Empowering IoT Security: Automated Identification of Standard Library Functions in RTOS Firmware with LLM and RAG

Zhihan Zheng*, Yu-an Tan†, Zihan Chen‡, Zheng Lu§, Weizhi Meng¶ and Shuo Wang‖

***Abstract*—Reverse engineering embedded firmware for Real-Time Operating Systems (RTOS) is a formidable challenge in Internet of Things security. The common practice of stripping symbolic information from firmware to optimize performance and storage makes identifying standard library functions exceptionally difficult, creating a significant bottleneck for functional analysis and vulnerability discovery. This paper introduces a novel, automated method for identifying these functions in RTOS firmware by leveraging Large Language Models. Our approach operates directly on raw binary code, requiring no symbolic or debugging information, and demonstrates broad generalizability across diverse hardware architectures and compilers. At its core, the method combines Retrieval-Augmented Generation (RAG) to enhance identification accuracy with a newly designed Adaptive Iterative Screening Algorithm (AISA), which optimizes analysis efficiency by prioritizing candidate functions based on a weighted score of call frequency, call depth, and address proximity to reduce token costs. We validated our method through rigorous experimentation on Zephyr RTOS firmware spanning nine architectures (e.g., ARM, MIPS, RISC-V). The results are compelling: our approach achieves a 90.59% accuracy rate in identifying standard library function names. Moreover, its application to commercial IoT firmware confirms its high efficiency, identifying functions significantly faster and more economically than traditional heuristic techniques. This work contributes a powerful, general-purpose, and cost-effective solution for automated firmware analysis.**

***Index Terms*—RTOS Firmware Analysis, Library Function Identification, Large Language Model (LLM), Retrieval-Augmented Generation (RAG), IoT Security.**

I. Introduction

With the rapid proliferation of the Internet of Things (IoT) and Industrial Control Systems (ICS), embedded devices have become integral components of modern information infrastructure [1]. While their low-power and high real-time characteristics meet the demands of specific application scenarios, they also introduce inherent security vulnerabilities, leading to a surge in attacks targeting IoT devices [2]. As the foundational software layer, the firmware directly controls hardware resources and implements device functionalities, making its security paramount to the stability and safety of the entire IoT ecosystem [3]. The presence of vulnerabilities in firmware can allow attackers to gain remote control of devices, exfiltrate sensitive information, or even orchestrate large-scale attacks, resulting in severe consequences.

In the field of firmware reverse engineering, library function name recovery is a foundational step in binary analysis, playing a critical role in security research activities such as vulnerability discovery and malicious code detection [4]. During the compilation process, symbolic information like function and variable names is often stripped, making it difficult for reverse engineers to intuitively comprehend the code's logic and functionality [5]. Library function name recovery aims to restore this lost information by analyzing structural features, data flows, and matching against standard library functions within the binary. Recovered symbols can significantly enhance the efficiency and accuracy of reverse analysis, enabling researchers to grasp the code's intent and functionality with greater clarity [6].

However, the analysis of vulnerability patterns and attack vectors is significantly complicated by the heterogeneity of IoT embedded firmware, its resource-constrained nature, and the prevalent use of custom compilation toolchains by manufacturers [7]. IoT devices span diverse domains, including smart homes, industrial control, and medical equipment, with substantial variations in firmware functionality, architecture, and implementation. Concurrently, highly optimized compilation strategies are often employed to adapt to resource-limited hardware environments, and many vendors utilize bespoke toolchains. These factors present formidable challenges to firmware reverse engineering. This is particularly true for RTOS-based systems, where firmware operates directly on the Hardware Abstraction Layer (HAL), devoid of Operating System level symbolic information. Consequently, conventional function identification methods that rely on formatted executables are rendered ineffective. In an RTOS environment, the absence of metadata support, such as symbol tables or import/export tables, prevents the direct application of traditional symbol recovery techniques. This challenge, combined with the tight coupling of firmware code to hardware registers and memory addresses, dramatically increases the analysis's difficulty and complexity. In the context of RTOS, the accurate identification of standard library functions is not merely a convenience but a critical prerequisite for security analysis. These functions often serve as the building blocks for complex logic, and their misidentification can lead to significant gaps in control flow reconstruction and vulnerability assessment.

Traditional analysis methods, including signature-based matching and statistical heuristics, fail in this heterogeneous RTOS landscape due to their reliance on specific compiler

Shuo Wang is corresponding author
*School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China, Email: zhengzhihan@bit.edu.cn
†School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China, Email: tan2008@bit.edu.cn
‡School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China, Email: zihan@bit.edu.cn
§School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China, Email: luzheng15@bit.edu.cn
¶School of Computing and Communications, Lancaster University, Lancaster LA1 4YW, United Kingdom, Email: weizhi.meng@ieee.org
‖School of Mechanical Engineering, Beijing Institute of Technology, Beijing 100081, China, Email: shuo.wang@bit.edu.cn

artifacts. To overcome these limitations, researchers have applied deep learning models, such as Long Short-Term Memory (LSTM) networks [8], Graph Neural Networks (GNNs) [9], and Transformers [10], to learn more generalized representations from Control Flow Graphs (CFGs) or instruction sequences. However, these structure-centric approaches still struggle with "semantic displacement"—a challenge we detail later—where the core logic of a function is offloaded to a private helper function. This breaks the structural pattern of the caller, rendering deep learning-based matching ineffective. In contrast, Large Language Models (LLMs) possess a unique advantage in semantic comprehension. They can infer function purpose from instruction semantics and context, much like a human analyst, making them resilient to structural variations that confound traditional neural networks [11]. However, directly applying LLMs to RTOS firmware analysis remains challenging. The sheer volume of functions in a firmware image makes exhaustive analysis prohibitively expensive due to high token costs and inference latency, while general-purpose models often lack the specific domain knowledge to accurately interpret stripped binary code without external retrieval.

To address these challenges, this paper proposes a novel method based on Large Language Models (LLMs) for identifying standard library functions in RTOS embedded firmware. This method transcends the limitations of traditional approaches that depend on compile-time symbols, enabling the identification of standard library functions within RTOS firmware in a black-box manner by analyzing only the binary code. Moreover, our approach demonstrates cross-architecture and cross-compiler versatility, achieving efficient and accurate identification across nine different hardware architectures and compilation toolchains. The effectiveness of this method has been validated through extensive experimentation, with results indicating its robust performance and reliability in practical RTOS firmware analysis.

Firmware for Real-Time Operating Systems is typically stripped of all symbols to optimize space and performance, posing a substantial challenge for reverse engineers. Analysts often expend considerable time manually identifying fundamental library functions (e.g., for memory operations, string manipulation) before they can begin to decipher the program's control and data flows [12], [13]. Our proposed analysis workflow dramatically accelerates this process. By rapidly and accurately identifying most standard library functions at a low cost during the initial analysis stages, our method automatically provides reverse engineers with a functionally "annotated" list of routines. This allows analysts to immediately focus on the non-library functions that implement the core business logic, thereby significantly enhancing overall analysis efficiency.

Many security vulnerabilities are directly linked to the improper use of specific standard library functions [14], such as buffer overflows caused by *strcpy* or format string vulnerabilities originating from *sprintf*. Our method provides an efficient target screening mechanism for the automated discovery of such vulnerabilities. Security auditors can leverage our approach to first generate a high-confidence list of library functions within the firmware and then automatically filter this list for known "dangerous functions." This process drastically narrows the scope of investigation from thousands of unknown functions to a few dozen high-risk targets. Auditors can then concentrate their efforts on conducting in-depth code reviews of these high-risk functions and their calling contexts, enabling a more efficient, precise, and scalable approach to identifying potential security flaws [15].

In summary, the main contributions of this paper are as follows:

1) We propose a comprehensive framework that integrates Large Language Models with Retrieval-Augmented Generation (RAG) to identify standard library functions in stripped RTOS binaries. This approach bridges the gap between general-purpose LLM capabilities and the specific domain knowledge required for binary analysis.
2) We design a Hierarchical Identification Strategy to resolve the "semantic displacement" problem—where core logic is offloaded to private helper functions. By modeling the entire call chain as a unified semantic context, our method succeeds where traditional single-function analysis fails.
3) We introduce the Adaptive Iterative Screening Algorithm (AISA), a novel pre-filtering mechanism that exploits the memory spatial locality and call graph features of library functions. This algorithm significantly reduces the search space, making expensive LLM-based analysis computationally feasible and scalable.
4) We demonstrate the method's effectiveness through extensive experiments on diverse RTOS firmware across nine architectures. Our approach achieves a stable accuracy of 90.59%, validating its robustness and cross-platform generalizability without the need for model retraining.

This paper is organized as follows. Section II reviews related work. Section III details our proposed methodology. Section IV presents our experimental evaluation. Section V discusses the findings, limitations, and future work. Finally, Section VI concludes the paper.

II. Related Work

This section reviews prior research pertinent to our study, focusing on binary function similarity analysis, the specific task of library function identification, and the emerging application of Large Language Models in reverse engineering.

A. Binary Function Similarity Analysis

Binary function similarity analysis is a cornerstone problem in software security, with significant progress driven by practical needs in vulnerability discovery and malware detection [16], [17]. Prevailing research revolves around code representation learning and cross-platform semantic alignment.

One major line of research leverages Control Flow Graphs (CFGs) and Graph Neural Networks (GNNs) to capture structural program features. For instance, jTrans [18] embeds control flow information into a Transformer model, enhancing robustness across different compilation environments

through pre-training tasks. PDM [19] utilizes an enhanced Attributed Control Flow Graph (ACFG+) to fuse multi-level positional distribution information, achieving improved accuracy in cross-architecture scenarios. While these graph-based methods effectively model program topology, they remain sensitive to code morphological changes introduced by compiler optimizations such as function inlining, and their computational overhead increases significantly with graph size.

To overcome these limitations, another research avenue explores semantic parsing via Intermediate Representation (IR). Asteria [20] employs Abstract Syntax Trees (ASTs) in conjunction with a Tree-LSTM to model cross-platform function semantics, demonstrating superior similarity detection accuracy and efficiency over tools like Diaphora [21] in IoT firmware analysis. UPPC [22] introduces a pseudocode translation technique, using a deep pyramid convolutional network to extract high-level semantic features, which notably improves detection robustness in obfuscated scenarios. Although these IR-based methods mitigate platform differences through the architecture-agnostic nature of IR, their performance is highly contingent on the accuracy of the underlying decompilation tools and they still face challenges when processing stripped binaries.

More recently, representation optimization frameworks based on contrastive learning have emerged as a key direction for enhancing model generalization. CEBin [23] combines embedding retrieval with fine-grained contrastive matching, achieving an 85.46% recall rate in searches across a million-function library, with an efficiency improvement of three orders of magnitude over traditional methods. BinCola [24] designs a diversity-sensitive contrastive loss function that mitigates data distribution shifts through dynamic weight adjustments, improving performance on cross-optimization-level tasks by 33.62%.Inter-BIN [25] implements dynamic semantic interaction modeling through co-attentional alignment of cross-architecture instruction sequences, showing excellent performance on cross-architecture binary similarity comparison tasks at various input granularities. These approaches achieve a balance between efficiency and precision through the synergistic design of feature engineering and model architecture, offering viable solutions for large-scale practical deployment.

B. Library Function Identification

The recovery of library functions is a vital task in reverse engineering, as it greatly simplifies the comprehension of program behavior. Existing techniques can be categorized as follows:

Signature-based matching is the most established and widely adopted technique in the industry. Its core principle is to create a unique ”fingerprint” or ”signature” for the binary code generated by each known library function under specific compilers and compilation options. When analyzing a target binary, these signatures are scanned to identify library functions. The most representative tool in this domain is the Interactive Disassembler (IDA) Pro’s Fast Library Identification and Recognition Technology (FLIRT) [26]. While this method offers high accuracy and speed upon a successful match, its primary drawbacks are its brittleness and high maintenance overhead. Signatures are tightly coupled to compiler versions, optimization flags, target architectures, and even minor source code modifications, any of which can invalidate them. Furthermore, covering the vast landscape of library versions and compilation environments requires the construction and continuous maintenance of a massive signature database.

To overcome the fragility of signature-based matching, researchers have turned to statistical features that are more resilient to compilation variations. These methods extract structural, syntactic, and statistical properties of a function—such as Control Flow Graph metrics [6], instruction set frequencies, and the presence of specific constants—to form a feature vector. Matching is then performed using machine learning classifiers (e.g., Support Vector Machines, decision trees) or similarity metrics [27]–[29]. Although these approaches exhibit better robustness against compiler optimizations, they typically yield lower accuracy than signature matching and are prone to a higher false-positive rate, especially when distinguishing between functions that are functionally similar but originate from different sources.

With the recent success of deep learning in natural language processing and code analysis, researchers have begun to treat binary code as a ”language” and model it using neural networks. Early works adapted Natural Language Processing (NLP) techniques, such as asm2vec [30] which learns instruction embeddings, or utilized Graph Neural Networks to match function CFGs [31], [32]. More recently, Transformer-based Large Language Models have demonstrated a superior capacity for understanding code semantics, enabling the direct identification of function purposes from assembly code [33]. These methods are highly robust to compiler optimizations and possess a degree of zero-shot capability for identifying new or unknown library function versions. However, their most significant bottleneck lies in the prohibitive token costs and inference latency, which make exhaustive, brute-force analysis of entire firmware impractical [34].

C. LLM-Assisted Reverse Analysis

Large Language Models have recently demonstrated transformative potential in the field of software reverse engineering [35]. Initial research on the semantic comprehension capabilities of LLMs for decompiled code provided early validation [36]. Experiments demonstrated that an early, non-fine-tuned general-purpose LLM code-davinci-001 could identify some variable roles and code intentions. However, its accuracy in a zero-shot reverse engineering evaluation task was merely 53.39%. This performance underscored the inherent limitations of zero-shot inference in scenarios lacking symbolic information, highlighting the necessity of constructing specialized datasets and fine-tuning models for reverse engineering applications.

To bridge the semantic gap in binary reverse engineering, subsequent efforts have focused on domain adaptation. ReSym [37] fine-tunes Large Language Models (LLMs) on the unique patterns of decompiled code to transfer their extensive knowledge of source code to the reverse engineering domain. By

breaking the problem down into two sub-tasks—recovering local variables and reconstructing user-defined structures—and fine-tuning a separate LLM for each, the system allows the models to focus on simpler, more specific tasks. DeGPT [38] designed a three-role mechanism to optimize the LLM's understanding of decompiler output structures, reducing the cognitive burden of this task by 24.4%. These works collectively establish the importance of domain-specific training for function recovery tasks.

On the front of model architecture optimization, Liu et al. [39] explored LLM-assisted taint propagation analysis, where the model's ability to perform cross-procedural context modeling offered a new avenue for identifying function parameters. Jelodar et al. [40] proposed a name generation framework constrained by control-flow semantics, enhancing the logical consistency of function names through program dependence graph modeling. Current research indicates that LLMs are evolving from passive semantic parsers to active inferential reconstruction agents. Nevertheless, breakthroughs are still needed in areas such as cross-architecture generalization and the parsing of complex control flows, pointing toward future directions for deep-program-understanding-based function recovery techniques.

III. Methodology

This section presents our three-stage methodology for identifying standard library functions in firmware. First, we introduce a Hierarchical Identification Strategy (Section III-A) to resolve the semantic complexity arising from multi-level function call chains. To make this deep analysis scalable, we then apply the Adaptive Iterative Screening Algorithm (AISA) (Section III-B), which serves as an efficient pre-filter to reduce the search space to a small set of high-probability candidates. Finally, we employ a RAG-enhanced LLM Validator (Section III-C) to perform a high-fidelity identification on this candidate set, ensuring accuracy by augmenting the model with an expert knowledge base. This integrated pipeline ensures our method is simultaneously deep in analysis, efficient in scale, and accurate in its results.

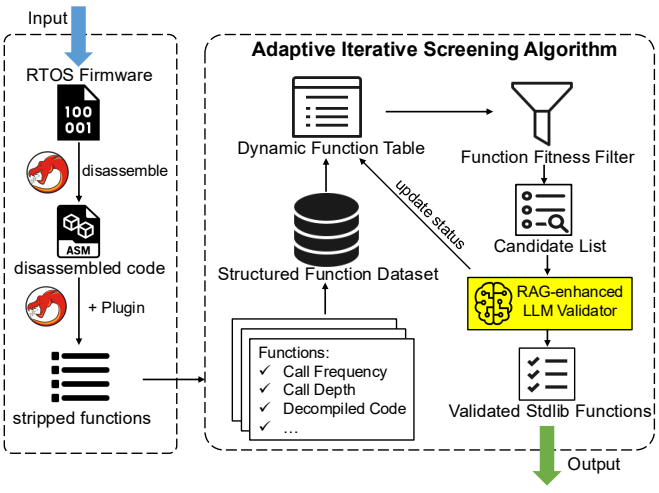

Fig. 1: Overview of the Proposed Methodology

A. Hierarchical Function Identification Strategy

In the domains of software reverse engineering and program analysis, the identification and interpretation of standard library functions represent a foundational research topic. The conventional understanding posits that standard library functions in firmware are atomic functional units with high internal cohesion; that is, their core processing logic is fully encapsulated within the function body without explicit calls to other functional units. However, our investigation reveals a significant phenomenon of functional logic layering within the implementation architecture of standard library functions, particularly when considering the diversity of compilation tool-chains, target platforms, and vendor-specific modifications. This phenomenon manifests as a multi-level function call structure, where standard library functions delegate core logic to subordinate functions through a call chain.

Through an extensive analysis of various firmware, we have categorized the multi-level call structures into two distinct patterns: Wrapper of Standard Function and Private Helper Function Call.

- Wrapper of Standard Function Pattern: In this model, a high-level standard library function accomplishes its task by invoking another, often lower-level, standard library function. A clear example is illustrated in Fig.2, where the atoi function delegates its core string-to-integer conversion logic to strtol. Both atoi and strtol are standard interface functions defined by the ISO/IEC 9899 C standard, representing a canonical, intra-library dependency.
- Private Helper Call Pattern: This pattern occurs when a standard library function's implementation relies on non-standard, private helper functions to execute its core logic. For instance, a given memset implementation, while exposing a standard interface, may internally call a platform-optimized function such as bfill to perform the actual memory-filling operation. Here, bfill is not defined by the ISO/IEC 9899 C standard; instead, it is a private symbol specific to the particular compiler or hardware vendor's implementation, designed for efficiency or to access specialized hardware features.

For the first pattern, such as atoi delegating to strtol, the structural features of the caller function's CFG are diminished. The atoi function is reduced to a simple wrapper, and its GNN-based embedding may fail to capture the complete semantic context provided by the callee (strtol), leading to inaccurate similarity scores.

More critically, the Private Helper Call Pattern fundamentally undermines these methods. When a standard function like memset offloads its core logic to a private helper like bfill, a phenomenon of semantic displacement occurs. Semantic displacement refers to the phenomenon where the core functional logic of a standard library function (e.g., memory filling in memset) is transferred to a non-standard private helper function (e.g., bfill), breaking the semantic link between the standard function's interface and its observable implementation. The primary identifying logic—in this case, the memory-filling loop—is no longer in memset's body. Consequently, the CFG of memset becomes structurally sparse, consisting

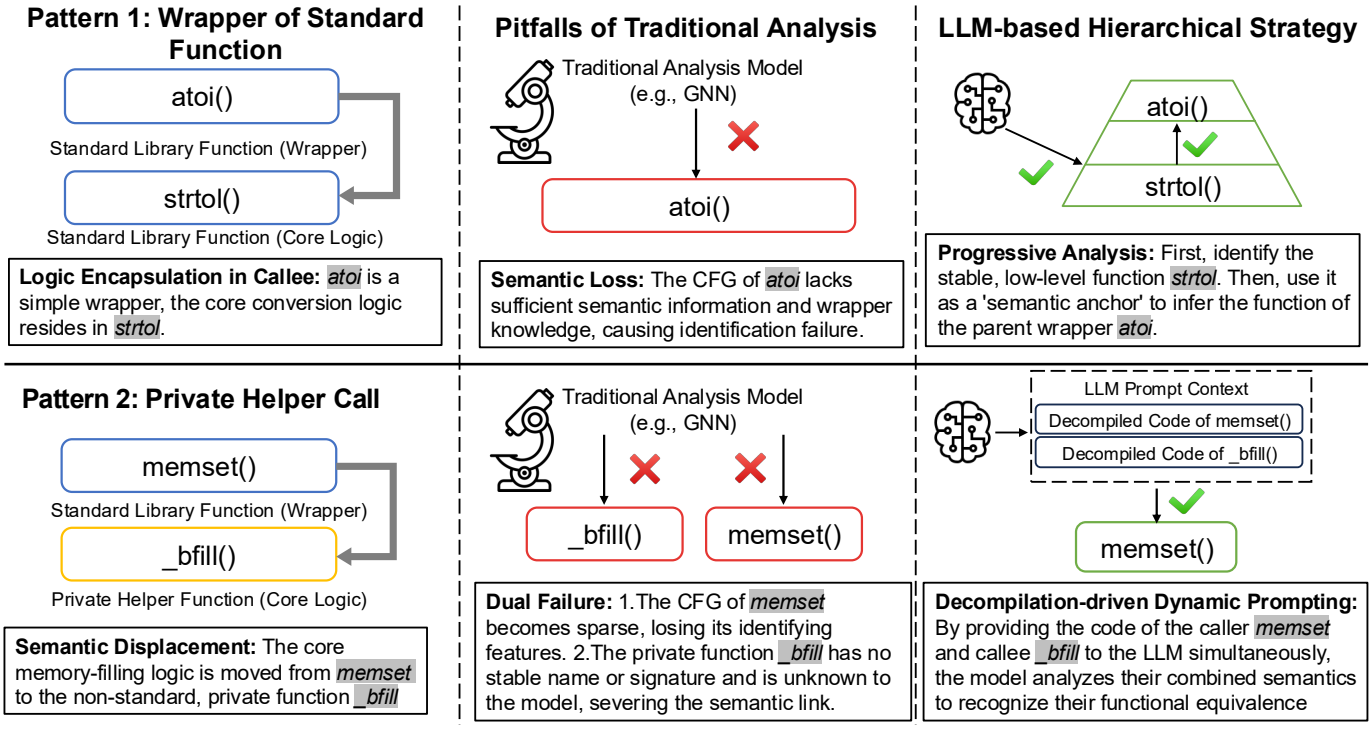

Fig. 2: Illustration of Semantic Displacement in Library Call Patterns and Our Proposed LLM-based Hierarchical Solution

mainly of argument preparation and a single call instruction. A model analyzing this sparse CFG cannot extract the features characteristic of a memory-setting operation. Furthermore, the helper function bfill itself is problematic: as a non-standard, private symbol, it lacks a consistent name or binary signature across different compilers or vendors. Machine learning models, trained on known library functions, are unable to recognize these arbitrary, unseen helper functions. This severs the semantic link between the standard interface (memset) and its core logic (bfill), making it exceedingly difficult to identify the memset function correctly.

To address these challenges, this study proposes a hierarchical identification strategy based on Large Language Models. For function call chains conforming to the ISO/IEC 9899 C standard, we employ a progressive analysis method. We first use a disassembler to identify leaf-node functions in the call tree (e.g., strtol) that contain no external calls, as these functions typically exhibit stable binary implementation patterns. These identified functions then serve as anchors to infer the functional logic of their callers (e.g., atoi). For the Private Helper Function Calls, we design a decompilation-driven dynamic prompting mechanism. We inject the decompiled code snippets of both the target function and its called private functions as context into the LLM prompt. This hierarchical association of code semantics enhances cross-layer inference. For example, to identify the private function bfill, the prompt template is embedded with the decompiled code of both the standard function memset and bfill (including parameter passing patterns and loop control structures). This guides the model to recognize their functional equivalence based on the topological similarity of their code semantics (e.g., memory block iteration logic, termination condition checks), rather than relying on the single-layer semantics of one function. This approach, through the explicit semantic representation of decompiled code, enables the LLM to penetrate the syntactic noise introduced by compiler optimizations and architectural differences.

B. Adaptive Iterative Screening Algorithm

Recent studies, such as DeGPT [38], have shown that Large Language Models exhibit a significant advantage in single-function semantic understanding, achieving a high accuracy in generating comments for stripped binaries from disassembly on small-scale datasets. However, when confronted with the large set of functions in a firmware image, applying LLMs for exhaustive identification incurs prohibitive temporal and computational costs. Based on our experimental estimates, a full LLM analysis of a typical, small-scale STM32 router firmware (containing approximately 10,000 functions) would consume over 50 million tokens, excluding the cost of disassembly.

Our empirical study of RTOS firmware reveals systematic differences in code characteristics between standard library functions and application-layer functions. In terms of call patterns, library functions exhibit high-frequency invocation due to their foundational nature. Regarding control flow structure, they generally have shallow call dependencies. From an interface design perspective, their parameter complexity is significantly lower than that of typical application-layer functions. Finally, concerning spatial distribution, library functions often exhibit memory locality, being located at contiguous or proximate addresses. These differentiating features provide a quantifiable basis for constructing a pre-screening mechanism.

Based on these findings, we propose the Adaptive Iterative Screening Algorithm (AISA). This algorithm constructs a multi-modal fitness evaluation model that dynamically integrates static function features with memory spatial distribution properties to achieve iterative screening optimization. During the initialization phase, initial weights are assigned to call frequency, call depth, and address proximity. Address proximity is defined as the memory address difference between the current function and its nearest neighbor in the historically validated set. As the iteration progresses, the algorithm sequentially performs three phases: Candidate Generation, High-Fidelity Validation, and Weight Reconfiguration. First, it calculates a weighted fitness score for each function—by positively normalizing call counts and inversely normalizing call depth and address distance—and selects the top-ρ percentile as the candidate set. Subsequently, an LLM validator is used to filter out false positives, forming a high-confidence subset of standard functions. An early stopping mechanism is implemented to terminate the iteration when the overlap between the validation results of two consecutive rounds exceeds a predefined threshold η. Finally, the algorithm outputs the cumulative set of validated standard library functions.

The call degree metric quantifies the call frequency of a function:

$$\hat{\xi}_{\text{call}}(f_i) = \frac{\text{call\_count}(f_i)}{\max_{f \in \mathcal{F}} \text{call\_count}(f)} \tag{1}$$

The call depth metric represents the hierarchical level of a function in the call chain, with standard library functions typically residing at shallow levels:

$$\hat{\xi}_{\text{depth}}(f_i) = 1 - \frac{\text{call\_depth}(f_i)}{\max_{f \in \mathcal{F}} \text{call\_depth}(f)} \tag{2}$$

The address proximity metric measures the spatial clustering of a function with already validated standard library functions in memory:

$$\hat{\xi}_{\text{addr}}(f_i) = 1 - \frac{\min_{f_j \in \mathcal{V}_{\text{prev}}} |\text{addr}(f_i) - \text{addr}(f_j)|}{\max_{f \in \mathcal{F}} \xi_{\text{addr}}(f)}, \quad \mathcal{V}_{\text{prev}} \neq \emptyset \tag{3}$$

where V_t is the set of validated functions at iteration t. Based on these metrics, we construct a linear weighted evaluation model:

$$\phi(f_i) = w_{\text{call}} \cdot \hat{\xi}_{\text{call}} + w_{\text{depth}} \cdot \hat{\xi}_{\text{depth}} + w_{\text{addr}} \cdot \hat{\xi}_{\text{addr}} \tag{4}$$

The convergence of AISA is theoretically guaranteed by the finiteness of the firmware function space and the design of the iterative mechanism. The process is driven by a positive feedback loop based on the memory spatial locality hypothesis: as the set of validated functions $\mathcal{V}_{\text{prev}}$ expands, the $\hat{\xi}_{\text{addr}}$ metric dynamically reinforces the fitness scores of spatially adjacent functions, accelerating the identification of function clusters. However, as clusters are fully identified, the marginal gain of the proximity feature diminishes, leading the algorithm into a saturation phase. To quantify this stability, we employ a convergence threshold η that monitors the overlap ratio between consecutive high-fidelity validation sets ($|\mathcal{V}_t \cap \mathcal{V}_{\text{prev}}|/|\mathcal{V}_t|$). A ratio exceeding η indicates that the high-probability search space has been exhausted. This dynamic stopping criterion, combined with the hard computational limit $T_{\max}$, effectively prevents infinite loops in worst-case scenarios.

Algorithm 1: Adaptive Iterative Screening Algorithm

Input : Function set $\mathcal{F} = \{f_1, ..., f_n\}$
Output: Validated stdlib function set $\mathcal{S}^*$
Initialize
  weights w
  Selection ratio ρ
  Max iterations $T_{\max}$
  Convergence threshold $\eta \in (0, 1)$
for $t \leftarrow 1$ **to** $T_{\max}$ **do**
  foreach $f_i \in \mathcal{F}$ **do**
    Calculate $\phi(f_i)$ using Equation 3;
  Select $\mathcal{C}_t$ = Top-ρ functions by ϕ;
  $\mathcal{V}_t \leftarrow \{f \in \mathcal{C}_t \mid \text{LLMVALIDATOR}(f) = \text{True}\}$;
  if $\frac{|\mathcal{V}_t \cap \mathcal{V}_{prev}|}{|\mathcal{V}_t|} > \eta$ **then**
    Break loop;
  $\mathcal{V}_{\text{prev}} \leftarrow \mathcal{V}_t$;
  $\mathcal{S}^* \leftarrow \mathcal{S}^* \cup \mathcal{V}_t$;
return $\mathcal{S}^*$;

The core innovation of Algorithm 1 is the introduction of the memory spatial locality hypothesis, quantified by the Address Proximity Feature. This feature is automatically disabled during the initial iterations and is activated only when the historically validated set V_t becomes non-empty. Experiments at IV confirm that this method effectively reduces the computational cost of LLM validation while maintaining a high recall rate for standard library function identification.

To operationalize our empirical findings, we constructed an automated pipeline for firmware pre-processing and feature extraction. This pipeline is built upon the Ghidra reverse engineering framework [41], chosen for its powerful decompilation engine and comprehensive scripting API. For seamless integration into existing workflows and to maximize utility for practitioners, we packaged the entire pipeline as a self-contained Ghidra plugin. The plugin's core logic, implemented via a Python script, automates the end-to-end process: it programmatically loads a target firmware binary and invokes Ghidra's auto-analyzer for disassembly and call graph generation. Subsequently, it iterates through each identified function to extract a predefined vector of structural features, including call in-degree, maximum call dependency depth, argument count, absolute memory address, and the complete C-like decompiled source. All extracted data is then serialized into a structured format, creating a dataset ready for subsequent analytical phases.

C. RAG-enhanced LLM Validator

In the challenging task of identifying functions within decompiled binary code, a significant "semantic gap" exists between the high-level conceptual purpose of the code and the output of decompilation tools. The domain knowledge limitations of a general-purpose Large Language Model often

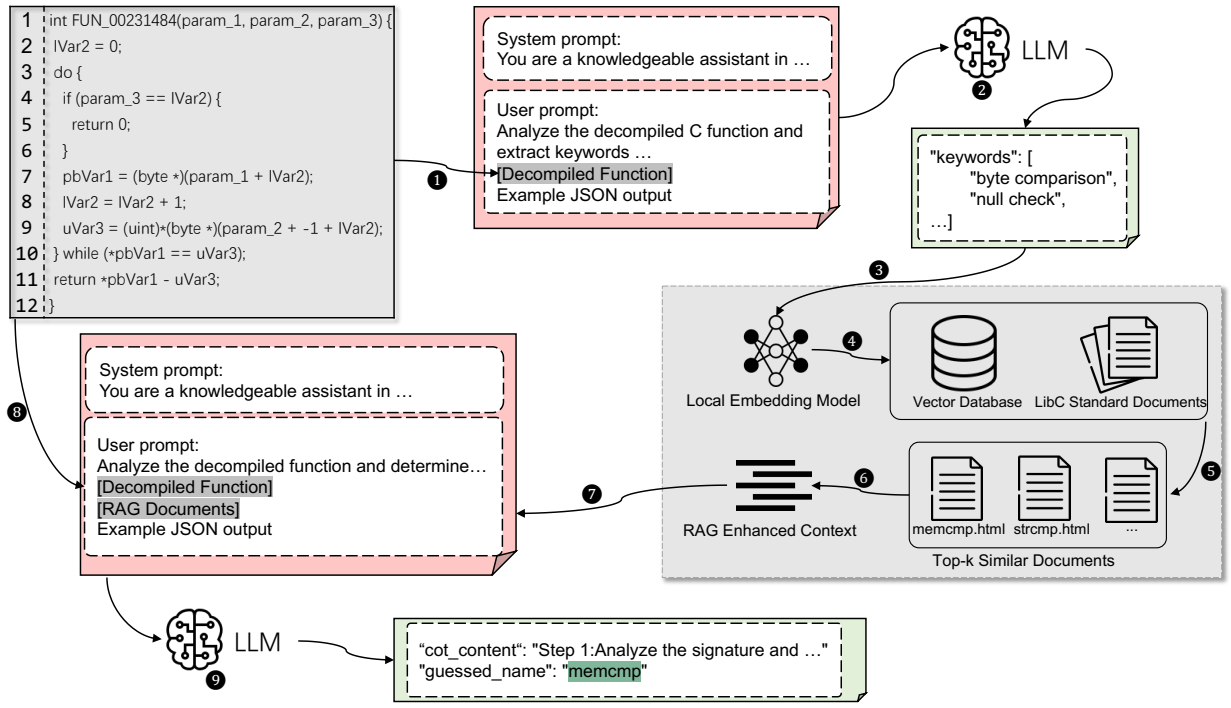

Fig. 3: The RAG-enhanced LLM Validator Workflow for Function Identification in Decompiled Code

prevent it from bridging this gap, which can compromise its analytical accuracy [42]. To systematically address this issue, our study implements and refines a Retrieval-Augmented Generation (RAG) framework. This framework enhances the model's domain-specific expertise by dynamically grounding its reasoning in a meticulously constructed, specialized knowledge base.

The foundation of our RAG system is a high-quality knowledge base derived from the official offline archives of the C standard library documentation on cppreference [43]. To ensure robust coverage for diverse RTOS firmware environments—ranging from legacy codebases to modern implementations—we selected documentation covering the full spectrum of major C standards, including C89/90, C95, C99, and C11. This comprehensive inclusion criterion mitigates the risk of knowledge omission regarding standard-specific library behaviors. The content includes critical, structured information such as function prototypes, detailed parameter descriptions, return value definitions, and illustrative code examples. To distill this raw data into a functional knowledge repository, our pre-processing pipeline executed several key steps on the 554 source HTML documents. This involved aggressive text cleaning and tag stripping to remove navigational artifacts and non-essential content, followed by semantic paragraph segmentation. This process yielded a multi-dimensional knowledge repository focused purely on the technical descriptions of C standard library functions. This clean corpus was then transformed into dense vector representations using the bge-base-en-v1.5 embedding model [44]. These embeddings were subsequently indexed in a FAISS [45] vector database, which utilizes a flat L2 index for highly efficient nearest-neighbor searches.

The function identification process, as shown in Fig.3, is executed through a precise, multi-stage workflow.

(1) Initially, the pipeline ingests raw C-like pseudo-code from Ghidra, which is pre-processed to normalize whitespace and remove decompiler artifacts.

(2) A Large Language Model then analyzes this normalized code, identifying salient features like constants usage, system call sequences, and data manipulation patterns and distilling this analysis into a set of semantically-rich keywords.

(3) Subsequently, these keywords are encoded into a vector embedding by a local sentence-transformer model.

(4) This vector is then used as a query against a knowledge base of API documentation and function descriptions, which has been pre-indexed using a FAISS IndexFlatL2 structure for exact similarity search.

(5) The FAISS engine calculates the cosine similarity between the query and all document vectors, returning the top-three candidate documents; this k-value (k=3) was empirically determined to optimize the balance between contextual richness and token-window constraints.

(6) These three retrieved documents are then formatted, truncated, and concatenated into a single text block, forming the RAG enhanced context.

(7) A final, composite prompt is constructed using a structured template that specifies the primary task instruction.

(8) This prompt incorporates the original decompiled code within [Decompiled Function] tags and the RAG-enhanced context within [RAG Documents] tags.

(9) Finally, this complete prompt is submitted to a powerful, instruction-tuned generative LLM, such as GPT-4o or Gemini 2.5 Flash, with a low temperature setting to ensure deterministic and factual output. The LLM performs a comparative

analysis, referencing the decompiled code's logic with the retrieved documentation to determine algorithmic and semantic equivalence.

The model's output is structured into a standardized JSON format, which includes the identified function name and a reasoning chain to provide machine-readable results for automated downstream analysis.

IV. Evaluation

This section is organized into three parts. First, we assess the accuracy of the LLM function validator. Second, we evaluate the overall framework, with a focus on quantifying the efficiency gains provided by the AISA algorithm. Third, we present case studies conducted on real-world RTOS firmware. Table I situates our work in contrast to prior art

TABLE I: Comparison of the proposed method with existing approaches

Method	Compiler/Toolchain Independent	Cross-Architecture Support
IDA FLIRT	No	No
Asm2vec	No	No
GNN (s2v)	No	Yes
Our Method	**Yes**	**Yes**

by comparing key operational requirements. A significant bottleneck for widely-used methods like IDA FLIRT and Asm2vec is their dual dependency on both a known compiler toolchain and a specific hardware architecture. While GNN-based approaches represent a step forward by offering cross-architecture capabilities, they do not eliminate the need for compiler-specific information. The primary advantages of our method, as shown in the table, are its complete independence from the compilation environment and its inherent support for cross-architecture firmware analysis. This makes our approach uniquely suited for analyzing stripped, real-world firmware where such metadata is often unavailable.

TABLE II: Firmware Compiled for 9 Different Architectures Using Zephyr RTOS

Platform	CPU	ISA Family	Endian
STM32H747	ARMv7-M	Variable (16/32b)	LE
BCM2711	ARMv8-A	Fixed 32-bit	LE
QEMU RV32	RISC-V	Fixed 32-bit (+16-bit C)	LE
SiFive FU740	RISC-V	Fixed 32-bit (+16-bit C)	LE
QEMU Malta	MIPS32	Fixed 32-bit	LE
LEON3 FPGA	SPARC V8	Fixed 32-bit	BE
MIMXRT595	Xtensa	Variable (16/24-bit)	LE
QEMU Atom	x86 (IA-32)	Variable (1-15 bytes)	LE
Raptor Lake S	x86-64	Variable (1-15 bytes)	LE

Our evaluation centers on the Zephyr Real-Time Operating System [46]. Its open-source nature and cross-architecture compatibility, which spans mainstream instruction sets such as ARM, MIPS, and RISC-V, provide an ideal testbed for validating the cross-platform analysis capabilities of our method. We utilized the official Zephyr SDK 0.17.0 build environment. This SDK provides a stable cross-compilation toolchain based on GCC 12.2.0 for all supported target architectures. We consistently applied the SDK's default optimization level of `-Os` (Optimize for Size) for all firmware builds. We selected nine representative architectures (as detailed in Table II) that cover a diverse hardware ecosystem, from microcontrollers (e.g., STM32H747) to high-performance processors (e.g., Raptor Lake S). An independent compilation toolchain was configured for each architecture to ensure the distinctness of the resulting function binaries, thereby comprehensively testing the model's generalization ability across different firmware builds. A key advantage of our proposed method is that it does not require prior knowledge of the compiler or toolchain used for the target firmware and achieves cross-architecture versatility.

To construct a representative ground truth for evaluation, we established a curated test set. The positive samplesV consist of 131 standard library function instances, which represent the cumulative total of standard library functions identified across the nine Zephyr firmware images. On average, each firmware image contains approximately 15 identifiable standard library functions. Collecting these instances across distinct architectures ensures our dataset captures the implementation diversity caused by different Instruction Set Architectures (ISAs) and compilation contexts. The specific selection of these functions was driven by two key criteria: statistical prominence and security relevance. We prioritized functions that exhibit high prevalence across architectures (e.g., `memcpy`, `strlen`) and those critically relevant to security auditing (e.g., `strcpy`, which is prone to buffer overflows). A detailed statistical profile of these samples, including their call frequencies and prevalence, is provided in the Appendix C.

Correspondingly, the set of non-standard functions VI (negative samples) was created by randomly sampling functions from the Zephyr system's driver layer and kernel API. These were filtered to include only functions with no external calls and instruction lengths comparable to the positive samples, ensuring that both sets were structurally analogous.

We evaluate the model's performance on two levels. At the macro-level, the task is defined as a binary classification problem to determine if the model can accurately distinguish between "library functions" and "non-library functions." This assesses the model's fundamental classification capability. At the micro-level, for samples correctly identified as library functions, the task is elevated to a multi-class classification problem to evaluate if the model can precisely predict the specific function name. This level directly measures the model's fine-grained identification accuracy.

A. RQ1: Accuracy of Library Function Classification

For this evaluation, we established a controlled experiment with two configurations: a baseline LLM validator (Base) and our Retrieval-Augmented Generation enhanced validator (RAG). The models evaluated include Gpt-4o, Gemini 2.5 Flash, Deepseek-V3, and Qwen3-235B-A22B.

To ensure reproducibility and prevent information loss, the simulation parameters were rigorously configured. Specifically, **to prioritize deterministic generation and minimize hallucinations, the temperature was set to 0.05 and nucleus sampling (top-p) was set to 0.95.** The maximum token limit was established at 8,000. This value is well within the context

window specifications of the employed LLMs (e.g., 128k for GPT-4o [47] and 1M+ for Gemini [48]). Furthermore, statistical analysis of our dataset indicated that the combined length of the decompiled function code and the top-3 retrieved RAG documents typically ranges between 1,000 and 6,000 tokens. Thus, the 8,000-token limit serves as a sufficient upper bound to prevent the truncation of critical semantic context during inference.

TABLE III: Performance Metrics of LLMs with Base Validator and RAG-enhanced Validator

Method	Model	Precision	Recall	F1-score
Base	Gpt-4o	0.936 ± 0.013	1.000	0.967 ± 0.007
	Gemini 2.5 Flash	0.970	1.000	0.985
	DeepSeek-V3	0.916	1.000	0.956
	Qwen3-235B	0.942 ± 0.010	1.000	0.970 ± 0.005
RAG	Gpt-4o	0.931 ± 0.015	1.000	0.964 ± 0.008
	Gemini 2.5 Flash	0.952 ± 0.013	0.997 ± 0.005	0.974 ± 0.009
	DeepSeek-V3	0.931 ± 0.023	0.989 ± 0.011	0.959 ± 0.011
	Qwen3-235B	0.970	0.985	0.977

Performance was measured using metrics derived from the confusion matrix: True Positives (TP, correct identification of standard library functions), False Positives (FP, misclassification of non-standard functions as standard), True Negatives (TN, correct exclusion of non-standard functions), and False Negatives (FN, missed detection of standard library functions). The F1-score was used as the primary evaluation metric.

At the macro-level evaluation (RQ1), all participating LLMs demonstrated excellent foundational identification capabilities. In the Base validator configuration, every model achieved a Recall of 1.000, indicating that they successfully identified all standard library functions in the test set without any false negatives (FN=0). Among them, Gemini 2.5 Flash delivered the best performance on the binary classification task. The introduction of RAG resulted in minor performance fluctuations, but the overall accuracy remained high. In summary, the LLMs employed in our method exhibit robust performance in this binary classification task.

B. RQ2: Accuracy of Function Name Identification

At this level, we assess the accuracy of specific function name prediction, treating the task as a multi-class classification problem. We use Overall Accuracy (the proportion of predictions, including both specific function names and a "None" category, that exactly match the ground truth labels) and per-class Precision, Recall, and F1-score.

The results show that the accuracy of the RAG method remained consistently high, within the 88.80% to 90.59% range, demonstrating excellent performance stability. In stark contrast, the accuracy of the baseline method was both lower and more volatile, ranging from 75.57% to 84.22%. More importantly, the RAG method achieved significant relative performance gains, improving accuracy over the baseline by 6% on Qwen3, 14% on Gpt-4o, 10% on Deepseek-V3, and 20% on Gemini 2.5 Flash. These results affirm the universal efficacy of our RAG-based approach across different LLM

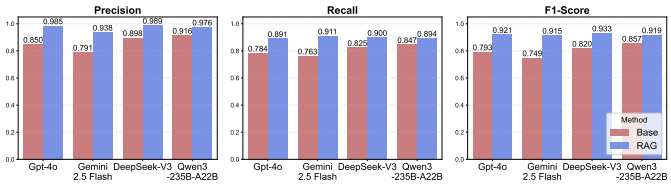

Fig. 4: Performance Metrics for Base Validator and RAG-enhanced Validator

platforms and establish its definitive superiority in overall performance.

A core limitation of the baseline method was its consistently low recall, which led to a high rate of false negatives. For instance, with the Base Gpt-4o setup, the recall for memchr was merely 0.222, and for strrchr on DeepSeek-V3, it was as low as 0.037. Our RAG-enhanced method successfully overcomes this bottleneck. While maintaining or even improving precision, our method dramatically boosts recall, frequently raising it to high levels. This improvement is critical, as it demonstrates a significantly enhanced ability to capture target functions without introducing additional false positives.

Our method also showcases superior discriminative power when handling difficult cases with high semantic similarity or functional overlap. For confusable function pairs like memcmp/strncmp and strchr/strrchr, where the baseline's performance was highly unstable or failed completely, our method achieved stable and high-precision identification. It also achieved a notable breakthrough in identifying the atoi function, which proved challenging for the baseline across multiple models.

Additionally, we analyzed the token consumption of different LLMs using identical prompts and parameters, conducting three repeated trials for each. Under the baseline method,

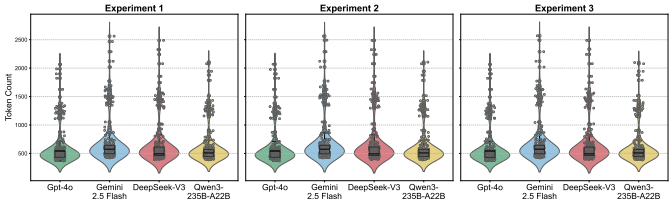

Fig. 5: Token Consumption Distribution for a Single Query with the Base LLM Validator

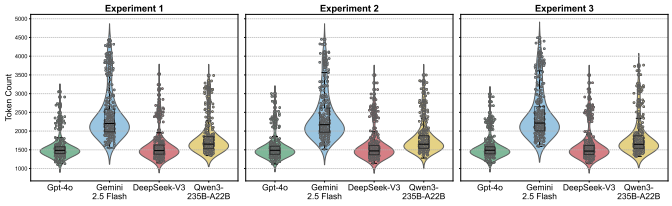

Fig. 6: Token Consumption Distribution for a Single Query with the RAG-enhanced Validator

all models operated at a low token level, demonstrating good computational efficiency with median token counts generally

TABLE IV: Multi-Class Classification Results for Specific Function Identification

Class	Model	C Standard Library Function (Correctly Identified)															Overall Accuracy (%)
		memset	memcpy	memmove	memcmp	memchr	strlen	strcpy	strncpy	strcmp	strncmp	strchr	strrchr	strstr	atoi	strtoul	
Base	Gpt-4o	21	21	27	24	6	26	27	27	24	18	27	16	24	0	18	77.86
	Gemini 2.5 Flash	21	24	27	27	27	24	27	20	27	0	27	10	24	0	12	75.57
	DeepSeek-V3	21	24	27	24	24	24	27	24	27	18	27	1	24	3	27	81.93
	Qwen3-235B-A22B	21	23	27	9	27	24	27	20	27	20	27	6	24	23	26	84.22
RAG	Gpt-4o	20	24	27	23	27	26	27	22	27	27	27	18	24	9	21	88.80(+0.14)
	Gemini 2.5 Flash	21	24	27	27	27	23	27	27	27	21	27	26	24	1	27	90.59(+0.20)
	DeepSeek-V3	20	23	27	27	22	15	26	27	27	25	27	23	24	14	26	89.82(+0.10)
	Qwen3-235B-A22B	21	24	27	23	27	23	27	24	27	25	27	27	24	10	14	89.06(+0.06)

between 300 and 600. Upon switching to the RAG method, token consumption increased by a factor of two to three. This is a predictable computational overhead introduced to achieve higher identification accuracy, representing the inherent cost of encoding additional knowledge into the prompt. RAG also amplified the token efficiency differences among models. Notably, Gemini 2.5 Flash exhibited a median and range of token consumption significantly higher than the other three models, whose consumption, while increased, grew more moderately.

C. RQ3: Algorithm Efficiency on RTOS Firmware

This evaluation assesses whether the AISA method can effectively increase identification speed and reduce token costs. Before conducting the broad efficiency evaluation, we first validated the rationale behind the specific weight assignment in AISA ($w_{call} = 0.3, w_{depth} = 0.3, w_{addr} = 0.4$) through an ablation study.

We selected three representative firmware images (ARM, MIPS, and RISC-V) and evaluated the screening efficiency under five different weight configurations:

- *Call-only*: $w_{\text{call}} = 1.0$, $w_{\text{depth}} = 0.0$, $w_{\text{addr}} = 0.0$
- *Depth-nly*: $w_{\text{call}} = 0.0$, $w_{\text{depth}} = 1.0$, $w_{\text{addr}} = 0.0$
- *Call+Depth*: $w_{\text{call}} = 0.5$, $w_{\text{depth}} = 0.5$, $w_{\text{addr}} = 0.0$
- *Uniform*: $w_{\text{call}} = 1/3$, $w_{\text{depth}} = 1/3$, $w_{\text{addr}} = 1/3$
- *Ours (Proposed)*: $w_{\text{call}} = 0.3$, $w_{\text{depth}} = 0.3$, $w_{\text{addr}} = 0.4$

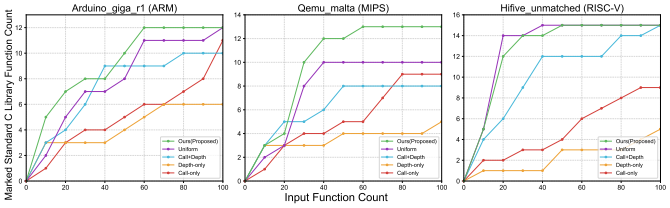

Fig. 7: Ablation study of AISA weight configurations on ARM, MIPS, and RISC-V firmware

As shown in Fig.7, the results confirm the necessity of a multi-feature approach. Single-feature baselines consistently exhibited the lowest performance. Crucially, the inclusion of the Address Proximity feature proved vital; the "*Proposed*" configuration significantly outperformed the "*Call+Depth*" variant on ARM and MIPS architectures. While the "*Proposed*" and "*Uniform*" settings yielded comparable results on the RISC-V platform, the "*Proposed*" configuration demonstrated superior generalizability across all tested architectures, justifying its selection as the default setting.

Leveraging this empirically optimized configuration, we then conducted the primary efficiency evaluation on the nine Zephyr-based firmware images. The control group employed a baseline heuristic strategy that ranks functions by their call frequency, an intuitive approach assuming that more frequently called functions are more likely to be library functions. For AISA, the weights for call frequency, call depth, and address proximity were set to 0.3, 0.3, and 0.4, respectively, with GPT-4o as the backend LLM. For each firmware, we measured the number of identified standard library functions within the first 100 function inputs.

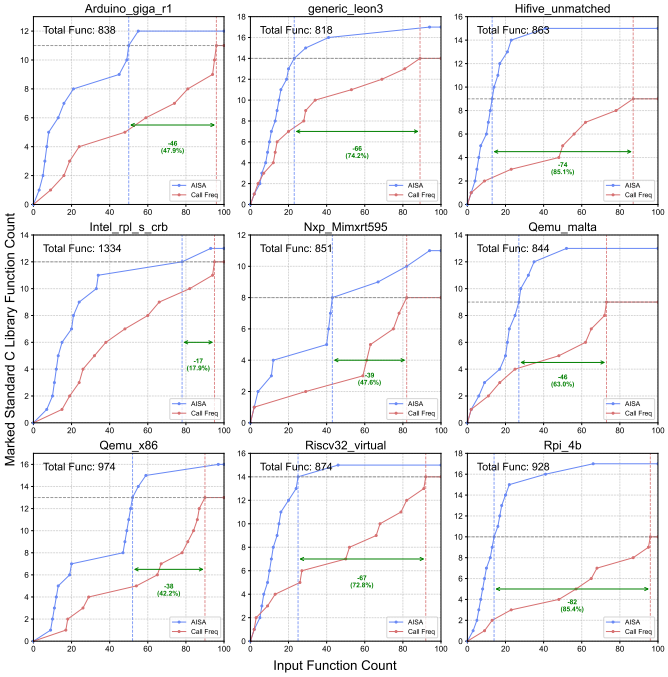

Fig. 8: Performance Comparison of AISA and Call Frequency Ranking on Zephyr Firmware

As shown in Fig.8, AISA demonstrates substantial efficiency improvements across all nine Zephyr firmware architectures. The dashed lines and arrows illustrate the round reduction achieved by AISA when reaching the same identifi-

cation milestone as the baseline method. Quantitatively, AISA reduces the average number of required analysis rounds by 52.78, representing a 59.56% efficiency improvement. The most significant gains were observed in Hifive_unmatched (85.1% reduction, 74 rounds saved) and Rpi_4b (85.4% reduction, 82 rounds saved), while even the most modest improvement in Intel_rpl_s_crb still achieved 17.9% efficiency gain (17 rounds saved).

To further validate the generalizability of AISA across firmware of varying types and complexities, we tested it on three commercial IoT devices that use RTOS: a STM32 telematics box, a TP-Link router (WDR-7660), and a Mercusys router (A12G).

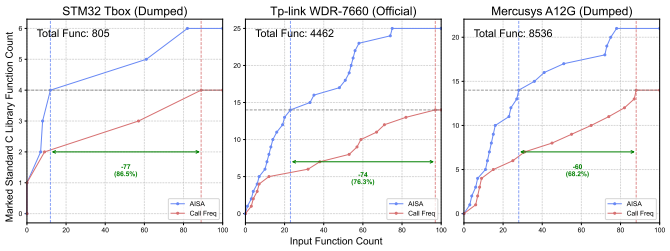

Fig. 9: Performance Comparison of AISA and Call Frequency Ranking on Commercial IoT Firmware

The results on commercial firmware, shown in Fig. 9, demonstrate consistent efficiency gains with AISA. The STM32 Tbox showed an 86.5% improvement (77 rounds saved), the TP-Link WDR-7660 demonstrated a 76.3% improvement (74 rounds saved), and the Mercusys A12G exhibited a 68.2% improvement (60 rounds saved), averaging 77.0% efficiency improvement across the three devices. These results provide encouraging evidence that AISA's adaptive screening strategy generalizes effectively to real-world commercial firmware with diverse implementations and complexities.

The Adaptive Iterative Screening Algorithm (AISA) demonstrated significantly superior identification efficiency on real-world firmware compared to the traditional call-frequency ranking method. In all test cases, AISA's performance curve was consistently positioned above and to the left of the baseline, indicating that for the same analysis cost (i.e., the same number of functions fed to the LLM), AISA identifies more standard library functions. Conversely, to achieve the same identification goal, AISA requires far fewer function inputs than the baseline. This superior efficiency directly translates to faster identification speeds and substantial reductions in token costs. By intelligently prioritizing high-value candidates, AISA achieves a higher "hit rate" early in the analysis process.

V. Discussion

While our methodology represents a significant advancement in automatically identifying library functions, verifying its performance in complex, real-world compilation environments is crucial. In this section, we quantify the robustness of our approach against compiler optimizations and discuss the remaining challenges and future directions.

A. Robustness Analysis: Impact of Compiler Optimization

A primary concern in binary analysis is the sensitivity of the method to compiler optimizations, which can drastically alter instruction patterns. In the context of RTOS development, firmware is typically generated via cross-compilation using official SDKs, where standard libraries are pre-built with specific optimization defaults (usually `-Os` for size). To systematically quantify the robustness of our approach, we positioned our primary evaluation (based on the standard `-Os` configuration) against two extreme configurations. We explicitly constructed custom cross-compilation toolchains using the **crosstool-ng** build system (hosted on **Ubuntu 22.04** with **GCC 11.4.0**). These toolchains were configured to generate firmware with **`-O0` (No Optimization)** and **`-O3` (Aggressive Optimization)**, thereby establishing a **broad range** of optimization intensity. We evaluated the RAG-enhanced Validator using the Gemini model across these scenarios.

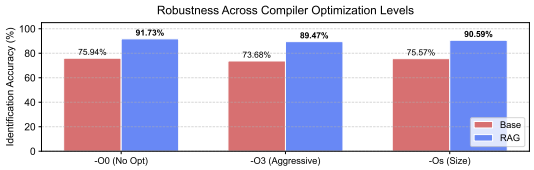

Fig. 10: **Robustness Across Compiler Optimization Levels.** The default optimization level in the official toolchain is `-Os`.

The results, as illustrated in Fig. 10, demonstrate remarkable stability across this broad range. The accuracy follows a logical and gradual trend: peaking at **91.7%** for the unoptimized `-O0` build, remaining high at **90.6%** for the standard `-Os`, and sustaining **89.4%** even under aggressive `-O3` optimization. The marginal drop of only 2.3% from `-O0` to `-O3` confirms that our semantic-based approach is largely immune to the low-level instruction shuffling and register reallocation typical of aggressive optimizations in pre-built libraries. Furthermore, across all levels, the RAG-enhanced model consistently outperforms the Base model (which hovers around 73-76%), highlighting that external knowledge is the critical factor in bridging the semantic gap widened by optimization.

B. Limitations and Future Directions

Despite the demonstrated robustness, our approach faces specific limitations. First, while resistant to instruction-level optimization, extensive function inlining fundamentally erases function boundaries. To address this structural loss, future work will explore sub-function chunk analysis, training LLMs to identify inlined logic patterns within larger function bodies. Second, analyzing vendor-specific or proprietary libraries without documentation requires shifting toward few-shot or zero-shot learning paradigms. Concurrently, to ensure economic viability for large-scale analysis, research into model distillation and smaller, domain-specific language models remains essential to provide more scalable solutions [49].

A logical progression for this research is to transcend mere function identification and pursue holistic program understanding for security purposes. The immediate next step involves

analyzing the calling context of identified high-risk functions to automatically detect vulnerabilities like buffer overflows. This requires enabling the LLM to perform localized data-flow analysis. Ultimately, by fusing the semantic power of LLMs with structural representations like Control Flow Graphs, we envision transforming this framework from a labeling tool into an end-to-end collaborative partner for automated vulnerability discovery and security auditing.

VI. Conclusion

This paper addresses the persistent challenge of identifying standard library functions in stripped RTOS embedded firmware. Unlike traditional static analysis techniques that rely on rigid syntactic signatures or graph matching—which often struggle with the diversity of architectures and compiler optimizations inherent in the RTOS ecosystem—our work introduces a novel automated analysis method based on Large Language Models. Furthermore, distinct from existing LLM-based reverse engineering works that predominantly rely on the model's implicit internal knowledge for direct inference, we establish a unique technical route based on Retrieval-Augmented Generation (RAG). By incorporating an external knowledge base, our method effectively bridges the gap between general model capabilities and specific binary analysis needs. This approach achieves a high identification accuracy of 90.59% on a test set spanning nine mainstream architectures, representing an average performance improvement of 12.5% over baseline approaches.

To further enhance analytical efficiency, we distinctively address the high resource overhead inherent in LLM interactions. Unlike prior LLM-based studies that typically apply a uniform analysis depth without specific mechanisms for cost optimization, we innovatively propose the Adaptive Iterative Screening Algorithm (AISA), which implements a hierarchical optimization strategy. Instead of exhaustively evaluating every candidate with full complexity, AISA transforms the identification task into a coarse-to-fine process. Through intelligent candidate prioritization, this layered strategy substantially reduces the token consumption of the LLM, enabling the attainment of identification goals at a significantly lower cost, as validated on real-world firmware.

Despite the demonstrated advantages of our method in accuracy, generalizability, and economy, we acknowledge that its performance is ultimately constrained by the capabilities of the underlying LLM. The approach remains sensitive to complex scenarios such as highly optimized code logic, heavy code obfuscation, and function inlining. Future work will focus on enhancing the model's robustness in comprehending complex code semantics and its resilience in adversarial scenarios. We also plan to explore multi-modal binary analysis frameworks and extend our method's applicability to a broader range of code reverse engineering and security analysis tasks.

In conclusion, by integrating RAG-based knowledge enhancement with the AISA hierarchical optimization strategy, this paper provides an efficient, versatile, and cost-effective automated solution for the analysis of RTOS embedded firmware. To foster reproducible research, we will make our code publicly available. This work stands to accelerate firmware functional comprehension and security auditing, offering broad application prospects in critical fields such as vulnerability discovery and reverse engineering.

References

[1] S. Jero, J. Furgala, R. Pan, P. K. Gadepalli, A. Clifford, B. Ye, R. Khazan, B. C. Ward, G. Parmer, and R. Skowyra, "Practical principle of least privilege for secure embedded systems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 1–13.

[2] R. Mishra and A. Mishra, "Current research on internet of things (iot) security protocols: A survey," *Computers & Security*, p. 104310, 2025.

[3] X. Feng, X. Zhu, Q.-L. Han, W. Zhou, S. Wen, and Y. Xiang, "Detecting vulnerability on iot device firmware: A survey," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 1, pp. 25–41, 2022.

[4] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 392–404, Nov. 2018.

[5] A. Qasem, M. Debbabi, B. Lebel, and M. Kassouf, "Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures," in *Proceedings of the ACM Asia Conference on Computer and Communications Security*, Jul. 2023, pp. 443–456.

[6] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Probabilistic naming of functions in stripped binaries," in *Annual Computer Security Applications Conference*, Dec. 2020, pp. 373–385.

[7] D. Gomes, E. Felix, F. Aires, and M. Vieira, "Static code analysis for iot security: A systematic literature review," *ACM Computing Surveys*, p. 3745019, Jun. 2025.

[8] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society.

[9] C. Zhu, Z. Li, A. Xue, A. P. Bajaj, W. Gibbs, Y. Liu, R. Alur, T. Bao, H. Dai, A. Doupé *et al.*, "{TYGR}: Type inference on stripped binaries using graph neural networks," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4283–4300.

[10] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with bert-based transferable similarity learning," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 361–374.

[11] X. Xu, Z. Zhang, Z. Su, Z. Huang, S. Feng, Y. Ye, N. Jiang, D. Xie, S. Cheng, L. Tan *et al.*, "Unleashing the power of generative model in recovering variable names from stripped binary," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025.

[12] X. Xie, J. Ye, L. Wu, and R. Li, "Rtosextracter: Extracting user-defined functions in stripped rtos-based firmware," in *2022 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, Oct. 2022, pp. 87–96.

[13] H. Kim, J. Bak, K. Cho, and H. Koo, "A transformer-based function symbol name inference model from an assembly language for binary reversing," in *Proceedings of the ACM Asia Conference on Computer and Communications Security*, Jul. 2023, pp. 951–965.

[14] Y. R. Siwakoti, M. Bhurtel, D. B. Rawat, A. Oest, and R. C. Johnson, "Advances in iot security: Vulnerabilities, enabled criminal services, attacks, and countermeasures," *IEEE Internet of Things Journal*, vol. 10, no. 13, pp. 11 224–11 239, 2023.

[15] N. Risse and M. Böhme, "Uncovering the limits of machine learning for automatic vulnerability detection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4247–4264.

[16] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2099–2116.

[17] A. Jia, M. Fan, X. Xu, W. Jin, H. Wang, and T. Liu, "Cross-inlining binary function similarity detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Apr. 2024, pp. 1–13.

[18] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jtrans: Jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, Jul. 2022, pp. 1–13.

[19] Z. Pan, T. Wang, L. Yu, and Y. Yan, "Position distribution matters: A graph-based binary function similarity analysis method," *Electronics*, vol. 11, no. 15, p. 2446, Jan. 2022.

[20] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2021, pp. 224–236.
[21] K. Joxean, "Diaphora," https://github.com/joxeankoret/diaphora, 2025.
[22] W. Zhang, Z. Xu, Y. Xiao, and Y. Xue, "Unleashing the power of pseudo-code for binary code similarity analysis," *Cybersecurity*, vol. 5, no. 1, p. 23, Dec. 2022.
[23] H. Wang, Z. Gao, C. Zhang, M. Sun, Y. Zhou, H. Qiu, and X. Xiao, "Cebin: A cost-effective framework for large-scale binary code similarity detection," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Sep. 2024, pp. 149–161.
[24] S. Jiang, C. Fu, S. He, J. Lv, L. Han, and H. Hu, "Bincola: Diversity-sensitive contrastive learning for binary code similarity detection," *IEEE Transactions on Software Engineering*, vol. 50, no. 10, pp. 2485–2497, Oct. 2024.
[25] Q. Song, Y. Zhang, B. Wang, and Y. Chen, "Inter-bin: Interaction-based cross-architecture iot binary similarity comparison," *IEEE Internet of Things Journal*, vol. 9, no. 20, pp. 20 018–20 033, Oct. 2022.
[26] I. Guilfanov, "Fast library acquisition for identification and recognition," https://docs.hex-rays.com/user-guide/signatures/flirt, Dec. 2024.
[27] Q. Jing, S. Xiaohong, and M. Peijun, "Library functions identification in binary code by using graph isomorphism testings," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Mar. 2015, pp. 261–270.
[28] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1661–1682, 2022.
[29] J. Carrillo-Mondéjar and R. J. Rodríguez, "Identifying runtime libraries in statically linked linux binaries," *Future Generation Computer Systems*, vol. 164, p. 107602, Mar. 2025.
[30] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 472–489, May 2019.
[31] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2017, pp. 363–376.
[32] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, Nov. 2020.
[33] Z. Sha, H. Wang, Z. Gao, H. Shu, B. Zhang, Z. Wang, and C. Zhang, "Llasm: Naming functions in binaries by fusing encoder-only and decoder-only llms," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–22, May 2025.
[34] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," *ACM Transactions on Intelligent Systems and Technology*, vol. 16, no. 5, pp. 1–72, Oct. 2025.
[35] X. Xu, Z. Zhang, Z. Su, Z. Huang, S. Feng, Y. Ye, N. Jiang, D. Xie, S. Cheng, and L. Tan, "Unleashing the power of generative model in recovering variable names from stripped binary," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025.
[36] H. Pearce, B. Tan, P. Krishnamurthy, F. Khorrami, R. Karri, and B. Dolan-Gavitt, "Pop quiz! can a large language model help with reverse engineering?" Feb. 2022.
[37] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, Dec. 2024, pp. 4554–4568.
[38] P. Hu, R. Liang, and K. Chen, "Degpt: Optimizing decompiler output with llm," in *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.
[39] P. Liu, C. Sun, Y. Zheng, X. Feng, C. Qin, Y. Wang, Z. Xu, Z. Li, P. Di, Y. Jiang, and L. Sun, "Llm-powered static binary taint analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, pp. 1–36, Mar. 2025.
[40] H. Jelodar, S. Bai, P. Hamedi, H. Mohammadian, R. Razavi-Far, and A. Ghorbani, "Large language model (llm) for software security: Code analysis, malware analysis, reverse engineering," Apr. 2025.
[41] NationalSecurityAgency, "Ghidra," https://www.nsa.gov/ghidra, Aug. 2025.
[42] J. C. Dos Santos Junior, R. Hu, R. Song, and Y. Bai, "Domain-driven llm development: Insights into rag and fine-tuning practices," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Aug. 2024, pp. 6416–6417.
[43] Cppreference.com. [Online]. Available: https://en.cppreference.com/index.html
[44] S. Xiao, Z. Liu, P. Zhang, N. Muennighoff, D. Lian, and J.-Y. Nie, "C-pack: Packed resources for general chinese embeddings," in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, Jul. 2024, pp. 641–649.
[45] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," Feb. 2025.
[46] Zephyr Project, "Zephyr project," https://www.zephyrproject.org/, 2025.
[47] Model - openai api. [Online]. Available: https://platform.openai.com
[48] G. Comanici, E. Bieber, M. Schaekermann, I. Pasupat, N. Sachdeva, I. Dhillon, M. Blistein, O. Ram, D. Zhang, E. Rosen *et al.*, "Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities," *arXiv preprint arXiv:2507.06261*, 2025.
[49] S. Muralidharan, S. Turuvekere Sreenivas, R. Joshi, M. Chochowski, M. Patwary, M. Shoeybi, B. Catanzaro, J. Kautz, and P. Molchanov, "Compact language models via pruning and knowledge distillation," *Advances in Neural Information Processing Systems*, vol. 37, pp. 41 076–41 102, 2024.

APPENDIX A

System Prompt

You are a knowledgeable assistant skilled in analyzing decompiled C functions. When solving a problem, explain your reasoning step-by-step. Start with your initial approach and question its potential flaws. Please output in JSON format.

Guidance Prompt

Analyze the decompiled C function and extract some technical keywords capturing core functionality and critical attributes. Focus on these aspects:
1. Strict argument count match with standard signatures
2. Memory manipulation
3. String manipulation characteristics
…
Infer function **FUN_08025ed8** based on the above features and the context of the code. Equivalent 2-3 standard library functions and 4-5 keywords that can describe the function **FUN_08025ed8**. Do not include the callee function.

[Decompiled Function]:

```
long FUN_08025ed8(long param_1,char param_2)
{
  while( true ) {
    lVar2 = callee_1(param_1);
    if (lVar2 == 0) break;
  …
```

[Callee Function]:

```
char * callee_1(char *param_1,char param_2)
{
  while( true ) {
    if (*param_1 == '\0') {
      if (param_2 != '\0') {
  …
```

Output Structure

Respond with a LIST of keywords in JSON format. EXAMPLE OUTPUT:
{"keywords": ["string manipulation", "memory allocation", ...]}

Fig. 11: **Keyword Extraction Prompt Template.** The prompt used to extract technical keywords from decompiled code for retrieval.

System Prompt

You are a knowledgeable assistant skilled in analyzing decompiled C functions. When solving a problem, explain your reasoning step-by-step. Start with your initial approach and question its potential flaws. Please output in JSON format.

Guidance Prompt

Analyze the given decompiled function and determine which standard C library function it most likely corresponds to. Consider the following aspects:
1. Strict argument count match with standard signatures
2. Memory manipulation patterns
3. String manipulation characteristics
…
Infer function **FUN_08025ed8** based on the above features and the context of the code.

[Decompiled Function]:

```
long FUN_08025ed8(long param_1,char param_2)
{
  while( true ) {
    lVar2 = callee_1(param_1);
    if (lVar2 == 0) break;
  …
```

[Callee Function]:

```
char * callee_1(char *param_1,char param_2)
{
  while( true ) {
    if (*param_1 == '\0') {
      if (param_2 != '\0') {
  …
```

RAG Prompt

Here are some documents that may help you understand the function better, Note that the implementation may not be exactly the same as the function you are analyzing. Please refer to the documents and then answer the question:

[FILE strrchr.html]

std::strrchr

Defined in header <cstring>

```
const char* strrchr( const char* str, int ch );
      char* strrchr(       char* str, int ch );
```

Finds the last occurrence of `ch` (after conversion to `char`) in the byte string pointed to by `str`. The terminating null character is considered to be a part of the string and can be found if searching for `'\0'`.

[FILE strchr.html]

std::strchr

Defined in header <cstring>

```
const char* strchr( const char* str, int ch );
      char* strchr(       char* str, int ch );
```

Finds the first occurrence of the character `static_cast<char>(ch)` in the byte string pointed to by `str`. The terminating null character is considered to be a part of the string and can be found if searching for `'\0'`.

…

Output Structure

If the function does not correspond to any standard library function, set final_answer to "None". Respond function name of **FUN_08025ed8** in JSON format.
EXAMPLE OUTPUT:
{"thinking_content": ["Step 1: ... ", "Step 2: ..."],"final_answer": "..."}]}

Fig. 12: **Function Identification Prompt Template.** The prompt used to combine RAG context with the target code for final identification.

APPENDIX B

Here we list the functions used as positive and negative samples in our test set.

TABLE V: Functions Used as Positive Samples

memset	memcpy	memmove	memcmp	memchr
strlen	strcpy	strncpy	strcmp	strncmp
strchr	strrchr	strstr	atoi	strtoul

TABLE VI: Functions Used as Negative Samples

char2hex	net_buf_simple_add
pkt_estimate_headers_length	net_eth_is_addr_broadcast
net_buf_simple_max_len	net_buf_simple_pull
net_buf_simple_pull_mem	net_if_ipv4_maddr_join
net_pkt_remaining_data	net_if_l2
net_nbr_ref	net_pkt_get_info
net_pkt_ref	net_stats_update_tcp_seg_drop
net_tcp_recv	

APPENDIX C

To illustrate the representativeness of our test set, Table VII presents the statistical metrics for the representative standard library functions, detailing their total call frequency, cross-architecture prevalence, and frequency ratio relative to average application functions.

TABLE VII: Statistical Analysis of Selected Standard Library Functions in the Test Set

Function	Total Calls	Prevalence	Freq. Ratio*
memcpy	625	8/9 (88.9%)	16.71
memset	310	7/9 (77.8%)	9.47
strlen	267	9/9 (100%)	6.35
memcmp	217	9/9 (100%)	5.16
memmove	106	9/9 (100%)	2.52
strcmp	80	9/9 (100%)	1.90
strcpy	77	9/9 (100%)	1.83
strchr	75	9/9 (100%)	1.78
strtoul	67	9/9 (100%)	1.59
strncpy	47	9/9 (100%)	1.12
strncmp	40	9/9 (100%)	0.95
atoi	19	9/9 (100%)	0.45
memchr	16	9/9 (100%)	0.38
strstr	16	8/9 (88.9%)	0.43
strrchr	9	9/9 (100%)	0.21

* *Freq. Ratio: The ratio of the function's call frequency to the average call frequency of all functions in the binary.*