

Structure-Guided Semantic Deobfuscation of Android Identifiers

Mingyang Chen*

School of Cyber Science and
Technology, Shandong University
Qingdao, China
ocean.cmy@mail.sdu.edu.cn

Zhentaο Xie

School of Cyber Science and
Technology, Shandong University
Qingdao, China
fxizenta@mail.sdu.edu.cn

Yaqi Gao

School of Cyber Science and
Technology, Shandong University
Qingdao, China
202200460074@mail.sdu.edu.cn

Shuang Li

School of Cyber Science and
Technology, Shandong University
Qingdao, China
lishuang128@mail.sdu.edu.cn

Wenrui Diao✉*

School of Cyber Science and
Technology, Shandong University
Qingdao, China
diaowenrui@link.cuhk.edu.hk

Abstract

Android developers frequently employ identifier renaming to obfuscate code and hinder reverse engineering. This practice effectively conceals program semantics. However, it severely impedes downstream security analysis tasks, particularly manual inspection of malicious and privacy-sensitive behaviors. Existing deobfuscation techniques rely on code similarity, learned naming priors, or local LLM reasoning. These methods often exhibit severe limitations when applied to proprietary, heavily obfuscated, or cross-file context-dependent code. In this paper, we present DEMASK, a novel framework that integrates static program analysis with Large Language Models (LLMs) to recover meaningful identifiers from obfuscated Android apps. Guided by program structure, DEMASK executes a bottom-up, context-aware inference process for whole-app semantic reconstruction. It incorporates tailored naming constraints and optimization strategies to mitigate LLM hallucinations and ensure scalability for real-world codebases. We implemented a prototype and evaluated it on diverse datasets of benign and malicious Android apps. The results demonstrate that DEMASK significantly outperforms prior work in semantic accuracy and naming consistency. Specifically, it achieves an average effective semantic recovery of over 88.6% across all evaluation settings.

CCS Concepts

• Security and privacy → Software reverse engineering.

Keywords

Android apps, Identifier deobfuscation, Large language model

ACM Reference Format:

Mingyang Chen, Zhentaο Xie, Yaqi Gao, Shuang Li, and Wenrui Diao. 2026. Structure-Guided Semantic Deobfuscation of Android Identifiers. In *Proceedings of the 41st IEEE/ACM International Conference on Automated*

*State Key Laboratory of Internet Architecture, Tsinghua University; State Key Laboratory of Cryptography and Digital Economy Security, Shandong University



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2882-2/2026/10

<https://doi.org/10.1145/3832783.3837525>

Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3832783.3837525>

1 Introduction

The Android software ecosystem hosts millions of apps and remains a major focus for security and program analysis. Developers frequently employ code obfuscation techniques such as identifier renaming, control flow flattening, and string encryption to protect their code from reverse engineering and thwart unauthorized modifications [31, 35]. Identifier renaming is especially prevalent among these techniques due to its low overhead and high effectiveness. This transformation replaces meaningful program element names (e.g., class, method, or field names) with short or arbitrary strings. While this practice successfully safeguards intellectual property by stripping away valuable semantic cues and rendering syntactically valid code largely incomprehensible, it simultaneously complicates downstream security tasks. These tasks often require analysts to manually inspect malicious behavior, vulnerable code, and privacy-sensitive logic. Therefore, recovering meaningful names is essential to restore code readability and semantic understanding.

However, identifier recovery remains difficult because obfuscation removes much of the context required to understand the code. Prior research introduces several tools to address this problem. Anti-ProGuard [24] uses code similarity to match obfuscated apps with large open-source repositories. It transfers known identifiers when it finds a match. This approach works well for common libraries but struggles with proprietary or app-specific logic. DeGuard [26] learns naming patterns from unobfuscated datasets and predicts likely identifiers based on statistical features. DeGuard is more flexible but often suggests generic names heavily influenced by its training data. DOBF [36] casts identifier recovery as a deobfuscation pre-training task over synthetically obfuscated code. It captures general naming patterns but remains influenced by its training corpus and available context. These methods all **rely heavily on existing code corpora and learned naming conventions**.

Large language models (LLMs) trained on code and natural language offer a new direction for deobfuscation. Unlike rule-based or statistical methods, LLMs can learn deep semantic relationships and reason about program intent. Prior work shows that LLMs can recover meaningful names in heavily obfuscated JavaScript and C code [25, 27]. Androidmeda [15] also uses LLMs for semantic

identifier deobfuscation in code snippets. However, current LLM-based methods still face practical limitations when applied to Android apps. Scalability is a primary challenge. Modern Android apps often comprise tens of thousands of methods. Existing LLM-based methods typically operate on small, isolated code snippets. However, these local approaches lack the broader context required for accurate name recovery, whereas *feeding an entire app directly into an LLM remains computationally prohibitive*. Unconstrained whole-program analysis also leads to *inconsistency or hallucination* in the form of incorrect identifiers, as the model lacks awareness of inter-method dependencies and struggles to process noisy code structures. LLM-based approaches risk severe inefficiency and semantic drift without structured guidance.

Our Work. To address these challenges, we present DEMASK, a structure-guided Android deobfuscation framework that combines static analysis and LLM-based semantic reasoning to recover identifiers. Rather than treating identifier recovery as an isolated local naming task with the LLM as a black-box generator, DEMASK uses well-established program analysis principles to guide a bottom-up, context-aware LLM reasoning process for whole-app, dependency-aware semantic reconstruction. This design successfully captures the advantages of both static structure and semantic understanding.

Specifically, DEMASK first performs static analysis to extract the app's structure and construct a customized Function Call Graph (FCG) capturing caller-callee relationships. The FCG provides a global view for organizing deobfuscation. The process follows a hierarchical, bottom-up strategy: the LLM first analyzes leaf methods locally, infers meaningful names and summaries, and then uses them as context for higher-level methods. Through iterative propagation, DEMASK incrementally restores semantics from methods to classes and packages. To improve accuracy and reduce token cost, DEMASK incorporates a Third-Party Library (TPL) mapping module that detects known libraries and restores their identifiers from verified sources. This allows the LLM to focus on developer code requiring semantic inference. Also, DEMASK applies strict naming constraints to mitigate hallucinations and ensure consistent, valid identifiers throughout the recovered codebase.

We implemented a DEMASK prototype and conducted a comprehensive evaluation to assess its effectiveness and compare it with prior work. Our experiments covered both benign and malicious Android apps and used multiple metrics to evaluate the readability and correctness of the recovered code. Recovery quality was assessed through an automated LLM-based code reviewer and a user study. In both settings, DEMASK outperformed existing tools, achieving 88.6% average effective semantic recovery in the LLM evaluation and 96.7% in the user study.

Contributions. The main contributions of this paper are:

- **Structure-Guided Identifier Recovery.** We propose DEMASK, a framework combining static program analysis and LLM-based reasoning to recover obfuscated identifiers in Android apps. DEMASK frames this task as a whole-app, dependency-aware semantic reconstruction. It uses program structure to guide a bottom-up LLM process that incrementally reconstructs code semantics, propagates context, and ensures consistent naming.
- **Scalable and Consistent Recovery.** We introduce targeted optimizations to improve scalability and robustness. DEMASK

detects and prunes TPL code to avoid redundant LLM queries, while focusing on app-specific logic. In addition, it applies naming constraints and syntax-aware filtering to mitigate hallucinations, improve token efficiency, and enforce naming consistency across the app.

- **Implementation and Evaluation.** We built an automated prototype of DEMASK and evaluated it on real-world obfuscated Android apps. Our results show that DEMASK substantially outperforms prior work in both semantic accuracy and naming consistency. Component contribution analysis further validates the effectiveness of our design components.

2 Background and Insights

This section provides the necessary background on deobfuscation.

2.1 Android Obfuscation Techniques

Android apps commonly adopt code obfuscation to hinder reverse engineering and static analysis [31]. Common methods like identifier renaming, control flow flattening, and string encryption obscure program logic and semantics, making manual inspection and automated analysis significantly harder.

To automate obfuscation, several industry-grade tools have been developed, including R8 [19], DexGuard [1], and DashO [5]. As the default obfuscator in the Android build system, R8 is widely deployed in practice [16]. While these tools differ in functionality, they all support *identifier renaming*, which serves as the core mechanism in real-world obfuscation [31, 35]. This work focuses on analyzing and reversing this key technique.

Identifier renaming replaces meaningful program element names with short or arbitrary strings, removing semantic cues and hindering reverse engineering and downstream analysis. For example, a class named `UserAuthenticator` may be renamed to `a`. R8 supports both default and customized renaming strategies: the default strategy generates short names like "a", "b", and "c", while custom rules produce random sequences.

2.2 Obstacles in Identifier Deobfuscation

Existing Identifier Deobfuscation. Deobfuscation is the reverse process of obfuscation and seeks to restore the readability of obfuscated apps. Among various tasks, this work focuses on *identifier deobfuscation*, which aims to recover the semantics of program identifiers. Existing techniques span both industry and academia. In practice, tools such as Jadx [8] and JEB [9] provide basic name recovery to structurally distinguish program elements, but they rarely recover meaningful names beyond structural hints. In academia, Anti-ProGuard [24] uses code similarity to map obfuscated code to known libraries. However, its scope is limited to open-source TPLs indexed in its repository, rendering it ineffective for proprietary or custom app logic. DeGuard [26] learns statistical patterns from unobfuscated APKs to predict original names, but it relies heavily on training data and adapts poorly to evolving APIs and libraries, resulting in reduced effectiveness against new or unseen patterns. DOBF [36] introduces a deobfuscation pre-training objective for recovering identifiers from synthetically obfuscated code. However, it remains limited by its training corpus and context window for app-specific semantics and cross-file dependencies.

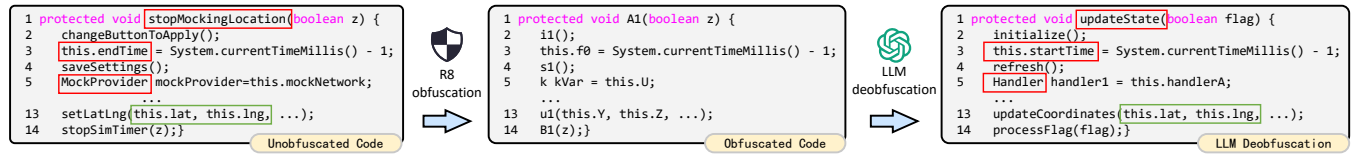


Figure 1: Case of context loss leading to inaccurate identifier recovery.

LLM-based Identifier Deobfuscation. Compared to traditional methods, recent advances in LLMs show strong potential for code deobfuscation. Academic work explores LLM-based methods using general-purpose models [25], fine-tuning [30], and JavaScript benchmarks [27]. In industry, tools like Androidmeda [15] apply LLMs to suggest meaningful identifiers by analyzing isolated code snippets, though they often lack rigorous whole-program constraints. Android apps are large, and the limited context window of LLMs restricts analysis to individual functions or classes, hindering the capture of global context, such as deep call chains and cross-module dependencies. This leads to less accurate identifier recovery and inconsistent naming in compiled languages like Java and C. Moreover, LLMs suffer from hallucination and unstable outputs. They sometimes generate syntactically valid but semantically incorrect identifiers [53], which can break logical dependencies, cause compilation failures, and undermine reliability.

Design Insights. To address these limitations, we propose DEMASK, a hierarchical framework combining static analysis and LLMs to recover identifier semantics across methods, classes, and packages. Static analysis provides reliable structural information such as class hierarchies, FCG, and dependencies, while LLMs infer semantic intent and contextual meaning. By combining these complementary strengths, DEMASK captures both local and global contexts, achieving accurate and consistent identifier recovery with reduced inference overhead in complex apps.

3 Design Challenges and Solutions

While the integration of static analysis and LLMs is promising, implementing DEMASK introduces several technical challenges, including accurate semantic recovery, maintaining consistency, and minimizing token usage. Here we discuss these challenges.

Challenge 1: How to effectively recover identifiers with correct semantics? Identifier renaming strips critical method-level semantics, such as invocation relationships and class associations. Such context loss often leads LLMs to generate generic or incorrect identifiers when analyzing isolated code fragments. As illustrated in Figure 1, when only a single method is provided as input, the original method `stopMockingLocation` is incorrectly recovered by GPT-5.4 as the overly broad `updateState`, the key field `endTime` is misinterpreted as `startTime`, and the class `MockProvider` is incorrectly inferred as `Handler`. Although some fields, such as `lat` and `lng`, are correctly restored, the overall results demonstrate that accurate semantic reconstruction cannot be achieved by relying solely on local code fragments.

Solution to Challenge 1: The essence of identifier deobfuscation lies in leveraging remaining information after obfuscation to infer the original semantics of identifiers. It is an information completion

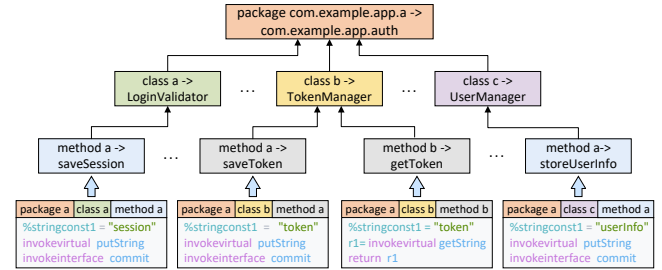


Figure 2: Design of the hierarchical reasoning process.

process that reconstructs the semantic properties obscured during obfuscation. Prior approaches rely solely on information within individual methods or classes, which is insufficient for accurate semantic inference. To address this, we integrate diverse program contexts to provide rich semantic references. Details are discussed in Section 4.2.

The contextual information above must be organized structurally for effective use by LLMs. Android apps are inherently hierarchical, where methods define class behavior and classes collectively implement package-level functionality. Leveraging this structure, we adopt a bottom-up approach for semantic recovery. We first build a complete FCG to capture method relationships. The analysis starts from leaf methods whose semantics can be inferred independently, then propagates upward to infer higher-level methods, culminating in class-level and package-level reconstruction. Figure 2 illustrates this hierarchical reasoning process.

Challenge 2: How to ensure the semantic and syntactic consistency of recovered identifiers? Java’s object-oriented features impose complex constraints such as inheritance, method overriding, and polymorphism. Randomness in LLM-generated names can violate these constraints, causing semantic and syntactic inconsistencies. For example, if a subclass method is mistakenly renamed to the same name as a method in its parent class, the compiler treats it as an override. This changes the program’s behavior because calls intended for the parent method will instead invoke the subclass method, altering the original logic.

Solution to Challenge 2: To preserve semantic and syntactic consistency during deobfuscation, we design a set of identifier renaming constraints based on the architectural characteristics of Android apps and Java’s syntactic rules. These constraints aim to maintain correctness by ensuring consistent method naming for overridden methods, avoiding naming conflicts among classes and fields, preserving the modularity of package structures through unique names, and guaranteeing consistent naming across call

chains. This approach effectively prevents semantic errors and compilation issues caused by improper renaming. Detailed descriptions of these constraints are provided in Sections 4.3 and 4.4.

Challenge 3: How to minimize the token usage of LLMs for deobfuscation tasks? Android apps often contain large volumes of code, and their core logic typically depends on multiple third-party libraries, forming extensive dependency chains. For instance, the shopping app Shopt [11] includes about 324,000 lines of code and 62,000 methods. Our estimation using tiktoken [13] shows that processing the entire codebase would require approximately 10.86 million tokens. Directly applying an LLM to such a large input results in substantial computational overhead and a high monetary cost, making the full-scale deobfuscation of identifiers impractical for real-world Android apps.

Solution to Challenge 3: To reduce token consumption during deobfuscation, we minimize the use of LLMs in cases where traditional static analysis is sufficient or where semantic recovery is unnecessary. On the one hand, some program elements, like method calls to widely used open-source libraries, can be accurately identified through static analysis. In these cases, the obfuscated method names can be replaced with known unobfuscated versions based on public API definitions, enabling precise recovery without involving LLMs. On the other hand, code that is never invoked at runtime, including unused methods or irrelevant third-party components, has little impact on the app's core functionality and can be safely excluded from LLM processing. This selective strategy preserves the accuracy of semantic recovery while significantly reducing token usage and improving deobfuscation efficiency.

4 Design of DEMASK

This section presents the detailed design of DEMASK, which consists of five main phases, as illustrated in Figure 3.

Phase 1: TPL Mappings. This step identifies third-party libraries and maps obfuscated identifiers to their unobfuscated counterparts to recover known semantics.

Phase 2: Context-Rich FCG construction. This step builds and prunes the FCG to reduce LLM overhead while enriching contextual information to guide LLM reasoning.

Phase 3: Method and Field Deobfuscation. This step aims to recover method and field names through bottom-up analysis while enforcing consistency and uniqueness constraints.

Phase 4: Class and Package Deobfuscation. This step aims to infer class and package names and apply strict naming constraints to ensure consistency.

Phase 5: Validation and APK Rebuilding. This step adjusts identifiers for naming compliance and generates the final deobfuscated APK.

4.1 Phase 1: TPL Mappings

This phase identifies TPLs and maps obfuscated components to their unobfuscated counterparts. The resulting mappings are preserved for later phases to enhance the FCG and code context.

Construction of Package and Class Mappings. DEMASK uses LibPecker [52] to identify TPLs by analyzing structural dependencies among classes, retaining libraries above its recommended *library similarity threshold* ($\tau = 0.6$). This structural matching ensures

high accuracy even when identifiers are fully obfuscated. Based on these results, it constructs initial mappings from obfuscated package and class names to their unobfuscated counterparts in known libraries, retaining similarity scores to indicate confidence.

Initial mappings may contain conflicts, such as the same package path mapping to different libraries or inconsistencies between parent and subpackage mappings. To address these issues, DEMASK first removes duplicate entries within each library, keeping only the candidates with the highest similarity. It then confirms parent package mappings based on the package hierarchy. Afterward, it filters subpackage mappings to resolve conflicts and ensure global consistency. Class mappings are retained only if they match the validated package structure.

Extraction of Method Mappings. Based on the class mappings, DEMASK further constructs method-level mappings. In Java, method signatures are uniquely determined by the method name and parameter types [2]. However, obfuscation complicates signature matching. To overcome this, we design a fuzzy matching strategy that integrates structural information. This process consists of three steps: (1) *Direct Signature Matching*: DEMASK first matches methods using signatures and records methods with unchanged names across the mapped class pair as unobfuscated. (2) *Type-Based Signature Recovery*: For unmatched methods, DEMASK marks only primitive parameters and reference parameters matching SDK, JDK, or class mappings as unobfuscated. It then replaces obfuscated parameters and return types with resolved class mappings, subsequently attempting to identify unique matches based on the updated signatures. (3) *Heuristic Resolution*: If a class contains only one remaining unmatched method, it is considered uniquely identifiable and mapped accordingly. Methods that still cannot be uniquely resolved are discarded to ensure mapping precision.

4.2 Phase 2: Context-Rich FCG Construction

This phase constructs a fine-grained FCG, prunes semantically irrelevant nodes to narrow the reasoning scope, and enriches the remaining nodes with contextual information. These steps ensure that the LLM focuses on semantically important developer-defined methods with sufficient and accurate contextual cues.

FCG Initialization. To ensure precise and comprehensive analysis, DEMASK utilizes PacDroid [28]. Compared to Androguard's XREF-based over-approximation [3] and FlowDroid's iterative reanalysis [22] for FCG stabilization, PacDroid is built on the Tai-e engine [46] and uses pointer analysis. It models Android features such as ICC and lifecycles within a unified pointer-analysis framework, reducing repeated analysis and better aligning with DEMASK's need for scalable Android context construction.

However, PacDroid's FCG focuses on data-flow analysis and retains only edges affecting data propagation. As a result, some lifecycle and asynchronous paths are omitted. For example, Activity launches typically connect only to the target's `onNewIntent` method. To enable more comprehensive semantic recovery, DEMASK augments the FCG by adding parallel edges from call sites to lifecycle methods of Activities and Services. It also adds edges to `AsyncTask` lifecycle methods, providing broader contextual coverage for subsequent analysis and deobfuscation.

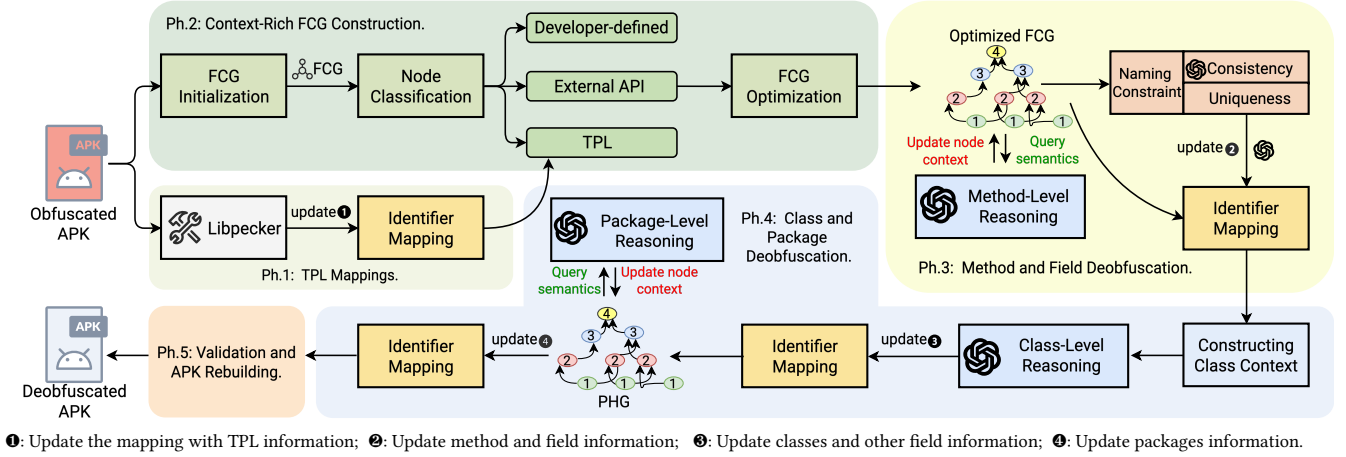


Figure 3: Overall design of DEMASK.

To support later deobfuscation and reasoning, DEMASK assigns structured attributes to each method node in the FCG, including method signatures, Tai-e IR snippets, node types, and semantic annotation fields for storing deobfuscated identifiers and inferred semantic information. These attributes provide essential contextual support for subsequent analysis.

Node Classification. To facilitate downstream semantic reasoning and FCG refinement, DEMASK classifies each method node into one of three types based on its origin and semantic availability:

- **External API Nodes.** These nodes are methods from the Android SDK or JDK, identified by class path matching. Since they are not obfuscated and have clear semantics, they serve as stable anchors for contextual inference.
- **TPL Nodes.** These nodes are divided into two subtypes: (1) **Exact-TPL:** Methods that have been precisely matched to a known library through TPL method mappings. (2) **LibLevel-TPL:** Methods that can be attributed to a specific TPL at the package or class level through TPL package and class mappings, but whose exact method-level match is unknown. Both provide partial semantic context and can facilitate downstream deobfuscation.
- **Developer-defined Nodes.** These are the remaining methods treated as being implemented by the app developer, representing core logic. As they are fully obfuscated and have no external references, they serve as the primary focus of LLM-based semantic recovery in later phases.

FCG Optimization. To reduce token usage and focus LLM-based deobfuscation on the most semantically relevant components, DEMASK prunes the FCG selectively. The strategy first excludes FCG entry points rooted in TPL and initiates traversal only from the remaining Developer-defined entry nodes. Since PacDroid models common Android lifecycle and callback methods as FCG entry points, DEMASK preserves most callback logic by traversing from entries classified as Developer-defined. During traversal, DEMASK applies horizontal pruning by truncating edges at selected boundary nodes. External API nodes and Exact-TPL nodes serve as unconditional termination points due to their fully determined semantics, while LibLevel-TPL nodes are pruned only when their contextual information is sufficiently rich.

To determine whether a LibLevel-TPL node provides sufficient semantic context for pruning, we introduce the **Context Information Score (CIS)**, which measures the proportion of unobfuscated semantic elements, such as field names, method names, and constants, within a method. DEMASK identifies unobfuscated elements using regular expression pattern matching and a whitelist of standard Java identifiers and common abbreviations collected from large-scale open-source projects, filtering out random or meaningless names from obfuscation. A node is considered contextually sufficient if its CIS exceeds 0.7¹.

Semantic Context Enrichment. To enhance semantic reasoning in LLM-assisted deobfuscation, DEMASK uses the constructed TPL mappings to replace obfuscated TPL identifiers with their corresponding unobfuscated forms, improving code clarity and reducing ambiguity. Additionally, DEMASK resolves resource identifiers by extracting their original type and name (e.g., mapping `R.id.$id` to `button_login` of type `id`). These names typically carry explicit semantics that reflect the resource's intended function, offering valuable cues for understanding surrounding logic. By integrating unobfuscated identifiers and semantically meaningful resource names, DEMASK enriches the code context and facilitates more accurate LLM reasoning.

4.3 Phase 3: Method and Field Deobfuscation

After optimizing the FCG and enriching context, DEMASK deobfuscates methods and fields in a bottom-up manner. It propagates semantic information upward through the FCG. To ensure correctness, it enforces consistency and uniqueness constraints, preserving original program semantics while standardizing naming.

Determining Analysis Order. To improve LLM inference accuracy, DEMASK adopts a layered analysis strategy based on the optimized FCG. Nodes are grouped by method call dependencies. This ensures that the semantic information of callees is available before analyzing their callers. Typically, the leaf nodes include

¹We tested 100 obfuscated methods from 0.1 to 0.9 to derive the threshold: the LLM generated names and descriptions, and LLM raters assessed their semantic consistency against original code. The rate saturated at 0.7 (92.8%), with only a 1.6% gain.

external APIs, Exact-TPL methods, and nodes with high CIS, which together provide sufficient context to guide deobfuscation.

For recursive or mutually dependent methods forming cycles in the FCG, DEMASK identifies strongly connected components (SCCs) [47] and treats each as a supernode. This converts the FCG into a directed acyclic graph (DAG) of SCCs. DEMASK then performs bottom-up analysis on this DAG. Each SCC is analyzed only after all its dependent SCCs have been processed, which ensures that semantic information flows correctly across cycles.

Within each SCC, analyzing a node without complete callee information may produce inaccurate results due to cyclic dependencies. To address this, DEMASK uses a two-round iterative process. In the first round, it repeatedly selects the best node (i.e., the node with the fewest unanalyzed callees) as the starting point. This strategy prioritizes nodes with richer context. The system performs LLM-based semantic analysis and propagates the results to all callers in the SCC, updating their callee information. After all nodes have been analyzed once, DEMASK identifies those that still lack sufficient context due to unresolved dependencies. These nodes are re-analyzed in the second round. During this phase, the global context established in the first round supplements the missing information. The process iterates until all SCC methods are analyzed or no new intra-SCC context can be obtained. At this point, the semantic interpretations are stable, and reanalysis of each method is bounded by its unresolved intra-SCC callees.

Performing Method-Level Reasoning. After determining the analysis order, DEMASK constructs a structured prompt context for each method node to guide LLM-based deobfuscation. This context includes the complete method signature, preprocessed Tai-e IR-format code snippets, accessed field indices, and the deobfuscation results of all callees, including their names and LLM-generated semantic explanations. When applicable, DEMASK further enriches this context by using TPL mappings, which replace obfuscated package or class paths with their unobfuscated counterparts, providing stronger semantic cues. The constructed context is fed to the LLM, which infers the functionality of the current method and returns suggested method names, field names, and their corresponding behavioral descriptions. Manual checking of 100 descriptions found 94% semantic consistency. Semantic information is propagated bottom-up along the FCG. As more callees are processed, the context for each method becomes increasingly complete.

For fields referenced by multiple methods, DEMASK uses the LLM to analyze semantic patterns and behavioral descriptions across contexts. It then selects the most semantically appropriate name based on naming richness. This ensures consistent and meaningful field naming throughout the codebase. Figure 4 shows a representative example of this method-level reasoning. In this case, DEMASK infers the semantics of `sanitizeInput` and `multiply` to rename the obfuscated fields `b` and `c` as `firstFactor` and `secondFactor`, and recovers the method name `calculateProduct`.

Enforcing Method Naming Constraints. To ensure consistent and semantically correct naming after deobfuscation, DEMASK enforces two constraints on method naming: (1) the *Consistency Constraint*, which ensures overriding methods in subclasses retain the same name as in their superclasses to preserve polymorphism, and (2) the *Uniqueness Constraint*, which prevents subclasses from

inadvertently overriding superclass methods and ensures that all method signatures within a class remain unique.

To satisfy the consistency constraint, DEMASK first constructs a class inheritance graph and clusters methods by traditional signatures. These clusters are further divided into override groups based on inheritance relationships. Within each override group, all methods must share the same name after deobfuscation. Since LLM-generated names may not respect these override semantics, DEMASK resolves conflicts by prioritizing names from known methods in external or Exact-TPL nodes. These known names provide higher confidence due to their unobfuscated origin. If no known names are available, DEMASK selects a unified name based on semantic richness and generality from among the LLM-suggested candidates, leveraging the LLM to evaluate and identify the most appropriate choice. This strategy helps preserve the object-oriented structure of Java programs.

To satisfy the uniqueness constraint, DEMASK extends DeGuard’s approach [26]. It performs a bottom-up topological sort of the class hierarchy and, for each class, collects non-private methods inherited from superclasses. Excluding methods grouped as overrides, DEMASK checks the remaining methods for naming collisions. If conflicts are found, new unique names are assigned. Since not all SDK or JDK methods appear in the Dex file, we built a protection list of public methods from Android SDK versions (14-36) and JDK LTS versions (8, 11, 17, 21). This list is used during LLM-based renaming to avoid conflicts with standard library methods, ensuring semantic correctness and avoiding unintended overrides.

4.4 Phase 4: Class and Package Deobfuscation

This phase aims to recover semantic information for classes and packages. Based on method-level deobfuscation results, DEMASK identifies only developer-defined classes as analysis targets. It also handles exceptional cases, such as inlined fields, via dedicated compensation analysis. After recovering classes, it rebuilds the package hierarchy and assigns meaningful package names, thereby forming a complete semantic recovery chain that spans from methods to classes and ultimately to packages.

Constructing Class Context and Performing LLM Reasoning. DEMASK selects target classes whose methods have been successfully deobfuscated, ensuring they originate from developer-defined code. For each class, it constructs a context that includes structural information (e.g., class path, superclass, interface path, and relevant annotations), behavioral details (e.g., complete method signatures and semantic descriptions generated by the LLM), and field information. Known fields are recovered via LLM inference or mapped from resource class `R` using resource IDs, while unknown fields mainly refer to inlined fields.

Field inlining is a compiler optimization that embeds field values directly into usage sites, replacing explicit field-access instructions with constant values. This optimization reduces observable field references, making early method-level recovery insufficient. The inlined values typically appear as constants, lacking direct connections to original field definitions. Recovering these fields requires class-level reasoning, where DEMASK leverages richer structural and semantic context. To enrich the context, DEMASK augments the class representation with semantics derived from known TPL

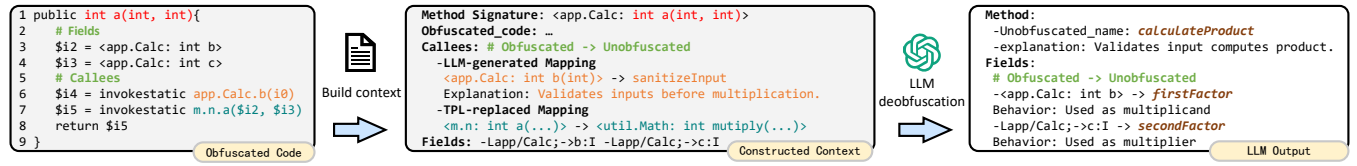


Figure 4: Example of method-level reasoning.

identifiers based on prior mappings. The assembled context is then encoded as a structured prompt and fed to the LLM. Through semantic reasoning, the LLM infers class names, field names, and concise class functionality summaries, completing the class-level deobfuscation process.

Constructing Package Context and Performing LLM Reasoning. DEMASK constructs a package hierarchy graph (PHG) based on the actual distribution of classes in the app. Each node in the PHG represents a package containing one or more class definitions. The graph captures both the structural nesting of packages and their class composition. Package deobfuscation is performed in a bottom-up manner: analysis begins with the leaf packages and progresses toward the root. This ordering ensures that subpackages are resolved before their parent packages, providing sufficient semantic context to support higher-level inference.

For each package, DEMASK collects a rich semantic context, including deobfuscated class names and their descriptions within the package, along with the semantic names and descriptions of the package’s direct subpackages. The combined information is then provided to the LLM. Using both the features of the classes and the semantics of subpackages, the LLM infers a meaningful package name and generates a concise functional description.

Applying Naming Constraints for Classes, Fields, and Packages. To ensure consistent identifiers and preserve the integrity of the code structure, DEMASK enforces strict naming constraints at the class, field, and package levels.

- (1) At the class and field levels, all classes within the same package must have distinct names, and all fields within a class must also be uniquely named. Field type is excluded from this constraint since fields are referenced by name rather than by signature. This contrasts with methods, which are invoked by signature, where parameter types are an essential component.
- (2) At the package level, DEMASK requires all root package names to be globally unique to prevent unintended merging. Direct subpackages under the same parent must also have unique names. This constraint is particularly important in large projects where obfuscated package names are renamed while maintaining the original logical separation.

4.5 Phase 5: Validation and APK Rebuilding

In the final repackaging phase, DEMASK verifies all identifiers renamed by the LLM to ensure compliance with Java language specifications. This step resolves any invalid names that may result from LLM hallucinations, such as identifiers containing illegal characters (e.g., `my-var!`, `foo@bar`). These names are sanitized by replacing disallowed characters with underscores, producing valid Java identifiers that can be correctly parsed and compiled.

After format validation, DEMASK applies the Identifier Mapping to substitute obfuscated names in the Smali code with their normalized counterparts. The updated code is then repackaged using Apktool [4] to generate the final deobfuscated APK, allowing security analysts to directly re-decompile it for security auditing.

5 Evaluation

This section presents a comprehensive evaluation of DEMASK to answer the following research questions.

- RQ1:** How well does DEMASK recover meaningful identifiers compared to existing tools?
- RQ2:** How does each component of DEMASK contribute to identifier deobfuscation?
- RQ3:** How does DEMASK affect the analysis of malicious samples?

5.1 Experiment Setting

We implemented DEMASK with roughly 9,100 lines of Java code. The system is built on the PacDroid [28] framework and integrates several open-source tools, including Apktool [4], SootUp [34] (for TPL method-matching metadata extraction), and LibPecker [52]. Experiments were conducted on a server running Ubuntu 22.04.5 LTS, equipped with a 96-core AMD EPYC 9654 CPU and 1 TB RAM.

Model Selection. DEMASK adopts GPT-5.4-mini as its unified LLM backend, chosen for its balance between performance and cost. Compared to larger models, GPT-5.4-mini offers significantly lower invocation costs [18] and supports high-frequency concurrent requests. In addition, its “thinking mode” [17] provides superior semantic reasoning, essential for large-scale code analysis. We set a low temperature of 0.1 for consistent identifier recovery.

Dataset. We evaluated DEMASK using three datasets:

- **Benign App Dataset.** We randomly selected 50 open-source apps from F-Droid [6]. The apps were obfuscated using R8 with default settings. The original versions serve as the ground truth for evaluating semantic correctness.
- **Malicious App Dataset.** To assess the effectiveness of DEMASK on real-world malware, we randomly selected 30 obfuscated apps from AndroZoo [21]. All samples were submitted within the past five years and were flagged as malicious by at least 30 detection engines on VirusTotal [14].
- **TPL Dataset.** We collected all apps released on F-Droid over the past two years, excluding those in the benign dataset. From these projects, we extracted libraries referenced by at least two projects and selected the most frequently used version for each. This process yielded a dataset of 741 unique TPLs.

Metrics. Since exact-match metrics cannot fully evaluate semantic recovery, we use one lexical and two semantic metrics:

Table 1: Results comparing with existing tools.

Metric		DeMask	Androidmeda	DOBF
Rater-GPT	ERR	79.2%	59.5%	31.6%
	BRR	67.2%	19.2%	13.6%
Rater-Gemini	ERR	95.6%	34.6%	22.3%
	BRR	70.3%	21.1%	8.6%
Rater-DeepSeek	ERR	89.0%	40.0%	22.8%
	BRR	69.3%	23.6%	7.1%
Rater-Claude	ERR	90.5%	44.5%	28.4%
	BRR	71.9%	21.8%	6.3%
LLM Rater Avg.	ERR	88.6%	44.7%	26.3%
	BRR	69.7%	21.4%	8.9%
User Study	ERR	96.7%	55.1%	33.8%
	BRR	73.1%	16.7%	10.2%
Subtoken F1	Method	0.45	0.36	0.19
	Field	0.39	0.31	0.13
	Class	0.44	0.33	0.16
	Package	0.59	0.47	N/A
Overhead / 1K Methods	Avg. Time	3.2 s	95.5 s	694.2 s
	Avg. Token	4.8K	714.1K	N/A

- **Subtoken F1.** This metric evaluates lexical recovery quality using the F1 score of overlapping subtokens between recovered and original identifiers.
- **Effective Recovery Rate (ERR).** This metric evaluates whether recovered identifiers accurately reflect original semantics. A snippet is effectively recovered if its names clearly convey the intended meaning to the evaluator, despite exact mismatch.
- **Best Result Rate (BRR).** This metric compares recovery quality across different deobfuscation tools for the same snippet. A snippet is the best result if its recovered identifiers are judged closest in meaning to the original.

Overhead. Due to the pruning effect of TPL methods, the overhead of DeMask grows non-linearly. To assess this, we divided the method counts of each app into four intervals. For each interval, the average time and token consumption per 1,000 methods (including TPL) in the benign dataset were as follows: 10.8 s and 8.4K (<10K methods), 7.4 s and 7.2K (10K–30K methods), 3.2 s and 4.8K (30K–100K methods), and 2.4 s and 4.2K (>100K methods). As the number of methods increases, the pruning of TPL methods grows, thereby lowering the overhead for processing developer-defined code.

To ensure consistent evaluation, we selected 37 apps with 30K–100K methods from the benign dataset for all subsequent overhead assessments. This range is common, representing 66.7% of the 100,000 apps we analyzed from AndroZoo [21]. As shown in Table 1, token usage per 1,000 methods reached 714.1K for Androidmeda, with a time consumption of 95.5 s. The significant disparity in efficiency between the two tools is primarily attributed to DeMask’s lightweight context and its pruning of TPL methods. For DOBF, runtime depends on the local configuration and is therefore reported only as a reference, not for direct comparison.

5.2 Comparison with Existing Tools (RQ1)

To evaluate DeMask’s effectiveness in recovering meaningful identifiers, we selected baselines with similar semantic restoration goals. Most existing tools differ in their design objectives. Anti-ProGuard focuses on TPL recovery. Decompilers like Jadx and JEB emphasize structural readability rather than the semantic recovery of key methods and classes. In contrast, Androidmeda [15], DOBF [36], and

DeGuard [26] align more closely with our objectives. Androidmeda uses an LLM-based approach but relies only on local context within individual files. DOBF is an identifier deobfuscation pre-training objective applicable to Java code. DeGuard employs probabilistic models to recover global semantics, but it is closed-source, and its online service [12] was unavailable during our experiments. Our inquiries to the maintainers received no response. Therefore, we used Androidmeda and DOBF as primary baselines on the full benign dataset, and compared Anti-ProGuard only against DeMask’s TPL component in Section 5.3.

Evaluation Approach. To evaluate deobfuscation effectiveness, we combined an objective lexical assessment, an automated LLM-based semantic assessment, and a user study. Since LLM-generated identifiers are semantic reconstructions, both LLM raters and user-study participants were instructed to use semantic consistency rather than traditional exact-matching strategies (e.g., DeGuard) as the criterion. Specifically, the objective lexical assessment measures subtoken-level overlap between recovered and original names, while state-of-the-art LLMs assess how accurately the deobfuscated code conveys its original meaning [44]. Furthermore, we conducted an IRB-approved user study to measure developers’ code understanding efficiency, providing practical usability evidence. The three evaluations are detailed below:

- **Objective Lexical Assessment.** We computed Subtoken F1 for methods, fields, classes, and packages. For each identifier type, we reported micro-averaged Subtoken F1 over all identifier pairs in the full benign dataset and summarized per-app distributions using the median and interquartile range [Q1,Q3].
- **LLM-Rater Assessment.** We evaluated the outputs using four advanced models: GPT-5.4, Gemini-3.1 Pro, DeepSeek-V3.2, and Claude-Sonnet-4.6. Each model was instructed to act as a code analysis expert, referred to as Rater-GPT, Rater-Gemini, Rater-DeepSeek, and Rater-Claude, respectively. To reduce the impact of hallucination [53] and instability, we ran each evaluation three times independently. For the full-dataset comparison, we randomly selected 1,000 snippet groups from the outputs of evaluated tools over 50 deobfuscated apps. The LLM raters evaluated each group with prompts designed for the ERR and BRR metrics. For each rater, a tool’s ERR and BRR were the three-run averages of its effective-recovery and best-result fractions.
- **User Study.** We conducted a user study involving 34 participants. They were undergraduate students majoring in computer science or related fields, representing a typical IT background. These participants also took part in the evaluation for RQ2. A quality-control questionnaire was included to filter out inattentive responses. Each participant evaluated 15 snippet groups randomly selected from the full dataset. Tool names were anonymized to reduce identity bias. After excluding four responses that failed the control checks, we retained 30 valid questionnaires for analysis. For the user study, a tool’s ERR and BRR were the snippet-averaged participant fractions for effective-recovery and best-result judgments, respectively.

Results: Subtoken F1. As shown in Table 1, DeMask achieves the highest Subtoken F1 across all identifier types. The modest absolute scores reflect that LLM-based semantic recovery may generate names that are semantically correct but lexically different from the

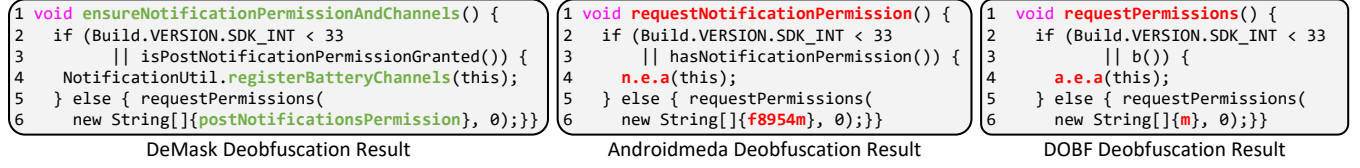


Figure 5: Deobfuscation case (biz.binarysolutions.healthybatterycharging)

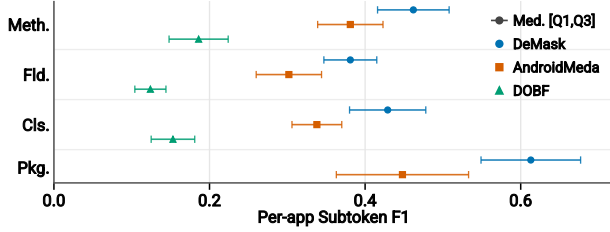


Figure 6: Per-app Subtoken F1 distributions.

ground truth. Nevertheless, DEMASK consistently outperforms the baselines, suggesting that structured context helps recover identifiers closer to the original names. Since DOBF does not support package recovery, its package-level Subtoken F1 is not applicable. Figure 6 further shows that DEMASK obtains higher median per-app Subtoken F1 for each identifier type, with interquartile intervals indicating that the advantage is not driven by a few outliers.

Results: ERR. As shown in Table 1, both the LLM-Rater evaluation and the user study indicate that DEMASK outperforms the baselines in terms of effectiveness for semantic recovery. This is primarily because Androidmeda’s single-class analysis and DOBF’s limited local context make cross-file semantics difficult to capture, whereas DEMASK performs structured semantic inference over program context. Additionally, we observed that DEMASK often generates identifiers with richer semantics than the original names. For instance, in the app cl.coders.faketraaveler, it restored `init()` as `initializeUiAndAuthState()`, clearly reflecting both UI initialization and authentication logic. This enables analysts to understand the method’s core intent without inspecting its body.

Results: BRR. DEMASK markedly outperforms Androidmeda and DOBF in semantic recovery quality. In the LLM-based evaluation, the average BRR was 69.7%, indicating strong consensus that it is a superior identifier recovery. In the user study, DEMASK achieved a BRR of 73.1%, indicating that most participants found its outputs more closely aligned with the original code. The differences between human and LLM ratings may stem from the number of snippets evaluated and evaluation logic: humans often read code sequentially and rely on local context, while LLMs evaluate each snippet with model-specific semantic criteria.

To illustrate the effectiveness intuitively, we present a comparison case in Figure 5, with the corresponding unobfuscated code shown in Listing 1. In this example, DEMASK successfully restores semantically meaningful identifiers: the method name `ensureNotificationPermissionAndChannels` clearly conveys the intent to check and ensure that notification permissions and channels are ready, making the code’s purpose immediately apparent.

Table 2: Results of ablation experiments.

Metric		DeMask	Direct-LLM	w/o-FCG	w/o-TPL
LLM Rater Avg.	ERR	89.4%	30.6%	63.3%	83.9%
	BRR	51.7%	1.7%	7.1%	39.5%
User Study	ERR	96.7%	27.6%	66.0%	91.3%
	BRR	49.8%	4.4%	13.6%	32.2%
Overhead / 1K Methods	Avg. Time	3.2 s	80.3 s	3.8 s	71.7 s
	Avg. Token	4.8K	411.8K	5.2K	224.7K

In contrast, the results of Androidmeda and DOBF retain partially obfuscated identifiers, such as the method call `n.e.a(this)` and the constant `f8954m`. In addition, DOBF recovers the method as the generic name `requestPermissions`, which fails to reflect the notification-channel registration semantics. These identifiers obscure the true semantics and hinder comprehension. This comparison demonstrates the clear advantage of DEMASK in improving code readability and semantic accuracy.

```

1 void createNotificationChannels() {
2   if (Build.VERSION.SDK_INT < 33
3     || hasNotificationPermission()) {
4     NotificationChannelUtil.createChannels(this);
5   } else { requestPermissions(
6     newString[]{PERMISSION_POST_NOTIFICATION}, 0);}}

```

Listing 1: Unobfuscated reference.

Short Answer to RQ1: DEMASK significantly outperforms the existing tools in identifier deobfuscation. It not only achieves a broader recovery of semantics from obfuscated code but also produces more informative, higher-quality identifier names.

5.3 Component Contribution Analysis (RQ2)

To evaluate the core components of DEMASK, we conducted ablation experiments and TPL-mapping sensitivity analysis. The ablation experiments followed the RQ1 setup, using 60 snippets from different variants for LLM-Rater assessment and 15 snippet pairs for the user study. As shown in Table 2, removing any core component caused a clear performance drop. For TPL sensitivity analysis, we sampled 20 benign APKs and evaluated TPL mapping quality and its downstream impact under different τ . The subsequent evaluation used the same snippet scale as the ablation experiments, with snippets selected from developer-defined code that directly or indirectly invokes TPL methods.

Direct-LLM. We removed all optimization and enhancement phases and fed the obfuscated IR directly to the LLM for Java code generation with a prompt. This variant used the same model and temperature setting as DEMASK, and each snippet was generated once without cherry-picking. Performance dropped sharply: the LLM-Rater assessment showed declines of 58.8% in ERR and 50.0%

in BRR on average, while the user study reported similar reductions of 69.1% and 45.4%, respectively. These results suggest that without structured context from the FCG and semantic cues from TPLs, the LLM struggles to recover meaningful identifiers, often generating incorrect or overly generic names. Moreover, lacking knowledge of which components are third-party versus developer-defined forces the LLM to deobfuscate the entire APK and generate complete Java code, greatly increasing runtime and token consumption.

Without-FCG. We removed all FCG-related components and enhancements, retaining only the TPL-based identifier mapping to support semantic reasoning. All TPL-related methods were pre-filtered to allow the LLM to focus solely on deobfuscating developer-written code. The same prompt as in the Direct-LLM setup was used to generate Java source code.

While the TPL mapping offers partial semantic support, this TPL-only variant still performs substantially worse than the full DeMask. The LLM-Rater assessment showed drops of 26.1% in ERR and 44.6% in BRR, with the user study reporting similar declines. These results indicate that lacking callee context in the FCG significantly hampers the LLM’s ability to capture structural semantics and accurately infer identifier meanings. In terms of efficiency, this variant increased runtime by only 0.6 seconds and token usage by 0.4K, mainly due to the longer output code. The minimal overhead confirms that the TPL filtering mechanism effectively reduces redundant processing on known libraries, reinforcing its value in both accuracy and efficiency.

Without-TPL. We constructed a variant that removes the TPL mapping component and its enhancements, while retaining all FCG-related mechanisms. In this setup, the LLM performs global reasoning based on FCG semantics but lacks the ability to prune library code or leverage TPL context. The prompt instructs the model to output recovered identifiers with their semantic explanations.

This configuration achieved moderate recovery effectiveness. The average ERR in the LLM-Rater assessment dropped by 5.5%, and by 5.4% in the user study, indicating that FCG-based semantic reasoning alone can partially support identifier recovery. However, recovery quality degraded substantially: the average BRR decreased by 12.2% (LLM) and 17.6% (user study), highlighting the limited semantic accuracy when inferring library-related identifiers without TPL guidance. Moreover, the lack of TPL filtering significantly increased system overhead, with runtime rising by 68.5 seconds and token usage by 219.9K. These results confirm that TPL recognition improves recovery precision and reduces computational cost by avoiding redundant analysis of TPL code.

TPL-Mapping Sensitivity. To assess the downstream impact of TPL-mapping quality, we evaluated the LibPecker-recommended threshold ($\tau = 0.6$) together with more permissive and more stringent settings. We constructed the TPL ground truth by identifying included libraries from dependency metadata and recovering their original TPL identifiers via R8-generated obfuscation mappings. DeMask achieved 91.6% precision and 51.6% recall on TPL mapping, compared with 69.1% and 45.1% for Anti-ProGuard. As shown in Table 3, varying τ changed the precision-recall trade-off and affected downstream semantic recovery: false positives may inject incorrect TPL semantics into the LLM context, whereas lower recall reduces available TPL context and leaves more identifiers to be inferred from

Table 3: Results of TPL mapping sensitivity analysis.

Metric		$\tau=0.4$	$\tau=0.6$	$\tau=0.8$
TPL Quality	Precision	76.5%	91.6%	93.1%
	Recall	61.4%	51.6%	39.2%
LLM Rater Avg.	ERR	77.1%	86.9%	83.5%
	BRR	16.7%	54.8%	28.5%
User Study	ERR	70.4%	82.9%	74.7%
	BRR	21.6%	48.2%	30.2%

program context. The limited recall mainly reflects conservative filtering of version-mismatched or partially included libraries, as well as low-confidence or non-unique mappings. Overall, $\tau = 0.6$ achieved the best semantic recovery among the tested settings.

Short Answer to RQ2: The results show that both components are essential. The FCG module provides interprocedural call context for semantic reasoning. The TPL component restores library identifiers, improving accuracy while reducing cost through library-aware filtering and mapping. Together, they enable effective deobfuscation.

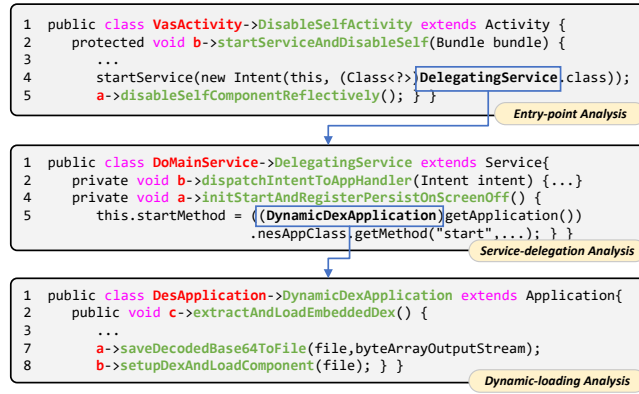
5.4 Impact on Malware Analysis (RQ3)

To assess the impact of DeMask on malicious app analysis, we applied it to the malicious app dataset comparing antivirus engine detections before and after deobfuscation. We also present a representative case study to illustrate the practical benefits of improved semantic recovery for security analysts. It is worth noting that, since AndroZoo [21] is closed-source, direct comparison with unobfuscated identifiers is not possible.

Malware Detection Results. To examine whether DeMask deobfuscation affects malware detection outcomes, we submitted the original malicious APKs to three multi-engine scanning platforms: VirusTotal [14], MetaDefender [10], and Hybrid-Analysis [7]. We recorded the number of engines that flagged each sample as malicious. After applying DeMask to deobfuscate, we resubmitted the samples for a direct comparison.

The results reveal a counterintuitive trend. After deobfuscation, detections drop substantially on all three platforms. On VirusTotal, the average number of detection engines dropped from 32 to 15; on MetaDefender, from 6 to 2, and on Hybrid-Analysis, from 6 to 1. This decrease may be partly attributed to the use of obfuscation-related features in malware classification. Some scanning engines rely on static signatures or machine learning models that treat meaningless or random identifiers as indicators of malicious intent [51]. When DeMask restores meaningful identifiers, these surface-level cues may be weakened, affecting detection outcomes. However, this observation should not be interpreted as strict causal evidence. Since DeMask rewrites and repackages APKs, the results may also be affected by signature changes, hash changes, DEX-layout modifications, or other packaging artifacts.

Case Study. To assess DeMask in practical malware analysis, we examined several deobfuscated malicious samples. The recovered semantic identifiers offered meaningful clues for understanding potential malicious behavior. Figure 7 presents a representative case that illustrates how DeMask enhances interpretability by enabling a clear trace from the entry point to the core logic.



Red: Obfuscated identifiers. Green: Identifiers deobfuscated by DeMask.

Figure 7: Malicious deobfuscation case (com.gnkr.xdwt).

In this case, the malware exhibits a clear three-stage workflow: it starts a service and then disables itself, delegates tasks to another service, and finally dynamically loads a malicious DEX payload. Facilitated by DEMASK's precise recovery of semantic identifiers, including the key method `extractAndLoadEmbeddedDex` and the class names `DisableSelfActivity` and `DelegatingService`, which explicitly reflect the extraction and loading of embedded malicious code, these stages become clear and greatly simplify the understanding of the malware's intent and behavior.

Short Answer to RQ3: DEMASK aids malware analysis by restoring meaningful identifiers that expose core malicious logic. This semantic clarity helps analysts locate and understand threats more efficiently. This demonstrates the practical value of DEMASK in real-world security analysis.

6 Discussion

This section discusses the practical limitations and considerations of DEMASK's current design and LLM-based approach.

Uncertainty of LLMs. DEMASK extracts rich context during processing, reducing the likelihood of incorrect name suggestions. However, as with any LLM-based approach, a residual risk remains. When the code logic is highly complex or the context is insufficient, the model may overinterpret partial evidence and produce plausible but incorrect identifiers, causing semantic drift. This reflects the inherent uncertainty of LLM generation under limited evidence.

Language and Tool Dependency. The current implementation of DEMASK relies on Java-specific features and tools. For example, LibPecker targets Java libraries and derives signatures from class dependencies. However, Kotlin introduces constructs like extension functions and companion objects that alter dependency structures after compilation and may cause signature mismatches. This lowers LibPecker's library matching coverage, leading to fewer pruned TPL methods and higher overhead in subsequent reasoning.

Callback Pruning Impact. Because Android relies heavily on callbacks, DEMASK may miss developer-defined methods accessible exclusively via TPL-rooted callback paths. An analysis of 10,000 pruned methods sampled from the benign dataset revealed that

only 238 (2.38%) were developer-defined. This result demonstrates that such omissions are relatively rare. In callback-intensive apps, however, these pruning errors can leave obfuscated identifiers unrecovered. These unresolved identifiers subsequently weaken downstream inference of identifiers for related classes or packages.

7 Related Work

Code Deobfuscation. Previous studies have proposed various approaches for recovering meaningful semantics from obfuscated code, most of which focus on scripting languages. Attias et al. [23] proposed a black-box deobfuscation system using search and inference rules for better semantic recovery. Noh et al. [45] introduced gMBA, a Transformer-based framework guided by semantic truth tables for improved deobfuscation. Chen et al. [29] developed JSimpo, combining symbolic and dynamic execution to recover JavaScript flow and data structures. Mariano et al. [43] proposed Chisel, a program-synthesis framework for reconstructing semantically equivalent code. Li et al. [38] introduced PowerPeeler, a dynamic PowerShell deobfuscation system tracking runtime execution. Li et al. [39] developed DeMinify, a deep learning method for variable recovery using type inference and identifier prediction. Li et al. [40] proposed a lightweight subtree-based deobfuscation system for PowerShell scripts. Tran et al. [48] presented JSNEAT, an IR-based method for restoring variable names using a JavaScript corpus. Vasilescu et al. [49] introduced Autonym, using statistical machine translation to recover identifiers from minified JavaScript. Unlike these approaches, DEMASK focuses on Android apps and integrates LLM-based semantic reasoning with static analysis. This enables it to recover meaningful identifiers and effectively reconstruct the semantic structure of obfuscated Android Java code.

LLM-driven Android App Analysis. Recent work leverages LLMs to enhance Android app analysis and automation. Guo et al. [33] propose VeriLog, combining graph reasoning and adaptive prompts to auto-generate release notes. Li et al. [37] introduced ForeDroid, an interpretable malware detector using scenario clustering and LLM summaries for anomaly detection. Alecci et al. [20] explored using LLMs for verifying Android privacy compliance by analyzing Smali code. Wen et al. [50] present AutoDroid, automating Android tasks with LLMs. Feng et al. [32] introduce AdbGPT, using few-shot learning for reproducible bug reports. Liu et al. [42] present InputBlaster, generating abnormal text inputs for crash detection. Liu et al. [41] also propose GPTDroid, converting GUI testing into question-answering tasks to improve coverage.

8 Conclusion

This paper presents DEMASK, a hybrid deobfuscation system that combines static program analysis with LLM-based reasoning to recover meaningful identifiers from obfuscated Android apps. DEMASK leverages call graph structures to guide a hierarchical, context-aware inference process and incorporates engineering techniques to improve scalability and naming consistency. Our prototype outperforms prior state-of-the-art approaches in both automated and human evaluations on benign and malicious apps. These results highlight the potential of combining program analysis with LLMs to enable accurate and scalable deobfuscation in real-world security settings.

Data Availability Statement

Our artifact and associated data are available at <https://doi.org/10.5281/zenodo.21359437>. Specifically, this repository contains the prototype of DeMASK, the datasets, the evaluation prompts, and the regular expressions and whitelist used for obfuscation detection. These prompts cover the Direct-LLM baseline, the generation of deobfuscated identifiers and descriptions for different identifier types, and the LLM-rater evaluation.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. This work was partially supported by the Natural Science Foundation of Shandong Province under Grants ZR2025MS991 and the Open Research Fund of the State Key Laboratory of Internet Architecture under Grant HLW2025MS29.

References

- [1] 2021. *DexGuard*. Retrieved October 12, 2025 from <https://www.guardsquare.com/dexguard>
- [2] 2024. *Trail: Learning the Java Language*. Retrieved October 2, 2025 from <https://docs.oracle.com/javase/tutorial/java/index.html>
- [3] 2025. *Androguard*. Retrieved February 25, 2025 from <https://github.com/androguard/androguard>
- [4] 2025. *Apktool*. Retrieved September 18, 2025 from <https://github.com/iBotPeaches/apktool>
- [5] 2025. *DashO*. Retrieved August 18, 2025 from <https://www.preemptive.com/products/dasho>
- [6] 2025. *F-droid*. Retrieved September 29, 2025 from <https://f-droid.org>
- [7] 2025. *Hybrid Analysis*. Retrieved September 28, 2025 from <https://hybrid-analysis.com>
- [8] 2025. *Jadx*. Retrieved September 9, 2025 from <https://github.com/skylot/jadx>
- [9] 2025. *Jeb*. Retrieved September 10, 2025 from <https://www.pnfsoftware.com/jeb>
- [10] 2025. *Metadefender*. Retrieved July 16, 2025 from <https://metadefender.com>
- [11] 2025. *Simple Android app for a shopping list*. Retrieved July 21, 2025 from <https://github.com/domob1812/shopt2>
- [12] 2025. *Statistical Deobfuscation for Android*. Retrieved September 9, 2025 from <http://apk-deguard.com>
- [13] 2025. *Tiktoken*. Retrieved August 9, 2025 from <https://pypi.org/project/tiktoken>
- [14] 2025. *Virustotal*. Retrieved August 11, 2025 from <https://www.virustotal.com>
- [15] 2026. *Androidmeda*. Retrieved January 20, 2026 from <https://github.com/In3tinct/Androidmeda/>
- [16] 2026. *Enable app optimization*. Retrieved January 5, 2026 from <https://developer.android.com/studio/build/shrink-code>
- [17] 2026. *Introducing GPT-5.4-mini and nano*. Retrieved March 17, 2026 from <https://openai.com/index/introducing-gpt-5-4-mini-and-nano/>
- [18] 2026. *LLM Pricing Comparison*. Retrieved July 13, 2026 from <https://openai.com/api/pricing>
- [19] 2026. *R8*. Retrieved January 5, 2026 from <https://r8.googlesource.com/r8>
- [20] Marco Alecci, Nicolas Sannier, Marcello Ceci, Sallam Abualhaija, Jordan Samhi, Domenico Bianculli, Tegawendé F. Bissyandé, and Jacques Klein. 2025. Toward LLM-Driven GDPR Compliance Checking for Android Apps. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion)*, Trondheim, Norway, June 23–28, 2025. doi:10.1145/3696630.3728508
- [21] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, Austin, TX, USA, May 14–22, 2016. doi:10.1145/2901739.2903508
- [22] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, United Kingdom, June 09–11, 2014. doi:10.1145/2594291.2594299
- [23] Vidal Attias, Nicolas Bellec, Grégoire Menguy, Sébastien Bardin, and Jean-Yves Marion. 2025. Augmenting Search-based Program Synthesis with Local Inference Rules to Improve Black-box Deobfuscation. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Taipei, Taiwan, October 13–17, 2025. doi:10.1145/3719027.3765134
- [24] Richard Baumann, Mykolai Protzenko, and Tilo Müller. 2017. Anti-ProGuard: Towards Automated Deobfuscation of Android Apps. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems (SHCIS)*, Neuchâtel, Switzerland, June 21–22, 2017. doi:10.1145/3099012.3099020
- [25] David Beste, Grégoire Menguy, Hossein Hajipour, Mario Fritz, Antonio Emanuele Cinà, Sébastien Bardin, Thorsten Holz, Thorsten Eisenhofer, and Lea Schönherr. 2025. Exploring the Potential of LLMs for Code Deobfuscation. In *Proceedings of the 22nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Graz, Austria, July 9–11, 2025. doi:10.1007/978-3-031-97620-9_15
- [26] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 24–28, 2016. doi:10.1145/2976749.2978422
- [27] Guoqiang Chen, Xin Jin, and Zhiqiang Lin. 2025. JsDeObsBench: Measuring and Benchmarking LLMs for JavaScript Deobfuscation. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Taipei, Taiwan, October 13–17, 2025. doi:10.1145/3719027.3744871
- [28] Menglong Chen, Tian Tan, Minxue Pan, and Yue Li. 2025. PacDroid: A Pointer-Analysis-Centric Framework for Security Vulnerabilities in Android Apps. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, Ottawa, ON, Canada, April 26–May 6, 2025. doi:10.1109/ICSE55347.2025.00208
- [29] Tianyu Chen, Ding Li, Ying Zhang, and Tao Xie. 2025. JSimp: Structural Deobfuscation of JavaScript Programs. *ACM Trans. Softw. Eng. Methodol.* 34, 7 (2025), 197:1–197:28. doi:10.1145/3714460
- [30] Byunggeon Choi, Hongjoo Jin, Dong Hoon Lee, and Wonsuk Choi. 2024. ChatDEOB: An Effective Deobfuscation Method Based on Large Language Model. In *Proceedings of the 25th International Conference on Information Security Applications (WISA)*, Jeju Island, South Korea, August 21–23, 2024. doi:10.1007/978-981-96-1624-4_12
- [31] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In *Proceedings of the 14th International Conference on Security and Privacy in Communication Networks (SecureComm)*, Singapore, August 8–10, 2018. doi:10.1007/978-3-030-01701-9_10
- [32] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, Lisbon, Portugal, April 14–20, 2024. doi:10.1145/3597503.3608137
- [33] Jiawei Guo, Haoran Yang, and Haipeng Cai. 2025. VerLog: Enhancing Release Note Generation for Android Apps using Large Language Models. *Proc. ACM Softw. Eng.* 2, ISSTA (2025), 1910–1932. doi:10.1145/3728961
- [34] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. 2024. SootUp: A Redesign of the Soot Static Analysis Framework. In *Proceedings of 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Luxembourg City, Luxembourg, April 6–11, 2024. doi:10.1007/978-3-031-57246-3_13
- [35] Ulf Kargén, Noah Mauthe, and Nahid Shahmehri. 2023. Characterizing the Use of Code Obfuscation in Malicious and Benign Android Apps. In *Proceedings of the 18th International Conference on Availability, Reliability and Security (ARES)*, Benevento, Italy, 29 August – 1 September. doi:10.1145/3600160.3600194
- [36] Marie-Anne Lachaux, Baptiste Rozière, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6–14, 2021, virtual*.
- [37] Jiaming Li, Sen Chen, Chunlian Wu, Yuxin Zhang, and Lingling Fan. 2025. ForeDroid: Scenario-Aware Analysis for Android Malware Detection and Explanation. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Taipei, Taiwan, October 13–17, 2025. doi:10.1145/3719027.3765207
- [38] Ruijie Li, Chenyang Zhang, Huajun Chai, Lingyun Ying, Haixin Duan, and Jun Tao. 2024. PowerPeeler: A Precise and General Dynamic Deobfuscation Method for PowerShell Scripts. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Salt Lake City, UT, USA, October 14–18, 2024. doi:10.1145/3658644.3670310
- [39] Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N. Nguyen. 2023. DeMinify: Neural Variable Name Recovery and Type Inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, San Francisco, CA, USA, December 3–9, 2023. doi:10.1145/3611643.3616368
- [40] Zhenyuan Li, Qi Alfred Chen, Chunlin Xiong, Yan Chen, Tiantian Zhu, and Hai Yang. 2019. Effective and Light-Weight Deobfuscation and Semantic-Aware Attack Detection for PowerShell Scripts. In *Proceedings of the 2019 ACM*

- SIGSAC Conference on Computer and Communications Security (CCS)*, London, UK, November 11–15, 2019. doi:10.1145/3319535.3363187
- [41] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, Lisbon, Portugal, April 14–20, 2024. doi:10.1145/3597503.3639180
- [42] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Zhilin Tian, Yuekai Huang, Jun Hu, and Qing Wang. 2024. Testing the Limits: Unusual Text Inputs Generation for Mobile App Crash Detection with Large Language Model. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, Lisbon, Portugal, April 14–20, 2024. doi:10.1145/3597503.3639118
- [43] Benjamin Mariano, Ziteng Wang, Shankara Pailoor, Christian S. Collberg, and Isil Dillig. 2024. Control-Flow Deobfuscation using Trace-Informed Compositional Program Synthesis. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 2211–2241. doi:10.1145/3689789
- [44] Daye Nam, Andrew Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, Lisbon, Portugal, April 14–20, 2024. doi:10.1145/3597503.3639187
- [45] Youjeong Roh, Joon-Young Paik, Jingun Kwon, and Eun-Sun Cho. 2025. gMBA: Expression Semantic Guided Mixed Boolean-Arithmetic Deobfuscation Using Transformer Architectures. In *Findings of the 2025 Association for Computational Linguistics (ACL)*, Vienna, Austria, July 27 – August 1, 2025. doi:10.18653/v1/2025.FINDINGS-ACL.1127
- [46] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, USA, July 17–21, 2023. doi:10.1145/3597926.3598120
- [47] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. doi:10.1137/0201010
- [48] Hieu Tran, Ngoc M. Tran, Son Nguyen, Hoan Nguyen, and Tien N. Nguyen. 2019. Recovering Variable Names for Minified Code with Usage Contexts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, May 25–31, 2019. doi:10.1109/ICSE.2019.00119
- [49] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar T. Devanbu. 2017. Recovering Clear, Natural Identifiers from Obfuscated JS Names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, Paderborn, Germany, September 4–8, 2017. doi:10.1145/3106237.3106289
- [50] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. AutoDroid: LLM-powered Task Automation in Android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom)*, Washington D.C., DC, USA, November 18–22, 2024. doi:10.1145/3636534.3649379
- [51] Junji Wu and Atsushi Kanai. 2021. Utilizing Obfuscation Information in Deep Learning-based Android Malware Detection. In *Proceedings of the 45th IEEE Annual Computers, Software, and Applications Conference (COMPSAC)*, Madrid, Spain, July 12–16, 2021. doi:10.1109/COMPSAC51774.2021.00185
- [52] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zheming Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso, Italy, March 20–23, 2018. doi:10.1109/SANER.2018.8330204
- [53] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *Proc. ACM Softw. Eng.* 2, ISSTA (2025), 481–503. doi:10.1145/3728894

Received 2026-03-26; accepted 2026-06-18