ForeDroid: Scenario-Aware Analysis for Android Malware Detection and Explanation

Jiaming Li*
College of Intelligence and Computing, Tianjin University
Tianjin, China
jiamingli@tju.edu.cn

Sen Chen*†
College of Cryptology and Cyber Science, Nankai University
Tianjin, China
senchen@nankai.edu.cn

Chunlian Wu
College of Intelligence and Computing, Tianjin University
Tianjin, China
chunlian@tju.edu.cn

Yuxin Zhang
College of Intelligence and Computing, Tianjin University
Tianjin, China
yuxinzhang@tju.edu.cn

Lingling Fan
College of Cryptology and Cyber Science, Nankai University
Tianjin, China
linglingfan@nankai.edu.cn

Abstract

Android malware continues to evolve, posing significant challenges in generalization, fine-grained detection, and interpretability for existing detection systems. Existing methods struggle to generalize to unseen malware, lack fine-grained behavioral understanding, and provide limited interpretability due to their reliance on rigid rules or the inability to recover complete causal behavior paths. To this end, we present `ForeDroid`, a unified and interpretable framework for Android malware detection and explanation via scenario-aware analysis. `ForeDroid` models malicious intent as behavioral inconsistencies within functional scenarios. It clusters semantically coherent scenarios, extracts sensitive API call chains, and summarizes them into natural language using LLMs. These summaries are embedded and compared against benign behavior distributions within the same scenario for unsupervised anomaly detection. High-risk behaviors showing strong semantic inconsistency are further interpreted by an LLM-driven module that generates fine-grained anomaly reports. We evaluated `ForeDroid` on two challenging tasks: zero-day malware detection and fine-grained behavior analysis. The result shows `ForeDroid` outperforms MaMaDroid, MalScan, DeepRefiner, and a continuous learning-based approach in zero-day malware detection under the temporal-split setting. Besides, `ForeDroid` achieves an F1-score of 0.94 in fine-grained behavior detection on the manually annotated GPMalware dataset, surpassing ProMal. Our results demonstrate `ForeDroid`'s ability to bridge low-level call graph analysis with high-level semantic reasoning, making it a practical, interpretable solution for malware detection.

*Sen Chen and Jiaming Li are co-first authors and contributed equally to this work.
†Sen Chen is the corresponding author of this paper.

CCS '25, Taipei

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765207

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Android Security; Context-Aware Analysis; Scenario Modeling; Explainable Malware Detection

ACM Reference Format:
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 '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765207

1 Introduction

Android malware, specifically crafted to compromise Android devices, has long posed a serious security threat to millions of users worldwide. Given the dominant position of the Android operating system in the mobile ecosystem, the research community has made sustained efforts to enhance malware detection capabilities [2, 3, 6, 7, 15, 17, 18, 32, 38–40, 44, 45]. Early approaches mainly relied on syntactic static features, such as frequency of sensitive API usage [2, 7, 9, 12], or permission requests [6]. While effective to a degree, these methods are vulnerable to code obfuscation and lack semantic understanding. To overcome these limitations, recent research has shifted toward behavior semantics modeling, including context-aware analysis [3, 44] and graph-based representations [25, 40, 45], aiming to capture richer execution semantics and behavioral context. However, they still face three key limitations: (1) many rely on handcrafted rules or rigid pattern matching, which hampers scalability and introduces noise; (2) they often overfit to specific training distributions, limiting robustness against novel or zero-day malware; and (3) most operate at the coarse app level, lacking the granularity to detect and explain malicious behavior at the function or call-chain level. Meanwhile, explainability techniques have been explored to improve malware detection transparency. Explainable artificial intelligence (XAI) methods such as Drebin [7], LIME [34], LEMNA [21], and XMal [38] highlight influential features used in decisions. ProMal [39] further constructs

a behavior knowledge graph (BxKG) to capture malicious execution trajectories. However, XAI methods cannot recover full causal paths, while graph-based approaches require extensive manual annotation, hindering adaptability and scalability in real-world settings. These issues underscore the need for a more generalizable, fine-grained, and interpretable behavior modeling framework.

To this end, we propose ForeDroid, a scenario-aware behavior analysis framework for automated, fine-grained, and interpretable Android malware detection. The name ForeDroid reflects its dual goals: proactive foresight and interpretable forensics in Android malware analysis. Our design is inspired by the observation that benign and malicious applications (apps) follow fundamentally different design philosophies-where benign apps implement functionalities aligned with user expectations, while malicious apps often violate scenario expectations to perform covert or harmful operations. For example, sending an SMS is expected during explicit user interaction in a messaging scenario, but doing so silently in a background update context violates this expectation and suggests malicious intent. Therefore, we define such behavioral inconsistencies within functional scenarios as a core signal of malicious intent. ForeDroid aims to automatically detect and explain such inconsistencies without relying on labeled malware samples. Despite the motivation, building such a system involves several technical challenges: **C1**: *Scenario Modeling*. Android apps are often composed of multiple functionalities, each associated with distinct entry points triggered by various UI interactions (e.g., button clicks, text input) or system events (e.g., broadcasts, lifecycle callbacks). Accurately partitioning these diverse and implicit entry points into semantically coherent functional scenarios is difficult due to the absence of a centralized "main" entry point and the diversity of application designs. **C2**: *Sensitive Behavior Representation*. Sensitive behaviors often span asynchronous invocations and inter-component communication (ICC), making them hard to be fully captured using traditional static tools like FlowDroid [8]. Additionally, diverse coding styles, obfuscation, and inconsistent naming conventions further complicate the extraction of robust semantic representations. **C3**: *Fine-Grained Detection*. Effectively identifying malicious behaviors at the level of API call chains, instead of relying on coarse-grained app-level classification, poses significant challenges. Such fine-grained detection requires robust semantic modeling and large-scale labeled data, which are often difficult to obtain in practice. Moreover, malicious behaviors frequently evolve and manifest in novel ways, making it difficult for supervised methods to generalize, especially in zero-day scenarios. **C4**: *Behavior Explanation Challenge*. Automatically explaining suspicious behaviors requires bridging the semantic gap between low-level execution codes and high-level behavioral intent. This involves identifying anomalous behaviors, interpreting their context, and reasoning about malicious intent—tasks that typically demand substantial expert knowledge. Generating accurate and sound explanations in a fully automated and scalable manner remains a challenge.

To address these challenges, ForeDroid integrates key components into a unified, interpretable detection pipeline, guided by the following insights: To tackle **C1** , ForeDroid clusters GUI- and system-triggered entry points based on semantic contexts (e.g., GUI text, intent actions), enabling scenario-aware partitioning of behaviors. For **C2**, ForeDroid reconstructs complete paths from entry points to sensitive APIs and abstracts them into intent-expressive summaries via LLMs, ensuring robustness against obfuscation, naming inconsistency, and code variability. To handle **C3**, ForeDroid learns scenario-specific benign behavior distributions and detects anomalies via unsupervised similarity scoring and One-Class SVM classification, supporting label-free, zero-day detection. To address **C4**, ForeDroid employs LLM-based reasoning over scenario context and call chains to generate structured analyst-style explanations for behaviors with high anomaly scores. Based on these insights, ForeDroid combines functional scenario modeling, sensitive behavior representation, and large-scale benign behavior distributions to enable unsupervised detection and explanation. It detects abnormal behaviors by comparing semantic behavior representations against benign patterns, without relying on labeled malware. While malicious apps may exhibit unseen behaviors, our approach generalizes well by grounding analysis in functional intent and behavioral consistency. By leveraging LLMs for both behavior representation and explanation, ForeDroid enhances fine-grained detection and enables interpretable analysis for decision-making.

We evaluate ForeDroid with a focus on its capability to detect zero-day Android malware. Under both *temporal* (i.e., training on older samples and testing on newer ones) and *family-based* (i.e., training on known malware families and testing on previously unseen ones) split settings, ForeDroid achieves an F1-score of 0.886 (temporal) and 0.893 (family-based). ForeDroid surpasses state-of-the-art baselines, including MaMaDroid [25] (F1-score: 0.453 temporal, 0.198 family-based), MalScan [40] (0.399 temporal, 0.273 family-based), DeepRefiner [42] (0.445 temporal, 0.337 family-based), and the continuous learning–based approach [13] (0.735 temporal). In terms of F1-score under the temporal-split setting, ForeDroid delivers improvements of up to 95.6%, 122.1%, 99.1%, and 20.5% over these methods, respectively. These results highlight ForeDroid 's strong generalization to previously unseen malware. Moreover, ForeDroid demonstrates strong performance in fine-grained behavior detection. On 102 manually labeled samples, it correctly detects 652 malicious behavior instances with an F1-score of 0.94, exceeding the performance of ProMal (634 behaviors, F1-score: 0.93). Unlike prior systems, ForeDroid achieves this through a fully automated pipeline without requiring manual rule construction or annotations. We also conduct an ablation study to assess the contribution of each core component of ForeDroid.

In summary, the paper makes the following major contributions:

- We present ForeDroid, a unified and automated framework for Android malware detection and explanation, built on the insight that behavioral inconsistencies within functional scenarios strongly indicate malicious intent.
- We construct a large-scale, representative corpus of scenario-behavior aligned reference library, enabling robust modeling of legitimate behavior across diverse functional scenarios.
- We conduct extensive evaluations on 41,665 Android apps, where ForeDroid achieves high zero-day detection accuracy under both temporal and family-based splits. Additionally, we further assess its fine-grained detection capability by decomposing coarse-grained malicious payloads into 25 detailed behavior types, achieving superior accuracy and interpretability compared to state-of-the-art baselines.

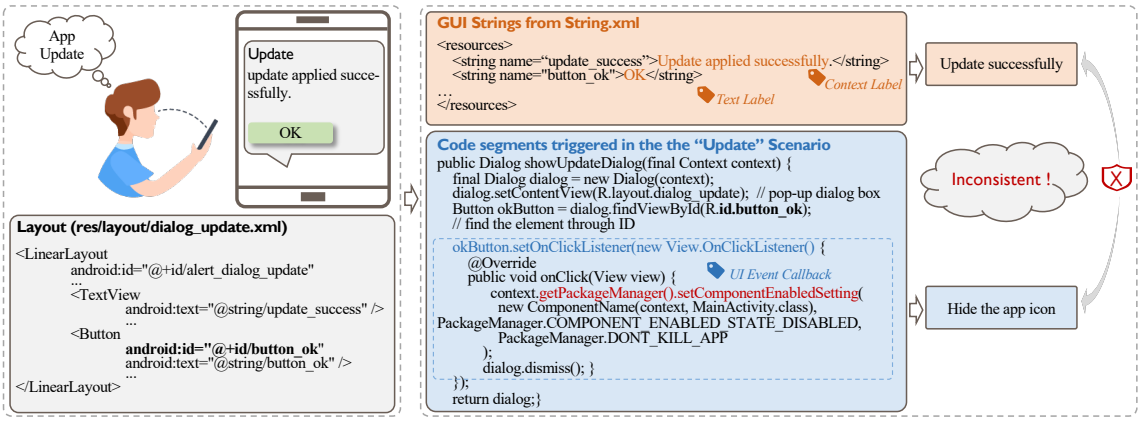


Figure 1: An example of behavioral inconsistency within the functional scenario about app update.

- We release `ForeDroid` on GitHub [29], along with the behavior-level detection results and auto-generated forensic reports for GPMalware samples, to support reproducibility and future research in interpretable Android malware analysis.

2 Background

2.1 Sensitive API

A sensitive API in the Android environment refers to any application programming interface (API) that accesses critical system resources or user data, such as device identifiers, location, camera, or system settings. These APIs may pose privacy or security risks [7, 16], necessitating special attention in analyzing app behaviors for malicious intent. In this work, we adopt the sensitive API set defined by MalScan [40], which covers APIs related to personal data access, communication, system control, and other security-critical operations. Specifically, we filter and retain only the security-relevant call chains and functional scenarios for downstream analysis based on the reachability of sensitive APIs from entry points. By removing benign-only or irrelevant logic, our analysis can focus on the execution paths that are critical to security.

2.2 Behavior Triggering Mechanisms

Android apps follow a component-based, event-driven architecture, where execution logic is driven by various user- or system-triggered events. Specifically, app behaviors implemented by API calls are typically initiated via two major types of triggering mechanisms:

- **GUI Interaction Triggers**: Behaviors explicitly triggered by user interaction with the graphical user interface (GUI), implemented through GUI-associated callbacks, such as `onClick()`, `onItemSelected()`, and other UI-bound handlers. Such callbacks are usually registered via XML layout attributes (e.g., `android:onClick`) or programmatically using event listener interfaces (e.g., `setOnClickListener()`).
- **Background Callback Triggers**: Behaviors implicitly triggered by the Android system in response to environmental changes, such as incoming SMS messages, boot completion, or component lifecycle transitions. These are handled by background callbacks like `onReceive()` in `BroadcastReceiver`, `onStartCommand()` in `Service`, or `onCreate()` in `Activity`.

This categorization of behavior triggers reflects a fundamental design pattern in Android, forming the foundation for our functional scenario modeling (Section 3.2). By distinguishing behaviors based on their triggering mechanism, we can more accurately model their execution context and identify potential malicious intent. To support behavior analysis, we take *GUI-associated callbacks* and *background-triggered callbacks* introduced above as ***entry points***, which serve as the starting nodes for subsequent functional scenario modeling and the call chain extraction of sensitive behavior representation in our approach.

2.3 Motivating Example

As highlighted in the introduction, benign apps typically behave as expected, whereas malicious ones violate functional intent for covert or harmful purposes. Figure 1 illustrates an instance of such behavioral inconsistency within a specific functional scenario. The scenario shows a situation where the app has been updated successfully, as indicated by the confirmation message ("Update applied successfully"), from which we can infer that the scenario is about an app update-information that is also conveyed to the user through GUI. However, clicking the "OK" button in this context triggers the invocation of `setComponentEnabledSetting()`, resulting in an unexpected behavior, i.e., hiding the app icon from the device's home screen. Hiding the icon is a common stealth strategy used by malware to evade user detection and hinder uninstallation or further inspection. The discrepancy between the expected behavior ("Update successfully") and the unexpected behavior ("Hide the app icon") that is silently triggered reveals a semantic mismatch within the current scenario. By identifying such mismatches, we can uncover potential anomalous or malicious behaviors that deviate from the app's intended function.

3 Approach

To address the limitations of existing methods in generalization, fine-grained behavior detection, and interpretability, we propose

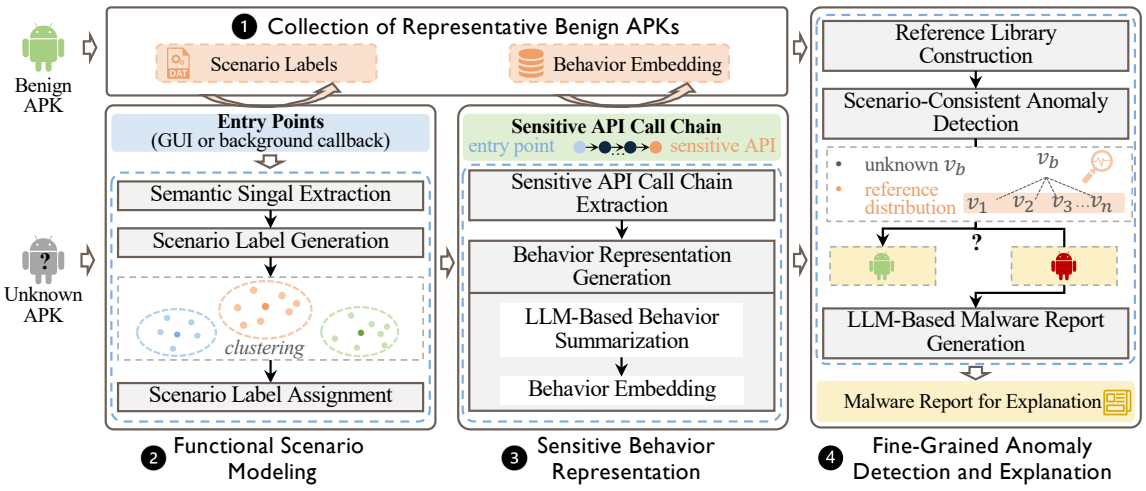


Figure 2: Overview of ForeDroid.

ForeDroid, a fully automated and interpretable system for analyzing malicious behaviors in Android apps. ForeDroid takes a single unknown APK file as input and outputs both a malware detection result and a structured malware report for the explanation of abnormal behaviors. Internally, it relies on a pre-constructed scenario-behavior aligned reference library built using benign APKs, which enables unsupervised and scenario-aware anomaly detection without requiring any labeled malware samples. As illustrated in Figure 2, ForeDroid consists of four main stages.

(1) **Collection of Representative Benign APKs,** which collects high-quality benign APKs from Google Play based on download volume and category balance, serves as a foundation for building a scenario-aligned reference library of benign behaviors for subsequent modeling and detection.
(2) **Functional Scenario Modeling,** which partitions app behaviors into semantically coherent scenarios based on the semantic signals of their triggering entry points, captures scenario labels that reflect the functional intent behind both GUI-triggered and background-triggered behaviors of apps.
(3) **Sensitive Behavior Representation,** which abstracts sensitive API call chains into semantically meaningful summaries using LLM, enables representation generation of scenario-specific behaviors by preserving execution context while reducing reliance on low-level implementation details.
(4) **Fine-Grained Anomaly Detection and Explanation,** which detects semantically deviant behaviors within specific functional scenarios and generates structured LLM-based reports, enabling accurate and explainable detection of malicious behaviors without relying on labeled training data.

This modular design enables our system to go beyond traditional permission- or API-level detection by modeling behaviors in scenario context, ensuring fine-grained accuracy and interpretable outputs for real-world malware analysis.

3.1 Collection of Representative Benign APKs

We collected benign APKs from the Google Play Store via the AndroZoo platform, focusing on apps published between 2015 and 2025. Google Play was chosen as the data source because it covers 49 functional categories (e.g., social, tools, games) and requires apps to pass security audits. To ensure dataset comprehensiveness and representativeness, we applied the following criteria: (1) *Download Volume Priority*: Only APKs with $\geq$ 5,000 downloads were selected to ensure active usage and diverse scenarios. (2) *Balanced Category Coverage*: We selected the top 600 APKs from each of the 49 categories, with all available apps from underrepresented categories like COMICS and EVENTS. After removing duplicates, filtering invalid APKs, and excluding 202 apps with incorrect category labels, we obtained 28,994 high-quality benign apps. This dataset forms the basis of a scenario-behavior aligned reference library, which is further constructed through the functional scenario modeling and sensitive behavior representation processes in subsequent stages.

3.2 Functional Scenario Modeling

To detect behavioral inconsistencies, we first need to understand the intended functional purpose under a given scenario. We posit that the functional intent of behaviors can be inferred from their triggering entry points based on the following insights, allowing us to partition them into semantically coherent functional scenarios.
Insight 1: GUI-triggered behaviors reflect functional intent through UI elements associated with entry points. In user-oriented scenarios, behaviors are typically triggered by explicit user actions, such as button clicks or text input. The semantics of these actions are often revealed in surrounding GUI texts (e.g., dialog messages, button captions). For example, an API call triggered by a button labeled "Send Message" intuitively suggests message-related functionality.
Insight 2: Background-triggered behaviors should align with expected benign behaviors. In background-triggered scenarios, behaviors are implicitly invoked via system events (e.g., BOOT_COMPLETED, SMS_RECEIVED) or lifecycle callbacks (e.g., onCreate(), onReceive()). In such cases, the execution context is defined by the semantics of

the corresponding Android system event or callback. This insight is supported by prior work [3], which shows that malicious apps are 6.89 times more likely than benign ones to perform sensitive operations such as HTTP requests at the same background callback entry point (e.g., `Service.onCreate()`). Such behavioral discrepancies highlight the strong indicative power of entry-point semantics in distinguishing malicious intent in system-invoked contexts.

Based on these insights, we organize functional scenario modeling into two parallel pipelines: (1) *GUI-Associated Scenario Modeling* and (2) *Background Callback-Associated Scenario Modeling*. Both pipelines adopt a shared three-step methodology: semantic signal extraction, scenario label generation, and scenario label assignment. To ensure the security relevance of modeled scenarios, we focus on entry points from which sensitive APIs are reachable within the call graph, as these points directly determine the functional contexts under which privacy-critical behaviors may be executed.

3.2.1 ***GUI-Associated Scenario Modeling.*** **Step 1: Semantic Signal Extraction from GUI Entry Points.** To extract meaningful semantic cues from GUI entry points, we leverage Backstage [23] for precise callback resolution and layout parsing, which allows us to map UI elements to their triggering methods and retrieve surrounding textual content (e.g., "update applied successfully" in Figure 1). In addition, we integrate OCR-based icon text extraction using Tesseract [37] to capture embedded labels in visual assets. So we use Backstage and Tesseract to extract three types of information for each UI element :

- **UI Callback:** The callback method that responds to the UI event (e.g., `onClick()` handler).
- **Text Label:** The textual content of the UI element, including both standard widget text and icon-derived text (e.g., "OK").
- **Context Label:** Nearby textual contents parsed from the layout, specifically the elements that share a common container (e.g., "update applied successfully").

We treat this callback method as the GUI entry point, while the text and context labels serve as its semantic signals.

Step 2: Scenario Label Generation from GUI Entry Points. After extracting semantic signals from GUI entry points, our goal is to cluster semantically similar entry points into coherent groups that represent distinct functional scenarios (e.g., login, register, share). These scenario labels serve as semantic anchors for grouping behaviors in downstream analysis. To achieve this, we follow a three-phase process: (1) textual preprocessing to preserve key semantics, (2) UI semantic embedding using BERT [14], and (3) unsupervised clustering with HDBSCAN [26].

(1) Textual Preprocessing. We perform lightweight NLP preprocessing on both text labels and context labels, focusing on preserving crucial semantic cues. This includes lemmatization and the removal of non-informative tokens such as punctuation and numeric values. Unlike general NLP tasks, we deliberately preserve prepositions and auxiliary verbs because they carry crucial functional meanings in GUI contexts (e.g., "log in" ≠ "log out").

(2) UI Semantic Embedding. For each GUI entry point e, we concatenate its text label and context label, and generate a semantic embedding using BERT:

$$v_e = \text{BERT}(\text{text label} \parallel \text{context label}) \tag{1}$$

Here, $\parallel$ denotes the concatenation of the two input strings before embedding. This encoding captures both local text semantics and broader context, making it suitable for short but semantically rich GUI texts. For entry points lacking GUI text are skipped in this process and are handled in the Background Callback-Associated Scenario Modeling pipeline (see Section 3.2.2).

(3) Unsupervised Clustering. We apply HDBSCAN, a density-based clustering algorithm, to group the semantic embeddings of GUI entry points into semantically coherent clusters. HDBSCAN automatically infers the number of clusters and is robust to noise, suiting the noisy and inconsistent naming patterns in real-world apps. Each resulting cluster represents a distinct functional scenario. For each cluster c_i, we compute its centroid vector v_{c_i}, and select the UI element closest to the centroid to serve as the human-readable scenario label l_{c_i}. By applying these processes to a large corpus of benign apps collected in Section 3.1, we construct approximately 800 functional scenario clusters and corresponding scenario labels, covering common user intents such as login, register, pay, navigate, and share. These scenario clusters form the conceptual foundation for downstream behavior grouping and scenario consistency analysis. Representative examples and lexical distributions of these clusters are provided in Appendix A.

Step 3: Scenario Label Assignment from GUI Entry Points. After constructing the reference scenario clusters from the collected benign APKs, we assign scenario labels to GUI entry points in unknown apps through two key steps: (1) semantic embedding of each new entry point, and (2) label assignment by comparing the embedding with pre-clustered scenario centroids.

(1) Semantic Embedding. Given an entry point e (associated with a UI callback method), we extract its corresponding text label and context label (if any) as Step 1, and compute its semantic embedding v_e using the same BERT encoder as in Step 2.

(2) Label Assignment. Let v_{c_i} denote the centroid vector of scenario cluster c_i, and its corresponding label be l_i. We compute the cosine similarity between v_e and each v_{c_i}:

$$\text{sim}(e, c_i) = \frac{v_e \cdot v_{c_i}}{\|v_e\| \|v_{c_i}\|} \tag{2}$$

We then assign the entry point e to the label l^* corresponding to the most similar cluster c^* :

$$\begin{aligned} c^* &= \arg\max_{c_i \in C} \text{sim}(e, c_i) \\ l^* &= l_{c^*} \end{aligned} \tag{3}$$

This step ensures consistent scenario-aware behavior grouping across unlabeled apps, essential for downstream anomaly detection.

3.2.2 ***Background Callback-Associated Scenario Modeling.*** **Step 1: Semantic Signal Extraction from Background Callback Entry Points.** Background behaviors are implicitly triggered by system events or lifecycle transitions. Unlike GUI-driven scenarios that expose rich semantic cues through visible UI elements, these entry points lack direct textual context, but their functional semantics are often revealed in the associated callback methods and intent actions. Specifically, for each entry point e_b identified as a background-triggered callback, such as `onCreate()`, `onStartCommand()`, and `onReceive()`, we extract the following semantic signals:

- **Callback Signature:** We use the method name (e.g., `onReceive()`) along with the enclosing component class (e.g., `MessageReceiver`) to form a complete signature (e.g., `MessageReceiver.onReceive()`). This signature reflects the lifecycle or system event context where the behavior executes, indicating its functional intent.
- **Intent Action (if any):** For components such as `Service` or `BroadcastReceiver`, the behavior is often triggered by Android system actions (e.g., `SMS_RECEIVED`, `BOOT_COMPLETED`). We extract these intent actions from both statically declared `intent-filters` in the AndroidManifest.xml and dynamically registered receivers via `registerReceiver()` calls. Intent actions provide explicit semantic labels that describe the external event responsible for triggering the entry point.

The two categories of semantic signals together capture the context of background behaviors, including the lifecycle position and external events, enabling effective scenario modeling even in the absence of GUI-level semantics.

Step 2: Scenario Label Generation from Background Callback Entry Points. Similar to the GUI-associated scenario modeling pipeline, this process involves the following three phases:

(1) Text Preprocessing. Unlike GUI texts, background callback strings (e.g., `MessageReceiver.onReceive()`, `SMS_RECEIVED`) often lack clear word boundaries and exhibit higher lexical redundancy. To enhance semantic granularity and enable meaningful clustering, we apply two additional preprocessing steps beyond those used in GUI modeling: (1) *Camel-Case Splitting*: Compound identifiers such as `onBatteryLow` are split into individual tokens ("on Battery Low") to improve token representation. (2) *Preserving Domain-Specific Tokens*: We retain Android-specific terms such as "intent", "receiver", and common intent actions to preserve key semantic cues otherwise discarded by general-purpose preprocessing.

(2) Background Semantic Embedding. After preprocessing, we convert each background callback entry point's semantic signals (i.e., the concatenation of callback signature and intent action) into a semantic vector. Compared to GUI texts which are typically short and user-oriented, background semantic signals often contain templated, repetitive, and technical phrases. To ensure scalability across large datasets and minimize computational overhead, we adopt Word2Vec [27] instead of BERT here. Formally, let a preprocessed semantic signal be represented as a token sequence $e_b = \{t_1, t_2, \ldots, t_n\}$. Its embedding vector is computed as the average of the corresponding token embeddings:

$$v_{e_b} = \frac{1}{n} \sum_{i=1}^{n} \text{Word2Vec}(t_i) \tag{4}$$

(3) Unsupervised Clustering. To improve clustering effectiveness and suppress noise, we apply Principal Component Analysis (PCA) [22] for dimensionality reduction. We retain the top 100 principal components, which preserve over 90% of the variance in the original embedding space. We then apply HDBSCAN over the reduced embeddings to form clusters of semantically related background callbacks. This step is consistent with the GUI scenario modeling described in Section 3.2.1. However, due to the higher volume of background triggers across apps, the resulting number of clusters is substantially larger. In our implementation, this process typically yields around 2,000 distinct scenario labels.

Step 3: Scenario Label Assignment from Background Callback Entry Points. We follow the same labeling pipeline used in GUI-based scenario modeling (Section 3.2.1). Given a new background entry point e_b of unknown APKs, we construct its semantic representation using the same preprocessing and embedding approach described in Step 2. We then assign e_b to the scenario label whose cluster centroid is most similar, using cosine similarity as defined in Equation (2), and select the top score as in Equation (3).

3.3 Sensitive Behavior Representation

After partitioning entry points into distinct functional scenarios, the next step is to abstract the sensitive behaviors exhibited under each scenario. Accurate behavior representation is essential for fine-grained anomaly detection. However, modeling sensitive behaviors in Android presents several technical challenges. Permission-based analysis, although widely adopted, only reflects the capabilities granted to an app without indicating how or when those permissions are actually used. As benign apps often over-request permissions, this coarse-grained approach tends to result in high false positives and limited behavioral fidelity. Alternatively, identifying sensitive API calls offers finer granularity but still lacks contextual understanding-treating API calls in isolation ignores the sequential and semantic dependencies among them. In practice, real-world behaviors are typically composed of multi-step workflows. For instance, a data exfiltration behavior may involve accessing local files, encrypting the data, and then transmitting it over the network. Although representation of behavior through complete API call chains allows for the capture of execution context, it often leads to overfitting to specific patterns and fails to generalize across apps that follow semantically similar but syntactically divergent paths.

To address these limitations, we introduce a two-stage sensitive behavior modeling approach: (1) *Sensitive API Call Chain Extraction*: We recover complete control-flow paths, referred to as *sensitive API call chains*, from scenario-labeled entry points to sensitive API invocations, leveraging enhanced static analysis. (2) *Behavior Representation Generation*: Each extracted call chain is abstracted into a semantically rich natural language summary using an LLM, and subsequently embedded into a semantic vector space to support further analysis. This two-stage design preserves the execution context of sensitive behaviors while abstracting away low-level implementation details, thereby improving generalization across semantically similar but syntactically diverse behaviors.

3.3.1 ***Sensitive API Call Chain Extraction.*** Android's component-based and event-driven architecture enables rich inter-component communication (ICC) through the `Intent` mechanism [5]. However, this flexibility also introduces challenges for static analysis. Sensitive behaviors often span multiple components and asynchronous callbacks, making it difficult to recover complete execution paths using existing static analysis tools such as FlowDroid. In particular: (1) *Inter-Component Discontinuities*: Traditional static analyzers struggle to track cross-component calls introduced by `startActivity()`, `startService()`, or `sendBroadcast()`, leading to disconnected call chains. (2) *Asynchronous Gaps*: APIs such as `Thread`, `Handler`, or `AsyncTask` introduce implicit edges between scheduling and execution points, which are typically missed by default call graph construction.

To address these challenges, we propose a three-stage refinement of the call graph: (1) *Initial Call Graph Generation*: We first use FlowDroid to generate the initial call graph of the Android app and identify all reachable component entry points. (2) *Asynchronous Edge Completion*: Following the prior work [45], we maintain a mapping of commonly used asynchronous call pairs (e.g., `Thread.start()` → `Runnable.run()`, `AsyncTask.execute()` → `doInBackground()`). To resolve these implicit edges, we apply def-use chain and pointer analysis [8] to infer the actual target methods of asynchronous invocations. We then add these links explicitly to the call graph, recovering hidden flows across threads and tasks. (3) *ICC Path Repair via ICCBot*: To bridge inter-component gaps, we integrate ICCBot [43], which tracks the flow of `Intent` objects via data-flow analysis and extracts both source and target component summaries, enabling recursive resolution of ICC paths, covering activity transitions, service invocations, and broadcast handling.

Through these refinements, we construct a comprehensive call graph that enables precise extraction of call chains from scenario entry points to sensitive APIs, forming the structural backbone for subsequent behavior representation and anomaly detection.

3.3.2 ***Behavior Representation Generation.*** Given that raw sensitive API call chains are often lengthy, obfuscated, and difficult to interpret, we use LLMs to convert raw sensitive API call chains into brief yet clear summaries, which perform better in improving the semantic expressiveness and interpretability. To further assess the necessity of this process, we provide a detailed visualization-based evaluation in Appendix B, which compares raw sensitive API call chains and their corresponding summaries in terms of embedding cohesion and inter-category separation.

LLM-Based Behavior Summarization. Since LLM outputs are highly influenced by prompt phrasing, we carefully design our prompts based on three key principles: terminology consistency, concise abstraction, and functional generalization. These principles guide the LLM in transforming low-level sensitive API call chains into high-level behavior summaries that are robust against code obfuscation and structural variability. The prompt template used in our system is detailed in Appendix C, including the instructions, guidelines, and output requirements.

Behavior Embedding. To support effective semantic comparison and anomaly detection, we encode each LLM-generated behavior summary into a fixed-dimensional vector space. We employ the `all-MiniLM-L6-v2` model from the SentenceTransformers library [33], which provides efficient and semantically meaningful sentence-level embeddings. Formally, given a behavior summary b, we compute its embedding vector v_b as:

$$v_b = \text{SentenceTransformer}(b) \tag{5}$$

These behavior embeddings will be used in later stages for semantic similarity computation across behaviors, enabling unsupervised anomaly detection based on deviations from known benign behavior distributions.

Figure 3 illustrates how a raw sensitive API call chain is abstracted into a semantically meaningful behavior representation through LLM-based summarization and embedding. The example shows a call chain extracted from an obfuscated advertisement SDK, where method names (e.g., `a()`, `b()`) provide little semantic

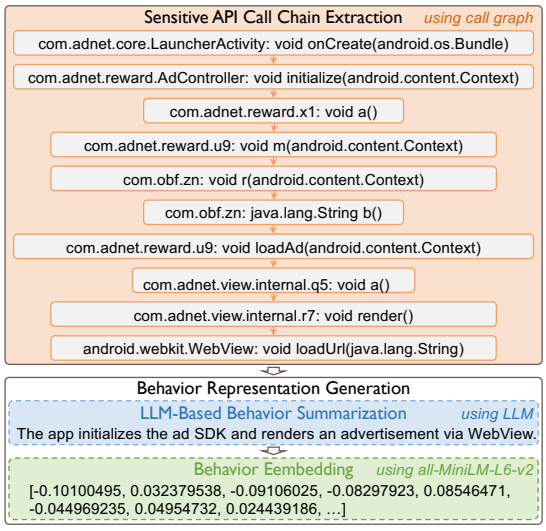


Figure 3: Raw Sensitive API Call Chain to Behavior Embedding via LLM-based Summarization and Embedding.

information. The LLM-generated summary accurately captures the high-level intent (i.e., "initializes the ad SDK and renders an advertisement via WebView"). This summary is then encoded into a fixed-dimensional semantic vector using all-MiniLM-L6-v2, enabling semantic comparison for anomaly detection.

3.4 Fine-Grained Anomaly Detection and Explanation

With functional scenario modeling and behavior embeddings in place, we now focus on detecting abnormal behaviors with fine granularity and semantic interpretability. To this end, this section introduces three core modules: (1) *Reference Library Construction*, which builds a **scenario-behavior aligned reference library** as the semantic foundation for anomaly detection; (2) *Scenario-Aware Anomaly Detection*, which identifies malicious behaviors by quantifying semantic deviations from scenario-specific benign behavior distributions derived from the reference library; and (3) *LLM-Based Malware Report Generation*, which enhances interpretability by producing structured, human-readable reports for the most anomalous behaviors detected.

3.4.1 ***Reference Library Construction.*** To support scenario-aware anomaly detection, we construct a scenario-behavior aligned reference library that serves as a semantic baseline. Specifically, it groups behavior embeddings of benign apps based on the functional scenario labels derived from their entry points, capturing scenario-specific distributions of legitimate behaviors. Concretely, for each cluster c and its scenario label l_c obtained from Section 3.2, we further generate their behavior embeddings from previously collected benign apps using the approach introduced in Section 3.3. These embeddings are then grouped by scenario label l_c to form a reference distribution $\mathcal{B}_{l_c}$:

$$\mathcal{B}_{l_c} = \{v_1, v_2, \ldots, v_n\}$$

where each v_i denotes the embedding of a benign behavior belonging to scenario label l_c. Intuitively, $\mathcal{B}_{l_c}$ characterizes the distribution of expected ("normal") behaviors within the scenario label l_c. It captures intra-scenario semantic regularities, such as submitting credentials after login or accessing location during map use, providing a reliable baseline for detecting anomalous behaviors.

3.4.2 ***Scenario-Aware Anomaly Detection.*** Once the reference library is established, we evaluate an unknown app by comparing its behavior embeddings, partitioned by scenario labels, against the corresponding benign behavior distributions in the reference library. This enables the detection of fine-grained semantic inconsistencies that may be overlooked by global detection schemes.

Scenario-Level Scoring. To balance robustness and semantic sensitivity, we adopt a top-k similarity averaging strategy. Given a new behavior embedding v^* associated with scenario label l_c^*, we compute its anomaly score based on its average cosine similarity to the k nearest benign embeddings in the corresponding reference distribution $\mathcal{B}_{l_c^*}$. Formally:

$$\text{AnomalyScore}(v^*, l_c^*) = 1 - \frac{1}{k} \sum_{i=1}^{k} \cos\left(v^*, \text{NN}_i(\mathcal{B}_{l_c^*}, v^*)\right) \quad (6)$$

where $\text{NN}_i(\mathcal{B}_{l_c^*}, v^*)$ denotes the i-th nearest neighbor of v^* in $\mathcal{B}_{l_c^*}$, measured by cosine similarity. A higher score indicates a greater semantic deviation from benign behaviors within the functional scenario. We empirically set $k = 50$ based on a grid search in the range $k = 10 \sim 100$, where it yielded the best balance between robustness and sensitivity. This strategy mitigates the risk of relying on a single noisy reference point while preserving discriminative power.

Global Anomaly Aggregation. To assess app-level anomalies, we aggregate all scenario-level anomaly scores into a vector $\mathbf{a} = [a_1, a_2, \ldots, a_k]$, where each a_i denotes the anomaly score of a behavior under a specific scenario. This vector is then fed into a one-class SVM [35] trained solely on benign apps. This enables zero-day detection by identifying outliers in the high-dimensional anomaly space, without requiring labeled malware during training.

3.4.3 ***LLM-Based Malware Report Generation.*** To enhance the interpretability of detection results, we introduce a malware explanation module powered by LLMs. Rather than merely identifying anomalous behaviors, this module refines candidates that truly suggest malicious intent and generates human-readable reports explaining why an app is flagged as malware, effectively addressing the explainability gap in traditional detection systems. We adopt DeepSeek as the underlying LLM due to its open-source availability, low deployment cost, and strong multi-step reasoning capabilities.

Candidate Filtering and Prompt Construction. To ensure report quality and relevance, we first filter behavior candidates based on their anomaly scores. Only behaviors with scores exceeding a configurable threshold (empirically set to 0.5, which offered the best trade-off in explanation filtering, with stable performance in the 0.4–0.6 range) are selected for explanation. For each behavior candidate, we provide DeepSeek with the following contextual elements: its complete resolved call chain from the scenario-labeled entry point to the sensitive API, and the associated functional scenario semantic signals obtained in Section 3.2, indicating the triggering context (e.g., SMS reception). These inputs are provided into a task-specific prompt that instructs the LLM to determine whether the behavior exhibits malicious intent, and to produce a three-part explanation in a standardized, analyst-friendly format: (1) *Anomaly Summary*: a concise description of the app's malicious functionality and the rationale for its potential threats; (2) *Risk Breakdown*: a detailed analysis of behavioral risks, such as privacy violations, data exfiltration, or permission abuse; and (3) *Mitigation Suggestions*: recommended follow-up actions, such as permission audits, traffic monitoring, or runtime tracing. The prompt is designed to simulate the workflow of human security analysts, encouraging the LLM to synthesize code and semantic cues into interpretable findings.

Interpretability via LLM Reasoning. A detailed report provided in the supplementary material [19] shows an example generated by DeepSeek for an app previously classified as malicious. Despite code obfuscation and ambiguous method names, this model correctly identifies high-risk behaviors, such as SMS-based command dispatch, sensitive data harvesting, and Tor-based exfiltration. Notably, DeepSeek's reasoning mimics the logic of human analysts: it associates the triggering context (e.g., SMS_RECEIVED) with behavioral intent, and maps sensitive API combinations to known threat behaviors (e.g., spyware, banking trojans). DeepSeek can also infer obfuscation techniques, detect control flow strategies, and recognize privilege misuse-all without handcrafted rules. Overall, this LLM-based explanation module significantly improves semantic interpretability, reduces manual effort, and bridges the gap between automated detection and human-understandable malware analysis.

We implemented `ForeDroid` using Java and Python. Specifically, we employed FlowDroid [8] and ICCBot [43] for static analysis tasks such as call graph extraction, entry point identification, and call chain resolution. For functional scenario clustering and anomaly detection, we utilized the Scikit-learn library [31] in Python. To generate natural language descriptions of behaviors based on the call chain, we deployed the `llama3-70b-8192` model locally via Ollama and used the `all-MiniLM-L6-v2` model from `sentence-transformers` for behavior embedding. For LLM-based anomaly explanation, we integrated the `deepseek-r1:14b` model, also deployed locally through Ollama.

`ForeDroid` supports end-to-end analysis in a few minutes per APK, including under 2 minutes for static analysis, under 10 seconds for anomaly detection, and around 1.3 minutes for LLM-based explanation—reducible to seconds when accessed via API.

4 Experiment

To evaluate the effectiveness of `ForeDroid`, we propose the following three research questions:

- **RQ1: Zero-Day Malware Detection.** Can `ForeDroid` effectively detect previously unseen (zero-day) malware samples under temporal and family-based evaluation settings?
- **RQ2: Fine-Grained Behavior Analysis.** How well does `ForeDroid` identify and explain malicious behaviors at a fine-grained level compared to existing baselines?
- **RQ3: Component Effectiveness.** To what extent does each core component (i.e., scenario-aware clustering, call chain completion, and LLM-based behavior summarization) contribute to the overall detection performance of `ForeDroid`?

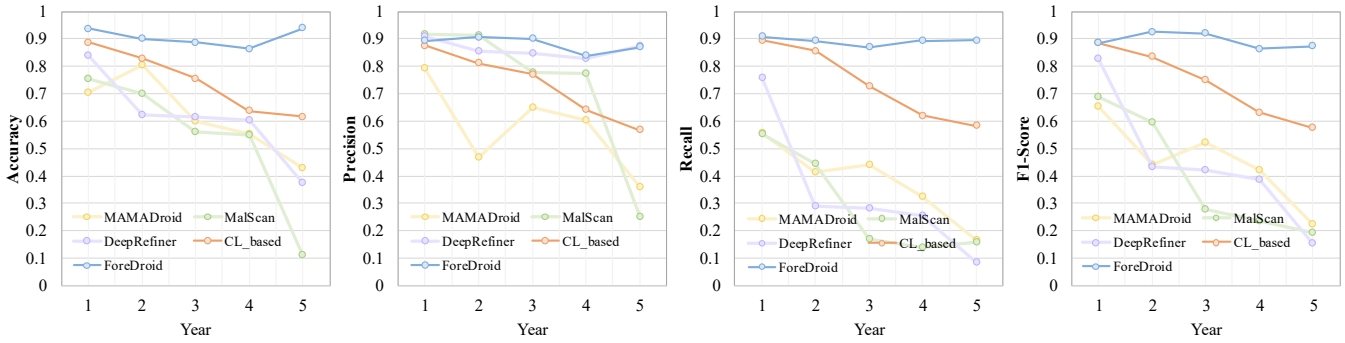


Figure 4: Performance trend of different methods over time across four metrics.

Table 1: Summary of datasets used in our experiments.

Class	Tag	Date Range	Apps (#)
Malicious	Drebin	–	5,560
	GPMalware	2016–2021	105
	AndrozooMal	2014–2022	18,000
Benign	AndrozooBen	2014–2022	18,000
Total	–	–	**41,665**

Dataset. Our dataset was built from three commonly used academic sources: Drebin [7], GPMalware [10], and AndroZoo [4] , and is organized into four dataset subsets as follows:

- **Drebin**: A widely used malicious dataset consisting of 5,560 malware samples with family annotations.
- **GPMalware**: A malicious dataset of 105 malware samples discovered on Google Play between Jan. 2016 and Jul. 2021, each accompanied by detailed manual reports.
- **AndroZooMal**: A malicious dataset of 18,000 malware samples collected from AndroZoo between 2014 and 2022, where each sample is flagged as malicious by at least 15 antivirus engines on VirusTotal [1].
- **AndroZooBen**: A benign dataset of 2,000 benign samples from the same 2014–2022 period in AndroZoo, where each sample is confirmed benign by all antivirus engines on VirusTotal. These are distinct from the 28,994 representative benign apps discussed in Section 3.1.

As summarized in Table 1, our final dataset contains **41,665** Android apps, with **23,665 malicious** and **18,000 benign** samples, spanning a period of 9 years.

4.1 Zero-Day Malware Detection

4.1.1 ***Setup.*** Zero-day malware refers to previously unknown malicious apps that have not yet been identified by anti-virus software [20]. To simulate realistic zero-day detection challenges, we adopt two evaluation settings:

(1) Scenario A: Temporal Split. This setting evaluates the model's ability to generalize across time, i.e., training on older samples and testing on newer ones. We split the dataset based on app timestamps, ensuring the testing set only contains apps collected after the training period. Specifically, we use the AndrozooMal and AndrozooBen datasets, covering the years 2014 to 2022 (2,000 apps per year). For baseline methods that require training samples, we use apps from 2014-2017 for 10-fold cross-validation and test on apps from 2018-2022. To ensure temporal fairness, we also re-evaluated ForeDroid by constructing its reference library solely from benign apps collected during 2014–2017, and then testing on the 2018–2022 set. Under this setting, we compare ForeDroid with four baselines. MaMaDroid [25]: a static Android malware detection system that models API call sequences as Markov chains. MalScan [40]: a detection framework leveraging sensitive API correlation graphs to identify malicious logic. DeepRefiner [42]: a semantic-based deep learning model leveraging LSTM and MLP for Android bytecode and XML semantics. Chen et al. [13] (*CL-based*): a continuous learning-based detection framework that adapts its model via incremental learning to cope with distribution drift.

(2) Scenario B: Family-Based Split. This setting assesses the model's ability to detect previously unseen malware families, i.e., training on known malware families and testing on previously unseen ones. We first annotate the family labels of the AndrozooMal dataset using AVCLASS++, an enhanced malware labeling framework built upon the state-of-the-art AVCLASS [36]. For baseline methods requiring labeled training data, we use the Drebin dataset as the training set. We then randomly select 2,000 samples from AndrozooMal whose family labels do not appear in Drebin (e.g., `Smsreg`, `Skymobi`) to form the zero-day test set. This setting is designed to assess the model's generalization ability in identifying malware that emerged with previously unseen behavior patterns. For this scenario, we compare with three baselines: MaMaDroid [25], MalScan [40], and DeepRefiner [42]. Chen et al. [13] is not included here, as it requires a continuous temporal data stream for incremental updates, which is not compatible with the family-based split setting.

To measure detection effectiveness, we adopt four standard metrics: Accuracy, Precision, Recall, and F1-Score, defined as follows: Accuracy = $\frac{TP+TN}{TP+TN+FP+FN}$, Precision = $\frac{TP}{TP+FP}$, Recall = $\frac{TP}{TP+FN}$, and F1-Score = $\frac{2\cdot\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}$, where TP, TN, FP, and FN refer to the numbers of correctly detected malware, correctly detected benign apps, benign apps misclassified as malware, and malware missed as benign, respectively. To further assess the robustness of malware detectors over time, we introduce the Area Under Time

Table 2: F1-Score (%) comparison for zero-day malware detection in Scenario A (Temporal Split).

Method	Cross Validation	2018	2019	2020	2021	2022	Average
MaMaDroid	93.2	65.4	43.9	52.4	42.2	22.6	45.3
MalScan	94.3	69.0	59.7	27.7	23.6	19.3	39.9
DeepRefiner	94.7	82.7	43.3	42.2	38.9	15.3	44.5
CL-based	99.7	88.5	83.4	74.9	63.1	57.6	73.5
ForeDroid	–	90.0	89.9	88.5	86.6	88.2	**88.6**

Table 3: Performance comparison in terms of AUT.

Method	AUT(Acc.)	AUT(Pre.)	AUT(Recall)	AUT(F1)
MaMaDroid	0.6320	0.5734	0.3851	0.4563
MalScan	0.5608	0.7615	0.2765	0.3879
DeepRefiner	0.6117	0.8559	0.3115	0.4335
CL-based	0.7435	0.7368	0.7362	0.7361
ForeDroid	**0.8976**	**0.8819**	**0.8895**	**0.8853**

(AUT) metric, which aggregates yearly scores into a single value to reflect stability and long-term reliability. Given T time points (i.e., 5 years from 2018 to 2022), and metric values $\{s_1, s_2, ..., s_T\}$, AUT is calculated as:

$$\text{AUT} = \frac{1}{T-1} \sum_{i=1}^{T-1} \frac{s_i + s_{i+1}}{2} \tag{7}$$

A higher AUT indicates better temporal stability; lower values suggest rapid degradation over time and limited generalization.

4.1.2 ***Results***. We evaluated the performance of zero-day malware detection under both scenarios:

(1) Scenario A (Temporal Split): Table 2 shows the year-wise F1-scores of ForeDroid and four baselines (MaMaDroid, MalScan, DeepRefiner, and CL-based under the temporal-split setting. MaMaDroid and MalScan show substantial performance degradation over time, with F1-scores dropping from 65.4% to 22.6% and from 69.0% to 19.3%, respectively, resulting in average F1-scores of only 45.3% and 39.9%. DeepRefiner, a deep learning approach, declines sharply from 82.7% to 15.3%, while CL-based approach achieves higher performance but still drops from 88.5% to 57.6% and requires continuous labeling and retraining. In contrast, our method maintains consistently high performance, with F1-Scores exceeding 86% each year and an average of 88.6%. Notably, our approach is unsupervised and does not require retraining, yet still outperforms supervised baselines. This stable performance across five consecutive years demonstrates ForeDroid's strong resilience to evolving malware behaviors. While ForeDroid does not rely on labeled malware or retraining, its performance may be influenced by the coverage and diversity of the reference library. In particular, rare or unconventional behaviors that are not well represented in the reference library may lead to less confident anomaly assessments. However, our results show that, when built upon a carefully curated and representative benign corpus, ForeDroid can generalize effectively, demonstrating strong practicality and robustness in temporal-split zero-day detection settings.

Table 4: Zero-day detection performance in Scenario B (Family-Based Split).

Method	Accuracy	Precision	Recall	F1-Score
MaMaDroid	0.276	0.393	0.133	0.198
MalScan	0.219	0.367	0.217	0.273
DeepRefiner	0.579	0.795	0.214	0.337
ForeDroid	**0.885**	**0.834**	**0.960**	**0.893**

Table 3 presents the AUT comparison across four evaluation metrics-Accuracy (A), Precision (P), Recall (R), and F1-Score (F1). ForeDroid achieves the highest AUT in all metrics, with an AUT(F1) of 0.8853, outperforming MaMaDroid (0.4563), MalScan (0.3879), DeepRefiner (0.4335), and CL-based (0.7361). These results indicate that our method not only performs well in individual years but also maintains long-term reliability, making it particularly suitable for real-world deployment where malware threats continuously evolve. Figure 4 further illustrates the performance trends of the four metrics over the five-year testing period. All baseline methods suffer noticeable performance degradation across metrics, particularly in Recall, where F1-scores fall below 0.5 within two years. The only exception is CL-based, whose decline is less severe but still evident, and whose approach requires periodic re-training. In contrast, ForeDroid maintains consistently high performance throughout, demonstrating temporal stability and resilience.

(2) Scenario B (Family-Based Split): Table 4 reports the zero-day detection performance under the family-based split setting. In this scenario, both MaMaDroid and MalScan exhibit poor generalization to unseen malware families, with F1-scores below 0.30. Their extremely low recall rates (0.133 and 0.217, respectively) indicate a severe inability to identify novel threats that deviate from previously learned family-specific behavior patterns. DeepRefiner achieves higher Precision (0.795) but continues to exhibit low Recall (0.214), resulting in an F1-score of only 0.337. In contrast, ForeDroid significantly outperforms both baselines, achieving an F1-score of 0.893 and a recall of 0.960. These results demonstrate the strong generalization capability of our unsupervised, scenario-aware detection framework, which operates independently of family-specific training data. By modeling behaviors within functional scenarios, ForeDroid effectively captures the semantic intent behind malicious actions rather than relying on fixed feature patterns, enabling it to robustly detect unfamiliar threats. This confirms the practical value of ForeDroid in real-world deployment environments, where emerging malware families must be accurately identified without prior knowledge or retraining.

4.2 Fine-grained Behavior Analysis

4.2.1 ***Setup***. To assess the fine-grained behavior detection capability of ForeDroid, we conduct a comparative evaluation against the baseline, ProMal [39], on the GPMalware dataset. ProMal is the state-of-the-art system designed for precise and interpretable malicious behavior analysis at a fine-grained level. While the original GPMalware taxonomy categorizes behaviors into eight high-level payload types-*Information stealing*, *Ad abuse*, *Premium charges*, *Cryptomining*, *Root exploit*, *Clipboard hijacking*, *Port forwarding*, and *Ransom*-we refine these categories into a set of 25 fine-grained

Table 5: Fine-Grained Behavior Types Used in Evaluation.

Type ID	Behavior Type	Payload Category
T1	Device fingerprinting (e.g., IMEI, IMSI)	Information stealing
T2	Location tracking	Information stealing
T3	App list scanning	Information stealing
T4	Contact/call record access	Information stealing
T5	Clipboard access	Clipboard hijacking
T6	Silent SMS sending	Premium charges
T7	Premium-rate call dialing	Premium charges
T8	Ad SDK loading via WebView	Ad abuse
T9	WebView + JS injection	Ad abuse / Info stealing
T10	Tor/VPN/socket-based comm	Port forwarding
T11	Shell command execution	Root exploit
T12	File encryption	Ransom
T13	Audio/photo capture	Spyware
T14	Suspicious background services	Unknown
T15	Remote control (e.g., SMS commands)	Premium / Spyware
T16	JS-based cryptomining	Cryptomining
T17	Local CPU-based mining	Cryptomining
T18	Mining backdoor command exec	Cryptomining / Backdoor
T19	Dynamic payload loading/execution	Backdoor
T20	SMS content leakage	Information stealing
T21	CPA fraud (click/convert spoofing)	Ad fraud
T22	Credential phishing	Phishing / Scam
T23	App icon hiding	Stealth
T24	Screen scraping / UI hijacking	Spyware
T25	File enumeration / exfiltration	Information stealing

Table 6: Comparison of Fine-Grained Behavior Detection on the GPMalware Dataset.

Method	#Detected	#FP	#FN	Precision	Recall	F1-Score
ForeDroid	652	40	27	0.9386	**0.9577**	**0.9480**
ProMal	634	38	43	**0.9401**	0.9327	0.9364

behavior types to support more precise evaluation. Table 5 summarizes the taxonomy, where each fine-grained behavior (e.g., Silent SMS sending) is mapped to a corresponding high-level payload category. Based on this refined taxonomy, we manually annotated a total of 639 ground-truth behaviors across 102 malware samples (out of 105 total samples; three failed due to call graph extraction errors). We then compared the outputs of ForeDroid and ProMal against these annotations on a per-sample, per-behavior-type basis.

4.2.2 ***Results.*** Table 6 compares ForeDroid and ProMal in terms of behavior detection counts, error rates (false positives and false negatives), and evaluation metrics (precision, recall, and F1-score) against ground-truth annotations on the GPMalware dataset. ForeDroid outperforms ProMal in fine-grained malicious behavior detection. It successfully identifies more ground-truth behaviors, with a slightly higher number of false positives but significantly fewer false negatives (27 vs. 43). This results in improved recall and a higher F1-score of 0.9480, compared to 0.9364 for ProMal. These results indicate that ForeDroid provides more comprehensive coverage of semantically diverse malicious behaviors. To gain deeper insights into ForeDroid's performance, we conduct a detailed error analysis focusing on both false negatives and false positives.

FP Analysis. Most false positives arise from semantically ambiguous behaviors that lack sufficient contextual signals to reliably distinguish between benign and malicious intent. These include benign-like API patterns (e.g., WebView with JavaScript injection), privacy-preserving networking (e.g., Tor/VPN usage), and advertisement SDKs that resemble ad fraud but lack concrete attribution evidence. Compared to ForeDroid, ProMal reports slightly fewer false positives (38 vs. 40), achieving higher precision (0.9401 vs. 0.9386). This is largely attributed to ProMal's design assumption that the input samples are already confirmed malware. Under this assumption, ProMal focuses solely on interpreting known malicious behaviors, allowing it to filter out benign-looking behaviors more conservatively. However, this also limits its generalization ability and disqualifies it as a true detection system, since it cannot be applied to unknown or unlabeled samples in practice. In contrast, ForeDroid is designed for open-world detection scenarios, where malicious intent is inferred from behavioral inconsistencies without prior knowledge. As a result, it may be more aggressive in flagging ambiguous patterns, such as WebView interactions or networking modules, leading to a slight increase in false positives.

FN Analysis. We observe that most missed behaviors (false negatives) fall into one of the following three categories: (1) behaviors relying on non-sensitive and generic APIs (e.g., clipboard access), which account for only 0.6% of all cases (4 out of 639) and are conservatively excluded from detection to avoid overgeneralization; (2) runtime-dependent behaviors involving dynamic payloads or code loading; and (3) web-driven threats such as phishing or injected JavaScript, which are not statically recoverable.

A more detailed breakdown of all false positive and false negative cases is provided in the supplementary material [19].

Beyond standard payload execution, ForeDroid also leverages scenario-aware entry point modeling and complete call chain analysis to uncover deception and persistence mechanisms. For instance, ForeDroid successfully identifies that AndroidOS.JSMiner disguises itself as a religious tool app (Rosario Prayer) while conducting background cryptomining; and FalseGuide blends ad fraud and data exfiltration under the cover of a football gaming app. ForeDroid also detects forensic evasion strategies and sandbox-aware logic in multiple samples, e.g., Hiddad conditionally disables payloads when executed in emulator environments; and Aladdin leverages `Runtime.availableProcessors()` to detect single-core CPUs as a proxy for sandbox environments. Furthermore, our system highlights persistence risks embedded in malicious apps. For example, `AlarmReceiver` registered for `BOOT_COMPLETED` enables malware like Haken to self-start upon device reboot, while `OnRestartReceiver` in Solid implements fallback mechanisms to ensure the continuous presence of fraudulent components. These advanced detection and reasoning capabilities demonstrate the effectiveness of ForeDroid in bridging low-level call chains analysis with high-level semantic interpretation.

4.3 Ablation Study of Core Components

4.3.1 ***Setup.*** To investigate the contribution of each core component in our system, we perform an ablation study using the AndrozooMal and AndrozooBen datasets (spanning 2018-2022). We evaluate all variants under the same anomaly detection pipeline,

Table 7: Result of the Ablation Study.

Variant	Acc.	Precision	Recall	F1
Full System (Ours)	**0.9000**	**0.8716**	**0.9183**	**0.8931**
✗ LLM-Based Behavior Summarization	0.6017	0.5587	0.6287	0.5901
✗ Scenario-Aware Clustering	0.6644	0.5676	0.7119	0.6316
✗ Call Chain Completion	0.7701	0.7813	0.7500	0.7653

changing only the target component while keeping all other settings fixed. The following three key components are considered: (1) *Scenario-Aware Clustering*: We compare our scenario clustering using semantic context extracted in Section 3.2 against a baseline clustering approach using only callback method names. (2) *Call Chain Completion*: We evaluate our asynchronous, ICC-aware call chain completion mechanism by comparing it with the initial call graph output generated by FlowDroid. (3) *LLM-Based Behavior Summarization*: We compare raw call chain embeddings with LLM-generated natural language summaries followed by embedding using `all-MiniLM-L6-v2`.

4.3.2 ***Results.*** Table 7 reports the impact of each core component on detection performance, measured by Accuracy (Acc.), Precision (Pre.), Recall, and F1-Score. Among the variants, we observe that removing the *LLM-Based Behavior Summarization* leads to the most severe performance drop, reducing the F1-score by over 30% to 0.5901. Notably, both precision and recall decline sharply, confirming that raw call chain embeddings lack sufficient semantic expressiveness. This highlights the central role of LLM-generated summaries in extracting meaningful behavior semantics from noisy or obfuscated call chains. We further analyze the impact of the other components: (1) *Scenario-Aware Clustering*: Disabling semantic clustering and using only callback method names degrades F1-score to 0.6316. This demonstrates that semantic context (e.g., component names, intents, GUI labels) is essential for accurate behavior grouping and anomaly localization. (2) *Call Chain Completion*: Without ICC-aware call chain expansion, the F1-score drops to 0.7653. While the effect is less severe than that of the LLM module, this confirms the importance of reconstructing complete control flow paths to expose full behavior paths. Overall, these results demonstrate that each component contributes meaningfully, but the LLM-based behavior summarization is most critical for achieving high precision and recall in fine-grained malware detection.

5 Limitations and Discussion

While `ForeDroid` demonstrates strong performance in both zero-day detection and fine-grained behavior analysis, several limitations remain due to inherent challenges in static analysis and scenario-level modeling. (1) *Static Analysis Blind Spots*: `ForeDroid` relies on enhanced static call graphs to reconstruct behavior paths. However, behaviors involving runtime-loaded payloads, reflective class loading, or dynamic UI overlays are inherently difficult to capture through static analysis. As a result, threats such as phishing via injected web content or payloads triggered through reflection may be underdetected. That said, while the malicious logic embedded within dynamically injected content may be invisible to static analysis, `ForeDroid` is still capable of identifying the associated loading behavior itself, such as remote code fetching or reflective class loading. When such operations occur within suspicious functional scenarios, they can still be flagged as anomalous. (2) *Limited Context Granularity*: Although `ForeDroid` constructs functional scenarios using GUI text, callback semantics, and complete call chains to model execution context as comprehensively as possible, certain behaviors still require finer-grained or runtime-specific information to accurately infer malicious intent. For example, distinguishing benign advertisement SDKs from ad fraud often involves examining actual ad content or click behavior, which may exceed the scope of static analysis. Similarly, privacy-preserving apps and malware may share common structural features (e.g., Tor/VPN usage), where traffic-level semantics are necessary for reliable disambiguation. (3) *Benign Behavior Coverage*: `ForeDroid` assumes that behaviors semantically similar to those in a benign corpus are safe. However, due to the diversity of Android apps, this corpus may not fully cover all benign usage patterns, potentially resulting in false positives for rare but non-malicious cases. To mitigate this, we have constructed a large-scale benign behavior corpus comprising 28,994 high-quality apps across 49 categories to maximize functional diversity and improve generalization. Additionally, we support periodic updates to incorporate new benign patterns and mitigate the effects of behavior drift over time. (4) *Adversarial Evasion and Obfuscation*: While `ForeDroid` avoids hand-crafted features and leverages non-obfuscated key system APIs to resist traditional evasion (e.g., feature-space [11], obfuscation, graph perturbation), advanced techniques—such as dummy API injection or benign-like behavior blending—may still affect semantic analysis. `ForeDroid` mitigates these via LLM-based abstraction and per-scenario semantic validation. Nonetheless, we leave stronger resistance to adaptive attacks and advanced obfuscation as future work. (5) *Side-Channel Threats*: `ForeDroid` focuses on explicit sensitive API misuse within identifiable functional scenarios. Implicit threats such as side-channel leakage are beyond the scope of static semantic modeling and remain important future work.

6 Related Work

6.1 Android Malware Detection

To keep pace with evolving Android malware, various detection methods have been proposed, including static analysis [8], dynamic analysis [16, 28, 30], and ML/DL-based approaches [2, 7, 42]. While deep models achieve high accuracy (above 99% [24]), they face two key limitations: (1) poor generalization to zero-day malware due to distribution shifts, and (2) limited interpretability, offering only binary outputs without pinpointing which behaviors are anomalous[7]. This black-box nature hinders trust and forensic utility. In contrast, `ForeDroid` detects semantic inconsistencies within functional scenarios in an unsupervised manner and generates fine-grained reports to support behavior-level analysis and explanation.

6.2 Behavior Semantic Modeling in Android Apps

Recent work has moved from syntactic pattern matching to semantic modeling of Android app behaviors. Methods such as DroidSIFT [45], AppContext [44], and PikaDroid [3] enhance semantic

understanding by modeling API dependencies, entry point associations, and execution contexts. ProMal [39] further constructs a behavior knowledge graph (BxKG) via manual malware labeling to capture malicious behavior trajectories. However, most methods rely on handcrafted rules or labeled data, limiting scalability to unseen threats. In contrast, `ForeDroid` introduces an end-to-end unsupervised framework that learns semantic behavior representations from entry-point contexts and call chains, enabling scalable detection of anomalies without labeled malware. Another line of work investigates UI-behavior inconsistencies. Backstage [23] statically clusters UI elements based on text and layout context to identify mismatches between user expectations and the APIs triggered. DeepIntent [41] models icon-text–behavior correlations to detect UI-level intent violations in benign apps. While effective in detecting UI anomalies, these methods are limited to GUI-triggered behaviors and rely on surface-level signals (e.g., API or permission usage), lacking semantic modeling of sensitive behaviors. They are thus insufficient for malware detection or behavior explanation. In contrast, `ForeDroid` aligns intent and behavior across both GUI and background contexts, reconstructs sensitive behavior paths, and employs LLMs for anomaly interpretation, enabling interpretable, fine-grained malware detection beyond UI-level inconsistencies.

6.3 Explainability in Malware Behavior Analysis

Explainability plays a crucial role in malware behavior analysis, especially in approaches that go beyond binary classification to provide actionable insights. Numerous studies have pursued interpret model decisions by highlighting influential features or providing attribution scores, including Drebin [7], LIME [34], LEMNA [21], and XMal [38]. While they extract which features contribute to a malicious classification, these methods typically operate at the feature level and cannot reconstruct full causal chains of malicious behavior, making it difficult to recover the contextual logic of an attack. Although ProMal supports behavior description to enhance explainability from a behavioral perspective, it heavily relies on domain-specific knowledge, which makes it costly to maintain and difficult to scale. Conversely, our approach provides behavior-specific insights using LLMs grounded in semantically rich and comprehensive call chains, and generates structured, human-readable reports that detail the attack process with minimal manual effort.

7 Conclusion

In this paper, we propose a scenario-aware framework for Android malware detection that enables unsupervised identification and explanation of malicious behaviors. By modeling behavioral inconsistencies within scenarios and leveraging LLM-generated malware reports, our method enhances generalization and interpretability. `ForeDroid` shows superior performance in zero-day malware detection as well as behavior analysis.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (No. 62472309).

References

[1] 2025. VirusTotal – Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/. Accessed: 2025-04-03.
[2] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in Android. In *International conference on security and privacy in communication systems*. Springer, 86–103.
[3] Joey Allen, Matthew Landen, Sanya Chaba, Yang Ji, Simon Pak Ho Chung, and Wenke Lee. 2018. Improving accuracy of Android malware detection with lightweight contextual awareness. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 210–221.
[4] 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*. 468–471.
[5] Android Developers. 2024. Intents and Intent Filters. https://developer.android.com/guide/components/intents-filters. Accessed March 2025.
[6] Anshul Arora, Sateesh K Peddoju, and Mauro Conti. 2019. Permpair: Android malware detection using permission pairs. *IEEE Transactions on Information Forensics and Security* 15 (2019), 1968–1982.
[7] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss*, Vol. 14. 23–26.
[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. doi:10.1145/2666356.2594299
[9] Yude Bai, Sen Chen, Zhenchang Xing, and Xiaohong Li. 2023. ArgusDroid: detecting Android malware variants by mining permission-API knowledge graph. *Science China Information Sciences* 66, 9 (2023), 192101.
[10] Michael Cao, Khaled Ahmed, and Julia Rubin. 2022. Rotten apples spoil the bunch: an anatomy of Google Play malware. In *Proceedings of the 44th International Conference on Software Engineering*. 1919–1931.
[11] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. 2018. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *computers & security* 73 (2018), 326–344.
[12] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. 2016. Stormdroid: A streaminglized machine learning-based system for detecting Android malware. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*. 377–388.
[13] Yizheng Chen, Zhoujie Ding, and David Wagner. 2023. Continuous learning for Android malware detection. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1127–1144.
[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 4171–4186. https://aclanthology.org/N19-1423
[15] Minhong Dong, Liyuan Liu, Mengting Zhang, Sen Chen, Wenying He, Ze Wang, and Yude Bai. 2025. Calmdroid: Core-Set Based Active Learning for Multi-Label Android Malware Detection. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, 37–48.
[16] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2, Article 5 (June 2014), 29 pages. doi:10.1145/2619091
[17] Ruitao Feng, Sen Chen, Xiaofei Xie, Guozhu Meng, Shang-Wei Lin, and Yang Liu. 2020. A performance-sensitive malware detection system using deep learning on mobile devices. *IEEE Transactions on Information Forensics and Security* 16 (2020), 1563–1578.
[18] Ruitao Feng, Jing Qiang Lim, Sen Chen, Shang-Wei Lin, and Yang Liu. 2020. Seqmobile: An efficient sequence-based malware detection system using RNN on mobile devices. In *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 63–72.
[19] ForeDroid. 2025. Supplementary Material for ForeDroid. https://github.com/ForeDroid/ForeDroid/raw/main/Supplementary_Material_for_ForeDroid.pdf.
[20] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. RiskRanker: scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (Low Wood Bay, Lake District, UK) *(MobiSys '12)*. Association for Computing Machinery, New York, NY, USA, 281–294. doi:10.1145/2307636.2307663
[21] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. Lemna: Explaining deep learning based security applications. In *proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 364–379.
[22] Harold Hotelling. 1933. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology* 24, 6 (1933), 417.

[23] Konstantin Kuznetsov, Vitalii Avdiienko, Alessandra Gorla, and Andreas Zeller. 2018. Analyzing the user interface of Android apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 84–87.
[24] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Explainable AI for Android malware detection: Towards understanding why the models perform so well?. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 169–180.
[25] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2016. Mamadroid: Detecting Android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433* (2016).
[26] Leland McInnes, John Healy, and Steve Astels. 2017. HDBSCAN: Hierarchical density based clustering. *The Journal of Open Source Software* 2, 11 (2017), 205. doi:10.21105/joss.00205
[27] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. http://arxiv.org/abs/1301.3781
[28] James Newsome and Dawn Xiaodong Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, Vol. 5. Citeseer, 3–4.
[29] GitHub of ForeDroid. 2025. ForeDroid: Scenario-Aware Analysis for Android Malware Detection and Explanation. https://anonymous.4open.science/r/ForeDroid-5AC6/.
[30] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. 2019. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. *ACM Comput. Surv.* 52, 5, Article 88 (Sept. 2019), 48 pages. doi:10.1145/3329786
[31] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.
[32] Qijing Qiao, Ruitao Feng, Sen Chen, Fei Zhang, and Xiaohong Li. 2022. Multi-label classification for Android malware based on active learning. *IEEE transactions on dependable and secure computing* (2022).
[33] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. ACL. https://www.sbert.net/
[34] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " Why should I trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.
[35] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. 2001. Estimating the support of a high-dimensional distribution. *Neural computation* 13, 7 (2001), 1443–1471.
[36] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. Avclass: A tool for massive malware labeling. In *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19*. Springer, 230–253.
[37] Ray Smith. 2007. An overview of the Tesseract OCR engine. In *Ninth international conference on document analysis and recognition (ICDAR 2007)*, Vol. 2. IEEE, 629–633.
[38] Bozhi Wu, Sen Chen, Cuiyun Gao, Lingling Fan, Yang Liu, Weiping Wen, and Michael R Lyu. 2021. Why an Android app is classified as malware: Toward malware classification interpretation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–29.
[39] Chunlian Wu, Sen Chen, Jiaming Li, Renchao Chai, Lingling Fan, Xiaofei Xie, and Ruitao Feng. 2025. Beyond decision: Android malware description generation through profiling malicious behavior trajectory. *ACM transactions on software engineering and methodology* 34, 7 (2025), 1–39.
[40] Yueming Wu, Xiaodi Li, Deqing Zou, Wei Yang, Xin Zhang, and Hai Jin. 2019. Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 139–150.
[41] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, et al. 2019. Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2436.
[42] Ke Xu, Yingjiu Li, Robert H Deng, and Kai Chen. 2018. Deeprefiner: Multi-layer Android malware detection system applying deep neural networks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 473–487.
[43] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. 2022. ICCBot: fragment-aware and context-sensitive ICC resolution for Android applications. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 105–109. doi:10.1145/3510454.3516864
[44] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. 2015. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE international conference on software engineering*, Vol. 1. IEEE, 303–313.
[45] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1105–1116. doi:10.1145/2660267.2660359

A Representative GUI-Associated Functional Scenario Labels

Figure 5: Visualization of representative UI text expressions across six typical functional scenarios using word clouds.

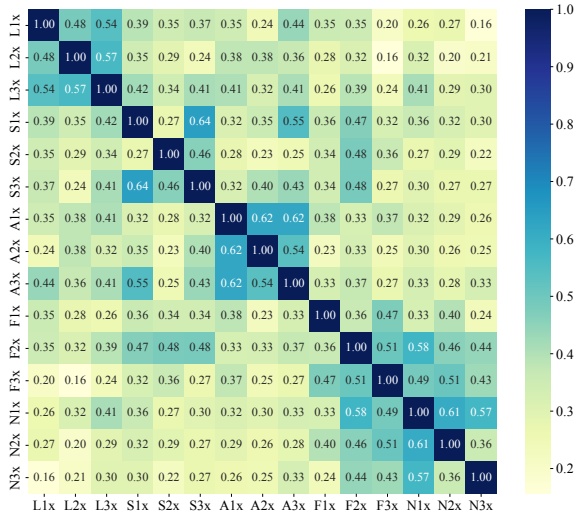
Figure 5 visualizes representative UI text expressions across six typical functional scenarios using word clouds, highlighting the semantic coherence and diversity of each cluster.

B Evaluating the Semantic Effectiveness of LLM-Generated Behavior Summaries

To assess the semantic cohesion and expressiveness of LLM-generated behavior summaries, we conduct a comparative visualization experiment involving both raw sensitive call chains and their corresponding LLM-based natural language summaries. We randomly select three representative sensitive call chains from each of five behavior categories-*Location*, *SMS*, *Audio*, *File*, and *Network*-based on the types of sensitive permissions required. This yields a total of 15 samples. Each call chain is paired with its corresponding LLM-generated summary, forming two parallel representations: the original sensitive call chains and their semantic summaries. To measure semantic similarity, we embed both raw call chains and summaries into a shared vector space using model `all-MiniLM-L6-v2` from the SentenceTransformers library. Pairwise cosine similarities are then computed and visualized as heatmaps.

B.1 Semantic Similarity Heatmap

Figure 6 compares the semantic similarity matrices of the two representations. Panel (a) shows the similarity scores among raw sensitive call chains, while panel (b) shows scores among their corresponding LLM-generated summaries. Each row/column corresponds to one behavior instance, grouped sequentially by behavior type (i.e., rows 1–3: Location, 4–6: SMS, etc.). As the heatmaps reveal, the LLM-based representations exhibit significantly stronger intra-group cohesion and reduced inter-group confusion. Darker blocks along the diagonal in (b) indicate tighter clustering within



(a) Semantic Similarity Heatmap of Raw Call Chains.

(b) LLM-Generated Summary Similarity Heatmap.

Figure 6: Semantic similarity comparison across five functional behavior groups (Location, SMS, Audio, File, Network). (a) illustrates pairwise similarity based on raw call chains; (b) shows similarity based on LLM-generated summaries.

Instruction: Please generate a concise and clear behavior description based on the given sensitive API call chain.

Guidelines:

- Use standard and meaningful verbs and nouns to ensure interpretability and consistent terminology;
- Avoid lengthy or complex sentences-focus on key behavioral actions and goals;
- Merge semantically related functions to remove redundancy and abstract low-level implementation details.

Output Requirement: Only return the natural language behavior description without additional explanations.

Figure 7: Prompt design for LLM-based behavior summary generation.

each behavior category, validating the ability of LLM-generated summaries to enhance semantic alignment.

B.2 Key Observations

- **Stronger Intra-Group Cohesion:** LLM-based summaries produce more consistent embeddings within the same behavior category.
- **Improved Inter-Group Discrimination:** Compared to raw call chains, LLM summaries reduce semantic overlap between categories, facilitating more accurate grouping.
- **Resilience to Obfuscation:** By abstracting low-level implementation details, LLM summaries maintain interpretability even in the presence of naming obfuscation.

These results underscore the value of using LLM-generated summaries as semantically rich representations for behavior modeling, and provide empirical support for their use in downstream tasks such as anomaly detection and malware interpretation.

C Prompt Design for LLM-Based Behavior Summary Generation

The prompt in Figure 7 guides the model to abstract low-level details and produce coherent summaries suitable for behavior embedding.