

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Exposing Resource-Exhaustion DoS Vulnerabilities with Leak-Oriented Minimum Path Covers

Lige Zhan and Yafei He, *Wuhan University;* Jiang Ming, *Tulane University;* Guojun Peng and Jianming Fu, *Wuhan University*

https://www.usenix.org/conference/usenixsecurity26/presentation/zhan

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Exposing Resource-Exhaustion DoS Vulnerabilities with Leak-Oriented Minimum Path Covers

Lige Zhan*
Wuhan University
ligezhan@whu.edu.cn

Yafei He*
Wuhan University
yaf2023@whu.edu.cn

Jiang Ming†
Tulane University
jming@tulane.edu

Guojun Peng*
Wuhan University
guojpeng@whu.edu.cn

Jianming Fu*,‡
Wuhan University
jmfu@whu.edu.cn

Abstract

Resource leaks—failures to properly release scarce Garbage Collection (GC)-unmanaged resources such as file descriptors, sockets, and database connections—are a pervasive root cause of resource-exhaustion denial-of-service (DoS) risks in Java programs. Prior approaches typically enumerate execution paths and inspect them to identify resources that are not properly released. However, exceptions can trigger non-local control transfers that bypass resource releases, and determining which such exceptions are leak-relevant requires context-sensitive reasoning beyond simple heuristics. Worse still, exhaustive path enumeration may trigger path explosion, significantly reducing the practicality of existing methods.

Therefore, we develop *LeakHunter*, a novel Java resource leak detector. LeakHunter first leverages an LLM to infer leak-relevant exceptions in context and statically validates them. Next, we propose a **L**eak-oriented **M**inimum **P**ath **C**over (**LMPC**) technique. Our insight is that there is no need to enumerate all paths: since (1) leak detection is existential, a single leak-prone path suffices to demonstrate that a resource-managing object is leak-prone, and (2) we can cover the resource-relevant code regions by leveraging a minimum path cover. Therefore, we aim to select a representative path set generated from MPC while being biased toward leak-prone behaviors. Accordingly, we construct witness paths from multiple MPCs and prioritize those with more acquisitions and fewer releases as the LMPC, as they are more likely to expose leaks. Across two public benchmarks, LeakHunter achieves the best detection performance while providing an **11.4**× speedup in path analysis over the latest approach. In our real-world study, LeakHunter identifies 67 true leaks with potential DoS impact, including **15** confirmed by developers.

1 Introduction

Availability is a first-class security property [1]. In Java applications and services, resource leaks can be weaponized into resource-exhaustion denial-of-service (DoS) conditions [2, 3]. By repeatedly exercising a leak-prone path, an attacker can gradually deplete acquired resources, eventually stalling critical functionality, triggering cascading failures, or crashing the process [4]. A resource leak occurs when a program acquires a system resource but fails to release it promptly or correctly [5]. In Java, garbage collection (GC) [6] does not reclaim many underlying system resources (e.g., file descriptors, sockets, and cursors), which we refer to as *GC-unmanaged* resources. Developers must therefore explicitly close or release them [7–10]. Although a single leak may not immediately trigger a failure, its impact accumulates over time. As a result, resource leaks are particularly dangerous in Java programs, where a latent defect can be repeatedly triggered and escalated into a reliable resource-exhaustion DoS condition.

Resource leaks and their potential severity have attracted sustained attention from the research community and industry. Over the past decade, numerous resource leak detection tools have been proposed, including academic prototypes [7, 8, 10–12] and industrial systems [13, 14]. Although their designs vary, a common strategy is to first identify resource-management APIs that perform resource acquisition and release operations. Next, they perform control-flow analysis and enumerate end-to-end witness paths from a specific entry point (e.g., a method entry) to an exit point (e.g., a return). Finally, they analyze each path to determine whether every acquired resource is released before the end of its lifetime. Despite notable progress, existing resource leak detectors still have important limitations, mainly due to their path analysis strategies, including the following two issues.

1) Incomplete Leak-Relevant Exceptions Handling Programs often contain various exceptions that can trigger non-local control transfers, bypassing the intended release operations. Because exhaustively enumerating exception-induced paths incurs substantial overhead and can hurt precision by introducing many infeasible exceptional transfers [7], recent approaches [10] adopt heuristic rules to approximate the impact of leak-relevant exceptions, trading precision for scalability. Unfortunately, exceptions are diverse, and their leak

‡Corresponding author

relevance depends on the surrounding semantic context. As a result, these rule-based heuristics often miss leak-relevant exception paths. More broadly and reliably analyzing the impact of leak-relevant exceptions therefore remains challenging.

2) Path Explosion Even with improved modeling of abnormal control flow, exhaustively enumerating possible paths is rarely practical for large codebases. Although prior work has recognized this problem and attempted to mitigate it, existing heuristic strategies exhibit clear limitations in both their applicability across diverse scenarios and their overall effectiveness. For example, the state-of-the-art tool InferROI [10] heuristically prunes paths that are unrelated to acquisition or release operations. However, heuristic pruning often yields limited benefits because even the set of paths that involve resource-management operations can still be extremely large in complex codebases. Consequently, InferROI explicitly acknowledges path explosion as a major limitation [10], especially for complex programs. Addressing the overhead caused by an excessive number of paths remains a key challenge.

Our Insight Two key observations guide our design. **First**, leaks often hinge on *leak-relevant exceptional transfers*. However, rule-based detectors typically handle exceptions conservatively for efficiency: heuristic rules struggle to pick out the few leak-relevant exceptions from many candidates because feasibility depends on the surrounding semantic context. We therefore augment static analysis with an LLM's semantic understanding [15, 16] to infer candidate leak-relevant exceptions, which are then statically validated before being incorporated into subsequent analysis. **Second**, leak detection is essentially an existential property: finding a single leak-prone path is sufficient to flag a resource-managing object as vulnerable. The goal is thus to surface such paths efficiently rather than to achieve path completeness over an exponentially large search space. Inspired by path-cover prioritization work [17], we use minimum path covers (MPCs) [18] as a compact backbone and incorporate domain knowledge to prioritize paths that are more likely to expose leaks. Taken together, these insights call for a detector that (i) exposes leak-relevant exceptional transfers and (ii) efficiently searches for a small set of high-value leak witnesses.

In this paper, we present *LeakHunter*, a novel system for detecting GC-unmanaged resource leaks that can lead to resource-exhaustion DoS vulnerabilities. LeakHunter first leverages an LLM to infer leak-relevant exceptions in context and validates the results through static analysis. This step guides subsequent analysis by identifying paths whose exceptional transfers may bypass resource releases and thus lead to leaks. Building on the LLM outputs, we introduce a novel leak-oriented minimum path cover (LMPC) to replace traditional exhaustive path enumeration. LMPC first prunes resource-irrelevant code regions (i.e., regions independent of resource acquisition and release operations), and then decomposes the program's control-flow graph into a set of directed acyclic graphs (DAGs) [19]. Next, it computes MPCs [18] over the remaining DAGs. Building on the generated MPCs, LeakHunter further constructs end-to-end witness paths and prioritizes those that are more likely to expose leaks by favoring paths with more acquisitions and fewer releases, thereby yielding a compact set of representative, leak-prone paths (i.e., the LMPC) for verification. Finally, LeakHunter analyzes each representative path in the LMPC to determine whether each acquired resource is properly released.

We conducted a series of experiments on representative public benchmarks with ground truth, such as DroidLeaks [20] and JLeaks [5], to evaluate LeakHunter's performance. Our comparative analysis included one state-of-the-art research tool, InferROI [10], and two widely used industrial analyzers, PMD [21] and SonarQube [22]. The results show that LeakHunter outperforms all baselines. Moreover, LeakHunter achieves an **11.4**× speedup over the latest approach, InferROI, in terms of path analysis runtime. In the real-world study, LeakHunter identifies 90 true leaks, of which 67 can potentially lead to resource-exhaustion DoS attacks, including **15** that have already been confirmed by developers. We further analyze a representative case showing that repeated triggering of a GC-unmanaged resource leak can exhaust underlying resources and cause the application to fail. In summary, this paper makes the following contributions:

- We reframe GC-unmanaged resource leaks as an availability risk and show how repeatedly triggering leak-inducing paths can escalate into resource-exhaustion DoS conditions.
- We introduce an LLM-assisted inference mechanism for leak-relevant exceptional transfers, augmenting static rules to better expose exception-induced leak paths.
- We propose LMPC, a leak-oriented minimum path cover-based path summarization algorithm that generates a compact, prioritized set of representative leak-prone paths, mitigating path explosion without enumerating all feasible paths.

2 Background & Related Work

We first ground our study in a threat model focused on availability risks from GC-unmanaged resource leaks. We then categorize common leak patterns, and finally introduce prior work, highlighting the gaps that motivate our design.

2.1 Threat Model and Security Scope

We focus on GC-unmanaged resource leaks in Java programs. A GC-unmanaged resource leak occurs when a program acquires an OS- or runtime-managed resource that is not reliably reclaimed by Java garbage collection (e.g., file descriptors, sockets and cursors) but fails to release it (e.g., by calling `close()`) on some control-flow paths. Because

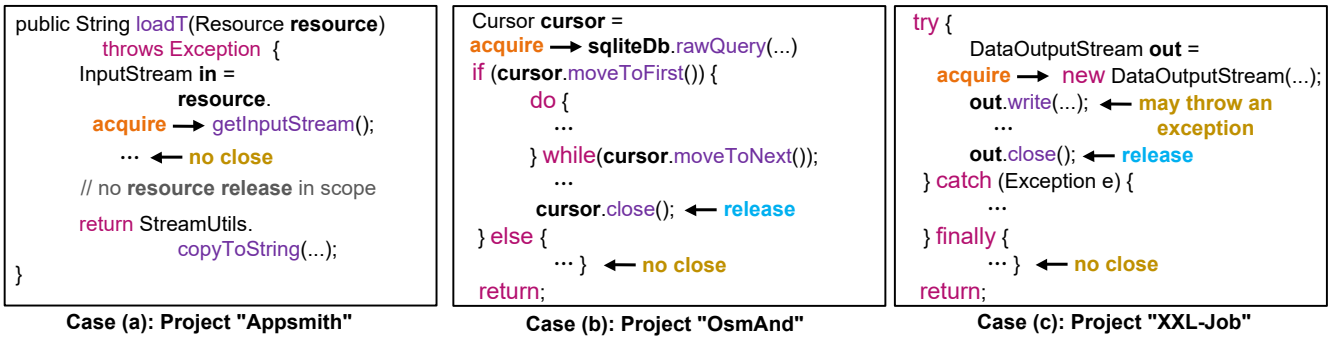


Figure 1: Major types of resource leaks in real-world applications.

such resources are outside the scope of Java garbage collection, leaked instances may accumulate and eventually exhaust bounded resource budgets, resulting in availability failures in the form of resource-exhaustion DoS when the leak is repeatedly triggered. We consider a worst-case setting where an adversary, or simply an adversarial workload, can repeatedly trigger resource leaks by exercising externally exposed input interfaces, request sequences, or other means to drive the program into leak-inducing behaviors, ultimately crashing the program. Our goal is to identify potential GC-unmanaged resource leaks in Java programs that may constitute the root cause of resource exhaustion, and to output concrete leak witnesses that support security auditing and prioritization for availability risks.

2.2 Resource Leak Types

Resource leaks are pervasive in real-world applications and even in online code examples [23]; they are widely regarded as among the most challenging problems in static program analysis [24]. The prevalence and persistence of resource leaks can be attributed to two factors. First, leaks are often latent defects whose effects accumulate gradually before manifesting as observable failures. As a result, an attacker can stealthily and repeatedly trigger leak-prone executions to progressively exhaust resources, undermining system availability and potentially enabling resource-exhaustion DoS attacks. Second, lifecycle management is intrinsically complex: nonlinear control-flows such as exceptional paths are ubiquitous and can easily bypass timely release along those paths [7, 10]. Resource leaks can typically be divided into two categories: *always-unreleased* leaks and *path-dependent* leaks.

Always-Unreleased Leaks In these cases, resources are never closed within the program scope. This usually occurs when developers neglect release operations entirely during implementation. In some cases, libraries or components expose acquisition operations without an explicit, publicly accessible release interface, making it difficult for downstream users to reliably release the resource in a disciplined way. A typical example is the leak observed in *Appsmith* (Figure 1(a) [25]).

Path-Dependent Leaks In these cases, developers do implement release operations in the code, but some execution paths still skip them, leading to path-dependent leaks. Broadly, we observe two common patterns of such leaks. First, *no-close-branch cases*: these occur when a resource is acquired before a conditional, but the matching release is placed in only one branch, while the other branch(es) reach a method exit without releasing the resource, leading to a leak. A representative example from the *OsmAnd* app is shown in Figure 1(b) [26], where the resource is leaked because it is not released in the *else* branch. Second, *exception-induced cases*: exceptions can abruptly redirect the control-flow away from the intended release site if they are not handled with appropriate cleanup. The leak in *XXL-Job* (Figure 1(c) [27]) exemplifies this pattern: an exception thrown by `out.write()` can bypass the subsequent release call `out.close()`, and the `finally` block does not enforce the missing cleanup, resulting in a leak. Compared with always-unreleased leaks, path-dependent leaks are more insidious because the resource is correctly released on many executions but silently leaked along these special paths, which makes them harder to detect and reproduce in practice.

2.3 Related Work

Automated resource leak detectors have been developed to address the growing severity of resource leaks and resource-exhaustion DoS attacks. Tracker [7], an early approach, first uses predefined rules to filter leak-relevant exceptions and then performs inter-procedural analysis to check for resource leaks along enumerated paths. Relda2 [8] is a path-aware resource leak detector for Android bytecode. It enumerates possible execution sequences induced by implicit callback orders and checks whether each global resource is released on every path originating from its acquire point.

Resource Leak Checker [12] models leak detection as an accumulation typestate problem and improves precision via lightweight ownership transfer, specialized resource alias analysis, and fresh-obligation creation for modular checking. However, it requires user-supplied must-call specifications and manual code annotations, which makes it semi-automated

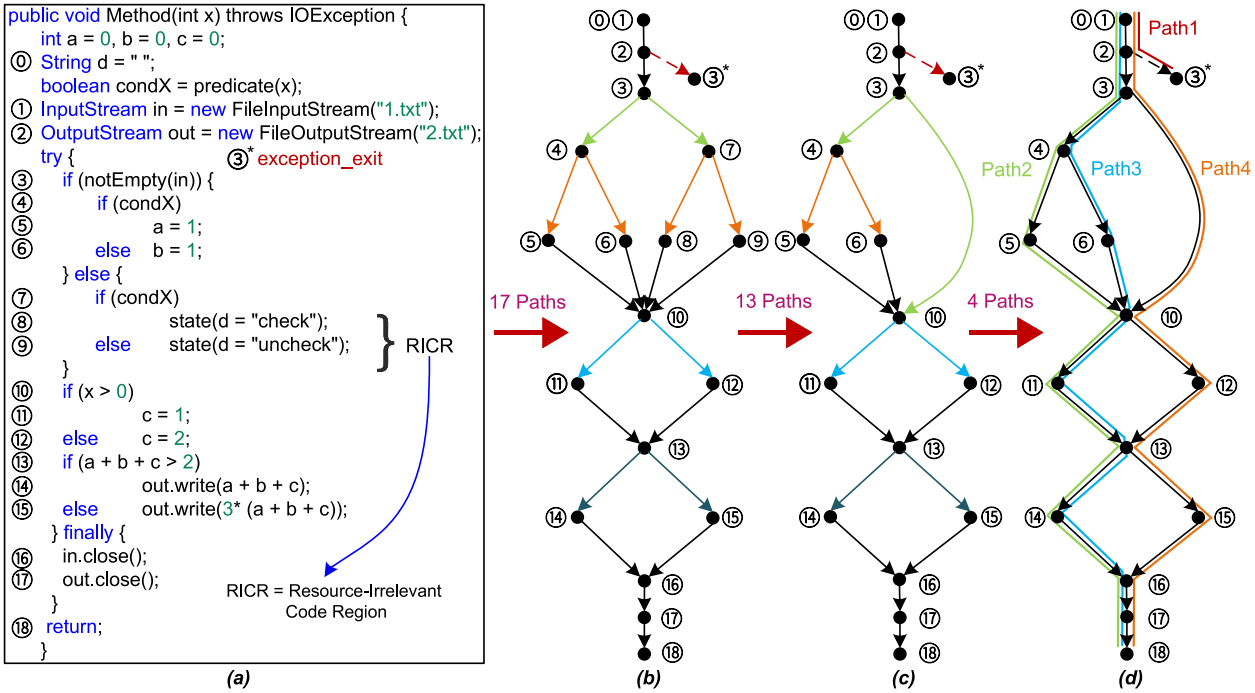


Figure 2: An example demonstrating how LeakHunter overcomes the challenges of path explosion. (a) the code snippet; (b) its original CFG; (c) its refined CFG; and (d) its LMPC. Node 0 denotes the assignment statements before node 1.

in practice. MIROK [9] mines resource-management API pairs (i.e., resource acquisition APIs and their corresponding release operations) from large-scale codebases to facilitate resource leak detection. However, predefined whitelists of such resource-management APIs still have limited coverage in real-world programs. Therefore, the state-of-the-art work, InferROI [10], leverages large language models (LLMs) [15, 16] to better recognize acquisition and release APIs. It then enumerates possible paths and applies two heuristic filters to prune resource-irrelevant normal paths and exception-induced paths, respectively, thereby reducing the number of paths to analyze. Nevertheless, these heuristics may miss some exception-induced paths. Moreover, even after retaining only resource-relevant paths, the remaining number can still be very large due to path enumeration, limiting the overall effectiveness.

Availability is a core security property, and resource-exhaustion DoS has been extensively studied in the security community [28, 29]. Rampart [28], for instance, defends web applications against CPU-exhaustion DoS by learning function-level execution models to block abusive requests. DR.D [29] uncovers another class of memory-exhaustion DoS vulnerabilities in Java web containers that stem from retaining user controlled data: objects originating from user requests may be inadvertently retained, accumulate in the heap, and eventually trigger remote memory exhaustion. In contrast, we focus on preventing resource exhaustion caused by leaks of GC-unmanaged resources in Java; while such leaks may be unintentionally introduced, they can still be exploited or repeatedly triggered to cause DoS-like availability degradation, making our approach complementary to prior DoS defenses.

3 Motivation & Overview

Resource-exhaustion DoS risks from GC-unmanaged resource leaks are often hidden in the "unhappy paths": a single leak-relevant exception can silently bypass cleanup, while the number of feasible executions makes exhaustive checking impractical. We use a concrete snippet to highlight these two obstacles that make resource-leak detection brittle in practice.

3.1 Motivating Example

In Figure 2(a), we present a code snippet in which the program ingests external input, applies conditional logic, and emits the result to an output stream. Notably, the snippet acquires two GC-unmanaged resources at nodes 1 and 2 by creating a `FileInputStream` for reading input data and a `FileOutputStream` for writing output data, and it intends to release both at nodes 16 and 17 by invoking `close` on each stream. This snippet also contains several method calls that may throw exceptions and redirect control flow, such as those at nodes 1-2, and 14-15. Notably, exceptions at node 2 must be considered because they can directly lead to a resource leak. If node 2 throws an exception after the resource is acquired, the execution may bypass the release at node 16,

leaving the resource unreleased. In contrast, exceptions at most other nodes are handled by the `finally` block, which still executes the release logic and prevents leaks.

This example illustrates that most exceptions are unrelated to resource leaks, while identifying the few that are leak-relevant requires contextual reasoning that is difficult to encode with heuristic rules while maintaining analysis efficiency. Therefore, existing approaches typically rely on simple filtering to approximate exception-induced leak analysis. InferROI [10] is a representative example. It applies a two-stage screening mechanism. First, it analyzes exception-induced paths only when the method declares a thrown exception in its signature. Second, it checks whether resources are released in a `finally` block, so that even if an exception causes non-local control transfers, the resources can still be released. If a case satisfies the first condition but violates the second, it assumes that some exception can bypass the resource release operations and therefore treats the case as leak-prone. However, heuristic rules inevitably have incomplete coverage. As shown in Figure 2(a), even when the release operations for `in` and `out` are placed in `finally`, an exception at node 2 may prevent the `close()` at node 16 from executing as intended, resulting in a leak—yet InferROI ignores this case.

In addition, traditional resource leak detectors often enumerate a large number of end-to-end witness paths and check them one by one for violations of the resource-usage policy, which can suffer from severe efficiency bottlenecks when analyzing large codebases. Recent approaches attempt to mitigate this cost by pruning resource-irrelevant code regions, retaining only regions that have control- or data-dependence on resource acquisition or release operations. As shown in Figure 2(b) and (c), even without considering the additional paths introduced by inter-procedural analysis, the refined program still contains 13 distinct paths. As branching continues, the number of paths grows exponentially, leading to path explosion, which highlights the limitation of this strategy.

3.2 Design Insights

Identifying leak-relevant exceptions is challenging because it requires reasoning about whether an exception can feasibly arise in the given program context and whether it can affect resource release. Such context-sensitive reasoning goes beyond existing heuristic rules. Meanwhile, making such analysis context-sensitive can introduce substantial analysis cost; scalable approximations often sacrifice precision, reflecting the precision–scalability trade-offs of context-sensitive analysis [30]. As a result, existing heuristic-based methods struggle to precisely identify leak-relevant exceptions. To address this challenge, we use large language models (LLMs) to infer potential leak-relevant exceptions. LLMs combine strong code understanding with context-aware reasoning, enabling them to infer plausible exception behaviors and distinguish those that are relevant to resource leaks. This design is motivated by prior evidence that LLMs learn rich statistical regularities from large code corpora and perform well on code understanding tasks [16, 31, 32], while also exhibiting useful reasoning behaviors in code-centric settings [33, 34]. We then statically validate and filter the LLM's results before using them in subsequent leak analysis.

Our insight for path generation in resource leak detection is that not all paths need to be evaluated. This insight stems from three considerations. First, leak detection is essentially existential: for a given resource-managing object, finding a single leaking execution path is sufficient to confirm a potential issue for that object. Second, under realistic time budgets, analyzing all paths is prohibitively expensive, so the analysis is better served by prioritizing a small subset of paths that are more likely to expose leaks for verification. Third, by graph theory, any directed acyclic graph (DAG) admits a minimum path cover (MPC) [18]. Moreover, resource-relevant code regions can be approximated as a set of DAGs, allowing us to cover it with a compact set of representative paths.

Guided by this insight, we first generate MPCs for each DAG in the program. Based on them, we construct witness paths by prioritizing those with the most acquisition operations and the fewest release operations, as such paths are the most likely to expose leaks. We then perform a resource-safety check on these paths to identify potential leaks. Overall, our algorithm prioritizes leak-prone representative paths, enabling efficient verification without enumerating all feasible paths. In Figure 2(d), our algorithm further reduces the number of paths to be verified from 13 to 4, building on the existing path refinement strategy. The selected set still includes leak-prone paths (e.g., *path1*), thereby significantly improving detection efficiency without compromising effectiveness.

3.3 LeakHunter Workflow

The architecture of LeakHunter is shown in Figure 3. It takes the Java source code as input. In the critical intention inference stage, LeakHunter constructs prompts and queries the LLM to infer three resource-relevant intentions in the program: resource acquisition APIs, resource release APIs, and leak-relevant exceptions. Next, LeakHunter builds an inter-procedural control-flow graph (iCFG) from the source code, prunes code regions that are independent of resource management based on control and data dependences, and eliminates unreachable paths via constant folding. Based on the refined iCFG, LeakHunter constructs a set of directed acyclic graphs (DAGs) and generates MPCs on each DAG. Building on the generated MPCs, it constructs end-to-end witness paths and prioritizes those most likely to expose leaks by favoring paths with more acquisition operations and fewer release operations; the resulting compact set of representative, leak-prone paths constitutes the LMPC. Finally, for each path in the LMPC, LeakHunter checks compliance with the resource usage policy, and any violating path is reported as a leak.

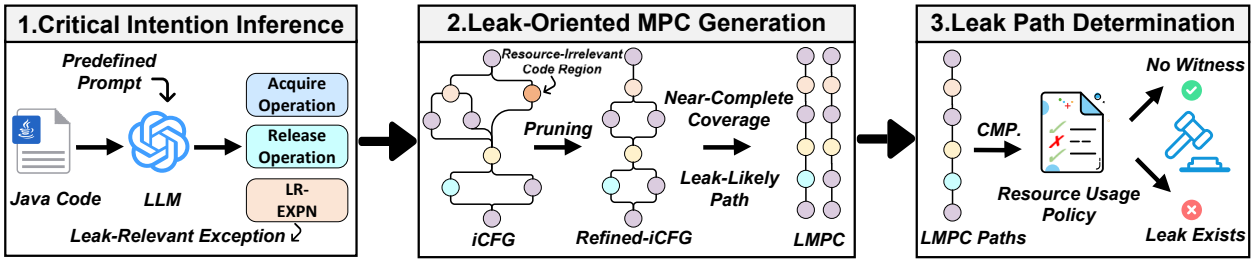


Figure 3: The architecture of LeakHunter.

Applying MPC to resource leak detection is non-trivial. It requires identifying leak-relevant exceptions that affect resource release, transforming cyclic iCFGs into DAGs suitable for MPC computation, and composing MPC segments from different DAGs into prioritized end-to-end witness paths. In the next section, we describe how to address these challenges.

4 System Design

LeakHunter concretizes the workflow outlined in Figure 3 by addressing the two bottlenecks: incomplete coverage of leak-relevant exceptional transfers and the prohibitive cost of path enumeration. It does so through two complementary mechanisms: leak-relevant exception inference (LLM-assisted and statically validated) to expose exception-induced leak-prone paths, and leak-oriented minimum path cover (LMPC) to generate a compact set of representative end-to-end witness paths for efficient leak checking.

4.1 Critical Intention Inference

Critical intention inference extracts the resource-relevant semantic facts needed by later stages: (1) resource-management operations, where a resource-managing object invokes resource-management APIs to acquire or release resources and (2) leak-relevant exceptions—exception-induced control-flow edges that can bypass the intended release and create leak-prone paths.

LLM assistance is needed because inferring both resource-management APIs and leak-relevant exceptions requires semantic generalization beyond fixed syntactic patterns. For resource-management operation inference, InferROI [10] has shown that detectors relying on predefined resource-management APIs can miss leaks when the relevant APIs are absent from the prepared sets, and that LLM-based inference can improve detection effectiveness. Moreover, inferring leak-relevant exceptions is challenging because their leak relevance depends on the surrounding semantic context, which is difficult to capture with heuristic rules alone. Without such reasoning, the analysis may either miss leak-relevant exceptions or introduce leak-irrelevant ones.

Therefore, we use an LLM to broaden coverage of resource-management APIs beyond predefined patterns; we further extend LLM assistance to infer leak-relevant exceptions whose plausible propagation can bypass the corresponding release in the given program context. However, while LLMs can reason well about relatively straightforward tasks such as resource-management API inference, their outputs may be unstable or even incorrect for tasks that require broader contextual and semantic reasoning, such as inferring leak-relevant exceptions. To mitigate this issue, rather than discarding rule-based checks altogether, we statically validate and filter the inferred exceptions before they affect leak reasoning (§4.2). Our ablation experiment further supports the practical benefit of this LLM-assisted inference pipeline, especially for identifying leak-relevant exceptions with static validation (§5.2).

We design prompt templates to query the LLM and parse its responses into structured representations for subsequent static analysis. Detailed prompts and example outputs are provided in Appendix A. The LLM outputs, for each resource-managing object, its Java type, initial identifier, and the source lines where acquisition or release occurs. It also identifies source lines where exceptions may arise and bypass the corresponding release operations.

4.2 Minimum Path Cover Generation

After locating critical intentions closely related to resource leak detection, the next step is to generate end-to-end witness paths for detecting resource leaks. We leverage the notion of a minimum path cover (MPC) to generate a representative set of paths to mitigate path explosion. In this section, we describe how to compute MPCs for a given program.

Generate Refined iCFG We first take the input Java program and construct an inter-procedural control-flow graph that links callee CFGs via call and return edges [35]. Since we focus on resource-relevant behaviors, we eliminate irrelevant code as an auxiliary pruning step to reduce its impact on detection and improve analysis efficiency. Similar to InferROI [10], we compute a dependence-based slice and prune away all other code, retaining resource acquisition and release operations as well as statements that are data- or control-dependent on these operations. Next, we apply lightweight constant folding and

reachability checking to simplify branch conditions; branches whose outcomes can be decided statically (e.g., conditions that fold to a constant) are collapsed, and obviously unreachable branch targets are removed. This yields a refined iCFG slice that is relatively smaller, while retaining the control- and data-flow behaviors that may affect resource leaks.

Validate Leak-Relevant Exception First, we check whether the locations of leak-relevant exception throwing points inferred by the LLM correspond to call sites. If so, we treat them as candidate leak-relevant exceptions. Next, we further validate and filter these candidate exceptions by the following strategy. Concretely, given a call site *cs* flagged by the LLM as potentially leak-relevant, we statically resolve the exception type t that may be thrown from *cs*. Then, we keep the candidate exception only if the program provides explicit evidence that t is anticipated, namely: (1) *cs* is dominated by a try region, and the exceptional edge from *cs* labeled with t can be dispatched to at least one catch handler whose caught type is compatible with t; (2) along inter-procedural exception propagation on the call graph, t can be caught by some caller of the enclosing method of *cs*; or (3) t is compatible with an exception type declared in the `throws` clause of the method enclosing *cs*. Otherwise, the exception will be pruned as noise. This design follows a common practice in static exception analysis: real-world Java programs typically make meaningful exceptional flows explicit by guarding them with compatible handlers or by declaring them in `throws`. Hence, an inferred exception with no such evidence is likely spurious. This strategy effectively reduces errors in the LLM's outputs when it is required to reason about complex semantics.

Transform iCFG into DAG Even after restricting enumeration to resource-relevant code regions, path explosion may still occur. Therefore, after obtaining the refined iCFG, we generate MPCs to avoid full path enumeration and mitigate path explosion. The challenge here is that we must eliminate cycles in the iCFG, since computing MPCs directly on a cyclic graph is NP-hard [36]. Thus, our algorithm first transforms the cyclic iCFG into one or more DAGs and then computes MPCs on these DAGs [37]. Similar to previous work [17], we distinguish two main kinds of cycles in the iCFG: loop cycles and caller-callee cycles. For loop cycles, recent leak detectors such as InferROI [10] adopt a simple treatment by unrolling each loop once to break the cycle. This is likely because a resource's lifetime usually spans the entire loop rather than individual iterations, and developers rarely place resource acquisition or release operations inside the loop body. Therefore, we follow the same strategy for handling loop cycles. Subsequently, we turn to the analysis of caller-callee cycles.

A caller-callee cycle arises when the same callee is invoked at multiple call sites. In the iCFG, control flow can return from the callee to the caller, reach another call site, and re-enter the same callee entry, thereby forming a cycle. For example, in Figure 4(a), the two call sites of method M induce such a

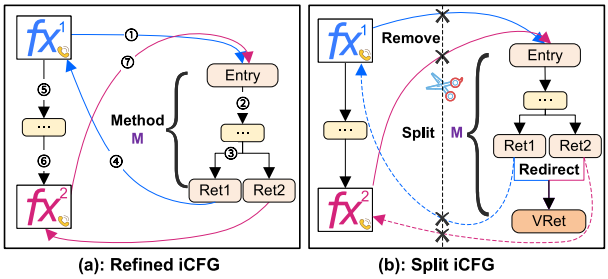


Figure 4: Caller-callee cycle (fx^1 and fx^2 denote two call sites that invoke the same method)

cycle; one traversal along this cycle is $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7$, which re-enters M's entry twice.

To eliminate caller-callee cycles, we split each callee method that may form such a cycle (denoted as a cl-method) from the refined iCFG and transform it into a one-entry-one-exit method subgraph as follows. First, we remove the edges from each cl-method call site in the refined iCFG to its callee method body to break cycles, while retaining the call-site vertex in the remaining iCFG (denoted as rCFG). The separated CFG of the cl-method is denoted as a cCFG. Next, to avoid different cCFG paths returning to the rCFG through different return nodes, we insert a virtual return node (*VRet*) after all return statements in the cCFG and redirect all the original return edges from these return statements to VRet, rather than back to the rCFG. VRet then serves as a unified return point. As a result, the return edges that originally led back to the rCFG are replaced by the redirected edges to VRet, as illustrated in Figure 4(b). This ensures that each extracted cCFG has a single entry (the callee entry block) and a single exit (VRet). After both loop cycles and caller-callee cycles are handled, the rCFG and the cCFGs can be treated as DAGs, and we can compute MPCs on each of them independently.

Compute MPC from DAG We now describe how we compute MPCs on the DAGs (i.e., the rCFG and the cCFGs). Given a DAG $G = (V, E)$, where V and E denote the vertex and edge sets, respectively, a path cover P is a finite set of paths $P = \{p_1, p_2, \ldots, p_k\}$ such that every $v \in V$ is visited by at least one path in P. A minimum path cover P_{mpc} is a path cover with minimum cardinality among all path covers on G. Computing MPCs on DAGs in polynomial time is well studied [18, 36, 38, 39], and we adopt Ntafos' maximum-matching algorithm [36] to obtain them. Algorithm 2 in Appendix B gives the detailed procedure for computing an MPC from a DAG. We apply it separately to the rCFG and each cCFG to obtain MPCs, and for each DAG we attempt to generate up to K distinct MPCs to enhance path diversity.

4.3 Leak-Oriented MPC Construction

MPCs computed on individual DAGs are only local path segments, whereas resource-leak detection requires prioritized

end-to-end witness paths, namely complete entry-to-exit paths in the refined iCFG. To address this challenge, LeakHunter goes beyond computing MPCs on each DAG: it turns MPC segments into a leak-oriented MPC (LMPC) that produces prioritized leak-prone witness paths. Instead of directly enumerating exponential iCFG paths, we treat the retained rCFG/cCFG MPC segments as a compact composition space and lazily splice them via best-first search under a leak-oriented priority, favoring acquisition-heavy and release-light behavior. We output only the Top-M witnesses (configurable M) to keep the construction efficient on complex programs.

The path construction starts with each unique rCFG path segment p_r covered by the generated rCFG MPCs. We then expand call-sites that invoke cl-methods by splicing callee path segments drawn from the generated MPCs of the invoked cl-methods. A global best-first procedure then outputs witness paths p_w under a leak-oriented priority order. We prioritize paths that are more likely to expose leaks by favoring acquisition-heavy and release-light behavior. In general, the larger the difference between the numbers of acquisition and release operations on a path, the more likely it is to expose a resource leak, because it suggests that resources may be acquired without being promptly released. We assign each path segment p in the generated MPCs an additive local leak score as a proxy for its leak-proneness, where p is an intra-procedural path from entry to exit in either the rCFG (p_r) or a cCFG (p_c). We define this local leak score as follows:

$$\textit{local leak score}(p) = ACQ(p) - REL(p) \tag{1}$$

where $ACQ(p)$ and $REL(p)$ denote the numbers of resource acquisition and release operations along p, respectively. This equation is computed for both the starting rCFG segment p_r and every spliced callee segment p_c, and the scores are accumulated during splicing. Then, for a witness path p_w, its leak score is the sum of the local leak scores of all constituent p_r segments and spliced p_c segments:

$$\textit{global leak score}(p_w) = \sum_{p \in p_w} \textit{local leak score}(p) \tag{2}$$

To avoid forming the full cartesian product of callee path segment choices across cl-method call-sites encountered during splicing starting from p_r, we perform best-first exploration under a *leak-oriented priority*—a best-case completion score that steers the search toward larger *global leak score*. For each cl-method m, we collect the path segments in the MPCs generated for its cCFG, remove duplicates, sort them by *local leak score* in descending order, and store the resulting sequence as a callee bank: $Bank(m) = \langle p_{m,1}, \ldots, p_{m,n} \rangle$. Given a (partial) witness path[1], its leak-oriented priority has two additive parts: (1) the sum of *local leak score* over all already-spliced path segments, and (2) for each remaining unexpanded call-site that invokes a cl-method m, the maximum local leak score among the path segments in $Bank(m)$. We use this priority as the queue key to rank partial witness-paths for best-first exploration. The priority is recomputed after each splice and becomes exact once the partial witness-path is fully expanded.

To leverage the leak-oriented priority to select the witness paths most likely to expose leaks, we create a generator backed by a local queue keyed by the leak-oriented priority. The generator is initialized with the unexpanded p_r as the initial partial witness-path. Repeatedly, it pops the element with the largest leak-oriented priority. If the popped element contains an unexpanded call site that invokes a cl-method, the generator expands it by splicing in the corresponding callee path segment from $Bank(m)$, producing new partial witness paths. Each newly formed partial witness-path is re-keyed with an updated leak-oriented priority and pushed back to the local queue. Because the leak-oriented priority is a best-case completion score for partial witness-paths and becomes exact (equal to the global leak score) once a witness path is fully expanded, each generator's best-first exploration prioritizes partial witness-paths that can still lead to larger global leak scores within the MPC-induced path space.

Finally, we coordinate all generators with a global max-priority queue to produce the Top-M witness paths[2] as our LMPC. Intuitively, the global queue only mirrors each local queue's head; therefore, when we choose one head from the global queue, we then pop the same element from its local queue to advance that generator. At each step, we pop the globally best partial witness-path. If it is fully expanded, we output it as the next witness path. Otherwise, we remove this partial witness path from the generator's local queue and expand its next unexpanded call site to a cl-method. We then push the newly formed partial witness path back into the same local queue with an updated leak-oriented priority. The generator's new head element is then re-inserted into the global queue. This interleaving ensures the budget M is spent on the most leak-prone witness paths across all p_r starts, rather than being consumed by any single start. The process continues until either the number of constructed paths reaches M or witness paths have been fully constructed. The detailed algorithm is provided in Algorithm 1.

Whole-program end-to-end path enumeration, even with a hard cap M, still has two drawbacks. First, it must commit to branch decisions throughout the iCFG, so a limited budget M can be consumed by many highly overlapping, low-yield paths that add little new coverage of resource-relevant code. Second, the global leak score of an end-to-end path is only known after the path is fully materialized, which makes it hard to steer enumeration toward the Top-M leak-prone paths without exploring far more than M path candidates. In contrast, our approach generates witnesses from a compact MPC-induced

[1] A partial witness-path is a prefix of a witness path that starts from p_r, has already spliced in some cl-methods, and still contains unexpanded call sites to other cl-methods.

[2] M is used to cap extreme cases, and it can also be set to an unbounded value when users want to explore a more complete set of witness paths.

Algorithm 1 LMPC Generation Algorithm

Require: M_r and $M_{c,m}$ // MPCs of rCFG and n cCFGs
1: $LMPC \leftarrow [\,]$, $Bank \leftarrow \{\}$
2: **for all** cl-method m **do**
3: $PS \leftarrow \{\}$ // paths and their local leak score
4: **for all** $p_c \in M_{c,m}$ **do**
5: $s_c \leftarrow$ GEN_LOCAL_LEAK_SCORE(p_c)
6: $PS \leftarrow PS \cup \{(p_c : s_c)\}$
7: **end for**
8: $Bank[m] \leftarrow$ RANK_UNIQUE_PATHS(PS)
9: **end for**
10: $Starts \leftarrow$ EXPLORE_RCFG_PATHS(M_r)
11: $GQ \leftarrow [\,]$ // global leak-oriented priority queue
12: **for all** $p_r \in Starts$ **do**
13: $s_r \leftarrow$ GEN_LOCAL_LEAK_SCORE(p_r)
14: $cand \leftarrow$ INIT_PATH_CANDIDATE(p_r, s_r)
15: $Q[p_r] \leftarrow [\,]$ // local leak-oriented priority queue
16: $Q[p_r]$.push($cand$, CALC_LEAK_PRIO$(cand, Bank)$)
17: GQ.push(TOP$(Q[p_r]), p_r)$
18: **end for**
19: **while** $|LMPC| < M$ **and** $GQ \neq \emptyset$ **do**
20: $(cand, id) \leftarrow$ POP_MAX(GQ)
21: POP_MAX$(Q[id])$ // advance the local queue of id
22: **if** EXPAND_COMPLETE$(cand)$ **then**
23: $p_w \leftarrow cand$ // no unexpanded cl-methods
24: $LMPC.append(p_w)$
25: **else**
26: $cs_m \leftarrow$ NEXT_CALLSITE$(cand)$ // call site of m
27: **for all** $(p_c, s_c) \in Bank[m]$ **do**
28: $cand' \leftarrow$ SPLICE$(cand, cs_m, p_c)$
29: $Q[id].push(cand', $CALC_LEAK_PRIO$(cand', Bank))$
30: **end for**
31: **end if**
32: **if** $Q[id] \neq \emptyset$ **then** $GQ.push($TOP$(Q[id]), id)$
33: **end while**

path space and performs best-first splicing under the leak-oriented priority. As a result, each produced witness path is coverage-informed by the MPC structure and prioritized by leak-oriented scoring, rather than being an arbitrary sample from the exponential iCFG path space.

4.4 Leak Path Determination

After constructing the LMPC set, we validate its end-to-end witness paths one by one in descending leak-oriented priority order. Following prior work [7, 8, 10], we treat a witness path as leak-prone if it leaves at least one acquired resource-managing object without a matching release. Concretely, based on the results of critical intention inference, we annotate each resource-management API call on a path with its associated resource-managing object o, identified by its type, initial identifier, and line number. For a witness path p_w, let $\mathrm{ACQ}(p_w, o)$ and $\mathrm{REL}(p_w, o)$ denote the numbers of resource acquisitions and resource releases associated with object o along p_w. We deem p_w potentially leak-prone if there exists an object o whose acquisitions outnumber its releases, i.e., **Condition 1** holds:

$$\exists o: \ \mathrm{ACQ}(p_w, o) > \mathrm{REL}(p_w, o).$$

Otherwise, when **Condition 2** holds, i.e.,

$$\forall o: \ \mathrm{ACQ}(p_w, o) \leq \mathrm{REL}(p_w, o),$$

we consider p_w leak-free.

Since leak detection is existential, once we find a witness path p_w satisfying **Condition 1** for an object o, we report that o is leak-prone and skip further witnesses for the same object to reduce redundant validation. If all paths in the LMPC are verified to be leak-free, we report that no leak witness is observed under the current LMPC.

$$leak = \begin{cases} Exist, & \exists\, p_w \in LMPC,\ \textbf{Cond}\ 1\ holds \\ No\ Witness, & \forall\, p_w \in LMPC,\ \textbf{Cond}\ 2\ holds \end{cases} \quad (3)$$

5 Evaluation

Leveraging two publicly available datasets, we conduct a comprehensive evaluation of LeakHunter against baseline tools, comparing both effectiveness and efficiency in §5.2. To highlight the advantages of our LMPC algorithm in reducing detection overhead, we further perform a fine-grained analysis in §5.3 to explain why LeakHunter achieves high efficiency. In addition, we conduct real-world experiments in §5.4 to assess LeakHunter's capability in practical scenarios. Finally, we characterize the security impact of the resource leaks we validated and analyze a representative newly discovered leak case to show how a resource leak can potentially lead to resource-exhaustion DoS attacks.

5.1 Experiment Settings

Implementation and Baselines LeakHunter is implemented in more than 6,400 lines of Python code, and utilizes the Src-toolkit [40] to parse source code. Building on this foundation, LeakHunter integrates an LLM-assisted component to infer resource-management APIs and leak-relevant exceptions. We evaluated multiple LLMs and found that GPT-5 [41] achieves the strongest overall performance; therefore, the experimental results in this section primarily use GPT-5. We also report results with open-source LLMs, including Llama4-Scout [42] and Gemma3-12B [43], in Table 2 and Table 3.

To evaluate LeakHunter's performance, we compare it against three representative baselines that span the academic state-of-the-art and widely used industrial tools: InferROI [10], a recent approach that represents one of the

strongest existing methods for resource leak detection; and PMD [21] and SonarQube [22], two industrial static analyzers that are widely used in real-world development for resource leak detection. These tools were selected because they are publicly available, operate on Java source code, and can automatically detect resource leaks. Consequently, we exclude leak detection tools that are not publicly available (e.g., Tracker [7] and Relda2 [8]), as well as approaches that require per-program manual annotation, such as Resource Leak Checker [12], from our baselines.

Datasets Collection Our controlled experiments require datasets with established ground truth to enable a rigorous evaluation of Java resource leak detection. Accordingly, we select DroidLeaks [20] and JLeaks [5] to form our *ground-truth dataset*. We choose these benchmarks because, to the best of our knowledge, they are the only publicly available datasets specifically curated for Java resource leak detection. Similar to InferROI [10], we refined the dataset primarily by excluding cases with syntax errors, as they cannot be reliably parsed into ASTs and thus prevent downstream static analysis by the evaluated tools. Overall, our benchmark suite contains 1,820 valid test cases, including 156 from DroidLeaks and 1,664 from JLeaks, which we use for subsequent evaluation.

To further assess the practical effectiveness of LeakHunter, we curate a real-world dataset of 100 popular open-source Android applications (apps), with a focus on those that contain externally accessible interfaces, restricting our selection to GitHub projects with more than $1,000$ stars to bias the dataset toward representative, widely used codebases. Android apps are suitable for this study because, when they expose externally accessible interfaces in a client-server setting, they represent meaningful targets for resource-exhaustion DoS analysis, while facilitating controlled and reproducible evaluation.

Evaluation Metrics Following standard practice in the evaluation of resource leak detection, the results of the detection can be categorized into four basic cases. A *true positive* (TP) denotes that the program indeed contains a resource leak and the tool correctly identifies it. A *false negative* (FN) indicates that a resource leak exists but is overlooked by the tool. A *false positive* (FP) refers to cases where the tool incorrectly reports a resource leak that does not actually exist. A *true negative* (TN) means that the program contains no resource leak and the tool correctly recognizes this fact. Based on these definitions, we measure Precision (Pre.), Recall (Rec.) and F1-score (F1) [44] to evaluate the performance of LeakHunter and the baseline tools.

Threshold Tuning LeakHunter has two tunable parameters, K and M, which control the number of MPCs generated per DAG and the total number of witness paths, respectively. For K, we empirically tuned it and set K=3 as the default value, as detailed in Appendix C. For M, we set it to be unbounded to faithfully reflect LMPC's performance, i.e., without imposing a cap on the number of generated paths.

5.2 Effectiveness and Efficiency

Comparison Testing Results As shown in Table 1, the industrial resource leak detection tools, including PMD and SonarQube, exhibit low precision and recall in our experiments. In general, PMD achieves only 57.2% precision and 10.4% recall, resulting in a 17.7% F1-score. SonarQube performs slightly better, with 52.0% precision and 13.1% recall (20.9% F1-score). This poor performance mainly stems from the limitations of heuristic rule-based analysis. Such rules cover only a subset of resource leak patterns and often fail to account for leak-relevant exceptions as well as their impact on leak-prone control flow, leading to many missed resource leaks. The cutting-edge approach, InferROI, performs substantially better, achieving F1-scores of 64.1% on DroidLeaks and 72.6% on JLeaks, resulting in an overall F1-score of 72.0%. This result is comparable to the performance reported in the InferROI paper [10] (66.7% F1-score on DroidLeaks and 68.7% on JLeaks). InferROI leverages an LLM to identify resource-management operations more broadly, enabling it to validate a wider range of resource-relevant paths and thus reduce false negatives. However, its limited handling of leak-relevant exceptions constrains further improvements.

In contrast, LeakHunter achieves the best overall performance among all compared tools, obtaining F1-scores of 86.8% and 81.6% on DroidLeaks and JLeaks, respectively, and improving the overall F1-score to 82.0%, which is 10.0% (82.0% - 72.0%) higher than InferROI. This gain mainly stems from LeakHunter's incorporation of leak-relevant exceptions inferred by the LLM and subsequently validated by static analysis. By doing so, LeakHunter can more accurately cover execution paths beyond regular control flow, thereby modeling more complete leak-prone paths and uncovering resource leaks missed by InferROI. Moreover, LeakHunter's outstanding performance also highlights the effectiveness of our LMPC algorithm: instead of exhaustively enumerating paths as traditional approaches do, LMPC retains only a small set of representative, highly leak-prone paths, yet does not noticeably sacrifice leak detection capability.

LeakHunter with Different LLMs We also conduct experiments with LeakHunter and InferROI using two representative open-source LLMs. The results on DroidLeaks and JLeaks are reported in Tables 2 and 3. Overall, LeakHunter and InferROI achieve the best performance when using GPT-5. On DroidLeaks, LeakHunter with GPT-5 improves the F1-score over LeakHunter with Llama4-Scout and Gemma3-12B by 7.6% (86.8%-79.2%) and 5.7% (86.8%-81.1%), respectively. On JLeaks, the improvements are 1.8% (81.6%-79.8%) and 1.4% (81.6%-80.2%). The main reason is that GPT-5 is more effective at locating resource-management operations and inferring leak-relevant exceptions, which in turn reduces false positives and false negatives caused by errors in the critical intention inference stage. In addition, regardless of which

Table 1: The comparison results among peer tools (LeakHunter^{I} = LeakHunter without leak-relevant exception analysis, LeakHunter^{II} = LeakHunter with traditional path enumeration algorithm, LeakHunter and InferROI both use GPT-5).

Tool	DroidLeaks							JLeaks							All		
	TP	TN	FP	FN	Pre.	Rec.	F1	TP	TN	FP	FN	Pre.	Rec.	F1	Pre.	Rec.	F1
LeakHunter	66	70	8	12	**89.2%**	**84.6%**	**86.8%**	710	633	199	122	**78.1%**	**85.3%**	**81.6%**	**78.9%**	**85.3%**	**82.0%**
LeakHunter^{I}	57	71	7	21	89.1%	73.1%	80.3%	436	705	127	396	77.4%	52.4%	62.5%	78.6%	54.2%	64.2%
LeakHunter^{II}	66	67	11	12	85.7%	**84.6%**	85.2%	710	596	236	122	75.1%	**85.3%**	79.9%	75.9%	**85.3%**	80.3%
InferROI	41	69	9	37	82.0%	52.6%	64.1%	581	645	187	251	75.7%	69.8%	72.6%	76.0%	68.4%	72.0%
PMD	23	59	19	55	54.8%	29.5%	38.3%	72	780	52	760	58.1%	8.7%	15.1%	57.2%	10.4%	17.7%
SonarQube	6	76	2	72	75.0%	7.7%	14.0%	113	724	108	719	51.1%	13.6%	21.5%	52.0%	13.1%	20.9%

Table 2: Results of different LLMs on DroidLeaks dataset

Tool	LeakHunter			InferROI		
	Pre.	Rec.	F1	Pre.	Rec.	F1
Llama4-Scout	86.4%	73.1%	79.2%	76.2%	41.0%	53.3%
Gemma3-12B	85.7%	76.9%	81.1%	76.6%	46.2%	57.6%
GPT-5	**89.2%**	**84.6%**	**86.8%**	**82.0%**	**52.6%**	**64.1%**

Table 3: Results of different LLMs on JLeaks dataset

Tool	LeakHunter			InferROI		
	Pre.	Rec.	F1	Pre.	Rec.	F1
Llama4-Scout	77.2%	82.7%	79.8%	74.6%	67.4%	70.8%
Gemma3-12B	77.0%	83.7%	80.2%	74.4%	68.1%	71.1%
GPT-5	**78.1%**	**85.3%**	**81.6%**	**75.7%**	**69.8%**	**72.6%**

LLM is used, LeakHunter consistently outperforms InferROI under the same LLM setting.

Stepwise Contribution Analysis LeakHunter introduces two new mechanisms: (1) LLM-assisted inference of leak-relevant exceptions, validated by static analysis, to expose leak-prone paths along exceptional control flow, and (2) an LMPC path exploration strategy to improve path analysis efficiency. To better demonstrate the benefit of each contribution, we conduct an ablation study. We evaluate LeakHunter without leak-relevant exception analysis and denote this variant as LeakHunter^{I} in Table 1. We also report results for LeakHunter^{II}, which replaces LMPC with a traditional path enumeration strategy. Comparing LeakHunter^{I} with LeakHunter quantifies the performance gain from our leak-relevant exception analysis, while the gap between LeakHunter and LeakHunter^{II} reflects the performance change from using the LMPC algorithm instead of the traditional path enumeration method.

From the results in Table 1, LeakHunter^{I} achieves an overall F1-score of 64.2%, whereas LeakHunter reaches 82.0%. This 17.8% improvement (82.0% - 64.2%) indicates that, by combining the LLM's semantic understanding with static validation, our method can effectively locate the associated leak-prone paths, thereby reducing false negatives and improving detection performance. In addition, we find that LeakHunter^{II} and LeakHunter achieve the same recall (85.3%), while LeakHunter attains a higher precision (78.9% vs. 75.9%). As a result, LeakHunter improves the F1-score by 1.7% (82.0% vs. 80.3%). The identical recall indicates that, on our dataset, LMPC does not miss any additional true positives relative to full path enumeration. The higher precision is because LMPC generates a representative set of high-likelihood leak-prone paths, rather than enumerating all paths as traditional approaches do. Enumerating more paths increases the risk of false positives during leak path determination, thereby reducing precision.

Detection Error Analysis Although LeakHunter achieves the best overall detection performance, with an F1-score of 82.0%, it still produces both false negatives and false positives. Further analysis shows that Android's callback-driven execution is the main source of false negatives, with more than 60% of them caused by callback-related issues. In these cases, resource acquisitions and releases may occur in different lifecycle callbacks [45] or asynchronous handlers, and their effective ordering is governed by framework events. Without a callback-aware execution model, static analysis cannot reliably match releases with their corresponding acquisitions across callbacks. For example, the analysis may incorrectly match an acquisition in one callback with a release in another that is not feasible under the framework's ordering rules. As a result, a truly unreleased resource may be considered released, leading to a missed detection. While incorporating precise callback scheduling could improve accuracy, it would substantially increase analysis complexity and runtime. Thus, similar to InferROI, LeakHunter does not model full callback scheduling, and we leave this to future work. In addition, many false positives stem from the LLM missing some resource-management operations. For instance, if the LLM fails to identify a resource release, LeakHunter may incorrectly conclude that the resource remains unreleased along the relevant paths, resulting in a false alarm.

Efficiency Existing static tools typically detect leaks by enumerating execution paths, which can incur substantial overhead. We therefore measure the static path analysis efficiency, including the time for path generation and for resource leak determination on the generated paths, and compare the efficiency of LeakHunter against the latest research approach InferROI,

Table 4: The average path generation time (before "/") and the leak path determination time (after "/") of peer tools.

Tool	DroidLeaks	JLeaks	All
LeakHunter	0.2s / 0.1s	1.4s / 1.1s	1.3s / 1.0s
LeakHunter$^{\text{I}}$	0.2s / 0.1s	1.2s / 0.9s	1.1s / 0.8s
LeakHunter$^{\text{II}}$	23.5s / 13.8s	16.3s / 9.6s	16.9s / 10.0s
InferROI	23.7s / 13.6s	15.7s / 9.4s	16.4s / 9.8s

which represents the traditional path enumeration strategy. We also report the results of LeakHunter$^{\text{I}}$ and LeakHunter$^{\text{II}}$ for further stepwise analysis. Table 4 summarizes the overhead statistics for path generation and leak path determination on our ground-truth dataset for each tool. For the per-case LLM analysis time of each tool, see Appendix D.

As shown in Table 4, on the ground-truth dataset, InferROI spends 16.4s on path generation and 9.8s on leak determination over the generated paths, for a total path analysis time of 26.2s (16.4s + 9.8s). In contrast, LeakHunter takes only 1.3s on average to generate paths. Because it generates far fewer paths, the leak determination time also drops to 1.0s, resulting in a total analysis time of 2.3s (1.3s + 1.0s). Overall, LeakHunter achieves an 11.4× speedup over InferROI. This efficiency gain primarily stems from our LMPC algorithm, which selects a compact set of representative, leak-prone paths and thus substantially reduces the cost of path exploration, indirectly lowering the overhead of determining leaks from these paths as well. For LeakHunter$^{\text{II}}$, it takes 16.9s to generate all paths and 10.0s to determine leaks on these paths, resulting in a total path analysis time of 26.9s (16.9s + 10.0s). Comparing LeakHunter with LeakHunter$^{\text{II}}$, the 24.6s gap (26.9s - 2.3s) highlights the efficiency advantage of LMPC over the traditional algorithms. Note that the ground-truth dataset typically contains relatively simple cases and thus may not fully reflect the efficiency gap between LMPC and traditional path enumeration. Results on real-world programs (§5.4) further demonstrate the advantages of LMPC.

5.3 LMPC's Contribution

Traditional path analysis attempts to explore possible execution paths; even with heuristic pruning, it may still suffer from path explosion on large codebases. By contrast, our previous experiments show that LeakHunter significantly improves efficiency compared with traditional approaches, while also achieving better ground-truth performance. In this part, we provide a deeper analysis by intuitively comparing the number of paths generated by LMPC with that produced by traditional path analysis. Therefore, we report the path ratio distribution of LeakHunter and InferROI on the ground-truth dataset. Specifically, for each example we compute the path ratio, defined as the number of generated paths divided by the total number of enumerable paths. We plot the empirical

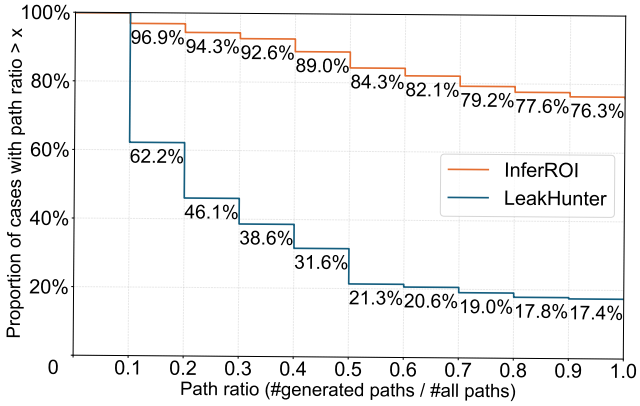


Figure 5: Path ratio distribution of LeakHunter and InferROI

complementary cumulative distribution function (CCDF) [46] of the path ratio in Figure 5. Specifically, for each path ratio value $p \in \{0, 0.1, \ldots, 1.0\}$ on the x-axis, we report the proportion of cases whose path ratio exceeds p on the y-axis.

Figure 5 clearly shows that the path ratio produced by InferROI is heavily concentrated in the 90-100% range, accounting for 76.3% of the cases. If we further count cases where this ratio exceeds 60%, the proportion rises to 82.1%. This indicates that, even with heuristics that prune paths completely irrelevant to resource-management operations, the pruning effect may remain limited, and the remaining path space can still be large for programs with complex control flow. In contrast, LeakHunter's CCDF drops quickly: only 46.1% of cases have a path ratio > 0.2, meaning 53.9% of cases are covered with $\leq$ 20% of enumerable paths. In addition, LeakHunter has only 17.4% of cases whose path ratio falls in the [0.9, 1.0] range, far below InferROI's 76.3%. These cases mainly correspond to simple programs with very few enumerable paths, where LMPC generates almost all of them. Overall, the results further demonstrate the advantage of our LMPC algorithm in generating a compact set of representative paths, while LeakHunter's strong performance in Table 1 indicates that these paths remain highly effective at exposing leaks.

5.4 Real-world Test of LeakHunter

To validate LeakHunter's practical applicability and its ability to uncover previously unknown resource leaks in real-world development scenarios, we extend our evaluation to the curated real-world dataset of 100 popular open-source Android apps on GitHub, with a focus on those that contain externally accessible interfaces. We compare LeakHunter against the state-of-the-art baseline, InferROI, and also report the results of LeakHunter$^{\text{I}}$ and LeakHunter$^{\text{II}}$ to support a stepwise analysis of our design choices. For each tool, we record the detected GC-unmanaged resource leaks, and then assess which validated leaks could potentially trigger resource-exhaustion DoS attacks. To reduce author bias in validating real-world results,

Table 5: Real-world performance of peer tools. (Total/Correct = number of reported potential leaks/number of validated true leaks; PAT Per App = average path analysis time per app.)

Tool	Total	Correct	PAT Per App
LeakHunter	111	90	89.4s
LeakHunter^{I}	104	84	86.7s
LeakHunter^{II}	111	90	1050.5s
InferROI	100	80	1038.3s

we adopt a strict anonymized manual validation process (see Appendix E) and report all validated leaks to the affected vendors to mitigate ethical concerns. Next, we analyze the security impact of the manually validated leaks. Finally, we present a representative case study to analyze how a GC-unmanaged resource leak can lead to resource-exhaustion DoS vulnerability.

Real-World Comparison Table 5 summarizes the detection results of LeakHunter, its variants, and InferROI on 100 Android apps. InferROI reports 100 potential resource leaks in total, of which 80 are validated as true leaks through manual validation. LeakHunter reports 111 potential resource leaks, and our manual validation confirms 90 true leaks. To date, developers have formally confirmed **15** leak defects reported by LeakHunter, compared to 9 for InferROI; all 9 are included in LeakHunter's developer-confirmed set. Appendix F further reports the detailed app categories associated with these leaks. Appendix G summarizes the developers' feedback.

Overall, LeakHunter outperforms InferROI in terms of validated true leaks. In addition, the increase of six additional true leaks (90 vs. 84) from LeakHunter^{I} to LeakHunter highlights the contribution of analyzing leak-relevant exceptions. Comparing LeakHunter^{II} with LeakHunter further indicates that, on these 100 apps, our LMPC algorithm does not reduce detection effectiveness, even though it analyzes only a representative set of paths rather than performing path enumeration. Moreover, Table 5 shows that InferROI spends an average of 1038.3s on path analysis per app, whereas LeakHunter requires only 89.4s. This corresponds to an 11.6× speedup. Comparing LeakHunter^{I} with LeakHunter shows that analyzing the leak-relevant exception paths incurs only 2.7s (89.4s - 86.7s) extra overhead. Contrasting LeakHunter^{II} with LeakHunter indicates that the LMPC algorithm reduces the path analysis time by 961.1s (1050.5s - 89.4s). The large analysis time gaps of LeakHunter^{II} and InferROI further highlight the scalability limitations of path enumeration when analyzing real-world apps and underscore the efficiency advantage of our LMPC algorithm.

Security Impact To better characterize the security impact of the 90 validated leaks, Table 6 summarizes whether they are externally exploitable, their triggering channels, and the typical consequences of repeatedly triggering these leaks. Among

```
public void M (String file) {
    String line;
    StringBuilder license= new StringBuilder();
    try {
        InputStream is = getContext().
                                getAssets().open(file);
        BufferedReader r = new BufferedReader(new
                        InputStreamReader(is, "UTF-8"));
       while ((line = r.readLine()) != null) {
          license.append(line).append("\n");
       }
     } catch (IOException e) {
       e.printStackTrace();
     }
    ... ← no close
}
```

Figure 6: A real-world code snippet illustrating a representative GC-unmanaged resource leak.

them, 67 leaks are externally exploitable under the DoS threat model in §2.1: they are externally triggerable through several types of externally accessible interfaces, which serve as direct triggering channels for an adversary-controlled client. This subset includes all 15 developer-confirmed cases.

Among the 67 leaks, externally accessible interfaces for data access form the dominant triggering channel, accounting for 68.7% (46/67). Representative concrete interfaces in this category include file-content query APIs that retrieve requested files. The remaining leaks are triggered through other interfaces such as data upload, connection management, and task dispatch, showing that the attack surface is not limited to one specific pattern. For these externally exploitable cases, an adversary-controlled client can repeatedly trigger the leaks through the corresponding externally accessible interfaces, potentially exhausting resources, denying the affected server-side functionality, and eventually crashing the server app. This is consistent with the availability consequence described by CWE-772 [3]. For the other 23 validated leaks, we did not find direct external triggering channels under our DoS threat model. However, they are still resource-management weaknesses consistent with CWE-404 [2]. When app-internal workflows repeatedly exercise the corresponding leak-prone functionality, these leaks can still lead to similar resource-exhaustion availability impacts and crashes in the affected app, whether client-side or server-side.

DoS Case Analysis To demonstrate the potential DoS impact of GC-unmanaged resource leaks, we select a representative example in which a leak can be repeatedly triggered to expose a resource-exhaustion DoS vulnerability. The victim app server exposes an externally accessible interface for data access, which allows clients to query license text. For each request, the server calls `getContext().getAssets().open()` to obtain an `InputStream` backed by a file descriptor, wraps it with a

Table 6: Security characterization of the validated real-world leaks (EAIs denote externally accessible interfaces; the "Externally Exploitable" column reports exploitability under the threat model in §2.1).

Externally Exploitable	Triggering Channel	#	Typical Consequences of Repeated Triggering
Yes	Data-access EAIs	46	Denial of data query; server-app crash
	Data-upload EAIs	2	Denial of upload functionality; server-app crash
	Data-write EAIs	6	Denial of write functionality; server-app crash
	Data-processing EAIs	6	Denial of processing pipeline; server-app crash
	Connection-management EAIs	5	Denial of connection establishment; server-app crash
	Task-dispatch EAIs	2	Denial of thread dispatch; server-app crash
No external trigger evidence	App-internal workflows	23	Denial of affected functionality; app crash

`BufferedReader` to read the license text into a string, and returns it. However, since `close()` is never invoked, the acquired resource is not released in the code, so the descriptor remains open and accumulates across repeated requests. Figure 6 shows the key code pattern that leads to this leak. As a result, this interface is leak-prone: an attacker can repeatedly invoke it, and each invocation leaves the underlying asset `InputStream` unclosed, steadily increasing the server process's open-descriptor count.

We use an Android device (Xiaomi Mi 6) with a maximum file descriptor limit of 32,768 to emulate the victim server app, while a client repeatedly issues requests to simulate an attacker. Specifically, the client issues requests at 100 requests per second, while we monitor the server process's open file descriptor count (Figure 7). As the leak is repeatedly triggered, the file descriptor count increases monotonically. The growth rate is initially high but gradually slows down as the count rises, due to reduced effective request throughput and resource acquisition under increasing resource pressure near exhaustion. After 391.4 seconds, the process reaches 32,768 open file descriptors and subsequently crashes. Note that even on devices with a higher file descriptor limit, the app may not crash at this exact point. Nevertheless, as long as the leak persists, the steadily increasing file descriptor count can accumulate over time and eventually trigger an availability failure. This case highlights the severe consequences of GC-unmanaged resource leaks and underscores the importance of leak detection for mitigating resource-exhaustion DoS risks.

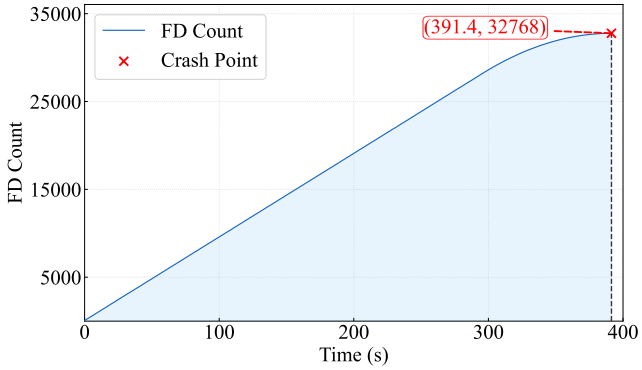


Figure 7: File descriptor (FD) count curve.

6 Limitations

LLM Reasoning Reliability LeakHunter relies on an LLM to infer resource-management operations and leak-relevant exceptions that are difficult to capture with purely heuristic analysis. This design therefore inherits the LLM's imperfect reasoning and potential knowledge gaps. While we apply static validation to filter and refine the LLM's outputs, it cannot fully eliminate erroneous outputs or address incorrect detections caused by missed LLM inferences. For example, our experiments show that the LLM may miss some leak-relevant exceptions, causing certain leak-prone paths to be skipped and thus preventing the tool from successfully identifying the leaks. To address this, we plan to fine-tune the model on a curated, labeled resource-leak corpus. This domain adaptation is expected to help the LLM more effectively infer resource-management APIs and leak-relevant exceptions.

External Triggerability of Resource Leaks Although resource leaks can cause severe availability and security impacts, not all validated leaks are directly triggerable through externally accessible interfaces. Some leaks are exercised only by app-internal workflows and therefore require additional preconditions to be triggered externally. For example, triggering such leaks may require user interactions that repeatedly exercises the relevant internal workflow. These preconditions make such leaks harder to trigger in practice. Nevertheless, our real-world experiment shows that externally triggerable resource leaks are still prevalent in practice.

Leak-Prone Path Coverage of LMPC We adopt the LMPC algorithm to replace exhaustive path enumeration for generating candidate leak paths. Although LMPC attempts to prioritize paths that are more likely to expose leaks, it does not guarantee full coverage of all leak-prone paths compared with enumeration. Therefore, LMPC may theoretically miss some leak cases. However, our ablation study shows that this design does not degrade performance on either the ground-truth ex-

periments or the real-world experiments. In these experiments, LMPC and full path enumeration identify the same number of true resource leaks, while LMPC achieves significantly higher efficiency than traditional path enumeration.

7 Future Work

This paper explores how to combine the minimum path cover principle with domain-specific knowledge for resource leak detection to simplify the analysis problem, and proposes an LMPC algorithm with strong empirical performance. While our LMPC algorithm is specialized for resource leak detection, the underlying minimum path cover perspective is general. It constructs a compact set of paths that cover all code regions, avoiding exhaustive enumeration. This perspective is most suitable for analyses where task-specific knowledge can characterize the importance of paths, and analyzing a small number of high-value paths is sufficient to establish the target property. In this setting, MPC can potentially serve as a reusable path-space reduction mechanism for other path-related analyses. For example, in taint analysis, it could be used to prioritize a small set of representative source-to-sink paths that cover critical nodes and edges, improving scalability. We leave this to future work.

8 Conclusion

Resource-exhaustion DoS attacks triggered by resource leaks remain a serious threat to Java programs. Existing approaches often fail to capture hidden leak-relevant exceptions and are constrained by the scalability bottleneck of path enumeration. To address these issues, we propose LeakHunter, a novel resource leak detector for identifying GC-unmanaged resource leaks in Java programs. LeakHunter leverages an LLM to infer leak-relevant exceptions and uses static analysis to further validate the inferred results, exposing latent leak paths that are difficult to capture with conventional heuristics. We further introduce LMPC, which replaces path enumeration with a compact set of representative paths and prioritizes the most leak-prone paths for validation. Extensive experiments demonstrate that LeakHunter substantially outperforms existing approaches in both effectiveness and efficiency.

Ethical Considerations

This paper studies GC-unmanaged resource leaks in Java programs and their security implications, in particular the risk of resource-exhaustion denial of service (DoS) when leak-inducing paths can be triggered repeatedly. We present LeakHunter, a static analysis tool that prioritizes and instantiates a bounded set of witness paths to help developers identify leak defects with potential availability impact. Following the USENIX Security'26 ethics guidelines, we summarize a stakeholder-based ethics analysis below.

Stakeholders The stakeholders potentially impacted by this work include: (1) maintainers and contributors of the analyzed Java/Android open-source projects; (2) end users and organizations that depend on these projects and may be affected by availability failures; (3) the security community (defenders, auditors, and researchers) who may use our techniques to improve software robustness; (4) potential adversaries who could misuse improved leak-finding techniques; and (5) the research team and reviewers involved in developing and evaluating this work.

Benefits The primary expected benefit is improved availability: identifying GC-unmanaged resource leaks earlier can help maintainers prevent outages and resource-exhaustion DoS conditions. Our approach also aims to reduce analysis overhead compared to naive path enumeration by generating a small set of representative, leak-prone paths, making leak auditing more practical for large codebases.

Potential Harms and Dual Use LeakHunter (and similar analysis techniques) may be misused to accelerate the discovery of DoS-relevant resource leaks by attackers. In addition, imperfect reports (false positives or non-actionable warnings) may impose time and attention costs on maintainers. Finally, our pipeline includes an LLM-assisted component for reasoning about resource-management operations and leak-relevant exceptions; in settings where users analyze proprietary code with a third-party LLM service, this may raise confidentiality concerns.

Mitigations We took the following steps to mitigate harms: (i) *No testing on live systems without consent.* Our evaluation is performed offline on publicly available source code and does not involve probing or stressing third-party production services. (ii) *Reducing maintainers' burden.* Before reporting real-world findings, we manually validated tool outputs and organized issues into actionable reports, aiming to avoid flooding communities with low-confidence warnings. (iii) *Responsible disclosure.* We reported validated issues to the responsible open-source communities via their standard channels and tracked remediation outcomes (e.g., confirmations and merged fixes). We do not provide exploit scripts or step-by-step instructions for turning an individual leak into an operational DoS attack; the paper focuses on root-cause identification and defensive guidance.

Decision and Remaining Risks We believe the benefits of improving availability and helping developers remove leak defects outweigh the residual risks of dual use. Nevertheless, dual-use concerns cannot be eliminated entirely: any technique that improves defect discovery may also aid adversaries. We therefore pair publication with disclosure and remediation efforts and aim to release artifacts in a manner consistent with responsible use (see the Open Science Section).

Open Science

We release LeakHunter's prototype and evaluation datasets on Zenodo to facilitate reproduction and reuse. To support responsible use, the public artifact is intended for defensive analysis and does not include automated exploitation/DoS triggers or instructions for attacking third-party systems; any sensitive details about real-world issues are released only after coordinated disclosure and remediation where applicable.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable feedback and guidance in improving this paper. This work was sponsored by the National Natural Science Foundation of China under Grant No. 62272351 and Grant No. 62572354. Jiang Ming is supported by National Science Foundation (NSF) Grant No. 2417055 and a Google Research Scholar Award.

References

[1] Michael Nieles, Kelley Dempsey, and Victoria Yan Pillitteri. An Introduction to Information Security. *NIST Special Publication*, 800:12, 2017.

[2] Common Weakness Enumeration. CWE-404: Improper Resource Shutdown or Release. https://cwe.mitre.org/data/definitions/404.html. Accessed: 2025-11-1.

[3] Common Weakness Enumeration. CWE-772: Missing Release of Resource after Effective Lifetime. https://cwe.mitre.org/data/definitions/772.html. Accessed: 2025-11-1.

[4] Akshay Utture and Jens Palsberg. From Leaks to Fixes: Automated Repairs for Resource Leak Warnings. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE' 23)*, 2023.

[5] Tianyang Liu, Weixing Ji, Xiaohui Dong, Wuhuang Yao, Yizhuo Wang, Hui Liu, Haiyang Peng, and Yuxuan Wang. JLeaks: A Featured Resource Leak Repository Collected From Hundreds of Open-Source Java Projects. In *Proceedings of the 46th ACM/IEEE International Conference on Software Engineering (ICSE '24)*, 2024.

[6] Oracle. Java Garbage Collection Basics. https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. Accessed: 2025-11-1.

[7] Emina Torlak and Satish Chandra. Effective Interprocedural Resource Leak Detection. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, pages 535–544, 2010.

[8] Tianyong Wu, Jierui Liu, Xi Deng, Jun Yan, and Jian Zhang. Relda2: An Effective Static Analysis Tool for Resource Leak Detection in Android Apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*, pages 762–767, 2016.

[9] Chong Wang, Yiling Lou, Xin Peng, Jianan Liu, and Baihan Zou. Mining Resource-Operation Knowledge to Support Resource Leak Detection. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE' 23)*, 2023.

[10] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Boosting Static Resource Leak Detection via LLM-based Resource-Oriented Intention Inference. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE' 25)*, 2025.

[11] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps. *IEEE Transactions on Software Engineering*, 42(11), 2016.

[12] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D Ernst. Lightweight and Modular Resource Leak Verification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE' 21)*, 2021.

[13] Facebook. Infer: A Tool to Detect Bugs in Java and C/C++/Objective-C Code Before it Ships. https://fbinfer.com/. Accessed: 2025-11-1.

[14] SpotBugs Team. SpotBugs: A Tool for Static Analysis to Look for Bugs in Java code. https://spotbugs.github.io/. Accessed: 2025-11-1.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS '17)*, 2017.

[16] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A Comprehensive Overview of Large Language Models. *ACM Transactions on Intelligent Systems and Technology*, 16(5), 2025.

[17] Shuangjie Yao and Dongdong She. Empc: Effective Path Prioritization for Symbolic Execution with Path Cover. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy (S&P '25)*. IEEE, 2025.

[18] Robert Z Norman and Michael O Rabin. An Algorithm for a Minimum Cover of a Graph. *Proceedings of the American Mathematical Society*, 10(2):315–319, 1959.

[19] Krishnaiyan Thulasiraman and Madisetti NS Swamy. *Graphs: Theory and Algorithms*. John Wiley & Sons, 2011.

[20] Yepang Liu, Jue Wang, Lili Wei, Chang Xu, Shing-Chi Cheung, Tianyong Wu, Jun Yan, and Jian Zhang. DroidLeaks: A Comprehensive Database of Resource Leaks in Android Apps. *Empirical Software Engineering*, 24(6):3435–3483, 2019.

[21] Adangel. PMD Source Code Analyzer. https://pmd.github.io/. Accessed: 2025-05-29.

[22] Sonarsource. SonarQube: Code Quality and Security. https://www.sonarsource.com/products/sonarqube. Accessed: 2025-05-29.

[23] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. Are Code Examples on an Online Q&A Forum Reliable? A Study of API Misuse on Stack Overflow. In *Proceedings of the 40th international conference on software engineering (ICSE '18)*, 2018.

[24] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. How Practitioners Perceive the Relevance of Software Engineering Research. In *Proceedings of the 10th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE' 15)*, 2015.

[25] Appsmithorg. Issue 41495. https://github.com/appsmithorg/appsmith/issues/41495. Accessed: 2025-11-1.

[26] Krabaey. Commit D820369. https://github.com/osmandapp/Osmand/commit/d82036938acf78efc8bbdbd22697581ed963aaea. Accessed: 2025-11-1.

[27] Xuxueli. Commit 783de41. https://github.com/xuxueli/xxl-job/commit/783de41921fe1692fc1fb0a0dddb1bc3b9ba23ec. Accessed: 2025-11-1.

[28] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. Rampart: Protecting Web Applications from CPU-Exhaustion Denial-of-Service Attacks. In *Proceedings of the 27th USENIX Security Symposium (USENIX Sec '18)*, 2018.

[29] Keke Lian, Lei Zhang, Haoran Zhao, Yinzhi Cao, Yongheng Liu, Fute Sun, Yuan Zhang, and Min Yang. Careless Retention and Management: Understanding and Detecting Data Retention Denial-of-Service Vulnerabilities in Java Web Containers. In *Proceedings of the 34th USENIX Security Symposium (USENIX Sec '25)*, 2025.

[30] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective Analysis: Context-Sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, 2014.

[31] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS '20)*, 2020.

[32] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training Language Models to Follow Instructions with Human Feedback. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS' 22)*, 2022.

[33] Xinzhe Li. A Review of Prominent Paradigms for LLM-Based Agents: Tool Use, Planning (Including RAG), and Feedback Learning. In *Proceedings of the 31st International Conference on Computational Linguistics (COLING '25)*, pages 9760–9779, 2025.

[34] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa: Escaping Coverage Plateaus In Generation with Pre-trained Large Language Models. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE '23)*, 2023.

[35] Flemming Nielson and Hanne Riis Nielson. Interprocedural Control Flow Analysis. In *Proceedings of the 8th*

European Symposium on Programming Languages and Systems, pages 20–39, 1999.

[36] SC Ntafos and SL Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions on software engineering*, 5(5), 1979.

[37] David P Williamson and David B Shmoys. *The Design of Approximation Algorithms*. Cambridge university press, 2011.

[38] Delbert R Fulkerson. Note On Dilworth's Decomposition Theorem For Partially Ordered Sets. *Proceedings of the 1956 American Mathematical Society*, 7(4):701–702, 1956.

[39] Robert P Dilworth. A Decomposition Theorem for Partially Ordered Sets. In *The Dilworth Theorems: Selected Papers of Robert P. Dilworth*, pages 7–12. 1990.

[40] WingCHWang. Srctoolkit. https://pypi.org/project/srctoolkit/. Accessed: 2025-11-1.

[41] OpenAI. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/. Accessed: 2025-11-1.

[42] Meta. Llama 4 | Model Cards and Prompt formats. https://www.llama.com/docs/model-cards-and-prompt-formats/llama4/. Accessed: 2025-11-1.

[43] Google. Gemma 3 Model Overview. https://deepmind.google/models/gemma/gemma-3/. Accessed: 2025-11-1.

[44] David MW Powers. Evaluation: From Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation. *arXiv preprint arXiv:2010.16061*, 2020.

[45] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. Understanding and Detecting Callback Compatibility Issues for Android Applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, 2018.

[46] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-Law Distributions in Empirical Data. *SIAM review*, 51(4), 2009.

[47] Xiaoying Zhang, Baolin Peng, Ye Tian, Jingyan Zhou, Lifeng Jin, Linfeng Song, Haitao Mi, and Helen Meng. Self-Alignment for Factuality: Mitigating Hallucinations in LLMs via Self-Evaluation. *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL '24)*, 2024.

Appendix

A LeakHunter prompt and LLM output

Problem Description:

Do you know non-memory Java resource leaks? A non-memory resource leak occurs when a program fails to properly release an acquired resource that is not automatically or reliably reclaimed by Java's garbage collector. If you understand this problem, please perform the following tasks.

Task Requirements:

(1) Resolve the types of the involved objects, then identify which types represent leakable resources.
(2) Identify the API/method calls that acquire these resources. After that, identify the API/method calls that release the acquired resources.
(3) Identify the throwing points of leak-relevant exceptions that redirect control flow and will definitely bypass the code that releases acquired resources (excluding leaks caused by exceptions thrown during resource release).

Desired Format:

Leakable Resources:
<resource type>: <resource variable>

API/Method Calls for Acquiring Resources:
line <line number>: `<API call>` acquires `<resource variable>` resource

API/Method Calls for Releasing Resources:
line <line number>: `<API call>` releases `<resource variable>` resource

Leak-Relevant Exceptions:
line <line number>: `< throwing points of leak-relevant exception>` will definitely cause the resource-release code to be skipped

Input Code:

Java Source Code ```{ Code }```

Figure A1: Prompt Template of LeakHunter.

Leakable Resources:

java.io.InputStream: in
java.io.OutputStream: out

API/Method Calls for Acquiring Resources:

line 1: new `FileInputStream("1.txt")` acquires `in` resource
line 2: new `FileOutputStream("2.txt")` acquires `out` resource

API/Method Calls for Releasing Resources:

line 16: `in.close()` releases `in` resource
line 17: `out.close()` releases `out` resource

Leak-Relevant Exceptions:

line 2: `new FileOutputStream("2.txt")` will definitely cause the resource-release code to be skipped

Figure A2: LLM output for the prompt in Figure A1, using the Java source code in Figure 2(a) (please note that the line numbers here correspond to node IDs in Figure 2(a)).

We present our prompt in Figure A1. The first part is the *problem description*, where we specify the task context and what the LLM needs to understand. The second part, *task requirements*, outlines the concrete tasks we expect the LLM to perform, including resolving the types of resource-managing objects and their initial identifiers, locating the source lines where resources are acquired or released, and inferring leak-relevant exceptions that can ultimately bypass the corresponding resource release operations. The third part, *desired format*,

defines the exact structure of the expected output. Finally, the fourth part appends the Java source code to be analyzed. Typically, the input is organized at the method level, meaning that each query provides the source code of a single Java method that the user wants to check for leaks. To facilitate the LLM's domain understanding and enable meaningful responses, in the *problem description*, we first ask whether it comprehends the notion of resource leaks and provide a concise description of the problem. Only when the LLM indicates that it understands the problem do we ask it to perform the subsequent tasks. This design helps guide the LLM to focus on the specific technical domain and reduces hallucinations [47].

B MPC Generation Algorithm

We present the MPC generation algorithm in Algorithm 2. The algorithm first builds a bipartite graph that encodes reachability in the input DAG: each original vertex v_i is copied into x_i and y_i, and an edge (x_i, y_j) is added if $i \neq j$ and v_i can reach v_j in the DAG (lines 2–3). A maximum matching on this bipartite graph then selects as many non-conflicting reachability pairs as possible, where no two selected pairs share the same copied vertex (line 4). The algorithm then maps each matched pair back to a concrete subpath p_{sub} in the original DAG (lines 5–10). If the two matched vertices are directly connected by an edge, that edge forms p_{sub}; otherwise, the algorithm retrieves a concrete path from v_i to v_j to materialize the reachability relation.

Next, the algorithm inserts p_{sub} into the current cover by repeatedly merging it with an existing path p_i whose end vertex matches the start vertex of p_{sub}, or whose start vertex matches the end vertex of p_{sub} (lines 11–21). If no such path exists, the current p_{sub} is inserted into the cover as a new path (lines 13–15). Otherwise, p_i and p_{sub} are concatenated into a longer path according to the matched start/end vertices, p_i is removed from the cover, and the merged path becomes the new p_{sub} for the next merge attempt (lines 16–20). After all matched pairs are processed, any vertex not covered by the constructed paths is added as a singleton path (line 23), and the resulting path set is returned as one MPC for the DAG.

Algorithm 2 Generate One MPC for a DAG

Input: DAG = $G(V,E)$ with vertex set V and edge set E
Output: MPC

1: $MPC \leftarrow \{\}$; $n \leftarrow |V|$, $m \leftarrow |E|$
2: $V_b = \{x_1, x_2, \ldots, x_n\} \cup \{y_1, y_2, \ldots, y_n\}$; $E_b = \{(x_i, y_j) \mid 1 \leq i, j \leq n,\ i \neq j,\ v_i$ reaches v_j in $G\}$
3: $G_b = (V_b, E_b) =$ BUILD_BIPARTITE_GRAPH(G)
4: $M_m = \{(x_{i_1}, y_{j_1}), (x_{i_2}, y_{j_2}), \ldots, (x_{i_c}, y_{j_c})\} =$ GET_MAXIMUM_MATCHING(G_b)
5: **for** $(x_{ik}, y_{jk}) \in M_m$ **do**
6: **if** $(v_{ik}, v_{jk}) \notin E$ **then**
7: $p_{sub} =$ BUILD_REACHED_SUBPATH(G, v_{ik}, v_{jk})
8: **else**
9: $p_{sub} =$ BUILD_DIRECT_SUBPATH(v_{ik}, v_{jk})
10: **end if**
11: **while true do**
12: Find a path $p_i \in MPC$ such that end_vertex$(p_i) =$ start_vertex(p_{sub}) or end_vertex$(p_{sub}) =$ start_vertex(p_i)
13: **if** no such path p_i exists **then**
14: $MPC \leftarrow MPC \cup \{p_{sub}\}$
15: **break**
16: **else**
17: $p_i' \leftarrow$ MERGE(p_i, p_{sub})
18: $MPC \leftarrow MPC \setminus \{p_i\}$
19: $p_{sub} \leftarrow p_i'$
20: **end if**
21: **end while**
22: **end for**
23: $MPC \leftarrow MPC \cup \{\langle v \rangle \mid v \in V$ and v is not covered by any path in $MPC\}$
24: **return** MPC

C Threshold Tuning

Since the threshold K affects LeakHunter's performance, a larger K allows LMPC to compose more witness paths but also incurs higher analysis time. Therefore, we empirically tuned K by constructing 50 additional resource-leak cases and evaluating LeakHunter under different K values. Figure A3 reports the resulting F1-scores. Overall, the F1-score varies only slightly as K changes, indicating that generating a small number of MPCs per DAG and prioritizing the most leak-prone paths is sufficient to detect the vast majority of resource leaks. Moreover, increasing K beyond 3 yields no additional gain; therefore, we set K=3 for all subsequent experiments.

D LLM Time Overhead

We report the average LLM analysis time of LeakHunter and InferROI on the ground-truth datasets in Table A1. The results show that the two tools have comparable LLM analysis time.

Table A1: The average per-case LLM overhead on the ground-truth dataset.

Tool	DroidLeaks	JLeaks	All
LeakHunter	6.3s	8.9s	8.7s
InferROI	6.2s	9.0s	8.8s

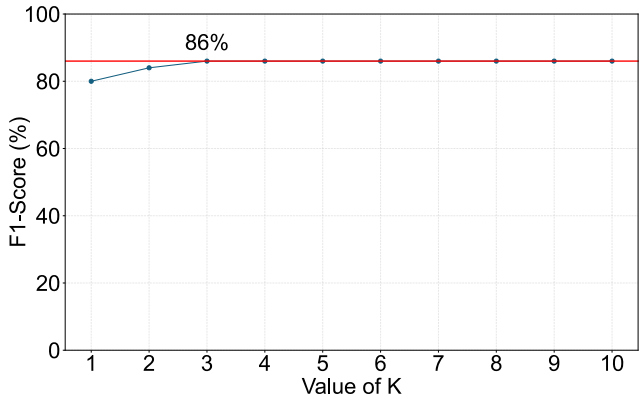


Figure A3: Tuning results of K.

E Real-World Results Validation Procedure

To reduce author bias in validating real-world results, we adopted a five-person verification protocol: one researcher (the "referee") aggregated all results, anonymized them by assigning IDs, and was the only person who knew which tool produced each result. The remaining four researchers formed two groups that independently reviewed all results and labeled each ID. When the two groups disagreed on an ID, they discussed the case to reach a final consensus label. Finally, the referee collected and summarized the final labels for all tools. We then reported all validated leaks to the corresponding app vendors to avoid ethical concerns.

F Real-World App Categories and Detected Leaks

Table A2: Statistics of tested apps and validated leaks across categories

Category	Media	Social	Reading	Framework	Business	Total
# Apps	23	14	9	28	26	100
# Leaks	48	7	2	9	24	90

The apps collected from GitHub span five distinct categories: media, social, reading, framework, and business. Combining Table A2 with Table 5, we find that LeakHunter identified 111 resource leaks in the dataset of 100 apps. Subsequent manual verification confirmed 90 of them as true leaks. Notably, 15 of these leaks have already been confirmed by the developers. These confirmed cases are distributed across 4 media apps (8 leaks), 3 social apps (4 leaks), and 3 business apps (3 leaks), involving resource types such as thread pools, database cursors, and file descriptors.

G Developer Feedback

We interacted with developers through a structured reporting process. We first submitted detailed reports for the resource leaks we had manually validated. In each report, we described the root cause of the leak, explained how the leak could be repeatedly triggered, and discussed the security consequences that such repeated triggering could cause. After receiving our reports, developers assessed the reported leaks by verifying the existence of the leaks and the validity of our reports, and then provided responses based on their validation. Some responses further included explicit security assessments, concrete fix information, or acknowledgment of a planned fix. For more up-to-date developer feedback, such as links to fix patches, please refer to the repository linked in our **Open Science** section.