Reliable and Interpretable Android Malware Detection at Scale

Michael Tegegn
The University of British Columbia, Canada
mtegegn@ece.ubc.ca

Julia Rubin
The University of British Columbia, Canada
mjulia@ece.ubc.ca

***Abstract*—Machine learning approaches have shown impressive performance in Android malware detection. Yet, most if not all of these approaches face tradeoffs between accuracy, interpretability, and scalability. Approaches based on simple features are interpretable but miss complex behaviors. At the same time, approaches that capture holistic application patterns obscure the exact code responsible for malicious activity.**

In this paper, we outline our vision for an accurate, scalable, and interpretable method-level malware detection. The core idea behind our approach is to filter out non-discriminative application parts before analyzing the remaining, application-specific behaviors at the fine level of granularity. We further discuss the key challenges that must be addressed to effectively implement our proposed approach and provide suggestions for future directions.

I. Introduction

Malicious software impersonates users, penetrates their business organizations through their devices, and locks user data to demand ransom [1], [2], [3]. Specifically, for Android mobile applications, more than 100 malware detection techniques have been proposed in the past decade, to help identify mobile applications and prevent them from entering the official app store. Most of these approaches are based on Machine Learning (ML): they represent an app as a set of features and then employ a binary classifier to learn how to distinguish between malicious and benign app representations.

The earliest, yet still one of the most accurate and efficient of these approaches is Drebin [4], [5]. It extracts a set of categorical features (permissions, API calls, network addresses, intent filters, etc.) statically from APK files and uses these features to represent an app as a high-dimensional binary vector. It then used a binary classifier to distinguish benign vs. malicious apps.

While malicious vs. benign decisions made by Drebin are interpretable, i.e., they can be traced back to human-readable features, such as specific permissions or API calls, the tool only focuses on "easy to extract" app characteristics, without considering the full app functionality.

More recent approaches, e.g., [6], [7], [8], [9], [10] look at apps in a more holistic manner. For example, Iadarola et al. [6] converting Android applications into a visual representation (essentially, images) and leverages the power of Convolutional Neural Networks (CNNs), which are highly effective at image analysis. It further generates a heatmap that highlights the specific areas of the input image (i.e., the parts of the app's code) that the model focused on when making its decision. This allows a security analyst or researcher to see why the model classified an app as malicious.

Sun et al. [9] introduce a new deep learning framework, named DetectBERT, which uses a variant of Multiple Instance Learning (MIL) to treat classes of an application as an "instance", further aggregating them to create an app-level representation. This allows the model to analyze the relationships and interactions between different classes, which are often key indicators of malicious activity.

However, one of the main limitations of these approaches is that, to ensure scalability, they are quite aggressive in aggregating low-level details into higher-level representations. For example, Iadarola et al. [6] aggregates large fractions of the app's code into a single image, obscuring fine-grained details of the malicious code in the aggregated visual representation. Likewise, DetectBERT [9] aggregates all class code into a single representation, making it impossible to pinpoint which specific method, or line of code is responsible for the malicious activity. This limits the applicability of these tools in practice as an analyst cannot validate the decisions made by the tool.

Moreover, prior work [11] raises a question of the appropriateness of using binary classification for malware detection. This is because, in practice, malware applications can do everything benign software does (and more). For example, a classifier could learn that benign software often uses certain analytics libraries while malware does not (probably because malware developers have a lower incentive to analyze user behaviors and improve their apps). While such observation is statistically correct, it renders the detection unreliable and prone to simple repackaging attacks.

Inspired by these observations, our goal is to *investigate feature and classifier properties that lead to accurate, scalable, and interpretable malware detection*. Ultimately, we aim to devise (a) an interpretable feature set appropriate for dealing with mobile malware and (b) learning paradigms that are appropriate for malware detection and that can work at scale.

A naive idea for developing such a malware detection approach could be to 'feed' the entire application into an LLM-based agent and ask it to detect malicious parts of the application. Such an approach is unlikely to work as typical mobile applications contain over 30 million tokens, in around 30,000 methods per app on average. This makes direct LLM ingestion computationally infeasible and likely ineffective, as the model would be overwhelmed by irrelevant details.

2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00025

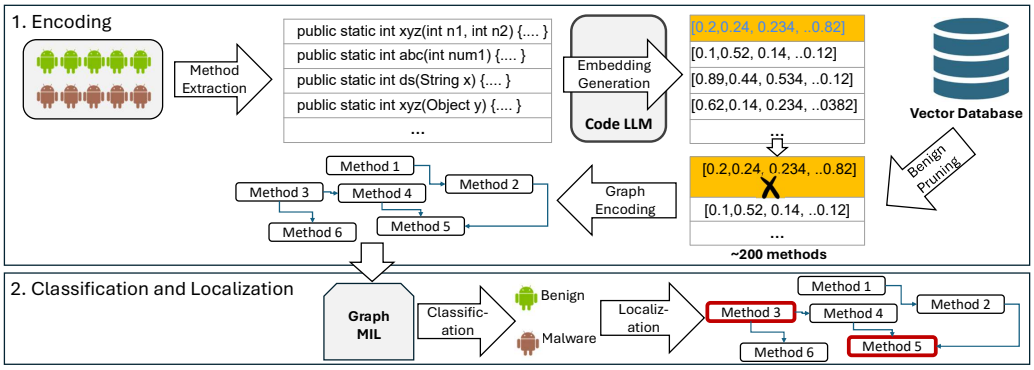

Fig. 1. Approach Overview.

Another relatively straightforward idea is to identify a semantically-meaningful representation of each application behavior, e.g., at a method level. Then, an application could be represented as a graph, where nodes are method representations and edges are code relationships between the methods. Malicious and benign graphs could be fed to a Graph-based Multi-Instance Learning (MIL) model [12], where each method is treated as an "instance" and the entire application graph as a "bag", similar to the approach taken by DetectBERT for classes. Graph-based MIL can learn complex malware patterns by identifying suspicious method-level sub-graphs and their interactions. However, such an approach faces major scalability issues due to a large number of methods in each app.

Inspired by this direction and the goal to provide scalable yet explainable malware detection approach, we investigated the method-level characteristics of a large set of malicious and benign apps from 2020 to 2023, downloaded from AndroZoo [13] – a popular repository with over 10 million Android apps from various markets. Specifically, we focused on malware and benign apps that appeared on Google Play, ensuring that our dataset is reliable and representative.

The results of our analysis show that a very large fraction of methods are shared between malicious and benign apps. This is likely because the apps rely on common libraries and frameworks, which are insignificant for malware detection. Our core idea is thus to filter out such methods, to focus the analysis on application-specific behaviors that better capture malicious intent and distinguish malware from benign software more precisely. Operating at the method level preserves interpretability, allowing us to trace predictions back to specific methods and reason about why an application is classified as malicious, while scaling the approach to apps of realistic size and complexity.

In what follows, we outline our proposed approach and the key challenges that must be addressed to effectively implement it in a scalable and interpretable manner.

II. Proposed Approach

Figure 1 shows the overview of our proposed approach. It consists of two stages. In the first stage, *Setup and Embedding*, we construct a fast-searchable repository of methods. Specifically, we extract all methods from a large corpus for benign and malicious applications, generate their embeddings using a code-aware LLM model, and store them in a vector database.

We further use the repository to identify and filter-out methods frequently used in both benign and malware applications using cosine-similarity-based neighbor lookup. The filtering step discards irrelevant context and non-informative methods, significantly reducing noise and providing a cleaner representation for subsequent MIL-based detection.

In the second stage, *Classification and Localization*, the remaining methods of each application (after pruning) are aggregated into higher-level structures and fed into a Multiple Instance Learning (MIL) framework. A straightforward representation is to treat them as a set of instances within the MIL framework, where each method independently contributes to the bag-level classification. Alternatively, richer structural information can be incorporated by devising a graph representation that captures relationships among methods (e.g., call relations, co-occurrence within classes or packages). This flexibility allows us to balance scalability with the richness of structural information preserved.

III. Discussion and Next Steps

We now discuss the main challenges in realizing the vision outlined in Section II.

1. **Method Embedding.** A critical component of our approach is the embedding function, as it determines how well method semantics are preserved and how effectively discriminative behaviors can be separated from common app functionality. To this end, we experimented with multiple embedding strategies, ranging from general-purpose code models (e.g., CodeBERT) to more recent LLM-based embeddings that incorporate both syntactic and semantic context.

Table I shows the results of our experiments on four sample methods, denoted C1-C4. These methods are shown in Figures 2-5, respectively. We selected these methods to evaluate the ability of LLM-based embeddings to distinguish between semantically similar and dissimilar code as the majority of app methods are actually getters and setters and such methods are commonly shared across samples.

In our examples, methods C1–C3 are semantically equivalent except for differences in identifier names, while method C4,

TABLE I
COMPARISON OF EMBEDDING MODELS FOR METHOD REPRESENTATION

	Cosine Similarity of Embeddings			
Model	**C1–C2**	**C1–C3**	**C1–C4**	
CodeBERT	0.9997	0.9995	0.9947	✗
ContraBERT	0.9348	0.7564	0.6335	?
CodeSage-Small	0.9996	0.9993	0.9985	✗
CodeSage-Base	0.9998	0.9993	0.9986	✗
CodeSage-Base-v2	0.9997	0.9992	0.9980	✗
UniXcoder	0.7941	0.6470	0.1367	?
CodeT5p	0.9695	0.9293	0.7670	●
StarCoder-1B	0.9830	0.9415	0.7905	●
Qwen3-Embed-0.6B	0.9153	0.8238	0.4563	?

despite being short and having similar syntactic properties, is functionally unrelated to the other methods. We measure the cosine similarity between the embedding of C1 and the rest of the methods. A desirable model should yield high similarity for the C1-C2 and C1-C3 pairs, low similarity for the C1-C4 pair, and relatively moderate "drop" in similarity for the C1-C3 pair compared with C1-C2.

```
public static String power(int f7)
{
  return Math.pow(2, f7);
}
```

Fig. 2. Method C1

```
public static String power2(int f8)
{
  return Math.pow(2, f8);
}
```

Fig. 3. Method C2

```
public static String power2(int inputNum)
{
  return Math.pow(2, inputNum);
}
```

Fig. 4. Method C3

```
public static void setHeight(int h) {
  this.height = h;
}
```

Fig. 5. Method C4

The results of this experiment show that many off-the-shelf embedding models lack the semantic awareness required for this task. Specifically, CodeBERT and all three CodeSage-based models we used in our experiments, failed to separate semantically dissimilar methods, producing near-perfect cosine similarity values even when comparing C1-C4. ContraBERT, UniXcoder, and Qwen3-Embed demonstrated a stronger ability to differentiate unrelated methods, though at the cost of possibly underestimating the similarity between semantically equivalent ones (e.g., C1-C3).

To estimate the overall overlap between methods of malware and benign apps, in this work, we utilized the CodeT5p embedding model due to its better alignment with expected similarity scores among candidate pairs. Yet, more work is needed to better evaluate the robustness of these models across a broader and more diverse set of malware and benign methods. Fine-tuning or developing a proprietary embedding model tailored to the malware detection setting, where malicious and benign applications often share substantial amounts of common library code, may also be necessary to achieve the level of semantic precision required for this task.

2. **Filtering.** We intend to experiment with other types of filtering beyond simple frequency-based removal. For example, one direction is to apply clustering over embeddings to identify and prune families of near-duplicate methods that appear across both benign and malicious apps. We are also considering to adapt a dynamic threshold on similarity, to adapt filtering depending on application size. Another option is to incorporate semantic filtering by leveraging pre-trained models to detect utility code (e.g., getters, setters) that is unlikely to carry discriminative malware-specific behavior.

3. **Graph Construction.** Following the filtering step, our malware detection pipeline involves performing graph MIL over method graphs. We believe graph MIL is more appropriate than the instance-based one as it allows encoding of relationships between methods as edges. To preserve the graph structure during the filtering step, we intend to compute and retain transitive relationships between nodes, even when intermediate nodes are removed.

4. **Classification.** In addition to Graph-based MIL, we intend to explore alternative aggregation and learning strategies that can operate at the method level. For example, we plan to experiment with attention-based MIL, which can weight more discriminative methods higher, highlighting the code most responsible for malicious behavior. Another direction is an outlier detection model, such as a one-class SVM, that focuses on learning malicious functionality while treating benign applications as counter-examples. This approach allows the model to identify behaviors that deviate from the benign baseline, capturing the asymmetry that malware often performs all benign behaviors plus additional malicious actions. We intend to explore the scalability and effectiveness of these approaches for performing accurate classification and are looking forward to further discuss these approaches with the research community.

IV. RELATED WORK

We now discuss related work, focusing on the two main dimensions below.

A. LLM Embeddings for Android and Malware Detection

Following the successful application of LLMs in many domains, recent works already experimented with LLM-based approaches to improve the representation and understanding of Android applications. DetectBERT [9], already extensively discussed in the Introduction section, builds a pre-trained model called DexBERT, which is specifically designed to understand the Smali code (a human-readable representation of Android bytecode). Sanchez et al. [14] introduce a contrastive-learning-based fine-tuning approach that facilitates learning embeddings that are aligned with expert-generated threat reports of known malware families. Specifically, they aim to make explanation text generated by decoder LLMs match that of human analysts by using the two texts as positive pairs in the contrastive learning pipeline. Their framework shows that contrastive learning that uses hard negatives from most similar samples

from different families helps separate the unique malicious behaviors in malware families.

Rahali et al. introduce MalBERT [15] and MalBERTv2 [16], which leverage a code-aware BERT-based architecture for Android malware identification. Their work adapts standard BERT-based model training approach in the context of malware detection by re-training the tokenizer to have more information about Android specific tokenization rules. Chaieb et al. propose BERTroid [17], an Android malware detection approach that uses the original BERT model to extract latent features from natural language cues present in Android permissions. Their results show that BERTroid can adapt to evolving permission sets, highlighting the importance of language models in extracting more robust salient features compared to binary encodings of code elements. We intend to further experiment with these methods in our work, towards identifying approaches that accurately capture semantic method-level code similarity.

B. Explainable Detection and Localization

Another research direction emphasizes explainability by focusing on fine-grained localization of malicious code. MKL-Droid [18] constructs multiple program-graph kernels and applies multiple kernel learning to not only detect malicious applications but also score individual components (e.g., classes, methods, packages) to identify those most responsible for malicious behavior.

AMCDroid [7] follows a similar goal of explainable detection, enhancing Android malware analysis through the identification of fine-grained malicious components by combining heterogeneous characteristics of code elements in a graph structure or detection. Droidetec [19] employs an attention-based sequence model, where the summation of API attention values allows the system to localize suspicious code fragments directly within the application. MsDroid [20] stores malicious code snippets in a reference database and matches them against candidate applications, thereby enabling the pinpointing of known malicious components with high precision. As discussed in Section I, the main limitation of these approaches is that they aggregate low-level details into higher-level representations, making course-grained explainability suggestions. Moreover, none of these approaches utilizes LLM embedding, rather basing their predictions on a set of pre-identified features. Such feature can miss the nuances nature of malware; we thus aim with our approach to leverages fine-grained, method-level embeddings that capture code semantics more accurately.

V. Conclusion

In this paper, we presented our vision for developing an accurate, interpretable, and scalable method-level Android malware detection. We discussed key challenges, including filtering non-discriminative code, designing embeddings that capture semantic differences, and selecting an appropriate learning paradigm for the task. Our analysis highlights the limitations of current off-the-shelf embedding models and the need for task-specific tuning to reliably distinguish malicious from benign behaviors. We hope our work will stimulate discussions and collaboration with the research community on developing practical solutions to the identified challenges.

References

[1] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *In Proc. of the IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

[2] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep Ground Truth Analysis of Current Android Malware," in *Proc. of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[3] M. Cao, K. Ahmed, and J. Rubin, "Rotten Apples Spoil the Bunch: An Anatomy of Google Play Malware," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2022, pp. 1919–1931.

[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014, pp. 23–26.

[5] N. Daoudi, K. Allix, T. F. Bissyandé, and J. Klein, "A Deep Dive Inside DREBIN: An Explorative Analysis Beyond Android Malware Detection Scores," *ACM Transactions on Privacy and Security (TOPS)*, pp. 1–28, 2022.

[6] G. Iadarola, F. Martinelli, F. Mercaldo, and A. Santone, "Towards an Interpretable Deep Learning Model for Mobile Malware Detection and Family Identification," *Computers & Security*, p. 102198, 2021.

[7] Z. Liu, L. F. Zhang, and Y. Tang, "Enhancing Malware Detection for Android Apps: Detecting Fine-Granularity Malicious Components," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1212–1224.

[8] J. Zheng, J. Liu, A. Zhang, J. Zeng, Z. Yang, Z. Liang, and T.-S. Chua, "MaskDroid: Robust Android Malware Detection With Masked Graph Representations," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2024, pp. 331–343.

[9] 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 *Proc. of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2024, pp. 420–426.

[10] T. Sun, N. Daoudi, W. Pian, K. Kim, K. Allix, T. F. Bissyande, and J. Klein, "Temporal-Incremental Learning for Android Malware Detection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pp. 1–30, 2025.

[11] M. Cao, S. Badihi, K. Ahmed, P. Xiong, and J. Rubin, "On Benign Features in Malware Detection," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1234–1238.

[12] M. Tu, J. Huang, X. He, and B. Zhou, "Multiple Instance Learning with Graph Neural Networks," in *Proc. of the IICML Workshop on Learning and Reasoning with Graph-Structured Representations*, 2019.

[13] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," in *Proc. of the International Conference on Mining Software Repositories (MSR)*, 2016, pp. 468–471.

[14] I. M. Sanchez, S. Mitra, A. Piplai, and S. Mittal, "Semantic-Aware Contrastive Fine-Tuning: Boosting Multimodal Malware Classification with Discriminative Embeddings," *arXiv*, 2025.

[15] Z. Xu, X. Fang, and G. Yang, "Malbert: A novel pre-training method for malware detection," *Computers & Security*, p. 102458, 2021.

[16] A. Rahali and M. A. Akhloufi, "Malbertv2: Code aware bert-based model for malware identification," *Big Data and Cognitive Computing*, p. 60, 2023.

[17] M. Chaieb, M. A. Ghorab, and M. A. Saied, "Detecting android malware: From neural embeddings to hands-on validation with bertroid," *arXiv*, 2024.

[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*, pp. 1222–1274, 2018.

[19] Z. Ma, H. Ge, Z. Wang, Y. Liu, and X. Liu, "Droidetec: Android malware detection and malicious code localization through deep learning," *arXiv*, 2020.

[20] Y. He, Y. Liu, L. Wu, Z. Yang, K. Ren, and Z. Qin, "MsDroid: Identifying Malicious Snippets for Android Malware Detection," *IEEE Transactions on Dependable and Secure Computing*, pp. 2025–2039, 2022.