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

**Lige Zhan**[1,2], Yafei He[1], Jiang Ming[2], Guojun Peng[1], Jianming Fu[1]

[1]Wuhan University

[2]Tulane University

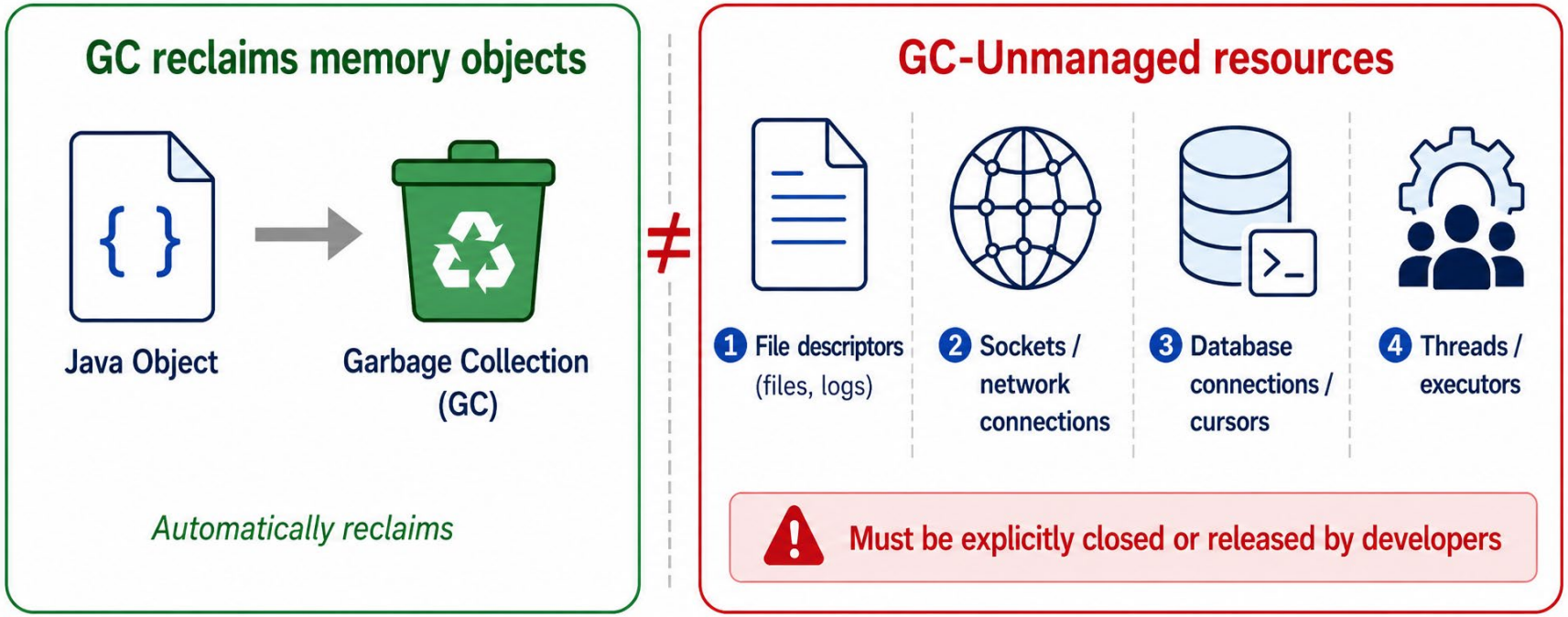





# GC-Unmanaged Resources

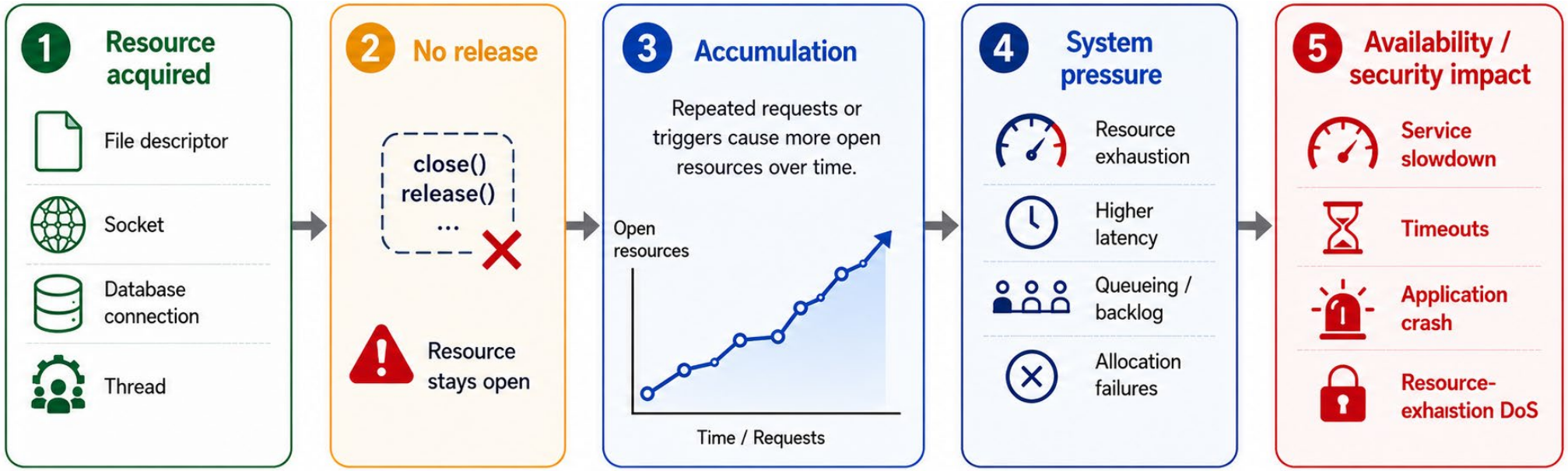
- **Definition**

  GC (garbage collection)-unmanaged resources are system resources that Java's garbage collection does not automatically reclaim.



# Why GC-Unmanaged Resource Leaks Matter

- **Consequences of unreleased GC-unmanaged resources**



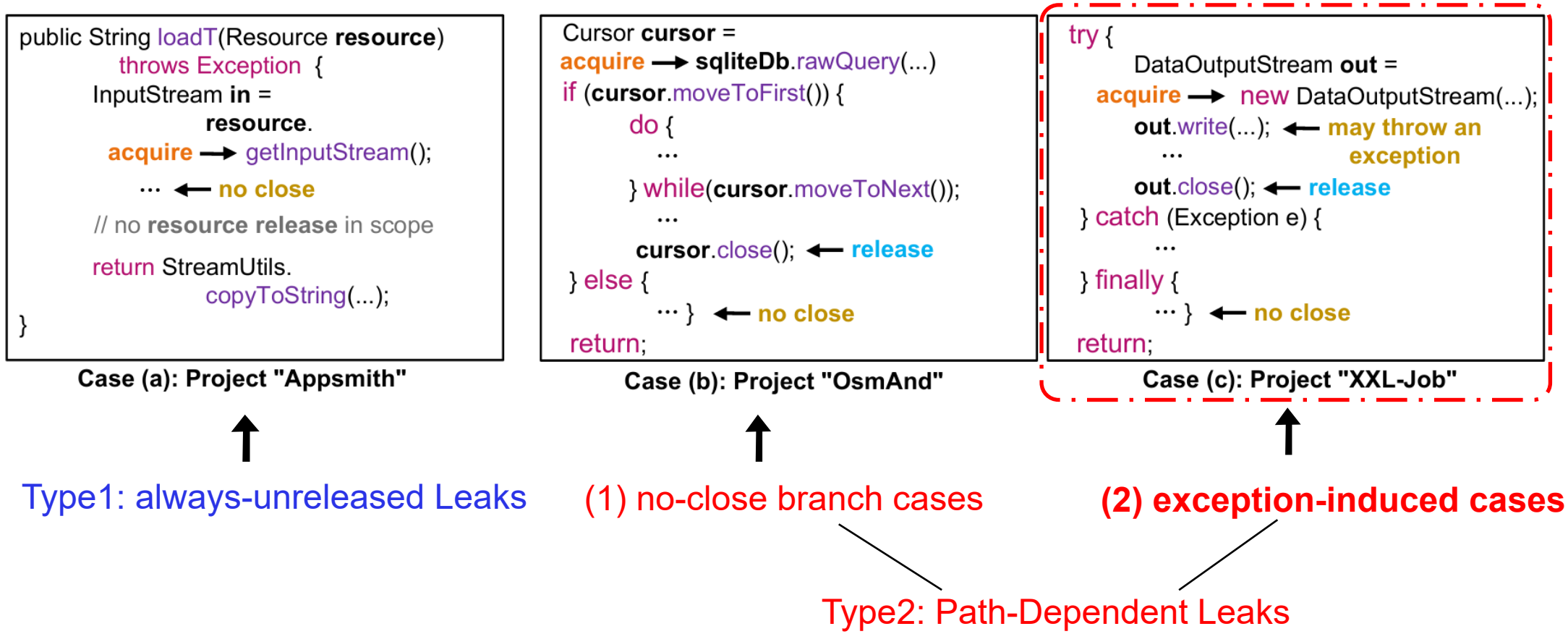

## Why this matters

**Accumulation effect**
Small leaks accumulate and can eventually exhaust system resources.

**Attack amplification**
Attackers can repeatedly trigger leaks to cause resource-exhaustion DoS.

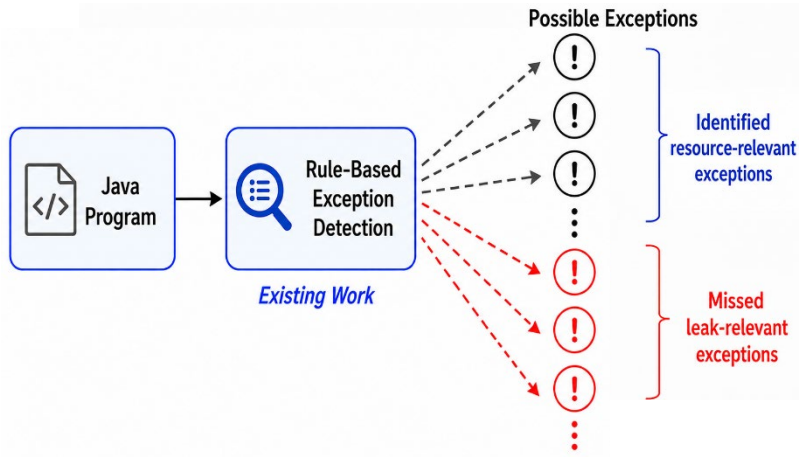
# Common Resource Leak Types

- **Two main types of real-world resource leaks**

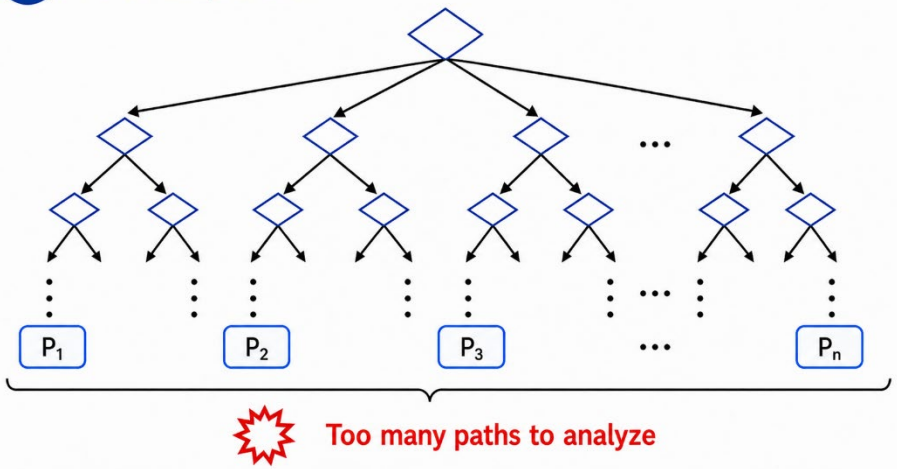


# Limitations of Existing Approaches

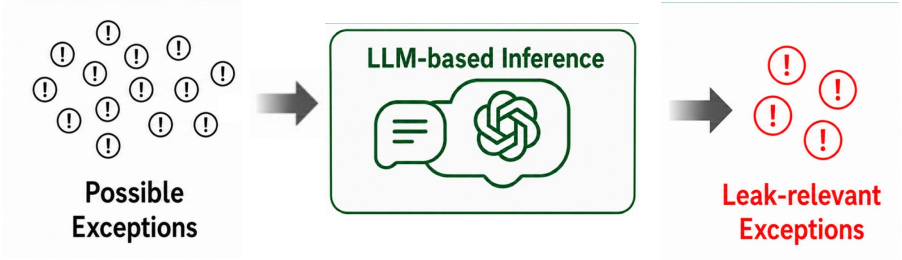
## 1 Incomplete Leak-Relevant Exceptions Handling



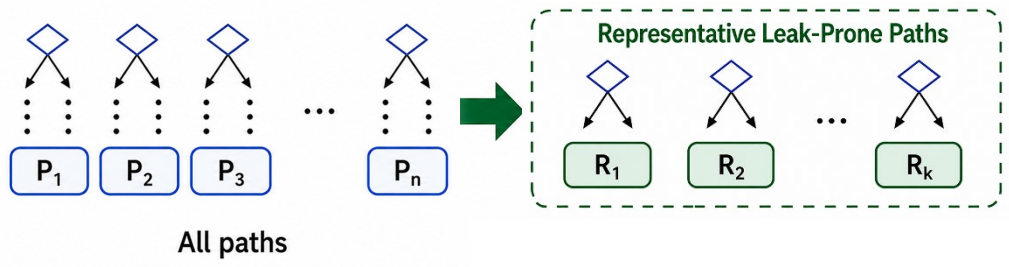

## 2 Path Explosion



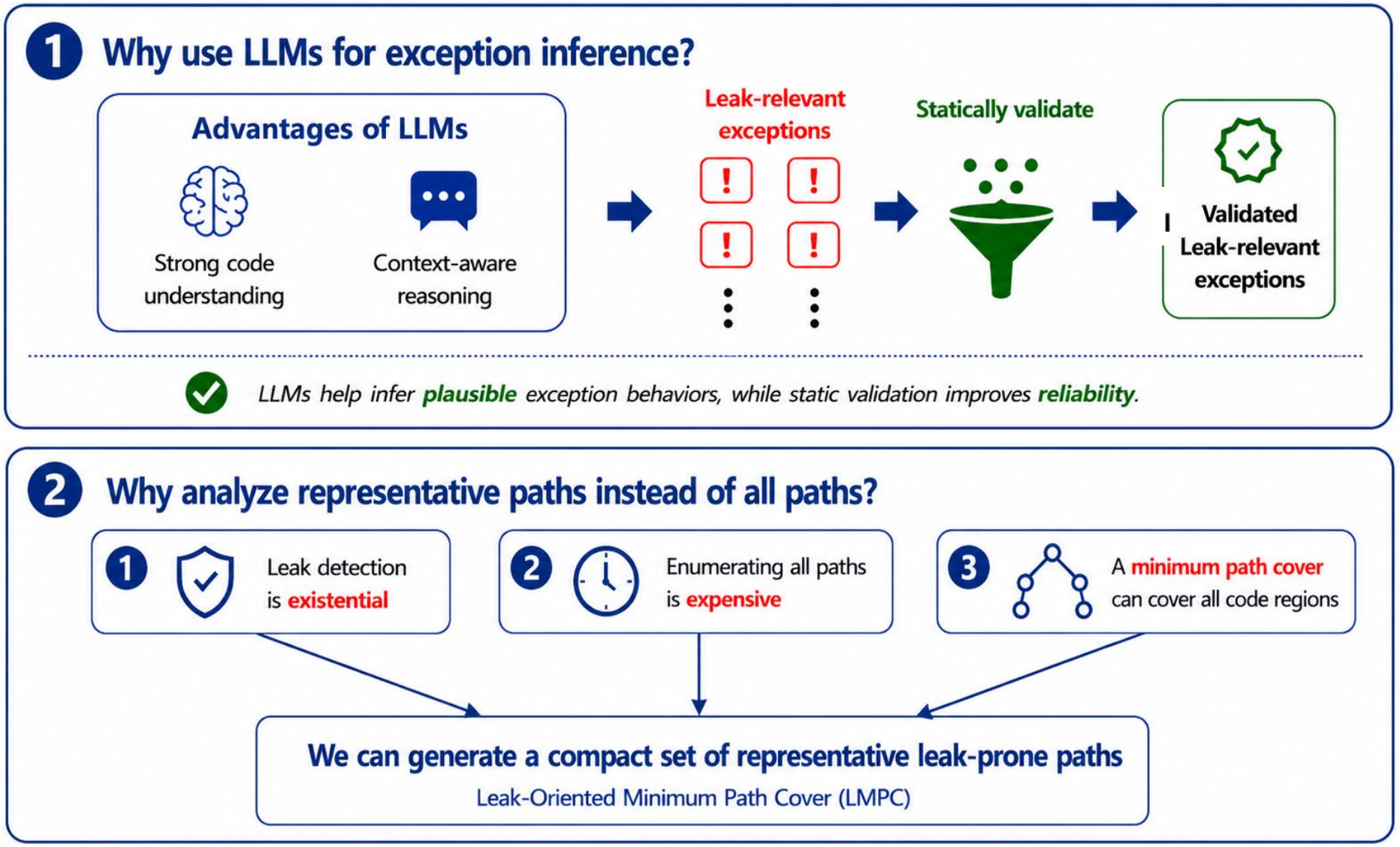

**Our Insight 1** Use LLMs to infer leak-relevant exceptions



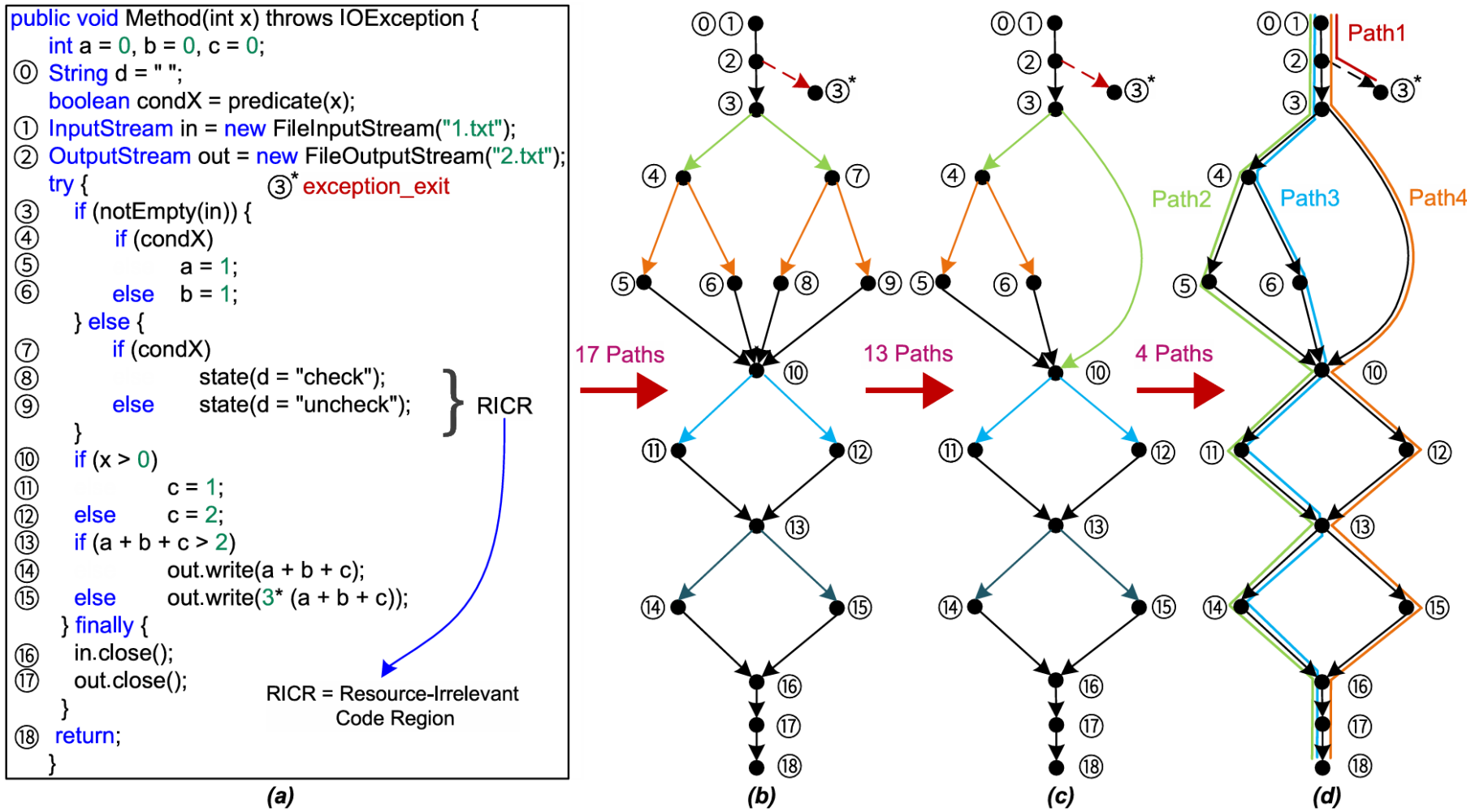

**Our Insight 2** Analyze representative leak-prone paths instead of enumerating all paths



# Reason Behind our Insights

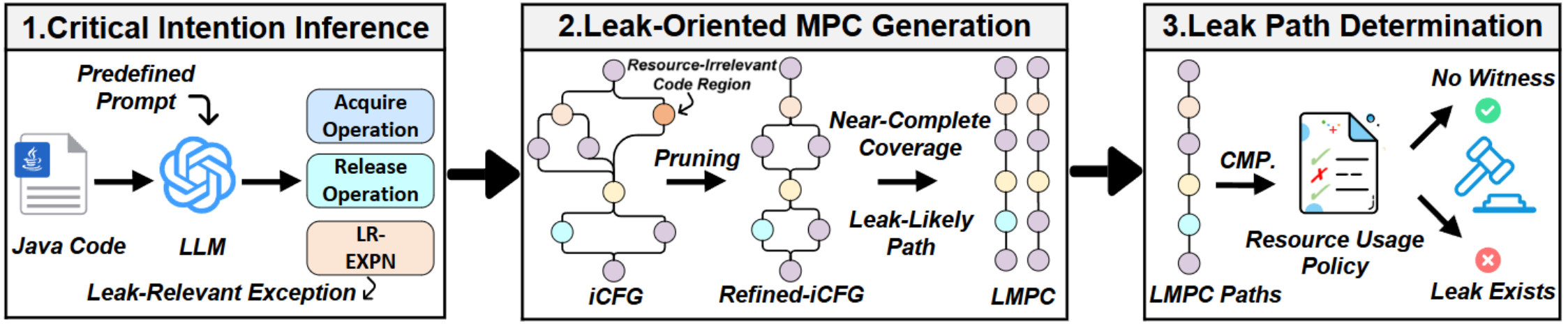


# Motivating Example




*(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.*

# Workflow of LeakHunter




**Addressing two bottlenecks:**

**(1) Incomplete coverage of leak-relevant exceptions (in phase 1).**

**(2) Prohibitive cost of path enumeration (in phase 2).**

# Critical Intention Inference

## Prompt：

**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 }```

## LLM to help infer:

**(1) Resource-management APIs:** resource acquire and release operations.

**(2) Leak-relevant exceptions**

## Answer：

**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

# Minimum Path Cover Generation

- **Generate refined iCFG**

  **(1) Dependence-based slice：** prune resource-irrelevant code.

  **(2) Lightweight constant folding：** resolve constant branch conditions.

  **(3) Reachability checking:** remove unreachable branches.

- **Validate leak-relevant exception**

  **An LLM-identified candidate exception ⟨*cs* (*Call-site*), *t* (*exception type*)⟩**

  **(C1) Caught locally：** the exceptional edge for *t* from *cs* reaches a compatible catch handler.

  **(C2) Caught by a caller：** along inter-procedural exception propagation on the call graph, *t* can be caught by some caller of the enclosing method of *cs*.

  **(C3) Explicitly declared：** *t* is compatible with an exception type declared in the *throws* clause of the method containing *cs.*

  **C1** ∨ **C2** ∨ **C3** ? 👉 Keep the exception; otherwise, prune it as noise.

# Minimum Path Cover Generation

- **Transform iCFG into DAGs**

  Computing MPCs directly on a cyclic graph is NP-hard, so LeakHunter eliminates two types of cycles—**loop cycles** and **caller-callee cycles**—to transform the cyclic iCFG into DAGs.

  **For loop cycles**

  Unrolling each loop once to break the cycle.

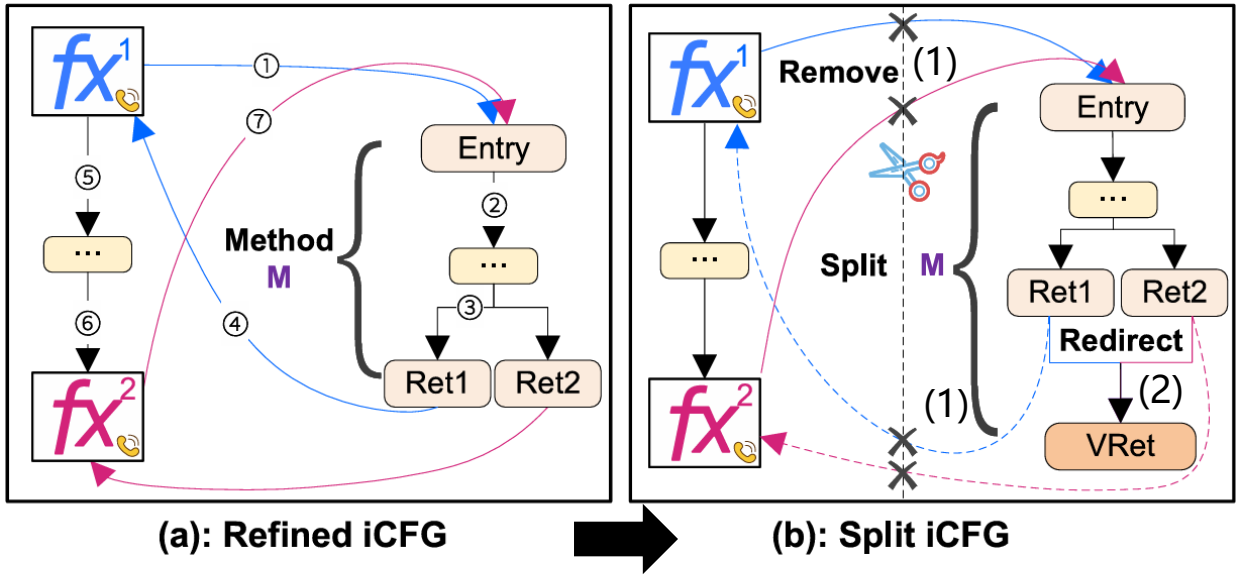
  **For caller-callee cycles**

  (1) Remove call-to-callee edges while retaining call-site vertices.
  (2) Redirect all callee returns to a virtual return node (VRet).

- **Compute MPC from DAG**

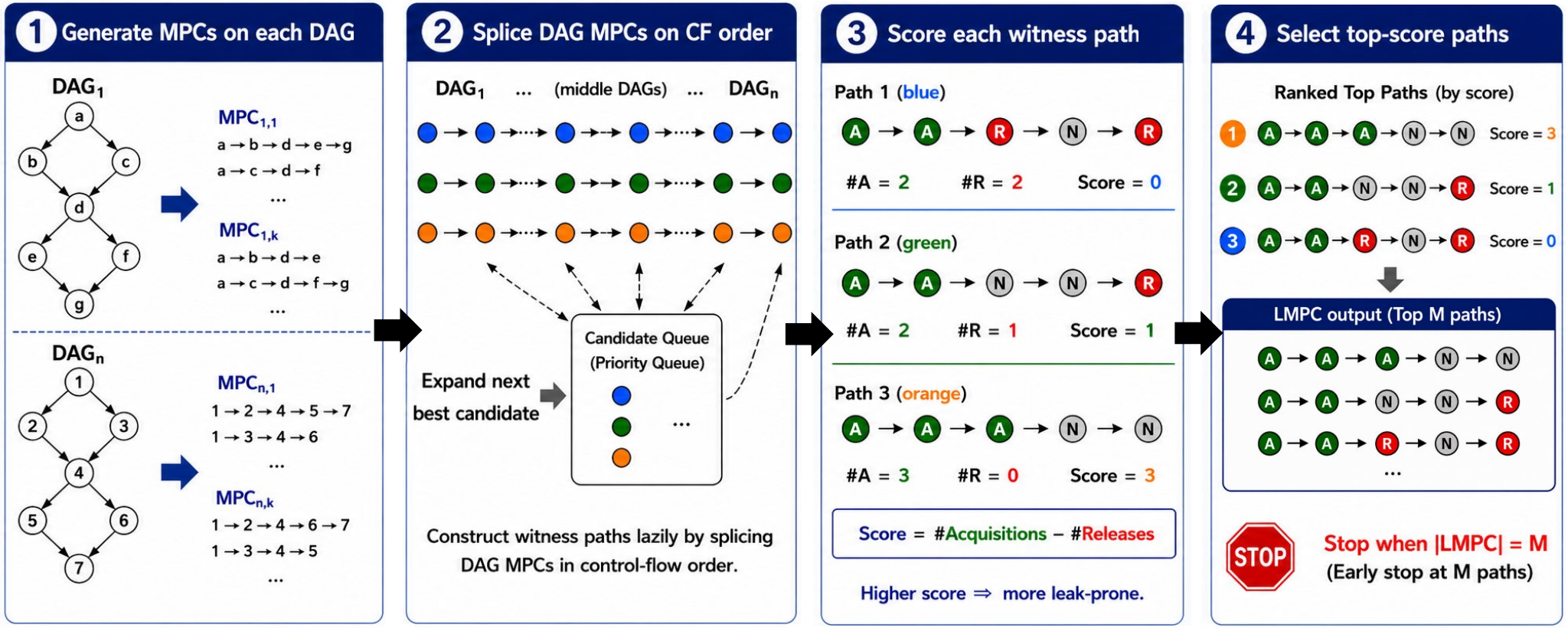
  Then, we compute MPCs separately on each DAG.

**caller-callee cycle :①②③④⑤⑥ re-enters M's entry twice**

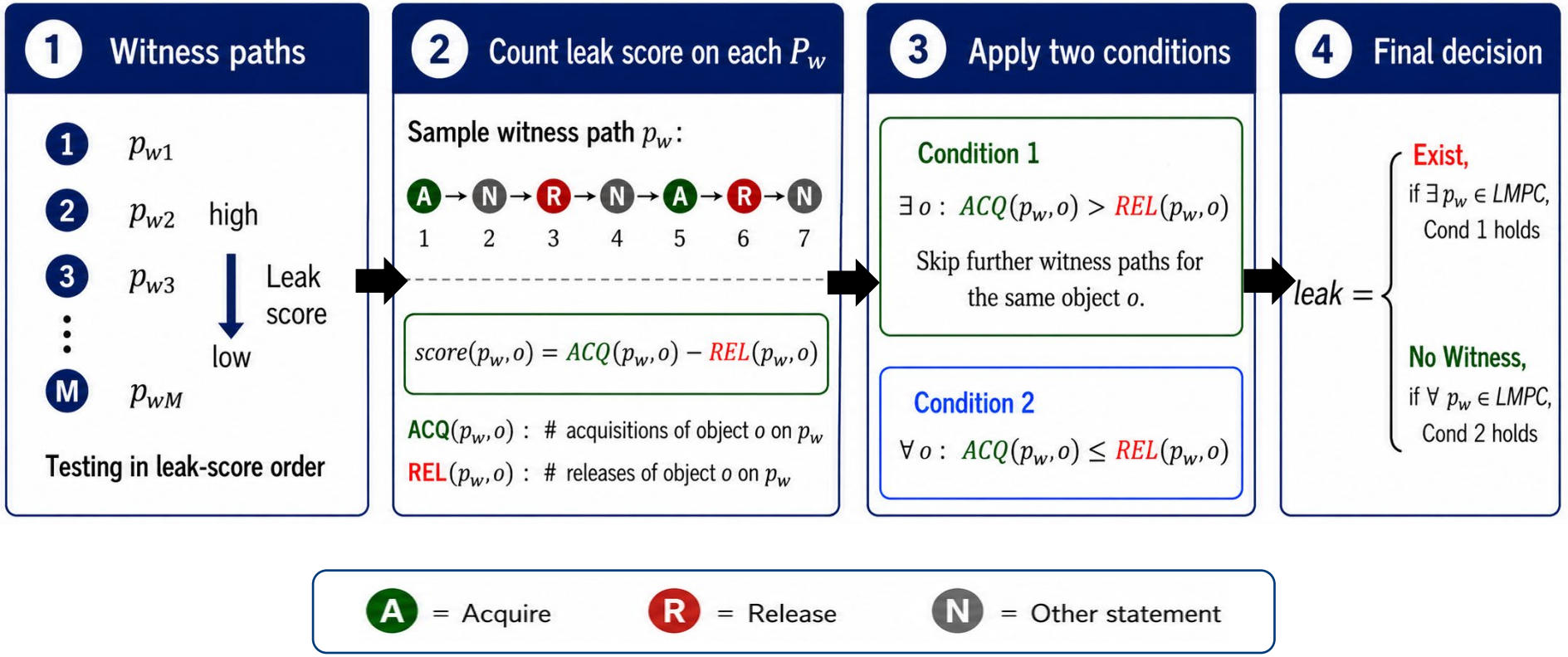


# Leak-Oriented MPC Construction

**Goal:** Select a representative path set that (1) covers all resource-relevant code regions (via MPC), and (2) is biased toward leak-prone behaviors (more acquisitions, fewer releases).



# Leak Path Determination



# Evaluation

- **Testing results on benchmarks**

| 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% |

**10 percentage-point** F1-score improvement over the SOTA (*InferROI*).

| 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%** |

| 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%** |

Best F1-score achieved with GPT-5.

# Benchmark

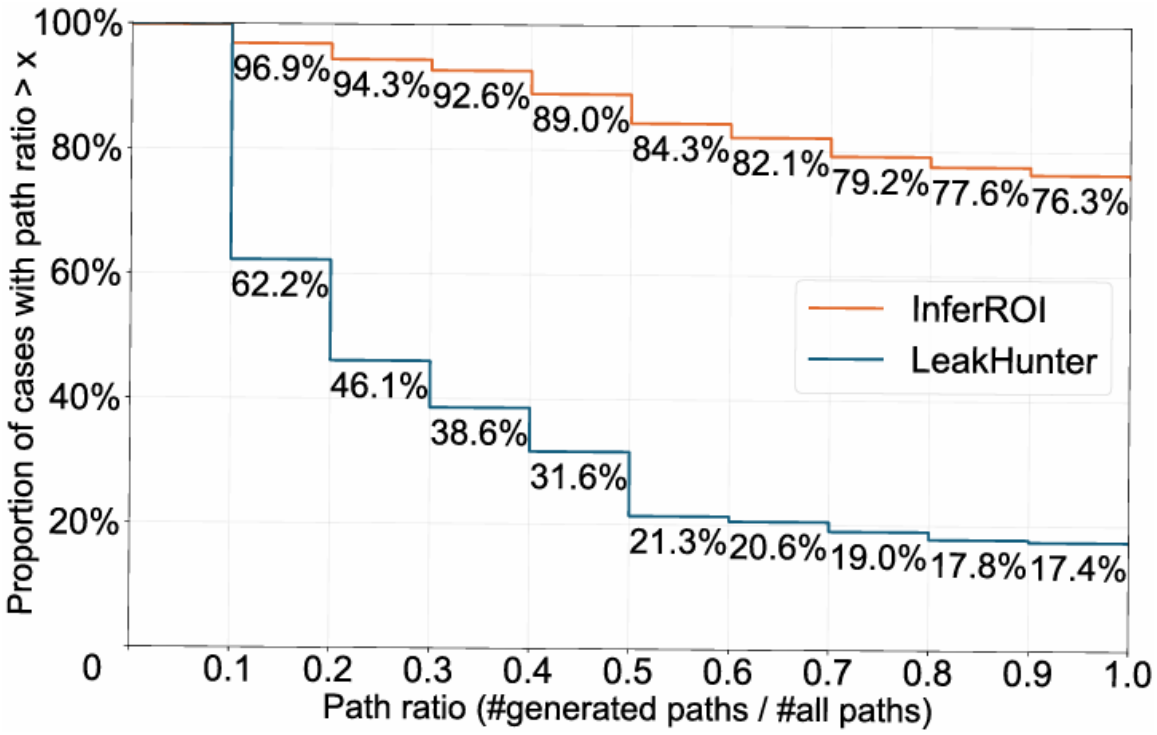
- **Testing overhead on benchmarks**

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$^{I}$ | 0.2s / 0.1s | 1.2s / 0.9s | 1.1s / 0.8s |
| LeakHunter$^{II}$ | 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 |

✓ Average path analysis overhead: LeakHunter = 2.3s (1.3s +1.0s), SOTA = 26.2s (16.4 + 9.8s). LeakHunter achieves **11.4×** speedup.

✓ Only **17.4%** of LeakHunter's cases require analyzing **90–100%** of the paths, compared with **76.3%** for InferROI.



# Real-World Experiment

- **Testing results on 100 real-world applications**

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 |

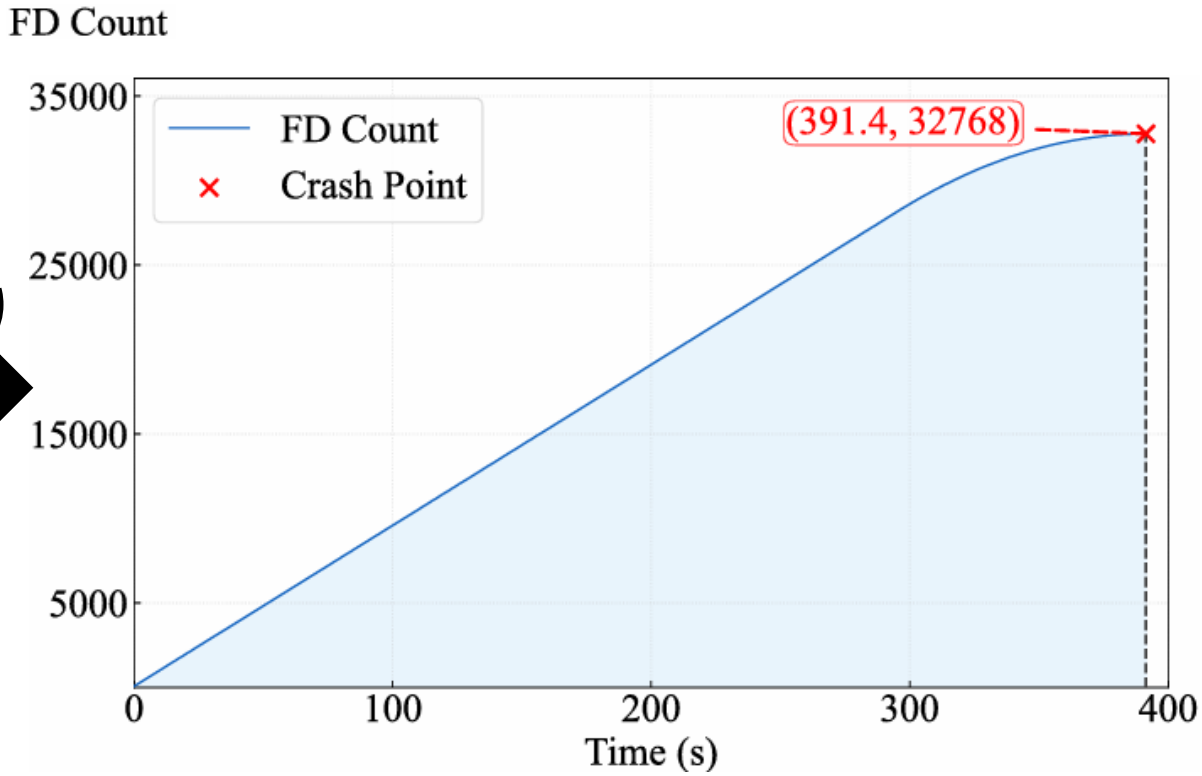
- ✓ LeakHunter reports **111** potential leaks, and our manual validation confirms **90** of them as true leaks.
- ✓ To date, developers have formally confirmed **15** leak defects reported by LeakHunter.
- ✓ LeakHunter achieves **11.6×** speedup in path analysis over SOTA.

# Real-World Experiment

- **Case Study**

```
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
}
```






- An attacker can repeatedly trigger resource leaks through the application's externally accessible interfaces, causing the number of open file descriptors to continuously increase.
- The application process crashes when it reaches the device's (Xiaomi Mi 6) limit of 32,768 file descriptors.

# Conclusion

- This paper reframes GC-unmanaged resource leaks as an availability risk and shows 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 an LMPC algorithm that generates a compact, prioritized set of representative leak-prone paths, mitigating path explosion.
- Comprehensive experiments demonstrate that LeakHunter achieves both effectiveness and efficiency.

# THANKS