# Automated Detection and Extraction of String Deobfuscation Functions in Malware Binaries via Static and Dynamic Analysis

Kelsie A. Edie

Brown University
Providence, RI
kelsie_edie@brown.edu

**Abstract**

String obfuscation remains one of the most pervasive challenges in malware reverse engineering, significantly complicating the analysis and timely response to cyber threats. Current automated solutions for identifying and extracting deobfuscation functions within malicious binaries are limited by their reliance on narrowly defined heuristics and specialized, malware-specific techniques. This paper presents a novel automated pipeline, combining comprehensive static analysis heuristics with statistical feature ranking to systematically detect and extract candidate string deobfuscation functions from malware binaries across multiple executable formats (PE, ELF, Mach-O). Leveraging Pyhidra and Ghidra's reverse engineering features, the implemented static framework evaluates each function against diverse heuristic indicators, including instruction patterns, XOR operation density, entropy analysis, and control-flow characteristics. Evaluation against six known malware binaries demonstrated high accuracy, correctly identifying all true deobfuscation functions among top-ranked candidates, while maintaining an interactive runtime suitable for practical incident response scenarios. However, high false-positive rates in smaller binaries revealed the need for complementary dynamic validation methods. Future extensions include integrating micro-execution and symbolic execution techniques to reduce false positives and employing machine learning methods to enhance heuristic specificity. This research provides a foundational step toward a comprehensive, scalable, and generalizable automated malware string deobfuscation framework, ultimately enabling faster defensive responses to rapidly, evolving cyber threats.

**Keywords:** Malware Analysis, Reverse Engineering, String Deobfuscation, Static Analysis, Dynamic Analysis, Pyhidra, Ghidra, Heuristic-based Detection.

# 1. Introduction

Malicious software, commonly known as malware, has rapidly become one of the most pervasive threats in today's ever-changing digital landscape. From destructive ransomware that paralyzes critical infrastructure, to insidious spyware that exfiltrates sensitive data; malware continues to cause widespread disruption and financial loss across all sectors worldwide. Even more alarmingly, ransomware attacks surged by 45% in the first quarter of 2025 compared to the previous year, a spike attributed to attackers compensating for declining ransom payments by increasing attack frequency [1, 2]. At the same time, the proliferation of spyware, capable of infiltrating devices and silently harvesting personal or organizational data, further underscores the sophistication and scale of modern cyber threats [3].

To effectively combat these evolving threats, cyber security professionals increasingly depend on advanced malware analysis and reverse engineering techniques. Malware analysis enables practitioners to examine malicious code, understand its behavior, identify its targets, and respond accordingly. Analysts leverage both static and dynamic analysis to detect vulnerabilities, generate signatures, and build timely defensive strategies. Reverse engineering further enriches this process by revealing concealed functions, decryption routines, and embedded logic otherwise undetectable at a surface level. These techniques are especially vital in addressing today's rapidly adapting malware variants [4, 5].

A critical element of this analytical workflow is string deobfuscation. Malware authors routinely obfuscate strings within binaries to thwart reverse engineering efforts and evade detection mechanisms. These strings often contain vital information such as command-and-control (C2) addresses, embedded URLs, configuration files, and encryption keys. Obfuscated strings severely hinder rapid triage and impede the development of behavioral or signature-based detections. However, most existing tools remain limited in scope, capable of analyzing only specific malware families or narrow classes of obfuscation techniques (e.g., fixed-key XOR, AES, or RC4 encryption) [6, 7, 8]. The need for more generalized and automated approaches that can address a wide spectrum of string obfuscation strategies is therefore imperative.

Currently, analysts often rely on labor-intensive manual methods for string deobfuscation, which requires both substantial expertise and time. Malware authors exploit this weakness through a range of obfuscation strategies, including arithmetic and bitwise manipulation, API-level indirection, and layered cryptographic encryption. This not only increases the complexity of the malware but also amplifies the difficulty of identifying the deobfuscation routines embedded within it [9, 10]. Although some automated deobfuscation tools exist, many are purely signature-based or limited to static decoding patterns, leaving them unable to generalize across architectures or complicated obfuscation styles.

This paper attempts to address that gap. It explores the question: *How can the automated identification and extraction of deobfuscation functions within malware binaries*

*be improved using static and dynamic analysis to facilitate string deobfuscation?* To that end, this research presents a framework that combines static feature extraction with while enabling further verification through dynamic validation to streamline the process of identifying likely string deobfuscation routines.

Due to dataset constraints and challenges with dynamic execution reliability, this work currently focuses on just static analysis. A custom script was developed using Pyhidra, a python library that provides direct access to API of Ghidra (a reverse engineering tool), to analyze each function in a binary and evaluate it against a suite of string deobfuscation indicators. These include instruction frequencies, entropy measures, control-flow patterns, and specific instruction (e.g., `MOV-XOR-MOV` sequences or fixed-key XOR signatures). The script then assigns $z$-scores to rank the statistical outliers per heuristic, which are then exported to a JSON format suitable for manual triage or automated post-processing. Future work will expand this pipeline by integrating symbolic execution tools and micro-execution environments to confirm the presence of actual string deobfuscation behavior.

**Contributions.** This work provides two key deliverables:

- a Pyhidra-based static analysis script that outputs a list of outlier functions likely to perform string deobfuscation, along with JSON-formatted results that facilitate further integration and automated tooling[1];
- a small, labeled dataset of malware binaries from various families with ground-truth annotations identifying known or suspected deobfuscation functions[2].

Together, these contributions advance the state of automated reverse engineering by enhancing analysts' ability to quickly identify and understand string deobfuscation logic, thereby empowering faster responses to sophisticated threats and accelerating analytical tool development.

# 2. Background and Related Work

Malware string deobfuscation lies at the intersection of malware analysis, reverse engineering, and, more recently, applied machine learning. This section surveys the disciplines informing this research, describes the ongoing technical challenges, and reviews current state-of-the-art methods that this work builds upon.

## 2.1 Malware Analysis and Reverse Engineering

Malware analysis involves identifying malicious code, examining its logic, and documenting its indicators of compromise (IOCs) to enable defenders to detect, block, or mitigate threats [4, 5]. Analysts typically rely on three primary methodologies:

[1] https://github.com/KelsieEdie/Static-String-Deobfuscation-Function-Detection/tree/main/v.1.0

[2] https://github.com/KelsieEdie/Static-String-Deobfuscation-Function-Detection/blob/main/Malware-Dataset.zip

- *Static analysis* examines a binary without execution. Tasks include disassembly, control-flow-graph reconstruction, string detection, entropy analysis, and comparative analysis to known samples. Static methods are quick and safe but struggle with packed, obfuscated, or self-modifying code [5].

- *Dynamic analysis* executes malware samples within controlled environments such as sandboxes, employing API tracing, taint tracking, debugging, and memory forensics to capture runtime behaviors, including outbound command-and-control (C2) traffic and real-time string decryption. Dynamic analysis is thorough but slower, resource-intensive, and vulnerable to evasion through anti-analysis techniques [5].

- *Reverse engineering* combines static and dynamic analysis, frequently using advanced techniques such as symbolic execution, data-flow slicing, and program instrumentation to reconstruct higher-level logic from low-level code structures [4].

## 2.2 Common String Obfuscation Techniques

Malware authors commonly obfuscate critical strings containing URLs, IP addresses, registry keys, or passwords, which can reveal their intentions. Table 1 summarizes frequently observed string obfuscation methods identified in contemporary malware reports [9, 10, 8].

Table 1: Common string obfuscation techniques used in malware

| **Obfuscation Type** | **Description** |
|---|---|
| Fixed-key XOR | Each plaintext byte is XORed with a single constant key. Simple to implement and easily reversible if the key is reused [9]. |
| Rolling XOR | A multi-byte key applied cyclically increases the string data entropy, but complicates brute-force recovery [9]. |
| Arithmetic Masking | Strings obfuscated via arithmetic (add/subtract) or bit-shift operations, altering original byte patterns significantly [9]. |
| Base64 Encoding | Binary data encoded into ASCII form to evade static string detection, commonly used for layering obfuscation [8]. |
| Cryptographic Ciphers (e.g., RC4, AES) | Secure cryptographic algorithms with dynamically generated keys, significantly complicating analysis and recovery [8]. |

As indicated by Table 1, adversaries frequently combine simple and complex obfuscation techniques, increasing analytical complexity and motivating the need for the development of versatile automated solutions.

## 2.3 Research on Automated Deobfuscation

Researchers have explored various methodologies to automate deobfuscation. Table 2 classifies these approaches and discusses their practical limitations based on recent literature.

Table 2: Current string deobfuscation techniques and limitations

| **Research Technique** | **Approach** | **Limitations** |
|---|---|---|
| Heuristic Extraction [6, 7, 11] | Pattern-matches common decoding routines and emulates code paths to recover strings. | Fails with novel encryption or crypto-based loaders; signature-driven methods require continuous updates. |
| Statistical Fingerprints [10] | Detects encoded data regions using entropy and n-gram statistical anomalies. | High false-positive rate with compressed or multimedia content; indirect identification of decoding routines. |
| Formal Methods [12] | Transforms binaries into intermediate representations (IR), solving semantic equivalence constraints via SMT solvers. | Scalability issues with large binaries; dependency on accurate IR translation; struggles with properly reconstructing obfuscated control flow. |
| ML-assisted Classification [13] | Uses convolutional neural networks (CNNs) to classify code segments containing cryptographic routines or decoding logic. | Demands extensive labeled datasets; retraining required for new instruction set architectures (ISAs); limited generalization to novel obfuscation types. |
| Reinforcement Learning [14] | Agent learns decoding logic via trial-and-error within a virtual machine (VM). | Effective primarily in controlled scenarios (e.g., Java bytecode); slow training convergence; impractical for large binaries. |
| Large-language Models (LLMs) [15, 16] | Prompt-based approach to LLMs for reconstructing decoder logic or plaintext from disassembly snippets. | Context limitations; prone to hallucinations; requires considerable analyst oversight and validation. |

Collectively, these studies listed in Table 2 demonstrate substantial progress but also highlight fragmented approaches, typically focusing narrowly on certain malware families, architectures, or obfuscation techniques.

## 2.4 Current Tools for Automated Deobfuscation

Table 3 details existing tools leveraged by cyber security analysts, including their capabilities and constraints.

Table 3: Commonly used deobfuscation tools

| Tool | Capabilities | Constraints |
|---|---|---|
| FLARE-FLOSS [17] | Heuristic tool detecting stack-/heap strings using common decoding patterns. | Limited to x86 PE files; ineffective against strong cryptographic encryption. |
| Pikabot-Deobfuscator [18] | IDA Pro plugin targeting Pikabot malware; extracts keys and reconstructs configuration data. | Family-specific; ineffective beyond known Pikabot structures. |
| Sidekick 2.0 [19] | Generates Binary Ninja scripts to analyze disassembly and deobfuscate strings using AI-driven methods. | Requires manual validation; commercial licensing restricts accessibility. |
| PowerPeeler/PowerDrive [20] | Dynamic analysis of obfuscated PowerShell scripts; unwraps Base64/Gzip/AES layers. | Limited to PowerShell scripts; not applicable to native binaries; Windows dependent. |

Despite their utility, these tools listed in Table 3 remain specialized and insufficient for general application across different file formats and obfuscation techniques.

## 2.5 Recent Advances and Ongoing Gaps

Recent advancements have seen significant integration of static and dynamic methodologies, reducing sandbox analysis time considerably [21]. Machine learning, reinforcement learning, and particularly LLMs have enhanced analyst productivity by automating deobfuscation to varying extents [15, 16, 22]. However, constraints like limited context-windows, model hallucinations, and manual supervision still pose challenges. No publicly available solution currently integrates cross-architecture support, explainable scoring, automatic static-to-dynamic handoff, and an openly licensed dataset for model training - emphasizing a clear research gap that this work aims to address.

# 3. Approach

## 3.1 Overview

The proposed automated pipeline facilitates malware string deobfuscation through four distinct stages:

1. **Static feature extraction.** Each function within an analyzed binary is processed using a Pyhidra script, generating detailed heuristic metrics (Section 3.2).

2. **Statistical ranking.** Functions are evaluated using per-binary $z$-scores for each heuristic metric, highlighting statistical outliers indicative of string deobfuscation logic.

3. **Deobfuscation candidate function export.** Outlier functions and associated feature vectors are outputted to a structured JSON file for straightforward integration with subsequent analysis stages.

4. **Dynamic confirmation.** Candidate functions identified statically can then be executed dynamically using micro-execution and/or symbolic execution to confirm actual string deobfuscation behavior.

## 3.2 Static Analysis Methodology

The script leverages Ghidra's robust disassembly and control-flow graph reconstruction via Pyhidra, systematically evaluating each function against several heuristic indicators. Inspired by and expanding upon methodologies from existing heuristic frameworks such as FLOSS [17, 7], these heuristics specifically target typical string deobfuscation and decoding behaviors:

- **Instruction-level metrics.** Functions are evaluated for the presence of non-zeroing XOR operations, since XOR encryption commonly characterizes string obfuscation. Particularly, loops containing XOR operations suggest repeated decoding patterns, which strongly indicate a decoding routine [6].

- **Control-flow complexity.** The script reconstructs the control-flow graph (CFG) and checks for loops and indirect jumps. Obfuscation routines often exhibit complex loops and atypical branch patterns, detected through heuristics such as branch density and indirect call frequency.

- **String-specific heuristics.** Dedicated indicators detect patterns strongly associated with string decoding: a `MOV-XOR-MOV` instruction sequence followed by conditional or unconditional jumps; consistent immediate XOR keys indicative of fixed-key XOR obfuscation; and calls to known string-related library functions like `strlen`, `strcpy`, or `memcpy`.

- **Entropy analysis.** XOR immediate operands undergo Shannon entropy calculation. Lower entropy values (typically below 2.5 bits per byte) are characteristic of meaningful ASCII or repetitive byte sequences, suggesting fixed-key XOR obfuscation rather than truly random cryptographic encryption.

- **Cryptographic heuristic detection.** Detection of AES or RC4 decryption logic is accomplished through identifying known cryptographic instructions (`AESENC`, `AESDEC`) and RC4 key-schedule patterns (`XCHG` operations with constants like 0x100 or 256).
- **SIMD instruction detection.** Presence of vectorized instructions such as `MOVAPS`, `MOVDQA`, and related SIMD instructions indicates potential cryptographic or optimized data transformations, flagged explicitly within the heuristic indicators.
- **Mixed-operation loops.** Loops with a high density of mixed operations (e.g., XOR combined with arithmetic and bitwise operations) indicate complex deobfuscation logic often used to evade simpler heuristic detections.

These metrics, derived from extensive malware-analysis literature and practical reverse-engineering experience, ensure comprehensive coverage of common and advanced obfuscation techniques.

### 3.3 Proposed Dynamic Analysis Integration

Deobfuscation Candidate functions, ranked and described via JSON, can be verified through dynamic analysis to confirm actual deobfuscation functionality. Tools like PANDA (for micro-execution) or angr (for symbolic execution) can execute candidate functions in isolation, verifying if decoded (deobfuscated) strings appear in memory. The results of this either confirm decoding routines or prompt further manual string deobfuscation analysis.

## 4. Implementation

### 4.1 Dependencies

The implemented system depends upon several integrated technologies:

- **Ghidra v11.3** provides disassembly, instruction semantics (P-code), and control-flow analysis.
- **Pyhidra v0.3** exposes Ghidra's Java API directly in Python, allowing scripting and automation for analysis.
- **Python v3.12** handles the analytical computations (statistics, entropy calculations) and the structured data (JSON output).
- **Dynamic analysis (not implemented)** via angr and PANDA can be used for automated confirmation of the candidate functions identified statically.

## 4.2 Script Architecture

The implementation comprises two primary scripts:

### 4.2.1 Main Analysis Script (`deobfuscation_analysis.py`).

This file conducts the overall analysis:

1. Initializes the Ghidra project and loads the binary.
2. Iterates over all identified functions, forcing full disassembly if necessary.
3. Computes metrics via helper functions (`count_non_zeroing_xor`, `is_loop_detected`, `get_string_deobfuscation_indicators`, `compute_stats`, etc.).
4. Calculates statistical summaries (mean, median, standard deviation) and assigns z-scores.
5. Produces a structured JSON output summarizing function analysis results and outlying candidate functions (based on the highest z-score for each heuristic).

### 4.2.2 Helper Functions (`deobfuscation_analysis_helpers.py`).

This file contains the core heuristic computations and utilities; a few representative functions include:

- **Entropy calculations** (e.g., `compute_entropy`) measure the randomness in XOR keys.
- **Control-flow analysis functions** such as `is_loop_detected` and `get_loop_blocks` determine loop existence and complexity.
- **Heuristic indicators** implemented in `get_string_deobfuscation_indicators` detect specific decoding patterns, including instruction sequences, consistent XOR keys, library calls, cryptographic operations, mixed loop operations, and SIMD instructions.

## 4.3 Script Snippet

Listing 1 shows the Python helper function that counts non-zeroing XOR instructions.

```
    def count_non_zeroing_xor(instructions):
        count = 0
        for instr in instructions:
            if instr.getMnemonicString().upper() == XOR_MNEMONIC:
                op0 = instr.getOpObjects(0)
                op1 = instr.getOpObjects(1)
                if op0 != op1:
```

```
                    count += 1
        return count

```

Listing 1: Code snippet for counting non-zeroing XOR operations

This logic captures XOR instructions that modify rather than clear register values, which is strongly indicative of obfuscation routines commonly reported in malware analyses [6, 7].

### 4.4 Output Format and Interoperability

The structured JSON output generated contains:

- Metadata about the analyzed binary (file type, total functions).
- Overall statical results (mean, median, standard deviation) for each metric.
- Outlier functions with the highest z-score per metric.
- Per-function metrics and normalized z-scores.
- Program's call graph.

### 4.5 Current Status and Planned Extensions

The static analysis framework, JSON output capability, and the curated labeled malware dataset are currently complete. Future extensions include:

- **Dynamic confirmation integration.** Automating the handoff from static to dynamic analysis using tools such as angr and PANDA.
- **Machine learning enhancement.** Incorporating deep learning (CNNs) or large language models (LLMs) to augment or replace heuristic-based ranking methods, improving generalization across architectures and malware families.

This implementation provides a significant step toward a comprehensive, automated deobfuscation analysis pipeline that combines the strengths of heuristic static analysis and confirmatory dynamic execution, ultimately enhancing incident response capabilities.

## 5. Evaluation

This section describes how I assess the static deobfuscation pipeline on real malware samples, presents quantitative results, and discusses key takeaways as well as future work.

## 5.1 Experimental Setup

**Dataset.** I conducted this evaluation on six ground-truth binaries spanning Mach-O, PE, and ELF formats. Each contains a single known deobfuscation function (Table 4). Four additional unlabeled binaries (Pikabot_S1, Pikabot_S2, Mekotio, Mirai) were omitted from quantitative metrics due to lack of certainty about their deobfuscation functions.

**Ground truth.** A JSON file (`ground_truth.json`) maps each project name to its true deobfuscation function's identifier ("`<function_name> @ <entry_address>`), matching the `identifier` field in my analysis output.

**Metrics.**

- *Detection rate* (TP (True Positive) / (TP (True Positive) + FN (False Negative))): the fraction of binaries where the true deobfuscation function appears among the flagged candidates.
- *False-positive rate* (FP (False Positives) / (Total Functions – 1)): the fraction of non-deobfuscation functions erroneously flagged.
- *Runtime*: the execution time of the static analysis script.

**Environment.**

- *Hardware* - MacOS v15.3.2 running on a 2.6GHz 6-Core Intel Core i7 CPU with 16GB RAM.
- *Software* - Ghidra v11.3, Pyhidra v0.3, Python v3.12.

Table 4: Malware samples and their ground-truth deobfuscation functions

| Sample | Function Identifier | Format / Obfuscation Type |
|---|---|---|
| simple_deobs | _deobfuscate @ 0x100000f25 | Mach-O / fixed-key XOR |
| simple_deobs.exe | _deobfuscate @ 0x401450 | PE / fixed-key XOR |
| Darkside_S1.exe | FUN_00404C84 @ 0x404C84 | PE / 128-bit ARX cipher |
| Darkside_S2.exe | FUN_0040209C @ 0x40209C | PE / 16-byte ARX decryption |
| Lockbit_E_S2 | FUN_004056D0 @ 0x4056D0 | ELF / looped XOR variants |
| Amadey | FUN_00403249 @ 0x403249 | PE / fixed-key XOR |

## 5.2 Evaluation Methodology

I automated this evaluation using a driver script (see Listing 2). For each sample, it:

1. Invokes `deobfuscation_analysis.py` with the sample's file paths.
2. Records the analysis runtime.

3. Loads the resulting JSON and extracts all "`highest_zscores`" (functions identified as having some sort of deobfuscation behavior) as the predicted set.

4. Computes True Positives, False Negatives, and False Positives against the ground truth.

```python
def main():
    gt = load_ground_truth(GT_PATH)
    perfs, metrics = [], []

    for sample in SAMPLES:
        print(f"\n=== Evaluating {sample['project']} ===")
        rt, out = run_analysis(sample)
        print(f"Runtime: {rt:.1f}s")
        cands, total = extract_candidates(out)
        stats = evaluate_sample(sample["project"], cands, gt, total)
        stats["runtime"] = rt
        print(f" Detection rate: {stats['DR']*100:.1f}%  |  False-pos
   rate: {stats['FPR']*100:.1f}%")
        perfs.append(rt)
        metrics.append(stats)

    # Summary
    avg_rt = mean(perfs)
    avg_dr = mean(m["DR"] for m in metrics)
    avg_fpr= mean(m["FPR"] for m in metrics)
    print("\n=== Summary ===")
    print(f"Average runtime: {avg_rt:.1f}s")
    print(f"Average detection rate: {avg_dr*100:.1f}%")
    print(f"Average false-positive rate: {avg_fpr*100:.1f}%")
```

Listing 2: Main function of the evaluation script

## 5.3 Results

Table 5 summarizes per-sample detection rate, false-positive rate, and runtime. Across all six binaries, I achieve a perfect detection rate, false positives vary inversely with code complexity, and static analysis completes in under 25s even on the largest sample.

## 5.4 Discussion

**Strengths.** My static analysis pipeline demonstrates several notable strengths:

- *High accuracy on known deobfuscation functions.* Across all six labeled samples evaluated, the analysis consistently identified the correct decoding functions among its top-ranked deobfuscation candidates. This perfect detection rate underscores the soundness of my heuristic selection, especially for common obfuscation patterns such as XOR-based encryption and loop-driven decoders.

Table 5: Per-sample evaluation results

| **Sample** | #Funcs | Runtime (s) | Detection (%) | False-pos (%) |
|---|---|---|---|---|
| simple_deobs | 4 | 9.0 | 100.0 | 100.0 |
| simple_deobs.exe | 86 | 9.9 | 100.0 | 65.9 |
| Darkside_S1.exe | 149 | 12.4 | 100.0 | 45.3 |
| Darkside_S2.exe | 101 | 11.7 | 100.0 | 51.0 |
| Lockbit_E_S2 | 774 | 22.1 | 100.0 | 54.2 |
| Amadey | 2 | 9.3 | 100.0 | 100.0 |
| **Average** | — | 12.4 | 100.0 | 69.4 |

- *Cross-format compatibility.* Unlike many existing tools, my approach supports Mach-O, PE, and ELF binaries without having to modify the heuristic rule-set or implementation details. This format-blind design significantly broadens the applicability of my method, providing utility across diverse operating systems and malware families.

- *Practical runtime performance.* The entire static analysis consistently completes in under 25 seconds, even for relatively large samples containing hundreds of functions. Such performance, demonstrated on typical commodity hardware (a 6-core Intel Core i7 laptop), confirms that the pipeline is suitable for interactive usage and rapid incident-response workflows.

**Weaknesses.** Despite these strengths, several important limitations emerged from this evaluation:

- *High false-positive rate, particularly in smaller binaries.* In extremely small binaries (e.g., the Amadey sample, with only two functions), my pipeline flagged all available functions as potential decoders, resulting in a false-positive rate of 100%. Similarly, mid-sized PE samples (such as the Darkside variants) also exhibited relatively high false-positive rates ( 45–66%). This indicates that while heuristics effectively identify decoding behavior, they currently lack sufficient specificity to exclude unrelated but structurally similar functions.

- *Dependence on Ghidra's disassembly quality.* My pipeline heavily relies on Ghidra's disassembly accuracy. Poorly disassembled or intentionally obfuscated binaries could generate incomplete instruction streams, leading to incorrect heuristic scoring and reduced accuracy. For example, binaries employing aggressive packers, anti-disassembly techniques, or highly dynamic/self-modifying code could substantially degrade the effectiveness of my current implementation.

**Key Takeaways.** The evaluation results highlight several critical insights about my static pipeline's practical application:

- While my heuristics reliably detect true deobfuscation functions (with high accuracy), high false-positive rates remain a key challenge. This emphasizes the importance of complementing the static analysis with secondary validation methods, such as dynamic execution or symbolic analysis, to efficiently validate or reject flagged deobfuscation candidate functions.

- The lower false-positive rates observed in larger binaries suggest heuristic scores become more discriminative as binary complexity increased. Small binaries inherently have fewer functions and less diversity, thereby elevating the relative significance of any heuristic matches. Adjusting heuristic sensitivity or introducing size-dependent normalization could mitigate this issue.

- Cross-format support without the need for any modification is a major strength, indicating my method's general applicability. This suggests that future iterations could benefit significantly from extending the corpus of known decoder functions across a wider range of malware families and (de)obfuscation strategies.

**Future Work.** Based on these insights, several directions are clearly warranted to enhance the robustness and effectiveness of the pipeline:

- *Integration with dynamic analysis tools.* Incorporating automated micro-execution or symbolic execution tools such as `PANDA`, `angr`, Qiling, or Unicorn would allow rapid confirmation or rejection of potential candidate functions. Such an approach would drastically reduce false positives by filtering the heuristic matches through runtime behavior validation.

- *Improved heuristic specificity.* Leveraging machine learning approaches, such as supervised classifiers or large-language models (LLMs), trained on a broader labeled corpus of malware samples, could improve heuristic specificity and reduce false-positive rates. Machine learning-based models could dynamically weight heuristic indicators according to empirical feature relevance, thereby optimizing discrimination between true deobfuscation functions and benign segments of code.

- *Adaptive heuristic weighting.* Current heuristics operate under the same of assumptions across all binaries. Future iterations might employ adaptive weighting strategies that factor in binary characteristics, such as binary size, code complexity, and instruction set architecture, to dynamically adjust heuristic thresholds and relevance scores.

- *Expanding ground-truth corpus.* Building a more extensive labeled dataset, encompassing diverse obfuscation types (e.g., custom block ciphers, metamorphic malware), would further improve heuristic calibration. A more comprehensive dataset would not only refine existing heuristics but also facilitate the creation of novel indicators tailored explicitly to emerging obfuscation strategies.

- *Enhanced resilience against anti-analysis techniques.* Improving my heuristic robustness against adversarial methods, such as control-flow flattening and anti-disassembly patterns, represents a crucial future direction. Techniques like IR-based analysis or hybrid static-dynamic disassembly approaches could help overcome some current limitations stemming from disassembly incompleteness.

Altogether, the combination of high accuracy, broad applicability, and practical runtime demonstrates strong initial promise. However, significantly reducing false positives, integrating dynamic verification, and refining heuristic adaptability remain critical steps toward building a truly comprehensive and robust automated deobfuscation pipeline.

# 6. Conclusion

In this paper, I introduced an automated, static analysis framework designed to systematically detect and extract string deobfuscation functions within malware binaries. By leveraging a diverse set of heuristic indicators and employing statistical ranking techniques, the developed pipeline demonstrated effective and consistent identification of true decoder functions across multiple executable formats. The evaluation conducted on a representative set of labeled malware samples yielded high accuracy, ensuring all true deobfuscation routines were correctly flagged among top-ranking candidates. Additionally, the practical runtime observed (under 25 seconds per binary) validated the approach's suitability for real-world incident response workflows.

Despite these strengths, the high false-positive rate, particularly apparent in smaller binaries, highlights critical areas for improvement. Dependence on Ghidra's disassembly accuracy underscores vulnerabilities to adversarial obfuscation methods like packing and anti-disassembly techniques. Future work will directly address these limitations by integrating automated dynamic validation methods such as micro-execution (via `PANDA` or similar tools) and symbolic execution (`angr`) to quickly and reliably confirm or reject candidate functions identified statically. Furthermore, refining heuristic specificity through machine-learning-based models trained on a broader, diversified ground-truth corpus will significantly improve accuracy. To this point, this research marks a significant advancement toward a unified, robust pipeline for malware string deobfuscation.

# Availability

The Deobfuscation Candidate Finder is available at: https://github.com/KelsieEdie/Static-String-Deobfuscation-Function-Detection/

# References


[1] S. Alder, "Cybersecurity firms report record-breaking quarter for ransomware attacks," 2025. [Online]. Available: https://www.hipaajournal.com/q1-2025-ransomware-report/

[2] R. Harpur, "The state of ransomware 2025," 2025. [Online]. Available: https://www.blackfog.com/the-state-of-ransomware-2025/

[3] M. Shaheryar, "Understanding spyware malware, infiltration techniques and protective measures," 2025. [Online]. Available: https://mobizinc.com/resources/spyware-malware/

[4] CanaryTrap, "Understanding malware analysis and reverse engineering," 2024. [Online]. Available: https://www.canarytrap.com/blog/malware-analysis/

[5] A. Arharbi, "Comparing static and dynamic malware analysis: Unveiling the secrets of cyber threats," 2023. [Online]. Available: https://medium.com/@aa.adnane/comparing-static-and-dynamic-malware-analysis-unveiling-the-secrets-of-cyber-threats-f6be509b2

[6] M. Raabe and W. Ballenthin, "Automatically extracting obfuscated strings from malware using the fireeye labs obfuscated string solver (floss)," 2016. [Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/automatically-extracting-obfuscated-strings/

[7] W. Ballenthin, M. Raabe, and B. Stancill, "Floss version 2.0," 2022. [Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/floss-version-2/

[8] ReasonLabs, "What is deobfuscation?" 2023. [Online]. Available: https://cyberpedia.reasonlabs.com/EN/deobfuscation.html

[9] S. S., "Advanced string obfuscation techniques," 2023. [Online]. Available: https://steve-s.gitbook.io/0xtriboulet/just-malicious/advanced-string-obfuscation

[10] L. Glanz, P. Müller, L. Baumgärtner, M. Reif, S. Amann, P. Anthonysamy, and M. Mezini, "Hidden in plain sight: Obfuscated strings threatening your privacy," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. ACM, Oct. 2020, p. 694–707. [Online]. Available: http://dx.doi.org/10.1145/3320269.3384745

[11] H. Säıdi, P. A. Porras, and V. Yegneswaran, "Experiences in malware binary deobfuscation," 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:15552623

[12] R. David, "Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes," Theses, Université de Lorraine, Jan. 2017. [Online]. Available: https://theses.hal.science/tel-01549003

[13] P. Mondon and R. de Lemos, "Detecting cryptographic functions for string obfuscation," in *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2024, pp. 315–320.

[14] M. Sewak, S. K. Sahay, and H. Rathore, "Drldo a novel drl based de obfuscation system for defence against metamorphic malware," *Defence Science Journal*, vol. 71, no. 1, p. 55–65, Feb. 2021. [Online]. Available: http://dx.doi.org/10.14429/dsj.71.15780

[15] C. Patsakis, F. Casino, and N. Lykousas, "Assessing llms in malicious code deobfuscation of real-world malware campaigns," 2024. [Online]. Available: https://arxiv.org/abs/2404.19715

[16] M. A. Ferrag, F. Alwahedi, A. Battah, B. Cherif, A. Mechri, N. Tihanyi, T. Bisztray, and M. Debbah, "Generative ai in cybersecurity: A comprehensive review of llm applications and vulnerabilities," 2025. [Online]. Available: https://arxiv.org/abs/2405.12750

[17] Mandiant, "Flare obfuscated string solver," 2025. [Online]. Available: https://github.com/mandiant/flare-floss

[18] ThreatLabz, "Pikabot deobfuscator," 2025. [Online]. Available: https://github.com/threatlabz/pikabot-deobfuscator

[19] B. Knudson, "Sidekick in action: Deobfuscating strings in amadey malware," 2024. [Online]. Available: https://binary.ninja/2024/08/12/sidekick-in-action-deobfuscating-strings-in-amadey-malware.html

[20] R. Li, C. Zhang, H. Chai, L. Ying, H. Duan, and J. Tao, "Powerpeeler: A precise and general dynamic deobfuscation method for powershell scripts," 2024. [Online]. Available: https://arxiv.org/abs/2406.04027

[21] P. Shijo and A. Salim, "Integrated static and dynamic analysis for malware detection," *Procedia Computer Science*, vol. 46, pp. 804–811, 2015, proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace Island Resort, Kochi, India. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915002136

[22] H. Wang, N. Luo, and P. LIu, "Unmasking the shadows: Pinpoint the implementations of anti-dynamic analysis techniques in malware using llm," 2024. [Online]. Available: https://arxiv.org/abs/2411.05982