MalLoc: Toward Fine-grained Android Malicious Payload Localization via LLMs

Tiezhu Sun, Marco Alecci, Aleksandr Pilgun, Yewei Song, Xunzhu Tang,
Jordan Samhi, Tegawendé F. Bissyandé, Jacques Klein

University of Luxembourg, Luxembourg
{firstname.lastname}@uni.lu

***Abstract*—The rapid evolution of Android malware poses significant challenges to the maintenance and security of mobile applications (apps). Traditional detection techniques often struggle to keep pace with emerging malware variants that employ advanced tactics such as code obfuscation and dynamic behavior triggering. One major limitation of these approaches is their inability to localize malicious payloads at a fine-grained level, hindering precise understanding of malicious behavior. This gap in understanding makes the design of effective and targeted mitigation strategies difficult, leaving mobile apps vulnerable to continuously evolving threats.**

To address this gap, we propose *MalLoc*, a novel approach that leverages the code understanding capabilities of large language models (LLMs) to localize malicious payloads at a fine-grained level within Android malware. Our experimental results demonstrate the feasibility and effectiveness of using LLMs for this task, highlighting the potential of *MalLoc* to enhance precision and interpretability in malware analysis. This work advances beyond traditional detection and classification by enabling deeper insights into behavior-level malicious logic and opens new directions for research, including dynamic modeling of localized threats and targeted countermeasure development.

***Index Terms*—Android Malware Analysis, Malicious Payload Localization, Large Language Models**

I. Introduction

Android powers billions of mobile devices worldwide [1], enabling a vast ecosystem of applications (apps) that enhance our productivity, entertainment, and daily life. However, the widespread adoption and open nature of the Android platform have made it an attractive target for attackers seeking to exploit users and systems through malicious apps. Over time, Android malware has evolved rapidly, adopting sophisticated techniques such as code obfuscation [2], dynamic behavior triggering [3], and payload repackaging [4] to evade traditional detection mechanisms [5], [6]. These evolving threats pose critical challenges to the maintenance and evolution of secure mobile apps, requiring continuous advancements in malware analysis and mitigation strategies.

While malware detection techniques have improved in identifying whether an app is malicious or benign [7]–[12], and malware family classification techniques can further categorize malicious apps into known families [13]–[17], these advances still fall short of providing actionable insights at the code level. Specifically, most existing models lack the capability for fine-grained localization of malicious payloads, making it difficult to accurately identify the specific code locations responsible for harmful behaviors. This limitation impedes the understanding of how malware operates and evolves, and it prevents researchers from extracting high-quality, behaviorally meaningful features that could strengthen detection models and improve long-term resilience. In addition, many learning-based detection and classification models act as black boxes, offering little insight into their decision-making processes. This lack of transparency hinders analysts' ability to validate predictions, understand model limitations, and derive actionable countermeasures—ultimately limiting their utility in real-world, security-critical contexts.

A few existing works [18], [19] have attempted to localize malicious payloads at the class level. However, they suffer from limited localization precision, making it difficult to precisely pinpoint the actual code implementing malicious behaviors. Moreover, they lack the ability to provide specific behavior descriptions that explain how the identified payloads operate. To the best of our knowledge, our approach *MalLoc* is the first to explore **fine-grained** malicious payload localization along two key dimensions: ❶ **Method-level localization**, which focuses on identifying the small executable code units (Smali methods) responsible for malicious behaviors, improving precision over class-level approaches; ❷ **Detailed behavioral explanations**, which provides human-readable insights into the specific actions and intent of the localized malicious payloads, supporting explainability and analyst validation. To achieve this, *MalLoc* innovatively leverages malware family knowledge as guidance, incorporates LLM-powered semantic reasoning, and employs a two-phase approach to progressively and precisely localize malicious methods and identify their behavioral roles.

To enable a reliable evaluation of *MalLoc*, we developed a demo Android app, *MalApp*, from scratch. It implements several common malicious behaviors observed in real-world malware, such as privacy theft and aggressive advertising, and provides fine-grained ground truth for controlled and quantitative assessment of localization capabilities. In addition,

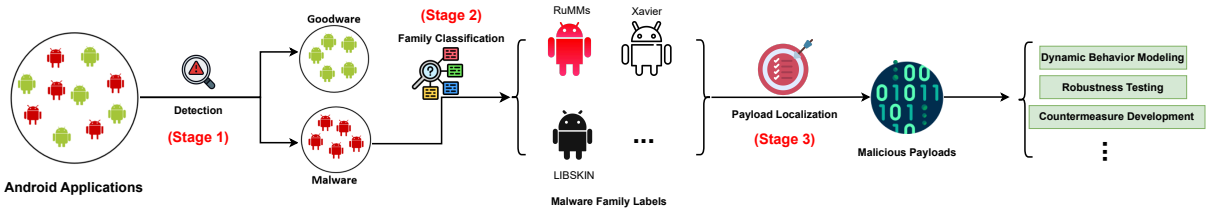

Fig. 1: The overview of different stages of Android malware analysis.

we evaluate *MalLoc* on a real-world Android malware sample from MalRadar [20] and manually analyze its predictions to validate their correctness. Experimental results highlight the potential of *MalLoc* to advance malware analysis toward more precise and explainable behavior localization. We believe this work can inform future research directions, such as dynamic analysis of localized payloads, the development of interpretable malware detection models, and the design of targeted mitigation strategies grounded in behavior-level insights.

The contributions of this work are summarized as follows:

- We propose *MalLoc*, a novel LLM-driven approach for fine-grained localization of malicious payloads in Android malware, simultaneously generating corresponding behavioral explanations.
- We develop a demo app and analyze a real-world malware sample, enabling preliminary empirical evaluation at the method level and demonstrating the potential of *MalLoc* to advance analysis precision and explainability.
- To support reproducibility and future research, we publicly release the dataset and source code of *MalLoc* at: https://github.com/Trustworthy-Software/MalLoc

II. Background

Android Malware Analysis. Machine learning-based approaches [7], [9], [21]–[24] for Android malware analysis have been extensively explored over the past decade. More recently, researchers have begun investigating the potential of applying LLMs to this domain [25]. However, the majority of prior work remains focused on early-stage tasks—namely, malware detection (Stage 1 in Figure 1) and family classification (Stage 2 in Figure 1). Despite their importance, these early-stage techniques fall short of fulfilling the ultimate goal of malware analysis: enabling effective malware defense. This includes tasks such as dynamic modeling of malicious behavior, robustness testing of detection systems, and the design of targeted countermeasures. Bridging the gap between detection/classification and actionable defense requires a deeper, more granular understanding of how malicious behaviors are implemented and triggered within an application.

A critical missing bridge in this process is fine-grained malicious payload localization (Stage 3 in Figure 1)—the ability to identify and interpret specific methods or code segments responsible for malicious actions. Without this capability, security analysts are left with limited insight into the internal workings of malware, hindering both interpretability and mitigation efforts. The mission of *MalLoc* is to close this gap by advancing malware analysis beyond coarse-grained classification: *fine-grained localization and behavioral explanation represent the core* novelty *of our work*. *MalLoc* paves the way for more precise, explainable, and actionable malware analysis.

Smali Code. Android apps are primarily written in Java or Kotlin and compiled into DEX (Dalvik Executable) bytecode [26], which is stored in `.dex` files within APKs (Android Package files that bundle the compiled code and resources for distribution). Since these packages usually do not include the original source code, direct access to high-level code is not feasible. To enable analysis, tools such as `ApkTool` [27] can decompile the bytecode into Smali, a low-level, human-readable representation of DEX code. Smali serves as a practical intermediate format for examining app behavior when the original source is unavailable. Prior work has shown that LLMs can effectively interpret and reason about Smali code [28], motivating our focus on Smali-based analysis in this work.

During the compilation process, a single Java method can be transformed into multiple synthetic Smali methods. This occurs when Java features like lambdas, anonymous classes, or inner classes are compiled, as they are often translated into separate methods to support efficient runtime dispatch and maintain the language's object-oriented and functional behavior within the Android environment. For example, in our demo app, a single Java method, `onCreateView()`, was translated into five distinct Smali methods (as shown in Figure 2). This transformation implies the challenge of mapping malicious behavior to a single Smali method. Motivated by this observation, we design a two-phase localization approach: first identifying the malicious Smali class, then pinpointing the specific methods responsible for the behavior.

III. Approach

We begin with a simple baseline that applies LLMs directly to Smali classes. Based on its limitations (discussed later), we develop *MalLoc*, a more structured two-phase approach. As a preprocessing step, we decompile each APK using `ApkTool`, which converts DEX bytecode into Smali format.

Baseline. As a starting point, we implement a straightforward baseline approach that applies LLMs directly to Smali classes, prompting the model to identify and explain malicious behaviors from a predefined list without leveraging any structural context or multi-stage reasoning. The behavior

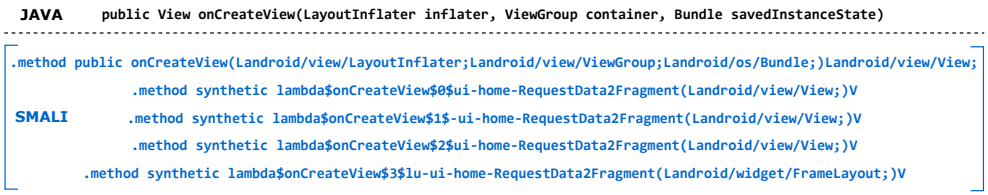

Fig. 2: Example of the translation of a Java 'onCreateView()' method into multiple Smali methods.

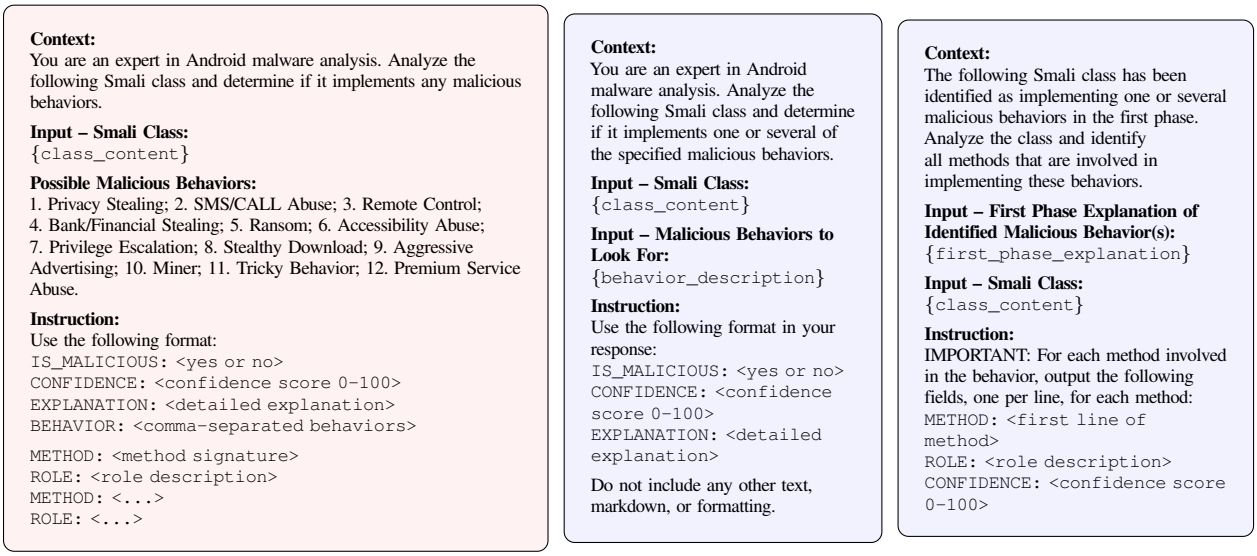

Fig. 3: Prompt templates used in: Baseline Approach (left), Phase 1 of MalLoc (middle), and Phase 2 of MalLoc (right).

list consists of 12 distinct malicious behaviors derived from MalRadar [20], a high-quality benchmark based on real-world Android malware, with manually verified family labels. In this dataset, each malware family is associated with certain particular behaviors from the list. The behavior list and prompt template used in this baseline approach are illustrated on the left side of Figure 3. As we demonstrate later in Section IV-B, the baseline approach yields poor performance. Through detailed analysis, we attribute this limitation to the inherent complexity of the task and the insufficient contextual guidance provided to the LLMs. Specifically, the baseline prompt implicitly requires an LLM to perform three tasks simultaneously: ❶ determine whether the given Smali class is malicious or benign; ❷ identify which malicious behaviors are implemented if the class is malicious; and ❸ localize the specific methods involved in each identified behavior. These challenges motivate our proposed method, *MalLoc*, described hereafter.

MalLoc. We propose *MalLoc*, a two-phase approach that decomposes the problem along semantic and structural dimensions. Inspired by prior work showing that decomposing complex tasks can improve LLM performance [29], we isolate each subtask and enrich the prompt with targeted context. As illustrated in Figure 4, *MalLoc* separates localization into two distinct phases: Phase 1 focuses on identifying malicious Smali classes associated with a specific behavior, and Phase 2 drills down to pinpoint the individual methods responsible for that behavior. This design enables *MalLoc* to achieve more accurate and explainable localization of malicious payloads. To implement *MalLoc*, we begin by creating detailed descriptions for each of the 12 predefined malicious behaviors, which are available in the replication package. Using metadata from MalRadar benchmark [20], we construct a family behavior lookup table that maps each malware sample to a subset of relevant behaviors based on its family label. The two-phase pipeline is applied iteratively to each behavior linked to the input malware sample. The prompt template for Phase 1 is shown in the center of Figure 3. It combines the Smali class content with the corresponding behavior description to prompt the LLM to determine whether the class implements that specific malicious behavior. If a class is flagged as benign in Phase 1, the process terminates for that class. If it is deemed malicious, the class—along with the explanation generated by the LLM—is forwarded to Phase 2. As shown on the right side of Figure 3, Phase 2 uses a second prompt template that incorporates the Phase 1 explanation to guide the LLM in identifying which methods within the class contribute to the specified behavior. The output includes both the method-level localization and a role description for each method. Finally, the results are subject to manual verification by security analysts to ensure accuracy and interpretability.

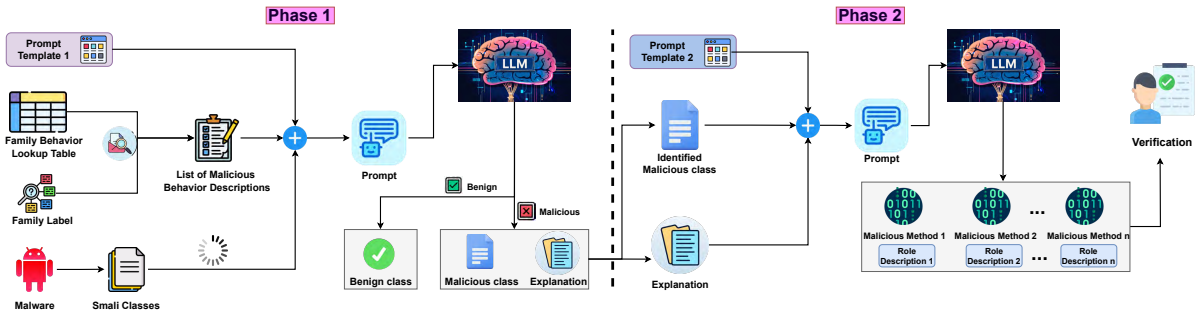

Fig. 4: The overview of *MalLoc* Pipeline.

IV. Experiments

A. Apps Under Analysis

A key challenge in evaluating malicious behavior localization is the lack of ground-truth datasets with fine-grained annotations. In the absence of such datasets, evaluating localization accuracy requires extensive manual effort to validate behavior predictions at both class and method levels—a time-consuming and labor-intensive process. As a result, we focus our evaluation on two representative Android apps that allow for reliable and interpretable assessment. Specifically, we use: ❶ *MalApp*, a controlled demo app we developed with 3 known malicious behaviors observed in real-world malware, and ❷ a real-world malware sample drawn from the MalRadar benchmark.

1) MalApp: This app is specifically designed to cover a diverse set of representative malicious behaviors under controlled conditions, enabling efficient verification by providing precise ground truth about where each behavior is implemented. The developer code of *MalApp* consists of 51 Smali classes and 165 methods, among which three malicious behaviors are implemented across 3 classes and 12 methods. It implements three distinct behaviors: ❶ *Privacy Stealing*, ❷ *Aggressive Advertising*, and ❸ *Tricky Behavior*. For *Privacy Stealing*, the app retrieves contact data from the device's storage and transmits it to an external server. *Aggressive Advertising* is implemented via fake click generation, where a transparent overlay tricks users into interacting with an advertisement URL. *Tricky Behavior* includes both label/icon manipulation—changing the app's icon on user interaction—and app hiding, where the app disappears from the launcher after a trigger. To ensure safety and ethical compliance, all external endpoints (e.g., the server receiving contact information and advertisement links) are simulated or non-functional, and the app is intended strictly for research purposes.

2) RuMMs App: For our real-world analysis, we select a malware sample from the largest family in MalRadar—RuMMs—which is annotated with five malicious behaviors: *Privacy Stealing*, *SMS/CALL Abuse*, *Remote Control*, *Bank/Financial Stealing*, and *Tricky Behavior*. We chose a small app in the family to minimize analysis complexity while keeping our experiments representative. The APK contains 21 classes and 66 methods, and masquerades as an MMS/SMS messenger by using a corresponding icon and name. The "AndroidManifest" lists 12 permissions—including `READ_CONTACTS`, `SEND_SMS`, and `BIND_ACCESSIBILITY_SERVICE`—and defines four activities, four services, and three broadcast receivers, suggesting a wide range of capabilities.

The app employs obfuscation techniques such as meaningless class and method names and limited reflection (e.g., classes named `a`, `b`, `Charge`, `NeglectDefend`, etc.). Upon launch, it prompts users to enable accessibility services and attempts to become the default SMS app. It maintains persistent background services, monitors system and banking apps, and exfiltrates user contacts. Although we could not confirm the exact intent of the SMS command mechanism due to an inactive C2 server, the app clearly performs unauthorized actions like SMS sending, remote monitoring, and user tracking. Unlike *MalApp*, reverse-engineering this malware required significant manual effort. Importantly, MalRadar provides only family-level behavior labels without payload locations or semantic explanations. Therefore, we manually verified all LLM-generated outputs to assess the accuracy and interpretability of *MalLoc*. This real-world case allows us to test *MalLoc* 's robustness beyond controlled environments. Due to the time-intensive nature of such manual verification, scaling to a broader set of real-world apps is left as future work.

B. Results

TABLE I: Comparison of Baseline and *MalLoc* on *MalApp*.

Method	Model	C-Prec	C-Rec	C-F1	M-Prec	M-Rec	M-F1
Baseline	Phi-4	0.00	0.00	0.00	0.00	0.00	0.00
	GPT-4.1	0.67	0.67	0.67	0.70	0.58	0.64
MalLoc	Phi-4	0.67	1.00	0.78	0.62	0.87	0.70
	GPT-4.1	0.83	1.00	0.89	0.65	1.00	0.76

1) MalApp: For our evaluation, we employed two LLMs: the open-source model Phi-4 [30] and the commercial model GPT-4.1 from OpenAI [31]. The performance results are presented in Table I, where the prefix **C-** indicates class-level metrics and **M-** indicates method-level metrics. A prediction is a true positive (TP) if it correctly identifies a malicious class or method with the right behavior; false positives (FP) refer to

incorrect behavior labels or misclassified benign components; false negatives (FN) denote missed malicious components.

The baseline approach performs poorly across both models. In particular, Phi-4 fails to produce any correct predictions. Upon manual inspection, we observed that although it successfully identified one malicious class, it misclassified the behavior type and failed to identify the relevant methods—resulting in zero scores across all metrics. While GPT-4.1 achieves slightly better performance under the baseline setup, its overall precision and recall remain low, especially at the method level.

In contrast, integrating the same LLMs into our proposed *MalLoc* framework yields significantly improved performance. As shown in Table I, both models demonstrate higher precision, recall, and F1 scores at both the class and method levels. Notably, with GPT-4.1, *MalLoc* correctly identifies all malicious classes and methods, achieving perfect recall and high precision. Specifically, it predicts 4 positive classes and 22 positive methods, compared to the total 165 methods in the app—reducing the manual analysis workload by approximately 87%. This highlights *MalLoc* 's strong potential to assist human analysts in accurately and efficiently localizing malicious payloads within Android applications.

2) RuMMs App: We apply *MalLoc* with GPT-4.1 to the selected real-world malware sample from the RuMMs family. Due to the absence of fine-grained ground truth in MalRadar, we cannot compute all the standard class- or method-level metrics such as recall and F1 Score. However, through detailed manual analysis, we found that *MalLoc* successfully identified 4 out of the 5 annotated behaviors associated with this sample. Specifically, out of the app's 21 Smali classes and 66 methods, *MalLoc* correctly localized 6 classes and 17 methods as malicious. All predictions and associated role descriptions were manually verified as accurate, resulting in 100% precision at both the class and method levels. This includes recognizing key operations such as exfiltration of contact information (*Privacy Stealing*), unauthorized SMS sending (*SMS/CALL Abuse*), background service persistence and remote interaction mechanisms (*Remote Control*), as well as auto-clicking consent dialogs (*Tricky Behavior*). An illustrative example prediction is shown in Figure 5.

Class: `b` (obfuscated)
Behavior: *Privacy Stealing*
Method: `.method public static f(Landroid/content/Context;)Ljava/util/ArrayList;`
Role Explanation: This method enumerates the user's contact list by querying the contacts content provider and extracting names and phone numbers.
Method: `.method public static a(Landroid/content/Context;ILjava/lang/String;)V`
Role Explanation: This method exfiltrates the sensitive data by embedding it into Intent extras and starting a background service.
Method: .....

Fig. 5: An example prediction by *MalLoc*, showing class-level behavior and method-level role explanations.

These findings underscore *MalLoc* 's ability to perform precise and semantically rich localization even in wild, obfuscated environments. Despite the lack of formal annotations, its outputs aligned well with expert analysis, suggesting strong potential to assist human analysts in real-world malware inspection.

V. Discussion

This work explores the feasibility of leveraging LLMs for the under-explored task of fine-grained malicious payload localization in Android apps. Rather than aiming for full automation, our goal is to support human analysts by narrowing the search space and reducing manual effort in identifying and understanding malicious behaviors. Our preliminary results demonstrate the promise of LLMs in this domain, particularly in enhancing interpretability and precision. At the same time, the study highlights key challenges that remain—such as improving efficiency, scaling to large apps, and calibrating confidence to guide analyst attention effectively.

Research Outlook. Looking ahead, we envision that LLM-driven localization can serve as a foundation for a broad range of downstream security tasks, including dynamic behavior modeling of localized payloads, explainable malware detection, and behavior-specific mitigation strategies. We hope this work will act as a catalyst toward building more interpretable, context-aware, and analyst-assistive malware analysis systems—bridging software engineering and security, and fostering stronger synergy between LLM-based reasoning and software maintenance practices.

VI. Conclusion

In this paper, we presented *MalLoc*, a novel two-phase framework that leverages LLMs for fine-grained localization of malicious payloads in Android apps. Unlike traditional work focused on coarse-grained detection or family classification, *MalLoc* enables a deeper understanding of malicious behaviors by identifying not only the presence of malicious logic at the class level but also pinpointing the specific methods involved, along with their functional roles.

Future Work. We identify several promising directions for future research. First, we plan to expand our evaluation to include additional LLMs and a more diverse set of real-world malware samples and families, in order to assess the generalizability of *MalLoc* across behavior types and obfuscation strategies. Second, we aim to develop mechanisms for confidence calibration and uncertainty estimation to better support human analysts in high-stakes security settings. Third, we intend to explore techniques for improving the efficiency and scalability of *MalLoc*, particularly when analyzing large and complex apps.

Acknowledgment

This research was funded in whole or in part by the Luxembourg National Research Fund (FNR), grant references 16344458 (REPROCESS) and 18154263 (UNLOCK).

References

[1] A. Turner, "How many android users are there? global and us statistics (2025)," https://www.bankmycell.com/blog/how-many-android-users-are-there, 2025, accessed: 2025-06-02.

[2] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *Security and privacy in communication networks: 14th international conference, secureComm 2018, Singapore, Singapore, August 8-10, 2018, proceedings, part i*. Springer, 2018, pp. 172–192.

[3] Y. Zhang, S. Torabi, J. Yan, and C. Assi, "Dynamic trigger-based attacks against next-generation iot malware family classifiers," *Computers & Security*, vol. 149, p. 104187, 2025.

[4] K. Tian, D. Yao, B. G. Ryder, G. Tan, and G. Peng, "Detection of repackaged android malware with code-heterogeneity features," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 1, pp. 64–77, 2017.

[5] P. Faruki, R. Bhan, V. Jain, S. Bhatia, N. El Madhoun, and R. Pamula, "A survey and evaluation of android-based malware evasion techniques and detection frameworks," *Information*, vol. 14, no. 7, p. 374, 2023.

[6] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, and S. Aonzo, "Unmasking the veiled: A comprehensive analysis of android evasive malware," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 383–398.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, no. 1, 2014, pp. 23–26.

[8] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin, "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, 2019, pp. 139–150.

[9] N. Daoudi, J. Samhi, A. K. Kabore, K. Allix, T. F. Bissyandé, and J. Klein, "Dexray: a simple, yet effective deep learning approach to android malware detection based on image representation of bytecode," in *Deployable Machine Learning for Security Defense: Second International Workshop, MLHat 2021, Virtual Event, August 15, 2021, Proceedings 2*. Springer, 2021, pp. 81–106.

[10] T. Sun, N. Daoudi, K. Allix, and T. F. Bissyandé, "Android malware detection: looking beyond dalvik bytecode," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 2021, pp. 34–39.

[11] T. Sun, N. Daoudi, K. Kim, K. Allix, T. F. Bissyandé, and J. Klein, "Detectbert: Towards full app-level representation learning to detect android malware," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 420–426.

[12] T. Sun, N. Daoudi, K. Allix, J. Samhi, K. Kim, X. Zhou, A. K. Kabore, D. Kim, D. Lo, T. F. Bissyandé *et al.*, "Android malware detection based on novel representations of apps," in *Malware: Handbook of Prevention and Detection*. Springer, 2024, pp. 197–212.

[13] F. Alswaina and K. Elleithy, "Android malware family classification and analysis: Current status and future directions," *Electronics*, vol. 9, no. 6, p. 942, 2020.

[14] C. Ding, N. Luktarhan, B. Lu, and W. Zhang, "A hybrid analysis-based approach to android malware family classification," *Entropy*, vol. 23, no. 8, p. 1009, 2021.

[15] H.-I. Kim, M. Kang, S.-J. Cho, and S.-I. Choi, "Efficient deep learning network with multi-streams for android malware family classification," *IEEE Access*, vol. 10, pp. 5518–5532, 2021.

[16] S. Freitas, R. Duggal, and D. H. Chau, "Malnet: A large-scale image database of malicious software," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 3948–3952.

[17] T. Sun, N. Daoudi, W. Pian, K. Kim, K. Allix, T. F. Bissyandé, and J. Klein, "Temporal-incremental learning for android malware detection," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–30, 2025.

[18] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "A multi-view context-aware approach to android malware detection and malicious code localization," *Empirical Software Engineering*, vol. 23, pp. 1222–1274, 2018.

[19] T. Sun, K. Allix, K. Kim, X. Zhou, D. Kim, D. Lo, T. F. Bissyandé, and J. Klein, "Dexbert: Effective, task-agnostic and fine-grained representation learning of android bytecode," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4691–4706, 2023.

[20] L. Wang, H. Wang, R. He, R. Tao, G. Meng, X. Luo, and X. Liu, "Malradar: Demystifying android malware in the new era," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 2, pp. 1–27, 2022.

[21] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," *arXiv preprint arXiv:1612.04433*, 2016.

[22] J. Liu, J. Zeng, F. Pierazzi, L. Cavallaro, and Z. Liang, "Unraveling the key of machine learning solutions for android malware detection," *arXiv preprint arXiv:2402.02953*, 2024.

[23] J. Samhi, L. Li, T. F. Bissyande, and J. Klein, "Difuzer: Uncovering suspicious hidden sensitive operations in android apps," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 723–735. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3510135

[24] M. Alecci, J. Samhi, L. Li, T. F. Bissyande, and J. Klein, " Improving Logic Bomb Identification in Android Apps via Context-Aware Anomaly Detection ," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 05, pp. 4735–4753, Sep. 2024. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TDSC.2024.3358979

[25] X. Qian, X. Zheng, Y. He, S. Yang, and L. Cavallaro, "Lamd: Context-driven android malware detection and classification with llms," *arXiv preprint arXiv:2502.13055*, 2025.

[26] "Dalvik executable format," https://source.android.com/docs/core/runtime/dex-format, accessed: 2025-06-02.

[27] "Apktool," https://apktool.org/, accessed: 2025-06-02.

[28] M. Alecci, N. Sannier, M. Ceci, S. Abualhaija, J. Samhi, D. Bianculli, T. F. d. A. BISSYANDE, and J. Klein, "Toward llm-driven gdpr compliance checking for android apps," in *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion'25)*, 2025.

[29] A. Prasad, A. Koller, M. Hartmann, P. Clark, A. Sabharwal, M. Bansal, and T. Khot, "Adapt: As-needed decomposition and planning with language models," in *Findings of the Association for Computational Linguistics: NAACL 2024*, 2024, pp. 4226–4252.

[30] M. Abdin, J. Aneja, H. Behl, S. Bubeck, R. Eldan, S. Gunasekar, M. Harrison, R. J. Hewett, M. Javaheripi, P. Kauffmann *et al.*, "Phi-4 technical report," *arXiv preprint arXiv:2412.08905*, 2024.

[31] OpenAI, "Gpt-4.1 api," https://openai.com/index/gpt-4-1/, 2025.