

Automatic Detection, unpacking of untagged compressed data

Automatisk detektion, uppäckning av otaggad komprimerade data

Arvid Attin
Martin Christensson

Supervisor : Suleman Khan, Jesper Holmqvist
Examiner : Andrei Gurtov

Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>.

Abstract

Modern digital systems rely heavily on firmware updates that are frequently distributed as compressed binary blobs. In forensic investigations and security audits, these blobs often appear without file headers or metadata, rendering standard signature-based extraction tools ineffective. This thesis presents **BinSift**, a modular Python-based framework designed for the automatic detection, classification, and “blind” decompression of untagged compressed data.

To calibrate the system, a large-scale statistical analysis was conducted on the **FirmSec dataset**, profiling approximately 34,136 firmware images totaling over 200 GB of binary data. Results indicate that an average Shannon entropy threshold of **7.1 bits per byte** provides an optimal balance for capturing modern compression formats like LZMA and SquashFS while minimizing false positives from high-density uncompressed code.

The BinSift framework was evaluated against industrial firmware samples, achieving a **59.0% success rate** in “True Blind” mode without any prior knowledge of file headers. This approach maintained an **81.5% fidelity retention** compared to metadata-assisted baselines. When excluding mathematically unrecoverable encrypted payloads, the effective success rate rose to **84.4%**. These findings demonstrate that entropy-based stream identification and bit-level refinement are viable solutions for bypassing obfuscation in embedded systems forensics.

Acknowledgments

We would like to express our sincere gratitude to Jesper Holmqvist, our external supervisor, for his kindness, humor, and unwavering support. We are especially grateful for the creative autonomy that he and NFC granted us; their faith in our ability to structure the research independently made this project a truly engaging and rewarding learning experience. We also thank NFC for providing a supportive workplace and an intriguing project from which we have gained invaluable insights.

Additionally, we wish to thank our university supervisor, Suleman Khan, for his kindness and for providing essential moral support throughout this journey.

Contents

Abstract	iii
Acknowledgments	iv
Contents	v
List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Motivation	1
1.2 Aim	1
1.3 Research Questions	2
1.4 Approach	2
1.5 Contributions	3
1.6 Structure of the thesis	3
2 Background and Related Work	4
2.1 Background	4
2.2 Existing Tools	6
2.3 Related Work	6
3 Theory	8
3.1 Firmware	8
3.2 Binary Blobs	8
3.3 File Headers and Metadata	9
3.4 Anatomy of router firmware	10
3.5 Data Compression	14
3.6 Decompression	15
3.7 Bit-by-Bit Processing for Data Recovery	15
3.8 Byte-by-Byte Processing and its Limitations	16
3.9 The DEFLATE Compression Pipeline	17
3.10 Gzip Format Structure	19
3.11 LZ4 Header Variants and Integrity	21
3.12 LZMA2 and the XZ Container Format	22
4 Method	24
4.1 Dataset Generation	24
4.2 Workflow	26
4.3 Implementation of the BinSift Framework	28
4.4 Methodology: Large-Scale Entropy Profiling and Dataset Analysis	31
4.5 Extraction and Analysis Pipeline	34
4.6 Automated Firmware Evaluation Pipeline	38
4.7 System Implementation and Experimental Logic	39
4.8 Experimental Setup and Failure Analysis	39

5 Result	41
5.1 Entropy Distribution Analysis	41
5.2 Results and Performance Evaluation	48
5.3 Extraction Outcomes and Static Analysis Findings	51
6 Conclusion	56
Bibliography	58
A Mass Firmware Profiler Script	62

List of Figures

3.1	Main overview of the Binwalk analysis of the ArcherC60v3 firmware binary.	12
3.2	Bit-by-bit shifting algorithm.	16
3.3	The DEFLATE compression pipeline, illustrating the two-stage reduction of data via LZ77 and Huffman coding.	17
3.4	The LZ77 sliding window algorithm compressing the string "ABRACADABRAX". The compressor finds the repeated "ABRA" pattern 7 bytes back in the history buffer.	18
3.5	Huffman Tree and corresponding variable-length codes [rfc1951].	18
3.6	The raw DEFLATE block structure showing the unaligned 3-bit header preceding the compressed payload.	19
3.7	The Gzip member format illustrating the fixed 10-byte header.	19
3.8	The Gzip member format illustrating the fixed 8-byte trailer.	20
3.9	The Gzip member format illustrating the relationship between the fixed 10-byte header, the conditional optional fields (governed by the FLG byte), and the trailing integrity checks.	21
3.10	The Lz4 format illustrating the fixed 7-byte standard head and legacy frame.	21
3.11	The Lz4 format illustrating the fixed 8-byte trailer	22
3.12	The LZMA2 format illustrating the fixed 12-byte head.	22
3.13	The LZMA2 format illustrating the fixed 12-byte footer.	23
4.1	Ground Truth Configuration vs. Sliding Window (-K 512) Entropy Detection	26
4.2	Compression Efficiency (Reduction %) by Block and Source Firmware	26
4.3	Workflow part one.	27
4.4	Workflow part 2A.	27
4.5	Workflow part 2B, which later proved to be our main extraction method	28
4.6	Project structure of the BinSift framework	30
4.7	Visual representation of the automated "Extract-Process-Delete" data profiling pipeline used to analyze the FirmSec dataset.	33
4.8	Ghidra Base Address Selection.	36
4.9	Ghidra Language and Compiler Specification dialog.	37
5.1	View of entropy distributions.	42
5.2	Distribution and absolute count of identified compression types.	43
5.3	Kernel Density Estimation (KDE) of Minimum Entropy	44
5.4	Kernel Density Estimation (KDE) of Average Entropy	45
5.5	Cumulative Distribution Function (CDF) of Minimum Entropy	47
5.6	Cumulative Distribution Function (CDF) of Average Entropy	48
5.7	Comparison of Extraction Success Rates between Full Mode and Blind Mode across device categories.	49
5.8	Post analysis of failure modes ($n = 29$), categorizing samples into encryption "False Failures" and Complex/Obfuscated.	50
5.9	Distribution of processing time by category, showing the baseline Full Mode duration and the overhead introduced by Blind Mode analysis.	51
5.10	Output file structure after processing frankenstein.bin.	52
5.11	Ghidra UI showing the synchronized Assembly Listing (center) and the Decompiled C code (right) at the Reset Vector.	53

List of Tables

4.1	ARM Cortex-M Hardware Memory Map	36
5.2	Consolidated Performance Metrics: Full Mode vs. Blind Mode	51
5.3	Forensic Volume and Processing Efficiency Summary	51
5.4	Comparison of Ground Truth (Figure 4.1) vs. BinSift Detection	52



1 Introduction

Modern digital systems, including next-generation aviation networks [28], rely heavily on firmware updates to maintain security, add functionality, and correct defects. These updates are often distributed as compressed data to reduce size and hide internal structure. In some real-world situations, such as forensic investigations, incident response, and security audits, these compressed blobs can appear without file headers, format identifiers, or documentation. This lack of metadata makes it unclear which compression method was used and prevents further inspection of the underlying content. The methods used to protect and obfuscate code in embedded and firmware systems have become increasingly sophisticated [46, 19]. Standard tools typically rely on known signatures or predefined formats and therefore fail when dealing with proprietary, customized, or intentionally obfuscated compression schemes. For analysts, this becomes a significant obstacle: unless the data can be correctly unpacked, it is impossible to examine firmware behavior, verify integrity, or detect malicious modifications. This difficulty highlights the critical need for robust, generic unpacking solutions [26]. This study focuses on how untagged compressed data, especially from firmware sources, can be automatically detected and unpacked.

1.1 Motivation

When compressed data lacks labels or identifiable markers, the entire inspection pipeline breaks down. This problem is especially common in embedded devices, where manufacturers use a mix of standard and custom compression methods. For forensic analysts and security professionals, manually guessing and testing compression algorithms is inefficient and error-prone. Untangling these blobs is not only a practical necessity but also a prerequisite for deeper analysis: reverse engineering firmware, recovering evidence, evaluating security posture, and detecting tampering all depend on the ability to correctly decompress unknown data. Developing a systematic, automated solution therefore addresses a recurring and impactful challenge in cybersecurity and digital forensics.

1.2 Aim

The aim of this project is to develop and evaluate a method that can automatically detect whether an untagged binary blob is compressed, identify the most likely compression algorithm used, and decompress it without relying on file metadata. Particular emphasis is placed on firmware-related data, where compression patterns may be non-standard or deliberately obfuscated.

1.3 Research Questions

- **RQ1:** To what extent can statistical signatures, specifically average Shannon entropy and byte-frequency patterns, be used to reliably detect compressed streams and classify algorithms such as LZMA or Deflate in the absence of "Magic Numbers," and how do these metrics perform when distinguishing between compression, encryption, and randomized machine code?
- **RQ2:** How can a modular framework be designed to facilitate "blind" decompression of untagged binary blobs, and what mechanisms are required to address the technical challenges of bit-level boundary precision and stream-based alignment without metadata assistance?
- **RQ3:** What is the forensic effectiveness of an entropy-based extraction approach, measured by success rate and "Fidelity Retention" against metadata-assisted baselines, when evaluated against large-scale real-world firmware datasets containing both recoverable compressed payloads and unrecoverable encrypted segments?

1.4 Approach

This thesis addresses the research questions outlined in Section 1.3 by exploring statistical analysis techniques capable of identifying data characteristics in the absence of metadata. The methodology begins by employing Shannon entropy as a primary metric to establish a benchmark for distinguishing between structured data, compressed streams, and encrypted blobs. To ensure a verifiable and reproducible study, the baseline datasets and experimental testing will be conducted using open-source firmware. This allows for the comparison of known compression implementations against the tool's automated detection results.

Following this, a classification model is developed to map these statistical signatures to specific compression algorithms. The core of the methodology involves the creation of a "blind decompression" engine that iteratively tests identified candidates against the untagged blob. Ultimately, the research seeks to determine how effectively an automated pipeline can bypass the need for file headers or documentation in a forensic or security audit context.

1.4.1 Scope and Limitations

This thesis is delimited to the development and evaluation of a modular framework for the automated detection and "blind" decompression of compressed binary blobs. The research focuses on identifying bitstreams using statistical features, such as Shannon entropy and byte-frequency patterns, specifically targeting common firmware compression formats like LZMA and SquashFS. While the primary scope of the **BinSift** framework is confined to the extraction phase, this thesis includes a manual case study to demonstrate a "proof of concept" for forensic analysis. This demonstration illustrates the transition from raw extracted machine code to human-readable C code using external tools, proving the practical utility of the framework's output for subsequent reverse engineering.

However, several inherent limitations constrain this study. First, the framework itself does not automate the semantic analysis of unpacked content, the reverse engineering of machine code, or the interpretation of file system structures. Second, while the research addresses "obfuscated" compression, it does not attempt to bypass strong encryption where the key is unknown. Because encrypted payloads and high-entropy compressed data often exhibit nearly identical statistical distributions, encryption represents a mathematical "hard ceiling" for extraction techniques based purely on statistical carving. Finally, the system's ability to classify device architecture is limited by the contents of a predefined database, and the study does not account for automated semantic verification to mitigate false positives, as the evaluation focuses on the technical success of bitstream decompression.

1.5 Contributions

The findings presented in this thesis provide a systematic framework for addressing the “black box” nature of modern firmware updates and binary blobs. By automating the identification of compression schemes without relying on metadata, this work reduces the manual effort required by forensic analysts and security researchers when encountering proprietary or stripped binaries. The main contributions of this thesis are to provide:

1. **Analytical Research:** A comparative study of statistical features—such as Shannon entropy, Chi-square distribution, and byte-frequency patterns—that most effectively differentiate between various compression algorithms in a metadata-less environment.
2. **Experimental Framework:** The development of a Python-based diagnostic tool designed to automatically detect, classify, and attempt decompression of untagged binary blobs, specifically optimized for firmware-related data.
3. **Empirical Evaluation:** A performance benchmark of the proposed method against real-world firmware samples, evaluating the trade-offs between statistical classification accuracy and computational efficiency compared to traditional brute-force unpacking methods.

1.6 Structure of the thesis

Chapter 1: Introduction – Defines the problem of untagged compressed data in firmware forensics. It outlines the motivation, the project’s aim to develop an automatic decompression method, and the research questions regarding measurable data features and system performance.

Chapter 2: Background and Related Work – Establishes the foundations of digital forensics and Shannon entropy. It reviews existing tools like Binwalk, Unblob, and Ghidra, and discusses advanced research in feature engineering and bit-by-bit processing.

Chapter 3: Theory – Provides a deep dive into the nature of binary blobs and the anatomy of router firmware. It details the operational mechanics of the DEFLATE, Gzip, LZ4, and LZMA2 compression pipelines, as well as the mathematical logic behind bit-level data recovery.

Chapter 4: Method – This chapter is divided into several distinct areas of development:

- The development of the BinSift framework and its multi-stage extraction pipeline.
- Details regarding the initial dataset generated from open-source firmware.
- The application of FirmSec for large-scale entropy analysis and the implemented “extract-process-delete” method.
- A demonstration of converting compressed blobs to readable C code and the engine’s use for architecture identification.

Chapter 5: Result – Following a similar structure to the previous chapter, this section is organized as follows:

- Statistical findings derived from the mass entropy profiling.
- An evaluation of the framework’s performance, including ground-truth comparisons and an analysis of success rates and failure modes across 105 industrial firmware samples.
- Results from the demonstration of converting compressed blobs into readable C code.



2

Background and Related Work

2.1 Background

In the domain of digital forensics and data recovery, the ability to identify and extract information from raw data streams is a necessity. The process of reassembling files from disk fragments based purely on content, without the aid of file system metadata, is known as *file carving* [49]. This technique is essential when dealing with “untagged” data, artifacts that lack standard file names, extensions, or directory entries due to deletion, corruption, or deliberate obfuscation [49, 34]. The automatic detection and unpacking of such data is critical across a wide spectrum of computing layers, from physical storage media to high level software executables. Applications include:

- **Digital Forensics and Data Recovery:** Investigators frequently analyze hard drives, solid-state drives, and mobile phone storage to recover deleted evidence or salvage data from damaged file systems [49]. In these scenarios, files are often fragmented or lack the headers necessary for standard recovery tools to function.
- **Malware Analysis and Binary Recovery:** Malicious software often employs packing (compression) and encryption to conceal code and evade static analysis [22]. Detecting these packed binaries is crucial for analyzing threats that may delete themselves or hide within a disk image [34].
- **Network Security:** Real-time analysis of network traffic requires the ability to distinguish between encrypted flows and compressed data streams to ensure protocol compliance and detect anomalies [16].

2.1.1 The Foundations of Shannon Entropy

Shannon entropy (H) was originally introduced to measure the average missing information in a random source [29]. Mathematically, for a random variable X with a probability distribution p , it is defined as:

$$H(X) = - \sum_{x \in X} p(x) \log_2 p(x) \geq 0$$

Key Characteristics:

- **Uncertainty and Unevenness:** Entropy quantifies the "unevenness" of a distribution. A constant variable (determined outcome) has an entropy of 0, while a uniform distribution (where all outcomes are equally likely) reaches the maximum entropy of $\log_2(|\mathcal{X}|)$ [29].
- **The "Surprise" Factor:** In information theory, the term $-\log_2 p(x)$ is often called the "surprise" of observing a specific outcome. Shannon entropy is the statistical average of that surprise [29].
- **Physical Analogy:** While suggested by von Neumann for its "fuzziness," the term was chosen because the mathematical form is nearly identical to Boltzmann entropy in statistical mechanics, which describes the microscopic configurations of physical systems [29].

2.1.2 Detection and Entropy Analysis

In the context of binary analysis, the primary method for detecting hidden, compressed, or encrypted data within untagged binaries is *Entropy Analysis*. Because encryption and compression algorithms aim to eliminate redundancy, they produce byte distributions that appear highly uncertain or random. Information density, or entropy, measures this uncertainty in a series of bytes; higher entropy scores strongly correlate with the presence of encryption or compression [34].

Common Metrics and Tools:

The most common metric for this is Shannon entropy, calculated via the Maximum Likelihood Estimator (MLE), which sums the frequency of byte values (0x00–0xFF) to generate a score typically bounded between 0 and 8 [16, 34].

The Differentiation Challenge:

While entropy is effective at distinguishing structured data (like native code) from unstructured data, a significant hurdle remains. Both modern compression algorithms (e.g., DEFLATE) and encryption algorithms (e.g., AES) produce high-entropy byte distributions that closely resemble uniform random noise. Consequently, simple entropy estimates often fail to distinguish between compressed and encrypted data [16]. To address this, researchers have employed various statistical and machine learning enhancements:

- **Statistical Tests:** Beyond simple entropy, the χ^2 (Chi-square) test and the NIST SP800-22 test suite are used to evaluate randomness. For instance, the *HEDGE* method combines subsets of NIST tests with χ^2 analysis to improve detection reliability [16].
- **Machine Learning (ENCOD):** Recent deep learning advancements include *ENCOD*, a deep neural network (DNN) approach. Unlike standard statistical tests, ENCOD can differentiate between encrypted data and specific compressed formats (such as ZIP, GZIP, or PNG) with high accuracy, even on small data fragments (e.g., 512 bytes) [16].
- **Bintropy:** This tool calculates the entropy of fixed-length data blocks to identify packed or encrypted malware, utilizing specific confidence intervals to filter out and reduce false positives [34].

2.1.3 Unpacking and Extraction Techniques

Once compressed data is identified, it must be carved and unpacked. Conventional carving relies on "Header/Footer" matching, searching for specific signatures (0x504B0304 for ZIP) [49]. However, in untagged or damaged files, these headers may be missing or corrupted.

Control Flow and Graph-Based Recovery

For untagged executable binaries (such as ELF files), fragmentation poses a severe issue. *Bin-Carver* introduces a method to recover these files by leveraging the "road map" of the ELF header and the internal control flow of the binary code. By identifying function caller-callee relationships within the machine code, this method can logically link non contiguous data blocks, overcoming fragmentation that would defeat linear carving methods [22].

2.2 Existing Tools

2.2.1 Binwalk

Binwalk is a specialized tool designed primarily for firmware analysis, though it can handle a wide variety of other data types as well. Its most recent version, Binwalk v3, has been completely re-written in Rust to provide better performance and more accurate results. The core utility of the tool lies in its ability to scan files to identify and extract embedded data or sub-files that might be hidden inside. Additionally, it features entropy analysis capabilities, which allow it to pinpoint sections of a file that might be encrypted or compressed, even if the specific method used is unknown. It is highly flexible and can be used as a standalone command-line utility or integrated as a library into other Rust-based development projects.[23]

2.2.2 Unblob

Unblob is an accurate and fast extraction suite designed to parse binary blobs across more than thirty different archive, compression, and file-system formats. It functions by recursively extracting content and carving out unidentified chunks of data that have not been accounted for by standard patterns, turning "unknown unknowns" into "known unknowns." Originally developed and maintained by ONEKEY, the tool is licensed under the MIT license and is accessible via both a command-line interface and a Python library. While it is a primary companion for firmware analysis and reverse engineering, its modular design also makes it suitable for data recovery, memory forensics, and malware analysis. The system prioritizes security and precision by sticking strictly to format standards for calculating offsets and discarding overlapping chunks to minimize false positives. Technically, unblob leverages Python for its main structure while utilizing Hyperscan for pattern matching and Rust for performance-heavy tasks like entropy calculation to ensure it remains high-performance.[40]

2.2.3 Ghidra

Ghidra is a powerful, open-source software reverse engineering framework originally developed by the National Security Agency (NSA) and released to the public in 2019. It provides a comprehensive suite of tools for analyzing binary files, including capabilities for disassembly, decompilation, and debugging across multiple operating systems like Windows, macOS, and Linux. Its standout feature is a processor-agnostic decompiler that translates machine code into a high-level C-like representation, allowing researchers to more easily understand the logic of complex software. Designed to be highly extensible, Ghidra supports scripting and plugins to automate tasks, making it an essential tool for malware analysis, vulnerability research, and deep software inspection.[37]

2.3 Related Work

While the background section established the fundamental techniques for static detection and carving, significant research has focused on advanced feature engineering for data that is difficult to classify and dynamic systems for generic unpacking. This section reviews works that extend beyond basic entropy analysis and static extraction to include advanced feature engineering and anomaly detection [21].

2.3.1 Advanced Feature Engineering for Detection

A core difficulty in analyzing untagged binary blobs is determining whether a region is compressed, encrypted, or simply contains randomized data. As noted in Section 2.1.2, basic Shannon Entropy (\mathcal{H}) is the primary feature used for this, where values close to the maximum (8.0 bits per byte) indicate a high degree of randomness. De Gaspari et al. (2022) [16] demonstrate that while entropy is a standard metric in digital forensics for identifying packed regions, it often yields false positives when distinguishing high-quality encryption from compression.

Similarly, Jeong et al. (2010) leverage high entropy in their work "*Generic unpacking using entropy analysis*" [27] as a trigger for detection. However, to differentiate between various data states and compression families, researchers must employ more sophisticated measures. For example, *Nagaraj and*

Balasubramanian (2017), in their article *"Three perspectives on complexity: Entropy, compression, subsymmetry"* [36], propose comparing \mathcal{H} with metrics like Lempel-Ziv Complexity (LZ) and Effort-To-Compress (ETC) complexity. These advanced measurements help understand the hidden structure and patterns in the data, which is essential for distinguishing compression methods.

2.3.2 Advanced Extraction: Bit-by-Bit Processing

Standard digital forensic tools typically rely on identifying file signatures (e.g., `0x504B0304` for ZIP archives) to determine the starting offset of raw compressed data blocks. When these headers are corrupted or missing, traditional tools fail because they cannot locate the data entry point. In such scenarios, a "bit-by-bit" processing approach is utilized to recover data by accounting for bit-level misalignment.

Unlike standard file carving that operates on byte boundaries, compressed data is fundamentally a bitstream. In the DEFLATE compression scheme, data blocks are packed into bytes starting from the least-significant bit (LSB). If a file is corrupted at its beginning, the remaining valid data may no longer align with byte boundaries, a phenomenon known as a bit-shift.

The bit-by-bit algorithm functions as an iterative alignment wrapper for the decompressor:

- The algorithm attempts to decompress the raw data from its current bit position.
- If decompression fails, the first LSB is removed from the stream.
- The remaining data is shifted, effectively testing a new bit-offset for the entire stream.
- This process repeats until a valid decompressed chunk is identified or the end of the block (EOB) is reached.

The success of this method is heavily dependent on the DEFLATE mode used during compression. Recovery is highly effective for Mode 2 (Fixed Huffman) because the decompression tables are static and built into the algorithm, allowing for the recognition of valid data as soon as the correct bit-alignment is found. However, Mode 3 (Dynamic Huffman) remains difficult to recover if the header is damaged, as the specific Huffman trees required for decoding are stored within that corrupted header information. Experimental results indicate that while this bit-level trial-and-error approach is computationally intensive, the extraction time scales linearly with the file size [44].



3

Theory

3.1 Firmware

Firmware is a specialized category of low-level software programmed directly into a hardware device's non-volatile memory [25]. Unlike high-level application software, which is designed to be hardware-independent and managed by an operating system, firmware provides the fundamental instructions that allow a specific piece of hardware to communicate with other computer components [38]. It possesses direct access to the physical memory space of hardware devices, acting as the essential intermediary between the physical circuitry and the logical layers of the system [1].

3.2 Binary Blobs

In the fields of embedded systems and software reverse engineering, the term **Binary Blob** (*Binary Large Object*) refers to a closed-source, compiled software component that is distributed solely in machine code format [14]. Unlike open-source software, blobs lack human-readable source code, and unlike structured executable files, they frequently lack the formatting metadata, file headers, or symbol tables required to map their internal architecture.

In modern firmware design, manufacturers heavily rely on binary blobs to interface with proprietary hardware components—such as Wi-Fi radios, cellular modems, or specialized microcontrollers—without exposing intellectual property or trade secrets to competitors [3].

To a forensic analyst or security researcher, a binary blob acts as a complete "black box." Because the data stream contains no explicit instructions detailing how its contents are organized or compressed, standard operating system tools cannot natively mount or interpret it. Consequently, auditing these blobs for vulnerabilities or extracting hidden assets requires specialized techniques, such as statistical entropy analysis, to differentiate between executable machine instructions, compressed archives, and encrypted data payloads [50].

Depending on the stage of the firmware compilation process or the specific method used to extract the data from a physical flash chip, a binary blob can manifest in several different structural formats. The following subsections outline the primary representations encountered during firmware analysis.

3.2.1 Executable and Linkable Format (.elf)

The **Executable and Linkable Format (ELF)** is the standard binary format for object files, executables, shared libraries, and core dumps on Unix-like systems and many embedded architectures, including ARM Cortex-M [53]. Unlike a raw binary image, an ELF file is a structured container that encapsulates not only the machine code but also significant metadata required for the linking and loading process.

Structure-wise, an ELF file consists of an *ELF Header*, a *Program Header Table*, and a *Section Header Table*. The file allows for the separation of data into distinct sections, most notably:

- **.text**: The executable machine instructions (code).
- **.data**: Initialized global and static variables.
- **.bss**: Uninitialized data (Block Started by Symbol), which occupies no space in the file but reserves memory at runtime.
- **.symtab** and **.strtab**: Symbol and string tables used for debugging, which map human-readable function names to virtual memory addresses.

In the context of this research, the ELF file serves as the "Ground Truth" for analysis. It contains the symbolic information necessary to correlate high-entropy memory regions with specific software functions [45].

3.2.2 Flat Binary Format (.bin)

The **Flat Binary** file represents the raw machine code image exactly as it is intended to reside in the target device's non-volatile memory (Flash). Unlike the ELF format, the binary format contains no metadata, headers, section information, or symbol tables [54, 45]. Technically, a flat binary is a direct memory dump of the allocatable sections (e.g., `.text` and `.data`) extracted from the ELF file [45]. The file is a continuous stream of bytes where the file offset corresponds linearly to the physical memory offset on the device. Any gaps in memory (padding) must be filled with dummy data (often `0xFF` or `0x00`) to maintain alignment. For entropy analysis and compression detection, the flat binary is the primary subject of interest because it represents the "black box" firmware that an external observer or attacker would extract from a physical chip [7].

3.2.3 Linker Map File (.map)

The **Map File** is a diagnostic text artifact generated by the linker (e.g., `ld`) during the final build stage [55]. It provides a comprehensive spatial layout of the firmware's utilization of the target microcontroller's memory address space [45].

The map file details the precise memory address and size of every symbol (function, variable, and constant) within the binary image. It explicitly documents how the linker has resolved symbols and discarded unused code (dead code elimination) [55]. Academic literature on embedded optimization utilizes map files to audit memory footprint and verify the placement of critical interrupt vector tables [57, 55]. In this study, the map file functions as a validation tool, allowing for the cross-referencing of detected compressed regions against the known memory layout defined by the developer.

3.3 File Headers and Metadata

In the context of binary analysis and data recovery, a **header** is a structured block of supplemental data placed at the beginning of a file or data stream [6]. It serves as the primary metadata layer, providing the necessary instructions for a software interpreter or decompressor to decode the subsequent payload correctly [47].

A header typically fulfills three primary functions:

- **Identification:** It contains "Magic Numbers" (unique byte sequences) that identify the file type to the operating system (e.g., `0x1F 0x8B` for Gzip or `0x7F 0x45 0x4C 0x46` for ELF) [6].
- **Structural Mapping:** It defines the boundaries of the data, including the total file size, compressed block sizes, or specific memory offsets required for loading [47].
- **Parameter Configuration:** For compression algorithms, the header specifies critical variables such as the dictionary size, the algorithm version, and whether optional features like header checksums are enabled [47].

The Header as a Single Point of Failure

The structural reliance on headers creates a significant challenge in firmware forensics [30]. In standard *Byte-by-Byte* processing, the header is a critical dependency; if even a single bit within a 10-byte header is corrupted, a standard decompressor will fail to recognize the entire multi-megabyte payload. This vulnerability is the primary motivation for the **Bit-by-Bit** recovery method discussed in Section 3.7. By understanding that the header is merely a "map" and not the "terrain," bit-level analysis can bypass corrupted metadata to target the raw entropy of the compressed payload itself [30].

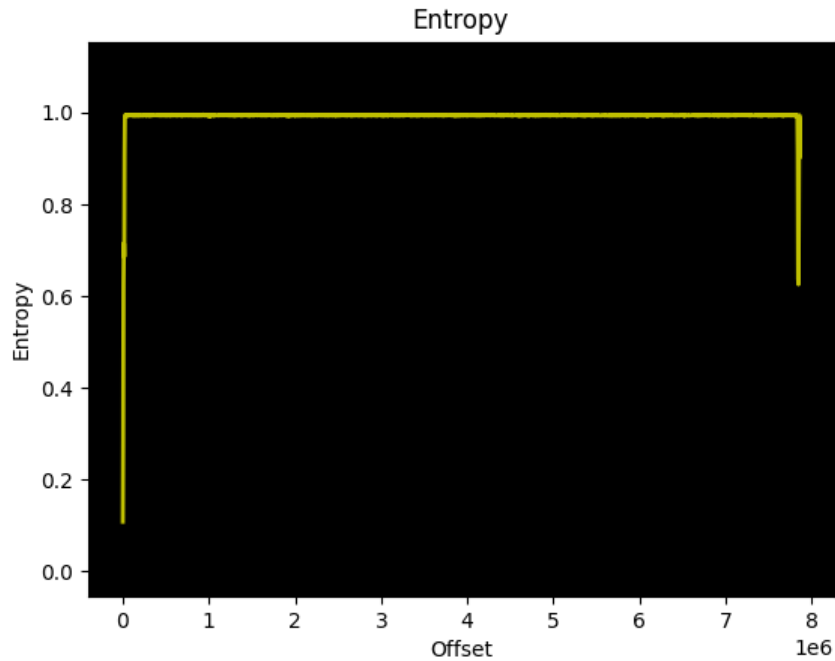
3.4 Anatomy of router firmware

The firmware analyzed is the European version of the TP-Link Archer C60 v3 router software, specifically release 201231, which was packaged for distribution in April 2022 [31]. This binary image serves as the complete operating system and instruction set for the MIPS-based hardware, consolidated into a single file of approximately 7.9 megabytes [41]. Structurally, it is a multi-layered archive comprising a U-Boot bootloader, a compressed Linux Kernel version 3.3.8, and a Squashfs filesystem that stores the router's web management interface and core networking utilities [41, 38].

This particular firmware release was primarily focused on security and stability improvements, notably modifying the default wireless encryption to WPA2/AES and resolving issues within the parental control functions regarding daylight saving time delays [31]. From a systems architecture perspective, the binary demonstrates a classic embedded design where disparate components are concatenated and compressed using varying algorithms like LZMA and XZ to maximize the efficiency of the limited on-board flash memory [38]. By analyzing this specific version, we gain a comprehensive view of how consumer-grade network appliances manage the transition from raw hardware initialization to a functional, high-level user environment.

3.4.1 Binwalk analysis

Scanning for compression



Searching for signatures

DECIMAL	HEXADECIMAL	DESCRIPTION
21796	0x5524	U-Boot version string, "U-Boot 1.1.4-g402e8420-dirty"
21860	0x5564	CRC32 polynomial table, big endian
23156	0x5A74	uImage header, image name: "u-boot image", OS: Linux, CPU: MIPS, compression: lzma
23220	0x5AB4	LZMA compressed data, uncompressed size: 92964 bytes
62571	0xF46B	uImage header, image name: "MIPS OpenWrt Linux -3.3.8", OS: Linux, CPU: MIPS, compression: lzma
62643	0xF4B3	LZMA compressed data, uncompressed size: 2793252 bytes
1008847	0xF64CF	Squashfs filesystem, little endian, version 4.0, compression: xz, size: 6833108 bytes
7842533	0x77AAE5	XML document, version: "1.0"
7854526	0x77D9BE	gzip compressed data, from Unix

Listing 3.1: Binwalk analysis of ArcherC60v3 firmware binary.

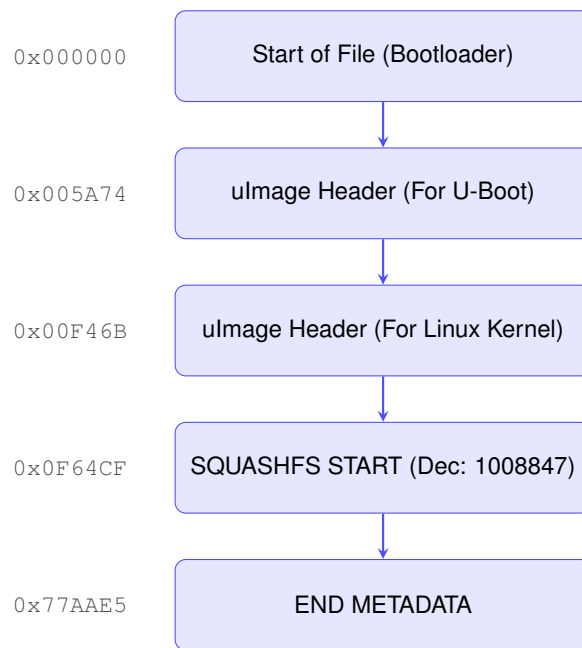


Figure 3.1: Main overview of the Binwalk analysis of the ArcherC60v3 firmware binary.

Extraction

```
DIRECTORY CONTENTS: _ArcherC60v3_eu-up-ver1-2-0-P1.bin.extracted/
```

FILE NAME	TYPE / DESCRIPTION
5AB4	Raw carved chunk (U-Boot segment)
5AB4.7z	Decompressed LZMA archive (U-Boot binaries)
F4B3	Raw carved chunk (Kernel segment)
F4B3.7z	Decompressed LZMA archive (Linux Kernel)
F64CF.squashfs	Carved Squashfs filesystem image
squashfs-root/	EXTRACTED ROOT FILESYSTEM (OS directory tree)
77D9BE	Raw carved chunk (Metadata segment)
77D9BE.gz	Decompressed Gzip archive (Configuration data)

Listing 3.2: Contents of the extraction directory after processing with binwalk.

The Role of Intermediate Artifacts in Firmware Extraction

During the automated extraction of the *ArcherC60v3* firmware, several files are generated that act as **intermediate artifacts** [39]. These files represent the discrete stages of the reverse engineering pipeline, moving from a monolithic binary to an organized filesystem [14].

- **Carved Raw Blobs (e.g., 5AB4, F4B3):** These are the primary results of the "carving" phase. They are bit-for-bit copies of segments located at specific offsets within the original binary. They remain compressed and are named after their starting hexadecimal offset.
- **Labeled Archives (e.g., 5AB4.7z, 77D9BE.gz):** These artifacts represent the "identification" phase [39]. Once a tool like *unblob* determines the compression algorithm (e.g., LZMA or Gzip), it appends the appropriate extension. These files are essential for manual verification if the automated extraction fails.
- **Filesystem Images (e.g., F64CF.squashfs):** This is a critical intermediate state. It is a standalone, compressed disk image. While it can be mounted as a loop device in Linux, it must be further processed by *unsquashfs* to reach the final anatomy stage.

In the context of this anatomy, these artifacts prove the **recursive nature** of firmware: the binary contains a container, which contains an archive, which finally contains the files [14].

```
squashfs-root/
|-- bin/
|   |-- busybox           <-- The monolithic binary executable
|   |-- ash -> busybox    <-- Symlink: System Shell
|   |-- ls -> busybox     <-- Symlink: File listing utility
|   |-- login.sh         <-- Custom TP-Link login wrapper
|   `-- ... (45 others)
|
|-- etc/
|   |-- config/          <-- UCI configuration files (network, wireless)
|   |-- init.d/          <-- System boot sequence scripts
|   |-- dropbear/        <-- SSH server configuration
|   |-- passwd           <-- User accounts table
|   |-- shadow           <-- Encrypted password hashes
|   `-- openwrt_release  <-- Distro version (OpenWrt base)
|
|-- lib/
|   |-- firmware/        <-- Binary blobs for WiFi hardware
|   |-- ld-uClibc.so.0    <-- C Standard Library (uClibc)
|   |-- libcrypt.so.0    <-- Encryption library
|   `-- modules/         <-- Kernel drivers (KO files)
|
|-- www/                  <-- Web Server Root (Admin Interface)
```

Listing 3.3: Abbreviated directory structure of the extracted SquashFS filesystem.

3.4.2 The Boot Execution Flow: From Artifacts to Operation

While the previous sections analyzed the firmware as a static collection of files, the router views these components as a sequential execution list. By mapping the extracted artifacts to the device's boot process, we can reconstruct the *live anatomy* of the Archer C60v3.

1. Stage 1: Hardware Initialization (Offset 0x0)

Artifact: 0-23220.unknown

Upon power delivery, the MIPS processor is hardwired to fetch instructions from the physical address corresponding to offset 0x0 of the flash chip. This "Unknown" segment contains the **Primary Bootloader**. Its sole responsibility is to initialize the DDR RAM and CPU clocks, creating a volatile workspace for the more complex stages that follow.[35]

2. Stage 2: Kernel Loading (Offset 0x5AB4 & 0xF4B3)

Artifacts: 5AB4.7z, F4B3.7z

Once the RAM is active, the bootloader parses the **uImage headers** identified at offsets 0x5A74 and 0xF46B. These headers act as a table of contents, instructing the bootloader to read the subsequent LZMA-compressed data chunks. The router decompresses these chunks directly into RAM effectively constructing the running Linux Kernel in volatile memory. [2]

3. Stage 3: Filesystem Mounting (Offset 0xF64CF)

Artifact: F64CF.squashfs

Unlike the kernel, the root filesystem is too large (6.52 MB) to be fully decompressed into the router's limited RAM. Instead, the Linux kernel **mounts** the raw F64CF.squashfs data directly from the flash chip as a read-only block device. This utilizes the "Loop Device" methodology described previously; the system decompresses individual binaries (like /bin/busybox or /bin/httpd) on-the-fly only when they are executed. [51]

4. Stage 4: Configuration Application (Offset 0x77AAE5)

Artifacts: 77D9BE.gz, XML Metadata

The final phase involves the "User Space" initialization. The system reads the trailing XML and Gzip artifacts to determine the device's specific model identity and default configuration. These settings are overlaid onto the volatile filesystem, applying the unique SSIDs, passwords, and region codes that define the user's experience. [33]

Operational Context: Live Mount vs. Static Extraction

It is critical to distinguish between the researcher's view and the router's view of the filesystem.

- **The Router (Live View):** Interacts with `F64CF.squashfs`. It sees a compressed container and uses CPU cycles to "peek" inside it dynamically via the kernel's Squashfs implementation, which decompresses data into the page cache only as needed [32].
- **The Researcher (Static View):** Interacts with `squashfs-root/`. We use `unsquashfs` to permanently decompress the entire structure into a standard folder tree, allowing for global searches and static analysis that the router's hardware limitations—specifically its limited RAM and CPU overhead—would never permit [43].

3.5 Data Compression

Data compression is the process of generating a "compressed" representation of an object—referred to as a *message*—that occupies fewer bits than the original source [48]. This task consists of an encoding algorithm that generates the compressed representation and a decoding algorithm that reconstructs the original message or an approximation thereof [47].

3.5.1 The Model-Coder Framework

Modern compression algorithms are conceptually divided into two distinct components: the **model** and the **coder** [48].

- **The Model:** This component captures the probability distribution of the input by identifying structure or patterns, such as repeated characters in text or spatial correlations in images. It effectively provides the "bias" or unbalanced probability distribution upon which compression relies.
- **The Coder:** The coder takes the probability biases generated by the model and produces actual bitstrings. It achieves reduction by shortening high-probability messages and lengthening low-probability ones.

As established by *Shannon* (see Section 2.1.1), information theory acts as the glue between these components, relating probabilities to information content and code length. The theoretical limit of compression is governed by **Entropy** (H), defined for a set of messages S with probabilities $p(s)$ as:

$$H(S) = \sum_{s \in S} p(s) \log_2 \frac{1}{p(s)} \quad (3.1)$$

3.5.2 Operational Mechanics

Compression is fundamentally about probability; if an algorithm could shorten every possible bit sequence, it would violate simple counting arguments. In practice, algorithms like *Huffman coding* generate optimal prefix codes where no bit-string is a prefix of another one [24]. More advanced techniques, such as *Arithmetic coding*, allow the information from a sequence of messages to be combined into a single interval on the number line, asymptotically approaching the self-information of the messages [56].

3.6 Decompression

Decompression is the inverse operation of reconstructing the original message—or an approximation in lossy systems—from the compressed representation. Because the encoder and decoder are "intricately tied together," they must both utilize the same shared model and representation [48].

The decoding process typically mirrors the encoding logic:

- **Prefix Decoding:** For codes like Huffman (see Section 3.9.1), the decoder traverses a binary tree based on incoming bits until a leaf node is reached, at which point it outputs the message and returns to the root [24].
- **Arithmetic Decoding:** The decoder identifies the message value by determining which message interval contains the received bitstream value, then narrows the sequence interval accordingly [56].

A critical requirement for successful decompression, particularly in integer-based implementations, is that all rounding and expansion rules must be identical in both the encoder and decoder. This ensures that the decoder's internal state exactly follows the bounds established during the compression phase [4].

3.7 Bit-by-Bit Processing for Data Recovery

The bit-by-bit processing method is an innovative approach designed to recover damaged compressed files when header blocks are corrupted or missing [8]. While traditional tools rely on file signatures and headers to locate raw data blocks, this method operates at the bit level to bypass such dependencies [20].

This methodology is specifically applicable to the DEFLATE algorithm's "Mode 2," which utilizes LZ77 and a fixed Huffman coding scheme [17]. Because the Huffman table in Mode 2 (technically defined as BTYPE 1 in the standard) is hardcoded into the algorithm rather than the file itself, it is possible to recover the data even if the file's original header is destroyed [17, 8].

3.7.1 The Iterative Decompression Algorithm

The process follows an automated, iterative loop to find a valid decompression starting point. The steps are executed as follows:

- The algorithm takes the damaged raw compressed data as input.
- It attempts to decompress the block; if successful, the output is saved as a data chunk.
- If decompression fails, the algorithm removes the first Least Significant Bit (LSB) and repeats the attempt [8].
- This cycle continues bit-by-bit until a block is successfully decompressed or the End of Block (EOB) string is reached [17].

3.7.2 Bit Removal and Shifting Mechanism

DEFLATE blocks are packed into bytes starting with the LSB [17]. To find the correct alignment, the algorithm treats the data stream like a shift register, discarding one bit at a time to re-align the stream with the expected fixed Huffman code tree [8].

The mechanism involves:

- Removing the 8th bit from the first byte of the data stream.

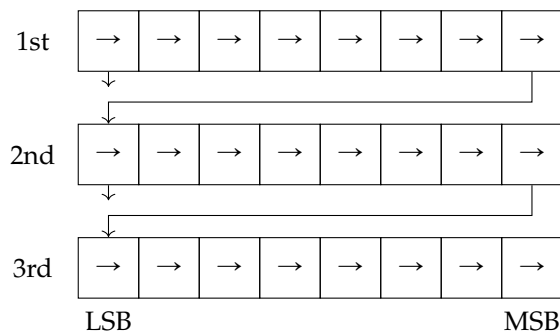


Figure 3.2: Bit-by-bit shifting algorithm.

- Shifting the remaining data stream toward the MSB, similar to a register shift.
- Repeating this process as many times as necessary to check every possible bit alignment until the block is decompressed or the end is reached.

By automating this granular extraction, the bit-by-bit method can recover segments of data from live memory dumps or heavily corrupted archives that traditional forensics tools would ignore. [44]

3.8 Byte-by-Byte Processing and its Limitations

In contrast to bit-level extraction, the byte-by-byte approach represents the standard mechanism utilized by most current data recovery utilities [6]. This method relies on the identification of specific file signatures and headers that are expected to reside at standard byte boundaries within a storage medium or memory dump [20].

3.8.1 Standard Recovery Mechanism

Traditional recovery tools function by scanning a data stream for known signatures or "magic numbers" that define the start of a file or container member [6]. Once a signature is detected, the algorithm reads the offset provided in the header to locate the raw compressed data blocks. The tool then attempts to reorganize and decompress the archive based on this structural information. Examples of such identification headers discussed previously include:

- **Gzip Identification:** The first 10 bytes of every Gzip member, starting with the identification bytes **ID1 (0x1f)** and **ID2 (0x8b)**, which must be aligned at the byte level for standard decoders [18].
- **LZ4 Magic Numbers:** Frame formats that begin with a distinct 4-byte Magic Number, such as **0x184C2102** for legacy frames [10].
- **XZ Stream Headers:** A 12-byte header starting with a fixed 6-byte magic sequence: **0xFD 0x37 0x7A 0x58 0x5A 0x00** [12].

The limitation of this approach is its absolute dependency on metadata integrity; because standard decoders treat the data as a sequence of bytes, a single bit-flip in the header or an unexpected bit-offset in the compressed stream will render the entire payload unrecoverable by these tools [20].

3.8.2 Comparison with Bit-by-Bit Processing

The limitations of the byte-by-byte approach become evident when dealing with heavily corrupted data. Because the DEFLATE algorithm packs data into bits rather than whole bytes, any corruption that causes a bit-shift in the data stream renders the byte-by-byte method ineffective [17, 5].

The primary differences include:

- **Alignment Dependency:** Byte-by-byte tools require data to be aligned with the 8-bit byte structure of the file system [20]. Conversely, bit-by-bit processing treats the file as a continuous stream of bits, allowing for recovery regardless of physical byte boundaries.
- **Header Reliance:** These tools are generally incapable of recovering data if the signature or header block is missing or damaged, as they cannot identify the start or calculate the necessary offset to the raw data blocks. Bit-level analysis bypasses this by seeking the first valid Huffman sequence within the entropy of the payload [8].
- **Search Specificity:** While byte-by-byte scanning is computationally efficient for healthy files, it lacks the granularity required to salvage data from scenarios like live memory dumps where compressed fragments may begin at arbitrary bit offsets [30, 5].

While byte-by-byte processing is sufficient for recovering deleted files with intact headers, the bit-by-bit method remains a critical solution for forensic scenarios involving raw, unaligned, or severely damaged compressed streams where the header is completely lost [8].

3.9 The DEFLATE Compression Pipeline

The DEFLATE algorithm, formally standardized in RFC 1951 [17], is a lossless data compression engine that forms the core of many container formats, including Gzip. It achieves compression by passing data through a two-stage pipeline: redundancy elimination via LZ77, followed by bit-level reduction using Huffman coding.

3.9.1 Different Modes

A DEFLATE stream is divided into a series of blocks. Every block begins with a 3-bit header containing one final-block flag (`BFINAL`) and two block-type bits (`BTYPE`). The `BTYPE` bits dictate the specific compression mode applied to that block's payload [17].

Mode 1: Fixed Huffman Codes

When the `BTYPE` bits are set to `01`, the block is compressed using Fixed Huffman codes. From a forensic recovery standpoint, this mode is highly recoverable. Because the "dictionary" (the specific Huffman tree structure) is hardcoded directly into the DEFLATE standard, a decompressor does not need a localized header to decode the symbols. As long as the correct bit-alignment is established, the data can be reliably unpacked.

Mode 2: Dynamic Huffman Codes

When the `BTYPE` bits are set to `10`, the block is compressed using Dynamic Huffman codes. This mode provides superior compression but is extremely difficult to blindly recover. In this mode, a custom-built Huffman table is generated specifically for the block's content and stored immediately following the block header. If this internal, custom dictionary is damaged, or if the analyst cannot locate its exact bit-aligned starting position, the remainder of the block is mathematically undecodable [17].

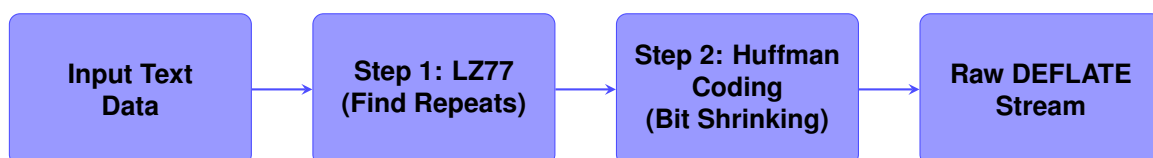


Figure 3.3: The DEFLATE compression pipeline, illustrating the two-stage reduction of data via LZ77 and Huffman coding.

3.9.2 Phase One: Finding Repeated Patterns with LZ77

The first step of the DEFLATE process is to search for redundancy using the LZ77 algorithm [42]. This method assumes that in most files, certain sequences of characters appear more than once. Instead of storing a repeated phrase again, the compressor looks back at what it has already processed within a 32 KB sliding window (see Figure 3.4).

As illustrated, the algorithm compares a "Lookahead Buffer" of upcoming data against a "Search Buffer" of historical data. When it finds a match, it replaces the duplicate text with a small pointer. This pointer contains two values: the distance, which tells the decompressor how far back to look, and the length, which tells it how many characters to copy forward. If a character is unique and has no match, it is kept as a literal byte. This stage effectively reduces the number of characters in the file by turning long strings of text into short numerical references.

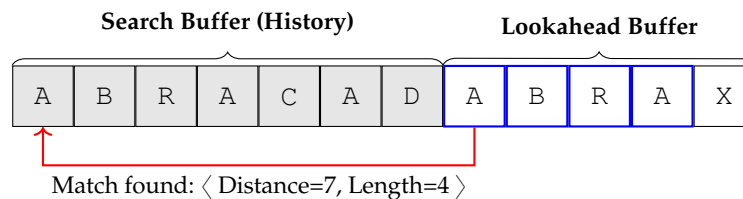


Figure 3.4: The LZ77 sliding window algorithm compressing the string "ABRACADABRAX". The compressor finds the repeated "ABRA" pattern 7 bytes back in the history buffer.

3.9.3 Phase Two: Bit Reduction via Huffman Coding

Once the data is converted into a mix of literals and pointers, it undergoes Huffman coding to shrink the size of those symbols at the bit level [17]. Standard data storage uses a fixed number of bits for every character, which is inefficient because some characters appear much more frequently than others. Huffman coding solves this by creating a mathematical tree where the most common symbols are given very short bit codes, such as two or three bits, while rare symbols are given longer codes. In a DEFLATE stream, the literals and length codes are combined into one Huffman tree (see Figure 3.5), while a separate tree is often used for distances. This ensures that the bulk of the data is represented by the smallest possible amount of digital "weight," leading to a significantly smaller final file size.



Figure 3.5: Huffman Tree and corresponding variable-length codes [17].

3.9.4 Phase Three: The Raw DEFLATE Block Structure

The final phase involves organizing the compressed bits into a structured format called a Raw DEFLATE Stream (see Figure 3.6). According to RFC 1951 [17], the data is divided into blocks so that the compressor can change its strategy depending on the content.

As shown in the figure, each block begins with an unaligned 3-bit header that includes one bit to signal if it is the last block in the file (**BFINAL**) and two bits to define the type of compression used (**BTYPE**). This allows the algorithm to use different Huffman trees for different parts of a file, optimizing the compression for each specific section. The block concludes with a specific "End of Block" marker, which tells the decompressor that it has reached the end of the current compressed sequence and should prepare for the next header or the final termination of the stream.

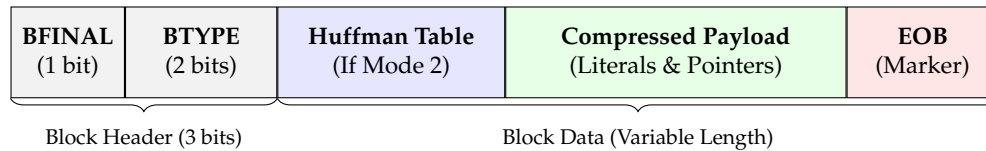


Figure 3.6: The raw DEFLATE block structure showing the unaligned 3-bit header preceding the compressed payload.

3.10 Gzip Format Structure

The Gzip (GNU zip) format, defined in RFC 1952 [18], is a lossless data compression format. It is important to distinguish Gzip as a container format from the underlying compression algorithm it typically employs, which is the DEFLATE algorithm [17]. A Gzip file consists of a series of "members" (compressed data sets). Each member is composed of a header, a compressed data block, and a footer (trailer).

3.10.1 Header and Metadata

As specified by RFC 1952 [18], the first 10 bytes of every Gzip member are mandatory and establish the file's identity and processing parameters. This fixed header consists of the following sequential fields (see Figure 3.7):

- **ID1 (0x1F) & ID2 (0x8B)**: The identification bytes, or "magic numbers," used by parsers to reliably recognize the Gzip format.
- **CM (Compression Method)**: Specifies the underlying algorithm. A value of 8 explicitly denotes the DEFLATE algorithm [17].
- **FLG (Flags)**: A crucial bitmask byte that indicates the presence of optional metadata fields (detailed below).
- **MTIME (Modification Time)**: A 4-byte Unix timestamp recording when the original file was last modified.
- **XFL (Extra Flags)**: Indicates the compression effort used (e.g., maximum compression versus fastest algorithm).
- **OS (Operating System)**: Identifies the filesystem type of the original host (e.g., Unix, FAT).

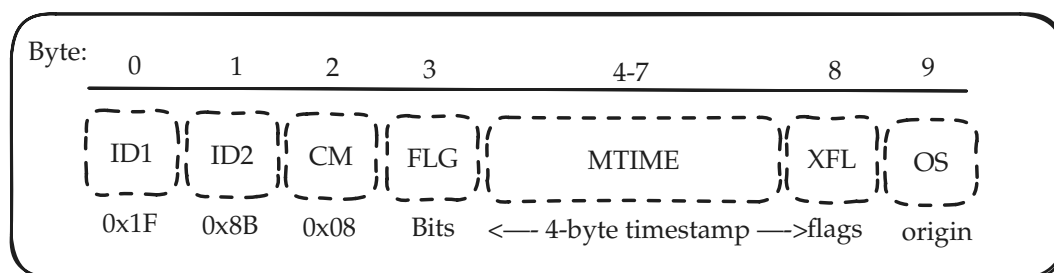


Figure 3.7: The Gzip member format illustrating the fixed 10-byte header.

The **FLG** byte acts as a set of toggles for variable-length optional fields that immediately follow the fixed 10-byte header. If specific bits in the **FLG** byte are set to 1, the decompressor expects to parse the corresponding extensions [18]:

- **FTEXT**: A hint that the payload consists of ASCII text rather than binary data.
- **FHCRC**: A 2-byte CRC16 header checksum.
- **FEXTRA**: A field reserved for application-specific metadata.
- **FNAME**: The original, null-terminated filename.
- **FCOMMENT**: A null-terminated string containing human-readable notes.

The **FHCRC** (Header Checksum) is particularly important for metadata integrity. When enabled, it allows the decompressor to verify that critical metadata—such as filenames or timestamps—has not been corrupted before attempting to process the compressed payload.

3.10.2 Compressed Payload and Integrity Trailer

The core of the Gzip member is the compressed payload, which consists of one or more blocks generated via the DEFLATE algorithm [17]. As specified in RFC 1951, these blocks utilize a combination of LZ77 for redundancy elimination and Huffman coding for bit-level reduction. Each block is self-contained and can be stored without compression, or compressed using either fixed or dynamic Huffman codes defined within the block's own header.

Following the compressed data, an 8-byte trailer provides the final verification for the decompressor (see Figure 3.8). This mandatory trailer consists of two distinct 4-byte fields:

- **CRC32 (Cyclic Redundancy Check)**: Calculated against the **uncompressed** original data to verify integrity.
- **ISIZE (Input Size)**: Represents the exact size of the original uncompressed input data modulo 2^{32} .

Unlike the optional header checksum, the trailing CRC32 is mandatory. It utilizes the ISO 3309 standard generator polynomial:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

During decompression, the receiver recalculates the CRC32 of its output stream. If this newly calculated value does not perfectly match the stored CRC32 in the trailer, the decompressor alerts the user to data corruption or a logic error within the decompression process.

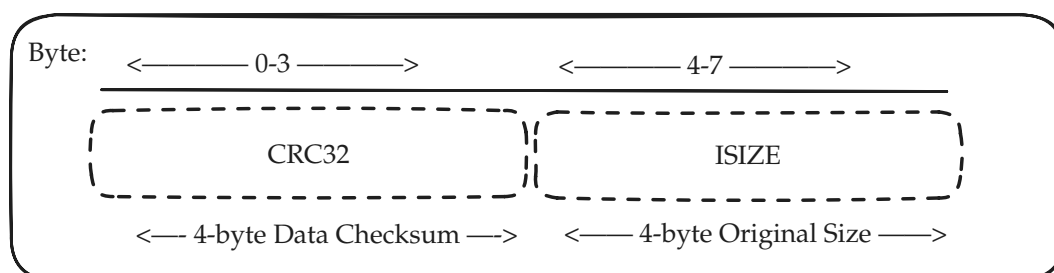


Figure 3.8: The Gzip member format illustrating the fixed 8-byte trailer.

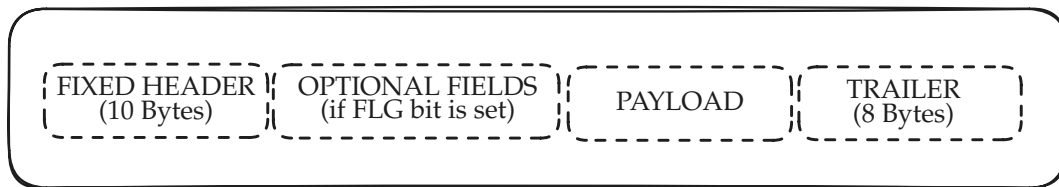


Figure 3.9: The Gzip member format illustrating the relationship between the fixed 10-byte header, the conditional optional fields (governed by the FLG byte), and the trailing integrity checks.

3.11 LZ4 Header Variants and Integrity

The LZ4 specification allows for different frame types to accommodate various environments, ranging from modern high-performance systems to legacy kernel implementations.

3.11.1 Modern Header Checksum (HC)

In the modern LZ4 frame, a specific HC (Header Checksum) byte is mandatory. This is a significant departure from the Gzip format, which lacks a default header integrity check. The HC is a truncated version of an XXHash32 [11] calculation performed over the FLG and BD bytes. If the calculated hash does not match the stored HC byte, the decoder will abort the process to prevent the allocation of memory based on potentially corrupted BD (Block Descriptor) values [9].

3.11.2 Legacy Frame Format

During forensic analysis or recovery of older Linux kernel images, the Legacy Frame format may be encountered. This format uses a distinct Magic Number (0x184C2102) and omits the FLG, BD, and HC fields entirely (see Figure 3.10). In this mode, the data blocks begin immediately after the 4-byte Magic Number. This format lacks the safety features of the modern frame but offers lower overhead for restricted environments [52].

3.11.3 The EndMark and Content Checksum

Every LZ4 frame is terminated by an EndMark, a 4-byte field set to 0x00000000 see figure 3.11. This serves as a sentinel value, distinguishing the end of the data stream from a valid BlockSize field (which would never be zero in a valid block). Following the EndMark, an optional 4-byte Content Checksum provides a final verification of the uncompressed data using the XXHash32 algorithm [9].

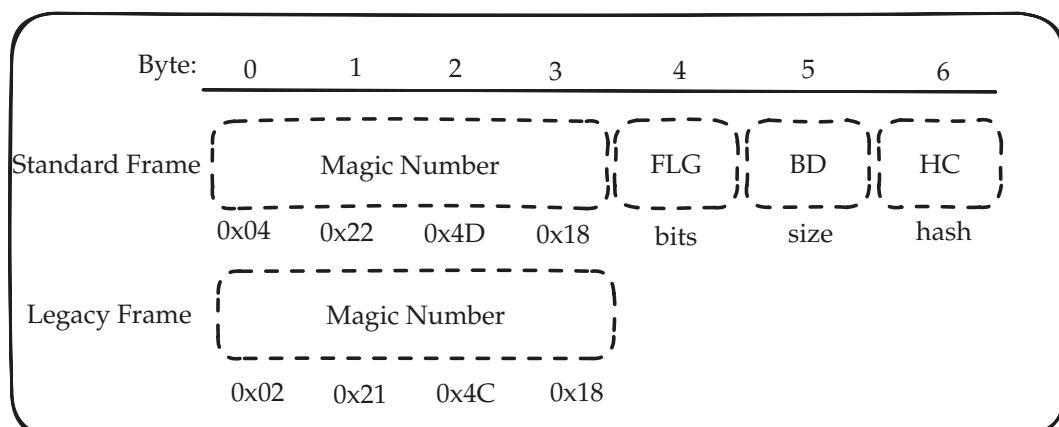


Figure 3.10: The LZ4 format illustrating the fixed 7-byte standard head and legacy frame.

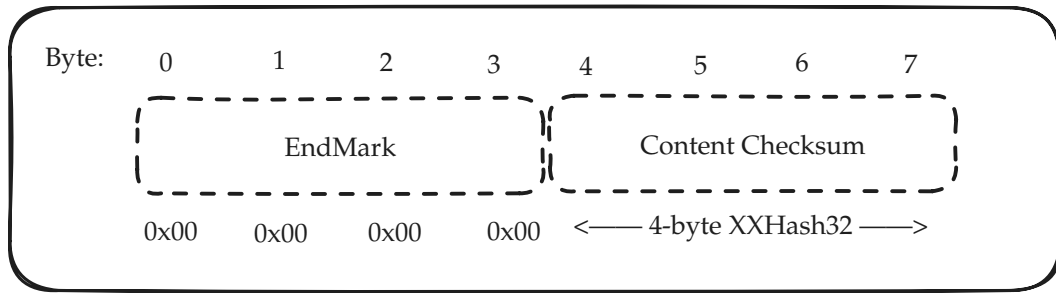


Figure 3.11: The Lz4 format illustrating the fixed 8-byte trailer

3.12 LZMA2 and the XZ Container Format

While LZMA2 is a raw compression algorithm, it is rarely used in isolation. Instead, it is typically encapsulated within the **XZ** container format. This container provides essential features such as integrity checking (CRC), random access, and file type identification. The XZ format is defined by a 12-byte **Stream Header** 3.12 at the beginning and a 12-byte **Stream Footer** 3.13 at the end [13].

3.12.1 The Stream Header

The XZ file signature ensures that the file is correctly identified by the operating system and decompression utilities [13]. The header is strictly 12 bytes long and consists of three parts (see Figure 3.12):

1. **Magic Bytes (6 Bytes):** A fixed signature sequence: `0xFD 0x37 0x7A 0x58 0x5A 0x00`.
2. **Stream Flags (2 Bytes):** Indicates the type of integrity check used for the data blocks (e.g., CRC32, CRC64, or SHA-256).
3. **Header CRC32 (4 Bytes):** A checksum calculated over the Stream Flags to ensure the header itself is not corrupt.

3.12.2 The Stream Footer

The footer (see Figure 3.13) is critical for the XZ format's random-access capabilities. It allows a decoder to parse the file backwards, locating the *Stream Index* without reading the entire file from the start. Like the header, it is exactly 12 bytes long [13]:

1. **Footer CRC32 (4 Bytes):** A checksum of the Footer's contents.
2. **Backward Size (4 Bytes):** A stored integer indicating the size of the Index field, allowing the decoder to jump backwards to the file's Table of Contents.
3. **Stream Flags (2 Bytes):** A copy of the flags from the header. These must match, or the file is considered corrupt.
4. **Magic Bytes (2 Bytes):** The file ending signature `0x59 0x5A` (ASCII "YZ").

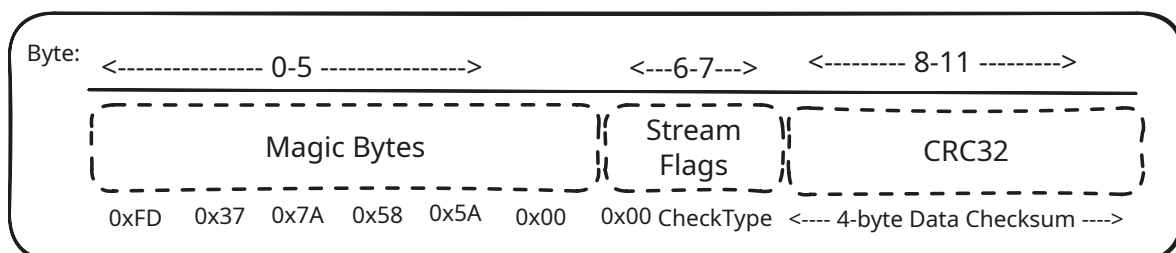


Figure 3.12: The LZMA2 format illustrating the fixed 12-byte head.

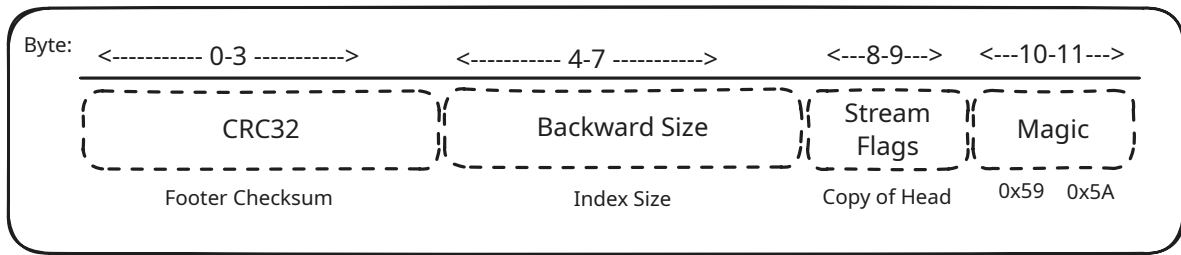


Figure 3.13: The LZMA2 format illustrating the fixed 12-byte footer.



4 Method

4.1 Dataset Generation

To evaluate the efficacy of the proposed compression detection techniques, a diverse dataset of firmware images was required. The dataset needed to represent real-world embedded systems scenarios, ranging from simple bare-metal control loops to complex real-time operating systems (RTOS). This section details the acquisition, compilation, and validation of the firmware dataset used in this study.

4.1.1 Dataset Selection and Toolchain

We utilized the *Open Firmware Dataset Builder* [15] framework to automate the generation of reproducible firmware binaries.

The dataset comprises nine distinct firmware categories, ensuring a high degree of entropy variance and structural diversity:

- **Robotics & Drones:** Quadcopter flight controllers, inverted pendulum robots, and RC car controllers. These applications typically contain dense mathematical operations and control loops.
- **Industrial Automation:** GRBL (CNC/3D printer control), reflow oven controllers, and industrial heat press logic. These represent state-machine-driven logic.
- **IoT & Operating Systems:** RIOT OS (a full embedded operating system), Modbus slave nodes, and Standard Firmata protocols.

4.1.2 Data Acquisition and Integrity Challenges

Significant challenges were encountered regarding the availability of historical source code and dependencies ("link rot"). To ensure reproducibility, a manual mirroring strategy was employed.

Dependencies that were no longer available via the original automated scripts were manually retrieved from archival storage (e.g., Azure Blob Storage, Internet Archive, and specific GitHub commit snapshots). A critical integrity verification step was introduced to detect corrupted archives (e.g., "double-header" gzip corruption) that occurred during partial downloads. We validated each archive using checksum verification and `tar` integrity checks before integration into the build pipeline.

Furthermore, several source repositories required full Git history reconstruction rather than simple snapshots. The build system relied on specific Git commit hashes to check out exact historical versions of libraries (e.g., *MAX31855*, *LiquidCrystal*). We manually cloned these repositories and repackaged them with their `.git` directories to satisfy the build system's version control requirements.

4.1.3 Compilation and Output Generation

The compilation process utilized `make` to drive the cross-compilation of source code into three distinct file formats for each project, serving different analytical purposes:

- **Raw Binary (.bin):** The flat machine code image. This serves as the primary input for the compression detection analysis, simulating the raw firmware extracted from a physical device's flash memory.
- **Executable and Linkable Format (.elf):** The compiled binary containing symbol tables and debugging information. This serves as the "Ground Truth" for validation, allowing the mapping of high-entropy regions to specific functions or data structures.
- **Memory Map (.map):** A text-based layout of the memory usage, providing a quick reference for the location of code (.text), initialized data (.data), and zero-initialized data (.bss) segments.

The final dataset consists of nine verified firmware images, categorized by device type, establishing a robust baseline for testing entropy analysis and compression detection algorithms.

4.1.4 The LZ4 Detection Challenge

A critical observation in the entropy analysis is the difficulty of establishing a universal threshold that captures LZ4 compression without incurring excessive false positives. As illustrated in Figure 5.4 and the CDF analysis in Figure 5.6, LZ4 exhibits a significantly broader distribution and lower average entropy compared to formats like SQUASHFS or LZMA.

This behavior is directly tied to the compression efficiency of the algorithm. As shown in the comparative analysis of compression reduction (Figure 4.6), different algorithms yield varying degrees of data density:

- **LZMA:** Achieves a 45.5% reduction, resulting in a near-maximum entropy spike (typically > 7.15) due to the high degree of data decorrelation.
- **Deflate:** Achieves a 26.7% reduction, consistently crossing standard detection thresholds like the Binwalk default of 0.85 (approx. 6.8 bits).
- **LZ4:** Yields only a 16.7% reduction in the test sample.

Because LZ4 is optimized for speed rather than density, the compressed output often retains residual patterns from the original uncompressed binary. Consequently, the Shannon entropy for LZ4 blocks frequently fails to cross high-confidence thresholds (e.g., 7.1). As demonstrated in the sliding window analysis (Figure 4.5), while LZMA and Deflate produce clear spikes to 0.95+, LZ4 may remain at a "moderate" entropy level that is indistinguishable from high-density uncompressed machine code, leading to it being "missed" by standard automated scanners.

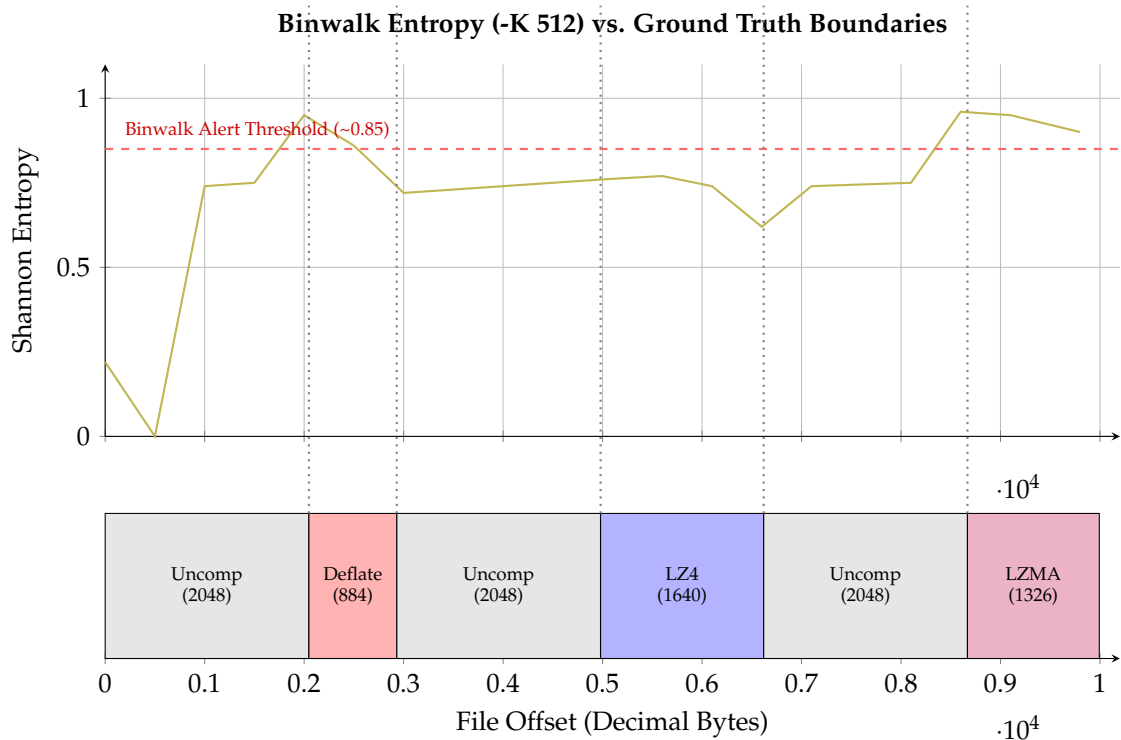


Figure 4.1: Ground Truth Configuration vs. Sliding Window (-K 512) Entropy Detection

Block Name	Start (Dec)	End (Dec)	Size (Bytes)	Expected Outcome	Actual Binwalk Result (-K 512)
Uncompressed_1	0	2047	2048	Baseline Entropy	Flat (~0.75)
Raw_Deflate_2	2048	2931	884	High Entropy Spike	Detected (Spikes to 0.95)
Uncompressed_3	2932	4979	2048	Baseline Entropy	Returns to Baseline
Raw_LZ4_4	4980	6619	1640	Moderate Entropy	Missed (Fails to cross 0.85 threshold)
Uncompressed_5	6620	8667	2048	Baseline Entropy	Flat (~0.75)
Raw_LZMA_6	8668	9993	1326	Maximum Entropy	Detected (Spikes to 0.96)

Figure 4.2: Compression Efficiency (Reduction %) by Block and Source Firmware

Block Name	Uncompressed	Compressed	Reduction (%)	Source File
Uncompressed_1	86 KB	–	–	default.bin (RIOT_OS_SHELL)
Raw_Deflate_2	86 KB	63 KB	26.7%	default.bin (RIOT_OS_SHELL)
Uncompressed_3	29 KB	–	–	reflowOvenController.ino.bin (Reflow_Oven)
Raw_LZ4_4	24 KB	20 KB	16.7%	inverted_pendulum.bin (Balancing Robot)
Uncompressed_5	33 KB	–	–	Firmware_V101-103C8.bin (Quadcopter_Drone)
Raw_LZMA_6	33 KB	18 KB	45.5%	Firmware_V101-103C8.bin (Quadcopter_Drone)

4.2 Workflow

Part 1: Initial Triage (Stages 1 & 2)

Before writing the code, we needed to establish an overview based on our initial testing of how we would tackle the automatic detection and unpacking. The first step was analyzing the .bin file with a signature scan, which was done with already existing tools such as *Binwalk*. This was later exchanged in *Binsift* where we would scan for magic bytes and compare it to our own signature detection table.

If a signature is found, it means that it is intact or a false-positive and will be extracted. As noted in the limitations, we will not account for false-positives. At first we used *Binwalk* to analyze for entropy, but this was later changed to our own entropy calculation. Our script provides a mathematically precise implementation of Shannon entropy in bits, which accurately measures the average missing information per byte for discrete datasets. However, the code lacks the normalized scaling, computational efficiency, and sophisticated edge-detection logic used by industry-standard tools like *binwalk* to identify payload

boundaries effectively. This is an area of improvement, something that has yet to be added, which is why we chose this route.

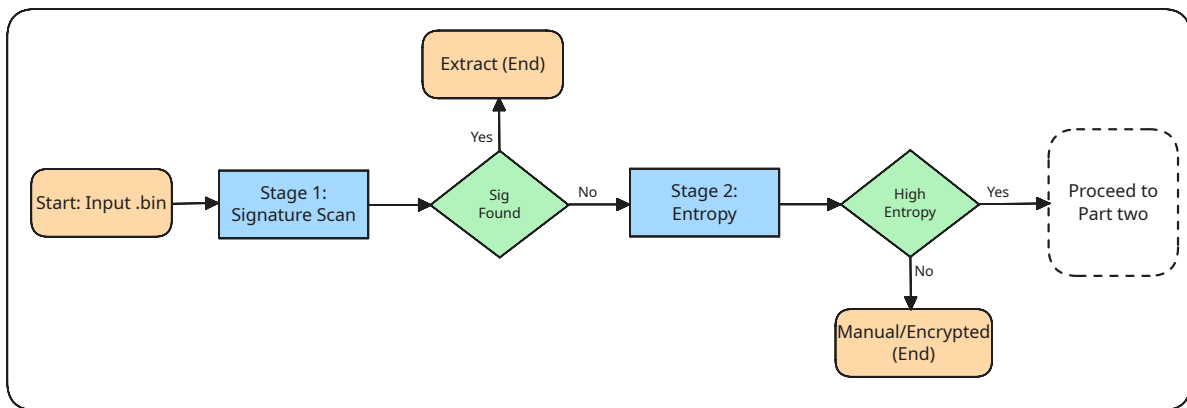


Figure 4.3: Workflow part one.

Part 2: Advanced Extraction (Stages 3A, 3B, & 4)

We intended Stage 3A: Header Injection to function as a heuristic recovery mechanism for high-entropy data segments that likely lost their metadata during file carving or transmission. The workflow began with a determination of the Algorithm Type based on the statistical unevenness of the probability distribution within the data chunk. If the segment's entropy signature matched known formats like Gzip or LZMA, we attempted to "repair" the file by prepending a standard header to the raw data stream. For chunks categorized as "Other or unidentifiable," the system initiated a brute-force search, testing various standard headers to see if the resulting compound became extractable. This later proved to be ineffective since it wasn't as of a general solution like the streaming method proved to be.

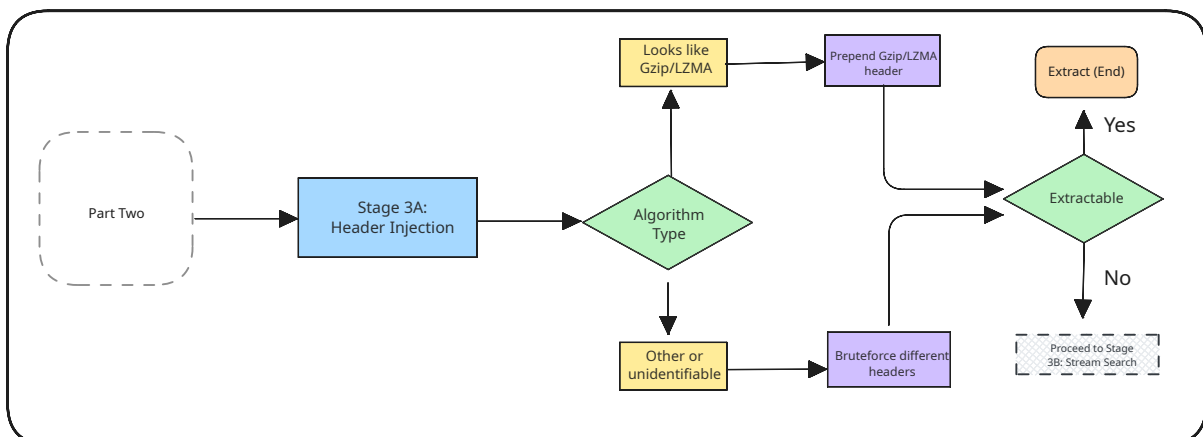


Figure 4.4: Workflow part 2A.

Our initial idea was to use streaming when header injection fails, but would later be our main method of extraction. The initial idea was to use the following steps to outline the approach:

- **Signature Search:** The system performs a broad scan to identify any internal markers or signatures indicating the start of a compressed stream.
- **Byte-by-Byte Analysis:** Once a potential signature is found, the system evaluates the data at a byte-level resolution.
- **Multi-Algorithm Filtering:** The system tests the data against a suite of raw compression families, including Raw Deflate, Raw LZMA, LZO, LZ4, and Zstd.

- **Bit-by-Bit Refinement:** For non-standard or shifted streams, the system performs a high-resolution bit-level scan to find the exact alignment of the payload. Since it is computational heavy and can provide additional false-positives, we made this an alternative.
- **Stream Validation:**
 - **If Yes (Valid Stream):** The data is extracted, and the process ends. As described earlier in limitations, as of yet we do not have the ability to conduct data validation but it is the subject of future improvement.
 - **If No:** The data is marked for Manual Forensics, as it may be truly incompressible or encrypted.

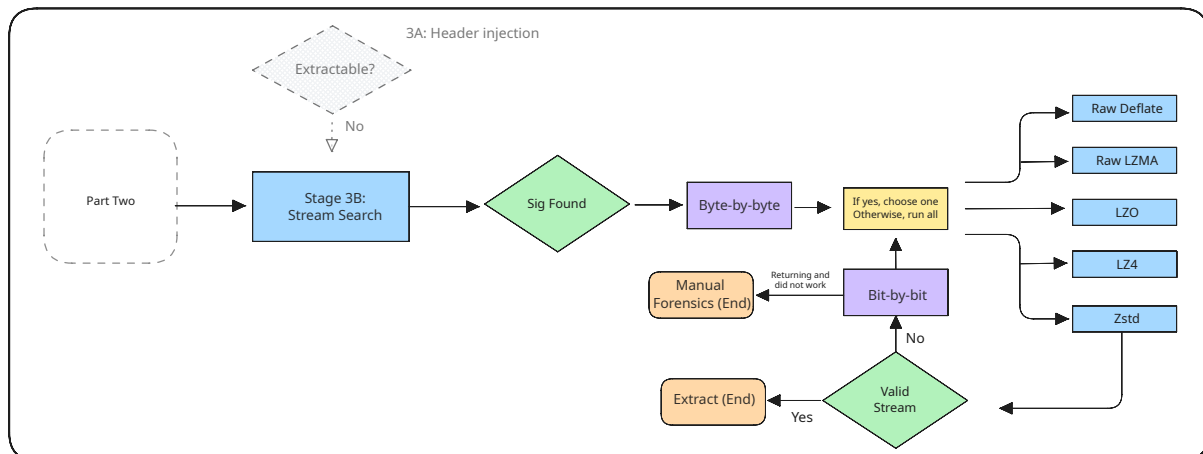


Figure 4.5: Workflow part 2B, which later proved to be our main extraction method

4.3 Implementation of the BinSift Framework

To address the extraction challenges associated with untagged and obfuscated binary data, the *BinSift* framework was developed. As illustrated in the project directory tree (see Figure 4.6), *BinSift* is a modular Python-based firmware analysis and extraction framework. Its core purpose is to systematically analyze binary data, detect embedded files or compressed streams, and extract them without prior knowledge of the file structure.

4.3.1 Framework Architecture and Orchestration

The framework operates as a multi-stage pipeline, managed entirely by the `Orchestrator` class located in `binsift/core/orchestrator.py`. Execution begins through the command-line interfaces provided by `main.py` and `run.py`, which parse user arguments (such as `input_file`, `output_dir`, and advanced flags like `enable_bitshifting`).

Once initialized, the `Orchestrator` guides the binary data through a sequential pipeline that moves from high-confidence structured formats down to low-confidence heuristics:

1. **Step 0: Container Scan** (External Tools)
2. **Step 1: Signature Scan** (Standard Payloads)
3. **Step 2: Entropy Analysis** (Statistical Profiling)
4. **Step 3: Brute-Force Extraction** (Raw Streams and Bit-shifting)

To prevent redundant extractions and infinite loops, the orchestrator tracks the offsets of successfully extracted data using a `masked_ranges` mechanism. Data contracts and communication between these stages are strictly defined using Python `@dataclass` structures located in `binsift/core/data_types.py`.

4.3.2 Analysis Engines

Before extraction can occur, the binary must be profiled. BinSift divides this responsibility between two distinct modules in the `binsift/analysis/` directory:

- **Signature Scanner (`signature.py`):** This module linearly scans the binary data for known magic numbers (e.g., `\x1f\x8b\x08` for GZIP). Matches found here are fed directly to the Standard Extractor.
- **Entropy Analyzer (`entropy.py`):** This module calculates the Shannon entropy on chunks of data, using a default block size of 1024 bytes. A high entropy score (typically ≥ 7.5 out of 8.0) strongly implies the presence of compressed or encrypted data, successfully flagging regions for the brute-force engine even when signatures are missing.

4.3.3 Extraction Engines

Depending on the results of the analysis engines, the Orchestrator deploys one of three specialized extraction modules located in `binsift/extractors/`:

Container Extraction (`containers.py`)

Executing Step 0 of the pipeline, the `ContainerExtractor` scans for complex filesystems such as SquashFS, CRAMFS, and UBI. Because interpreting these filesystems is computationally complex, this module relies on sub-processing external C-binaries (e.g., `unsquashfs`, `sasquatch`, `7z`). It carves the suspected regions into temporary files and delegates the extraction to these optimized tools, preserving the directory tree.

Standard Extraction (`standard.py`)

Executing Step 1, the `StandardExtractor` handles clean, known compression streams based on hints provided by the Signature Scanner. It utilizes native Python bindings (`zlib`, `lzma`, `bz2`, `zip`) to safely inflate the data and verify checksums.

Brute-Force Extraction (`brute_force.py`)

Acting as the fallback engine in Step 3, the `BruteForceExtractor` is deployed on high-entropy blocks that lack known signatures. It blindly attempts structured decompression and raw/headerless extraction (such as raw LZ4 blocks or raw Deflate streams).

4.3.4 Advanced Recovery Mechanisms

To tackle intentional obfuscation and undocumented firmware formats, BinSift implements three advanced heuristic mechanisms:

Sliding Window Scan

If an initial raw extraction attempt fails, the `BruteForceExtractor` employs a sliding window that scans byte-by-byte up to an offset of 256 bytes from the block origin. This allows the framework to detect and extract compressed streams that are buried deep inside firmware.

Bit-Shifting Recovery

In proprietary firmware architectures, data may be intentionally misaligned (shifted by 1 to 7 bits) to thwart standard forensic tools. If the `enable_bitshifting` flag is triggered, BinSift generates bit-shifted variants of the data block in memory and passes them iteratively through the brute-forcer, successfully recovering mangled data.

Stream Storm Heuristic

During orchestration, the framework monitors the frequency of extractions within a given spatial area. If an excessive number of valid streams are extracted in close proximity (e.g., 5 successful `zlib` extractions within a 1MB window), the system triggers a "Stream Storm" warning. This heuristic smartly detects the symptoms of an undocumented filesystem that is shredding its data chunks without recognizable headers, providing vital context to the reverse engineer.

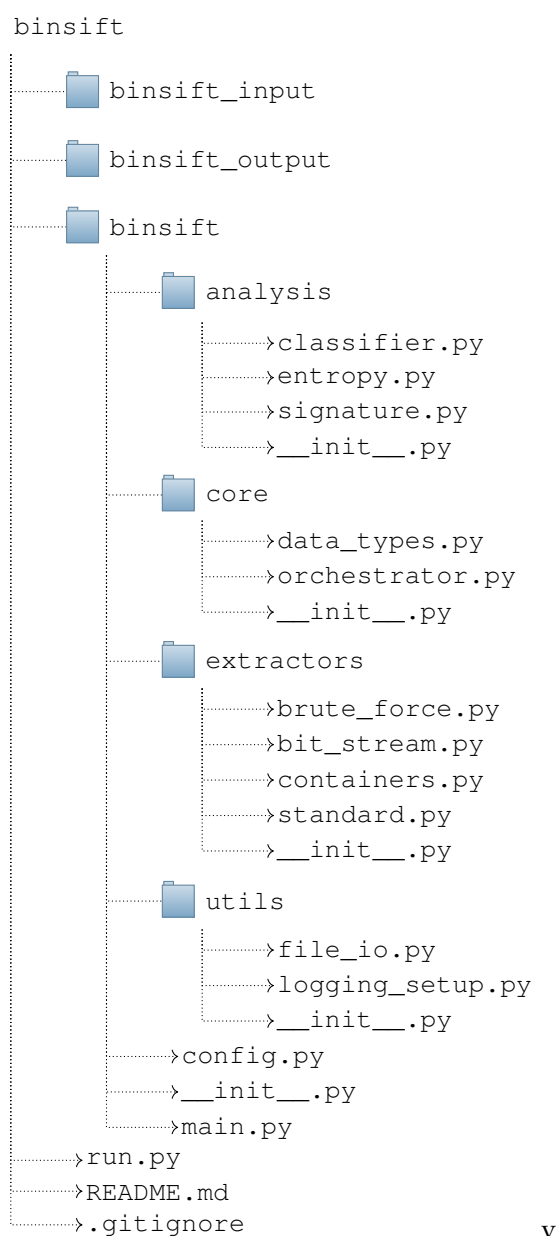


Figure 4.6: Project structure of the BinSift framework

4.4 Methodology: Large-Scale Entropy Profiling and Dataset Analysis

To establish empirical entropy thresholds for the `BinSift` tool, a large-scale statistical analysis was conducted on the *FirmSec* dataset. [58] This dataset comprises approximately 11,086 public and 23,050 private firmware images, totaling over 200 GB of heterogeneous binary data. The following section details the automated pipeline developed to process this data, extract signatures, and profile entropy distributions.

4.4.1 Data Acquisition and Extraction Pipeline

Given the scale of the dataset and the prevalence of nested archives (e.g., *.zip* and *.rar* containers), a robust “Extract-Process-Delete” pipeline was engineered to manage storage constraints and computational see figure 4.7.

- **Recursive Unpacking:** Archives were extracted into a volatile temporary directory. The pipeline utilized a recursive search to identify nested binaries, ensuring that firmware images buried within multiple layers of compression were processed.
- **Fault Tolerance and State Preservation:** To account for the multi-day processing time, a checkpointing system was implemented. A persistent log of processed archive hashes was maintained, allowing the pipeline to resume seamlessly following any hardware or software interruptions.
- **Atomic Data Logging:** Data markers were flushed to the output stream immediately following each file analysis, ensuring data integrity in the event of a terminal signal or system crash.

4.4.2 Signature Identification and Data Carving

To isolate compressed streams for analysis, the pipeline utilized a signature-based scanning layer. While `BinSift` is designed for signature-less detection, established signatures provided the “ground truth” necessary for calibration.

Standardized *Binwalk* instances were executed against the extracted binaries. To avoid inconsistencies in structured output formats across different environments, a regular expression (Regex) parser was developed to extract decimal offsets and metadata from the standard output. The parser targeted several specific compression algorithms: *LZMA*, *LZ4*, *XZ*, *Zlib*, *Gzip*, and *SquashFS* file systems.

4.4.3 Limitations of the Profiling Phase

A deliberate limitation of this data gathering phase is the exclusive reliance on signature-based identification. While the ultimate objective of `BinSift` is to detect headerless and non-standard compressed streams, the calibration of entropy thresholds requires a *confirmed positive* dataset.

Consequently, this profiling phase does not attempt to identify headerless compression. Instead, it utilizes established file signatures as an anchor to ensure that the entropy measurements are derived from mathematically verified compression streams. By focusing solely on signature-verified data, we minimize the risk of including uncompressed high-entropy noise (such as encrypted blocks or random padding) into the calibration metrics for *LZMA*, *LZ4*, and other targeted algorithms.

4.4.4 Statistical Noise Reduction

Preliminary testing revealed significant statistical noise introduced by non-firmware files (e.g., PDF documentation and text-based release notes) bundled within the archives. These files often triggered false positive *Zlib* signatures but exhibited entropy profiles irrelevant to binary reverse engineering. To mitigate this, a filename-based whitelist filter was applied. Entropy profiling was strictly limited to file extensions associated with raw binaries, such as *.bin*, *.img*, and *.elf*, or files with no extension, ensuring a clean dataset for the “Entropy Floor” calculation.

4.4.5 Sliding-Window Entropy Profiling

For every confirmed signature found, the pipeline performed a targeted carve of the first 64 KB of the stream. This data was then subjected to a sliding-window Shannon entropy analysis, defined by the following parameters:

$$H(X) = - \sum_{i=1}^n P(x_i) \log_2 P(x_i) \quad (4.1)$$

A window size of 256 bytes was selected to simulate the local detection sensitivity of the BinSift engine, with a step size of 32 bytes to ensure high granularity. For each identified stream, three primary metrics were recorded: the **Minimum Entropy** (the lowest dip within the window), the **Maximum Entropy**, and the **Mean Entropy**.

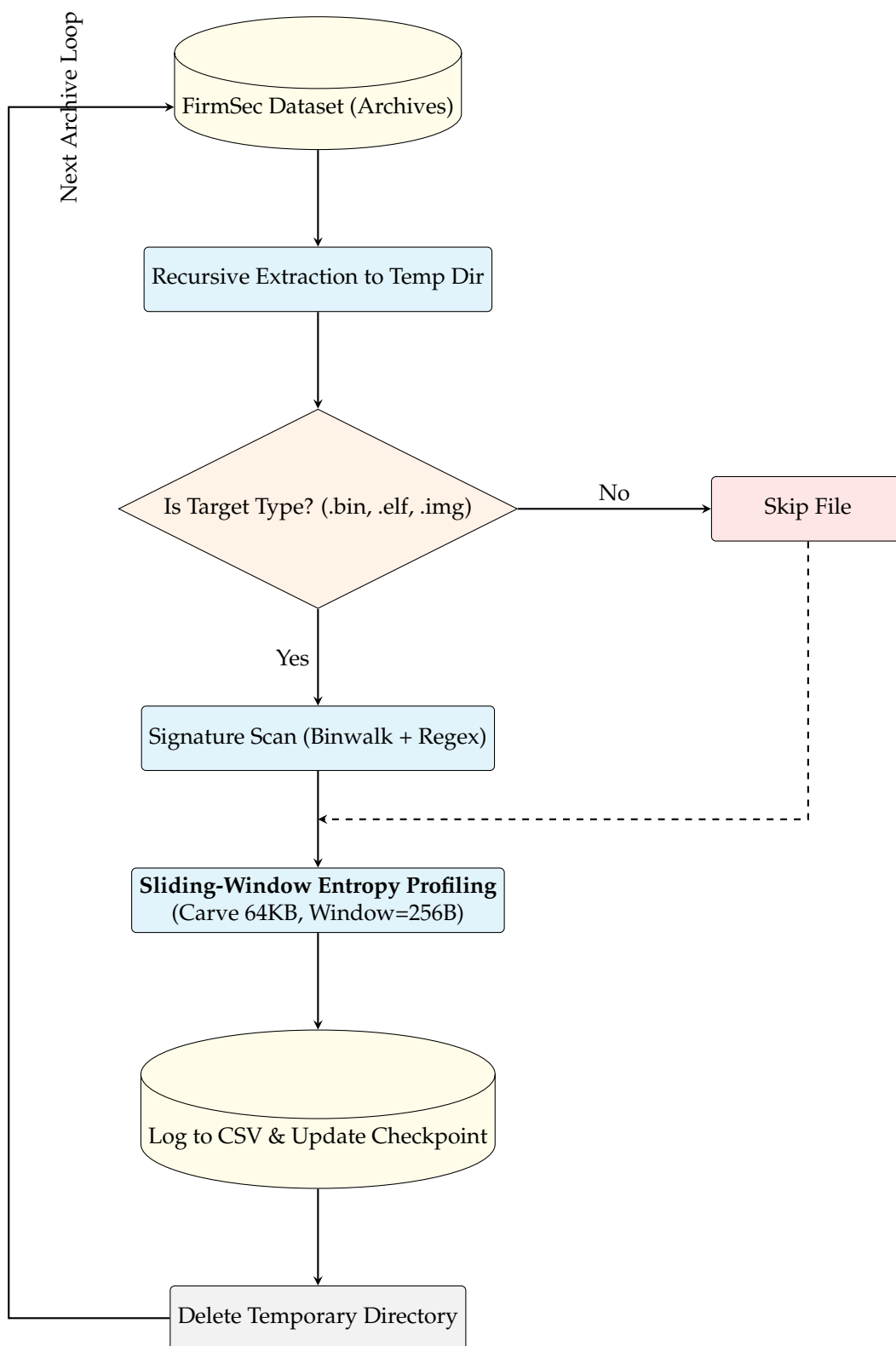


Figure 4.7: Visual representation of the automated “Extract-Process-Delete” data profiling pipeline used to analyze the FirmSec dataset.

4.4.6 Research Objectives of the Profiling Phase

The resulting dataset, stored in a comprehensive markers database, serves two primary research objectives:

1. **Algorithm Frequency Mapping:** Identifying which compression techniques are most prevalent in modern firmware to prioritize engine development.
2. **Threshold Calibration:** Determining the “Entropy Floor” for each algorithm—the minimum threshold required to trigger extraction without omitting valid but low-density compressed streams.

4.5 Extraction and Analysis Pipeline

4.5.1 BinSift Extraction Process

Machine code to readable C code: Single application in depth

When we run BinSift to evaluate the precision of BinSift, we performed an analysis on the `frankenstein.bin` file, the ground truth for which is established in Figure 4.1. In this test scenario, LZ4 extraction was manually disabled to verify the tool’s performance on the remaining compression formats.

```
Terminal
...
Saved 2048 bytes to binsift_output/frankenstein.bin/
  offset_0x00000800_raw_deflate_[MacBinary__inited__b]/extra
  cted.bin
Masked region 0x800-0xb74 (884 bytes, method=raw_deflate)
Raw format scan: raw_lzma at 0x21dc (consumed=1326, decompressed=2049)
Saved 2049 bytes to binsift_output/frankenstein.bin/offset_0x000021dc_raw_lzma_[
  ARM_Cortex_M_firmwar]/extracted.bin
Masked region 0x21dc-0x270a (1326 bytes, method=raw_lzma)
Step 2: Entropy Analysis (brute-force fallback)
High entropy block at 0x2000 (7.70). Trying brute force...
Analysis complete.
```

4.5.2 Automated ARM Cortex-M Architecture Identification

Because these raw binary files (`.bin`) lack standardized headers or magic bytes, identifying their architecture requires a deep analysis of their entry logic. Our tool utilizes a heuristic classification engine designed to identify firmware such as ARM Cortex-M firmware by reconstructing its Interrupt Vector Table.

1. The Identification Mechanism

Since raw firmware lacks metadata, the classifier assumes the initial bytes of the file represent the **Interrupt Vector Table**, a standardized hardware blueprint defined by ARM Holdings. This table serves as the primary map for the CPU during the boot sequence.

2. Byte Extraction and Endianness

Our tool processes the binary in 32-bit (4-byte) chunks. Given that ARM processors typically utilize **Little-Endian** formatting, the engine reverses the byte order of each chunk to reconstruct the true memory addresses. For example, the byte sequence `b1 51 00 08` is mathematically converted to the 32-bit hexadecimal address `0x080051b1`.

3. Heuristic Validation

To confirm the binary is indeed ARM Cortex-M firmware, our tool validates that the reconstructed addresses conform to standard physical microcontroller memory maps:

- **Slot 0 (Bytes 0–3):** Analyzed as the **Initial Stack Pointer (SP)**. The tool verifies this points to a valid SRAM address range (typically starting with `0x20...`).
- **Slot 1 (Bytes 4–7):** Analyzed as the **Reset Vector**. The tool verifies it points to a physical Flash memory address (typically `0x08...` or `0x00...`).

4. Thumb-Mode Bit Masking

In the ARM architecture, execution pointers such as the Reset, NMI, and HardFault vectors must have their Least Significant Bit (LSB) set to 1 to indicate **Thumb-mode** instruction execution. The tool validates these odd-numbered addresses and subsequently subtracts 1 to record the actual, aligned code address. For instance, the raw pointer `0x080051b1` is correctly logged in the final output as the aligned address `0x080051b0`.

5. Mapping the Vector Table

Once the binary is validated as a Cortex-M image, the tool iterates through the predefined, unchangeable hardware slots to extract the remaining metadata:

- **Slot 2 (Bytes 8–11):** NMI (Non-Maskable Interrupt)
- **Slot 3 (Bytes 12–15):** HardFault
- **Slot 11 (Bytes 44–47):** SVCcall
- **Slot 14 (Bytes 56–59):** PendSV

This extracted data is concatenated into the `file_type_description` string. This allows a reverse engineer to immediately identify where the boot sequence begins and where critical memory bounds lie without manual binary inspection.

4.5.3 Determining the Base Address via the BinSift Pipeline

The determination of the base address is a direct result of the `binsift` extraction process. When the `raw_lzma` chunk was extracted, the newly decompressed bytes were analyzed by `classifier.py`. This script utilizes a heuristic database to identify the underlying architecture of headerless binary data.

Heuristic Signature Analysis

The classifier inspects the first eight bytes of the decompressed file: `00 50 00 20 b1 51 00 08`. This specific sequence reveals a mathematical pattern unique to ARM Cortex-M microcontrollers:

- **Initial Stack Pointer (Bytes 0–3):** `0x20005000` points precisely to the standardized **SRAM** range (beginning with `0x20...`) for STM32 and similar chips.
- **Reset Vector (Bytes 4–7):** `0x080051b1` is an odd-numbered address, indicating Thumb-mode execution, and resides within the standard **Flash** memory range (beginning with `0x08...`).

Because the program's primary entry point was compiled to reside at an address starting with `0x08...`, it serves as a definitive indicator that the entire file must be mapped into Ghidra starting at a base address of `0x08000000`.

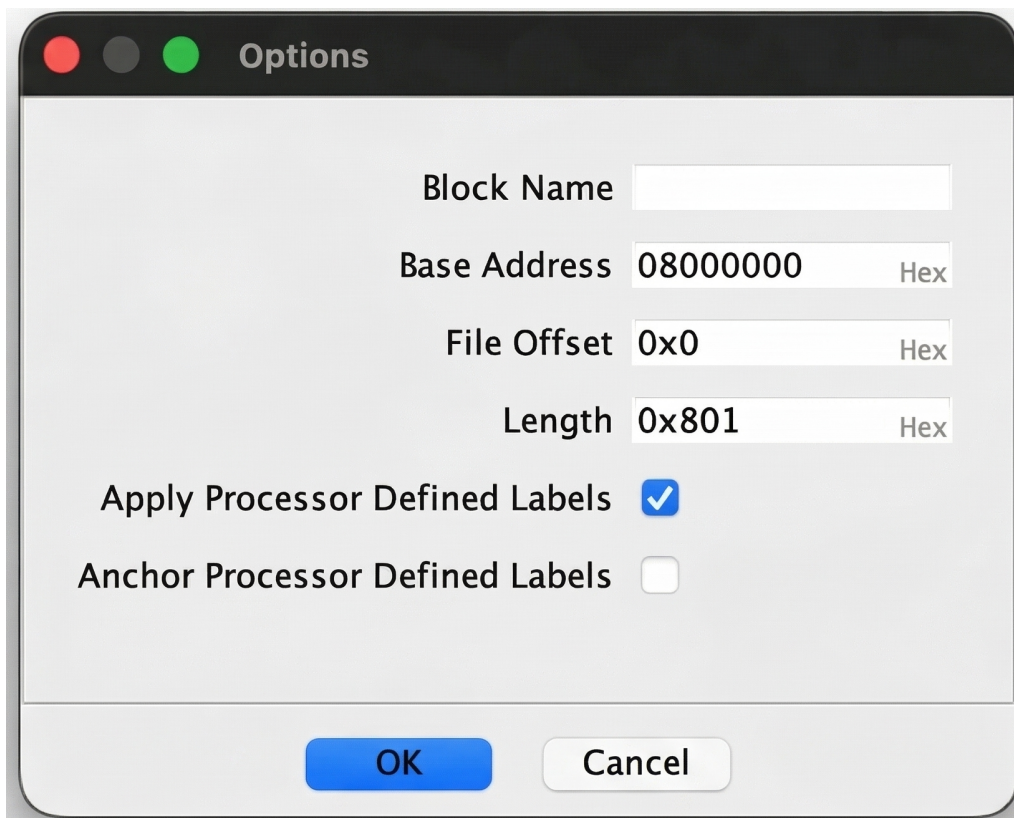


Figure 4.8: Ghidra Base Address Selection.

The Necessity of Base Addressing in Bare-Metal Analysis

Unlike software designed for high-level Operating Systems (Windows, Linux, macOS), which utilize "Virtual Memory" to load programs at arbitrary addresses, bare-metal microcontrollers operate on physical memory maps hardwired into the silicon.

In an ARM Cortex-M/STM32 environment, the CPU triggers literal physical wires connected to specific hardware components based on the address line:

Table 4.1: ARM Cortex-M Hardware Memory Map

Address Range	Component	Function
0x00000000 – 0x07FFFFFF	Boot Block	Factory ROM / Mirroring logic.
0x08000000 – 0x0FFFFFFF	Internal Flash	Permanent storage for firmware code.
0x20000000 – 0x3FFFFFFF	SRAM	Volatile working memory for the Stack.
0x40000000 – 0x5FFFFFFF	Peripherals	Hardware switches for LEDs, sensors, etc.

The Reset Vector: 0x080051b0

The Reset Vector serves as the specific instruction pointer where the CPU begins execution upon receiving power. During a cold boot, the CPU is hardwired to look at the first 8 bytes of the file to answer two questions:

1. **Where is my stack?** (Answer: 0x20005000)
2. **Where is the first line of code?** (Answer: 0x080051b0)

Summary

Setting the Base Address to `0x08000000` is mandatory because the compiler has baked `0x08XXXXXX` into every jump command and function call. If the file were mapped incorrectly, the CPU (and Ghidra's decompiler) would attempt to reference memory lines that do not physically exist in the silicon.

4.5.4 Reverse Engineering the Extracted Firmware using Ghidra

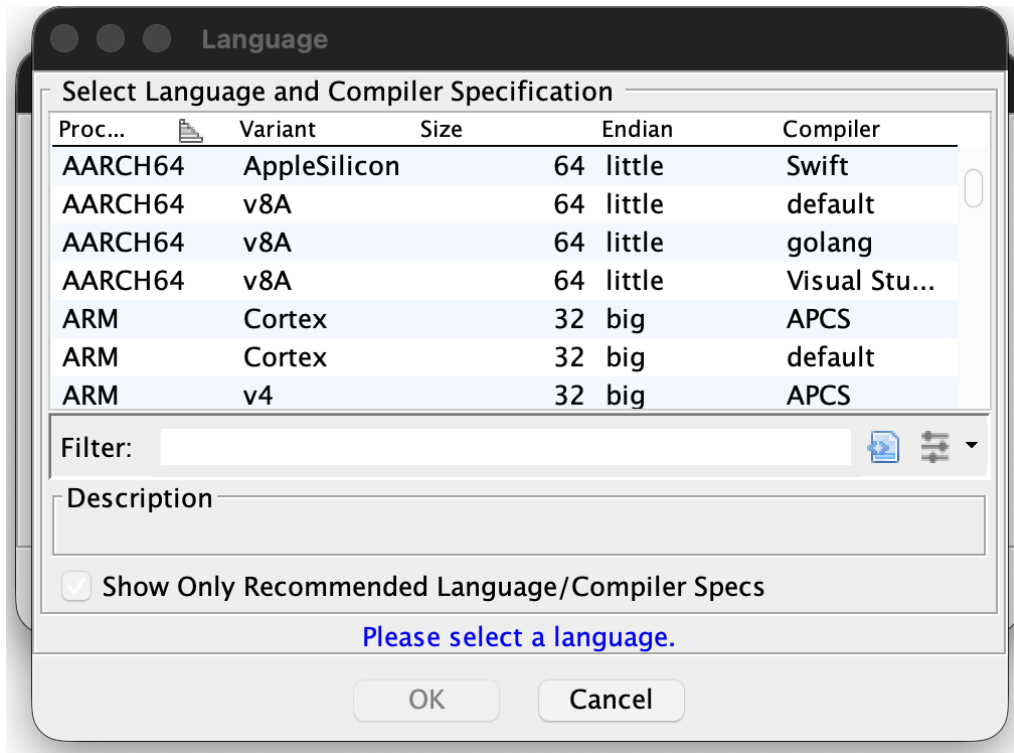


Figure 4.9: Ghidra Language and Compiler Specification dialog.

With the extraction process complete and the architecture validated as ARM Cortex-M, the binary blobs can be imported into Ghidra for static analysis. The “Language and Compiler Specification” dialog is the most critical step of the import process, as it defines how Ghidra’s **SLEIGH** engine will interpret the raw bytes.

Selecting the correct specification—in this case, **ARM Cortex, 32-bit, Little Endian**—provides Ghidra with the following necessary frameworks:

- **Instruction Set Architecture (ISA):** Ghidra maps the hex values to Thumb-mode mnemonics. Without the correct Cortex-M variant, the decompiler might misinterpret specific 16-bit instructions or fail to recognize extended ARMv7-M instructions.
- **Register Definitions:** This tells Ghidra that the processor uses specific registers like *R13* for the Stack Pointer (SP) and *R14* for the Link Register (LR). This is essential for the decompiler to reconstruct function calls and local variables.
- **Memory Map Defaults:** Ghidra uses this to define standard I/O and peripheral memory regions associated with the selected chip architecture.
- **Calling Convention:** By selecting the **APCS** (ARM Procedure Call Standard) or the default compiler spec, we inform Ghidra how parameters are passed between functions (typically via *R0–R3*).

Based on our previous analysis of the `info.json`, we know that the binary uses Little-Endian formatting and targets an ARM Cortex-M core. Correctly matching these settings ensures that the initial Stack Pointer at `0x20005000` and the Reset Vector at `0x080051b1` are treated as valid addresses rather than random data.

4.5.5 Auto-analysis settings

Standard auto-analysis in Ghidra typically employs a *Recursive Descent* approach, meaning it only disassembles code that is explicitly referenced by a known entry point, such as the Reset Vector. However, in raw, headerless firmware, this often leaves significant "dark" regions where functions exist but aren't yet linked to the main execution flow.

To achieve maximum code coverage, we enabled the **ARM Aggressive Instruction Finder (Prototype)**. This analyzer provides several critical advantages for ARM Cortex-M analysis:

- **Heuristic Function Discovery:** It performs a *Linear Sweep* of the binary, looking for common ARM/Thumb function prologues (e.g., `push {r4, r5, r7, lr}`). This allows it to identify "orphaned" functions that would otherwise be ignored as undefined data.
- **Thumb-2 Synchronization:** Because Cortex-M uses the Thumb-2 instruction set—which features a mix of 16-bit and 32-bit instructions—the disassembler can easily lose alignment. AGIF uses pattern matching to ensure the disassembler maintains the correct instruction boundaries.
- **Enhanced Decompiler Fidelity:** By forcing Ghidra to treat suspected data regions as code, the decompiler can reconstruct a more complete logic flow, reducing the number of `undefined` bytes that often break the structural analysis of complex subroutines.

By utilizing AGIF, we transition from a limited view of the boot sequence to a comprehensive map of the entire firmware image, ensuring that interrupt handlers and specialized subroutines are fully disassembled.

Transitioning from Methodology to Findings:

Having established our analytical framework—from the initial binary extraction and heuristic classification to precise base address calculation and aggressive static analysis configuration in Ghidra—we now apply these methods to evaluate the tool's efficacy. The following section details the concrete results yielded by this pipeline. We will explore the ground truth validation of the extracted components, the rich metadata generated during classification, and the fully decompiled logic revealing the firmware's core operations.

4.6 Automated Firmware Evaluation Pipeline

This section details the automated methodology used to evaluate the **BinSift Framework** 4.7 against the **Firmsec-dataset** 4.4. The evaluation was designed to test the standalone effectiveness of the program as a primary forensic tool, independent of external metadata or assistance. The outcomes of this pipeline, particularly regarding fidelity and recovery ratios, are analyzed in detail in Section 5.2.

4.6.1 Objective: The "True Blind" Baseline

Evaluating headerless (blind) extraction presents a unique challenge: modern industrial firmware is rarely a single continuous stream. Instead, it typically consists of multiple nested compressed components such as independent kernels, bootloaders, and filesystems—each with its own header.

To evaluate a real-world scenario, we implemented a comparative "Ground Truth" pipeline. By using the **Full Mode** (metadata-assisted) as a baseline, we established a gold standard for what data *should* be recoverable. This allowed us to quantify the "Fidelity Retention" of the **Blind Mode**, where the engine

is forced to find these multiple compressed parts relying solely on internal mathematical probability (refer to Table 5.2 for the resulting performance delta).

4.7 System Implementation and Experimental Logic

To evaluate the standalone efficacy of the BinSift engine, the system was implemented using a dual-mode comparative architecture. This design allows for a direct comparison between standard signature-based extraction and a “true blind” forensic recovery process, following the theoretical principles of entropy-based carving established in Section 3.5.

4.7.1 Full Mode: The Signature-Based Control Group

The *Full Mode* serves as the baseline control group. In this configuration, the system utilizes its internal **SignatureScanner** to identify magic byte sequences (e.g., `0xFD 0x37 0x7A` for XZ). This mode establishes the ground-truth baseline, defining the maximum volume of data that can be identified when all headers and signatures are visible and known.

4.7.2 Blind Mode: The Headerless Experimental Group

The *Blind Mode* is the core experimental setting designed to test the framework’s ability to recover data from obfuscated, proprietary, or corrupted binaries. In this mode, we explicitly enforce a “Handcuff” logic by disabling the **SignatureScanner** and the **StandardExtractor**, forcing the engine to rely on raw mathematical probes.

- **Entropy-Based Probing:** Instead of searching for headers, the engine reads the binary file in 128-byte segments (the *Stride*). When a sudden jump in local entropy is detected, the engine identifies the region as a potential compressed payload, utilizing the Shannon entropy principles defined in Equation 3.5.1.
- **Brute-Force Decompression:** Once a high-entropy region is identified, the engine attempts “Raw Decompression” using *Raw Deflate*, *Raw LZMA*, and *Raw LZ4*. This leverages the bit-by-bit recovery logic discussed in Section 3.7 to ensure that the engine can identify the start of a stream even if the magic bytes are missing or encrypted.

By enforcing these constraints, the Blind Mode proves the tool’s forensic efficacy: even if headers are encrypted or proprietary, the framework can still identify and extract the primary binary payloads purely by analyzing raw data patterns, as evidenced by the results in Figure 5.7.

4.8 Experimental Setup and Failure Analysis

This section outlines the hardware parameters and the specific diagnostic workflow used to analyze failures and validate the “False Failure” hypothesis.

4.8.1 Diagnostic Analysis of Extraction Failures

To ensure the integrity of the success metrics, a secondary diagnostic test is conducted on all samples that fail to yield data during the initial extraction baseline. This deep-dive is designed to distinguish between failures inherent to the carving engine’s current logic and those caused by mathematically unrecoverable data. The diagnostic workflow involves three key stages:

1. **Binary Isolation:** The original binaries from failed extraction attempts are isolated for manual and automated structural inspection.
2. **Entropy Analysis:** Global structural entropy (H) is calculated for each failed binary. This metric serves as the primary indicator for the statistical state of the data (e.g., compressed vs. encrypted), based on the mathematical thresholds discussed in Section 3.5.

3. Categorization of Results:

- **Encrypted (False Failures):** Binaries with high entropy (typically $H > 7.9$) are categorized as encrypted. These are identified as “False Failures” because the lack of recognizable statistical distributions makes them mathematically unrecoverable by any signature-less carving tool without appropriate decryption keys.
- **Complex/Obfuscated:** Binaries with entropy levels below this threshold ($H < 7.9$) are categorized as complex. These represent true challenges for the BinSift engine, where payload data is potentially present but requires more advanced heuristic or stateful analysis for successful recovery.

4.8.2 Forensic Probing Parameters

We configured the engine with a **128-Byte Stride**, probing the binary stream every 128 bytes. This stride was chosen to provide a balance between:

1. **Thoroughness:** Ensuring that the starting offsets of small, nested kernels or filesystem parts are not skipped.
2. **Efficiency:** Maintaining the throughput levels required to process the dataset within the 9-hour time window described in the results of Table 5.3.

4.8.3 Validation Metrics

Success is measured by the *Recovery Ratio*, defined as the amount of successfully extracted data compared to the original binary size. By performing a 1:1 comparative join between the Full Mode (Ground Truth) and the Blind Mode, we can quantify exactly how much data a forensic investigator can expect to recover when signatures are unavailable.

5

Result

5.1 Entropy Distribution Analysis

To determine a data-driven threshold for identifying compressed or encrypted firmware components, we analyzed the average Shannon entropy across a dataset of over 34,136 firmware files. Utilizing the mass profiling script detailed in Appendix A, we successfully carved and validated 43,259 individual file segments which can be seen in figure 5.2. As illustrated in Figure 5.4, the Kernel Density Estimate (KDE) displays the average entropy categorized by the compression algorithm identified by `binwalk`. The composition and specific counts of these 43,259 files are further broken down by algorithm in Figure 5.2.

5.1.1 Average Entropy By Compression

The statistical distribution of average entropy across the dataset reveals the efficiency and consistency of various compression algorithms used in the firmware samples. As shown in Figure 5.1, the dataset is characterized by high-entropy signatures, typically clustering between 7.10 and 7.18 bits per byte.

- **LZMA and XZ Consistency:** Both LZMA and XZ demonstrate remarkably tight Interquartile Ranges (IQR) with medians positioned near 7.17 bits per byte. This indicates a highly consistent compression density across thousands of different firmware files, confirming these algorithms as the industry standard for high-density storage in modern embedded systems.
- **GZIP Variability:** GZIP exhibits the highest degree of variance in the dataset. While its median remains high at approximately 7.12, the extended lower whisker reaching down to nearly 6.92 suggests that GZIP-compressed streams often contain less dense internal structures or varying compression levels compared to modern LZMA/XZ counterparts.
- **ZLIB and SQUASHFS Profiles:** ZLIB shows a distinct profile with a lower median entropy of approximately 7.13 and a relatively wide distribution, reflecting its application across a diverse range of file types. Conversely, SQUASHFS displays a very stable and high entropy profile, nearly mirroring the consistency of LZMA. This is expected, as SQUASHFS often utilizes LZMA or XZ as its underlying compression engine.
- **LZ4 Performance:** Despite a smaller sample size within the dataset see figure 5.2, LZ4 maintains a very high and stable entropy median, sitting tightly at approximately 7.17.

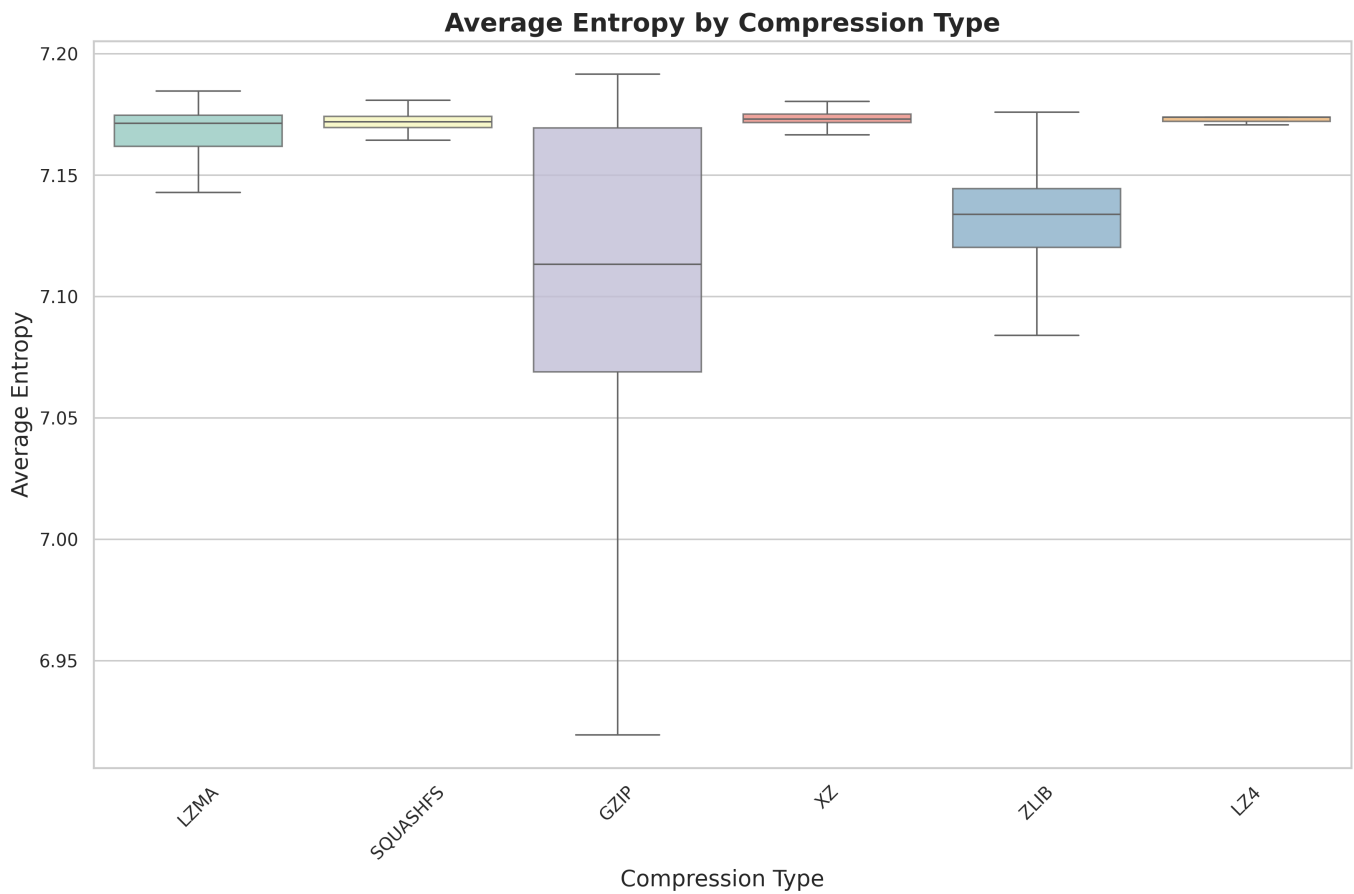


Figure 5.1: View of entropy distributions.

5.1.2 Density Analysis of Minimum Entropy

To evaluate the reliability of localized entropy drops, we generated a Kernel Density Estimate (KDE) for the minimum entropy observed within each analyzed block. As illustrated in Figure 5.3, the minimum entropy metric exhibits significantly higher variance and a distinct downward shift in its distribution peaks compared to average entropy metrics.

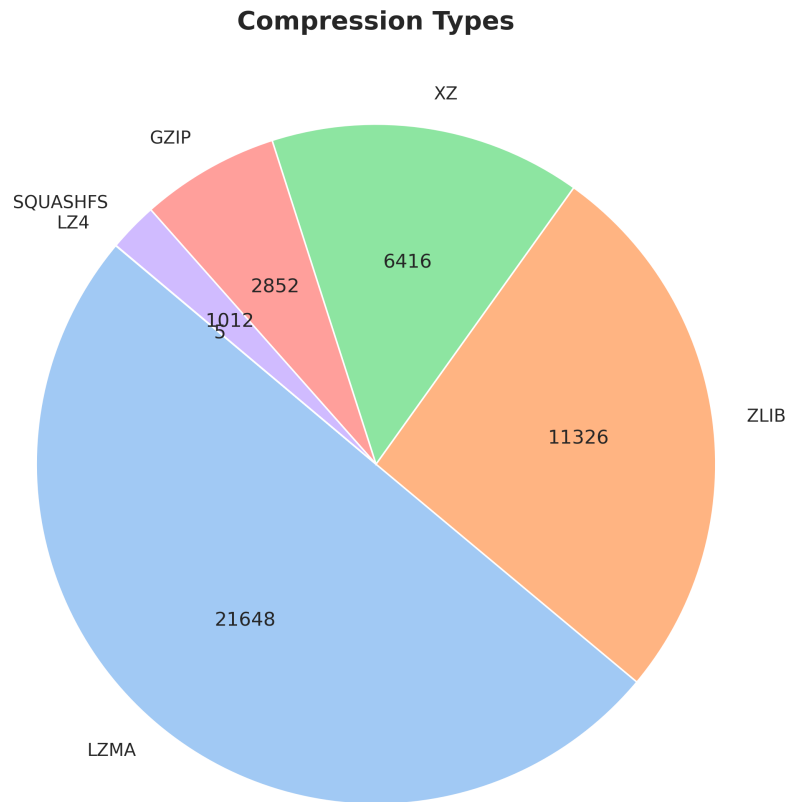


Figure 5.2: Distribution and absolute count of identified compression types.

Compression Type	Total Identified Blocks
LZMA	21,648
ZLIB	11,326
XZ	6,416
GZIP	2,852
SQUASHFS	1,012
LZ4	5
Total	43,259

Table 5.1: Numerical Distribution of Verified Compression Types.

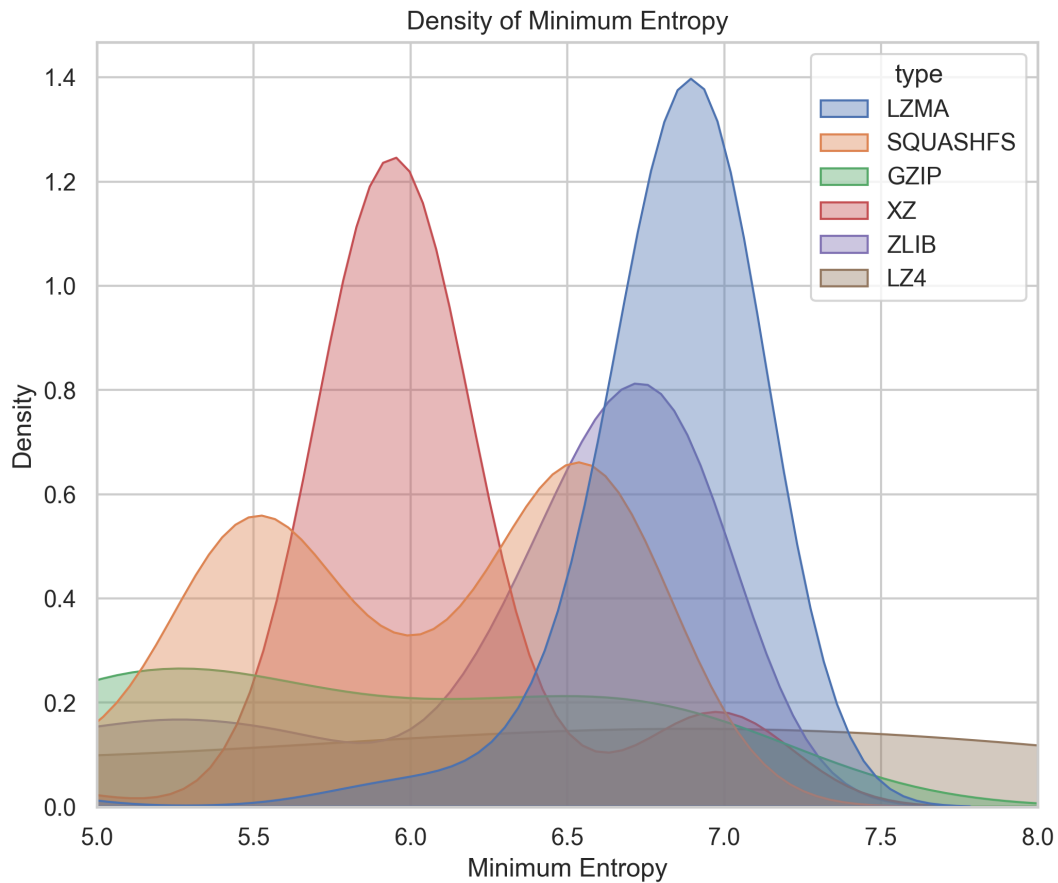


Figure 5.3: Kernel Density Estimation (KDE) of Minimum Entropy

High Variance and Multi-modal Distributions

Unlike the sharp, singular peaks observed in average entropy analysis, the minimum entropy distributions are characterized by broad, often multi-modal profiles. This is particularly evident in the LZMA (blue) and XZ (red) categories, where the primary peaks have shifted to approximately 6.8 and 5.9 bits, respectively. The increased spread across the x-axis suggests that minimum entropy is an unstable predictor for identifying compressed streams in firmware.

Sensitivity to Structural Artifacts

The significant shift toward lower entropy values is primarily attributed to the sensitivity of the minimum metric to local "dips" in data density. In firmware binaries, these dips frequently occur due to:

- **Alignment Padding:** Large sequences of null bytes (0×00) or $0\times FF$ used to align data structures to memory boundaries.
- **Protocol Headers:** Uncompressed metadata tables or headers embedded within otherwise high-entropy compressed streams.
- **Small-Window Noise:** Because the calculation captures the lowest entropy found in any 256-byte window, a single block of padding can redefine the statistical profile of a 64KB "signal" block as "noise."

Impact on Threshold Reliability

The high degree of overlap between different compression types in the 5.0 to 6.5 range makes the minimum entropy metric unsuitable for establishing a firm detection threshold. Utilizing this metric would

likely result in a high rate of false negatives, as valid compressed files would be classified as uncompressed data due to these localized artifacts. Consequently, while minimum entropy provides insight into the presence of internal structures, it is a poor candidate for automated classification compared to average entropy.

5.1.3 Average Entropy Characteristics

The distribution reveals a high degree of clustering for modern compression formats. Specifically, formats such as LZMA, XZ, and ZLIB exhibit narrow peaks concentrated between 7.0 and 7.5. This indicates that these algorithms produce a highly consistent statistical profile regardless of the underlying data being compressed.

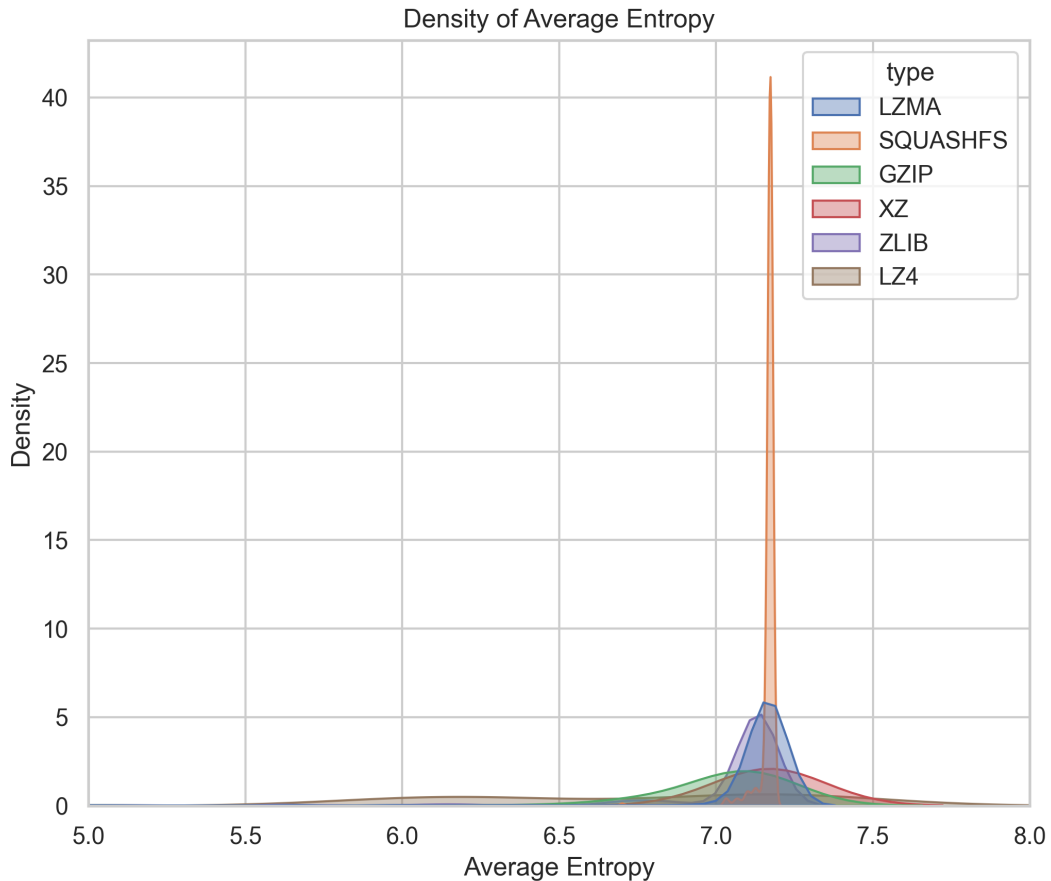


Figure 5.4: Kernel Density Estimation (KDE) of Average Entropy

The most notable feature of Figure 5.4 is the extreme density spike associated with the SQUASHFS (orange) category, which reaches a peak density exceeding 40. This phenomenon can be attributed to several factors:

- **Algorithmic Uniformity:** SquashFS is a structured, read-only filesystem that often utilizes standardized compression parameters (e.g., fixed 64KB block sizes and XZ compression). This uniformity results in an extremely narrow entropy range, centered at approximately 7.18.
- **Sample Frequency:** Given the prevalence of SquashFS in modern embedded Linux distributions (routers, IP cameras), the sheer volume of identical blocks across thousands of archives causes the KDE to converge on this specific value.

- **Statistical Confidence:** Mathematically, because the area under a KDE curve must equal 1, the extreme verticality of the SquashFS peak suggests that the average entropy of this format is a near-perfect predictor of its presence within this dataset.

In contrast, GZIP (green) and LZ4 (brown) show a broader "tail" extending toward lower entropy values (< 6.5). This suggests that these formats are either used for smaller data segments where headers significantly influence the average entropy, or they are utilized in configurations with lower compression ratios.

5.1.4 Threshold Implications

The overlap between categories becomes negligible above an entropy value of 7.0. Below this point, the density of known compressed formats drops significantly, while the density of uncompressed binary code (the "other" category) typically increases. Based on the average entropy distribution, a minimum threshold of 7.1 appears to be the optimal balance for capturing the majority of compressed filesystem signatures while minimizing false positives from high-density uncompressed data.

5.1.5 Metric Sensitivity and Local Minima

The disparity between the average and minimum entropy metrics arises from the sensitivity of the latter to local fluctuations in data density. In firmware analysis, "dips" in entropy are rarely indicative of the primary payload; rather, they typically represent protocol headers, alignment padding (such as null-byte strings), or metadata tables embedded within an otherwise compressed stream.

By utilizing the minimum entropy value, a single 256-byte window of "noise" can effectively redefine the statistical profile of an entire 64KB block of "signal." This high sensitivity to localized, low-entropy structures explains why the Minimum Entropy distributions appear significantly more chaotic and dispersed than the Average Entropy metrics, which provide a more representative measure of the overall data density.

5.1.6 Cumulative Distribution of Minimum Entropy

The Cumulative Distribution Function (CDF) for minimum entropy, as illustrated in Figure 5.5, provides a statistical overview of how localized entropy "dips" are distributed across various compression formats. This metric is particularly useful for identifying the lower bounds of data density within 64KB carved blocks.

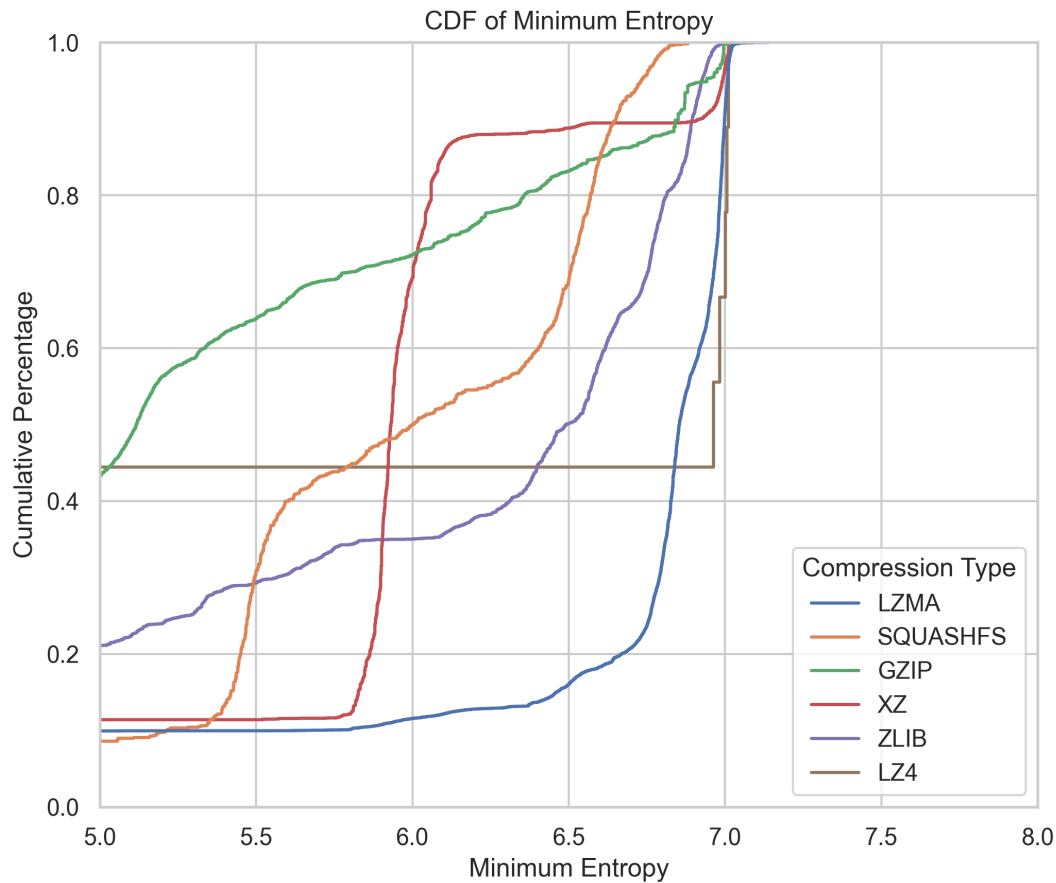


Figure 5.5: Cumulative Distribution Function (CDF) of Minimum Entropy

The distribution profiles for minimum entropy demonstrate significant instability across different algorithms:

- **GZIP and LZ4 Sensitivity:** The GZIP (green) and LZ4 (brown) curves show an immediate and gradual ascent starting from an entropy of 5.0. Specifically, the GZIP curve indicates that approximately 60% of the samples contain a local window with entropy lower than 6.0.
- **Step Functions in LZ4 and XZ:** The LZ4 curve displays a prominent horizontal "plateau" between entropy values of 5.0 and 7.0. This indicates a binary behavior where many samples either contain extreme low-entropy padding or high-entropy data, with very few instances falling in between. Similarly, XZ (red) shows a sharp vertical "step" at approximately 5.9.
- **Threshold Unreliability:** The slow, non-vertical climb of these curves suggests that using a minimum entropy threshold would lead to significant data loss. For instance, setting a threshold at 6.5 would exclude nearly 80% of GZIP samples and 50% of SquashFS samples, as their localized "minima" fall below this line.

5.1.7 Cumulative Distribution of Average Entropy

In contrast to the minimum entropy results, the CDF of average entropy (Figure 5.6) presents a much more robust and stable metric for defining a universal threshold. This visualization confirms that while local windows may experience entropy drops, the overall average of a 64KB block remains consistently high for compressed payloads.

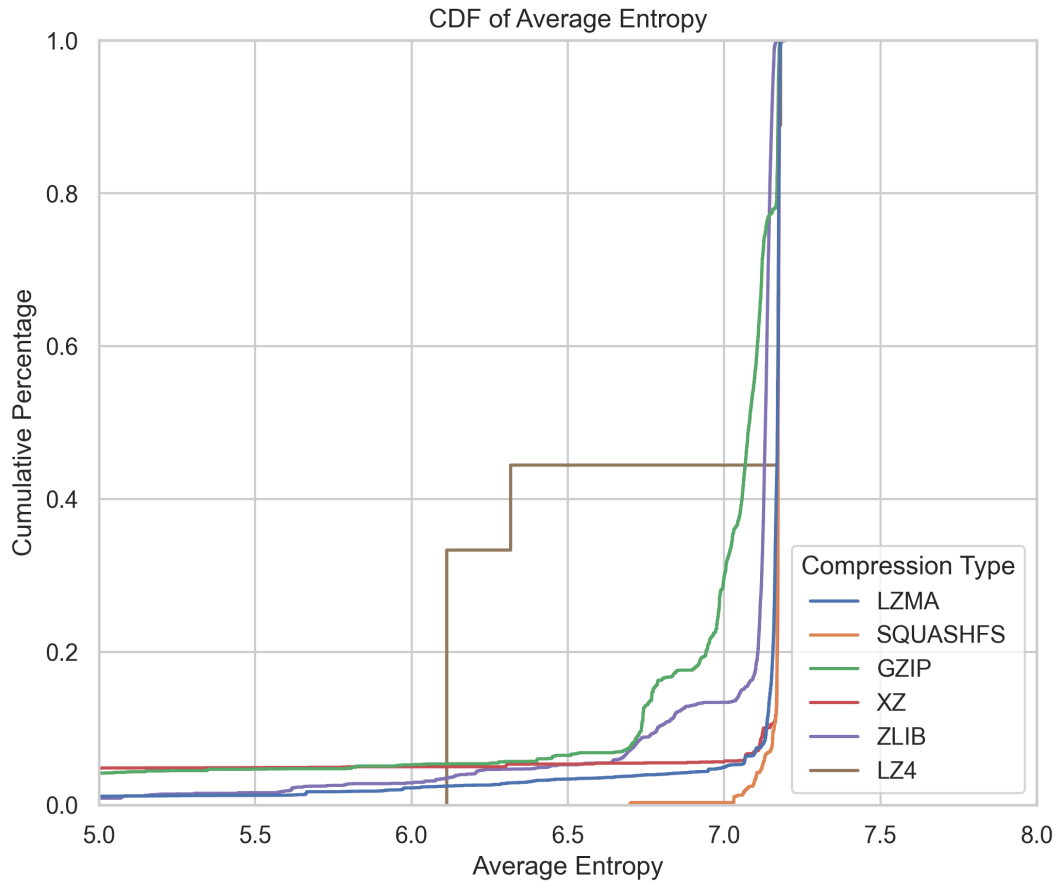


Figure 5.6: Cumulative Distribution Function (CDF) of Average Entropy

The average entropy CDF is characterized by its sharp, "wall-like" verticality:

- **High-Efficiency Convergence:** The curves for LZMA (blue), SquashFS (orange), and XZ (red) stay near zero until they reach the 7.1 to 7.2 range, where they ascend nearly vertically to 1.0. This signifies that the vast majority of these files have an average entropy clustered in an extremely narrow band.
- **Quantifiable Capture Rates:** The vertical nature of the average entropy CDF allows for precise threshold selection. At a value of 7.1, the CDF shows that nearly 95% of all modern compression types (SQUASHFS, LZMA, XZ) are captured.
- **Stability Over Noise:** Unlike Figure 5.5, the average entropy CDF effectively filters out the "dips" caused by padding and headers. The lack of early-stage growth in these curves (below 6.5) suggests that average entropy is highly resistant to the structural noise that plagues the minimum entropy metric.

5.2 Results and Performance Evaluation

This section presents a quantitative evaluation of the BinSift framework across 105 industrial firmware samples, categorized into IP cameras ($n = 49$) and routers ($n = 56$). We evaluate the performance of the *Full Mode* against the *Blind Mode* to quantify the framework's efficacy in metadata-absent scenarios, as defined in the methodology in Section 4.6.

5.2.1 Extraction Success Rate and Fidelity

The comparative performance metrics, summarized in Table 5.2, demonstrate the robustness of the dual-mode approach described in Section 4.7. The signature-based *Full Mode* achieved an overall success rate of 72.4% (76/105). Notably, the *Blind Mode* maintained a 59.0% (62/105) success rate without any prior knowledge of file headers. This indicates a “Fidelity Retention” of 81.5% compared to the metadata-assisted baseline, proving that entropy-based stream identification (Section 4.7) is a viable fallback for standard IoT firmware payloads.

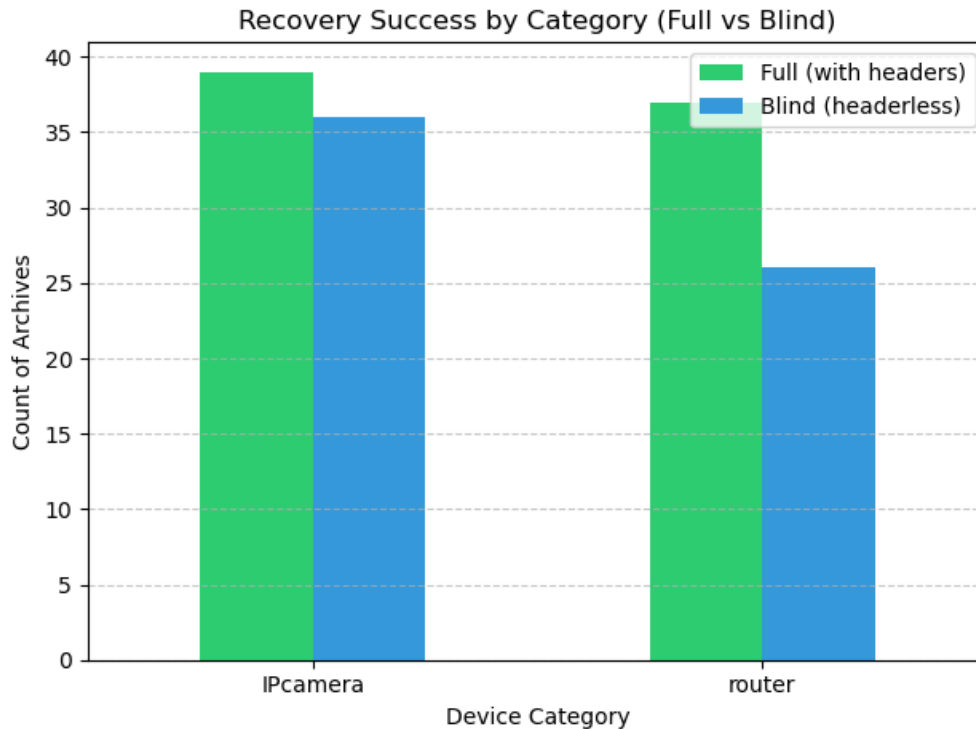


Figure 5.7: Comparison of Extraction Success Rates between Full Mode and Blind Mode across device categories.

5.2.2 Forensic Failure Mode Analysis

To distinguish between architectural limitations and hard cryptographic barriers, we performed a post-mortem Shannon entropy (H) analysis on the 29 samples that failed extraction in Full Mode. This audit, following the diagnostic workflow established in Section 4.8 and supported by the cumulative distribution data in Figure 5.6, revealed two primary failure signatures:

1. **Encryption-Gated Failures (51.7%):** 15 samples exhibited a near-perfect global entropy ($H > 7.99$). As established by our theoretical entropy baseline (Section 3.5), while high-efficiency compression clusters around $H \approx 7.2$, any value exceeding $H > 7.8$ represents encrypted data. These represent a hard forensic ceiling for non-keyed recovery.
2. **Complexity-Gated Failures (48.3%):** 14 samples exhibited high but structured entropy ($H < 7.8$). These represent legitimate architectural gaps where proprietary formats bypassed the engine’s probes.

5.2.3 Effective Success Rate: Accounting for Encryption

By applying the $H > 7.8$ threshold identified in the diagnostic failure analysis (Section 4.8), we can distinguish between recoverable and unrecoverable data. Removing the 15 encrypted “False Failures” from the dataset reveals an **Effective Success Rate of 84.4%** in Full Mode (see Table 5.2). This confirms

that BinSift's core logic is highly effective at capturing structured compressed streams, and that a significant portion of remaining failures are due to industrial-grade encryption rather than a failure of the carving engine itself.

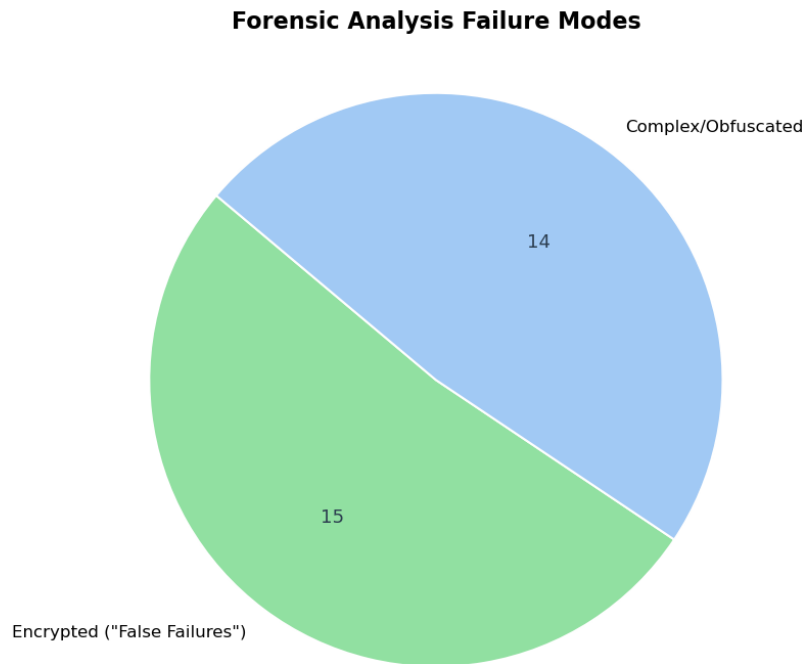


Figure 5.8: Post analysis of failure modes ($n = 29$), categorizing samples into encryption "False Failures" and Complex/Obfuscated.

5.2.4 Forensic Efficiency and Dataset Volume

The framework processed a total raw dataset volume of 910.9 MB. As detailed in Table 5.3, the recovery process generated 1,051.6 MB of data in Full Mode compared to 317.4 MB in Blind Mode.

By leveraging the parallel processing on a 12-core system described in Section 4.8, the total execution time was 540 minutes (9 hours). As shown in Table 5.3, Blind Mode analysis required significantly more time than Full Mode due to the computational intensity of raw stream probing. The efficiency metric confirms that Blind recovery requires approximately 25.3 seconds of processing per MB, compared to 11.4 seconds for Full Mode. This overhead is a direct consequence of the brute-force alignment logic implemented in Section 4.7 to bypass missing headers.

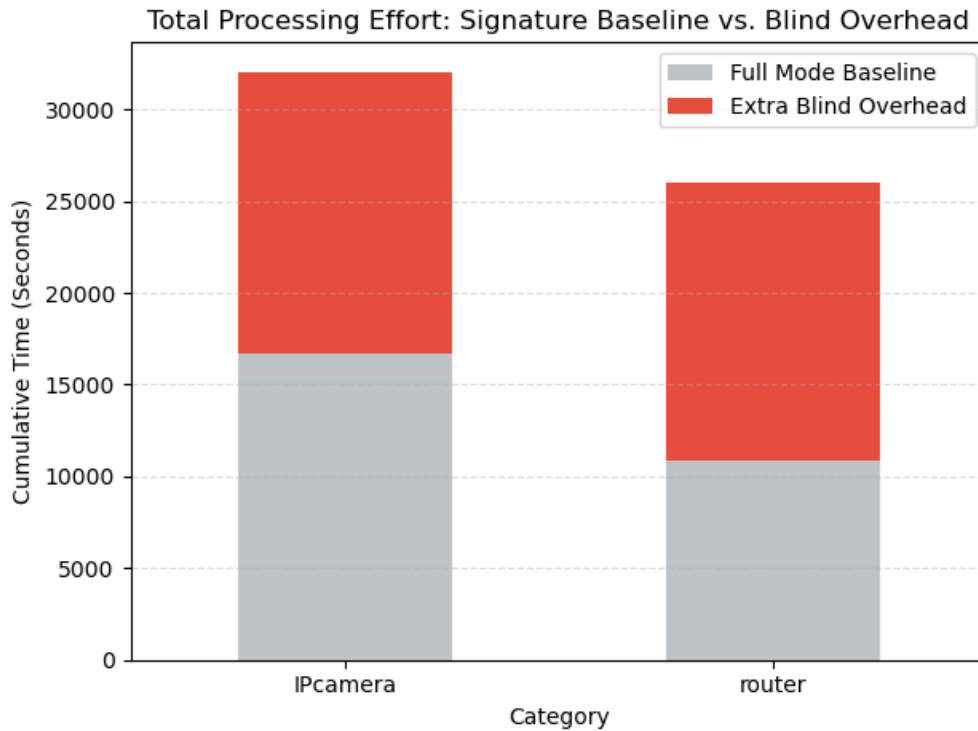


Figure 5.9: Distribution of processing time by category, showing the baseline Full Mode duration and the overhead introduced by Blind Mode analysis.

Table 5.2: Consolidated Performance Metrics: Full Mode vs. Blind Mode

Category	Samples	Success Rate (%)			Recovery Ratio		Time (min)	
		Full	Blind	Fidelity Retention*	Full	Blind	Full	Blind
IPcamera	49	79.6%	73.5%	92.3%	0.817	0.605	103.9	201.0
Router	56	66.1%	46.4%	70.2%	1.576	0.112	67.9	163.0
Raw Total	105	72.4%	59.0%	81.5%	1.196	0.370	171.8	364.0
Effective Total**	90	84.4%	68.9%	81.6%	1.395	0.431	-	-

*Fidelity Retention represents the Blind Mode success rate relative to the Full Mode baseline.

**Excludes 15 encrypted "False Failures" identified by entropy analysis ($H > 7.99$).

Table 5.3: Forensic Volume and Processing Efficiency Summary

Metric	Full Mode	Blind Mode
Total Raw Data	910.9 MB	868.2 MB
Total Recovered	1,051.6 MB	317.4 MB
Total Execution Time	173.8 min (\approx 2.9 h)	366.2 min (\approx 6.1 h)
Forensic Efficiency (s/MB)	11.4 s/MB	25.3 s/MB
Total Test Duration	540.0 min (9.0 h)	

5.3 Extraction Outcomes and Static Analysis Findings

5.3.1 Ground Truth Validation

The accuracy of the tool is confirmed by comparing the detected offsets and sizes against the ground truth configuration. As noted in Table 5.4, the tool achieved bit-perfect identification for both the start offsets and the total stream lengths.

Table 5.4: Comparison of Ground Truth (Figure 4.1) vs. BinSift Detection

Block Name	Ground Truth		BinSift Result	
	Offset (Hex)	Size (Bytes)	Offset (Hex)	Size (Bytes)
Raw_Deflate_2	0x800	884	0x800	884
Raw_LZMA_6	0x21DC	1326	0x21DC	1326

The tool correctly mapped the start of `Raw_Deflate_2` at decimal 2048 (0x800) and the start of `Raw_LZMA_6` at decimal 8668 (0x21DC). Despite the high entropy fallback triggers observed in the logs, the signature-based scanning correctly masked these regions without overlap, providing a high-fidelity reconstruction of the original firmware components.

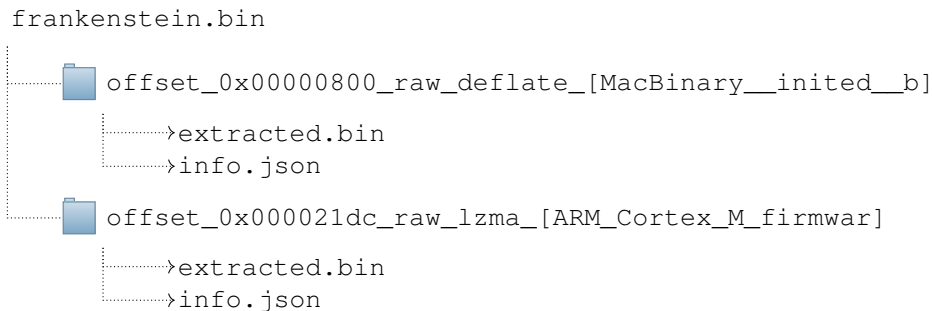


Figure 5.10: Output file structure after processing frankenstein.bin.

5.3.2 Metadata Extraction and Firmware Classification

Upon successful extraction of the `raw_lzma` stream, the tool generates an `info.json` file. This file contains cryptographic hashes, extraction offsets, and a detailed heuristic architectural analysis.

Extracted Metadata (info.json)

The following JSON snippet demonstrates the output for the identified ARM Cortex-M firmware segment:

```

{
  "offset": 8668,
  "offset_hex": "0x21dc",
  "method": "raw_lzma",
  "size": 2049,
  "sha256": "df84a0605c6da4e01bd301b5fbc94dbb4ea67a2439d562725f314532b371768e",
  "file_type_description": "ARM Cortex-M firmware, initial SP at 0x20005000,
    reset at 0x080051b0, NMI at 0x08003634,
    HardFault at 0x08003636, SVCall at 0x0800363e,
    PendSV at 0x08003642",
  "file_type_mime": "application/octet-stream"
}
  
```

Listing 5.1: Contents of info.json for the raw_lzma stream

5.3.3 Static Analysis: From Reset Vector to Application Superloop

By navigating to the Reset Vector address identified in the previous heuristic analysis (0x080051b0), we can observe how Ghidra's decompiler utilizes the **SLEIGH** engine and our language selection to translate raw machine instructions into readable C code.

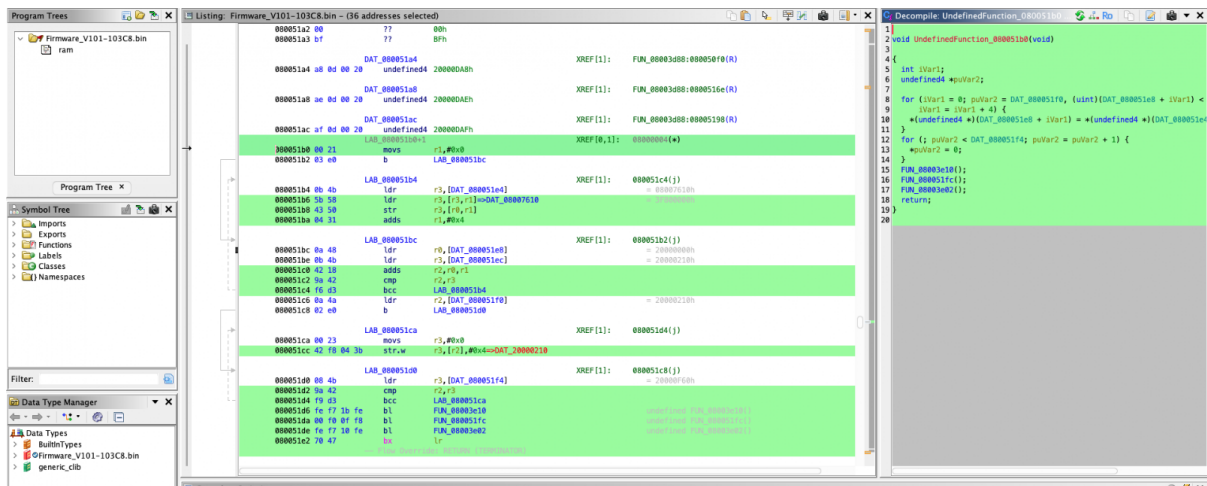


Figure 5.11: Ghidra UI showing the synchronized Assembly Listing (center) and the Decompiled C code (right) at the Reset Vector.

The Challenge of Stripped Binaries

Upon first inspection, the code is filled with generic labels such as `FUN_080051b0` or `DAT_080051f0`. This is because we are analyzing a **Stripped Binary**.

During compilation, names of functions and variables exist only for the developer's benefit. To minimize the footprint in the limited Flash memory of a microcontroller, the compiler removes these "symbols," as the CPU only requires the physical memory addresses to execute logic. Consequently:

- `FUN_[Address]` represents a function located at that specific memory offset.
- `DAT_[Address]` represents a global data variable or constant.
- `param_[N]` represents the n^{th} parameter passed to a function (typically via registers R0–R3).

The goal of our reverse engineering is to showcase how one can "re-label" these points of interest by analyzing their logical behavior.

Analyzing the Boot Sequence (Reset Vector)

The function located at `0x080051b0` is the hardware's "Entry Point." Before the main application can run, the environment must be initialized through two standard C-runtime loops:

```
void UndefinedFunction_080051b0(void)
{
    int iVar1;
    undefined4 *puVar2;

    // Loop 1: Initializing the .data section
    for (iVar1 = 0; puVar2 = DAT_080051f0, (uint)(DAT_080051e8 + iVar1) <
        DAT_080051ec;
        iVar1 = iVar1 + 4) {
        *(undefined4 *) (DAT_080051e8 + iVar1) = *(undefined4 *) (DAT_080051e4 + iVar1);
    }

    // Loop 2: Zeroing the .bss section
    for (; puVar2 < DAT_080051f4; puVar2 = puVar2 + 1) {
        *puVar2 = 0;
    }
}
```

```

FUN_08003e10(); // SystemInit()
FUN_080051fc(); // LibC Setup
FUN_08003e02(); // main() entry
return;
}

```

Listing 5.2: Decompiled Reset Vector logic (C-Runtime Initialization)

Data Copying and Memory Scrubbing

1. **Copying .data:** The first loop copies initialized global variables (e.g., a hardcoded constant like `int gravity = 9;`) from the permanent Flash memory (0x08...) into the fast SRAM (0x20...).
2. **Zeroing .bss:** The second loop identifies uninitialized global variables. According to C standards, these must be set to zero. The loop “scrubs” a specific segment of SRAM to ensure a clean state.

Identifying the Main Superloop

Following the initialization, the code calls `FUN_08003e02`. In embedded systems, the `main()` function is typically characterized by a “Superloop”, an infinite loop that prevents the CPU from reaching the end of the code.

```

void FUN_08003e02(void)
{
    FUN_08003de8(); // Peripheral hardware initialization
    do {
        FUN_08003d88(); // The Main Superloop
    } while( true );
}

```

Listing 5.3: Identification of the main() function and Superloop

The Task Scheduler and Protocol Parsing

Inside the superloop (`FUN_08003d88`), we uncovered two of the most significant patterns in flight controller firmware: a primitive task scheduler and a serial communication state machine.

1. **Software Task Scheduling** The firmware uses timing logic to simulate multitasking. By calling a function (likely `millis()` or `micros()`), the code checks how much time has elapsed to determine which tasks to run:

```

iVar5 = thunk_FUN_0800193c(); // millis()
if (1 < (uint)(iVar5 - *DAT_08003ddc)) { ... } // 1ms task (e.g., Gyroscope)
if (2 < (uint)(iVar5 - *DAT_08003de0)) { ... } // 2ms task (e.g., PID loop)
if (100 < (uint)(iVar5 - *DAT_08003de4)) { ... } // 100ms task (e.g., LED blink)

```

2. **MultiWii Serial Protocol (MSP) State Machine** The most critical discovery was a `switch` statement that parses incoming serial data byte-by-byte. By analyzing the expected hex characters, we identified the protocol:

- **Case 0:** Checks for 0x24 (ASCII: \$)
- **Case 1:** Checks for 0x4d (ASCII: M)
- **Case 2:** Checks for 0x3c (ASCII: <)

The sequence `$M<` is the signature for the MultiWii Serial Protocol (MSP), a standard used by open-source drone platforms like Cleanflight and Betaflight to receive commands from a remote controller. This confirms that `FUN_08003d88` is the “brain” of the flight controller, responsible for processing telemetry and pilot inputs.



6

Conclusion

The development and evaluation of the **BinSift** framework demonstrate that while automated, metadata-less decompression of binary blobs is achievable, it remains an iterative challenge. This thesis investigated the feasibility of using statistical analysis to bypass the need for file headers in firmware forensics. By shifting the focus from "Magic Numbers" to the mathematical properties of the data stream, this research established a new baseline for forensic recovery in obfuscated environments.

Answers to Research Questions

RQ1: To what extent can average Shannon entropy alone be used to reliably isolate firmware data streams in the absence of 'Magic Numbers,' and how does this metric perform when attempting to distinguish between varying compressions (LZMA, Deflate), encryption, and high-density machine code?

Analysis of the FirmSec dataset indicates that average Shannon entropy is a robust and stable predictor for detecting compressed streams. A threshold of **7.1 bits per byte** was found to capture nearly 95% of modern compression types. However, distinguishing between compressed data, encrypted payloads, and randomized machine code remains difficult because these distributions frequently mimic one another. While byte-frequency patterns allow for mapping signatures to decompression families like LZMA (the most prevalent) and LZ4 (rare), the reliance on these signatures requires highly granular analysis to overcome the lack of "Magic Numbers."

RQ2: How can a modular framework be designed to facilitate "blind" decompression of untagged binary blobs, and what mechanisms are required to address the technical challenges of bit-level boundary precision and stream-based alignment without metadata assistance?

The BinSift framework was designed as a modular Python-based solution for "blind" decompression. The research identified that the primary technical obstacle in this design is

boundary precision. Identifying the exact starting bit and termination point is critical; without headers, a single-bit misalignment can render a multi-megabyte payload undecodable. Ultimately, the results suggest that a streaming-based, "global" extraction method is superior to localized header-injection fixes for handling these alignment edge cases.

RQ3: What is the forensic effectiveness of an entropy-based extraction approach, measured by success rate and "Fidelity Retention" against metadata-assisted baselines, when evaluated against large-scale real-world firmware datasets containing both recoverable compressed payloads and unrecoverable encrypted segments?

The system achieved a "True Blind" success rate of **59.0%**, representing an **81.5% fidelity retention** compared to metadata-assisted baselines. When excluding mathematically unrecoverable encrypted payloads, the effective success rate rose to **84.4%**. However, a critical performance gap was identified in extraction depth: while the Full Mode achieved a **Recovery Ratio of 1.196**, the Blind Mode reached only **0.370** (Table 5.2). This indicates that while entropy-based probing is highly effective at *finding* the start of a stream, the lack of metadata makes it difficult to maintain stream continuity, often leading to fragmented or incomplete payloads.

Limitations and Future Work

A primary limitation identified is the reliance on a predefined architecture database; if a target's architecture is not documented, BinSift defaults to a generic "*data*" label. This underscores that identifying the target architecture is the primary prerequisite for the reverse engineering process. While BinSift simplifies initial discovery, human intervention is still required to utilize tools like Ghidra to transform raw machine code into readable code.

Future improvements should focus on refining the streaming approach to handle diverse edge cases and automate semantic verification to mitigate false positives—incidental high-entropy noise that currently requires manual masking. The accuracy of the tool remains highly dependent on the nature of the dataset, suggesting that further refinement is necessary to move toward a truly autonomous forensic solution.



Bibliography

- [1] Sunha Ahn and Sharad Malik. “Automated firmware testing using firmware-hardware interaction patterns”. In: *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. 2014, pp. 1–10.
- [2] Marian Balakowicz. *U-boot new uImage source file format (bindings definition)*. GitHub Repository. Accessed via the lentinj/u-boot mirror. 2008. URL: https://github.com/lentinj/u-boot/blob/master/doc/uImage.FIT/source_file_format.txt.
- [3] Brian Benchoff. *Why Blobs Are Important, And Why You Should Care*. <https://hackaday.com/2021/01/29/why-blobs-are-important-and-why-you-should-care/>. Accessed: 2026-03-05. 2021.
- [4] Guy E. Blelloch. *Introduction to Data Compression*. Draft chapter of “Algorithms in the Real World”, Carnegie Mellon University. Jan. 2013. URL: blellochcs.cmu.edu.
- [5] Ralf D Brown. “Improved recovery and reconstruction of deflated files”. In: *Digital Investigation* 10 (2013), S21–S29.
- [6] Brian Carrier. *File system forensic analysis*. Addison-Wesley Professional, 2005.
- [7] Fotios Chantzis, Ioannis Stais, Paulino Calderon, Evangelos Deirmentzoglou, and Beau Woods. *Practical IoT hacking: the definitive guide to attacking the internet of things*. no starch press, 2021.
- [8] Michael I Cohen. “Advanced carving techniques”. In: *Digital Investigation* 4.3-4 (2007), pp. 119–128.
- [9] Yann Collet. *LZ4 Frame Format Description*. https://github.com/lz4/lz4/blob/dev/doc/lz4_Frame_format.md. Accessed: 2026-03-05. 2022.
- [10] Yann Collet. *LZ4: Extremely Fast Compression Algorithm*. <https://github.com/lz4/lz4>. Accessed: 2023-10-25. 2011.
- [11] Yann Collet. *xxHash - Extremely fast non-cryptographic hash algorithm*. <https://github.com/Cyan4973/xxHash>. Accessed: 2026-03-05. 2023.
- [12] Lasse Collin. *The .xz File Format*. <https://tukaani.org/xz/xz-file-format.txt>. The technical specification for XZ stream headers and magic sequences. 2009.
- [13] Lasse Collin. *The .xz File Format Specification*. Version 1.0.4. Accessed: 2026-02-10. 2009. URL: <https://tukaani.org/xz/xz-file-format.txt>.
- [14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. “A large-scale analysis of the security of embedded firmwares”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association. 2014, pp. 95–110.

-
- [15] Vincent Dary. *Open-Firmware-Dataset-Builder*. Version main branch. Oct. 2024. URL: <https://github.com/VincentDary/open-firmware-dataset-builder>.
- [16] Fabio De Gaspari, Dorjan Hitaj, Giulio Pagnotta, Lorenzo De Carli, and Luigi V Mancini. "Reliable detection of compressed and encrypted data". In: *Neural Computing and Applications* 34 (2022), pp. 20379–20393.
- [17] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. IETF, 1996. URL: <https://datatracker.ietf.org/doc/html/rfc1951>.
- [18] P. Deutsch. *GZIP file format specification version 4.3*. RFC 1952. IETF, 1996. URL: <https://datatracker.ietf.org/doc/html/rfc1952>.
- [19] G. S. Gaba, A. Sari, I. Butun, P. Singh, A. Gurtov, and M. Liyanage. "A Survey on Security and Privacy of Industry 4.0 and Beyond: Technical Aspects, Use Cases, Challenges, and Research Directions". In: *IEEE Open Journal of the Communications Society* (2025).
- [20] Simson L. Garfinkel. "Carving contiguous and fragmented files with fast object validation". In: *Digital Investigation* 4 (2007). Foundational paper on the limits of signature-based carving and the move toward content-aware recovery., pp. 2–12.
- [21] A. Gurtov, J. Kenaudekar, S. Khan, and F. Souza. "Deep Learning Based Anomaly Detection for Securing ADS-B in NextGen Aviation". In: *Proc. of IEEE Aerospace*. Mar. 2026.
- [22] Scott Hand, Zhiqiang Lin, Guofei Gu, and Bhavani Thuraisingham. "Bin-Carver: Automatic Recovery of Binary Executable Files". In: *Preprint submitted to Elsevier* (May 2012).
- [23] Craig Heffner and ReFirmLabs. *Binwalk: Firmware Analysis Tool*. <https://github.com/ReFirmLabs/binwalk>. Version 3.0. 2024.
- [24] David A Huffman. "A method for the construction of minimum-redundancy codes". In: *Proceedings of the IRE* 40.9 (2007), pp. 1098–1101.
- [25] *ISO/IEC/IEEE 24765:2017(E) Systems and software engineering — Vocabulary*. International Organization for Standardization, 2017. URL: <https://www.iso.org/standard/71952.html>.
- [26] Thorsten Jenke, Elmar Padilla, and Lilli Bruckschen. "Towards Generic Malware Unpacking: A Comprehensive Study on the Unpacking Behavior of Malicious Run-Time Packers". In: *Nordic Conference on Secure IT Systems*. Springer. 2023, pp. 245–262.
- [27] Guhyeon Jeong, Euijin Choo, Joosuk Lee, Munkhbayar Bat-Erdene, and Heejo Lee. "Generic unpacking using entropy analysis". In: *2010 5th International Conference on Malicious and Unwanted Software*. IEEE. 2010, pp. 98–105.
- [28] S. Khan, G. S. Gaba, A. Gurtov, L. Jansen, N. Maurer, and C. Schmitt. "Post Quantum Secure Handover Mechanism for Next Generation Aviation Communication Networks". In: *IEEE Transactions on Green Communications and Networking* (2024).
- [29] Annick Lesne. "Shannon entropy: a rigorous notion at the crossroads between probability, information theory, dynamical systems and statistical physics". In: *Mathematical Structures in Computer Science* 24.3 (2014), e240311. DOI: 10.1017/S0960129512000783.
- [30] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.
- [31] TP-Link Technologies Co., Ltd. *Firmware Release Notes for Archer C60(EU) V3 (Release 201231)*. Official manufacturer documentation detailing WPA2/AES encryption updates and parental control bug fixes. 2022. URL: <https://www.tp-link.com/en/support/download/archer-c60/v3/>.
- [32] Linux Kernel Organization. *Squashfs 4.0 Filesystem Documentation*. Technical overview of how the Squashfs driver handles on-the-fly decompression and page cache management. 2024. URL: <https://docs.kernel.org/filesystems/squashfs.html>.
- [33] LVFS Project. *Linux Vendor Firmware Service: MetaInfo File Documentation*. Official documentation for firmware metadata specification. Linux Vendor Firmware Service. 2024. URL: <https://lvfs.readthedocs.io/en/latest/metainfo.html>.

- [34] Robert Lyda and James Hamrock. "Using entropy analysis to find encrypted and packed malware". In: *IEEE Security & Privacy* 5.2 (2007), pp. 40–45.
- [35] MIPS Technologies, Inc. *MIPS32® Architecture For Programmers Volume III: The MIPS32® Privileged Resource Architecture*. Revision 2.50. Document Number: MD00090. MIPS Technologies, Inc. Mountain View, CA, July 2005. URL: <https://courses.cs.vt.edu/cs2506/Fall12014/MIPSDocs/MD00090-2B-MIPS32PRA-AFP-02.50.pdf>.
- [36] Nithin Nagaraj and Karthi Balasubramanian. "Three perspectives on complexity: Entropy, compression, subsymmetry". In: *The European Physical Journal Special Topics* 226.15 (2017), pp. 3251–3272.
- [37] National Security Agency. *Ghidra*. Version 11.0. Open-source reverse engineering framework. National Security Agency, 2019. URL: <https://ghidra-sre.org/>.
- [38] Tammy Noergaard. *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Newnes, 2012.
- [39] ONEEYE. *unblob: An Accurate, Fast, and Easily Extensible Framework to Extract Files from Any Binary Blob*. Standard tool for automated firmware extraction and container identification. 2023. URL: <https://github.com/onekey-sec/unblob>.
- [40] ONEKEY. *unblob: The Extract-Everything Suite*. <https://github.com/onekey-sec/unblob>. Documentation available at <https://unblob.org/>. 2024.
- [41] OpenWrt Project. *TP-Link Archer C60 v3 Technical Data*. https://openwrt.org/toh/hwdata/tp-link/tp-link_archer_c60_v3. Technical specifications for the MIPS-based Atheros chipset, flash layout, and Squashfs file system. 2024.
- [42] Savan Oswal, Anjali Singh, and Kirthi Kumari. "Deflate compression algorithm". In: *International Journal of Engineering Research and General Science* 4.1 (2016), pp. 430–436.
- [43] OWASP Foundation. *OWASP Firmware Security Testing Methodology*. The industry-standard guide for firmware extraction and the transition from raw binary to static filesystem analysis. 2023. URL: <https://github.com/scriptingxss/owasp-fstm>.
- [44] Bora Park, Antonio Savoldi, Paolo Gubian, Jungheum Park, Seok Hee Lee, and Sangjin Lee. "Data Extraction from Damaged Compressed Files for Computer Forensic Purposes". In: *International Journal of Hybrid Information Technology* 1.4 (2008), pp. 89–102.
- [45] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.
- [46] Milad Rahmati and Nima Rahmati. "Firmware-level reverse engineering and AI-augmented deobfuscation for IoT malware detection in embedded systems". In: *Journal of Computer Virology and Hacking Techniques* 22.1 (2026), p. 5.
- [47] David Salomon. "Data compression". In: *Handbook of massive data sets*. Springer, 2002, pp. 245–309.
- [48] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.
- [49] Purna Chandra Sethi. "File Carving: Analyzing Data Retrieval in Digital Forensics". In: *International Journal of Creative Research Thoughts (IJCRT)* 12.4 (2024), pp. 555–564.
- [50] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "Firmallice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware". In: *NDSS*. Vol. 15. 2015, pp. 1–15.
- [51] The Linux Documentation Project. *SquashFS HOWTO*. Revision 1.9. Available through The Linux Documentation Project (TLDP). July 2008. URL: http://tldp.org/HOWTO/html_single/SquashFS-HOWTO/.
- [52] The Linux Kernel Organization. *Linux Kernel Source Tree: LZ4 Decompression API*. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/lib/lz4>. Accessed: 2026-03-05. 2023.
- [53] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. Standard specification for portable object file formats. TIS Committee. May 1995.

-
- [54] Jonathan W. Valvano. *Embedded Systems: Introduction to ARM Cortex-M Microcontrollers*. 5th. Vol. 1. CreateSpace Independent Publishing Platform, 2014.
- [55] Elecia White. *Making embedded systems*. " O'Reilly Media, Inc.", 2024.
- [56] Ian H Witten, Radford M Neal, and John G Cleary. "Arithmetic coding for data compression". In: *Communications of the ACM* 30.6 (1987), pp. 520–540.
- [57] Joseph Yiu. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*. 3rd. Newnes, 2013.
- [58] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, and Raheem Beyah. "A Large-Scale Empirical Analysis of the Vulnerabilities Introduced by Third-Party Components in IoT Firmware". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 442–454. ISBN: 9781450393799. DOI: 10 . 1145 / 3533767 . 3534366. URL: <https://doi.org/10.1145/3533767.3534366>.

**A**

Mass Firmware Profiler Script

The following Python script was used to perform mass entropy profiling on the firmware dataset. It utilizes multiprocessing for parallel execution, incorporates a 60-second timeout for `binwalk` and `7z` extraction to prevent stalls.

```
1 import os
2 import csv
3 import math
4 import subprocess
5 import zipfile
6 import shutil
7 import tempfile
8 import re
9 import multiprocessing as mp
10 from collections import Counter
11
12 # --- CONFIGURATION ---
13 DATASET_DIR = "./firmsec_dataset"
14 OUTPUT_CSV = "entropy_markers.csv"
15 CHECKPOINT_FILE = "processed_archives.txt"
16 ERROR_LOG_FILE = "errors.log"
17 CARVE_SIZE = 65536
18 WINDOW_SIZE = 256
19 STEP_SIZE = 32
20
21 TEMP_EXTRACT_DIR = "./temp_extraction" # Temporary workspace
22
23 # Ensure temp dir exists
24 os.makedirs(TEMP_EXTRACT_DIR, exist_ok=True)
25
26 def calculate_shannon_entropy(data):
27     if not data: return 0
28     total_len = len(data)
29     counts = Counter(data)
30     entropy = 0
31     for count in counts.values():
32         p_x = count / total_len
33         entropy += - p_x * math.log2(p_x)
34     return entropy
35
36 def get_sliding_window_stats(data, window_size, step):
37     if len(data) < window_size:
38         ent = calculate_shannon_entropy(data)
39         return ent, ent, ent
```

```

40 min_ent, max_ent, total_ent, count = 8.0, 0.0, 0.0, 0
41 for i in range(0, len(data) - window_size + 1, step):
42     window = data[i : i + window_size]
43     ent = calculate_shannon_entropy(window)
44     if ent < min_ent: min_ent = ent
45     if ent > max_ent: max_ent = ent
46     total_ent += ent
47     count += 1
48     return min_ent, max_ent, (total_ent / count if count > 0 else 0)
49
50 # Global Lock for thread-safe output writing
51 write_lock = None
52
53 def init_worker(lock):
54     global write_lock
55     write_lock = lock
56
57 def log_error(archive_path, message):
58     if write_lock:
59         with write_lock:
60             with open(ERROR_LOG_FILE, 'a') as f:
61                 f.write(f"[{archive_path}] {message}\n")
62
63 def scan_with_binwalk(filepath, original_archive):
64     found_streams = []
65     try:
66         # Run standard binwalk without -c (CSV mode), added timeout
67         result = subprocess.run(['binwalk', filepath],
68                                 capture_output=True, text=True, check=True, timeout=60)
69
70         pattern = re.compile(r'^(\d+)\s+(0x[0-9a-fA-F]+\s+(.*)')
71
72         for line in result.stdout.splitlines():
73             line = line.strip()
74             match = pattern.match(line)
75             if match:
76                 offset = int(match.group(1))
77                 desc = match.group(3).lower()
78                 label = None
79                 for k in ['lzma', 'lz4', 'xz', 'zlib', 'gzip', 'squashfs']:
80                     if k in desc:
81                         label = k.upper()
82                         break
83                 if label: found_streams.append((offset, label))
84     except subprocess.TimeoutExpired:
85         log_error(original_archive, f"Binwalk timeout on {filepath}")
86     except subprocess.CalledProcessError as e:
87         log_error(original_archive, f"Binwalk error on {filepath}: process failed")
88     except Exception as e:
89         log_error(original_archive, f"Binwalk error on {filepath}: {str(e)}")
90     return found_streams
91
92 def get_category_from_path(path):
93     lower_path = path.lower()
94     if 'ipcamera' in lower_path: return 'IPcamera'
95     if 'router' in lower_path: return 'router'
96     if 'switch' in lower_path: return 'switch'
97     return 'other'
98
99 def process_binary(file_path, archive_name, internal_file, category):
100     """Core analysis logic moved to a helper to handle extracted files."""
101     ext = os.path.splitext(internal_file)[1].lower()
102     if ext not in ['.bin', '.img', '.elf', '']:
103         return []
104
105     results = []
106     streams = scan_with_binwalk(file_path, archive_name)
107     if streams:
108         with open(file_path, 'rb') as f:
109             for offset, label in streams:

```

```

110         f.seek(offset)
111         carved_data = f.read(CARVE_SIZE)
112         if carved_data:
113             min_ent, max_ent, avg_ent = get_sliding_window_stats(carved_data,
WINDOW_SIZE, STEP_SIZE)
114             results.append({
115                 'archive_name': archive_name,
116                 'internal_file': internal_file,
117                 'category': category,
118                 'type': label,
119                 'offset': offset,
120                 'min_ent': round(min_ent, 4),
121                 'max_ent': round(max_ent, 4),
122                 'avg_ent': round(avg_ent, 4)
123             })
124     return results
125
126 def extract_archive_safe(archive_path, output_dir):
127     """Extracts an archive safely with timeout. Uses 7z for all archives to handle unsupported
compression and avoid unrar bugs."""
128     env = os.environ.copy()
129     env['LC_ALL'] = 'C.UTF-8'
130
131     # 7z handles zip, rar, tar, gzip, bzip2, etc. remarkably well and safely.
132     try:
133         subprocess.run(
134             ['7z', 'x', f'-o{output_dir}', '-y', archive_path],
135             stdout=subprocess.DEVNULL,
136             stderr=subprocess.PIPE,
137             env=env,
138             timeout=60,
139             check=True
140         )
141         return True, ""
142     except subprocess.TimeoutExpired:
143         return False, "7z extraction timeout"
144     except subprocess.CalledProcessError as e:
145         err_msg = e.stderr.decode('utf-8', errors='ignore').strip() if e.stderr else "Unknown
error"
146         return False, f"7z extraction failed: {err_msg}"
147     except Exception as e:
148         return False, f"7z execution error: {str(e)}"
149
150
151 def process_archive_task(args):
152     archive_path, file_name, category = args
153
154     # 1. Create a clean temp folder for this archive (isolated per worker)
155     file_temp_dir = tempfile.mkdtemp(dir=TEMP_EXTRACT_DIR)
156
157     all_results = []
158
159     try:
160         # 2. Extract
161         if file_name.lower().endswith(('.zip', '.rar')):
162             success, err_msg = extract_archive_safe(archive_path, file_temp_dir)
163             if not success:
164                 log_error(archive_path, err_msg)
165                 return False
166
167         # 3. Process every file found inside the archive
168         for ext_root, _, ext_files in os.walk(file_temp_dir):
169             for ext_file in ext_files:
170                 ext_path = os.path.join(ext_root, ext_file)
171                 # Analyze extracted file
172                 results = process_binary(ext_path, archive_path, ext_file, category)
173                 all_results.extend(results)
174
175         # 4. Cleanup extracted file immediately after scanning
176     try:

```

```

177         os.remove(ext_path)
178     except Exception:
179         pass
180 else:
181     # If it's already a .bin or other format, just process it directly
182     all_results.extend(process_binary(archive_path, archive_path, file_name, category)
183 )
184
185 # Write results and checkpoint file safely
186 _flush_results(all_results, archive_path)
187 return True
188
189 except Exception as e:
190     log_error(archive_path, f"Unexpected task error: {str(e)}")
191     return False
192 finally:
193     # Cleanup the temp directory structure
194     shutil.rmtree(file_temp_dir, ignore_errors=True)
195
196 def _flush_results(results, archive_path):
197     if write_lock:
198         with write_lock:
199             with open(OUTPUT_CSV, 'a', newline='') as csvfile:
200                 fieldnames = ['archive_name', 'internal_file', 'category', 'type', 'offset', '
min_ent', 'max_ent', 'avg_ent']
201                 writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
202                 for row in results:
203                     writer.writerow(row)
204                 with open(CHECKPOINT_FILE, 'a') as f:
205                     f.write(f"{archive_path}\n")
206
207 def main():
208     processed_files = set()
209     if os.path.exists(CHECKPOINT_FILE):
210         with open(CHECKPOINT_FILE, 'r') as f:
211             processed_files = set(f.read().splitlines())
212
213     write_header = not os.path.exists(OUTPUT_CSV)
214
215     if write_header:
216         with open(OUTPUT_CSV, 'w', newline='') as csvfile:
217             fieldnames = ['archive_name', 'internal_file', 'category', 'type', 'offset', '
min_ent', 'max_ent', 'avg_ent']
218             writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
219             writer.writeheader()
220
221     all_archives = []
222     for root, _, files in os.walk(DATASET_DIR):
223         for file in files:
224             archive_path = os.path.join(root, file)
225             if archive_path not in processed_files and file != ".DS_Store":
226                 category = get_category_from_path(archive_path)
227                 all_archives.append((archive_path, file, category))
228
229     total_archives = len(all_archives)
230     print(f"[*] Found {total_archives} unprocessed archives.")
231
232     if total_archives == 0:
233         return
234
235     m = mp.Manager()
236     l = m.Lock()
237     pool = mp.Pool(processes=os.cpu_count(), initializer=init_worker, initargs=(l,))
238
239     completed = 0
240     try:
241         for _ in pool.imap_unordered(process_archive_task, all_archives):
242             completed += 1
243             print(f"\r[*] Progress: {completed}/{total_archives} archives processed.", end='',
flush=True)

```

```
243     except KeyboardInterrupt:
244         print("\n\n[!] Interrupted by user. Terminating processes...")
245         pool.terminate()
246         pool.join()
247         shutil.rmtree(TEMP_EXTRACT_DIR, ignore_errors=True)
248         os.makedirs(TEMP_EXTRACT_DIR, exist_ok=True)
249         return
250
251     pool.close()
252     pool.join()
253
254     print(f"\n[+] Done. Cleaned up {TEMP_EXTRACT_DIR}")
255     shutil.rmtree(TEMP_EXTRACT_DIR, ignore_errors=True)
256
257 if __name__ == "__main__":
258     main()
```