. We achieved all primary objectives establishing unified platform eliminating fragmentation, implementing high-efficiency firmware emulation with 78.4% success rate approaching 5× improvement over baseline 16%, creating comprehensive network discovery completing Class C scans in under 5 minutes with 88.5% device classification accuracy, and deploying AI-powered malware detection achieving 99.28% classification accuracy with GPU-accelerated real-time processing capability.

7.1 Future Enhancements

Several promising directions exist for extending FIRMAI capabilities. Enhanced firmware analysis could incorporate automated exploit generation, machine learning-based vulnerability prediction, symbolic execution integration, and firmware patching automation. Advanced network capabilities could add support for additional protocols including BLE, Zigbee, Z-Wave, enhanced topology inference using machine learning, anomaly detection for unusual behaviors, and SDN/NFV integration. Malware detection improvements could implement online learning, adversarial robustness training, explainable AI, and transfer learning. System scalability enhancements could deploy distributed architecture, implement horizontal scaling with Kubernetes, add GPU cluster support, and optimize resource allocation. User experience improvements could develop mobile applications, implement advanced visualization, add natural language query interface, and create automated remediation workflows.

7.2 Limitations

Despite significant achievements, limitations exist. Firmware emulation limitations include 78.4% success rate meaning 21.6% cannot be emulated, limited PowerPC support, timeout constraints, and NVRAM emulation imperfections. Network scanning limitations include primarily IPv4 focus, Layer 2 ARP requiring same subnet, firewall interference, and passive discovery

absence. Malware classification limitations include training data dependency, novel threat challenges, false positive rate of 0.28%, and protocol-specific focus. Integration limitations include manual configuration required, single-server deployment in current implementation, limited SIEM integration, and customization complexity. Performance limitations include GPU dependence, resource requirements, parallel scaling limits, and database optimization opportunities.

7.3 Final Remarks

The development of FIRMAI represents significant advancement in IoT security addressing critical gaps through intelligent integration of multiple security domains. The 78.4% firmware emulation success rate demonstrates substantial improvement while 99.28% malware classification accuracy validates deep learning effectiveness. Our open-source approach ensures accessibility enabling organizations with limited budgets to deploy professional-grade security capabilities. The modular architecture provides foundation for continued enhancement ensuring long-term relevance. We believe FIRMAI makes meaningful contribution to IoT security landscape providing practical tools while advancing state-of-the-art in automated security analysis.

Appendix A

SAMPLE CODE

A.1 Firmware Extractor Class

```python
import binwalk
import os

class FirmwareExtractor:
def __init__(self, firmware_path):
self.firmware_path = firmware_path
self.extract_dir = None

def extract(self):
"""Extract firmware filesystem"""
for module in binwalk.scan(self.firmware_path,
signature=True,
quiet=True,
extract=True):
for result in module.results:
if result.file.path in module.extractor.output:
self.extract_dir = module.extractor.output[
result.file.path].directory

return self.extract_dir
```

Bibliography

[1] Mohammad Al Abbas, Muna Al-Hawawreh, and Ahamed Aljuhani. Intelligent IoT malware detection using RNN-based NLP techniques. In *2023 International Conference on Cyber Security and Internet of Things*, 2023.

[2] Rajasekhar Chaganti, Vinayakumar Ravi, and Tuan D. Pham. Cross-architecture deep learning based android malware detection. *Journal of Information Security and Applications*, 2022.

[3] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[4] S. Jeyalakshmi and K. Nallaperumal. Hybrid CNN-GAN and DBN based IoT malware detection. *Journal of Network Security*, 2024.

[5] Imran Ahmed Khan, Nour Moustafa, Dechang Pi, Waqas Haider, Bin Li, and Alireza Jolfaei. An enhanced multi-stage deep learning framework for detecting malicious activities from autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 2023.

[6] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryoung Kim, Yeongjin Jang, and Yongdae Kim. FirmAE: Towards large-scale emulation of IoT firmware for dynamic analysis. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[7] Gordon F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009.

[8] Stratosphere Laboratory. IoT-23 dataset. https://www.stratosphereips.org/datasets-iot23, 2020.