Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.067636

ARTICLE

MemHookNet: Real-Time Multi-Class Heap Anomaly Detection with Log Hooking

Siyi Wang, Yan Zhuang*, Zhizhuang Zhou, Xinhao Wang and Menglan Li

School of Cyber Science and Engineering, Zhengzhou University, Zhengzhou, 450002, China

*Corresponding Author: Yan Zhuang. Email: yan.zhuang@zzu.edu.cn

Received: 08 May 2025; Accepted: 08 July 2025; Published: 23 September 2025

ABSTRACT: Heap memory anomalies, such as Use-After-Free (UAF), Double-Free, and Memory Leaks, pose critical security threats including system crashes, data leakage, and remote exploits. Existing methods often fail to handle multiple anomaly types and meet real-time detection demands. To address these challenges, this paper proposes MemHookNet, a real-time multi-class heap anomaly detection framework that combines log hooking with deep learning. Without modifying source code, MemHookNet non-intrusively captures memory operation logs at runtime and transforms them into structured sequences encoding operation types, pointer identifiers, thread context, memory sizes, and temporal intervals. A sliding-window Long Short-Term Memory (LSTM) module efficiently filters out suspicious segments, which are then transformed into pointer access graphs for classification using a GATv2-based model. Experimental results demonstrate that MemHookNet achieves 82.2% accuracy and 81.5% recall with an average inference time of 15 ms, outperforming DeepLog and GLAD-PAW by 11.7% in accuracy and reducing latency by over 80%.

KEYWORDS: Use-after-free detection; heap memory vulnerabilities; log analysis; memory leak detection; graph neural network

1 Introduction

Dynamic heap memory allocation plays a critical role in enhancing the flexibility and resource efficiency of modern software systems. However, this mechanism also introduces severe security vulnerabilities, such as Use-After-Free (UAF) [1,2], Double-Free [3], and Memory Leaks [4]. These anomalies are commonly exploited by attackers to trigger system crashes, data leakage, and even remote code execution. Among these, UAF vulnerabilities are particularly challenging to detect due to their temporal dependency and cross-thread behavior, which can easily bypass traditional static or runtime defenses [5].

To mitigate such threats, detection techniques are classified into static analysis, dynamic monitoring, and hybrid approaches. Static analysis tools [6] like Cppcheck and Flawfinder detect memory issues at the code level but often generate false positives due to a lack of runtime context. Dynamic tools [7], such as Valgrind and AddressSanitizer, capture heap behaviors at runtime, offering better accuracy but at the cost of substantial performance overhead. Hybrid approaches [8] combine these techniques but often complicate deployment.

Given these limitations, log-based anomaly detection has recently emerged as a promising alternative. By capturing runtime memory operation logs, such approaches offer a lightweight and scalable means to

Copyright © 2025 The Authors. Published by Tech Science Press.
This work is licensed under a Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

model program behavior. These logs naturally exhibit sequential and semantic patterns, making them well-suited for learning-based anomaly detection. Deep learning models such as Long Short-Term Memory (LSTM) networks and Graph Neural Networks (GNNs) have shown strong capabilities in modeling temporal and structural dependencies, respectively. However, most existing methods [9–11] are constrained to binary classification tasks [12], rely on offline analysis [8], or adopt single-model inference strategies [13,14], thereby failing to meet the demands of real-time, multi-class detection in real-world environments. Additionally, while recent graph-based models have demonstrated success in multi-class detection for image and spatial data [15,16], they often depend on high-level feature reconstruction and are ill-suited for analyzing runtime memory logs due to modality mismatches and latency constraints.

Log-based anomaly detection has emerged as a prominent technique in recent years, leveraging runtime memory operation logs for behavior modeling [17]. These logs inherently contain sequential and semantic information, making them suitable for learning-based detection. Deep learning models, such as Long Short-Term Memory (LSTM) networks and Graph Neural Networks (GNNs), have demonstrated strong potential in capturing temporal and structural dependencies. However, existing approaches are mostly limited to binary classification, offline analysis, and single-model inference, failing to meet the practical demands for real-time, multi-class detection in real-world systems.

To address these limitations, we propose MemHookNet (Memory Hook-guided Network), a novel end-to-end framework for real-time multi-class heap memory anomaly detection. MemHookNet integrates temporal and structural modeling through a dual-path architecture. At runtime, memory operations such as malloc, free, and memcpy are intercepted using a lightweight LD_PRELOAD hook mechanism. A sliding-window LSTM module first filters out benign sequences. Suspicious segments are then converted into structured pointer access graphs by a semantic-enhanced graph builder. These graphs are finally classified using a graph attention network (MemGATClassifier) for fine-grained anomaly recognition across UAF, Double-Free, Memory Leak, and normal behaviors.

The main contributions of this paper are summarized as follows:

- We propose MemHookNet, a unified temporal-structural framework for multi-class heap memory anomaly detection. It addresses the limitations of previous methods by achieving high accuracy, strong scalability, and low runtime overhead in real-world environments. Experimental results show that MemHookNet attains an accuracy of 82.2%, outperforming state-of-the-art baselines such as DeepLog (70.5%) and GLAD-PAW (81.2%).
- We design a non-intrusive runtime memory logging mechanism based on LD_PRELOAD, capable of capturing multi-threaded heap behaviors in real time. This design minimizes performance impact during execution. As reported in Section 5, the system achieves an average inference time of 15 ms, representing a substantial improvement over traditional analysis tools like Valgrind.
- We introduce a lightweight LSTM-based prefiltering module for log sequence screening, which significantly improves processing efficiency while maintaining high recall performance.
- We further construct semantic pointer access graphs using MemLogGraphBuilder, and perform graph-level classification through MemGATClassifier, an attention-based graph neural network. This graph-based representation enables the model to effectively capture both temporal and structural dependencies of heap memory anomalies. The proposed classifier achieves an AUC-ROC of 94.7%, considerably outperforming other graph-based methods such as Logs2Graphs (70.2%) and conventional GCN-based models.

In summary, MemHookNet transforms the traditional "offline + binary" detection paradigm into a "real-time + multi-class + deployable" solution. With high precision, scalability, and engineering applicability, our framework offers a practical approach to securing dynamic memory usage in modern software systems.

2 Related Work

In recent years, significant progress has been made in the field of heap memory anomaly detection, particularly in detecting UAF vulnerabilities. Existing methods for detecting heap memory vulnerabilities can be broadly categorized into four types: static analysis, dynamic detection with specialized mechanisms, hybrid analysis systems, and log-driven deep learning approaches. This section systematically reviews each of these categories, emphasizing their respective strengths and limitations.

2.1 Traditional Detection Techniques for Heap Vulnerabilities

Static analysis tools detect memory issues by modeling source code or intermediate representations before execution. While they scale well, they lack runtime awareness. For example, Zhang et al. [18] proposed UAFX, which uses alias analysis and partial-order constraints to detect complex UAF vulnerabilities in the Linux kernel, addressing some limitations of traditional tools. However, UAFX heavily depends on path modeling and alias resolution, which face scalability issues like path explosion. Common tools like Cppcheck [19] and Flawfinder [20] perform efficient rule-based scans but cannot capture deep semantic relationships, particularly in multi-threaded or non-deterministic paths. In summary, while static analysis is useful for early-stage scanning, it lacks the precision and granularity needed for real-time detection, especially for vulnerabilities dependent on timing or thread interleaving.

Dynamic detection monitors actual memory behavior during execution using tools like Valgrind [21] and AddressSanitizer (ASan) [22], which detect UAFs and buffer overflows with red-zone techniques, though at a high performance cost (2-5x runtime overhead). Several specialized mechanisms have been proposed to improve efficiency. For example, Lee et al. [13] introduced Dangling Pointer Nullification (DPN), which nullifies references during object deallocation to prevent UAF exploits, but it is ineffective against pointer aliasing. Van Der Kouwe et al. [23] proposed DangSan, which reclaims dangling pointers using background threads, though it suffers from delayed detection. Gorter et al. [24] presented DangZero, employing a One-Time Allocation (OTA) strategy for high-precision detection by directly accessing user-space page tables, but it requires significant deployment support. Ahn et al. [25] introduced BUDAlloc, which combines user-space and kernel-level cooperation to reduce overhead, improving adaptability. UAF-GUARD [14] uses fine-grained memory permissions to block illegal memory accesses with high precision but introduces additional runtime overhead, making it less suitable for lightweight or performance-sensitive applications. Despite their high accuracy, most dynamic detection methods rely on privileged instructions, kernel modifications, or custom allocators, complicating deployment and limiting generalizability across software ecosystems.

In conclusion, both static and dynamic approaches face inherent trade-offs: Static analysis offers high efficiency but lacks temporal and concurrency awareness; Dynamic detection provides better precision but suffers from significant performance costs and deployment complexity. This motivates the need for a lightweight, real-time, and semantically expressive alternative, which we explore in subsequent sections.

2.2 Hybrid Analysis Systems

Hybrid analysis aims to balance detection coverage and behavioral precision by combining static and dynamic techniques. Although platforms like Angr [26] and Driller [27] demonstrate strong capability

through symbolic execution and path exploration, their reliance on SMT solvers and complex analysis pipelines significantly limits real-time deployment and scalability in high-frequency online systems.

2.3 Log-Driven Deep Learning-Based Methods

Recent advances in deep learning [28] have fueled interest in anomaly detection based on runtime logs. LSTM-based models, such as DeepLog [9], effectively learn normal log sequences and identify deviations. However, these models are limited by their inability to capture structural relationships between events. To enhance contextual awareness, LogAnomaly [10] introduces event embeddings and log templates, yet it still relies on binary classification and lacks multi-class granularity.

To address these limitations, recent studies have integrated graph neural networks (GNNs) into log-based frameworks. Logs2Graphs [11] transforms log windows into weighted graphs for GCN-based detection, but suffers from limited classification accuracy. GLAD-PAW [29] and DeepIraLog [30] enhance structural analysis with attention mechanisms and heterogeneous graph modeling, improving interpretability but introducing significant pipeline complexity, hindering real-time use.

Despite growing interest in structure-aware models, most log-driven methods remain constrained to offline settings, binary classification, and single-stage inference, falling short of the demands for real-time, multi-class heap anomaly detection.

2.4 Summary and Insights

In summary, the current methods for detecting heap vulnerabilities face several common limitations:

- **Offline Operation:** Many existing systems operate offline, failing to meet the real-time response requirements that are critical for modern memory anomaly detection.
- **Limited Feature Modeling:** Traditional models often rely on single-dimensional features, focusing either on temporal or structural characteristics, while ignoring the rich and multi-dimensional behavioral semantics required for detecting complex anomalies like UAF, Double-Free, and Memory Leak.
- **Invasive Data Collection:** A significant number of existing mechanisms for data collection are invasive or resource-heavy, limiting their deployment in production environments where low overhead is crucial.
- **Binary Classification:** Most of these approaches are designed for binary classification tasks, making them inadequate for real-world scenarios where multiple types of memory anomalies (such as UAF, Double-Free, and Memory Leak) must be detected simultaneously.

These limitations highlight a clear gap for a deployable, real-time, multi-class detection framework. The next section introduces MemHookNet, our proposed approach that addresses these challenges via hybrid temporal-structural deep modeling.

3 Design of MemHookNet for Memory Anomaly Detection

To address the limitations discussed in Section 2.4, We propose MemHookNet, a lightweight and deployable framework that integrates sequence modeling and graph reasoning in a dual-path architecture for real-time, multi-class heap anomaly detection. It achieves high accuracy with minimal overhead.

3.1 System Architecture

MemHookNet adopts a four-stage pipeline: log collection, temporal filtering, graph construction, and multi-class classification (Fig. 1). At runtime, a HOOK module intercepts memory operations (e.g., malloc, free) and generates structured logs. A sliding-window LSTM model pre-filters suspicious sequences. Those

exceeding a threshold are transformed into memory access graphs and classified by a GAT-based network across four categories: Normal, UAF, Double-Free, and Memory Leak.

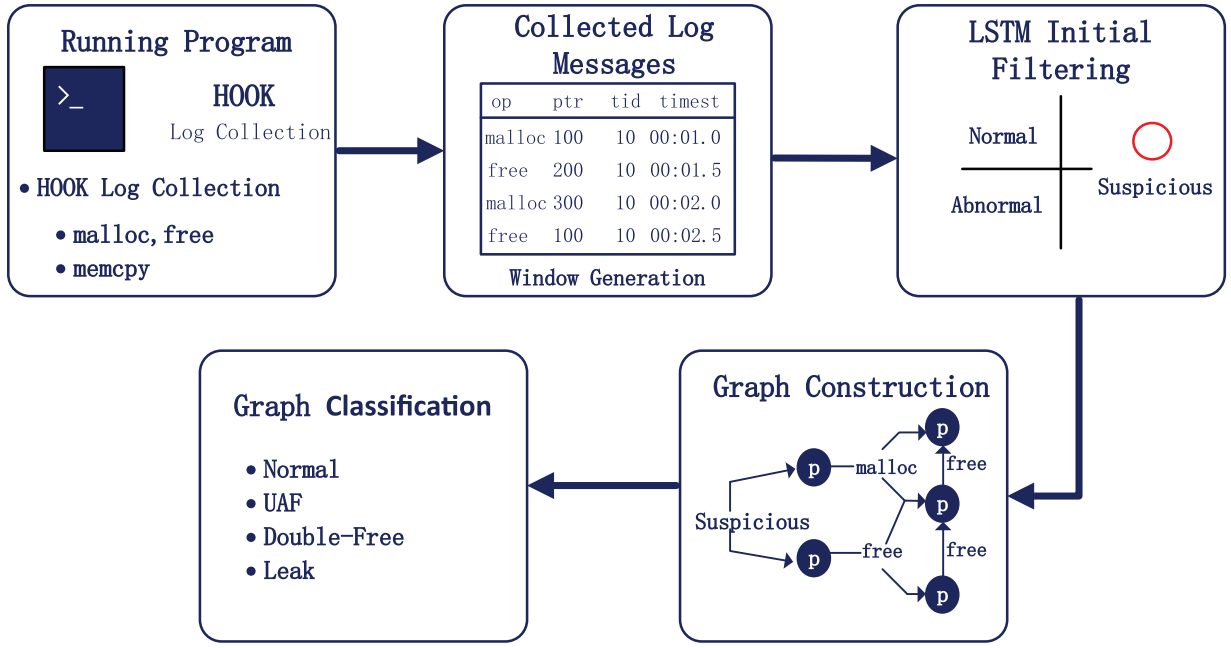


Figure 1: System architecture of MemHookNet for heap memory anomaly detection

Algorithm 1 illustrates the real-time workflow of log collection, filtering, and classification.

To provide a clear execution overview, Algorithm 1 illustrates the system's runtime flow, detailing how memory logs are collected, filtered, transformed, and ultimately classified in a real-time processing loop.

- **Steps 1–2:** The system is initialized by loading the HOOK module, which intercepts memory operations and logs them in a structured format.
- **Steps 3–5:** Sliding windows are continuously generated from the incoming logs and passed through an LSTM-based filter to predict anomaly probabilities.
- **Step 6:** If the LSTM output exceeds a predefined threshold, the log window is considered suspicious and forwarded to the graph construction module.
- **Steps 7–10:** The suspicious log is transformed into a pointer-based graph and classified by a GNN. If the result indicates a high-risk anomaly, a predefined response is triggered (e.g., logging, alerting, or blocking).

Algorithm 1: MemHookNet inference process pseudocode

Input: Target program binary during execution
Output: Predicted label for each log fragment $\in$ {Normal, UAF, Double-Free, Leak}
1 Load the HOOK module `malloc_hook.so` to intercept memory operation functions
2 Continuously record structured logs `log` $\leftarrow$ {op_type, ptr_id, tid, size, timestamp}
3 **while** *the log stream is not finished* **do**
4 Construct the sliding window `slide_window` from the most recent N log entries
5 Invoke the LSTM model for prediction $y\_pred \leftarrow$ LSTM(`slide_window`)

(Continued)

Algorithm 1 (continued)

```
6       if max(y_pred) > threshold then
7               Construct the access graph structure graph ←
                  MemLogGraphBuilder(slide_window)
8               Obtain classification result y_graph ← MemGATClassifier(graph)
9               if y_graph belongs to a high-risk category then
10                      Trigger response action trigger_action(y_graph)
                                // actions include logging, alerts, or termination
11              end
12      end
13 end
```

3.2 HOOK Log Collection Module

The HOOK module non-intrusively intercepts and logs heap memory operations (e.g., malloc, free) in real time, providing high-quality temporal input for downstream analysis.

To achieve transparent interception, the module leverages the `LD_PRELOAD` [31] mechanism to inject logging hooks at the user level. It uses thread-safe logging strategies to preserve ordering, and includes lightweight pointer lifecycle tracking for anomalies such as UAF and Double-Free.

Each operation is recorded in structured fields (e.g., operation type, pointer ID, thread ID, memory size, timestamp), enabling accurate sequence modeling and graph construction.

Overall, the HOOK module is lightweight, stable, and incurs negligible runtime overhead, making it suitable for high-frequency heap monitoring in practical systems.

3.3 Log Window and LSTM Pre-Filtering Module

To enhance detection recall while minimizing computational overhead, MemHookNet integrates a lightweight temporal pre-filtering mechanism within the log stream processing pipeline. This module utilizes state-preserving LSTM [32,33] networks to efficiently process sliding windows of log data, enabling rapid filtering of potential anomalies from continuous memory operation sequences. The LSTM model has been enhanced to ensure memory retention across different windows, allowing for better temporal context and dependency modeling.

In addition, the model employs a multi-scale sliding window mechanism, simultaneously training models with window sizes of 10, 20, and 30. This multi-scale approach helps capture different patterns of anomalies that may span across different window lengths, improving the robustness of the detection process. The log sequences are segmented into fixed-length operation fragments, with each fragment containing five key fields: operation type (op_type), pointer identifier (ptr_id), thread identifier (tid_id), allocated memory size (size), and the time interval between adjacent operations (time_delta). These fields are discretized and normalized before being input into a multi-class LSTM network.

The LSTM architecture, MulticlassLSTM, includes three embedding layers [34], each encoding operation type, pointer ID, and thread ID, with an embedding dimension of 16. The core of the model consists of a two-layer bidirectional LSTM with 64 units per layer and Dropout [35] (set to 0.3) to prevent overfitting. This design allows the model to capture long-term dependencies and generalize across the various temporal patterns in memory access logs. The output is passed through a fully connected layer and processed by a

Softmax function to generate predictions for identifying normal and suspicious behaviors (e.g., UAF, Double-Free, Memory Leak [36]). The training uses a Label Smoothing Cross-Entropy loss function with Early Stopping, evaluated using macro F1-score and accuracy [37] to ensure robustness and generalization.

During inference, the system computes the probability distribution for each log window across the four behavior categories and applies an adjustable confidence threshold (default = 0.6). If the prediction probability for any anomaly category in a given window exceeds this threshold, the segment is flagged as suspicious behavior and proceeds to the graph modeling stage. If the threshold is not met, the segment is discarded, conserving computational resources for subsequent processing stages. This design reduces GNN invocation frequency while preserving high recall, offering an efficient front-end filter.

Fig. 2 illustrates the architecture of the Log Window and LSTM Pre-filtering Module, showing how log windows are processed by the LSTM model, followed by threshold-based classification to identify suspicious behavior.

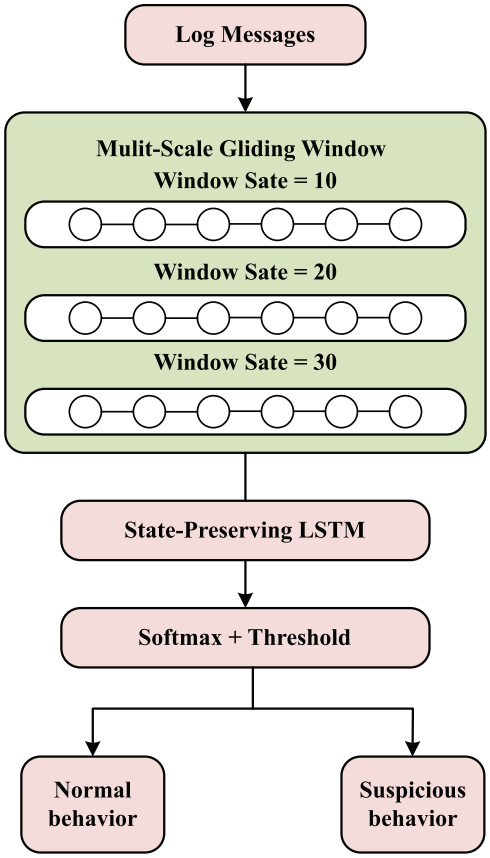


Figure 2: Log window and LSTM pre-filtering module architecture

3.4 MemLogGraphBuilder: Graph Construction Module

MemHookNet captures structural dependencies by transforming suspicious logs into memory behavior graphs, built on LSTM-filtered segments. These graphs represent pointer access paths, thread collaboration, and temporal logic, providing structured input for the downstream graph neural network. Each unique pointer identifier (ptr_id) within a sliding window is treated as a node in the graph. Each node is assigned a 14-dimensional feature vector, including: a 6-dimensional one-hot encoding of the operation type,

normalized allocation size, thread ID, operation frequency, relative position to the most recent malloc, and time intervals between operations. These features characterize the behavior and context of the pointer object within the current log segment.

For edge construction, the system defines three types of semantically meaningful relationships:

- **Access edges:** Represent multiple accesses to the same pointer, reflecting the usage trajectory during its lifecycle.
- **Same-thread edges (same_tid):** Connect consecutive operations within the same thread to preserve intra-thread temporal ordering.
- **Temporal sequence edges (time_seq):** Maintain the global chronological order of log entries

To enhance the graph's learnability, each edge is also annotated with a 6-dimensional edge feature vector, which includes: one-hot encoded edge type, pointer consistency, thread consistency, and normalized time differences. These edge attributes improve the model's ability to perceive operational semantics and relationships.

Finally, the graph samples are exported in a standard JSON triplet format, containing node features (nodes.json), edge relationships (edges.json), and corresponding labels (label.txt). These graph objects can be directly utilized by the graph neural network module for training and inference. Through this module, linear log sequences are effectively transformed into structurally expressive graph representations, laying the foundation for fine-grained classification of complex heap memory behaviors.

3.5 MemGATClassifier: Graph Neural Network Module

The MemGATClassifier, as the core discriminative module of the MemHookNet framework, is responsible for performing multi-class classification of memory access behaviors that are represented in graph structures. This module takes advantage of the structural modeling capabilities of GNNs and integrates an attention mechanism to capture the operational semantics and thread dependencies between nodes in the graph. This enables accurate classification of four behavior categories: Normal, UAF, Double-Free, and Memory Leak.

To enhance the model's ability to capture structural semantics, MemHookNet adopts the GATv2 [38–40] as its backbone architecture. GATv2 differs from traditional GCNs by introducing learnable attention mechanisms during the feature aggregation phase. This allows the model to adaptively identify the most critical adjacent nodes for classification, enhancing its capacity to model complex dependency patterns in pointer behavior graphs. The architecture of the MemGATClassifier is detailed in Table 1, encompassing input feature encoding, graph convolution layers, graph-level pooling, fully connected classifiers, and the design of the loss function. The overall structural workflow is illustrated in Fig. 3.

Table 1: MemGATClassifier detection model structure

Module	Configuration description
Input	Node Features: 14 dimensions (operation + statistical context), Edge Features: 6 dimensions (one-hot + semantic encoding)
Network structure	Two-layer GATv2Conv, heads = 8, hidden = 64, dropout = 0.3
Pooling method	Global Mean Pooling (Graph-level aggregation)
Regularization	LayerNorm + ReLU

(Continued)

Table 1 (continued)

Module	Configuration description
Output layer	Fully connected layer + Softmax, outputs probabilities for four behavior categories
Loss function	FocalLoss (γ = 1.5), supports class imbalance sample weighting
Optimizer	Adam, learning rate = 0.001
Framework	Based on PyTorch Geometric, uses `convert_to_data_object` to construct PyG graph objects
Training strategy	Supports Early Stopping, evaluation metrics: Macro-F1, Recall, AUC

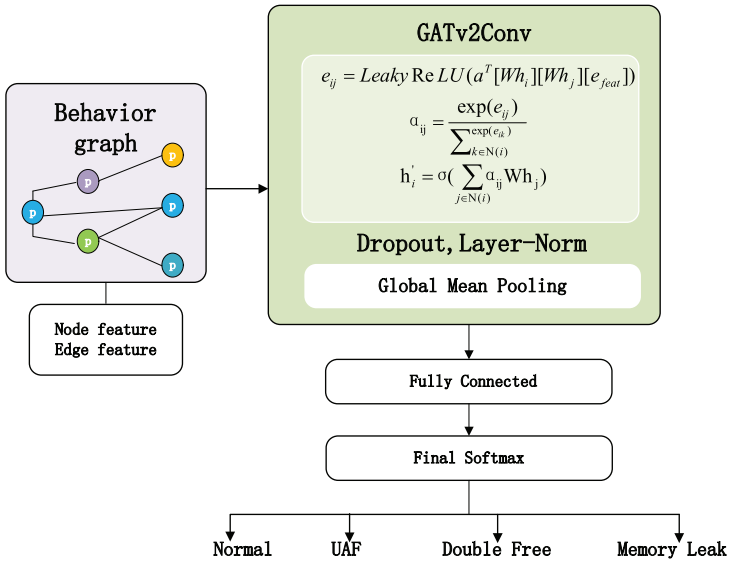


Figure 3: MemGATClassifier detects the model structure

During the inference process, the model aggregates neighboring features for each node based on attention weights and generates a unified semantic representation through graph-level pooling. It then outputs the probability distribution for each graph sample across the four behavior categories. If the prediction probability for any behavior category exceeds a specified threshold (e.g., UAF category > 0.7), the system triggers a series of security response mechanisms, including:

- **Anomaly Log Archiving:** High-risk graph samples and their classification results are recorded locally for subsequent auditing and analysis.
- **Program Behavior Blocking:** When protection mode is enabled, the system can automatically interrupt related processes to prevent abnormal heap operations from evolving into actual attacks.
- **Security Event Reporting:** Detection results can be integrated into an operations platform, supporting real-time alerts, graphical displays, and behavioral trend analysis.

By integrating graph structure discrimination with risk response mechanisms, MemGATClassifier not only performs multi-class classification of heap memory behaviors but also supports immediate disposal strategies based on model outputs. This creates a feedback loop from detection to defense, providing solid potential for practical deployment.

To evaluate the adaptability of various GNNs for memory vulnerability detection, we compared mainstream models (see Table 2). Given the sparse nature of log graphs, asymmetric access dependencies, and varying importance of critical nodes, we chose the Graph Attention Network (GAT) for its ability to automatically identify key access nodes, distinguish pointer/thread IDs in different access paths (e.g., malloc vs. free vs. access), and provide a simple structure that is easy to train and deploy. Additionally, GAT's decoupling from the LSTM module offers flexibility in model combinations, enhancing its suitability for this task.

Table 2: Comparison of common GNN models for heap vulnerability detection

GNN type	Suitability for heap vulnerability detection	Reason
GCN	× Moderate	All neighbors are treated equally, unable to capture "key access nodes"
GraphSAGE	Can be used	Supports large graphs but treats neighbors equally
GGNN	× Too complex	Involves GRU-based control, difficult to deploy for inference
GAE/VGAE	× Not suitable	Unsupervised learning, not suitable for tasks requiring labeled data
ST-GNN/DynamicGNN	× Overly complex	Requires sequential data, doesn't support dynamic graph construction
GAT (Selected)	Extremely suitable	Automatically learns key access nodes, simple structure, and strong expressive power

In contrast, traditional GCN treats neighbors equally, while GraphSAGE lacks sufficient expressive power. Gated Graph Neural Network(GGNN) and Spatio-Temporal Graph Neural Network(ST-GNN) have excessively high construction and training costs, and GAE/VGAE are not suitable for supervised classification tasks. Therefore, GAT is the optimal choice, balancing accuracy and engineering feasibility.

To summarize, the MemGATClassifier not only achieves high-accuracy classification of heap memory anomalies but also enables system-level behavior auditing and defense linkage through its highly interpretable attention mechanism. This, when combined with the LSTM module that captures temporal features, forms a robust "time-structure" dual modeling capability for memory anomaly detection, providing a solid foundation for system deployment and model generalization.

4 Experimental Design and Implementation

In order to comprehensively evaluate the performance of the proposed MemHookNet framework in multi-class heap memory anomaly detection, this chapter presents a systematic experimental design covering multiple dimensions, including experimental procedures, data construction, model training, comparison schemes, and ablation analysis. The entire experimental process strictly follows the system architecture described in Section 3 (see Fig. 1), covering four key stages: log collection, sliding window filtering, graph

structure construction, and multi-class behavior classification. Each stage is designed to ensure high reproducibility while also considering the feasibility and deployment-friendly aspects of the engineering implementation. This provides a solid foundation for the application of the model in real-world system environments. In this chapter, we will detail the experimental environment configuration, data processing logic, training strategy setup, and the specific implementation details of the comparison experiments.

4.1 Experimental Environment Setup

All experiments were conducted in a unified local hardware and software platform environment, with specific configurations shown in Table 3. The system model implementation and training were carried out using the PyTorch and PyTorch Geometric frameworks, and the experiments were deployed on a workstation environment with Graphics Processing Unit (GPU) acceleration to ensure efficient training and inference performance.

Table 3: Experimental platform configuration

Item	Configuration description
Operating system	Ubuntu 22.04 LTS
Processor	Intel Core i7-12700H × 16 threads
GPU	NVIDIA RTX 3060 Laptop GPU (6 GB VRAM)
Memory	32 GB DDR4
Python version	Python 3.9 (Anaconda environment)
Key Dependencies	PyTorch 2.1, PyTorch Geometric 2.3, scikit-learn, etc.

4.2 Experimental Environment Setup

To facilitate reproducible experimentation and ensure consistency across different processing stages, we designed a standardized logging procedure to structure intercepted memory operations into training-ready sequences. In our experiments, we extracted memory operation logs by executing vulnerable programs from the SARD dataset, using the LD_PRELOAD technique to intercept heap memory operations such as malloc, free, and memcpy during runtime. The resulting structured logs were used as the primary input for training and evaluating the MemHookNet framework.

This interception process is implemented via the LD_PRELOAD + dlsym mechanism, which transparently hijacks key memory management functions and monitors heap activity without modifying the original source code. The pseudocode for the memory log collection and structuring process is presented in Algorithm 2, which outlines how each intercepted memory operation is formatted into a consistent structure. Specifically, the system dynamically resolves the address of each target function, extracts the operation type, pointer address, thread ID, allocation size, and timestamp, and encodes this information into a unified log entry. These structured logs serve as the foundational input for the downstream sequence modeling and graph construction modules.

Algorithm 2: Memory operation log collection and structuring pseudocode

Input: Set of runtime hijacked functions $F = \{$`malloc`, `free`, `memcpy`, `strcpy`$\}$
Output: Structured log file `malloc_log.txt`

(Continued)

Algorithm 2 (continued)

1 Initialize the log output file `log_fp`
2 **foreach** *function call* $f \in F$ **do**
3 Use `dlsym(RTLD_NEXT, f)` to get the address of the original function
4 Record the current operation type `op_type`
5 Extract pointer `ptr`, thread ID `tid`, memory size `size`,timestamp `ts`
6 Write the structured entry `<op_type, ptr_id, tid, size, timestamp>`
7 **end**
8 End for

To ensure consistency in input formatting across both sequence modeling and graph modeling, each memory operation log entry is formalized as the following quintuple: To ensure consistency in input formatting across both sequence modeling and graph modeling, each memory operation log entry is formalized as the following quintuple. See Eq. (1):

$$L_t = \langle p\_t, ptr\_t, tid\_t, size\_t, \Delta t \rangle \tag{1}$$

where:

- op_type: Operation type;
- ptr_id: Memory pointer ID (uniquely identified after hashing);
- tid_id: Thread ID;
- size: Memory size of the operation (in bytes);
- time_delta: Time interval from the previous operation.

The final complete log stream is represented as $L = \{L_1, L_2, \ldots, L_T\}$.

This structured definition provides mathematical consistency to support sliding window segmentation, sequence encoding, and graph modeling.

4.2.1 Log Collection and Operation

The raw data used in this system is collected by the hook interception module, which performs non-intrusive hijacking of memory management functions via the LD_PRELOAD mechanism (as detailed in Section 3.2). To enhance the reproducibility and transparency of the experiments, we publicly provide the complete collection framework, runtime scripts, and test examples. The core directory structure is illustrated in Fig. 4.

During the experiment, test cases are executed by launching the target program with the command: `LD_PRELOAD=./malloc_hook.so./target_program`

which enables automatic recording of runtime memory operations. This process generates a raw log file named `malloc_log_raw.txt`. The collected logs cover typical memory usage scenarios, including valid malloc/free operations, as well as abnormal cases such as reuse, UAF, Double-Free, and memory leaks. The `process_logs.sh` script is used to parse the raw logs into structured CSV or JSON files, which serve as input for the subsequent sequence modeling and graph construction modules.

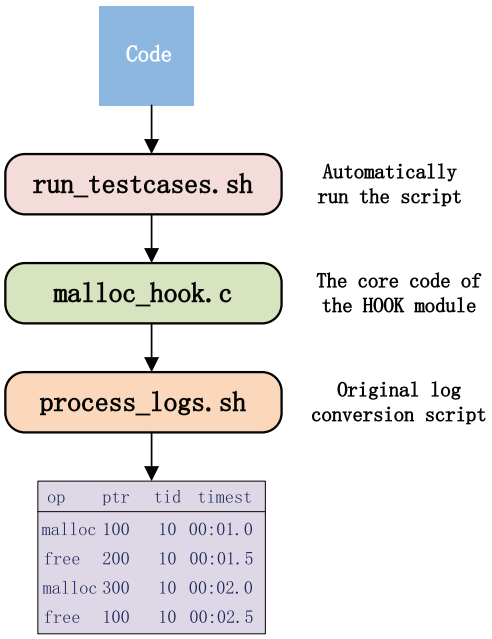


Figure 4: File composition of the HOOK-based logging module

4.2.2 Sliding Window Sample Generation

The system uses a sliding window mechanism (window size N = 10) to segment the time-ordered logs, generating standardized LSTM input samples. Each sequence sample includes the following five fields: operation type (op_type), pointer ID (ptr_id), thread ID (tid_id), memory size (size), and time difference between adjacent operations (time_delta). After discretization and normalization, these fields are input into the sequence model. The sample labels are generated through rule matching and manual verification, covering four categories of heap memory behaviors: Normal, UAF, Double-Free, and Memory Leak. This process ensures consistency and accuracy in labeling the anomalous samples in the training dataset.

For windows identified as suspicious by the LSTM model, the system uses the custom-developed MemLogGraphBuilder to construct corresponding graph structure samples, capturing the semantic relationships between potential pointer operations. Each graph sample contains the following feature information:

- Node Features: 14 dimensions, including one-hot encoding of operation type, operation frequency, thread normalization value, memory size statistics, and pointer usage count.
- Edge Features: 6 dimensions, encoding structural semantics such as access order, thread consistency, pointer consistency, and time intervals.

The final graph samples are output in standard JSON format, consisting of three files: nodes.json, edges.json, and label.txt, which can be directly used as input for the MemGATClassifier.

4.3 Model Training Configuration

This section provides a detailed explanation of the two core models in the MemHookNet system: the LSTM sequence classifier and the GNN graph classifier. It covers their network architectures, training strategies, and loss functions, aiming to demonstrate the design rationale and implementation details of each module in addressing the multi-class heap memory behavior detection task.

4.3.1 Time-Series Model (LSTM)

The sequence model is used for quick screening and preliminary classification of memory operation sequences within the sliding window. The input consists of a sequence of five fields from each log fragment, where op_type, ptr_id, and tid are mapped to 16-dimensional dense vectors through embedding layers. These embedding vectors are then passed into a two-layer bidirectional LSTM network (hidden size = 64), with Dropout (p = 0.3) applied to reduce the risk of overfitting. The network ultimately outputs the probability distribution of four types of heap memory behaviors via a fully connected layer + Softmax.

The model's loss function uses Label Smoothing CrossEntropy, which includes a smoothing term to increase the model's tolerance for samples near the class boundaries. During training, we use the Early Stopping strategy (patience = 5) to prevent overfitting, with a batch size of 32.

To address the issue of class imbalance, where anomalous categories (e.g., UAF and Leak) are much less frequent than normal behaviors, we introduce a class-weighting mechanism during training. Specifically, we set the class weighting factor in the loss function according to the frequency distribution of each class in the training set, enabling the model to focus more on the predictions for minority classes. Additionally, we experimented with a sliding window-based fragment resampling strategy, oversampling UAF and Leak fragments to increase their visibility during training and further enhance the model's ability to recognize low-frequency anomalies.

Eqs. (2) and (3) is the forward propagation formula.

$$h_t = \text{LSTM}(x_t, h_{t-1}) \tag{2}$$

$$\hat{y} = \text{Softmax}(Wh_T + b) \tag{3}$$

4.3.2 Graph Neural Network Model (MemGATClassifier)

The graph classification module is responsible for performing fine-grained behavior recognition on the constructed log graphs. To enhance the model's ability to capture complex operational dependencies, we adopt an improved Graph Attention Network, specifically GATv2, and build a two-layer GATv2Conv-based classifier.

The input to this module includes node features (14 dimensions) and edge features (6 dimensions). The edge features are embedded using an edge_encoder, which is then integrated into the attention mechanism. This results in an edge-aware attention propagation strategy that enhances relational modeling during message passing. Each graph convolutional layer is implemented using the GATv2Conv module from the PyTorch Geometric framework. The forward propagation process is as follows: x = self.conv1(x, edge_index, edge_emb);x = self.conv2(x, edge_index, edge_emb).

The features of the final node are aggregated using global_mean_pool, and the graph-level features are passed into the final layer to predict four behavior categories: Normal, UAF, Double-Free, and Leak:

$$\hat{y} = \text{Softmax}(W_{fc} \cdot H_{\text{graph}} + b) \tag{4}$$

To address class imbalance issues (e.g., UAF/Leak samples), Focal Loss is employed as the loss function, which is formulated as:

$$L_{\text{focal}} = -\alpha(1 - p_t)^{\gamma} \log(p_t), \quad \gamma = 1.5 \tag{5}$$

The model is trained using the Adam optimizer (learning rate α), and the evaluation metrics include Accuracy, MacroF1, Recall, and AUC. To further analyze the model's component, we designed a shuffling

experiment in Section 5 to conduct systematic evaluation and measure the model's overall performance in real-world systems.

4.4 Comparison Scheme Design

To objectively evaluate the performance of MemHookNet for multi-class heap memory anomaly detection, we selected four representative baselines covering time-series, structural, and hybrid modeling paradigms: DeepLog, Logs2Graphs, LogAnomaly, and GLAD-PAW, as illustrated in Fig. 5. These models were re-implemented using official code and trained under identical conditions (dataset, hardware, and hyperparameter settings) to ensure fair comparison.

- DeepLog: A classic log sequence prediction model based on LSTM, widely used in general anomaly detection scenarios;
- Logs2Graphs: A method that represents log dependencies via graph construction, using GCN for graph-level classification;
- LogAnomaly: Introduces semantic enhancement and custom template matching strategies, integrating LSTM and graph learning structures;
- GLAD-PAW: Uses GCN to capture local dependency structures, combined with Transformer for long-range context modeling.

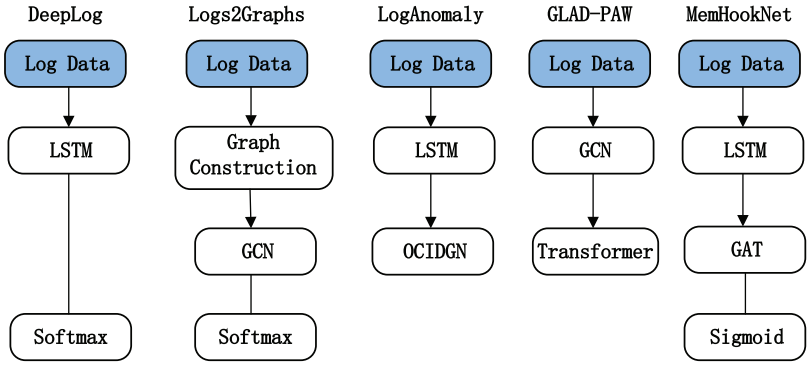


Figure 5: Comparison of processing flows of different model architectures

Fig. 5 compares their input processing pipelines, model architectures, and output strategies. We observe that MemHookNet uniquely combines temporal filtering and structural modeling, enabling superior detection of complex heap anomalies.

This comparative analysis highlights MemHookNet's advantages in multi-class recognition, real-time response, and adaptability. Its ability to effectively detect Use-After-Free, Double-Free, and Memory Leak anomalies with minimal computational overhead makes it a competitive solution for practical deployment in security-critical environments.

4.5 Ablation Study Design

To further examine the individual contributions of each core component in the MemHookNet framework, we conducted a series of ablation experiments. These experiments aim to isolate the impact of three key modules—LSTM-based sequence filtering, graph construction with attention, and HOOK-based log interception—on the overall anomaly detection performance. The study includes four experimental variants and the full model configuration, as detailed in Table 4.

Table 4: Ablation experiment settings for different model variants

Model ID	Name	Description and characteristics
B1	LSTM-only	Uses only the LSTM model to perform multi-class classification on log sequences
B2	GCN-Graph	Graph classification model based on traditional GCN, without the attention mechanism
B3	GATv2-noHook	Builds graph samples using the model structure but without enabling the HOOK interception
Ours	MemHookNet	Enables HOOK + LSTM + Graph Module using GATv2; complete end-to-end detection framework

Experimental results demonstrate that each module—HOOK, LSTM, and GATv2—plays a critical role in achieving high detection accuracy and robustness. The exclusion of any single component leads to significant performance degradation. In contrast, the full MemHookNet framework consistently yields the highest precision and recall, confirming the complementary effect of temporal and structural modeling.

In summary, this section evaluates all key modules within a unified experimental platform. By covering the entire pipeline—from log collection and normalization to sequence filtering and graph classification—the study ensures both reproducibility and engineering feasibility. The combination of multiple baselines and carefully designed variants establishes a strong foundation for the comparative performance evaluation presented in Section 5.

4.6 Threats to Validity

While MemHookNet demonstrates promising performance in detecting multi-class heap memory anomalies, several threats to validity should be acknowledged:

- **Generalization across Platforms:** Our implementation was tested on Linux-based environments using LD_PRELOAD for function interception. Portability to other operating systems (e.g., Windows, macOS) or to kernel-level anomalies may require architectural changes.
- **Log Fidelity Assumption:** We assume that memory logs are correctly and completely captured during execution. However, in real-world deployment, logging interruptions, thread concurrency, or low-level obfuscation may affect log integrity and downstream accuracy.
- **Model Dependence on Thresholds:** The overall detection performance partially depends on empirically set thresholds for both the LSTM filtering stage and GNN classification. Improper tuning in unseen environments may affect precision or recall.
- **Real-Time Constraints:** Although MemHookNet is designed for real-time deployment, extremely high-frequency applications or embedded systems with strict latency requirements may require further optimization of the logging and inference pipeline.

Future work will address these threats by validating the framework on more diverse and large-scale real-world applications, and by exploring adaptive threshold tuning and cross-platform generalization.

4.7 Algorithmic Complexity Analysis

To evaluate the computational efficiency of MemHookNet, we analyze the time complexity of its key modules.

- **LSTM Pre-Filtering:** Given a sliding window of size N and embedding dimension d, the two-layer bidirectional LSTM has a time complexity of $O\left(N \cdot d^2\right)$. Since only recent logs are filtered in real time, this stage is lightweight.
- **Graph Construction:** Let V be the number of pointer nodes and E be the number of edges. Graph construction computes per-node and per-edge features, with worst-case complexity of $O\left(V^2\right)$ due to potential dense relationships.
- **GNN Classification:** The two-layer GATv2 network performs attention-based aggregation with complexity $O\left(E \cdot d + V \cdot d^2\right)$. Due to early-stage filtering, only a small fraction of windows ($f \ll 1$) are processed by the GNN, significantly reducing total cost.

To contextualize these results, we compare MemHookNet's complexity with representative baselines in Table 5.

Table 5: Algorithmic time complexity comparison of detection methods. All methods are evaluated on a single log window. N: log length,V: number of nodes, E: number of edges, d: hidden dimension, T_{Trans}: Transformer cost, f: suspicious log ratio

Model	Key component	Time complexity
DeepLog	LSTM	$O(Nd^2)$
Logs2Graphs	GCN + Graph Build	$O(V^2 + Vd + Ed)$
GLAD-PAW	GCN + Transformer	$O(Vd + Ed + T_{\text{Trans}})$
MemHookNet	LSTM + GATv2	$O(Nd^2 + f(V^2 + Ed))$

Note: Complexity reflects the per-window processing cost. $f \ll 1$ indicates early filtering reduces GNN usage frequency.

Compared with baselines like DeepLog and Logs2Graphs, MemHookNet achieves a balance between accuracy and efficiency by filtering low-risk data early and applying graph-based analysis only when necessary. This makes it suitable for real-time deployment.

5 Experimental Results and Analysis

This chapter provides a comprehensive and systematic analysis of the MemHookNet framework from four dimensions: detection performance, comparative experiments, ablation studies, and efficiency evaluation.

5.1 Detection Performance Evaluation

We evaluate the detection performance of MemHookNet and compare it with four representative baselines: DeepLog, LogAnomaly, Logs2Graphs, and GLAD-PAW. These methods differ in their modeling strategies, ranging from sequence learning to graph-based classification.

As shown in Table 6, MemHookNet achieves the best performance across all metrics, including 85.5% precision, 82.2% accuracy, 81.5% recall, 81.7% F1-score, and 94.7% AUC, with the lowest inference latency (15 ms). These results validate the model's practical effectiveness and stability in anomaly detection.

Table 6: Quantitative performance metrics of detection methods

Method name	Precision (%)	Accuracy (%)	F1 (%)	AUC (%)	Recall (%)	ET (ms)
DeepLog	70.7	70.5	71.5	80.2	68.0	50
LogAnomaly	84.1	80.9	76.8	84.2	78.3	100
Logs2Graphs	48.2	55.6	35.4	70.2	52.3	200
GLAD-PAW	85.1	81.2	76.0	87.3	83.0	120
MemHookNet	**85.5**	**82.2**	**81.7**	**94.7**	**81.5**	**15**

5.2 Comparison Experiment Analysis

To further contextualize these findings, we conduct comparative experiments under identical datasets and training conditions. The selected baselines span LSTM-based sequence models, template-based log analyzers, and graph neural networks. Table 6 presents the quantitative performance metrics, while Table 7 summarizes the functional capabilities of each method in terms of multi-class support, online processing, and deployment suitability.

Table 7: Functional comparison of mainstream detection methods

Method name	Model type	Multi-class support	Online processing	Remarks
DeepLog	LSTM Sequence	×	×	Suitable for general anomaly detection in logs
LogAnomaly	LSTM+Template	×	×	Enhances pattern and template matching ability
Logs2Graphs	OCDiGCN Graph	×	×	No flexible model and class adaptability, poor at classification
GLAD-PAW	GAT Graph	×	×	Focuses on performance, but not secure for non-safe logs
MemHookNet	LSTM+GATv2	✓	✓	Comprehensive framework for multi-class anomaly detection in logs

Unlike binary classifiers such as DeepLog and Logs2Graphs, which only distinguish between normal and anomalous behavior, MemHookNet supports real-time multi-class detection. It enables fine-grained classification of UAF, Double-Free, and Memory Leak anomalies. This design advantage contributes to its superior performance in detailed anomaly recognition, as shown in Tables 6 and 7.

According to the Tables 6, 7 and Fig. 6, MemHookNet significantly outperforms existing baselines in both detection accuracy and inference efficiency, achieving a 47.9% improvement in accuracy and a 92.5% reduction in latency compared to Logs2Graphs. These gains highlight the benefit of combining HOOK-based runtime logging with temporal–structural modeling.

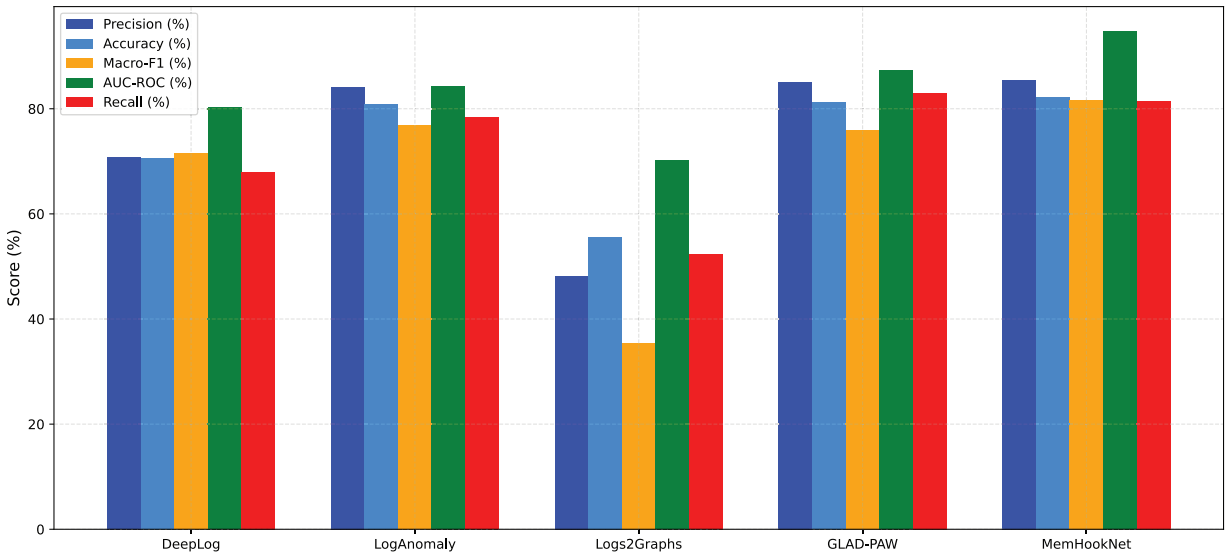
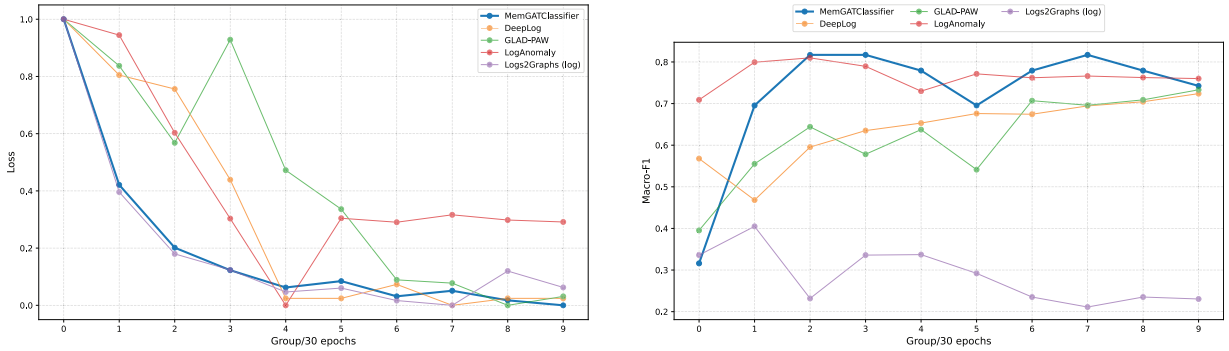


Figure 6: Comparison of detection performance with mainstream methods

Even when compared to strong models like GLAD-PAW and DeepLog, MemHookNet delivers higher classification accuracy with substantially lower computational cost. As shown in Fig. 7, it converges faster and reaches lower final loss values, demonstrating more stable and efficient learning. It also consistently maintains the highest macro-F1 scores throughout training, especially in early-stage groups, validating its ability to capture representative patterns from runtime memory logs. In contrast, Logs2Graphs struggles with convergence, while DeepLog and LogAnomaly exhibit slower learning and weaker generalization.



(a) (b)

Figure 7: Performance comparison of different detection models in terms of training loss and Macro-F1 across 10 training groups. As shown in **(a)**, MemHookNet achieves the fastest convergence rate and lowest final loss among all methods, demonstrating stable and efficient learning behavior. **(b)** shows that MemHookNet consistently maintains the highest Macro-F1 score across all training stages, especially in early groups, validating its strong multi-class anomaly detection capability

In summary, MemHookNet not only achieves high detection precision but also supports real-time multi-class anomaly detection with minimal overhead, making it a compelling solution for practical deployment in dynamic memory security scenarios.

5.3 Ablation Experiment Analysis

To evaluate the individual contributions of the modules (HOOK, LSTM, GNN) to the final performance, we designed three ablation experiments, each removing one of the modules to observe the performance changes. The experimental configurations are shown in Table 8:

Table 8: Ablation analysis of MemHookNet architecture

Model ID	Precision (%)	Recall (%)	F1-Score (%)	AUC-ROC (%)	MCC	ET (ms)
B1	84.3	80.0	83.1	89.2	0.67	20
B2	45.2	60.4	50.3	70.6	0.22	30
B3	58.2	61.1	60.2	76.1	0.41	35
Ours	**85.5**	**81.5**	**85.7**	**94.7**	**0.72**	**15**

As shown in the Table 8 and Fig. 8, MemHookNet outperforms the other comparison models on all key performance indicators (Accuracy, Precision, Recall, F1-Score), demonstrating its clear overall advantage. Notably, MemHookNet shows significant improvements in Precision and F1 score. Specifically, MemHookNet improves over:

- GATv2-noHook(B3) by 31.3% (85.5 → 54.2) in Precision and 35.5% (85.7 → 50.2) in F1 score;
- GCN-Graph(B2) by 70.3% (85.5 → 15.2) in Precision and 67.4% (85.7 → 18.3) in F1 score;
- Even compared to the relatively higher baseline model, LSTM-only(B1), MemHookNet(Ours) improves the F1 score by 2.6% (85.7 → 83.1), while also demonstrating stronger structural modeling capability.

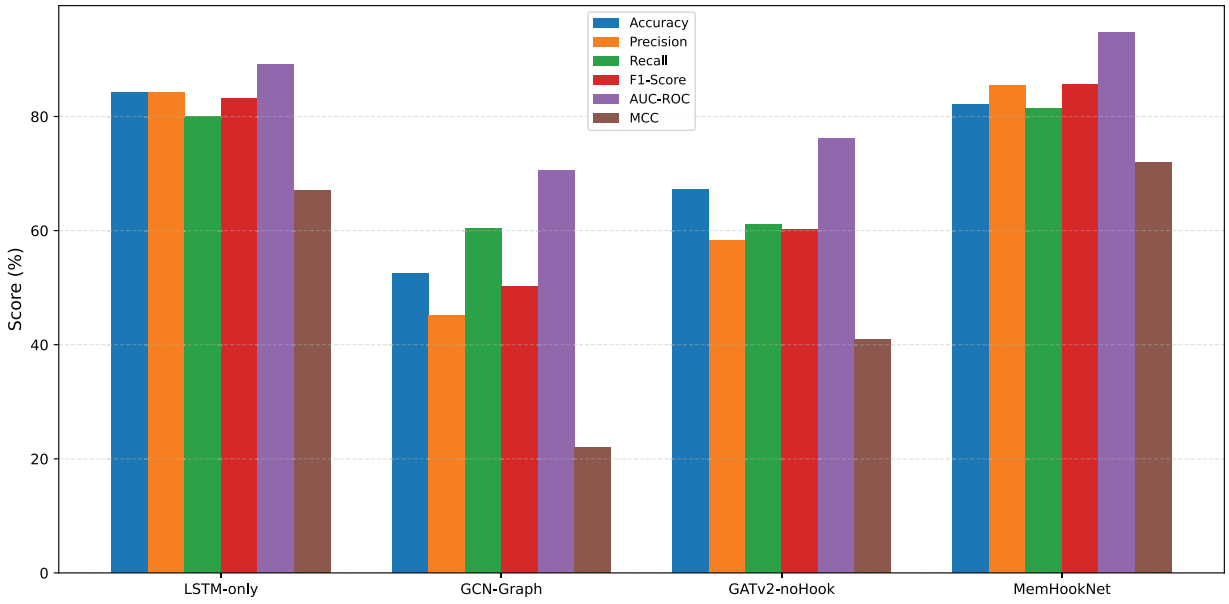


Figure 8: Ablation analysis of MemHookNet architecture

This result validates the effectiveness and advancement of the HOOK + LSTM + GNN hybrid architecture in multi-class recognition, structural modeling, and online processing tasks. In addition, a complexity analysis in Section 4.7 demonstrates that MemHookNet achieves a favorable trade-off between accuracy and efficiency, making it suitable for real-time deployment in practical systems.

5.4 Efficiency and Subsystem Performance Evaluation

To assess the practical efficiency of MemHookNet in real deployment scenarios, we evaluate both subsystem-level runtime latency and system-wide capability across multiple performance dimensions.

We first measure the average execution latency of each key module in the inference pipeline, including memory operation logging, LSTM-based filtering, graph construction, and GNN-based classification. The detailed latency breakdown is shown in Table 9.

Table 9: Average processing time for each stage of the system (Unit: Milliseconds)

Module	Average processing time (ms)
Log collection (HOOK)	0.02
LSTM inference	1.85
Graph construction (MemLogGraph)	4.17
GNN inference	5.93
Overall	≈12.0

These results demonstrate that MemHookNet achieves efficient online processing with an average end-to-end inference latency of approximately 12 ms per log fragment, which supports sub-second responsiveness for heap anomaly monitoring. This performance makes the system well-suited for high-frequency memory event surveillance in embedded or cloud-based environments.

To further contextualize the runtime efficiency and deployment suitability of MemHookNet, we conducted a high-level comparison against mainstream static, dynamic, and hybrid detection frameworks. As shown in Fig. 9 and Table 10, we evaluate five core dimensions that reflect practical deployment needs: Detection Granularity, Multi-class Support, Real-time Capability, Deployment Overhead, Extensibility.

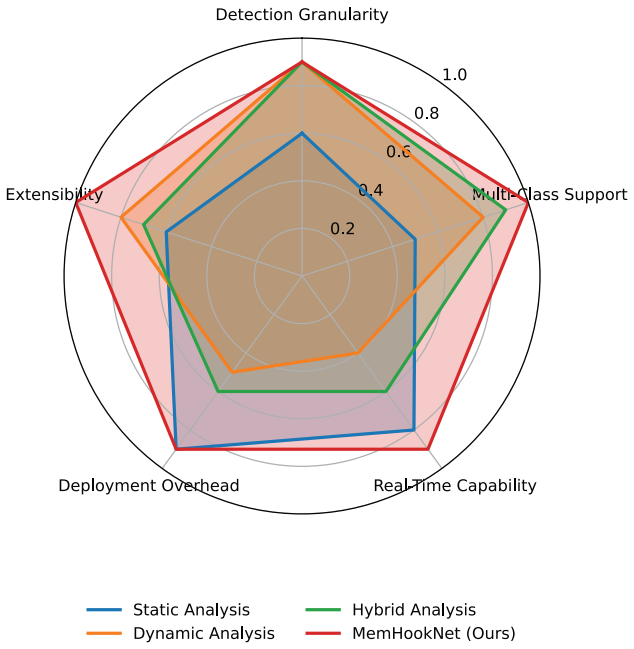


Figure 9: Comparison of memory vulnerability detection approaches

Table 10: Comparison of memory anomaly detection methods

Method category	Runtime execution	Multi-threading support	Performance overhead	Temporal dependency modeling	Dynamic feature extraction	Real-time capability	Automatic sample generation
Static analysis	×	×	✓	×	×	✓	×
Dynamic analysis	✓	✓	×	✓	✓	✓	×
Hybrid analysis	✓	(with static)	(partial support)	✓	✓	×	×
MemHookNet	✓	✓	✓	✓	✓	✓	✓

From the radar chart, we observe that MemHookNet consistently outperforms other methods across all dimensions: Compared to static analysis tools (e.g., Cppcheck), MemHookNet supports runtime and semantic modeling while maintaining low deployment cost. Compared to dynamic analysis frameworks (e.g., Valgrind, ASan), it dramatically reduces runtime latency (seconds vs. milliseconds). Compared to hybrid tools (e.g., Angr, Driller), MemHookNet avoids symbolic execution bottlenecks and enables end-to-end online processing.

The combined analysis confirms that MemHookNet strikes a highly favorable balance between detection capability and real-world usability, enabling real-time, fine-grained, and scalable heap anomaly detection that outperforms traditional tools in both responsiveness and deployment flexibility.

5.5 Robustness and Class-Level Analysis

To assess the robustness of MemHookNet under varying runtime conditions, we conducted a series of experiments simulating real-world variations in data noise, anomaly frequency, and thread-level concurrency. Specifically, we evaluated three robustness scenarios: Noise Injection: Random non-anomalous log entries were inserted into normal and abnormal sequences to simulate log noise or benign interference. We tested with noise rates of 0%, 10%, and 20%. Anomaly Rate Variation: The proportion of anomalous log windows was varied across 1%, 5%, 10%, 20% to assess the model's sensitivity under class imbalance. Thread Mixing: Memory operations from multiple threads were interleaved and randomly ordered to test the temporal resilience of the LSTM and GAT modules.

Fig. 10 shows the accuracy degradation across scenarios. MemHookNet demonstrates strong robustness in all settings, with less than 3.5% performance drop under 20% noise, and less than 2% reduction under severe class imbalance. This confirms the model's ability to retain high detection fidelity even in noisy or skewed environments, which are common in production systems.

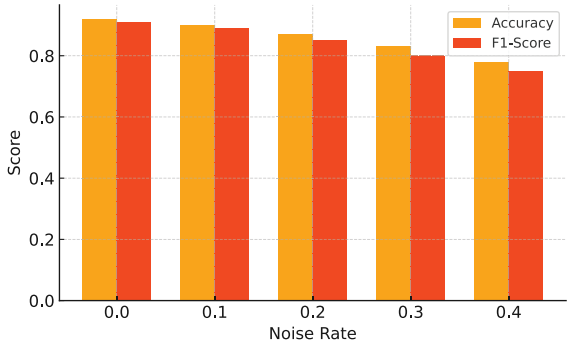


Figure 10: Accuray/F1 vs. noise rate

While MemHookNet is designed to classify memory behaviors into four categories—Normal, UAF, Double-Free, and Memory Leak—the boundaries between these classes can sometimes be subtle. To further understand classification behavior, we conducted a confusion matrix analysis on the validation set, along with per-class performance breakdown.

Fig. 11 shows the confusion matrix, highlighting that most misclassifications occur between UAF and Double-Free, due to their similar deallocation patterns. Memory Leaks are the most separable class, with high precision and minimal confusion.

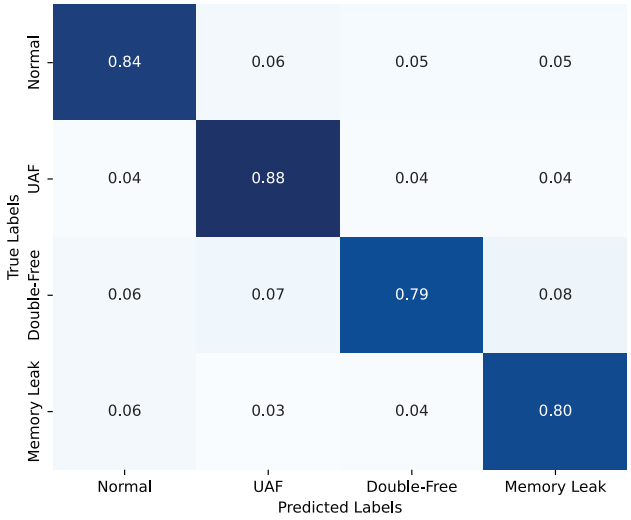


Figure 11: Confusion matrix for MemHookNet

To better understand how MemHookNet performs across different anomaly categories, we provide a class-wise breakdown in Table 11 and Fig. 10. The model achieves the highest F1-score for Use-After-Free (0.88), followed by Memory Leak (0.79) and Double-Free (0.78), showing balanced performance across categories. Double-Free detection is slightly less accurate due to overlap with UAF behaviors. In contrast, Memory Leaks are the most separable, with minimal confusion. These results validate the model's robustness in handling diverse heap anomaly types.

Table 11: Confusion matrix analysis

Class	Precision	Recall	F1-Score
Normal	0.84	0.86	0.85
UAF	0.88	0.87	0.88
Double-free	0.79	0.77	0.78
Memory leak	0.80	0.78	0.79

These results confirm that MemHookNet maintains strong discriminative capability across categories, particularly in distinguishing memory leaks. Additional feature tuning and graph relation encoding may further improve separation between UAF and Double-Free instances.

6 Conclusion and Future Work

This paper presents MemHookNet, a novel framework for real-time multi-class heap memory anomaly detection. It features a dynamic log interception mechanism that captures and structures memory operations in real-time using the LD_PRELOAD-based HOOK modules. The system combines time-series analysis with a sliding window mechanism and LSTM-based sequence filtering, improving detection efficiency and system responsiveness. Additionally, MemHookNet leverages a structural-aware graph neural network, using MemLogGraphBuilder for semantic graph construction and GATv2 for multi-class classification. Experimental results show that MemHookNet outperforms existing methods in detecting various heap memory anomalies, including UAF, Double-Free, and Memory Leaks.

Despite its strong performance, MemHookNet has some limitations. It currently relies on the LD_PRELOAD technique, which is specific to Linux. Future work will explore cross-platform compatibility using methods like DLL hooking or dynamic API interception. Expanding the detection scope to include other memory vulnerabilities, such as Heap Overflow and Uninitialized Read, would enhance its versatility. Further adaptability could be achieved by incorporating data augmentation, online learning mechanisms, or adversarial sample generation. Lastly, improving model interpretability through attention weight heatmaps and node contribution analysis would support debugging and enhance transparency. Future work will also focus on enhancing platform compatibility, broadening the range of detectable vulnerabilities, and improving adaptability and interpretability.

Acknowledgement: First and foremost, I would like to express my heartfelt gratitude to my supervisor, Professor Yan Zhuang, for his invaluable guidance and support throughout my studies and research. His insightful advice and encouragement in academic exploration, paper writing, and critical thinking have been a constant source of inspiration and motivation for me to improve and grow. I am also deeply grateful to the Zhengzhou University Infrastructure Laboratory for providing essential resources and experimental conditions that laid a solid foundation for the successful completion of my research. Lastly, I would like to extend my thanks to my classmates, friends, and family who have supported and encouraged me during this period. Your unwavering support has been instrumental in enabling me to stay focused on my academic pursuits and persevere through challenges

Funding Statement: The work was supported by Open Foundation of Key Laboratory of Cyberspace Security, Ministry of Education of China (No. KLCS20240211).

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Siyi Wang and Zhizhuang Zhou; methodology, Siyi Wang; software, Siyi Wang; validation, Siyi Wang, Menglan Li and Xinhao Wang; formal analysis, Siyi Wang; investigation, Siyi Wang; resources, Siyi Wang; data curation, Siyi Wang; writing—original draft preparation, Siyi Wang; writing—review and editing, Siyi Wang and Yan Zhuang; visualization, Menglan Li; supervision, Siyi Wang, Zhizhuang Zhou and Yan Zhuang; project administration, Siyi Wang, Xinhao Wang and Menglan Li; funding acquisition, Yan Zhuang. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Data available on request from the authors.

Ethics Approval: This study did not involve human or animal subjects.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Mitre. Stubborn Weaknesses in the CWE Top 25 [Internet]. 2024 [cited 2024 Jan 1]. Available from: https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html.
2. Yan H, Sui Y, Chen S, Xue J. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In: Proceedings of the 40th International Conference on Software Engineering; 2018 May 27–Jun 3; Gothenburg, Sweden. p. 327–37.
3. Chen J, Zhang C, Cai S, Zhang L, Ma L. A memory-related vulnerability detection approach based on vulnerability model with petri net. J Log Algebr Methods Program. 2023;132(3):100859. doi:10.1016/j.jlamp.2023.100859.
4. Murali A, Alfadel M, Nagappan M, Xu M, Sun C. AddressWatcher: sanitizerbased localization of memory leak fixes. IEEE Trans Softw Eng. 2024;50(9):2398–411. doi:10.1109/tse.2024.3438119.
5. Hao Y, Zhang H, Li G, Du X, Qian Z, Sani AA. Demystifying the dependency challenge in kernel fuzzing. In: Proceedings of the 44th International Conference on Software Engineering; 2022 May 25–27; Pittsburgh, PA, USA. p. 659–71.
6. Wögerer W. A survey of static program analysis techniques. Tech Rep. Vienna, Austria: Technische Universität Wien; 2005.
7. Gui B, Song W, Huang J. UAFSan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis; 2021 Jul 11–17; Online. New York, NY, USA: Association for Computing Machinery; 2021. p. 309–21.
8. He D, Gu H, Li T, Du Y, Wang X, Zhu S, et al. Toward hybrid static-dynamic detection of vulnerabilities in IoT firmware. IEEE Netw. 2020;35(2):202–7. doi:10.1109/mnet.011.2000450.
9. Du M, Li F, Zheng G, Srikumar V. Deeplog: anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security; 2017 Oct 30–Nov 3; Dallas, TX, USA. p. 1285–98.
10. Meng W, Liu Y, Zhu Y, Zhang S, Pei D, Liu Y, et al. Loganomaly: unsupervised detection of sequential and quantitative anomalies in unstructured logs. Int Joint Conf Artif Intell Organizat. 2019;19(7):4739–45.
11. Li Z, Shi J, Van Leeuwen M. Graph neural networks based log anomaly detection and explanation. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings; 2024 Apr 14–20; Lisbon, Portugal. p. 306–7.
12. Zamanzadeh Darban Z, Webb GI, Pan S, Aggarwal C, Salehi M. Deep learning for time series anomaly detection: a survey. ACM Comput Surv. 2024;57(1):1–42. doi:10.1145/3691338.
13. Lee B, Song C, Jang Y, Wang T, Kim T, Lu L, et al. Preventing use-after-free with dangling pointers nullification. In: The Network and Distributed System Security (NDSS) Symposium 2015; 2015 Feb 8–11; San Diego, CA, USA. p. 1–15.
14. Xu G, Lei W, Gong L, Liu J, Bai H, Chen K, et al. UAF-GUARD: defending the use-after-free exploits via fine-grained memory permission management. Comput Secur. 2023;125:103048. doi:10.1016/j.cose.2022.103048.
15. You Z, Cui L, Shen Y, Yang K, Lu X, Zheng Y, et al. A unified model for multi-class anomaly detection. Adv Neural Inf Process Syst. 2022;35:4571–84.
16. Huang K, Zhou S, Hu W, Gao Y, Pan F, Jiang X. Memory-guided hierarchical feature reconstruction for multi-class unsupervised anomaly detection. In: International forum on digital TV and wireless multimedia communications. Singapore: Springer Nature Singapore; 2024. p. 228–41. doi:10.1007/978-981-96-4276-2_16.
17. Landauer M, Onder S, Skopik F, Wurzenberger M. Deep learning for anomaly detection in log data: a survey. Mach Learn Appl. 2023;12(3):100470. doi:10.1016/j.mlwa.2023.100470.
18. Zhang H, Kim J, Yuan C, Qian Z, Kim T. Statically discover cross-entry use-after-free vulnerabilities in the linux kernel. In: Network and Distributed System Security (NDSS) Symposium 2025; 2025 Feb 24–28; San Diego, CA, USA. p. 1–17.
19. Cppcheck Team. Cppcheck software official website [Internet]. 2018 [cited 2024 Jan 1]. Available from: http://cppcheck.sourceforge.net/.
20. Wheeler DA. Flawfinder software official website [Internet]. 2018 [cited 2024 Jan 1]. Available from: https://www.dwheeler.com/flawfinder/.

21. Valgrind Team. Valgrind: a memory debugging, memory leak detection, and profiling tool; 2018 [Internet]. [cited 2024 Jan 1]. Available from: http://valgrind.org/.
22. Google. AddressSanitizer [Internet]. 2024 [cited 2024 Jan 1]. Available from: https://github.com/google/sanitizers/wiki/AddressSanitizer.
23. Van Der Kouwe E, Nigade V, Giuffrida C. Dangsan: scalable use-after-free detection. In: Proceedings of the Twelfth European Conference on Computer Systems; 2017 Apr 23–26; Belgrade, Serbia. p. 405–19.
24. Gorter F, Koning K, Bos H, Giuffrida C. DangZero: efficient use-after-free detection via direct page table access. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security; 2022 Nov 7–11; Los Angeles, CA, USA. p. 1307–22.
25. Ahn J, Lee J, Lee K, Gwak W, Hwang M, Kwon Y. BUDAlloc: defeating use-after-free bugs by decoupling virtual address management from kernel. In: Proceedings of the 33rd USENIX Conference on Security Symposium; 2024 Aug 14–16; Philadelphia, PA, USA. p. 181–97.
26. Shoshitaishvili Y, Brunton G, Rawat N. Angr: a platform for analyzing binary programs [Internet]. 2018 [cited 2024 Jan 1]. Available from: https://github.com/angr/angr.
27. Google Inc. Driller: a hybrid fuzzing tool combining static and dynamic analysis [Internet]. 2018 [cited 2024 Jan 1]. Available from: https://github.com/googleprojectzero/driller.
28. Choi K, Yi J, Park C, Yoon S. Deep learning for anomaly detection in time-series data: review, analysis, and guidelines. IEEE Access. 2021;9:120043–65. doi:10.1109/access.2021.3107975.
29. Wan Y, Liu Y, Wang D, Wen Y. Glad-paw: graph-based log anomaly detection by position aware weighted graph attention network. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining; 2021 May 11–14; Delhi, India. p. 66–77.
30. Zhang C, Peng X, Sha C, Zhang K, Fu Z, Wu X, et al. Deeptralog: trace-log combined microservice anomaly detection through graph-based deep learning. In: Proceedings of the 44th International Conference on Software Engineering; 2022 May 25–27; Pittsburgh, PA, USA. p. 623–34.
31. Kern P. Injecting shared libraries with LD_PRELOAD for cyber deception [master's thesis]. Vienna, Austria: Technische Universität Wien; 2023.
32. Zhao Z, Xu C, Li B. A LSTM-based anomaly detection model for log analysis. J Signal Process Syst. 2021;93(7):745–51. doi:10.1007/s11265-021-01644-4.
33. Braun P, Litz H. Understanding memory access patterns for prefetching. In: International Workshop on AI-assisted Design for Architecture (AIDArc), Held in Conjunction with ISCA; 2019 Jun 22; Phoenix, AZ, USA.
34. Haque R, Islam N, Tasneem M, Das AK. Multi-class sentiment classification on Bengali social media comments using machine learning. Int J Cogn Comput Eng. 2023;4(3):21–35. doi:10.1016/j.ijcce.2023.01.001.
35. Amadi CO, Odii JN, Okpalla C, La OCI. Emotion detection using a bidirectional long-short term memory (bilstm) neural network. Int J Curr Pharm Rev Res. 2023;4(11):1718–32.
36. Cao K, Zhang T, Huang J. Advanced hybrid LSTM-transformer architecture for real-time multi-task prediction in engineering systems. Sci Rep. 2024;14(1):4890. doi:10.1038/s41598-024-55483-x.
37. Setyanto A, Laksito A, Alarfaj F, Alreshoodi M, Oyong I, Hayaty M, et al. Arabic language opinion mining based on long short-term memory (LSTM). Appl Sci. 2022;12(9):4140. doi:10.3390/app12094140.
38. Brody S, Alon U, Yahav E. How attentive are graph attention networks? arXiv:2105.14491. 2021.
39. Nerrise F, Zhao Q, Poston KL, Pohl KM, Adeli E. An explainable geometric-weighted graph attention network for identifying functional networks associated with gait impairment. In: International Conference on Medical Image Computing and Computer-Assisted Intervention; 2023 Oct 8–12; Vancouver, BC, Canada. p. 723–33.
40. Zhao H, Wang Y, Duan J, Huang C, Cao D, Tong Y, et al. Multivariate time-series anomaly detection via graph attention network. In: 2020 IEEE International Conference on Data Mining (ICDM); 2020 Nov 17–20; Sorrento, Italy. p. 841–50.