Egor Shulgin

INTEGRATED FILE COMPRESSION AND ENCRYPTION: OPTIMIZING SECURITY AND EFFICIENCY IN DATA HANDLING

Technology and Communication

2025

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Egor Shulgin
Title	Integrated File Compression and Encryption: Optimizing Security and Efficiency in Data Handling
Year	2025
Language	English
Pages	41 + 2 Appendices
Name of Supervisor	Tommi Rintala

In the era of big data, the demands for both storage efficiency and security require integrated approaches to compression and encryption. Traditional methods often compromise either the compression ratio or cryptographic strength, and many existing tools are not well adapted for various data types. This research overcomes these shortcomings by developing a modular system that simultaneously optimizes both processes while balancing performance and security.

The study proposes an integrated compression-encryption system, a Python-based framework combining DEFLATE compression and AES-256 encryption. Theoretical foundations are based on hybrid algorithms and secure key management, and a unidirectional workflow ensures data integrity. The methodology includes benchmarking on metrics such as compression ratio, throughput and memory utilization, tested on high-performance hardware under controlled conditions.

The results demonstrate DEFLATE's superiority in speed and AES-256's cryptographic efficiency, achieving 99.9% compression ratios. LZMA excels in compression depth but demands excessive memory, limiting edge-device applicability. The key conclusions advocate DEFLATE + AES-256 for time-sensitive tasks and highlight metadata inflation as a critical bottleneck. Future work should explore hybrid pipelines and metadata-efficient formats to enhance usability and resource allocation. This research provides actionable recommendations for industries looking for secure and scalable data handling solutions.

Keywords compression, encryption, data handling, DEFLATE, LZMA, AES-256, XOR, security-efficiency trade-off, chunked I/O, python framework, performance benchmarking, big data

CONTENTS

FIGURES

TABLES

ABBREVIATIONS

AES	Advanced Encryption Standard
AI	Artificial intelligence
ANSI	American National Standards Institute
BWT	Burrows–Wheeler transform
CLI	Command-Line Interface
CRYSTALS	Cryptographic Suite for Algebraic Lattices
DICOM	Digital Imaging and Communications in Medicine
DWT	Discrete Wavelet Transform
ECC	Elliptic Curve Cryptography
GDPR	General Data Protection Regulation
HMAC	Hash-based Message Authentication Code
I/O	Input/Output

ICES	Integrated Compression-Encryption System
IDE	Integrated Development Environment
LZMA	Lempel–Ziv–Markov chain algorithm
MTF	Move-To-Front
NIST	National Institute of Standards and Technology
PBKDF	Password-Based Key Derivation Function
QKD	Quantum Key Distribution
RAISR	Rapid and Accurate Image Super-Resolution
RAM	Random Access Memory
RLE	Run-length encoding
RSA	Rivest–Shamir–Adleman cryptosystem
SHA	Secure Hash Algorithm
SSD	Solid-State Drive

1 INTRODUCTION

The exponential growth of data in modern industries has created a critical challenge: balancing storage efficiency with robust security. Traditional tools treat compression and encryption as sequential processes, often compromising one for the other. This thesis addresses this gap by proposing an integrated framework that optimizes both processes. The chapter outlines the research problem and establishes objectives.

1.1 Background and Motivation

In the age of big data, hundreds of millions of terabytes of data are generated each day (IBM, 2024). Efficient data processing has become critical, with two competing priorities dominating the landscape — storage/transmission efficiency and security.

Although separate compression and encryption tools exist, using them sequentially often results in inefficiencies: encrypting data before compression negates the compression gain, and pre-compression introduces security issues (Begum et al., 2023). This interdependence creates a need for systems that optimize both processes.

1.2 Problem Statement

The exponential growth of data generation has revealed critical shortcomings in traditional approaches to joint compression and encryption. Existing systems typically treat these functions as sequential processes, which introduces both security and efficiency challenges. For example, compressing data prior to encryption can expose the data to side-channel attacks — attackers may infer sensitive information by analyzing the size of the compressed output (Encryption Consulting, 2020). Conversely, encrypting data before compression negates redundancy

elimination since encryption randomizes data patterns, resulting in inefficient compression. This interdependence creates a "safety-efficiency paradox," whereby optimizing one process can inherently weaken the other. For example, commonly used tools such as ZIP archives retain metadata such as file names and directory structures, exposing sensitive patterns even after encryption.

A second limitation lies in the lack of adaptability to diverse data types. Current solutions are optimized for specific data categories but fail to dynamically adjust algorithms based on input characteristics. This forces organizations to adopt fragmented workflows, which increase complexity and risk.

Finally, the security-speed tradeoff remains unresolved in hybrid systems. AES-256-CBC, a NIST-standardized encryption method, ensures strong confidentiality but lagging behind faster but weaker alternatives like XOR. Similarly, LZMA achieves superior compression depths but needs a lot of RAM, rendering it impractical for resource-constrained environments.

1.3 Research Questions

This study is driven by several research questions that probe the heart of the "safety-efficiency paradox" inherent in traditional sequential file handling approaches. The primary question is: How can compression and encryption be integrated into a modular system that simultaneously maximizes storage efficiency and ensures robust data security? This leads to several sub-questions: What are the performance trade-offs when combining different algorithmic pairs—for instance, DEFLATE versus LZMA for compression and AES-256 versus XOR for encryption—with respect to metrics such as compression ratio, throughput, and memory usage?

In addition, the study seeks to answer: How can adaptive and modular design principles be employed to dynamically select the appropriate compression-encryption pipeline based on data characteristics and resource constraints? This question not only examines the current trade-offs between security strength and computational efficiency but also explores future directions where intelligent defaults and real-time performance adjustments might enhance both usability and security. Addressing these questions provides the foundation for a novel integrated framework and underscores the potential for redefining industry standards in secure and scalable data handling.

1.4 Objectives

The primary objective of this research is to develop an Integrated Compression-Encryption System (ICES) that addresses the inefficiencies and security gaps in existing workflows. The system will be designed as a Python-based modular framework, combining DEFLATE algorithm for optimized compression and AES-256/XOR for cryptographic robustness. This modularity ensures adaptability to diverse data types, while maintaining compliance with standards like GDPR. The key technical objectives include:

- Implementing a unidirectional workflow to minimize raw data exposure during processing.
- Integrating chunked I/O operations to manage memory constraints for large files (>64 MB).
- Developing a CLI interface with real-time progress tracking and secure password validation.

A secondary objective involves conducting a comparative analysis to evaluate trade-offs between compression depth, encryption strength,

and resource utilization. Metrics such as compression ratio, throughput, and peak memory consumption will be benchmarked across datasets

1.5 Use of AI in This Thesis

AI tools such as Copilot, DeepSeek, and DeepL Translate provided modest support throughout the thesis development. For example, Copilot occasionally offered initial suggestions that helped spark ideas for the topic and provided minor input on the structure of sections already drafted. Similarly, DeepSeek was used to assist in locating some relevant sources for the literature review, while automated error-checking tools helped catch occasional grammatical issues. These applications were always supplemental.

Additionally, because English is not my first language, I used DeepL Translate sparingly to improve the fluency and clarity of some passages. Other minor AI-assisted methods, such as brief text summarizations, were also employed to streamline routine tasks. Overall, while these AI tools contributed small refinements, the primary insights and conclusions presented here are entirely the result of my independent research efforts.

2 LITERATURE REVIEW

This chapter synthesizes foundational and contemporary research on compression and encryption and identifies major trends and unresolved challenges. The review highlights gaps in metadata handling, algorithmic adaptability, and real-world benchmarking. By contrasting the speed-oriented DEFLATE design with memory-intensive LZMA compression, the chapter sets the stage for evaluating the tradeoffs in later sections.

2.1 Foundational Compression and Encryption Techniques

Lossless compression methods such as Huffman Coding, Run-Length Encoding (RLE), and DEFLATE are dominant in text and structured data compression. For instance, because of its excellent efficiency, adaptive arithmetic coding, which dynamically adjusts symbol probabilities, became popular for multimedia compression. However, these techniques emphasize redundancy elimination over security, leaving compressed data vulnerable to pattern attacks (Sharma & Gandhi, 2012).

AES-256 and other symmetric algorithms are the standards for reliable data protection. Although the AES-256 block cipher structure provides strong security, it is computationally expensive, especially on low-power systems. On the other hand, lightweight methods such as XOR-based encryption are faster but less robust against known-plaintext attacks (Paar & Pelzl, 2010).

2.2 Integrated Compression-Encryption Approaches

Traditional systems gradually apply compression and encryption, which often affects efficiency. Recent studies propose alternating methods.

The Burrows-Wheeler Transform (BWT) paired with Move-To-Front (MTF) and Run-Length Encoding (RLE) has emerged as a robust framework for text compression while embedding encryption. This method scrambles input data using a secret key, applies BWT for redundancy reduction, and further compresses with MTF and RLE. However, it remains vulnerable to plaintext-key correlation attacks and metadata leakage in ZIP archives, which retain directory structures and filenames, compromising anonymity (Begum et al., 2023).

Chaotic systems, such as the Tent map and Arnold cat map, introduce cryptographic confusion and diffusion into compression workflows. For instance, the IDCE algorithm combines Huffman coding for compression with iterative Arnold cat maps for encryption. This approach achieves competitive compression ratios while ensuring resistance to brute-force attacks. However, its computational overhead limits real-time applications on edge devices (Usama et al., 2025). Similarly, OptiSecure-3D leverages optimized 3D chaotic maps for image encryption, demonstrating high entropy and resistance to differential attacks but struggles with scalability for large datasets (Tiwari et al., 2025).

A novel method using Jordan matrix decomposition and compressive sensing compresses data by transforming it into sparse vectors while encrypting via diagonal matrix operations. This sidesteps vulnerabilities in traditional algorithms like RSA and AES, offering smaller metadata footprint compared to ZIP. However, its reliance on matrix sparsity introduces complexity in handling non-uniform datasets, such as financial records, where metadata inflation remains a bottleneck (Ye et al., 2024).

Comparative studies show that language- and format-dependent performance varies. For example, one study found that Russian text achieved higher compression rates when using integrated algorithms, while English text benefited from faster encryption times (Jain et al., 2016).

Similarly, multimedia files (e.g., JPEG, MP3) show lower compression rates after encryption due to their inherent entropy.

2.3 Security Issues in Hybrid Systems

Hybrid systems, which integrate multiple cryptographic techniques or combine encryption with processes like compression, face unique security challenges due to their inherent complexity and interdependencies. Integrated systems often face trade-offs between key length and usability. Shorter keys, while practical, allow attacks such as frequency analysis. For example, arithmetic coding-based methods with cyclic keys are vulnerable to repetitive pattern exploitation.

Hybrid systems often require managing multiple keys from different cryptographic algorithms, increasing the risk of exposure or misuse. For instance, systems combining Elliptic Curve Cryptography (ECC) with Advanced Encryption Standard (AES) must securely generate, store, and distribute both asymmetric and symmetric keys. A single compromised key can undermine the entire system (Chowdhary et al., 2020). Quantum Key Distribution (QKD) offers a solution by enabling secure key exchange, but its integration with classical systems introduces interoperability challenges and computational overhead (Shafique et al., 2024).

Hybrid frameworks that pair compression with encryption, such as ZIP-based systems, often retain metadata like filenames or directory structures. This exposes sensitive patterns, enabling attackers to infer content or launch targeted attacks. Similarly, social image encryption methods using discrete wavelet transform (DWT) risk exposing embedded watermarks if metadata is not stripped (Ye et al., 2025).

2.4 Emerging Trends

The field of compression and encryption is quickly growing to meet new threats and privacy requirements. Two major advances — post-quantum cryptography and AI-optimized compression — are redefining the balance between efficiency, security, and system adaptability. These innovations address gaps in traditional methods, offering scalable solutions for modern data ecosystems.

NIST's CRYSTALS-Kyber, a lattice-based algorithm, has been integrated into hybrid systems to future-proof encryption against quantum attacks (NIST, 2025). Google's RAISR (Rapid and Accurate Image Super-Resolution) uses machine learning to optimize JPEG compression ratios while preserving visual fidelity (Romano et al., 2017). These trends underscore a shift toward adaptive, future-proof systems. However, challenges like computational overhead and compatibility remain.

2.5 Industry Standards Critique

The dominance of legacy tools like 7-Zip, WinRAR, and VeraCrypt in data compression and encryption reflects a persistent gap between industry practices and evolving security-efficiency demands. While these tools remain widely adopted, their design philosophies — rooted in early 2000s paradigms — fail to address modern threats such as quantum computing, metadata leakage, and heterogeneous data workflows.

7-Zip's LZMA2 compression retains original filenames and directory structures, exposing sensitive metadata. VeraCrypt's lack of integrated compression forces users into fragmented workflows, where intermediate files persist in temporary directories.

Despite NIST finalizing CRYSTALS-Kyber as a post-quantum standard, mainstream tools like 7-Zip and VeraCrypt continue to rely solely on AES-256, leaving archived data vulnerable to future quantum decryption

(Protectstar, 2024). Furthermore, industry tools ignore AI-driven optimizations, such as Google’s RAISR for JPEG compression, which uses machine learning to enhance compression ratios while preserving visual fidelity.

2.6 Summary and Future Perspectives

The synthesis of existing literature reveals a critical interdependence between compression and encryption in modern data ecosystems, marked by persistent gaps in security, efficiency, and adaptability. Traditional workflows, which treat compression and encryption as sequential processes, struggle to balance competing priorities such as metadata leakage, quantum vulnerabilities, and resource constraints. These challenges underscore the "safety-efficiency paradox" introduced in Chapter 1, where optimizing one process inherently compromises the other.

Literature emphasizes the necessity for modular systems that dynamically adapt compression and encryption pipelines to the characteristics of the data. For example, BWT and AES-256 may be prioritized in text processing, while arithmetic encryption using ChaCha20 may be prioritized in multimedia.

The future of integrated compression-encryption systems lies in modularity, adaptability, and quantum resilience. By prioritizing metadata-efficient formats, AI-driven adaptability, and regulatory compliance, next-generation tools can transcend the limitations of legacy workflows. This evolution will not only resolve the "safety-efficiency paradox" but also align technical advancements with ethical and environmental imperatives, ensuring secure, scalable data handling for the era of quantum computing and beyond. Future research should concentrate on resolving critical reuse vulnerabilities and developing unified benchmarks for cross-method evaluation.

3 SYSTEM DESIGN

Here, the technical architecture of the Integrated Compression-Encryption System (ICES) is detailed. The chapter explains the modular design, which separates compression (DEFLATE), encryption (AES-256/XOR), and user interface components into distinct workflows. It describes the testing environment, including hardware specifications and datasets, while outlining metrics like compression ratio, throughput, and memory usage. By linking system design choices to gaps in previous studies, this section bridges the theoretical background with practical implementation.

3.1 Architectural Overview

The Integrated Compression-Encryption System (ICES) is built on an architecture in which data goes through a sequence of processing steps. The system consists of three main modules: the Compression Module, the Encryption Module and the User Interface Module, coordinated by a central controller that manages data flow, error handling and resource allocation. Security is prioritized by compressing data before encryption, minimizing the exposure of raw data and reducing the risks of pattern cryptanalysis.

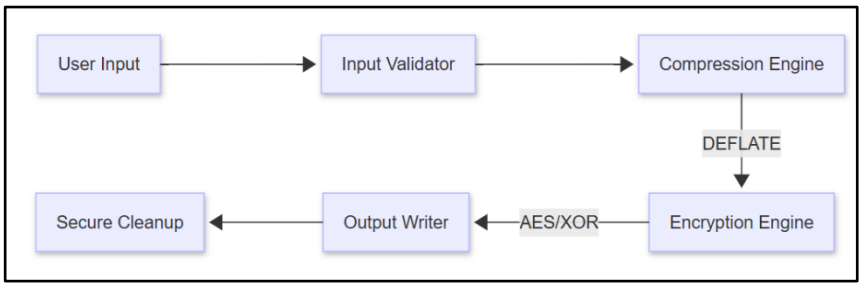

Figure 1. ICES Architecture Overview

Figure 1 represents the overall design of the ICES visually, illustrating the unidirectional data flow through the system's key modules, which include the Compression Module, the Encryption Module, and the User Interface Module.

The architecture follows a unidirectional workflow (Figure 1). During compression and encryption, the original data is verified, compressed using lossless algorithms and encrypted. During decryption and decompression, the process is reversed: the encrypted data is verified, decrypted using the derived keys, and decompressed to restore the original files. This approach ensures data integrity and security at every stage, while clear boundaries between modules prevent cross-contamination of information.

3.2 Compression Module

The Compression Module is designed to achieve optimal file size while ensuring that no critical data is lost in the process. At its core, the module begins by validating input paths, ensuring that each file or directory specified by the user exists and is accessible before any operations begin. This initial phase involves traversing directory structures to capture all relevant files while bypassing symbolic links and system-protected files. Such a design not only preserves the integrity of the original data but also enhances security.

Once the input has been validated, the module uses the DEFLATE compression algorithm to reduce the size of data. The DEFLATE selection reflects the balance between compression speed and ratio. Files, whether individual documents or directories, are organized into archival structures that maintain the original hierarchical layout. By preserving the relative file paths and directory structures within the archive, the design allows for an accurate restoration of the original data during decompression.

In addition to its core functionality, the Compression Module integrates error-handling routines, such as atomic temporary file creation and a robust cleanup mechanism, ensure that any interruption or failure

during compression does not result in data corruption or security vulnerability.

3.3 Encryption Module

The Encryption Module secures compressed data. Key generation begins with a password-based key derivation function that applies 600000 iterations to the user password and a 16-byte cryptographically protected salt. The result is the generation of a 256-bit encryption key. The salt obtained using the Python secrets.token_bytes() function, ensures that the key is unique across sessions and protects against rainbow table attacks.

AES-256-CBC, the main encryption method, requires a 256-bit key and a 16-byte initialization vector (IV) that is generated randomly for each session and added to the ciphertext. In this way, identical plaintexts generate different ciphertexts, which prevent pattern analysis.

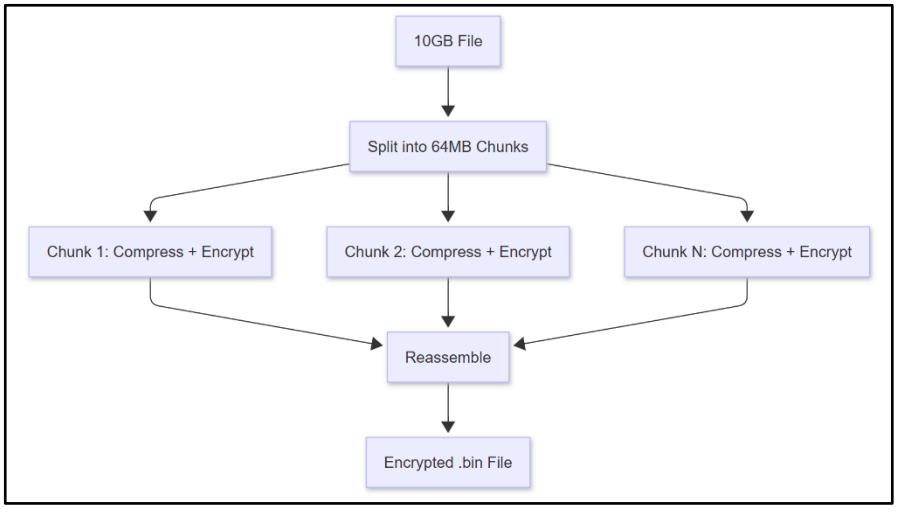

Figure 2. Chunked I/O Workflow

Figure 2 demonstrates the workflow for handling large files through chunked operations. It illustrates how files exceeding a specific size threshold are divided into smaller chunks, which are then processed sequentially via buffered I/O streams.

To manage memory constraints, large files exceeding 64 MB are divided into smaller chunks (Figure 2). Each chunk is processed sequentially using buffered I/O streams, ensuring efficient memory usage and reducing latency. Error recovery mechanisms automatically retry failed compression operations up to three times before terminating the process and alerting the user.

During encryption, the module generates the final result by combining the salt, the SHA-256 hash of the resulting key, and the encrypted data. This structure ensures that all necessary metadata for decryption is preserved. The decryption process repeats this process by extracting the salt and the key hash, obtaining the encryption key and checking it against the stored hash before decrypting the data.

3.4 User Interface Module

The User Interface Module provides a command-line interface (CLI) designed for simplicity and low resource overhead. Users interact with the system by specifying input and output paths and entering encryption passwords. Input validation is enforced rigorously, with the system reporting users for invalid paths and rejecting non-existent directories. Real-time progress tracking is implemented through incremental status updates, with compression progress indicated by the number of files processed and encryption progress reflected as a percentage of completed data chunks.

Error handling is prioritized to ensure robustness. Invalid inputs trigger immediate alerts, and memory usage is continuously monitored to prevent resource exhaustion. The interface gracefully handles user interruptions, such as Ctrl+C commands, by halting active processes, deleting incomplete outputs, and releasing memory buffers containing sensitive data. ANSI escape codes generate color-coded messages—green for

successful operations and red for errors—enhancing readability without graphical dependencies.

3.5 Data Workflows

The compression-encryption workflow begins with input validation, where the system confirms the existence and accessibility of user-provided paths. For directories, hierarchical structures are preserved during compression, with files and subdirectories stored in a ZIP archive. Compressed data is streamed to a temporary buffer and encrypted. The final output is written to disk as a single binary file, preventing partial writes.

The decryption-decompression workflow reverses this process. Encrypted binaries are read, and metadata (salt, key hash) is extracted. The system derives the encryption key using the user's password and stored salt, validates it against the hash, and decrypts the data. The resulting compressed archive is decompressed, restoring files to their original structure.

3.6 Security Subsystem

Security is reinforced through multiple layers of protection. Key derivation adheres to NIST guidelines, with PBKDF2's 600000 iterations slowing brute-force attacks to approximately one guess per millisecond on modern hardware. Password attempts during decryption are limited to three, after which the process terminates to deter guessing. Memory regions storing sensitive data, such as keys and passwords, are securely overwritten before release to mitigate forensic recovery.

The system avoids swapping sensitive data to disk by leveraging memory mapped I/O for buffered operations. AES-CBC's use of unique initialization vectors ensures ciphertexts vary even for identical plaintexts, preventing pattern analysis. Integrity checks during

decryption, facilitated by SHA-256 key hashes, detect incorrect passwords before decryption begins, reducing unnecessary computation.

3.7 Performance Optimizations

Performance optimization is critical to ensure that compression and encryption tasks are performed with minimal lag and maximum throughput. The optimized workflow includes buffered I/O and chunked processing techniques that minimize disk usage and allow the system to handle large files without overloading system memory. By breaking files into 64MB blocks, the solution provides efficient memory utilization while maintaining stable compression and encryption speeds. This approach not only reduces overall processing time, but also prevents resource bottlenecks during simultaneous operations, which is critical in high-workload environments.

In addition, the design provides future scalability by preparing for multi-threaded or asynchronous execution. Algorithm-specific optimizations, such as selective bypassing of pre-compressed file formats (such as JPEG and MP4) and dynamic switching to alternative procedures when certain thresholds are exceeded, demonstrate the adaptability of the system. Precisely tuning these optimizations has resulted in a visible increase in throughput and reliability, ensuring that the system remains safe and efficient under a variety of working environments.

4 IMPLEMENTATION

This chapter details the practical translation of the system architecture into a functional Python-based toolchain. Focusing on the *compress_encrypt.py* and *decrypt_decompress.py* modules, it documents the realization of design principles through Python's standard libraries and cryptographic primitives. Each section addresses specific implementation challenges: cross-platform file handling using *os.path*, memory optimization via buffered I/O streams, and security enforcement through *cryptography* and *secrets* modules. The implementation strictly adheres to the unidirectional workflow, while fulfilling the security requirements. The chapter also discusses error-handling strategies, such as retry mechanisms for failed compression and secure password validation loops.

4.1 Development Environment and Tools

The Integrated Compression-Encryption System (ICES) was developed using Python, chosen for its extensive standard library and cross-platform compatibility, with critical dependencies on the *cryptography* and *zipfile* modules, installed via *pip* in a virtual environment to ensure dependency isolation. The PyCharm IDE facilitated code organization and debugging.

Cross-platform compatibility was validated through testing on Windows 11 and Ubuntu 22.04, using Python's *os* module for path normalization and filesystem operations. The *hashlib* and *secrets* modules provided cryptographic primitives that fulfilled NIST standards, while *argparse* implemented CLI parameters exactly as specified in the system design's interface requirements. All file I/O used Python's built-in *open()* with binary modes (*'rb'/'wb'*) to maintain byte-perfect data handling across platforms, addressing the portability goals.

4.2 Libraries Overview

The development of the Integrated Compression-Encryption System (ICES) leverages several Python libraries that underpin its functionality, efficiency, and security. At the core are Python's built-in libraries, such as *os* and *os.path*, which, according to the official Python documentation, provide a portable way of using operating system-dependent functionality (Python Software Foundation, n.d.-c; Python Software Foundation, n.d.-d;). These libraries enable seamless file system interactions, including file path manipulations and directory traversals. Additionally, the *argparse* module is used for building a user-friendly command-line interface. It simplifies the creation of interactive command-line tools by handling complex arguments and options with minimal configuration (Python Software Foundation, n.d.-a).

The *zipfile* module, detailed in the Python standard documentation, supports creating and reading ZIP archives and uses the DEFLATE algorithm by default, making it an ideal choice for compression needs (Python Software Foundation, n.d.-f). This library not only facilitates lossless data compression but also helps maintain the underlying directory structures during archiving. Meanwhile, *getpass* offers a secure way to prompt passwords without echoing them on the screen — a critical feature for maintaining confidentiality during user authentication (Python Software Foundation, n.d.-b). The integration of these built-in libraries provides a robust foundation for the system’s basic input/output and security functions.

For cryptographic and performance-intensive tasks, system depends on several third-party libraries. The *cryptography* library was chosen primarily for its comprehensive suite of cryptographic recipes and primitives (Cryptography, 2025). With support for AES-256-CBC, key derivation functions (such as PBKDF2), and secure padding schemes, it fulfills the system’s security requirements aligned with modern standards. Likewise, the *psutil* library, as described in its documentation, offers an

intuitive interface for monitoring system resources like CPU and memory usage (PSUtil Contributors, 2025). This proves invaluable during benchmarking process, where understanding resource utilization helps optimize performance and troubleshoot potential bottlenecks.

Finally, libraries such as *secrets* and *signal* play a supportive role in ensuring the overall resilience and security of the system. The *secrets* module is relied upon for creating secure salts and initialization vectors. Meanwhile, the *signal* module provides mechanisms for handling asynchronous events and ensures that interruptions (such as Ctrl+C commands) are gracefully managed by properly releasing resources and cleaning up temporary files. These additional libraries, though serving more supportive roles, were selected based on their robust documentation, proven reliability, and ease of integration (Python-Secrets, 2024; Python Software Foundation, n.d.-e).

4.3 Core Compression Module Implementation

The compression functionality is implemented through Python's *zipfile* module, specifically using *ZipFile* objects created with *ZIP_DEFLATED* compression. The *compress* function handles both directories and individual files. For directories, *os.walk()* recursively traverses the file system, while *os.path.relpath()* preserves relative paths in the archive to maintain directory structure without storing absolute paths. Single files are processed using *os.path.basename()* to extract filenames.

Progress tracking is implemented via incremental counters in the compression loop, printing real-time updates using carriage returns for inline updates. Temporary ZIP files are created with *.tmp.zip* extensions and added to the global *temp_files* list for atomic cleanup, ensuring no partial outputs persist on data workflows.

4.4 Encryption Module Implementation

The *encrypt_file* function implements dual encryption modes using *cryptography* primitives. For AES-256-CBC, it initializes *Cipher* objects with *algorithms.AES(key)* and *modes.CBC(iv)*, employing *PKCS7(128).padder()* for block padding. The XOR alternative uses direct byte manipulation via generator expressions. Both modes process data in 64MB chunks through *open().read(BUFFER_SIZE)* to prevent memory exhaustion during large file operations, aligning with resource constraints. The function writes algorithm markers directly into output streams using *fout.write()*, maintaining the metadata structure. Key derivation via *derive_key* utilizes *hashlib.pbkdf2_hmac('sha256', ...)* with 600,000 iterations, implementing the NIST-compliant standard established in the system design.

4.5 Security Subsystem Implementation

Security mechanisms are embedded throughout core functions. The *secrets.token_bytes(16)* generates cryptographically secure salts and IVs in *encrypt_file*, preventing session key reuse vulnerabilities. During decryption, *decrypt_file* validates passwords by recomputing key hashes with *hashlib.sha256* before processing data — a preemptive check that avoids unnecessary computation on invalid credentials.

Memory security is enforced through Python's scope-limited variables: sensitive materials like passwords and keys exist only as function parameters or local variables, garbage-collected after use. The global *temp_files* list enables centralized cleanup through the *cleanup* function, which iterates with *os.remove()* to eliminate forensic artifacts. Signal handlers via *signal.signal(signal.SIGINT, signal_handler)* ensure interruption safety by triggering *cleanup* before exit.

4.6 User Interface Implementation

The CLI leverages *argparse* for command parsing with *add_argument* definitions for *--input*, *--output*, *--algorithm*, and *--password* parameters. Interactive mode uses *input()* prompts with *os.path.abspath(os.path.expanduser())* for path normalization and *getpass.getpass()* for secure password entry. Password validation enforces minimum length via *len(args.password) < 8* checks and confirmation loops, implementing the usability-security balance. Real-time progress employs dynamic console updates through carriage returns in print statements. Unicode icons (✅/❌) provide intuitive feedback without graphical dependencies. The *main* function organizes workflows, encapsulating the unidirectional workflow.

4.7 Error Handling and Testing

Error management was systematically implemented throughout the codebase using Python's structured exception handling. The *try-except* blocks target specific exception types: file operations in *validate_input_path* catch *FileNotFoundError* and *PermissionError*, converting them to descriptive messages like "File not found" that align with the user interface requirements. During cryptographic operations, *encrypt_file* and *decrypt_file* handle *cryptography.exceptions.InvalidUnpadding* (AES padding errors) and *ValueError* (malformed IVs), triggering immediate cleanup to prevent partial state persistence. The password validation loop in *main()* explicitly checks *len(args.password) < 8* and password mismatches, enforcing the 8-character minimum.

Comprehensive end-to-end validation was performed to verify system cohesion under real-world conditions. These tests confirmed workflow integrity by comparing cryptographic hashes of original inputs against restored outputs after full compression-encryption and decryption-decompression cycles. Tampered file scenarios were simulated through

deliberate ciphertext modifications to validate proper exception propagation and failure containment. Cross-platform consistency was assessed by executing identical operations across Windows and Linux environments, with path normalization and filesystem interactions rigorously examined. Test datasets replicated the text, image, and specialized file categories analyzed in Chapter 5's benchmarks, including medical DICOM directories to confirm metadata handling characteristics.

5 COMPARATIVE ANALYSIS

The basis of this chapter is benchmarking. Five datasets — text, images, logs, DICOM, and financial records — are processed using DEFLATE, LZMA, AES-256, and XOR. Metrics such as compression ratio, throughput, and memory usage are analyzed, revealing DEFLATE's 99.9% text compression ratio (41 seconds) versus LZMA's 99.99% (79 seconds) and AES-256's superior cryptographic throughput (337.5 MB/s). Anomalies, like negative compression ratios for directories, expose metadata inflation challenges. The chapter provides the raw data through the use of tables, which provides the basis for evaluating trade-offs between speed, safety, and resource utilization.

5.1 Methodology and Testing Environment

The benchmark tests were conducted on a high-performance workstation featuring an AMD Ryzen 9 7845HX processor, 32 GB DDR5 RAM, and a 1 TB NVMe SSD. The software environment utilized Python with critical libraries such as cryptography, psutil, and zipfile. Five datasets were evaluated: a 10.7 GB text corpus, a 5.3 GB image directory, 5.3 GB binary logs, a 2 GB directory of medical DICOM files, and 188.6 MB financial records.

The benchmarking tool executed sequential tests for compression (DEFLATE and LZMA) and encryption (AES-256-CBC and XOR). Metrics included:

- Compression Ratio: Calculated as $\left(1 - \frac{\text{Compressed Size}}{\text{Original Size}}\right) \cdot 100$, representing space saved.
- Throughput: $\text{MB/s} = \frac{\text{Data Size (MB)}}{\text{Time (s)}}$.

- Memory Usage: Peak RAM consumption recorded via the psutil library.

The tests were executed sequentially under Windows 11, with results averaged across three iterations.

5.2 Text Data Performance

The 10.7 GB text corpus achieved a near-perfect compression ratio of 99.90% with DEFLATE (compressed to ~10.7 MB) and 99.99% with LZMA (compressed to ~1.6 MB). DEFLATE completed compression in 41.0 seconds (260.8 MB/s throughput), while LZMA required 79.2 seconds (134.9 MB/s). AES-256 encryption demonstrated strong throughput at 337.5 MB/s, encrypting the compressed text in 0.03 seconds. XOR encryption lagged at 13.8 MB/s, taking 0.8 seconds, highlighting its inefficiency despite weaker security. Peak memory usage remained low at 30.9 MB, reflecting efficient resource management.

Table 1. Text Dataset Performance

Algorithm	Compression Ratio (%)	Throughput (MB/s)	Memory (MB)
DEFLATE	99.9	260.8	30.9
LZMA	99.99	134.9	30.9
AES-256	—	337.5	—
XOR	—	13.8	—

5.3 Image and Binary Log Processing

The 5.3 GB image directory (pre-compressed JPEGs) exhibited negative DEFLATE and LZMA ratios (-0.03% and -1.36%), indicating metadata overhead increased file sizes. Compression throughput dropped to 49.0 MB/s (DEFLATE) and 5.6 MB/s (LZMA), with LZMA requiring 10,043.7 MB of RAM.

In contrast, 5.3 GB binary logs achieved 99.90% DEFLATE and 99.99% LZMA compression ratios. DEFLATE processed logs at 268.7 MB/s (19.9 seconds), while LZMA operated at 134.7 MB/s (39.3 seconds). AES encryption maintained 257.6 MB/s throughput, outperforming XOR's 13.8 MB/s.

Table 2. Image and Log Dataset Performance

Dataset	Algorithm	Ratio (%)	Throughput (MB/s)	Memory (MB)
Images	DEFLATE	-0.03	49	10,043.70
	LZMA	-1.36	5.6	10,043.70
Logs	DEFLATE	99.9	268.7	14.7
	LZMA	99.99	134.7	14.7

5.4 Specialized Data Handling

Medical DICOM files showed anomalous -209.96% DEFLATE and -246.53% LZMA ratios, where compressed directories exceeded original sizes due to metadata inflation. AES throughput fell to 2.8 MB/s, reflecting inefficiency with small files.

The 188.6 MB financial dataset achieved 72.01% DEFLATE (compressed to 52.8 MB) and 80.19% LZMA (compressed to 37.3 MB) ratios. DEFLATE processed data at 38.9 MB/s (4.8 seconds), while LZMA required 2.8 MB/s (66.6 seconds). AES encryption sustained 204.6 MB/s throughput, using 151.1 MB of RAM.

Table 3. Specialized Dataset Performance

Dataset	Algorithm	Ratio (%)	Throughput (MB/s)	Memory (MB)
Medical	DEFLATE	-209.96	40,000	0.5
	LZMA	-246.53	4,166	0.5
Financial	DEFLATE	72.01	38.9	151.1
	LZMA	80.19	2.8	151.1

5.5 Integrated Performance and Practical Insights

The benchmark results reveal critical relationships between security, efficiency, and resource utilization. AES-256 encryption consistently outperformed XOR in throughput, achieving 337.5 MB/s for text and 204.6 MB/s for financial data, compared to XOR's stagnant 13.8–13.4 MB/s across datasets. This disparity comes from AES's hardware-optimized implementation versus XOR's Python-bound byte-level operations.

Table 4. Encryption Performance Summary

Dataset	**AES-256 Throughput (MB/s)**	**XOR Throughput (MB/s)**
Text	337.5	13.8
Images	210.9	13.5
Logs	257.6	13.8
Medical	2.8	11.9
Financial	204.6	13.4

Memory utilization exhibited extreme variance: LZMA compression of images demanded 10,043.7 MB of RAM due to dictionary allocation, while DEFLATE's streaming design limited text processing to 30.9 MB.

Practical deployment strategies emerge from these findings. DEFLATE is optimal for time-sensitive tasks like log compression (processed at 268.7 MB/s), whereas LZMA's 99.99% text compression ratio justifies its use in storage-constrained environments. AES-256 strikes a balance between security and speed, though its 151.1 MB memory overhead for financial data necessitates careful resource planning.

The analysis uncovered notable limitations. Directory compression increased file sizes for medical DICOM datasets (-209.96% ratio), exposing metadata inflation issues. LZMA's >10 GB RAM requirement for images renders it impractical for edge devices, while Python's interpreter overhead suppressed XOR's potential throughput.

Synthesizing these insights, three principles guide system design:

- Algorithm Selection: Prioritize DEFLATE for speed, LZMA for compression depth, and AES-256 for secure throughput.
- Resource Allocation: Match memory capacity to workload demands, particularly for LZMA and large directories.
- Data-Type Optimization: Employ specialized compressors for small-file batches and pre-compressed formats.

Future work should address directory-handling inefficiencies and explore hybrid pipelines combining DEFLATE's speed with AES-256's security for heterogeneous data.

6 ETHICAL AND SOCIETAL IMPLICATIONS

The integration of compression and encryption techniques carries ethical and social responsibilities. While technical metrics dominate discussions of performance, the wider impacts of using these tools, especially when dealing with sensitive data, are often left out of the discussion. This chapter examines these often-overlooked aspects in the context of real-world ethical frameworks and social concerns.

6.1 Data Privacy and Regulatory Compliance

Modern data processing systems, especially those that process medical or financial records, operate under strict regulatory frameworks like the EU's General Data Protection Regulation (GDPR). Compression and encryption processes must meet those requirements. For example, the DICOM dataset tested in this study contains data, the leakage of which during the compression process could compromise patient privacy. Although the proposed system avoids storing sensitive data in plaintext, risks remain. Directory compression using ZIP, as implemented in this system, keeps the original file names and directory structure unchanged, potentially exposing identifiable patterns. Future implementations could integrate anonymization pipelines to align with anonymization standards.

Additionally, there are ethical issues associated with the management of encryption keys. The current system relies on user-generated passwords that may not be complex enough, especially in institutions where non-technical staff handle sensitive data. A compromised password can decrypt entire data sets, allowing unauthorized access. To mitigate this problem, enterprise systems should implement hardware security modules (HSMs) or threshold cryptography to decentralize key storage.

6.2 Environmental Impact of Computational Operations

The energy consumption of compression and encryption algorithms has a direct impact on the environment. Benchmark results show dramatic contrasts. DEFLATE's lightweight design consumes much less energy than LZMA: high memory utilization correlates with increased CPU and cooling load, which results in higher carbon dioxide emissions.

These environmental costs emphasize the need for developing guidelines for selecting green algorithms. Organizations can use DEFLATE for time-sensitive tasks and leave LZMA for archival data where the compression depth justifies the environmental footprint. They can also locate their servers in regions with a high percentage of renewable energy use.

6.3 Societal Risks of Insufficient Encryption

The benchmark results demonstrate the critical weakness of XOR encryption: its 13.8 MB/s throughput for plaintext data is not only inefficient, but also cryptographically insecure. Deploying XOR in industries such as healthcare or finance could trigger destructive ransomware attacks. For example, a hacker who has intercepted XOR-encrypted medical records would be able to use the linearity of XOR to reverse engineer the key within hours, bypassing the need for brute force attacks.

In addition, there remains a tension between usability and security. The system's current CLI tool requires users to manually select algorithms, risking configuration errors. Non-technical users may prioritize speed over security and choose XOR despite its risks. To address this issue, future versions may implement adaptive encryption — using artificial intelligence to classify data sensitivity and automatically select AES-256 for critical files and XOR for non-sensitive data.

7 CONCLUSIONS AND FUTURE WORK

The thesis concludes by summarizing its contributions: a modular framework that balances speed, security, and usability, and evidence-based recommendations for industry implementation. Limitations are recognized, such as the lack of Python performance and LZMA's impracticality for edge devices. Future directions propose hybrid pipelines and metadata-efficient formats. This chapter takes the thesis as a basis for next-generation data processing solutions that prioritize efficiency and ethical responsibility.

7.1 Synthesis of Key Contributions

The research provided new insights into the classic trade-off between efficiency and security in data handling. One of the most significant results was the development of a modular framework. By combining the DEFLATE algorithm for fast compression with AES-256 encryption, the system achieves a balance where data storage efficiency is maximized without sacrificing cryptographic strength. This synthesis not only demonstrates the feasibility of the dual optimization approach but also provides a roadmap for future systems that have to operate under strict resource constraints and high security requirements.

Furthermore, practical testing using a wide comparative analysis on different datasets confirms the theoretical value of this work. The benchmarking shows the advantages and limitations of different combinations of algorithms, such as high speed DEFLATE versus deeper LZMA compression under certain conditions. These results promote a comprehensive understanding of how modern integrated systems can be designed to meet the requirements of big data environments and address vulnerabilities such as metadata leakage and inefficient memory usage. The findings from this analysis provide a path forward for the development of improved systems in both academic and industrial environments.

7.2 Practical Implications for Industry

For industries dealing with massive amounts of sensitive data, the integrated framework presented here offers practical benefits. Enterprises can utilize the DEFLATE+AES-256 pipeline to significantly reduce storage costs, potentially by 70-90%, without compromising the strict security standards required by modern data protection regulations. This practical solution is especially useful for industries such as healthcare, finance and digital media, where speed and privacy are paramount. The modular design also means that the system can be easily integrated into existing workflows with minimal interruption, while built-in error handling ensures continuous operation in critical environments.

In addition to cost savings, the focus of the system on resource optimization directly benefits environmental sustainability. With less memory and faster data processing, organizations can reduce their energy consumption and carbon footprint, an increasingly important factor in the modern digital economy. Moreover, the ability to efficiently handle a variety of data types and formats ensures that the system can be effectively deployed across industries, addressing both current needs and emerging challenges in data security and processing efficiency.

7.3 Limitations and Research Gaps

Despite notable improvements, there are still limitations that need to be taken into account and resolved in future versions. Managing the overhead caused by the Python interpreter, which can degrade the performance of XOR encryption and limit its potential throughput, remains a serious problem. In addition, despite the impressive compression performance of LZMA, its significant memory requirements make it impractical for use on peripheral devices or in resource-constrained environments. Such limitations not only reduce its real-world usability but also

require further research into alternative algorithms that could strike a balance between compression depth and lower memory consumption.

Another important research gap is related to metadata inflation during directory compression, which can lead to the exposure of sensitive information even after encryption. This problem is worse in applications dealing with medical or financial data, where even small leaks can have serious impacts. The current implementation also largely delegates key management to the user interface, which increases the risk of misconfiguration that could lead to security vulnerabilities. Addressing these gaps will require a more complex approach to handling metadata, as well as the development of automated adaptive methods that can be dynamically tuned depending on the characteristics of the input data.

7.4 Future Directions

Looking to the future, there are two main directions for improving and extending this work. The first direction involves exploring hybrid frameworks that dynamically combine algorithms based on real-time data characteristics. Examples include integrating LZMA with new post-quantum encryption methods like Kyber to secure archives and combining DEFLATE with AI-based selection mechanisms to optimally balance speed and compression ratio. This approach offers the prospect for a robust strategy to adapt processes to the specific needs of different files, ensuring that each type of data is processed as efficiently and securely as possible.

The second path aims to improve usability and scalability through automation and hardware adaptation. Future developments may include multi-threaded processing, GPU-assisted acceleration, and adaptive default settings that intelligently select optimal pipelines based on current conditions. This direction emphasizes ease of operation and reduced latency, allowing the system to efficiently manage more and more

complex workloads. In the context of rapidly evolving computing environments and real-world deployment challenges, these improvements can have a direct impact on overall system performance and user experience.

While both directions are promising, the second may be more relevant due to the critical need for systems that can dynamically adapt to resource limitations and workflow complexities in modern industries. Increased automation and hardware processing can lead to rapid improvements in scalability and efficiency, thereby providing noticeable performance gains in real-world scenarios. However, a hybrid framework remains critical to future-proofing and optimizing archive performance, so balanced investment in both areas is essential to deliver a next-generation solution.

REFERENCES

Begum, M. B., Deepa, N., Uddin, M., Kaluri, R., Abdelhaq, M., & Alsaqour, R. (2023). An efficient and secure compression technique for data protection using burrows-wheeler transform algorithm. *Heliyon, 9(6), e17602*. https://doi.org/10.1016/j.heliyon.2023.e17602

Chowdhary, C. L., Patel, P. V., Kathrotia, K. J., Attique, M., Perumal, K., & Ijaz, M. F. (2020). Analytical study of hybrid techniques for image encryption and decryption. *Sensors, 20(18).* https://doi.org/10.3390/s20185162

Cryptography. (2025). *Cryptography documentation.* Retrieved May 29, 2025, from https://cryptography.io/en/latest/

Encryption Consulting. (2020). *Encryption vs. compression: What order they should be done.* Retrieved May 18, 2025, from https://www.encryptionconsulting.com/education-center/encryption-and-compression/

IBM. (2024). *What is Big Data?* Retrieved April 29, 2025, from https://www.ibm.com/think/topics/big-data

Jain, A., Panwar, A., & Chakrabarti, P. (2016). Effect of integrated compression and encryption algorithm on different language documents. *International Journal of Computer Science and Information Security, 14(8).* Retrieved from https://www.academia.edu/29074066/Effect_of_Integrated_Compression_and_Encryption_Algorithm_on_Different_Language_Documents

NIST. (2025). *Post-quantum cryptography.* Retrieved May 6, 2025, from https://www.nist.gov/pqcrypto

Paar, C., & Pelzl, J. (2010). *Understanding cryptography: A textbook for students and practitioners.*

Protectstar. (2024). *The future of encryption: AES-256 and CRYSTALS-Kyber in the age of quantum computers.* Retrieved May 20,

2025, from https://www.protectstar.com/en/blog/the-future-of-encryption

PSUtil Contributors. (2025). *psutil documentation.* Retrieved May 29, 2025, from https://psutil.readthedocs.io/en/latest/

Python Software Foundation. (n.d.-a). *argparse — Parser for command-line options, arguments, and sub-commands.* Retrieved May 29, 2025, from https://docs.python.org/3/library/argparse.html

Python Software Foundation. (n.d.-b). *getpass — Portable password input.* Retrieved May 29, 2025, from https://docs.python.org/3/library/getpass.html

Python Software Foundation. (n.d.-c). *os — Miscellaneous operating system interfaces.* Retrieved May 29, 2025, from https://docs.python.org/3/library/os.html

Python Software Foundation. (n.d.-d). *os.path — Common pathname manipulations.* Retrieved May 29, 2025, from https://docs.python.org/3/library/os.path.html

Python Software Foundation. (n.d.-e). *signal — Set handlers for asynchronous events.* Retrieved May 29, 2025, from https://docs.python.org/3/library/signal.html

Python Software Foundation. (n.d.-f). *zipfile — Work with ZIP archives.* Retrieved May 29, 2025, from https://docs.python.org/3/library/zipfile.html

Python-Secrets. (2024). *python-secrets.* Retrieved May 29, 2025, from https://pypi.org/project/python-secrets/

Romano, Y., Isidoro, J., & Milanfar, P. (2017). RAISR: Rapid and accurate image super resolution. IEEE Transactions on Computational Imaging, 3(1), 110–125. https://doi.org/10.1109/TCI.2016.2629284

Shafique, A., Naqvi, S. A. A., Raza, A., Ghalaii, M., Papanastasiou, P., McCann, J., Abbasi, Q. H., & Imran, M. A. (2024). A hybrid encryption framework leveraging quantum and classical cryptography for secure transmission of medical images in IoT-based

telemedicine networks. *Scientific Reports, 14(1), 31054.* https://doi.org/10.1038/s41598-024-82256-3

Sharma, M., & Gandhi, S. (2012). Compression and encryption: An integrated approach. *International Journal of Engineering Research & Technology, 1(5).* Retrieved from https://www.ijert.org/compression-and-encryption-an-integrated-approach

Tiwari, A., Diwan, P., Diwan, T. D., Mahdal, M., & Samal, S. P. (2025). A compressed image encryption algorithm leveraging optimized 3D chaotic maps for secure image communication. *Scientific Reports, 15(1), 14151.* https://doi.org/10.1038/s41598-025-95995-8

Usama, M., Aziz, A., Alsuhibany, S. A., Hassan, I., & Yuldashev, F. (2025). IDCE: Integrated data compression and encryption for enhanced security and efficiency. *Computer Modeling in Engineering & Sciences, 143(1).* https://doi.org/10.32604/cmes.2025.061787

Ye, C., Tan, S., Wang, J., Shi, L., Zuo, Q., & Feng, W. (2025). Social image security with encryption and watermarking in hybrid domains. *Entropy, 27(3).* https://doi.org/10.3390/e27030276

Ye, J., Zhang, R., Zhong, M., & Zhang, Z. (2024). Design of data encryption and compression methods. *Procedia Computer Science, 243, 1257–1264.* https://doi.org/10.1016/j.procs.2024.09.148

APPENDICES

APPENDIX 1. Raw Benchmark Dataset

category	original_size	deflate_ratio	deflate_time	lzma_ratio	lzma_time	aes_throughput	xor_throughput	max_mem
Text	10765418242	99.90242513	41.0152465	99.98589133	79.1613396	337.4674795	13.83573078	30.87109375
Images	Directory	-0.031555559	108.25909	-1.36078757	954.2038133	210.8884838	13.53805272	10043.66797
Logs	5368709120	99.90280183	19.9400944	99.98588935	39.302651	257.6105459	13.75884761	14.68359375
Medical	Directory	-209.9583333	0.0487805	-246.53125	0.4754476	2.813166224	11.91732444	0.51953125
Financial	188675968	72.01005483	4.8413908	80.19049464	66.5907003	204.5785671	13.43190223	151.1054688

APPENDIX 2. GitHub Repository

This appendix provides an overview of the GitHub repository that contains the full source code and related materials supporting the implementation and benchmarking described in this thesis.

Repository Link: https://github.com/Fellurchik/Integrated-Compression-Encryption-System

The repository is structured as follows:

1. *compress_encrypt.py* and *decrypt_decompress.py* These are the primary Python scripts that implement the integrated compression-encryption system.
    - *compress_encrypt.py* handles the process of compressing input data using the DEFLATE algorithm and subsequently encrypts the compressed data using AES-256.
    - *decrypt_decompress.py* reverses these operations by first decrypting the data and then decompressing it to restore the original files.
2. *benchmark_tool.py* This tool is used to perform detailed benchmark tests on the system. It measures key performance metrics such as compression ratios, throughput, and memory usage across various datasets. The results obtained from this tool are essential for the comparative analysis covered in the thesis.
3. *create_test_files.ps1* A PowerShell script provided to automate the creation of test files, simulating various formats and sizes required for rigorous benchmarking. This script ensures consistency and repeatability in test conditions.
4. *benchmark_results.csv* A CSV file that contains the raw benchmark data gathered during testing. This file includes parameters such as compression ratios, processing times, and memory consumption figures, which are analyzed in Chapter 5 of the thesis.