

Research on Memory Leakage Detection and Optimization Methods for Android Applications

Wenhao Song

Information Networking Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, United States, 15213

Keywords: Android; memory leak; lifecycle; heap reference graph; dynamic analysis; program repair

Abstract: Memory leaks in Android applications generally occur because objects are still strongly referenced by GC roots after a component has been destroyed, leading to continuous growth of heap memory, frequent garbage collection, UI freezing, and eventually memory overflow. Given the shortcomings of the previous method, such as lacking lifecycle semantics, limited dynamic evidence coverage, and no risk constraints on remediation strategies, this paper proposes a lifecycle-reference chain collaborative LC-Hybrid method: record component states as event sequences, calculate retention paths with a heap reference graph, identify asymmetric releases of listeners, tasks, contexts, and heavy resources using static rules, and finally form tiered remediation recommendations through evidence fusion scoring. Based on the previous three years' public research and benchmark data, a supporting secondary statistics table has been prepared. According to the results from DroidLeaks, InferROI is a large-scale model intent inference and static analysis tool with a defect detection rate of 59.3% and an F1 score of 0.667; after fixing the problems, the total heap memory in the LeakPair public project decreased from 830.3 MB to 734.0 MB, and the project-level pairing showed significant improvement in the Wilcoxon test. Based on the above, this paper proposes an all-encompassing optimisation process of discovery, localisation and remediation, followed by regression validation, and clarifies its scope.

1. Research Background and Technical Objectives for Memory Leak Management in Complex Android Lifecycle Scenarios

With the development of Android applications, they are no longer simple single-Activity applications; now, they are multi-page applications that involve asynchronous tasks, reactive data streams and cross-process services, and thus their memory usage directly impacts the speed of startup, frame stability, and the frequency of crashes. The core problem of a memory leak is that objects are not technically "unreachable"; rather, they remain reachable via strong reference paths after they are no longer needed for business logic and thus form a retained set that grows proportionally with every user operation. Recent studies on the performance of Android have shown that more factors influencing it are not yet known, tools are lacking, and reproducible experiments

have not been carried out [1].

Large-scale empirical studies have found that there is no stable monotonic relationship between runtime, memory and power consumption in 215 Android applications and more than 1322 versions and over 27,900 real device execution scenarios. The same code practice may have different results on different devices and libraries [2]. Therefore, memory optimisation should not be reduced to lowering the peak amount of memory, but rather identify stable retained objects, lifecycle mismatches and user-path-triggered conditions. At the same time, a survey of 5183 open-source Android applications showed that only about 23 per cent of the projects have useful automated tests. Lack of test coverage cannot verify the stable operation and efficiency of the system following a leak-fix [3].

Based on this, this paper will address the following four problems: how to combine detection evidence, how to find the root cause of leaks, how to add remedial measures, and how to verify the effectiveness of the remediation. Rather than conducting experiments on individual devices, this paper uses public benchmarks for verifiable secondary statistics and proposes the LC-Hybrid method based on them. The following are the main contributions: Establishing a leak evidence model oriented towards the Android lifecycle; building a risk score that integrates heap graphs, event sequences and static semantics; proposing a low-intrusion, rollback-capable layered optimisation strategy; and providing statistical charts and evaluation criteria based on public data.

2. Current Status of Android Memory Leak Mechanisms, Detection, Location and Automatic Repair Technologies

2.1 Leakage Mechanism of Component Lifecycle Mismatch, Callback Residue, and Heavy Resource Consumption

Table 1: Classification of Root Causes and Evidence for Android Component Lifecycle, Callback Registration, Asynchronous Tasks, and Heavy Resource Leaks.

Leak pattern	Typical retainer	Runtime evidence	Priority
Lifecycle-retained component	Activity/Fragment/ViewBinding	Destroyed owner remains reachable after GC	Critical
Listener or observer	Anonymous callback/observer registry	Registration survives owner termination	Critical
Message and task	Handler / Runnable / Coroutine / Executor	Queued task retains outer object	High
Context misuse	Static singleton / cache	Activity context outlives screen	High
Heavy resource	Bitmap/WebView/Cursor/stream	Large retained payload or native peer	High
Collection growth	Map / list / queue	Monotonic retained-size growth	Medium

Android Runtime performs automatic garbage collection, but it only addresses the problem of releasing "unreachable objects"; it does not ensure that objects that have failed in a business context will be made unreachable in time. Activity, Fragment view, Adapter, Bitmap, WebView, Handler, coroutine task and listener can all extend their lifetime by using static fields, singleton cache, message queue or observer container. Specifically, in the image decoding and display stage, researchers have collected 216 inefficient image display problems from the 243 popular open-source Android applications, and it has been found that excessive decoding, main-thread decoding and unbounded caching will simultaneously increase heap usage and response latency [4].

Therefore, memory leak detection should also cover Java/Kotlin heap objects, native resources, caching strategies and life cycle states in the analysis.

Table 1 shows the six types of leaks: lifecycle components, callback registrations, asynchronous tasks, contexts, heavy resources and collection growth. The first two categories are generally of the "registration-deregistration" or "creation-destruction" asymmetric type and should be handled first. A small number of heavy resources can cause a large amount of memory pressure due to the native load of Bitmaps or WebViews. This division provides a single index for the above static rules, heap graph filters and repair templates.

2.2 Advances in Dynamic Heap Analysis, Static Protocol Checking and Automatic Patch Generation Methods

The three kinds of present methods are dynamic heap analysis, static resource protocol inspection and automatic repair. Dynamic methods find retained objects using heap snapshots, dominator trees and the shortest GC root path. The strength is that the evidence is particular; however, it is also prone to being based on the trigger path and has a high analysis cost. Static methods cover a large number of paths but lack semantic awareness of framework callbacks and custom encapsulations. InferROI infers the intention of resource acquisition, release and reachability verification through a large code model, and then combines it with two-stage static analysis to achieve defect detection rates of 59.3% and 62.5% on DroidLeaks and JLeaks, respectively [5]. Based on the above, it can be seen that although semantic inference can address the problems of hard-coded API pairs, there is still a false-positive rate exceeding 18%.

AddressWatcher extends the single-path Sanitizer evidence by adding dynamic address tracing across multiple test paths to pinpoint the root cause of 23 repair locations in the 50 C/C++ memory leaks and submit 25 repair requests, of which 21 have been merged [6]. Although it is not the object model of the Android managed heap, the idea of "accumulating evidence across test paths" and "mapping leak points to release locations" can be directly applied here. The Smart Status method monitors memory safety by observing changes in object state; although dynamic in accuracy, it has a relatively small runtime overhead, and state machines are suitable for describing the lifecycle of resources from creation to failure [7].

Cross-language research has also found that lifecycle boundaries are necessary. Canary has built a constraint model for Rust's semi-automatic memory management boundary and identified 19 leaky packages in more than 1,200 real packages, with an average analysis time of 8.4 seconds per package [8]. LeakPair actively generates release patches using repair patterns and verifies that the patches do not affect functionality through project testing [9]; Arodnep increased the proportion of repairable warnings in the NJR benchmark from 41% to 68% by resource wrapper specification inference and field inclusion analysis [10]. Based on the above experiments, reliable repairs should consider resource encapsulation, ownership boundaries, patch locations, and regression tests simultaneously.

Table 2 shows that the evidence in this paper includes an Android/Java resource leak benchmark, real-world device performance corpus and cross-domain remediation projects. DroidLeaks is used for detection performance comparison, JLeaks observes scale expansion, the mobile performance corpus supports the context of device fragmentation, and the LeakPair subset is employed to recalculate memory differences before and after remediation. The data sources cannot be combined to obtain a single accuracy rate; instead, the design of the method can be cross-validated in four dimensions: coverage, false positives, scale, and remediation benefits.

Table 2: Size and Uses of DroidLeaks, JLeaks, Mobile Performance Corpus, and LeakPair Public Datasets.

Dataset / corpus	Domain	Published scale	Role in this study
DroidLeaks	Java / Android resource leaks	86 defects; 172 snippets; 28 types	Detection trade-off
JLeaks	Java resource leaks	784 defects; 1,568 snippets; 368 types	Scalability reference
Mobile performance corpus	Android apps on physical devices	215 apps; 1,322 versions; >27,900 scenarios	External validity
LeakPair U1-U10	SPA memory leaks	10 published unknown-leak projects	Repair-effect reanalysis

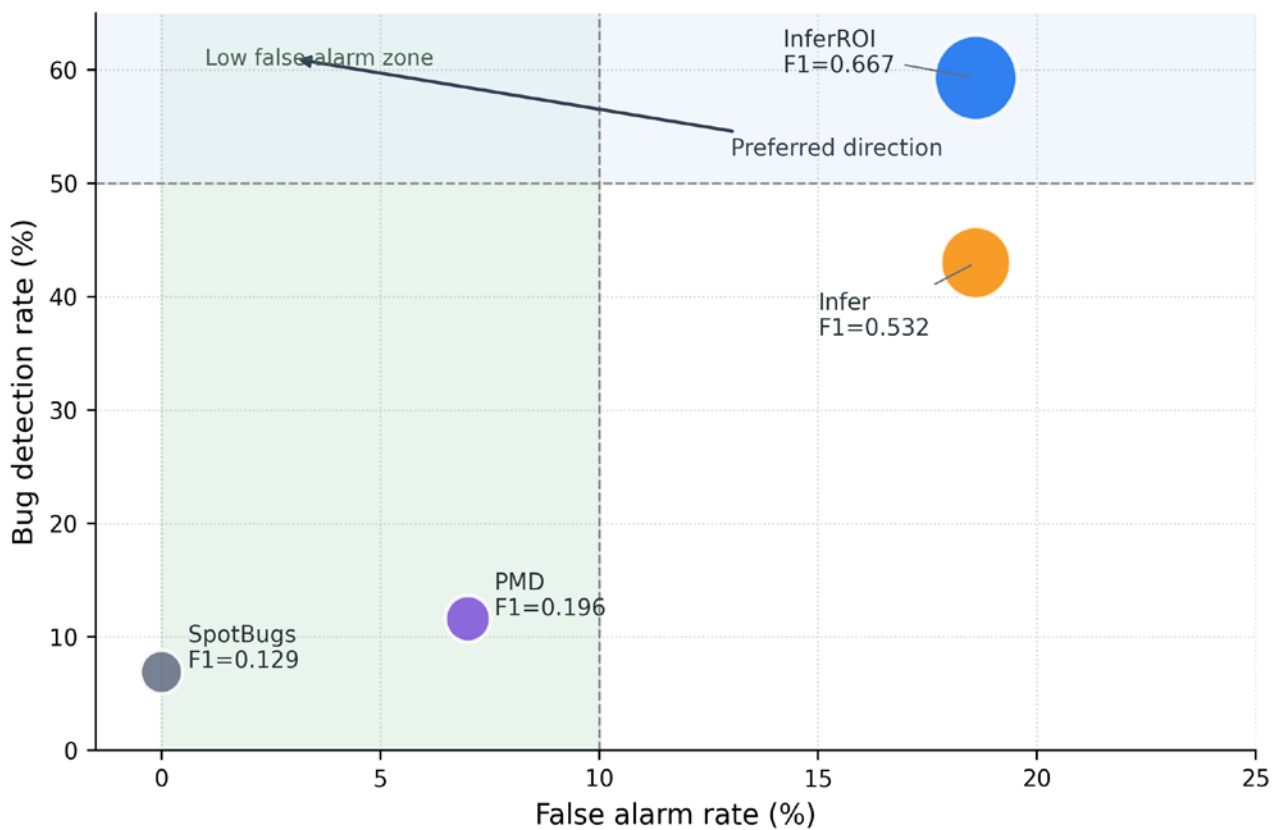


Figure 1: Trade-off between Detection Rate and False Alarm Rate for SpotBugs, PMD, Infer, and InferROI on the DroidLeaks Benchmark.

Figure 1 shows the false alarm rate on the horizontal axis and the defect detection rate on the vertical axis, and the area of the bubble represents the F1 score. SpotBugs and PMD had a low false-alarm rate, but their detection rates were only 6.9% and 11.6%, respectively. InferROI raised the detection rate to 59.3% and the false alarm rate to 18.6%, achieving an F1 score of 0.667; thus, it had a better general trade-off. Based on the above results, a hybrid model of "semantic inference + static verification" will be used to reduce false alarms further through dynamic evidence and threshold grading in engineering deployment.

Table 3: Comparison of TP, FP, BDR, FAR and F1 for the four types of resource leakage detectors on the DroidLeaks benchmark.

Detector	TP	FP	BDR (%)	FAR (%)	F1
SpotBugs	6	0	6.9	0.0	0.129
PMD	10	6	11.6	7.0	0.196
Infer	37	16	43.0	18.6	0.532
InferROI	51	16	59.3	18.6	0.667

Table 3 shows the original indicators corresponding to Figure 1. InferROI increased true positives by 14 compared with Infer, while keeping the number of false positives the same; thus, by combining resource intent and path verification, effective coverage was enhanced, but 16 false positives still increased the cost of manual verification. Therefore, the method proposed in this paper does not directly use a single static alarm as the basis for remediation, but requires that lifecycle mismatch, retained paths, and repeated triggering growth form at least two independent types of evidence.

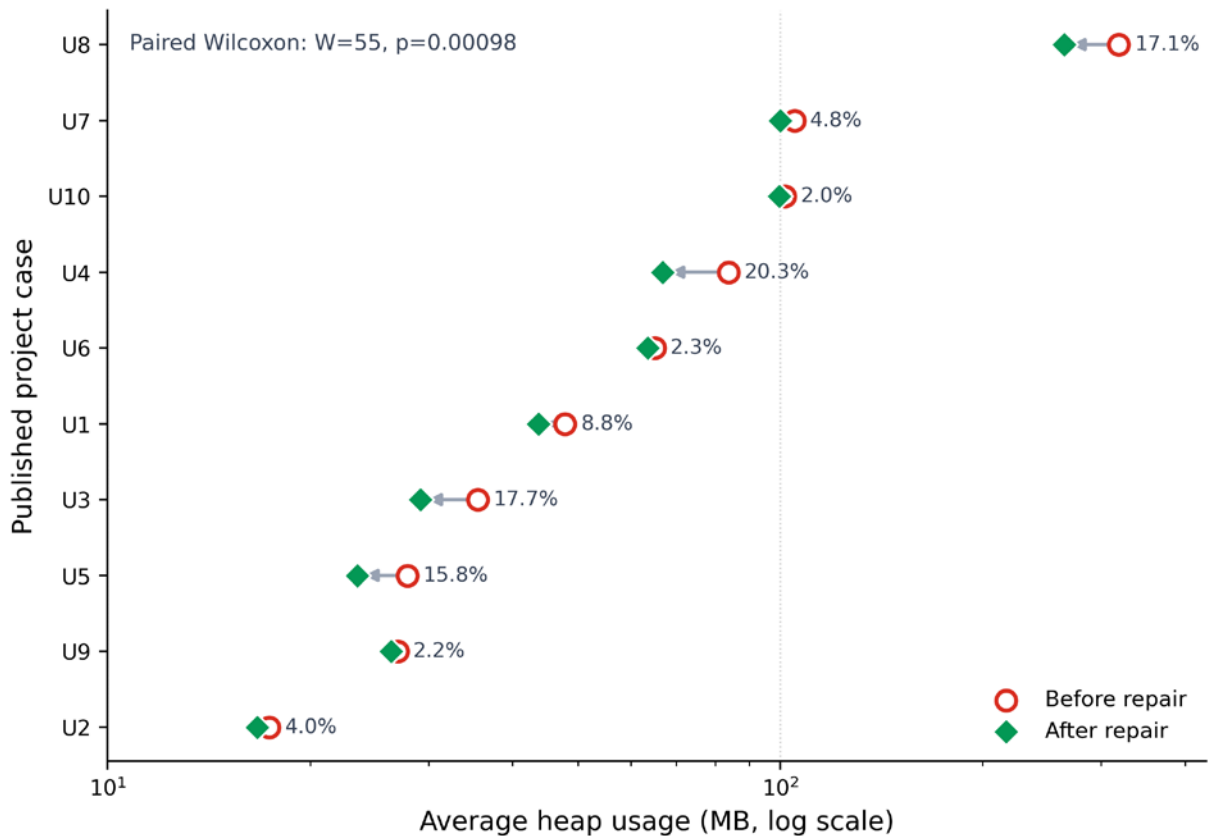


Figure 2: Paired Changes in Average Heap Usage Before and After Memory Leak Remediation for Ten LeakPair Public Projects.

Figure 2 shows the average heap occupancy before and after remediation for 10 publicly disclosed projects in a dumbbell plot, and arrows indicate a decrease in both values. Based on the mean values from the original report, the total heap occupancy decreased to 734.0 MB from 830.3 MB, a reduction of 96.3 MB; the average reduction per project was 9.5% and the median reduction

was 6.8%. There were arithmetic errors in the reduction columns for U1, U4 and U5 of the original table; therefore, this paper uses the recalculated mean values. The project-level one-sided Wilcoxon test showed $p=0.00098$.

3. Key Bottlenecks in Lifecycle Forensics, Heap Diagram Analysis and Patch Verification in Real-World Application Deployment

3.1 Lifecycle Expiration Determination and the Problem of Segmented Evidence in the GC Root Reference Chain

Heap dumps can tell us "who referenced the object", but not directly "whether the object is invalid in the business logic". Static analysis can find callback registration and resource release, but it is difficult to determine if a particular path was triggered on the actual device. If only large objects in a single snapshot are considered, caching and preloading will be prone to error; if only lifecycle callbacks are used, shared objects across pages may be incorrectly deleted. Therefore, component state, object growth trends and GC root paths need to be unified in the same evidence space. The retained heap growth rate of object o at the two stable sampling times is defined as:

$$g_o = [H_o(t_2) - H_o(t_1)] / (t_2 - t_1) \quad (1)$$

$H_o(t)$ is the size of the reserved heap that is dominated by or associated with object o , and t_1 and t_2 are stable sampling times obtained after completing the same user operation cycle. A continuously positive g_o does not prove leakage by itself; however, it can rule out a large number of single peaks.

3.2 Insufficient coverage of low-frequency triggering paths and noise in large-scale heap reference graphs

Android leaks often only occur during screen rotation, repeated page navigation, switching from a background process to the foreground, or retrying a network failure. A single test lacks coverage, and a full heap graph contains hundreds of thousands to millions of objects; direct traversal is thus prohibitively expensive. To quantify the long-term discrepancy in object states at various stages, a persistence mismatch rate is used:

$$p_o = (I_1 + I_2 + \dots + I_n) / N \quad (2)$$

As shown in equation (2), n is the N th checkpoint; if object o is still reachable from the GC Root in the k th heap graph and the component state has exceeded its allowed lifespan, then the indicator variable I_k takes the value 1; otherwise, it takes the value 0. The closer p_o is to 1, the more likely the object is to continuously violate lifespan constraints at multiple checkpoints.

3.3 Problem of unified measurement for leak alarm prioritisation, remediation benefits and behavioural damage risks.

The same reference chain can be caused by anonymous inner classes, third-party SDKs or system frameworks. To save memory, some references are removed; however, it may lead to listener failure or lost state upon early task termination. Existing tools often separate "leak discovery" from "patching" and do not provide an all-encompassing risk-benefit-rollback cost prioritization system. In addition, the F1 score of static detection, the location success rate of dynamic tools, and the memory reduction after repair are all different types of indicators and cannot be substituted for one another.

4. LC-Hybrid Multi-source Evidence Fusion Detection and Low-Risk Stratification Optimisation Method

4.1 Closed-loop process based on high-risk scenario reproduction, weak reference sentinels, and on-demand dumping

The five modules of the LC-Hybrid structure are: First, it builds reusable scenario scripts that cover high-risk paths such as page reconstruction, foreground/background switching, asynchronous failures, list scrolling, and WebView exit. Second, weak reference sentinels are placed at critical lifecycle boundaries to collect lightweight object counts, and a heap dump is only triggered when an object times out and is not released. Thirdly, it carries out package-level whitelisting, dominance tree pruning and shortest root path extraction on the heap graph, and employs static analysis to find listener registrations, Handler messages, coroutine scopes, static Contexts and resource closure protocols. Fourth, it combines multi-source evidence to form a risk score and issues tiered recommendations. Fifth, it conducts pre- and post-repair regressions on the same device, script and warm-up conditions.

4.2 Leakage Location Model Integrating Reserved Heap Growth, Lifecycle Mismatch and Shortest Root Path

For a candidate object o , let $\hat{g}_o$ be the normalised growth rate, p_o be the persistence mismatch rate, d_o be the shortest strong reference distance from the object to the GC Root, and c_o be the static asymmetric confidence score. The leakage risk score is then:

$$r_o = \sigma(wg \cdot \hat{g}_o + wp \cdot p_o + wd / (1 + d_o) + wc \cdot c_o) \quad (3)$$

Equation (3) is the Sigmoid function, and wg , wp , wd , and wc are non-negative weights obtained from the validation set. A higher $\hat{g}_o$ and p_o indicate stable growth and lifecycle default; a shorter d_o is a more direct root path, and c_o reflects an imbalance in the register-deregister, create-destroy, or open-close protocol. r_o in engineering is divided into three tiers: immediate, warning and blocking; thus, low-confidence alarms will not be treated as such.

4.3 Layered Patch Selection for Listeners, Asynchronous Tasks, Context and Heavy Resources

The five layers of the optimisation strategy are ordered by the degree of invasiveness: First, unregister listeners and callbacks in `onStop`, `onDestroyView`, or `onDestroy`; Second, bind tasks to `viewLifecycleOwner` or a cancelable scope, and clean up Handler messages and Executor tasks; Third, replace Activity Contexts that do not require UI semantics with Application Context, and prohibit static fields from holding views; Fourth, perform explicit release, capacity constraint or on-demand loading for Bitmap, WebView, Cursor, and Stream objects; Fifth, refactor cross-page caching and third-party SDK encapsulation. The selection target of the candidate patch P is:

$$P_{opt} = \operatorname{argmin} [\alpha(H_a/H_b) + \beta(T_a/T_b) + \gamma C(P) + \lambda B(P)] \quad (4)$$

As shown in Equation (4), P_{opt} is the candidate patch with the lowest total cost, H_b and H_a are the stable heap usage before and after the fix, T_b and T_a are the GC pause or critical scenario times, $C(P)$ is the code modification complexity, $B(P)$ is the risk of behavioural disruption, and α , β , γ and λ are weights. It is not to reduce memory consumption, but rather to improve operating efficiency and lower the maintenance expenses of functional safety at a reasonable cost.

4.4 Regression Validation System Based on Detection Accuracy, Stable Heap Occupancy and Interaction Delay

Each patch should be run in at least four scenarios: preheating, repeated operation, background persistence, and recovery, and stable heap size, GC count, long frame ratio and functional test results should be recorded. P, R and F1 score are employed to assess the quality of the alarm in the detection stage.

$$F_1 = 2PR/(P + R), \quad P = TP/(TP + FP), \quad R = TP/(TP + FN) \quad (5)$$

TP, FP and FN are the true positives, false positives and false negatives in equation (5). For the repair effect, absolute memory change, relative decrease, paired t-test p-value and effect size should be reported simultaneously; if the distribution is not normal, a Wilcoxon signed-rank test should be used instead. Only after passing the functional test, a significant decrease in stable heap occurs, and critical interaction latency does not increase, will the patch be added to the release branch.

4.5 Engineering Implementation Boundaries for Continuous Integration Environments, Third-Party SDKs, and Native Memory Scenarios

LC-Hybrid is suitable for incremental checks in continuous integration: static protocol matching and lightweight unit tests are performed at the commit stage; nighttime tasks run device scenarios and analyze heap graphs; and long-loop stress tests are conducted on high-risk modules before release. For third-party closed-source SDKs, the effect can be reduced by using external reference chains, version comparison and process isolation; for system caches, graphics drivers, and JNI native memory, only the Java heap is not sufficient for diagnosis, and native memory sampling and system tracing should be employed in combination. Public benchmarks show that hybrid methods can improve coverage, but the secondary analysis in this paper cannot replace verification at the target application, target device and real business traffic levels.

5. Optimization Effect, Application Limitations and Engineering Application Prospects of LC-Hybrid Detection

This paper addresses the problems of missing lifecycle semantics, high heap graph noise, insufficient triggering paths and difficulty in quantifying remediation risks for Android application memory leaks. It proposes an LC-Hybrid detection and hierarchical optimisation method for lifecycle-reference chain evidence. Repetitive Scenarios and weak reference sentinels are used to trigger dynamic evidence collection, retention sources are located via heap dominance relationships and the shortest GC root path, asymmetric management of listeners, asynchronous tasks, Context, and heavy resources is identified through static protocol analysis, and a closed loop from alerting to regression verification is achieved by risk scoring and multi-target patch selection. Based on the analysis of the above public data, by combining semantic inference with static verification, the detection coverage has been enhanced; at the same time, pattern-based remediation has also reduced heap memory consumption to some extent. The main deficiency of this study is the absence of end-to-end experiments on a unified Android device matrix for the LC-Hybrid prototype, and the LeakPair results are also cross-domain evidence. Future research will focus on building an Android-specific leak benchmark that includes Kotlin coroutines, Jetpack Compose, WebView, JNI and third-party SDKs, introduce online anonymous sampling and causal comparison, and further investigate automatic patch generation that can prove behavioural equivalence.

References

- [1] Liao D, Pan S, Yang S, Wang Y. Automatically Analyzing Performance Issues in Android Apps: How Far Are We? *arXiv:2407.05090*, 2024. DOI: 10.48550/arXiv.2407.05090.
- [2] Rua R, Saraiva J. A Large-Scale Empirical Study on Mobile Performance: Energy, Run-Time and Memory. *Empirical Software Engineering*, 2024, 29: 31. DOI: 10.1007/s10664-023-10391-y.
- [3] Mahmud T, Che M, Ngu AHH, Yang G. Why Android App Testing Falls Short: Empirical Insights from Open-Source Projects and a Practitioner Survey. *Empirical Software Engineering*, 2025, 30: 163. DOI: 10.1007/s10664-025-10726-x.
- [4] Murali A, Alfadel M, Nagappan M, Xu M, Sun C. AddressWatcher: Sanitizer-Based Localization of Memory Leak Fixes. *IEEE Transactions on Software Engineering*, 2024, 50(9): 2398–2411. DOI: 10.1109/TSE.2024.3438119.
- [5] Guo, X. (2025, April). Research on Financial Trading Algorithms Based on Deep Reinforcement Learning. In *2025 IEEE 3rd International Conference on Control, Electronics and Computer Technology (ICCECT)* (pp. 1711-1715). IEEE.
- [6] Guo, X. (2025, March). Research on Blockchain-Based Financial AI Algorithm Integration Methods and Systems. In *2025 IEEE International Conference on Electronics, Energy Systems and Power Engineering (EESPE)* (pp. 816-821). IEEE.
- [7] Han, W. (2025, November). An Empirical Study on Building a High-precision Forecasting System for Financial Market Trends Based on the PyTorch Deep Learning Framework. In *International Conference on Computational Technologies for Research in Data Analytics* (pp. 518-528). Cham: Springer Nature Switzerland.
- [8] Jing, X. (2025, November). In-Depth Analysis of the Optimization Decision Model of Financial Investment Portfolio Based on Machine Learning Algorithms in Python Scikit-Learn Library. In *International Conference on Computational Technologies for Research in Data Analytics* (pp. 130-140). Cham: Springer Nature Switzerland.