Boosting Static Resource Leak Detection via LLM-based Resource-Oriented Intention Inference

Chong Wang†, Jianan Liu†, Xin Peng†, Yang Liu‡, and Yiling Lou†*
†*School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China*
‡*School of Computer Science and Engineering, Nanyang Technological University, Singapore*
{wangchong20, pengxin@fudan.edu.cn, yilinglou@fudan.edu.cn}@fudan.edu.cn
yangliu@ntu.edu.sg

***Abstract*—Resource leaks, caused by resources not being released after acquisition, often lead to performance issues and system crashes. Existing static detection techniques rely on mechanical matching of predefined resource acquisition/release APIs and null-checking conditions to find unreleased resources, suffering from both (1) false negatives caused by the incompleteness of predefined resource acquisition/release APIs and (2) false positives caused by the incompleteness of resource reachability validation identification. To overcome these challenges, we propose INFERROI, a novel approach that leverages the exceptional code comprehension capability of large language models (LLMs) to directly infer resource-oriented intentions (acquisition, release, and reachability validation) in code. INFERROI first prompts the LLM to infer involved intentions for a given code snippet, and then incorporates a two-stage static analysis approach to check control-flow paths for resource leak detection based on the inferred intentions.**

We evaluate the effectiveness of INFERROI in both resource-oriented intention inference and resource leak detection. Experimental results on the DroidLeaks and JLeaks datasets demonstrate INFERROI achieves promising bug detection rate (59.3% and 62.5%) and false alarm rate (18.6% and 19.5%). Compared to three industrial static detectors, INFERROI detects 14~45 and 149~485 more bugs in DroidLeaks and JLeaks, respectively. When applied to real-world open-source projects, INFERROI identifies 29 unknown resource leak bugs (verified by authors), with 7 of them being confirmed by developers. In addition, the results of an ablation study underscores the importance of combining LLM-based inference with static analysis. Finally, manual annotation indicated that INFERROI achieved a precision of 74.6% and a recall of 81.8% in intention inference, covering more than 60% resource types involved in the datasets.

I. INTRODUCTION

Resource leaks, stemming from the failure to release acquired resources (e.g., unclosed file handles), represent critical software defects that can lead to runtime exceptions or program crashes. Such leaks are pervasive in software projects [1] and even appear in online code examples, even for some accepted answers in Stack Overflow posts [2].

To address this issue, researchers have proposed various automated techniques [3]–[7] for resource leak detection based on static analysis. These techniques primarily rely on two essential components. The first step involves identifying pairs of resource acquisition APIs and their corresponding resource release APIs; Subsequently, the code is analyzed to verify whether the release API is not appropriately invoked after the acquisition API on the control-flow paths, before the acquired resources become unreachable. For instance, consider the API pair for managing locks: `LockManager.acquireLock()` represents the resource acquisition API, and `LockManager.releaseLock()` denotes the corresponding resource release API. A resource leak occurs when `releaseLock()` is not subsequently called after `acquireLock()`.

However, these existing techniques mainly rely on mechanical matching of predefined resource acquisition/release APIs and null-checking conditions to find unreleased resources, leading to the following two limitations. ***(1) False negatives from the incompleteness of predefined resource acquisition/release APIs:*** the effectiveness of current resource leak detection techniques [3]–[5] hinges on the completeness of predefined API pairs, since they cannot detect resource leaks related to un-predefined resource acquisition/release API pairs. For example, the widely adopted detectors (e.g., SpotBugs, Infer, and CodeInspection) are confined to detecting only a small subset of common resource classes in JDK and Android [8] by default. Thus, the resource leaks involving emerging or less common resources (e.g., `AndroidHttpClient` in *apache httpclient* library) cannot be detected by these tools if the specific acquisition/release APIs for `AndroidHttpClient` are not additionally configured into these tools. ***(2) False positives from the incompleteness of resource reachability validation identification:*** the accuracy of reachability analysis for resources along control-flow paths significantly influences the occurrence of false alarms in existing detection methods, since unreachable resources would not cause leaks even without being released after acquisition. Approaches like those introduced by Torlak et al. [3] determine resource reachability by scrutinizing `null`-checking conditions (e.g., `cur!=null`) within `if`-statements. However, this approach falls short for certain resources with alternative means of determining reachability, such as through API calls (e.g., `!bank.isDisabled()`). Consequently, false alarms are triggered for resources that are actually *unreachable* along specific paths.

In this paper, we propose **INFERROI**, a novel approach that leverages large language models (LLMs) to boost static resource leak detection. *Our intuition is that LLMs have been trained on massive code and documentation corpus and thus have excep-*

* Yiling Lou is the corresponding author

tional code comprehension capability [9]–[12], suggesting their potential for inferring resource-oriented intentions (i.e., the code statements for resource acquisition/release and resource reachability validation) in any given code snippet. INFERROI consists of two stages, i.e., resource-oriented intention inference and lightweight resource leak detection. (1) For a given code snippet, INFERROI first prompts the LLM to directly infer the involved resource acquisition, release, and reachability validation intentions, without relying on any prior knowledge of resource API pairs. The intentions generated by LLMs (which are described in natural language) are then converted into structured expressions, so as to facilitate the subsequent static analysis stage. (2) INFERROI then incorporates a two-stage static analysis approach to check control-flow paths for resource leaks based on the inferred intentions. In particular, INFERROI first identifies potential leak-risky paths based on the inferred resource acquisition/release APIs and then prunes leak-risky paths with unreachable resources based on the inferred reachability validation, so as to reduce false alarms.

While our approach is general and not limited to specific programming languages, we currently implement INFERROI for Java given its prevalence. We conduct an extensive evaluation to assess the efficacy of INFERROI in detecting resource leaks. Firstly, we apply INFERROI to 86 and 784 bugs sourced from the DroidLeaks [8] and JLeaks [13] datasets, respectively, to evaluate its effectiveness in resource leak detection. The results reveal INFERROI demonstrates a bug detection rate of 59.3% with a false alarm rate of 18.6% for DroidLeaks, and a bug detection rate of 62.5% with a false alarm rate of 19.5% for JLeaks. Compared to three industrial static detectors, INFERROI detects 14~45 and 149~485 more bugs in DroidLeaks and JLeaks, respectively, while maintaining a comparable false alarm rate. Secondly, we employ INFERROI for resource detection in real-world open-source projects. INFERROI uncovers 29 previously unknown resource leaks from 200 methods, 7 of which are confirmed by developers as of the submission time. In addition, an ablation study underscores the importance of combining LLM-based inference with static analysis. The generalizability of INFERROI with different LLMs is confirmed by an evaluation involving Llmam3-8B and Gemma2-9B. Finally, manual annotation indicates that INFERROI achieves a precision of 74.6% and a recall of 81.8% in intention inference, covering 19 (67.9%) and 221 (60.1%) resource types involved in DroidLeaks and JLeaks, respectively.

To summarize, this paper makes the following contributions:

- **A novel LLM-based perspective** that infers resource-oriented intentions (resource acquisition, release, and reachability validation) in code instead of mechanically matching predefined APIs. The inferred intentions can be applied to boost static resource leak detection techniques.
- **A lightweight detection approach INFERROI** that combines the LLM-based intention inference and a static two-stage path analysis to detect resource leaks in code.
- **An extensive evaluation** that demonstrates the effectiveness of INFERROI in inferring resource-oriented intentions and detecting diverse resource leaks in both existing datasets and real-world open-source projects.

II. MOTIVATING EXAMPLE

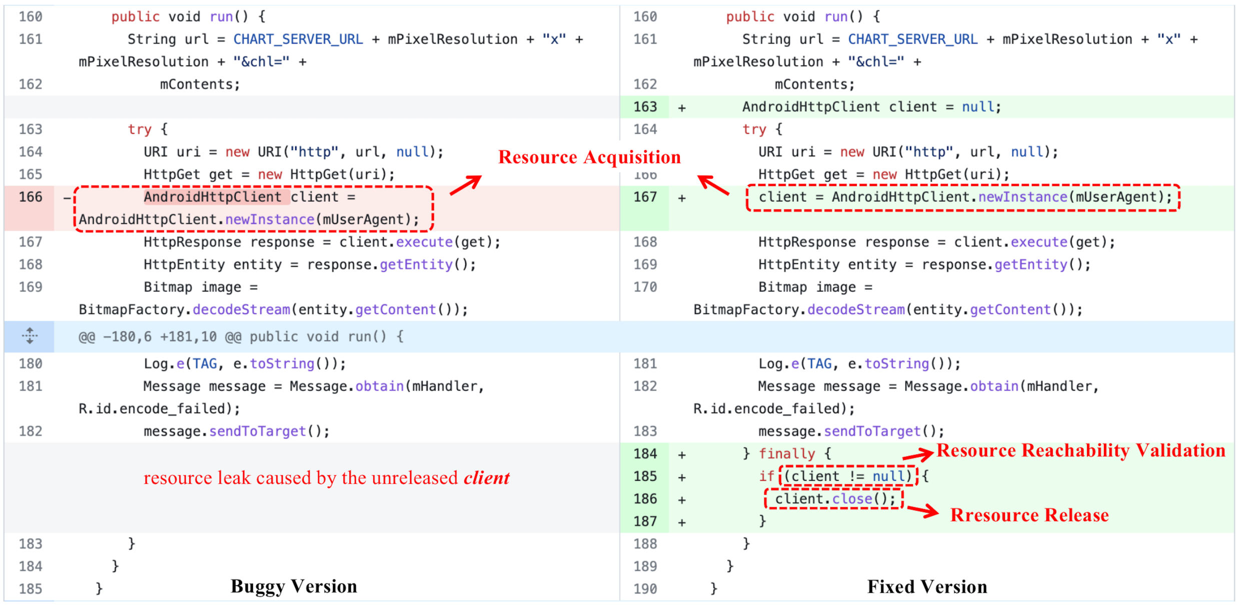
This section presents a motivating example to underscore the significance of directly inferring resource-oriented intentions in code through a combination of background knowledge of resource management and code context understanding, rather than relying solely on mechanical matching of predefined resource acquisition/release APIs and `null`-checking conditions. The resource-oriented intentions refer to the functionalities of resource acquisition, release, and reachability validation involved in code.

Figure 1 illustrates the code diff of a commit [14] of fixing a resource leak associated with `AndroidHttpClient`, as observed in the DroidLeaks dataset. In the buggy version, the API `AndroidHttpClient.newInstance(...)` is called to acquire an *HTTP client* resource at line 166, but the corresponding release API `AndroidHttpClient.close()` is not called, leading to a resource leak. In the fixed version, a `finally` structure is introduced at lines 184-187 to ensure the proper release of the acquired *HTTP client* resource. Through this example, we demonstrate the following key insights:

1. Identifying resource acquisition/release more comprehensively. In the aforementioned example, the presence of a resource leak related to the `AndroidHttpClient` class goes unnoticed by all the eight subject detectors (i.e., Code Inspection [15], Infer [7], Lint [16], SpotBugs [6], Relda2-FS [17], RElda2-FI [17], Elite [18], and Verifier [19]) evaluated in the DroidLeaks dataset. The reason for this oversight lies in the fact that `AndroidHttpClient` is not included in the predefined API sets used by these detectors. This limitation highlights a fundamental constraint of the existing detection paradigm, which primarily relies on locating API calls for resource acquisition/release through the resolution of API signatures in code (i.e., type inference) and subsequent matching with predefined API names.

However, by leveraging background knowledge encompassing pertinent concepts such as *networking* and *HTTP client*, and diligently analyzing the code contexts, which involve relevant terms like `URL` and `HTTPresponse`, it becomes evident that the API call `AndroidHttpClient.newInstance(...)` is specifically intended to acquire an *HTTP client* resource. This realization motivates the exploration of inferring resource acquisition and release intentions based on code semantics (e.g., the API calls and the surrounding contexts), enabling more comprehensive identification of resource leaks compared to conventional mechanical API matching.

Furthermore, in addition to `AndroidHttpClient`, a multitude of APIs are available in various libraries/packages that implement the functionality of *HTTP client*. For instance, within the *selenium* library [20], there exist related APIs like `HttpClient`, `JdkHttpClient`, `NettyClient`, `TracedHttp-Client`, `NettyDomainSocketClient`, `PassthroughHttpClient`, among others. The eight detectors evaluated in DroidLeaks also fail to detect resource leaks across all of them. This failure can



```
160      public void run() {
161        String url = CHART_SERVER_URL + mPixelResolution + "x" +
   mPixelResolution + "&chl=" +
162            mContents;

163        try {
164          URI uri = new URI("http", url, null);
165          HttpGet get = new HttpGet(uri);
166  -       AndroidHttpClient client =
   AndroidHttpClient.newInstance(mUserAgent);
167          HttpResponse response = client.execute(get);
168          HttpEntity entity = response.getEntity();
169          Bitmap image =
   BitmapFactory.decodeStream(entity.getContent());
   @@ -180,6 +181,10 @@ public void run() {
180          Log.e(TAG, e.toString());
181          Message message = Message.obtain(mHandler,
   R.id.encode_failed);
182          message.sendToTarget();

183        }
184      }
185    }
```

```
160      public void run() {
161        String url = CHART_SERVER_URL + mPixelResolution + "x" +
   mPixelResolution + "&chl=" +
162            mContents;
163  +     AndroidHttpClient client = null;
164        try {
165          URI uri = new URI("http", url, null);
166          HttpGet get = new HttpGet(uri);
167  +       client = AndroidHttpClient.newInstance(mUserAgent);

168          HttpResponse response = client.execute(get);
169          HttpEntity entity = response.getEntity();
170          Bitmap image =
   BitmapFactory.decodeStream(entity.getContent());

181          Log.e(TAG, e.toString());
182          Message message = Message.obtain(mHandler,
   R.id.encode_failed);
183          message.sendToTarget();
184  +     } finally {
185  +       if (client != null) {
186  +         client.close();
187  +       }
188        }
189      }
190    }
```

Resource Acquisition
resource leak caused by the unreleased *client*
Resource Reachability Validation
Rresource Release
Buggy Version
Fixed Version

Fig. 1. Comparison between Buggy Version (Left) and Fixed Version (Right) Associated with `AndroidHttpClient` Leak

be attributed to the detectors' limited capability of capturing the underlying common characteristics shared by these APIs, which essentially represent the same resource concept, namely, an *HTTP client*, and its usage. The presence of this limitation further emphasizes the necessity of incorporating background knowledge and code understanding in resource acquisition and release identification.

2. Identifying resource reachability validation more comprehensively. The validation of resource reachability is a crucial factor in determining the occurrence of resource leaks. Note that the meanings between *reachable/reachability* and the related terms (e.g., *open/closed/valid*) are different. *Open/closed/valid* describe the status of a resource itself; while *reachable/reachability* indicate whether the open status of a resource can be propagated to *a certain path, branch, or statement*. For example, in the fixed version in Figure 1, the `if`-condition at line 185 validates the *reachability* of the resource "HTTP client" by checking whether the object of client is null. As a result, the client resource is actually unreachable on the false branch of this `if`-condition, that is, the open status of the client resource cannot reach the false branch. If a resource leak detection technique ignores the *reachability* of the client resource, it would cause a false alarm in the false branch of this `if`-condition. Therefore, reachability is essential for reducing false alarms in static resource leak detection.

Existing detection techniques, such as Torlak et al. [3] and Cherem et al. [21], typically identify reachability validation by matching `null`-checking conditions. While this approach provides a simple and intuitive means to recognize the reachability validation at line 185, it may overlook certain validation intentions present in other code. For specific resource types, dedicated APIs, such as `isClosed()` and `isDisabled()`, are employed to perform reachability validation. In a similar manner to resource acquisition/release identification, a more comprehensive identification of such validation intentions necessitates the incorporation of background knowledge pertaining to common validation verbs, in conjunction with a deep understanding of the code contexts, such as the `if`-conditions. By considering these elements, resource reachability validation can be assessed more comprehensively, enabling a more accurate detection of resource leaks in code.

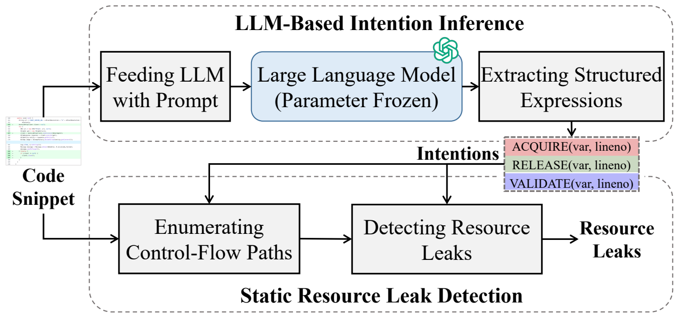
III. Approach

Based on these insights, we propose **InferROI**, a novel approach that directly **Infers R**esource-**O**riented **I**ntentions in code to boost static resource leak detection, by leveraging the power of large language models (LLMs). In the approach, LLMs function as extensive knowledge bases with exceptional code understanding capabilities [9]–[12], rendering them well-suited for inferring resource-oriented intentions.

Figure 2 represents the approach overview of InferROI. For a given code snippet, InferROI first instantiates the prompt template to instruct the LLM in inferring the involved resource acquisition, release, and reachability validation intentions, which does no require any prior information on resource API pairs. The intentions generated by LLMs (which are described in natural language) are then converted into structured expressions, so as to facilitate the subsequent static analysis stage. By aggregating these inferred intentions, InferROI then proceeds to apply a two-stage static analysis algorithm to analyze the control-flow paths extracted from the code. The algorithm first identifies potential leak-risky paths based on the inferred resource acquisition/release APIs, and then prunes leak-risky paths with unreachable resources based on the inferred reachability validation across these paths on sibling branches, so as to reduce false alarms of resource leak detection.

A. Resource-Oriented Intentions

We focus on three resource-oriented intentions involved in code, which play a crucial role in resource management.



LLM-Based Intention Inference
Feeding LLM with Prompt
Large Language Model (Parameter Frozen)
Extracting Structured Expressions
Code Snippet
Intentions
ACQUIRE(var, lineno)
RELEASE(var, lineno)
VALIDATE(var, lineno)
Enumerating Control-Flow Paths
Detecting Resource Leaks
Resource Leaks
Static Resource Leak Detection

Fig. 2. Approach Overview of INFERROI

Resource Acquisition. Statements acquire resources (e.g., locks or database connections), which are normally API invocations and involved with objects/variables of resources (e.g., `lock` or `db`). To structurally express the intentions of resource acquisition, we introduce the notation $\boldsymbol{ACQ(var, lineno)}$, where var denotes the variable storing the acquired resource, and $lineno$ indicates the line number containing the statement responsible for acquiring the resource.

Resource Release. Likewise, statements release previously acquired resources. We structurally represent the intentions of resource release as $\boldsymbol{REL(var, lineno)}$, where var corresponds to the variable storing the resource to be released, and $lineno$ indicates the line number of the statement responsible for performing the release operation.

Resource Reachability Validation. Conditional statements aim to validate whether acquired resources remain reachable within specific branches. To structurally represent the intentions of resource reachability validation, we employ the notation $\boldsymbol{VAL(var, lineno)}$, where var represents the variable storing the resource being validated, and $lineno$ indicates the line number of the statement responsible for performing the validation operation.

Example 3.1. In the method depicted in Figure 1, the intentions of the API calls at line 167, which aims to acquire a *HTTP client* resource, can be represented as $ACQ(\texttt{client}, 167)$. The intention of the API call at line 186, responsible for releasing the acquired *HTTP client* resource, can be denoted as $REL(\texttt{client}, 186)$. The intention of the `if` statement at line 185, intended to validate the reachability of the acquired *HTTP client* resource, can be expressed as $VAL(\texttt{client}, 185)$.

B. LLM-Based Intention Inference

To infer resource-oriented intentions present in code, we first prompt the large language model (LLM) with designed templates and then convert the LLM-generated answers into structured expressions for the subsequent analysis.

1) Feeding LLM with Prompt: For a given code, we construct a prompt using the template depicted in Figure 3. This prompt is then fed into the LLM (e.g., GPT-4) for intention inference. The prompt template is designed following the best practices for prompt engineering by OpenAI [22], consisting of three main components: Task Description & Instructions, Output Format Specification, and Code Placeholder.

The **Task Description & Instructions** component consists of several sequential instructions designed based on

Task Desction & Instructions
Analyze the information about resource leaks in the provided code snippet below. First, resolve the types of the involved objects. Then, identify the types representing leakable resources. Next, identify the API/method calls for acquiring the resources. After that, identify the API/method calls for releasing the acquired resources. Finally, identify the if-conditions for checking whether the acquired resources are closed or unclosed.

Output Format Specification
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

If-conditions for Checking Resources closed or not:
line <line number> `<if-condition>` checks `<resource variable>` resource

Code Placeholder
Code Snippet: ```java
{CODE}
```

Fig. 3. Prompt Template

Leakable Resources:
AndroidHttpClient: client

API/method Calls for Acquiring Resources:
line 167: `AndroidHttpClient.newInstance(mUserAgent)` acquires `client` resource

API/method Calls for Releasing Resources:
line 186: `client.close()` releases `client` resource

If-conditions for Checking Resources closed or not:
line 185: `if (client != null)` checks `client` resource

Fig. 4. Answer Generated by GPT-4 Model

the chain-of-thought (CoT) idea [23]. It begins with three leading instructions to introduce the overall task goal and guide the LLM in forming a smooth reasoning chain from a basic starting point, specifically by performing type inference for the involved objects and determining which types represent leakable resources. Based on the analysis, the LLM is then required to infer the three types of resource-oriented intentions by the follow-up three instructions. Throughout the analysis and inference process, the LLM leverages its extensive background knowledge to understand the code contexts and execute the instructions effectively.

**Output Format Specification** aims to specify the format of identified intentions, facilitating subsequent parsing.

**Code Placeholder** is where the given code with line numbers is inserted, creating a specific prompt.

***Example 3.2.*** Applying the template, we construct a prompt for the fixed version (right) illustrated in Figure 1. We then employ the GPT-4 model [24] to generate an answer, as shown in Figure 4. Notably, the GPT-4 model exhibits the capability to effectively analyze code semantics and discern resource-oriented intentions. Within the generated answer, resource acquisition intentions are highlighted in red, resource release intentions in green, and resource reachability validation intentions in cyan.
```

2) Extracting Structured Expressions: To extract structured expressions of the intentions identified by the LLM, we employ matching patterns that correspond to the output format specification specified in the template. Given an LLM-generated answer, we utilize regular expressions such as "line <line number>: <API call> acquires <resource variable> resource" to automatically extract the line number, API call, and resource variable involved in resource acquisition intentions. Similar regular expressions can be used to extract information related to resource release intentions and reachability validation intentions. We then use the extracted information to instantiate the structured expressions of the three types of intentions.

Example 3.3. From the answer illustrated in Figure 4, the three highlighted lines are matched by the matching patterns. Subsequently, the three structured intention expressions are extracted and instantiated as $ACQ(\texttt{client}, 167)$, $REL(\texttt{client}, 186)$, and $VAL(\texttt{client}, 185)$.

C. Static Resource Leak Detection

We then propose a lightweight static analysis based on the inferred resource-oriented intentions to detect resource leaks.

1) Enumerating Control-Flow Paths: Similar to most existing resource leak detection methods [3], [4], [6], [7], our detection approach is also built upon control-flow analysis. We construct control-flow graphs (CFGs) for the code to detect and extract pruned control-flow paths from these graphs. Within the CFG, nodes that generate multiple subsequent control-flow branches, such as `if`-statements and `switch`-statements, are referred to as `branch`-nodes. Next, we enumerate the paths within the CFG, considering the paths between the `entry`-node and `exit`-node. To ensure the efficiency of the enumeration process, we employ two pruning operations.

Loop Structures. Loop statements in the CFG can introduce circular paths, leading to infinite-length paths during the enumeration. To prevent this, we adopt a strategy that involves expanding the true-branch of a loop only once.

Resource-Independent Branches. The presence of branches in the CFG can exponentially increase the number of paths during enumeration, but most of the branches are not related to resource management. To address this, for a `branch`-node, we further prune its branches that are independent of resource acquisition/release and procedural exit (e.g, `return`-statements).

Example 3.4. For the buggy version in Figure 1, there is only one control-flow path that sequentially executes from line 160 to line 185. In the fixed version shown in Figure 1, the `if`-statement for reachability validation at line 185 introduces two control-flow paths, corresponding to the line ranges [160-185, 186, 187-190] and [160-185, 187-190].

2) Detecting Resource Leaks: After enumerating the control-flow paths and inferring the resource-oriented intentions, InferROI proceeds with resource leak detection.

For each concerned resource involved in the inferred intentions, the detection process involves a two-stage path analysis, as illustrated in Algorithm 1. This algorithm takes the concerned resource res, all the control-flow paths $\mathcal{P}$, and the resource-oriented intention set $\mathcal{I}$ as inputs. The first stage (lines 1-16) is

Algorithm 1 Leak Detection via Two-Stage Path Analysis

```
Input $res$: concerned resource; $\mathcal{P}$: paths; $\mathcal{I}$: intention set;
     ▷ Stage 1: Single-Path Analysis
 1: for $path \in \mathcal{P}$ do
 2:     $rd\_counter \leftarrow 0$;
 3:     for $node \in path$ do
 4:         $ln \leftarrow$ get line number of $node$;
 5:         if $ACQ(res, ln) \in \mathcal{I}$ then
 6:             $rd\_counter \leftarrow rd\_counter + 1$;
 7:         else if $REL(res, ln) \in \mathcal{I}$ then
 8:             $rd\_counter \leftarrow rd\_counter - 1$;
 9:         end if
10:     end for
11:     if $rd\_counter > 0$ then
12:         $path.risky \leftarrow$ true;
13:     else
14:         $path.risky \leftarrow$ false;
15:     end if
16: end for
     ▷ Stage 2: Cross-Path Analysis
17: $br\_nodes \leftarrow$ get all branch-nodes in the paths;
18: sort $br\_nodes$ by their line numbers in a descending order
19: for $br\_node \in br\_nodes$ do
20:     $br\_ln \leftarrow$ get line number of $br\_node$
21:     if not $VAL(res, br\_ln) \in \mathcal{I}$ then
22:         continue;
23:     end if
24:     $G_{prefix} \leftarrow$ group paths containing $br\_node$ by path prefixes;
25:     for $g \in G_{prefix}$ do
26:         $(B_1, B_2, ...) \leftarrow$ group paths in $g$ by branches of $br\_node$;
27:         PROPAGATE$(B_1, B_2, ...)$;
28:     end for
29: end for
     ▷ Reporting Leaks
30: for $path \in \mathcal{P}$ do
31:     if $path.risky$ then
32:         report a leak of $res$;
33:         break;
34:     end if
35: end for
 1: procedure PROPAGATE$(B_1, B_2, ...)$
 2:     if $(\forall p \in B_i \mapsto p.risky)$ and $(\nexists p \in \bigcup_{j \neq i} B_j \mapsto p.risky)$ then
 3:         foreach $p \in B_i : p.risky \leftarrow$ false;
 4:     end if
 5: end procedure
```

a *single-path analysis* that initially identifies leak-risky paths of the concerned resource based on the ACQ and REL intentions. The second stage (lines 17-29) is a *cross-path analysis* that greedily eliminates the false-alarm-introducing `branch`-nodes by considering the VAL intentions. After completing both stages, the algorithm (lines 31-36) checks if any path has a corresponding status of $true$. If such a path exists, a resource leak of res is reported.

Stage 1: Single-Path Analysis. The algorithm conducts an iterative process, sequentially processing each $path$ in $\mathcal{P}$ and traversing the nodes within $path$ (lines 2-17). During the traversal, a descriptor counter $rd\_counter$ keeps track of the ACQ and REL intentions involving res within $path$ (lines 3-11). If $rd\_counter$ is greater than 0, it indicates that $path$ is a potential leak-risky path for res, resulting in the risky status of $path$ (i.e., $path.risky$) is initialized as $true$ (lines 12-16).

When no released API is identified, there is no REL item in the inferred intention expression. For example, when there is an ACQ in a path but no corresponding REL, the rd_counter will be 1, marking the path as risky (Line 12).

Stage 2: Cross-Path Analysis. As outlined in Section II, the presence of branches, denoted as unreachable branches, where the resource is inaccessible, gives rise to the possibility of false-positive (FP) leak-risky paths. This, in turn, results in false alarms being reported at line 32. During this phase, we conduct cross-path analysis utilizing the inferred reachability validation intentions to eliminate the false alarms.

Intuitively, addressing this problem involves specifically analyzing each `branch`-node intended for reachability validation and identifying its unreachable branch. However, the task is complicated by the diverse implementations of validation APIs, as discussed in Section II. Some APIs (e.g., `isClosed()`) return true to indicate that the resources are reachable, while others (e.g., `isActive()`) indicate the opposite. To overcome this challenge, our approach utilizes a novel heuristic that collectively analyzes all branches of a `branch`-node for reachability validation to assess whether it introduces false alarms. If affirmative, such `branch`-nodes are designated as *false-alarm-introducing* (FAI) nodes. These FAI nodes are characterized by its one branch being *completely* leak-risky, while the others are *completely* not leak-risky. A branch is considered *completely* leak-risky when all the paths belonging to it exhibit a risky status of $true$. The completely leak-risky branch is the unreachable branch, and the paths within it are FP leak-risky paths for res. The insight behind this decision is that if the other branch (the completely not leak-risky branch) is unreachable, it suggests the presence of more severe functional bugs rather than resource leaks, such as *null pointer errors*.

To illustrate, consider the if-condition node (line 185) in the Fixed Version in Figure 1. It validates the `client` resource and then releases it. When one branch is completely leak-risky (marked in Stage 1 due to not containing `client.close()`) and the other is completely not leak-risky (marked in Stage 1 due to containing `client.close()`), we can essentially determine that this if-condition is a FAI node and its completely leak-risky branch is the unreachable branch. Otherwise, if `client.close()` is called in an unreachable branch (e.g., `if(client==null){client.close();}`), it would result in a more severe and obvious null-pointer exception.

To implement this heuristic, our algorithm initiates by identifying all `branch`-nodes in the CFG and iterates through them from back to front (lines 18-30). During this process, each `branch`-node br_node is examined to determine if it involves a VAL intention. If not, the algorithm skips br_node, recognizing that it does not influence the risky status of the paths (lines 21-24). On the contrary, when br_node involves a VAL intention, the algorithm conducts two grouping operations (line 24 and line 26, respectively). This results in the division of paths containing br_node into branches $(B_1, B_2, ...)$. All these branches share a common path prefix (a subpath from the `entry`-node to br_node) but belong to different branches of br_node. This implies that, at a specific execution path to br_node, $(B_1, B_2, ...)$ represent potential subsequent branches of br_node. Upon the branches, the algorithm performs the heuristic by executing the PROPAGATE procedure and updating the risky status of FP leak-risky paths to $false$ (line 28).

Example 3.5. In the buggy version depicted in Figure 1, the detection algorithm successfully identifies a resource leak of the variable `client`. In contrast, for the fixed version shown in Figure 1, the algorithm identifies a leak-risky path [160-185, 187-190] of the variable `client` in Stage 1. However, the UPDATE procedure is then executed in Stage 2, which effectively eliminates this path as an FAI path. Consequently, no resource leak of `client` is reported in this version, indicating that the detection algorithm correctly avoids false alarms and provides a more accurate analysis of resource leaks.

IV. IMPLEMENTATION

The current implementation of INFERROI utilizes the widely adopted GPT-4 as the LLM and sets the model temperature as 0 to remove generation randomness. For RQ4, we use *Meta-Llama-3-8B-Instruct* [25] and *gemma-2-9b-it* [26] versions available on the Hugging Face Platform, and run the model inference with the Transformers library [27]. To align with the temperature setting (i.e., temperature=0) of GPT-4, we use greedy search decoding for both models, ensuring there is no randomness. The experiments are run on a workstation with 4 AMD EPYC 9554 64-Core CPUs, 512 GB of memory, and a single Nvidia H100 (80GB) GPU. For the CFG construction in static resource leak detection, given a method-level code snippet, we begin by parsing it into an abstract syntax tree (AST) and then proceed to construct a CFG by traversing the AST nodes and inserting various control-flow edges between the code statements. The procedure is implemented based on the top of an established CFG construction tool [28]. Since some resources are intentionally kept open and returned for further use, we adopt a rule to exclude the leak reports where the inferred resource objects returned in `return`-statements, thereby reducing false positives. In addition, we take into account certain Java-specific language features, such as `try-with-resources` statements. For the `try-with-resources` statements, the acquired resources are automatically closed, and there is no need for manual resource closure [29]. To address this, we incorporate a rule-based post-processing step to filter out false alarms caused by `try-with-resources` statements. Specifically, for each reported resource leak, we traverse the AST of the examined method to determine if the leaked resource is acquired within a `try-with-resources` statement. If so, the reported leak is omitted.

V. EVALUATION

We conduct comprehensive experiments to evaluate the effectiveness of INFERROI in detecting resource leaks by employing the DroidLeaks dataset [8] and the JLeaks dataset [13].

First, we apply INFERROI to detect resource leaks in the code snippets from DroidLeaks and JLeaks datasets, comparing its effectiveness against baseline detectors (**RQ1**). Next, we evaluate the effectiveness of INFERROI in detecting previously unknown resource leaks in real-world open-source projects (**RQ2**). We then conduct an ablation study to evaluate the contributions of intention inference and static analysis in INFERROI (**RQ3**). Subsequently, we investigate the generalizability of integrating INFERROI with different LLMs and compare their effectiveness (**RQ4**). Finally, we manually annotate the resource-oriented intentions in code to measure how effectively INFERROI infers these intentions (**RQ5**).

All the research questions are listed as follows.

- **RQ1 (Effectiveness in Leak Detection)**: How effectively can INFERROI detect resource leaks in existing benchmarks?
- **RQ2 (Effectiveness in Project Scanning)**: How effective is INFERROI in detecting previously unknown resource leak bugs when applied to open-source projects?
- **RQ3 (Ablation Study)**: What are the contributions of intention inference and static analysis in INFERROI?
- **RQ4 (Generalizability with Different LLMs)**: How generalizable is INFERROI when integrated with different LLMs?
- **RQ5 (Accuracy in Intention Inference)**: How accurately can INFERROI infer resource-oriented intentions in code?

A. Effectiveness in Resource Leak Detection (RQ1)

In this evaluation, we assess the effectiveness of INFERROI with GPT-4 in detecting resource leaks using two datasets.

1) Data: We utilize two datasets, namely **DroidLeaks** [8] and **JLeaks** [13], constructed by analyzing commits dedicated to fixing resource leak bugs. Each bug in the datasets is associated with a concerned resource type, which is determined by reviewing the corresponding bug-fixing commit. By retrieving the bug-fixing commit and the previous commit, both the buggy and fixed versions are obtained for each bug.

- **DroidLeaks**: This dataset comprises resource leak bugs sourced from open-source Android applications. Originally, the DroidLeaks dataset included 116 bugs used to assess the performance of eight widely-used resource leak detectors [6], [7], [15]–[19]. In our evaluation, we attempt to acquire all 116 bugs, successfully obtaining both the buggy and fixed versions for 86 of them. Consequently, the final collection comprises 172 code snippets covering 28 different resource types, such as `Cursor`.
- **JLeaks**: This is a more recent dataset containing resource leak bugs from open-source Java projects. The original JLeaks dataset included 1,094 bugs used to evaluate the performance of three widely-used resource leak detectors [6], [7], [30]. For our evaluation, we filtered out instances based on two conditions: 1) the resource type is annotated, and 2) both the buggy and fixed versions are free of syntax errors. This filtering process resulted in 784 bugs (1,568 code snippets) encompassing 368 different resource types.

2) Baselines: In RQ1, we compare INFERROI with three well-maintained industrial static detectors, as they have been evaluated in both DroidLeaks and JLeaks papers [8], [13].

- **SpotBugs** [6]: SpotBugs, formerly known as FindBugs, is a static analysis tool that identifies bugs and potential issues in Java programs.

TABLE I
RESOURCE LEAK DETECTION ON DROIDLEAKS DATASET

Detector	#TP	#FP	BDR ↑	FAR ↓	F1 ↑
SpotBugs	6	0	6.9%	0%	0.129
Infer	37	16	43.0%	18.6%	0.532
PMD	10	6	11.6%	7.0%	0.196
INFERROI	51	16	59.3%	18.6%	0.667

TABLE II
RESOURCE LEAK DETECTION ON JLEAKS DATASET

Detector	#TP	#FP	BDR ↑	FAR ↓	F1 ↑
SpotBugs*	44	—	24.3%	—	—
Infer*	5	—	9.4%	—	—
PMD	341	40	43.5%	5.1%	0.585
INFERROI	490	153	62.5%	19.5%	0.687

* Due to encountering compilation issues, SpotBugs and Infer are able to process only 181 and 53 instances, respectively.

- **Infer** [7]: Infer is a static analysis tool developed by Facebook that detects various types of programming errors in C, C++, and Java code.
- **PMD** [30]: PMD is a widely-used static code analysis tool that identifies potential issues in code.

3) Procedure: For each bug in our experiments, we apply INFERROI to both its buggy version and its fixed version. If INFERROI reports a resource leak in the buggy version related to the concerned resource type, it is considered a *true positive* (TP); otherwise, there is a *false negative* (FN). Conversely, if INFERROI reports a resource leak in the fixed version for the corresponding resource type, it is a *false positive* (FP). Based on this, we count the numbers of TPs, FPs, and FNs, and calculate *Bug Detection Rate* (BDR) and *False Alarm Rate* (FAR) following a previous study [8].

$$BDR = \frac{\text{\# TP}}{\text{\# Buggy Versions}}, \quad FAR = \frac{\text{\#FP}}{\text{\# Fixed Versions}}$$

Additionally, to better reflect the overall effectiveness of INFERROI, we calculate the $F1$ score based on *Precision* and *Recall*:

$$F1 = \frac{2 \cdot \mathit{Precision} \cdot \mathit{Recall}}{\mathit{Precision} + \mathit{Recall}},$$

where *Precision* and *Recall* are calculated by $^{\#\text{TP}}/_{(\#\text{TP} + \#\text{FP})}$ and $^{\#\text{TP}}/_{(\#\text{TP} + \#\text{FN})}$, respectively. Note that we do not report *Precision* and *Recall* separately in the experimental results, as their implications are effectively captured by *Bug Detection Rate* and *False Alarm Rate*.

4) Results: The evaluation results for DroidLeaks and JLeaks are presented in Table I and Table II, respectively.

DroidLeaks: INFERROI demonstrates a bug detection rate of 59.3% with a false alarm rate of 18.6% across 86 bugs in DroidLeaks. Compared to the baseline detectors, INFERROI achieves notably higher bug detection, detecting 14~45 more bugs, while maintaining a comparable false alarm rate. Considering the overall effectiveness, INFERROI achieves an $F1$ score of 0.667, significantly outperforming the baselines by 25.4% to 417.1%.

JLeaks: INFERROI shows a bug detection rate of 62.5% with a false alarm rate of 19.5% across 784 bugs in JLeaks.

Compared to the baseline detectors, INFERROI achieves significantly superior bug detection, identifying 149~485 more bugs in JLeaks. As JLeaks does not release complete compilable project-level environments, we cannot run the baselines SpotBugs and INFER (which both require compiled projects) and Table II presents their bug detection rate as reported in JLeaks paper. For the baseline PMD which does not require compiled projects, we run PMD on JLeaks and report both its false alarm rate and bug detection rate. As shown in Table II, although PMD achieves lower false alarm rate than INFERROI, INFERROI achieves much better trade-off between bug detection rate and false alarm rate than PMD, i.e., INFERROI achieves an F1 score of 0.687, outperforming PMD by 17.4%.

Analysis: (i) *Bug Detection Analysis*: The higher bug detection rate of INFERROI can be attributed to the advantages of the resource-oriented intention inference utilized by INFERROI, as discussed in the insights provided in Section II (see RQ3.a for the resource coverage assessment). However, it is essential to acknowledge that the resource-oriented intention inference cannot entirely replace the value of more sound program analysis techniques. For instance, in the case of buggy versions not detected by INFERROI, a substantial portion is caused by the intricacies of Android's lifecycle management and callback mechanisms. This observation indicates an important future direction, i.e., to explore better integration of LLM-based resource-intention inference with more advanced program analysis techniques, which could enable more general and powerful resource leak detection. (ii) *False Alarm Analysis*: False alarms reported by INFERROI primarily stem from incorrect or missing intentions. When INFERROI incorrectly identifies an acquisition intention or overlooks a release intention, a false alarm occurs. To address this issue, we intend to explore the integration of LLM-based prompt engineering techniques, such as in-context learning, to enhance the capabilities of LLMs. Notably, a significant portion of these "false alarms" arise from incorrect ground truths. Since both datasets are curated based on bug-fixing commits, certain "fixed" versions still contain resource leaks. This underscores the ongoing challenge of ensuring the quality of resource leak datasets.

Summary: INFERROI demonstrates a bug detection rate of 59.3% with a false alarm rate of 18.6% across 86 bugs in DroidLeaks, and a bug detection rate of 62.5% with a false alarm rate of 19.5% across 784 bugs in JLeaks. Considering overall effectiveness, INFERROI significantly outperforms the baselines on both datasets, with $F1$ scores of 0.667 and 0.687. This shows a promising potential of INFERROI in detecting resource leaks.

B. *Effectiveness in Open-Source Project Scanning (RQ2)*

We apply INFERROI with GPT-4 on real-world Java projects to evaluate its efficacy in detecting previously-unknown resource leaks.

1) Data: To save time and reduce the cost of querying the GPT-4 API, we employ two strategies to sample two set of methods from open-source projects.

TABLE III
RESULTS OF OPEN SOURCE SCANNING

Detector	100 Suspicious Methods		100 Lucene Methods	
	#TP	#FP	#TP	#FP
INFERROI	26 (*7 Accepted PRs*)	3	3	0
PMD	18	2	0	0

Suspicious Methods Filtered by Matching Resources. We crawl 115 Java open-source projects with more than 50 stars, which are created after December 31, 2021. We randomly filter 100 suspicious methods from 13 projects, after matching 20 common resource terms reported in previous work [31], instead of completely scanning these projects. The employed terms are listed as follows: *stream*, *reader*, *client*, *writer*, *lock*, *player*, *connection*, *monitor*, *gzip*, *ftp*, *semaphore*, *mutex*, *camera*, *jar*, *buffer*, *latch*, *socket*, *database*, *scanner*, *cursor*.

Random Methods Sampled from Apache Lucene. To further evaluate the effectiveness of INFERROI in detecting resource leaks in the scenarios without relying on matching common resource terms, we also randomly sample 100 methods from the latest version of *Apache Lucene* [32], a widely used text retrieval engine.

Interprocedural Scanning with CodeQL on Apache Lucene. To investigate the potential of leveraging inferred intentions to enhance interprocedural scanning, we integrate the *Resource Acquisition* and *Resource Release* intentions inferred by INFERROI for Apache Lucene into CodeQL QL scripts. This integration is straightforward, as CodeQL supports the addition of custom resource types and their associated acquisition and release APIs. However, we have not yet incorporated *Resource Reachability Validation*, as modifying CodeQL's flow analysis to support it presents significantly greater complexity.

2) Procedure: We apply INFERROI on these 200 methods to detect resource leaks. For each reported leak, we first manually annotate whether it is a true bug by reading code and consulting related information. For the true bugs, we then submit the corresponding bug-fix pull requests and help the project developers review them. We run PMD on these methods and manually check the reported bugs. In RQ2, we use PMD instead of SpotBugs and Infer as our baseline, because (i) PMD shows the best detection rate on the large-scale benchmark JLeaks in RQ1, and (ii) PMD does not require compiled projects (while each instance in RQ2 is a method snippet).

3) Results: In the 100 suspicious methods filtered by matching resources, INFERROI reports 29 resource leaks and 26 are annotated as true bugs. At the submission time, 7 of the true bugs have been confirmed by the project developers, and the pull requests have been accepted. PMD reports 20 resource leaks in the 100 methods, with 18 of them being true bugs. Among these 18 bugs identified by PMD, INFERROI successfully detects 17 of them. The lone bug missed by INFERROI involves a method comprising over 400 lines of code, posing challenges for LLM processing. Conversely, INFERROI identifies 9 additional true bugs not caught by PMD. Noteworthy among these are resource types such as `ManagedBuffer` from the *Apache Spark* library, `InMemoryDi-`

`rectoryServer` from *UnboundID LDAP*, and a custom type `JDBCConnection` specific to the project.

In the 100 methods sampled from Apache Lucene, INFERROI reports 3 resource leaks, all of which are confirmed as true bugs, while PMD reports zero bugs. The 3 bugs detected by INFERROI involve project-defined resource classes `IndexReader`, `DirectoryReader`, and `DirectoryTaxonomyReader`. These resources are acquired and released by calling the `incRef` and `decRef` methods, respectively, which are challenging for static detectors to address.

For the interprocedural scanning, CodeQL reports 34 leaks before integrating INFERROI-inferred intentions and 49 leaks after integration. Among these, 29 (85.3%) and 41 (83.7%) are confirmed as true bugs, respectively.

Summary: INFERROI successfully detects 26 previously unknown bugs in 100 suspicious methods and 3 previously unknown bugs in 100 Lucene methods. The results demonstrate the practical capability of INFERROI in effectively detecting resource leaks in real-world software projects. Integrating INFERROI-inferred intentions enables CodeQL to identify 12 additional true bugs during interprocedural scanning on Apache Lucene.

C. Ablation Study

We compare INFERROI with three end-to-end GPT-based detectors that rely solely on prompting engineering to evaluate the contribution of static analysis.

1) Procedure: The three end-to-end GPT-based detectors without static analysis are denoted as GPTLEAK variants (The used prompts can be found in our replication package [33]).

- **GPTLEAK** employs a straightforward instruction to directly identify resource leaks in the provided code.
- **GPTLEAK-*exp*** employs an additional chain-of-thought (CoT) instruction "*First, explain the behavior of the code*" to explain code behavior, which have been proven effective in static bug detection in previous study [34].
- **GPTLEAK-*roi*** employs additional CoT instructions to infer resource-oriented intentions (ROIs) in code, similar to INFERROI, before identifying resource leaks.

We apply the three detectors to the 172 code snippets collected from the DroidLeaks dataset and calculate the BDR, FAR, and $F1$ score for each detector.

2) Results: As shown in Table IV, INFERROI outperforms three GPTLEAK variants (including two CoT variants) in overall detection effectiveness, achieving a higher F1 score with comparable BDR and much lower FAR. In particular, the two CoT variants outperform the straightforward GPTLEAK, indicating that the suitable design of CoT can help LLMs better identify bugs. In addition, INFERROI further outperforms the two CoT variants, indicating the benefits of combining LLMs and static analysis, which is also the main insight of INFERROI. More detailed analyses are presented below.

Comparison among CoT variants and the basic GPTLEAK. Compared to the straightforward GPTLEAK, the two CoT strategies in GPTLEAK-*exp* and GPTLEAK-*roi*

TABLE IV
ABLATION STUDY RESULTS

Detector	#TP	#FP	BDR ↑	FAR ↓	F1 ↑
GPTLEAK	57	58	66.3%	67.4%	0.567
GPTLEAK-*exp*	52	37	60.5%	43.0%	0.594
GPTLEAK-*roi*	49	26	57.0%	30.2%	0.609
INFERROI	51	16	59.3%	18.6%	0.667

show improvements in reducing false alarms while slightly decreasing the bug detection rate. For example, GPTLEAK-*roi* reduces more than half of the false alarms compared to GPTLEAK by employing CoT instructions to first inferring resource-oriented intentions (i.e., the same as INFERROI). It actually confirms that our proposed intention inference is also beneficial for purely LLM-based resource leak detection. Overall, incorporating suitable CoT prompt is beneficial when applying LLMs for resource leak detection.

Comparison between INFERROI and CoT variants. Although the two CoT variants (i.e., GPTLEAK-*exp* and GPTLEAK-*roi*) outperform the basic GPTLEAK, INFERROI further outperforms the two CoT variants in overall detection effectiveness. In particular, the static analysis can equip LLMs with more rigorous analysis, which can help reduce false positives in end-to-end LLM-based detection; while LLMs are also complementary to the static analysis as LLMs exhibit more general capabilities in code comprehension. As a result, by combining static analysis and LLMs, INFERROI can achieve a balanced and effective resource leak detection mechanism, ultimately enhancing its overall accuracy and reliability.

Summary: The results of the ablation study confirm the contributions of both LLM-based resource-oriented intention inference and static resource leak detection in INFERROI.

D. Generalizability with Different LLMs (RQ4)

We conduct an experiment to investigate the generalizability of INFERROI when integrated with different LLMs.

1) Selection of LLMs: In addition to the advanced closed-source GPT-4, we explore the effectiveness and efficiency of INFERROI when integrated with the following two state-of-the-art open-source LLMs.

- **Llama3-8B** [35]: Meta's Llama 3 is a family of large language models available in various sizes, featuring both pretrained and instruction-tuned generative text models. We use Llama3-8B, which has 8 billion parameters.
- **Gemma2-9B** [36]: Google's Gemma 2 models are lightweight, instruction-tuned large language models, built using the same technology as the Gemini models [37]. We employ the Gemma2-9B version, which has 9 billion parameters.

2) Results: Table V presents the comparison in the effectiveness and efficiency of INFERROI when integrated with different LLMs. Overall, INFERROI with different LLMs consistently outperforms existing baselines (i.e., SpotBugs, Infer, and PMD) in F1 score. The results show that open-source Gemma2-9B achieves comparable effectiveness (0.644 vs. 0.667 in $F1$ score) with GPT-4, while Llama3-8B shows relatively lower metrics.

TABLE V
RESULTS OF DIFFERENT LLMS ON DROIDLEAKS DATASET

LLM	#TP	#FP	BDR ↑	FAR ↓	F1 ↑	Time
Llama3-8B	39	18	45.3%	20.9%	0.545	7.1s
Gemma2-9B	48	15	55.8%	17.4%	0.644	11.4s
GPT-4	51	16	59.3%	18.6%	0.667	8.7s

The two smaller open-source models, Llama3-8B and Gemma2-9B, have longer response times than the larger GPT-4 due to the lack of optimizations like batch processing.

Summary: INFERROI demonstrates substantial generalizability when integrated with different LLMs, including closed-source GPT-4 and open-source LLMs, maintaining relatively consistent effectiveness in resource leak detection.

E. Accuracy in Resource-Oriented Intention Inference (RQ5)

In this evaluation, we evaluate the effectiveness of INFERROI's LLM-based resource-oriented intention inference.

1) Procedure: To evaluate the effectiveness of INFERROI's intention inference, two authors independently annotate all three types of resource-oriented intentions present in the 172 code snippets collected from the DroidLeaks dataset. In cases of disagreement, they engage in discussions to reach a final decision, ensuring consistent and accurate annotations. These manual annotations serve as the ground truth intentions (GTs) for the evaluation. Subsequently, we apply INFERROI to the code snippets to infer the intentions (denoted as $Preds$), and then calculate the *precision* and *recall* of the inference based on the manual annotations as follows:

$$precision = \frac{|GTs \cap Preds|}{|Preds|}; recall = \frac{|GTs \cap Preds|}{|GTs|}$$

We compare the intention inference of INFERROI with MiROK [31], which is a state-of-the-art technique for mining RAR (Resource Acquisition and Release) API pairs. Specifically, we use the abstract RARs reported by MiROK to match APIs in the code for identifying intentions and then calculate precision and recall using the aforementioned equations. Note that MiROK does not support VAL intentions, so its metrics are calculated only for ACQ and REL intentions. Furthermore, we assess the coverage of INFERROI-inferred intentions for the 28 resource types in DroidLeaks and the 368 resource types in JLeaks, and then compare this coverage with that of PMD.

2) Results: INFERROI achieves a precision of 74.6% and a recall of 81.8% in resource-oriented intention inference, indicating its ability to relatively accurately and comprehensively identify intentions within the code snippets. In contrast, intention identification based on MiROK yields only 54.0% precision and 42.3% recall, which is significantly lower than INFERROI. Moreover, INFERROI demonstrates superior versatility in addressing a broader spectrum of resource types, attaining a coverage of 67.9% (19 types) for DroidLeaks and 60.1% (221 types) for JLeaks, compared to PMD's coverage of 32.1% (9 types) and 12.8% (47 types), respectively.

Through analysis of the evaluation results, we attribute the effectiveness of INFERROI's intention inference to the powerful capabilities of LLMs in leveraging background knowledge and understanding code context. For example, INFERROI can identify the `AndroidHttpClient`-related intentions, which are ignored by other baseline detectors.

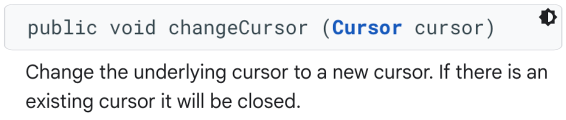

```
public void changeCursor (Cursor cursor)
```
Change the underlying cursor to a new cursor. If there is an existing cursor it will be closed.

Fig. 5. API Description of `CursorAdapter.changeCursor(Cursor)`

Summary: INFERROI exhibits a precision of 74.6% and a recall of 81.8% in inferring resource-oriented intentions, achieving resource type coverages of 67.9% and 60.1% for DroidLeaks and JLeaks, respectively. This achievement can be attributed to the capabilities of LLMs, coupled with the efficacy of our designed prompt template.

VI. DISCUSSION

A. Limitations

Intention Inference. Some reported false alarms arise because INFERROI fails to recognize certain resource release intentions. For example, five false alarms in DroidLeaks result from INFERROI not inferring that the `Cursor` resource is released when calling the Android method `CursorAdapter.changeCursor(Cursor)`. This oversight may be due to the API name `changeCursor` not explicitly indicating its resource release functionality, causing INFERROI to overlook it. In fact, as shown in Figure 5, the Android documentation specifies that `changeCursor(Cursor)` releases the existing `Cursor` resource. However, it appears that the LLM does not retain this information. To address this issue, future work can explore integrating retrieval-augmented generation (RAG) techniques to search relevant API documentation and achieve more accurate and comprehensive intention inference.

Static Analysis. It is important to acknowledge the limitations in the employed static analysis, especially in scenarios where certain types of resource leaks might remain undetected. For instance, resource leaks occurring within Android callbacks represent a challenging area where our lightweight static analysis may fall short. Our key insights lie in directly inferring resource-oriented intentions based on code context understanding to boost static leak detection. The advantage of inferring resource-oriented intentions is its potential applicability to various static leak detection techniques, including callback-aware analyses like Relda2 [17]. By integrating the inferred resource-oriented intentions into such existing techniques, we can enhance their capability to handle more comprehensive cases. In this regard, our LLM-based resource-oriented intention inference can provide *complementary insights and improvement* for existing static resource leak detection, with the potential to empower a wide range of analysis methods to achieve more comprehensive and accurate results.

B. Threats to Validity

The internal validity of our studies is potentially affected by the randomness in data sampling and the subjectiveness in data annotation. To mitigate these threats, we adopted commonly-used data sampling strategies and involved multiple annotators to minimize any preference bias. Another concern is the risk of data leakage, as the evaluation data might have been incorporated into the training data of the utilized LLMs. To the best of our knowledge, there is no existing resource leak benchmark that is guaranteed to be exempted from data leakage issues (i.e., only containing data after GPT-4 training data cutoff). Therefore, to mitigate this issue, we have made the following attempts to justify that the improvements of INFERROI are not simply caused by the potential data memorization. (i) We evaluate InferROI to find 26 previously-unknown bugs when scanning the open-source projects in RQ2. Even if the code itself could be seen by LLMs during its training, the previously-unknown bugs are not seen by LLMs before. (ii) We show INFERROI outperforms basic GPT-4 in the ablation study (RQ3.b). It indicates that the improvements of INFERROI do not simply come from the potential memorization of GPT-4 on similar examples from training. (iii) We include multiple datasets in RQ1 to address the overfitting issues. In addition to the classic benchmark DroidLeak, we also include the latest resource leak benchmark JLeaks which has shown to have better diversity and quality than previous benchmarks. As for the external validity, the evaluation data used in our work might not guarantee the full generality of our findings. To address this concern, we evaluate INFERROI on both existing datasets and real-world open-source projects to detect resource leaks.

VII. Related Work

Resource Leak Detection. Automated resource leak detection techniques [3]–[5], [8] have been proposed to detect whether some resource is not being released after its acquisition. Typically there are two important components for resource leak detection. First, identify the potential API pairs for resource acquisition and resource release; then based on the RAR pairs, analyze the code to check whether the release API is not subsequently called after the acquisition API. The majority of existing resource leak detection techniques are concentrated on the analysis part by proposing more precise and more scalable code analysis approaches [3]–[5], [38]–[40]. For example, Torlak et al. [3] combine intra-procedural analysis and inter-procedural analysis to enable more scalable and more accurate detection for system resource leaks (e.g., I/O stream and database connections); Wu et al. [4] propose an inter-procedural and callback-aware static analysis approach to detect resource leak in Android apps; Kellogg et al. [5] incorporate ownership transfer analysis, resource alias analysis, and fresh obligation creation to enable more precise analysis. The API pairs used in most existing techniques [3], [4] are often predefined by human expertise and heuristic rules, which not only require non-trivial human efforts but also have limited coverage in libraries and APIs. For example, Torlak et al. [3] manually collect API pairs and detect resource leaks for *stream* and *database* in JDK. In addition, SpotBugs [6] only considers the predefined *stream* related API pairs and thus could only detect *stream* related resource leaks. More recently, Bian et al. [41] propose SinkFinder, which mines and classifies frequent resource-related API pairs for a given project.

Our work does not rely on prepared API pairs and directly infers resource-oriented intentions in code by utilizing the exceptional code comprehension capability of LLMs.

LLMs for Software Engineering. LLMs have been widely used in software engineering tasks [42]–[53]. Recent studies have demonstrated that pre-trained language models can be utilized as knowledge bases by using synthetic tasks similar to the pre-training objective to retrieve the knowledge/information stored in the models [54], [55]. These works have shown that language models can recall factual knowledge without any fine-tuning by using proper prompts. Several works have focused on exploring effective prompt templates to improve performance of PLMs on downstream problems. In a recent survey by Liu et al. [56], prompt templates are broadly classified into two categories: *cloze prompts* [54], [57], [58], which entail filling in the blanks in text or code, and *prefix prompts* [59], [60], which continue generating content following a specified prefix. Recent studies have explored the application of prompt learning in various software engineering tasks. Wang et al. [61] investigated the effectiveness of prompt learning in code intelligence tasks such as clone detection and code summarization. Huang et al. [62] used prompt learning for type inference.

This works consider LLMs as knowledge bases of resource management and prompts them to infer resource-oriented intentions based on their powerful code understanding capability.

VIII. Conclusions and Future Work

In this work, we propose INFERROI, a novel resource leak detection approach which boosts static analysis via LLM-based resource-oriented intention inference. Given a code snippet, INFERROI prompts the LLM in inferring involved intentions from the code. By aggregating these inferred intentions, INFERROI utilizes a lightweight static-analysis based algorithm to analyze control-flow paths extracted from the code, thereby detecting resource leaks. We evaluate INFERROI for the Java programming language and investigate its effectiveness in resource leak detection. Experimental results on the DroidLeaks and JLeaks datasets demonstrate that INFERROI exhibits promising bug detection rate (about 60%) and false alarm rate (about 20%). Moreover, when applied to open-source projects, INFERROI identifies 29 unknown resource leak bugs. In the future, we will explore the extension of INFERROI to other programming languages and integrate INFERROI with more program analysis techniques.

Data Availability

To facilitate further research, we have released our code and data in the replication package [33].

Acknowledgement

This research project is supported by National Key R&D Program of China (2023YFB4503805) and National Natural Science Foundation of China under Grant No. 62302099.

References

[1] M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak, "Memory and resource leak defects and their repairs in java projects," *Empirical Software Engineering*, vol. 25, no. 1, pp. 678–718, 2020.

[2] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow," in *Proceedings of 40th IEEE/ACM International Conference on Software Engineering*, 2018, pp. 886–896.

[3] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 535–544.

[4] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Light-weight, inter-procedural and callback-aware resource leak detection for android apps," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1054–1076, 2016.

[5] M. Kellogg, N. Shadab, M. Sridharan, and M. 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*, 2021, pp. 181–192.

[6] (2023) Spotbugs. [Online]. Available: https://spotbugs.github.io/

[7] (2023) Infer. [Online]. Available: https://fbinfer.com/

[8] Y. Liu, J. Wang, L. Wei, C. Xu, S.-C. Cheung, T. Wu, J. Yan, and J. Zhang, "Droidleaks: a comprehensive database of resource leaks in android apps," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3435–3483, 2019.

[9] OpenAI, "Gpt-4 technical report," 2023.

[10] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[11] F. Petroni, T. Rocktäschel, P. Lewis, A. Bakhtin, Y. Wu, A. H. Miller, and S. Riedel, "Language models as knowledge bases?" 2019, pp. 2463–2473.

[12] Q. Huang, Z. Yuan, Z. Xing, X. Xu, L. Zhu, and Q. Lu, "Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[13] T. Liu, W. Ji, X. Dong, W. Yao, Y. Wang, H. Liu, H. Peng, and Y. Wang, "Jleaks: A featured resource leak repository collected from hundreds of open-source java projects," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2024, pp. 924–924. [Online]. Available: https://doi.ieeecomputersociety.org/

[14] (2023) Code diff. [Online]. Available: https://github.com/zxing/zxing/commit/de83fdf

[15] (2023) Code inspection. [Online]. Available: https://www.jetbrains.com/help/idea/2016.3/code-inspection.html

[16] (2023) Lint. [Online]. Available: https://developer.android.com/studio/write/lint

[17] T. Wu, J. Liu, X. Deng, J. Yan, and J. 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*, 2016, pp. 762–767.

[18] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 396–409.

[19] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, "Towards verifying android apps for the absence of {No-Sleep} energy bugs," in *2012 Workshop on Power-Aware Computing and Systems (HotPower 12)*, 2012.

[20] (2023) selenium library. [Online]. Available: https://github.com/SeleniumHQ/selenium

[21] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 480–491.

[22] (2023) Best practices for prompt engineering with openai api. [Online]. Available: https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api

[23] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[24] (2023) Gpt-4. [Online]. Available: https://platform.openai.com/docs/models/gpt-4

[25] (2024) Meta-llama-3-8b-instruct. [Online]. Available: https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct

[26] (2024) gemma-2-9b-it. [Online]. Available: https://huggingface.co/google/gemma-2-9b-it

[27] (2024) Transformers. [Online]. Available: https://github.com/huggingface/transformers

[28] (2023) Progex (program graph extractor). [Online]. Available: https://github.com/ghaffarian/progex

[29] (2023) The try-with-resoures statement. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

[30] (2023) Pmd. [Online]. Available: https://pmd.github.io/

[31] C. Wang, Y. Lou, X. Peng, J. Liu, and B. 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*, 2023, pp. 986–998.

[32] (2024) Apache lucene - commit cc3b412. [Online]. Available: https://github.com/apache/lucene/tree/cc3b412

[33] (2023) Replication package. [Online]. Available: https://github.com/cs-wangchong/InferROI-Replication

[34] C. Wen, Y. Cai, B. Zhang, J. Su, Z. Xu, D. Liu, S. Qin, Z. Ming, and T. Cong, "Automatically inspecting thousands of static bug warnings with large language model: How far are we?" *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 7, pp. 1–34, 2024.

[35] (2024) Meta llama 3. [Online]. Available: https://llama.meta.com/llama3

[36] (2024) Gemma 2. [Online]. Available: https://ai.google.dev/gemma/docs/model_card_2

[37] (2024) Gemini. [Online]. Available: https://deepmind.google/technologies/gemini/

[38] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting kernel memory leaks in specialized modules with ownership reasoning," in *Proceedings of 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, 2021.

[39] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *Proceedings of 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[40] W. Li, H. Cai, Y. Sui, and D. Manz, "Pca: memory leak detection using partial call-path analysis," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1621–1625.

[41] P. Bian, B. Liang, J. Huang, W. Shi, X. Wang, and J. Zhang, "Sinkfinder: harvesting hundreds of unknown interesting function pairs with just one seed," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1101–1113.

[42] X. Chen, X. Hu, Y. Huang, H. Jiang, W. Ji, Y. Jiang, Y. Jiang, B. Liu, H. Liu, X. Li *et al.*, "Deep learning-based software engineering: Progress, challenges, and opportunities," *arXiv preprint arXiv:2410.13110*, 2024.

[43] C. Wang, J. Zhang, Y. Feng, T. Li, W. Sun, Y. Liu, and X. Peng, "Teaching code llms to use autocompletion tools in repository-level code generation," *arXiv preprint arXiv:2401.06391*, 2024.

[44] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1703–1726, 2024.

[45] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[46] X. Du, G. Zheng, K. Wang, J. Feng, W. Deng, M. Liu, B. Chen, X. Peng, T. Ma, and Y. Lou, "Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag," *arXiv preprint arXiv:2406.11147*, 2024.

[47] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large language model-based agents for software engineering: A survey," *arXiv preprint arXiv:2409.02977*, 2024.

[48] J. Zhang, C. Wang, A. Li, W. Wang, T. Li, and Y. Liu, "Vuladvisor: Natural language suggestion generation for software vulnerability repair," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1932–1944.

[49] J. Zhang, C. Wang, A. Li, W. Sun, C. Zhang, W. Ma, and Y. Liu, "An empirical study of automated vulnerability localization with large language models," *arXiv preprint arXiv:2404.00287*, 2024.

[50] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, 2023.

[51] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A survey of large language models for code: Evolution, benchmarking, and future trends," *arXiv preprint arXiv:2311.10372*, 2023.

[52] Y. Wang, W. Zhong, Y. Huang, E. Shi, M. Yang, J. Chen, H. Li, Y. Ma, Q. Wang, and Z. Zheng, "Agents in software engineering: Survey, landscape, and vision," *arXiv preprint arXiv:2409.09030*, 2024.

[53] Y. Wang, Y. Wang, D. Guo, J. Chen, R. Zhang, Y. Ma, and Z. Zheng, "Rlcoder: Reinforcement learning for repository-level code completion," *arXiv preprint arXiv:2407.19487*, 2024.

[54] F. Petroni, T. Rocktäschel, S. Riedel, P. S. H. Lewis, A. Bakhtin, Y. Wu, and A. H. Miller, "Language models as knowledge bases?" in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, 2019, pp. 2463–2473.

[55] Z. Jiang, F. F. Xu, J. Araki, and G. Neubig, "How can we know what language models know?" *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 423–438, 2020.

[56] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Comput. Surv.*, vol. 55, no. 9, pp. 195:1–195:35, 2023.

[57] L. Cui, Y. Wu, J. Liu, S. Yang, and Y. Zhang, "Template-based named entity recognition using BART," in *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*, ser. Findings of ACL, C. Zong, F. Xia, W. Li, and R. Navigli, Eds., vol. ACL/IJCNLP 2021, 2021, pp. 1835–1845.

[58] C. Wang, J. Zhang, Y. Lou, M. Liu, W. Sun, Y. Liu, and X. Peng, "Tiger: A generating-then-ranking framework for practical python type inference," *arXiv preprint arXiv:2407.02095*, 2024.

[59] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds., 2021, pp. 3045–3059.

[60] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds., 2021, pp. 4582–4597.

[61] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 382–394. [Online]. Available: https://doi.org/10.1145/3540250.3549113

[62] Q. Huang, Z. Yuan, Z. Xing, X. Xu, L. Zhu, and Q. Lu, "Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 79:1–79:13.