TECHNISCHE UNIVERSITÄT BERLIN

Information Systems Management Degree Program in the
Faculty of Electrical Engineering and Computer Science

Master's Thesis

Transformer-based Mobile Malware Detection

BY

Joshua-Leon Dreger

Supervisor:
Prof. Dr. Konrad Rieck

Machine Learning and Security (MLSEC)

Berlin Institute for the Foundations of Learning and Data (BIFOLD)

March 2025

Abstract

For an overview of this document, see Chapter 4.

Provide a Thesis Abstract here (length: less than one page).

For a start, you may want to consult the concise instructions for writing the Summary Paragraph in *Nature*: https://www.nature.com/documents/nature-summary-paragraph.pdf. Moreover, consider Markus Kuhn's advice on differentiating abstract and introduction: https://www.lightbluetouchpaper.org/2007/03/14/how-not-to-write-an-abstract/.

A boilerplate scheme for an abstract is as follows: devote 25 % of the space on the purpose and importance of the research (introduction), 25 % of the space on what you did (methods), 35 % of the space on what you found (results), and 15 % of the space on the implications of the research (cf. https://writingcenter.gmu.edu/guides/writing-an-abstract).

More concrete advice for writing abstracts can be found on the website of the Writing Center of the University of North Carolina at Chapel Hill (https://writingcenter.unc.edu/tips-and-tools/abstracts/). Some useful phrases for abstracts can be found at http://dissertation.laerd.com/useful-phrases-when-writing-a-dissertation-abstract.php

Finally, you may also want to consider the excellent guide by Kent Beck on how to write good abstracts, which focuses on conference papers: https://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html.

Contents

1 Introduction

Here should be a text that states:

- Problem Statement
- Objective
- Research Contribution

2 | Related Work

2.1 Transformer

Transformer models have revolutionized how sequential data is processed in deep learning. Introduced by Vaswani et al. (2017) in the foundational paper "Attention Is All You Need" [Vas+23] , the Transformer architecture relies entirely on attention mechanisms. This was a paradigm shift from the previously prominent Recurrent Neuronal Network (RNN) [Elm90] or Convolutional Neuronal Network (CNN) [LeC+89] approaches. Transformer enables parallel processing of sequence elements and captures long range dependencies more effectively than RNN based approaches. At its core, a Transformer uses a mechanism called self-attention to weigh the importance of different tokens in a sequence relative to one another, allowing the model to focus on relevant context regardless of its positional distance. This ability to draw global dependencies between input and output makes Transformers powerful for tasks like machine translation, where the entire input sequence informs each output element.

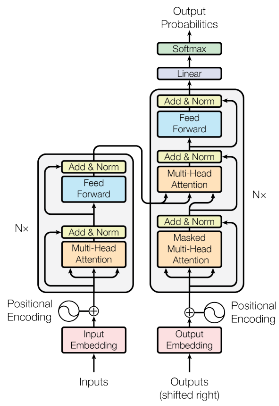

Figure 2.1: Taken From [Vas+23]

In a self-attention layer, the model computes attention scores between every pair of tokens in a sequence. Each token is first converted into an embedding vector, by looking up the token in the embedding matrix. This embedding matrix has a row for each possible token in the vocabulary of the model. Each row is the according embedding vector that represents this token. The embedding vector has a length of the "hidden size" hyperparameter of the model. From each embedding vector then a Query (Q), Key (K), and Value (V) vector is derived. This is done by multiplication of the embedding vector with model specific learned projection matrices (One matrix for each Q, K & V). Next attention scores are obtained by multiplying Q and K vectors via a dot product. These scores determine how much attention one token should pay to another when constructing its next representation. After a softmax normalization layer, each token V vector aggregates information from all other tokens V vectors, weighted by these attention scores. Finally, this weighted sum of V vectors produces a new contextualized embedding vector for that token, capturing relationships between words dynamically. Transformers employ multi-head attention, meaning this process is replicated in parallel multiple times (with different learned projection matrices). Multi-head attention allows the model to attend to different patterns or aspects of the data simultaneously, capturing different kinds of relationships in the sequence. The outputs of multiple heads are then combined, enabling richer representations than a single attention operation. Because the Transformer has no recurrent notion of sequence order, it adds positional encodings to token embeddings to include information about each tokens position in the sequence.

The original Transformer architecture is an encoder-decoder model, consisting of a stack of encoder layers and a stack of decoder layers [Vas+23]. Each encoder layer has two main sublayers: a multi-head self-attention sublayer that allows each input token to attend to others, and a feed-forward network sublayer that further transforms each tokens embedding vector (each sub-layer is wrapped with residual connections and layer normalization for stability). Stacking multiple such layers produces an encoder that maps an input sequence into a sequence of high level feature vectors. On the other side of the encoder-decoder, each decoder layer also contains a self-attention sublayer (applied to the decoders own inputs generated so far) and a feedforward sublayer, but additionally includes an encoder-decoder attention sublayer (also called cross-attention). This cross-attention allows the decoder to attend to the encoders output at each decoding step, effectively using the encoded source sequence context when predicting each token. The decoder generates one output token at a time, and each token embedding can only attend to earlier token embeddings (preventing a token from "seeing" future tokens during training). This encoder-decoder design proved extremely effective for sequence-to-sequence tasks like neural machine translation, where the encoder processes an input sentence into a context representation and the decoder generates an output sentence using that context [Liu+20; Yan+23].

While the original Transformer uses both an encoder and a decoder, many modern applications use only one of the two stacks, depending on the task.

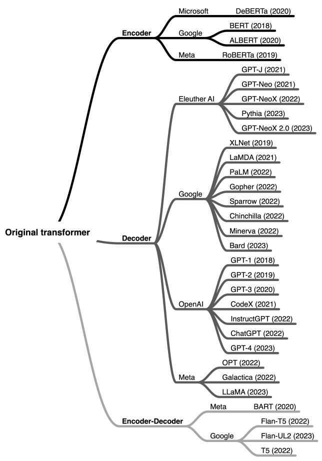

Figure 2.2: Taken From https://magazine.sebastianraschka.com/p/understanding-encoder-and-decoder

Encoder only Transformer architectures are designed exclusively with encoder components, leaving out the decoder segments present in encoder-decoder models. This design choice makes them particularly effective for tasks that require understanding and representation of input data without the necessity for sequence generation. "Bidirectional Encoder Representations from Transformers" (BERT) [Dev+19] is a popular example BERT is an encoder-only Transformer architecture that converts input text into a sequence of vectors representing the input text. It consists of multiple layers of encoders, each containing a self-attention mechanism and a feed-forward neural network. In BERTs pretraining, some tokens in the input are masked at random and the model must predict these missing tokens (Masked Language Modeling), and it also learns to predict if one sentence naturally follows another (Next Sentence Prediction). Through this process, BERT learns contextual embeddings that capture subtle relationships in language, as each tokens representation is influenced by the tokens on both its left and right. This bidirectional conditioning (as opposed to the one directional nature of a decoder) enabled BERT to achieve better performances on a wide range of NLP tasks once finetuned [Sal+20]. In BERT's architecture, there is a special [CLS] token inserted at the beginning of each input sequence, serving as an empty sheet to be contextualized by the entire sequence for classification tasks. During training, the final hidden state corresponding to the [CLS] token is utilized to capture the overall meaning of the input. The absence of a decoder in encoder only models means they are not typically used for generating new sequences from input data. Instead, they are used in scenarios where the goal is to derive meaningful representations or classifications from the input data.

Decoder-only Transformer architectures are built exclusively using decoder layers, leaving out encoder layers entirely. Such models are primarily designed for tasks that involve generating sequences, such as predicting text or continuing a given sequence. A prominent example is "Generative Pretrained Transformer" (GPT) [Bro+20; Ope+24; Rad+19; RN18], which is commonly used for text generation, summarization, and other creative writing tasks. Each decoder layer includes two key parts: a self-attention mechanism and a feed-forward neural network. The self-attention allows the model to consider previously generated tokens when predicting the next one, helping it maintain context and coherence across a sequence. However, unlike encoder models, decoder-only models use a form of attention called "masked attention". This means each token can only pay attention to tokens that came before it, preventing the model from "seeing" future tokens during training or generation. Decoder-only models like GPT learn by predicting the next token in a sequence, effectively teaching the model patterns in language and context over large datasets. Once trained, these models can generate coherent and contextually relevant sequences of text, making them powerful tools for tasks such as storytelling, dialogue systems, or code generation. Although decoder-only models excel at generating new text, they are usually less effective for tasks that require deep understanding or classification of input data compared to encoder or encoder-decoder models [QMN24].

Since "Attention Is All You Need", Transformer architectures have undergone numerous enhancements and there are a lot of different variants. Early on, researchers scaled Transformers to larger sizes and data (e.g., Llama 2 [Tou+23], Llama 3 [Gra+24] for text generation) and created optimized versions of BERT (such as RoBERTa [Liu+19], ALBERT [Lan+20], and DistilBERT [San+20]) to improve training effectiveness or model efficiency. A major focus has been on addressing the quadratic complexity of standard self-attention with respect to sequence length [Xio+21]. Vanilla Transformers compute attention between all pairs of n tokens, which scales with $O(n^2)$ in time and memory needed. This becomes a problem for long sequences like lengthy documents or code analysis (e.g., an entire mobile apps code can consist of thousands of tokens). Starting around 2020, new and more efficient Transformers emerged to tackle this limitation.

One notable example of an encoder only Transformer optimized for long text processing is Longformer [BPC20]. Longformer addresses the struggle of quadratic complexity by implementing a sparse attention pattern, allowing it to scale efficiently while preserving contextual understanding. Instead of each token attending to all others, Longformer uses a sliding window attention mechanism, where each token attends only to a fixed number of nearby tokens. Additionally, it employs global attention on select tokens (like the CLS token), enabling the model to capture broader contextual information across the sequence. This combination ensures efficient processing while keeping the ability to model long range dependencies. As a result, Longformer can process sequences of up to 4,000 tokens with a computational cost that grows linearly, rather than quadratically, with input sequence length. These design choices are supposed to make it effective for tasks such as document classification, question answering, and summarization [BPC20].

Another similar approach is BigBird [Zah+21], which also uses sparse attention mechanism to efficiently handle long sequences while maintaining context. Like Longformer, BigBird reduces the quadratic complexity of traditional self-attention by limiting the number of attended tokens. However, BigBird extends the sparse attention paradigm by incorporating three types of attention: local attention, where each token attends to its nearest neighbors; random attention, where tokens attend to a few randomly selected tokens; and global attention (like the CLS token), where a small set of tokens attend to all others. This approach ensures that every token can at least indirectly attend to all others, maintaining the model's ability to capture long range dependencies. BigBird can also process sequences up 4000 tokens, while maintaining linear complexity. The authors claim that it is particularly suitable for long document NLP tasks such as question answering, summarization, and classification. Compared to Longformer, which primarily relies on windowed local attention and selected global tokens, BigBirds use of random attention supposedly allows for a better balance between efficiency and the ability to propagate information across distant tokens. Empirical studies have shown that BigBird outperforms Longformer in tasks requiring deep contextual understanding across long documents, such as multi-hop question answering and document classification [Zah+21].

Another upgreade is promised by ModernBERT [War+24]. ModernBERT is an optimized encoder only Transformer model that builds upon BERT while employing showing improvements in efficiency, scalability, and sequence length. Unlike the original BERT, which was limited to 512 tokens, ModernBERT extends its native sequence length to 8,192 tokens, making it more capable of handling long context tasks. This is achieved without introducing sparse attention mechanisms like Longformer or BigBird, instead using more efficient attention computation, better normalization techniques, and optimized training strategies. A key feature of ModernBERT is its use of alternating attention layers, where some layers apply full global attention, while others apply localized attention to reduce computational overhead while maintaining strong contextual understanding. Additionally, Rotary Positional Embeddings replace BERTs absolute positional embeddings, allowing ModernBERT to generalize more effectively to long sequences. To further enhance efficiency, normalization is used before rather than after the attention sublayer, stabilizing training and improving gradient flow. ModernBERT was trained on 2 trillion tokens, covering diverse domains such as long text documents, retrieval tasks, and code based datasets. Unlike previous BERT-like models, ModernBERT removes padding tokens during computation (a technique known as "unpadding"), reducing redundant operations and making inference more memory-efficient. These optimizations allow ModernBERT to match or surpass SOTA results across classification, retrieval, and multi-vector search tasks, while remaining faster and more scalable than its predecessors. Compared to Longformer and BigBird, which rely on sparse attention mechanisms to scale to long sequences, ModernBERT maintains full bidirectional attention, ensuring stronger contextual modeling. While Longformer employs local windowed attention with select global tokens, and BigBird combines local, random, and global attention, ModernBERT balances efficiency end effectiveness through improved attention computation and optimized training techniques. This makes it a more general purpose solution, excelling in classification, retrieval, and document understanding, while remaining highly efficient for real world deployment on common GPUs [War+24].

Another approach is Nyströmformer [Xio+21] by Xiong et al. that applies the Nyström method (a technique for matrix approximation) to the self-attention computation. Instead of computing the full $n \times n$ attention matrix, Nyströmformer samples a set tokens and uses them to construct an approximate attention matrix at only O(n) cost. This approximation allows the model to scale linearly with sequence length, a big improvement in efficiency. Nyströmformer reports competitive accuracy on many NLP tasks compared to standard Transformer, despite using only a fraction of the computation. This kind of efficient attention is especially relevant for resource constrained settings. For instance, the recent DetectBERT [Sun+24] approach for Android malware analysis leveraged a Nyströmformer layer to handle a large amount of Embeddings at once (more details in section 3.2.2). By using Nyströmformer, DetectBERT reports to process each app in a single forward pass (up to 8000 tokens) with only 2 GB of GPU memory, achieving inference times on the order of 5 milliseconds per app.

2.2 Malware Detection

Malware detection approaches are commonly categorized into static, dynamic, and hybrid analysis methods [Rie24]. Each approach inspects malware from a different angle, offering unique advantages and tradeoffs in terms of accuracy, and efficiency. While this thesis focuses on static analysis, it is worth noting the other approaches for context. Static analysis examines APK features such as source code and binaries without executing the app. The key advantage of static analysis is efficiency and scalability, since no runtime environment is needed, large volumes of files can be scanned quickly. Prior research like [Arp+14] and [Sun+24] demonstrate the effectiveness and efficiency of static approaches. However, static analysis does face challenges, including evasion techniques. Obfuscation and encryption can hide malicious code from static detectors.

Dynamic analysis, in contrast, executes the program in a controlled environment such as a sandbox[Blä+10], to observe its behavior at runtime [Rie24]. During execution, the analysis may monitor network traffic, file modifications, memory usage, and other behavior that may reveal malicious actions [Wol22]. A big advantage of dynamic analysis is that it can detect malicious behavior even if the code is obfuscated. This makes dynamic analysis more robust than static approaches. Several research systems like TaintDroid [Enc+14] or DroidScope [YY12] use this approach However, dynamic analysis is computationally intensive and slower. Running each APK in a virtual machine or virtual environment leads to a lot of computation, scaling this to thousands of files is very challenging. Additionally, it is vulnerable to other evasive strategies, such as sandbox detection [VC14].

Hybrid techniques that integrate these methods aim to combine their strengths while weakening individual limitations. Here both a parallel and integrated approach are possible. In a parallel approach the malware set might first be clustered statically where the expensive dynamic approach then is only applied to high risk malware. Alternatively also both static and dynamic features might be evaluated by the same model. Hybrid analysis too, faces challenges such as computational and theoretical complexity. This thesis, focuses on static analysis as a scalable and efficient approach for malware detection.

Malware authors often employ evasion techniques to bypass detection mechanisms [Rie24]. In static analysis, common strategies include manipulating code structures through obfuscation [Bac+18] and polymorphism [Cat+22]. Obfuscation involves transforming code into a more complex or less readable form to conceal its true purpose, whereas polymorphism allows malware to dynamically change its appearance while maintaining its core functionality. These approaches exploit the limitations of analyzing code without running it. While dynamic and hybrid methods address some of these weaknesses, they introduce their own vulnerabilities, such as susceptibility to sandbox detection and reliance on communication pattern anomalies. Transformers ability to handle obfuscation and learn robust representations of static code helps mitigate some of these challenges [Roz+21]. In addition to Transformers having the potential to be robust to obfuscation, they also show capabilities of deobfuscating code through encoder decoder systems as discussed in chapter 2.1.

Another challenge in the realm of malware detection is concept drift due to malware authors continuously evolving their techniques. Concept drift refers to the evolution of malware patterns, which often leads to a decline in the performance of detection models over time. Work like Transcend [Jor+17] have tackled this issue by introducing mechanisms to detect and adapt to concept drift during deployment. However, these methods are resource intensive and may struggle to generalize across diverse datasets. Based on the Transcend approach the same research group published work on "Transcending Transcend" [Bar+24], where they used their mechanism to construct a dataset. Addressing concept drift remains a critical challenge for static Android malware detection and is one criteria used in this thesis. Some works have proposed automated drift detectors that trigger model updates when the input data distribution shifts [Ala+24].

Somewhat similar to the challenge of concept drift is the challenge of biased datasets. Tesseract [Kan+24] shows, how datasets are always biased and never represent the true distribution of software. The authors differentiated bias in spatial and temporal bias. The spatial bias stems from unrealistic class distributions, such as an artificially high proportion of malware, which do not match real world conditions. Temporal bias bases on concept drift and occurs when the data is trained and evaluated on the same timespan. In real world scenarios malware detectors will always be trained until a certain point in time and will then be used to process new software. This means that a model has to be somewhat robust to concept drift. To account for temporal bias the authors propose to split the dataset that is used in a temporal manner, where the training data is comes before the evaluation data. While still being biased, this time based split makes the evaluation of models more realistic.

2.2.1 Android Malware Detection

Detecting malware on the Android platform presents unique challenges and opportunities due to its open ecosystem and the high volume of apps and malware samples. The availability of many labeled datasets such as Androzoo, Drebin, Transcend, and DexRay has driven research in this domain.

Drebin [Arp+14] is a static analysis method for detecting Android malware using machine learning. It extracts features from an apps manifest and code to build a set of descriptive strings. These features are encoded in a high dimensional vector space, where each APK is represented by a sparse binary vector. Drebin uses a linear Support Vector Machine (SVM) to learn a decision boundary that separates benign apps from malware. The method detects about 94% of malware samples while keeping the false positive rate around 1%. It is designed to be lightweight and can run directly on smartphones, typically taking about 10 seconds per application. The approach also provides explanations by showing the most important features that led to the detection. This helps users understand why an app is flagged as malicious. In addition, the Drebin dataset, which includes labeled malware samples and complete APK files, is a foundational resource for further research.

Transcending [Bar+24] introduces a dataset that spans five years of real-world Android applications. The dataset includes thousands of malware samples collected from sources like AndroZoo. It is split into training and testing sets that represent different time periods based on the work of Tesseract [Kan+24]. The study showed that static detection models like Drebin trained on older data lose accuracy as new malware emerges. Based on the work of Transcend [Jor+17] it introduces a rejection mechanism to identify samples that do not match the training distribution. This approach helps prevent misclassification by flagging drifting samples. By using this comprehensive dataset, the work highlights the impact of concept drift on static malware detection. It has set a new benchmark for evaluating the robustness of detection models over extended periods.

In this thesis we will call the dataset that was used in the Transcending paper Transcend.

DexRay [Dao+24] is a deep learning approach that detects Android malware by converting app bytecode into a grey-scale vector image. It first extracts the raw DEX bytecode from an APK file and then transforms the bytes into a one dimensional image. This image is resized to a fixed size and fed into a simple one dimensional convolutional neural network for classification. The model is evaluated on a dataset of 158,803 Android applications collected from AndroZoo. The dataset spans apps from 2019 to 2020 and includes both benign and malicious samples. Benign apps are defined as those not flagged by any antivirus on VirusTotal [1], while malware apps are identified if at least two antivirus engines detect them. A key feature of the DexRay dataset is, that it includes both non obfuscated and obfuscated APKs. This design is supposed to allow researchers to study the impact of code obfuscation on detection performance. DexRay achieves an impressive F1-score of 96%, demonstrating its effectiveness in malware detection. Its simple yet effective approach makes it a solid baseline for malware detection.

Androzoo [All+16] is a large repository that contains millions of Android apps collected from various app stores. It spans more than a decade of Android development, which makes it a valuable resource for studying long term trends. It includes both benign and malicious APKs, in addition to metadata for each app, such as compilation dates, sizes, antivirus reports and more. This extra information enables further analysis of the Android app landscape. The scale and diversity of Androzoo have helped improve the reproducibility of research studies. Researchers can use Androzoo to track how malware evolves over time and to test the effectiveness of detection methods. Androzoo enabled the research of this thesis by providing an API key to download APKs from its repository. This key was then used to download the DexRay dataset to analyze it together with the previously mentioned ones in chapter 3.1.

2.3 Transformer based malware detection

Recent research in cybersecurity has produced a range of Transformer based models for malware detection. Many of these novel approaches build on the BERT [Dev+19] architecture or its variants, adapting them to capture the semantics of malicious code and improve detection performance. For example, MalBERT [RA21] uses a BERT variant that is finetuned on Android app manifest files for static analysis. By treating an apps manifest (which lists permissions, components, etc.) as input text, MalBERT achieved excellent results, with about 97% accuracy and an F1 score around 95% in distinguishing malware from benign apps, even when reproduced by other authors [Sou+22]. Its success indicates that Transformers can leverage even simple limited textual features (like manifests) for malware detection. However, focusing only on manifest content may miss code level clues e.g. obfuscated or malicious code could

[1] https://www.virustotal.com/gui/home/upload

be overlooked if not reflected in the manifest. To address this, a followup model MalBERTv2 [RA23] extended the approach to be code-aware. MalBERTv2 processes actual app code by extracting the most informative code files and applying a custom tokenization before using a BERT based model. This richer static analysis led to improved performance across multiple datasets: MalBERTv2 reports an F1 score of 97% when evaluated on a mix of Android malware datasets. On smaller subsets, MalBERTv2s F1 score ranged from about 82% up to 94%. These BERT based detectors demonstrate the Transformers ability to learn meaningful features from app data (manifest or code), resulting in high precision and recall. A limitation is that their large size (e.g. BERT base with 110 million parameters) can be expensive to execute.

Purely codebased is another approach called DetectBERT [Sun+24]. DetectBERT is a transformer based approach to Android malware detection that builds upon DexBERT [Sun+23], another transformer based model designed for smali class level analysis of Android Smali code. Using a Correlated Multiple Instance Learning (c-MIL) framework, DetectBERT aggregates Smali class embeddings of DexBERT into an app level representation. The DetectBERT model is further analyzed in Chapter 3.2.2 where it is evaluated as baseline.

Another recent model, BERTroid [CGS24] , further showcases Transformers potenital in Android malware detection. BERTroid is based on a BERT architecture and integrates neural embeddings from both static and dynamic analysis to classify apps. It was evaluated on several public datasets. BERTroid is reports to outperform prior SOTA solutions, achieving near perfect detection rates for instance, an Accuracy of 99.74% and F1-score of 99.87% on one benchmark, with precision and recall also around 99.87%. Such results indicate a very high true positive and true negative rate. The authors note BERTroids resilience against concept drift, claiming it maintains high performance even as malware characteristics change over time. This hints that training on diverse and up to date data empowered the model to be robust to concept drift. It should be noted, however, that achieving nearly 99%+ accuracy in malware detection can sometimes be a sign of evaluation on simplified scenarios real world deployment may see lower numbers due to novel malware samples or adaptive attackers. Nonetheless, BERTroid underscores the effectiveness of Transformer encoders in this domain.

Similarly, Liu et al. propose SeMalBERT [Liu+24] , which focuses on learning semantic representations of code using BERT. SeMalBERT achieved high detection rates, with accuracy improving from about 88.6% up to 98.75%. This suggests that capturing semantic meaning makes the detector more robust to code obfuscation and variants, since semantically similar malware will be clustered in the Transformers feature space even if their surface syntax differs. Roziere et al. showed that pretrained Transformers can deobfuscate code and recover its functionality to an extent [Roz+21] , underscoring the potential for resilience against obscured or polymorphic code. One key advantage is their ability to learn deep, contextual representations of code and API usage, which helps in resisting many common obfuscation techniques. Traditional static detectors often rely on specific keywords, signatures, or simple features that malware authors can easily obfuscate (e.g., renaming variables, reordering code, ...). Transformers, by contrast, attend to the broader context and meaning of code tokens. That said, Transformers are not impervious to obfuscation, especially sophisticated obfuscation that changes control flow or encrypts malicious payloads. In such cases, the model might not see any learned pattern unless it has been trained on similarly obfuscated examples.

Another strength of Transformers is their capacity to incorporate multiple feature types. As seen, some models combine textual code features with other modalities (images, graphs, etc.), and the attention mechanism can naturally combine information from different sources. This is valuable in malware detection, where one may want to consider app permissions, API calls and graphs together. The HTGT (Heterogeneous Temporal Graph Transformer) [Fan+21] is a prime example: it jointly models an entire Android malware ecosystem as a graph with various entity types (apps, developers, markets) and learns relations both spatially (graph connections) and temporally (evolution over time). By using a specialized Transformer on this heterogeneous graph, their system (called Dr.Droid) can detect malware while explicitly tracking how malware campaigns evolve and propagate. This approach directly tackles concept drift: as new malware appears, the temporal graph Transformer updates the representation of the "drifting" nodes, helping to catch novel threats that differ from past training data. In evaluations on large scale industry data from Tencent Security, HTGT significantly outperformed baseline detectors on identifying evolving Android malware, and notably Dr.Droid has been deployed in production, protecting millions of users in Tencents ecosystem [Fan+21] . Such real-world deployment indicates the approaches scalability and effectiveness against concept drift in practice. The tradeoff, is complexity, constructing and maintaining a temporal graph of the entire app ecosystem and running a graph transformer is resource heavy and may only be practical for big providers with extensive data (like app store operators). For most use cases, a standard code focused Transformer like BERT or Longformer, periodically retrained on newly collected samples, may suffice to handle gradual concept drift.

Beyond the Android realm, Transformers have also been adapted for other cybersecurity tasks, illustrating their generality. For example, in Windows malware detection, Li et al. introduced I-MAD [Li+21] , a framework using a Galaxy Transformer network to analyze binary assembly code at multiple levels (basic blocks, functions, executables). This approach can not only classify Windows executables as malicious or benign, but also provide interpretability by pinpointing which code segments contribute to the detection.

Another line of work leverage Transformers to evaluate different kinds of APK feature types. Ullah et al. [Ull+22] combine textual and visual features in an explainable Android malware detector: they finetune BERT on textual logs and simultaneously convert network traffic data into images, then use a Vision Transformer (along with CNNs) to detect malicious patterns. While these hybrid systems go beyond pure Transformer classifiers, they highlight the utility of Transformer based feature extractors in cybersecurity, ranging from purely static code analysis to network behavior analysis.

When comparing the performance of these Transformer based solutions, we find they often surpass traditional machine learning methods in malware detection accuracy. Across the board, models like MalBERT, BERTroid, DetectBERT, and MalBERTv2 report substantially higher F1-scores than earlier approaches based on fixed features or classical classifiers [CGS24] For instance, MalBERTv2 obtained an average F1 around 97%, whereas a classical SVM baseline might linger much lower (in one comparison, SVM was 82% F1 [RA23]). Even when Transformers are applied to more challenging tasks like family classification or zero day malware, they tend to perform impressively due to their representation learning strength. That said, direct metric comparisons should be made with caution: studies use different datasets (Drebin, DexRay, Transcend, etc.), and some datasets are far more challenging than others. This has been proven during the baseline creation of this work where the same very basic algorithm achieved performances varying from 8% to 94% depending on the dataset it was used on (table 3.3 in section 3.2).

Efficiency metrics are also crucial, model size, inference time, and memory usage vary in between models and approaches. A vanilla BERT based model might require a GPU and a few seconds per app to extract features, whereas an optimized approach using Nyströmformer [Xio+21] or distilled models can cut this dramatically [Sun+24]. For deployment at large scale, even small differences in speed are significant. In practice, industry solutions likely use a combination of cloud based heavy models and on device lightweight models. Large cloud providers (Google, Tencent, etc.) can afford to run Transformer detectors on their backends to check apps (the deployment of Dr.Droid is an example [Fan+21]), whereas on mobile devices the malware scanning must be conservative in resource use.

Lastly, security against adversarial attacks is an emerging concern [Rie24]. Transformers can be fooled by adversarial examples malicious inputs subtly modified to evade detection. Real world deployment must consider robustness techniques (adversarial training, ensemble models, rule based checks) alongside the Transformer to prevent adversaries from exploiting the learned patterns. In summary, the landscape of Transformer based architectures for mobile malware detection is rapidly evolving, with novel models pushing the SOTA in accuracy and adaptability. From BERT inspired models like MalBERT, BERTroid, and SeMalBERT that excel at static code analysis, to efficiency oriented Transformers (Longformer, Nyströmformer) enabling large scale and long sequence processing, these approaches greatly enhance our ability to detect malware under the challenging conditions of obfuscation, concept drift, and huge app volumes. They consistently demonstrate higher precision and recall than earlier methods, often achieving F1-scores above 0.95 on benchmark datasets [CGS24; RA23; Sun+24]. Their strengths lie in modeling complex patterns and contexts other models cannot. Yet, there are trade-offs to consider: Transformers demand substantial computational resources. . Ongoing research is addressing these issues through model compression and specialized architectures. Importantly, some Transformer based detectors have matured to real world deployment (protecting users at scale [Fan+21]), highlighting their practical value. As the malware arms race continues, Transformer architectures, with their adaptability and power, are capable to play a central role in the next generation of mobile malware defense systems.

A deeper survey of transformer based malware detection is given by Alshomrani et al. [Als+24].

3 | Methodology

3.1 Dataset Evaluation

The evaluation of datasets forms a critical foundation for the success of any machine learning task, especially in security sensitive applications such as mobile malware detection. A robust dataset not only enables effective training of models but also ensures generalizability across various scenarios, including concept drift. This section outlines the dataset evaluation process undertaken for this research.

The datasets that are evaluated are introduced in subsection 2.2.1. The Drebin, Transcend, and DexRay datasets are well suited for comparison because they are specifically designed to serve as benchmarks for training and evaluating models on android malware detection. Each dataset is designed to train specific models and assess their performance. Androzoo is notably different because it serves as a large repository for Android APKs rather than a closed dataset specifically designed for training and benchmarking algorithms. Additionally, it is continuously updated and expanded by the authors.

When comparing the number of APKs of each dataset, Androzoo has the most APKs with 25,059,602 (as of 06.103.2025 14:23). Since it is a repository, it will continue to grow larger. The biggest of the benchmarking datasets is Transcend, which has 259,230 available APKs. This is followed by DexRay with 159,803 APKs, and then Drebin with 129,013 APKs. In general, larger datasets are better for training and evaluating a model. However, this is only true if the dataset size correlates with novelty, meaning it introduces unique and diverse cases. For example, a dataset containing applications from various time periods or regions ensures that the model can generalize to unseen scenarios. This has been shown by [Kap+20], where they demonstrated that the performance of Transformer Decoder Models correlates with the size of the data used to train them.

Figure 3.1 shows the label distribution for each dataset. Notably, the DexRay dataset has an unusually high percentage of malware APKs. On one hand, this could help the model learn more diverse representations of malicious APKs. On the other hand, Arp et al. refer to this as a sampling bias, where "the collected data does not adequately represent the true distribution of the underlying security problem" [Arp+22]. While the actual label distribution is unknown (and likely changes depending on the models use case), it seems unlikely that 39% of APKs are generally malware. Similarly, the Transcend dataset also has a high malware rate,

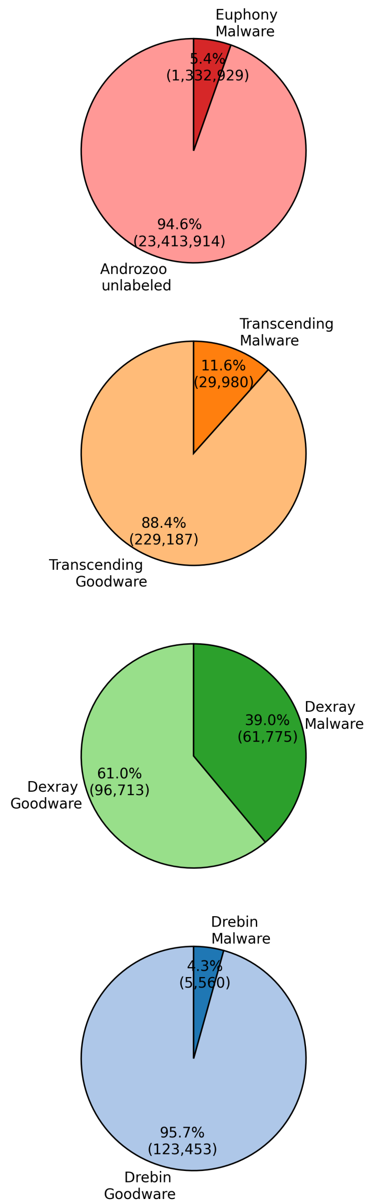

Figure 3.1: Distribution of malware and goodware samples across datasets shown as pie charts. The datasets analyzed are are ordered by size from largest to smallest. The number of APKs contained in the Dataset are shown in brackets

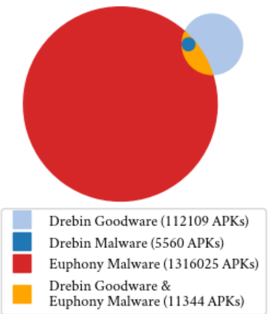

Figure 3.2: Overlap between the Drebin and Euphony datasets. The light blue circle represents Drebin, where 112,109 APKs are classified as goodware (excluding overlap). The red circle represents Euphony, which contains 1,316,025 APKs labeled as malware (excluding overlap). The dark blue circle represent those 5560 APKs, that are considered malware for both Drebin and Euphony. The intersection in orange highlights 11,344 APKs classified as goodware by Drebin but as malware by Euphony.

with over 10% of its samples being malicious. In comparison, Androzoo and Drebin show about 5% malware APKs in their distributions. The Androzoo Malware labels were derived from virustotal[1] by Euphony [Hur+17] in 2017, which means that they are outdated as newer APKs were not evaluated.

One important aspect is that there is no clear definition of malware, which significantly impacts the analysis. This lack of a universal definition leads to inconsistencies in classification across datasets, making it challenging to validate labels and draw reliable comparisons. Without a standardized framework, each dataset might apply its own criteria. Each dataset can have different interpretations of what is considered malware, often based on criteria such as the presence of specific permissions, API calls, or behaviors flagged by antivirus software. One dataset might classify an APK as malware due to aggressive adware practices, while another might only consider code executing malicious payloads. These differences make the validation of labels very difficult. One Example that shows this issue is evident when comparing Drebin with Euphony (figure: 3.2). Euphony and Drebin have a common subset, none of the sets is a true subset of the other. However, the APKs labeled as malware by Drebin are a true subset of Euphony. Additionally, Euphony classifies multiple APKs as malware that were classified as goodware by Drebin. The fact that there are no APKs that are classified as malware by Drebin and Goodware by Euphony shows that Drebin applies a more narrow definition on what is considered malware. Further Overlaps of different Datasets are visualized in the Appendix (figure: A.1).

3.1.1 Temporal Evaluation

The temporal evaluation of APKs is often based on metadata of the classes.dex component, as it contains crucial timestamp information reflecting the compilation date of the code. This metadata serves as a reliable indicator of the APKs development period, providing a foundation for chronological analyses. While this often works (figure: 3.3), it becomes more common that this meta information is not available. This lack of

[1] https://www.virustotal.com/gui/home/upload

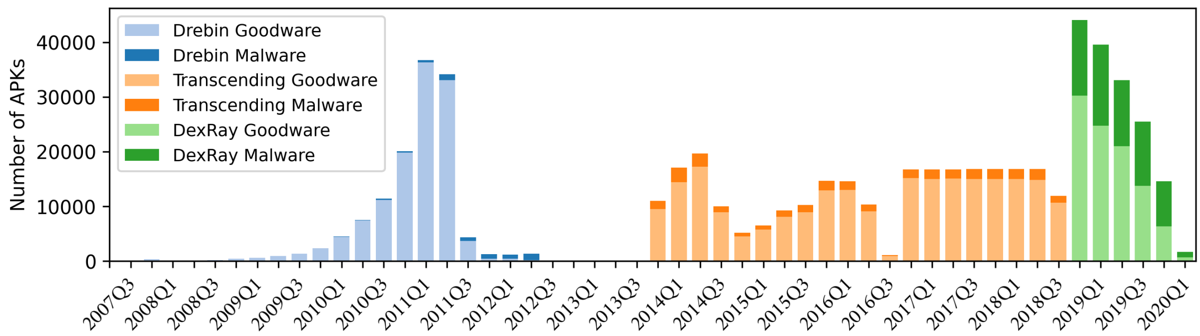

Figure 3.3: Temporal distribution of Android APKs across three datasets (Drebin, Transcend, and DexRay), categorized into goodware and malware. The Data is derived from metadata of the classes.dex file of each APK.

metadata poses challenges for temporal evaluation, as it limits the ability to analyze trends and shifts in malware development over time, as seen by the temporal evaluation of the androzoo repository (figure: 3.4).

The distribution of the .dex dates of the three datasets (figure: 3.3) shows that the datasets are in succession of one another. There is no temporal overlap, which explains why there is no APK present in more than one of these datasets (figure: A.1). There is a gap between the third quarters of 2012 and 2013 where neither dataset has samples.

While the Transcend Dataset seems to be evenly distributed, the other two datasets seem to show varying volumes over time. This holds true for the overall volume but also about the evenness of the label distribution. Here Drebin and DexRay show unbalanced malware sample behavior. This uneven distribution can affect the generalizability of findings.

One Example where concept drift is visible is, that the size of packaged APKs grows with time A.2. The three datasets show different distributions of APK sizes, where Drebin shows the smallest and DexRay the biggest APKs on average. The Plot also shows that the labels of all datasets are more evenly distributed by the APK size, compared to the .dex date of the APK.

When evaluating a model regarding the concept drift of malware, the Transcend dataset shows the most potential. However, only considering one dataset limits the generalizability of an approach, so the evaluation on multiple datasets is even better. One Problem is, that all three datasets only contain APKs where the .dex date is available. The temporal evaluation of Androzoo 3.4 showed that this is for most modern APKs not the case. It follows that each of the datasets are Biased in their selection of APKs and are therefore not fit for representing the True APK landscape.

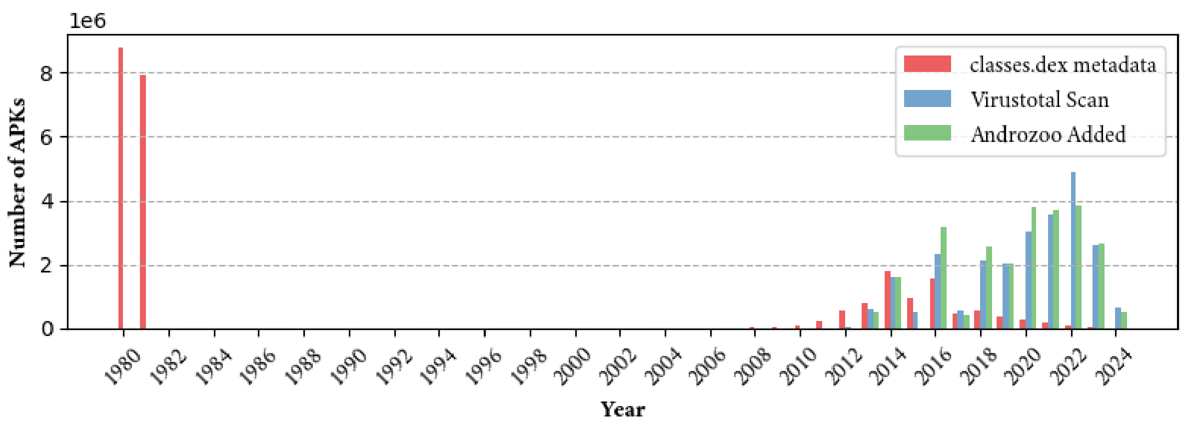

Figure 3.4: Temporal distribution of APKs based on three key attributes: classes.dex metadata, Virustotal Scan, and Androzoo Added. The red bars (classes.dex metadata) show a large spike in 1980 to 1982, likely due to incorrect or missing metadata values. The blue (year of first scan by virustotal on that APK) and green (year this APK was added to the androzoo repository) bars indicate a consistent increase in APK activity from 2010 onward, peaking around 2020 to 2022, reflecting the growing adoption of Android and corresponding malware collection efforts. The discrepancies between attributes highlight potential issues in dataset metadata accuracy and consistency.

3.1.2 Code Evaluation

When checking for Malware, an important representation of an APK is its code. As the code describes the logic of what the App does, it seems logical that, if the code is sufficiently evaluated, the harmfulness of the APK can be derived. Especially Transformer models show promise in interpreting the code of an APK as they work very well as LLMs.

It follows that not just time distributions of the datasets should be evaluated, but also the distribution of code. There are two major ways of decompiling an APK into a code representation. The classes.dex file can either be decompiled into Java code or into lower level Smali code.

For the decompilation of the APK into Java code, dex2jar[2] is first used to decompile the classes.dex file into a JAR file. The JAR file can then be further decompiled into plain Java code. The resulting Java code is on the one hand easy to interpret due to its higher level nature, however the decompiled code is prone to obfuscation that can make it hard to interpret. One example on how obfuscation can make the logic of the App more complex is by changing the package names, so that it is not possible to see the entry points that are defined in the Manifest file. This phenomenon is further evaluated in tables 4.6 and 4.7 in section 4.2.

There also is an option to decompile the classes.dex file into Smali code. Smali is the human readable representation of the Dalvik bytecode that is found in the raw classes.dex file. Smalie therefore is to Dalvik bytecode, what Assambly is to machine code. The Smali representation of an APK can be retrieved by using Apktool[3] on the classes.dex file. While the Smali representation is more accurate the original bytecode and also is more robust to obfuscation, the readability is much harder due to the lower level and niche nature.

[2] https://github.com/pxb1988/dex2jar
[3] https://apktool.org/

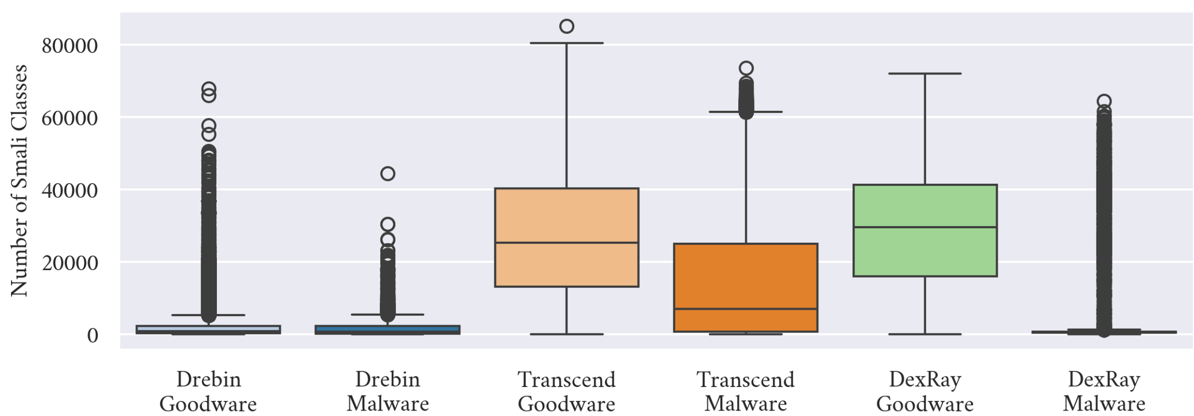

Figure 3.5: The boxplot shows the distribution of Smali classes in Android apps across the Drebin, Transcend, and DexRay datasets, split into Goodware and Malware. Drebin and Transcend have an similar Smali class distribution between Goodware and Malware. The DexRay Dataset shows a high inbalance in the number of Smali classes between the two labels.

As part of the code evaluation of this thesis, the same three datasets as in the time evaluation were considered. For each APK of the datasets, the classes.dex file was decompiled in both an Java and an Smali code representations. The code representation of an APK is quite complex and only hardly allows statistical aalysis. One approach that was possible is to count the classes of the code representations for each APK, reducing the complexity from a repository per app to a number. Figure 3.5 shows the Boxplots of the distribution of Smali classes of the apps in the dataset distinguishing between the label of the APK. The Boxplots show, that Drebin has a generally low amount of Smali classes per APK, wich can be explained by the Apps being older (as shown in table 3.3) and smaller (as shown in table A.2) than those of the other datasets.

A second observation, is that the DexRay Dataset is highly inbalanced. This is apparent both from the Boxplot (figure 3.5) and also by table 3.1, that quantifies the distribution of smali classes by computing the Q1, Median and Q3 values of each of the Datasets. When looking further into 3.1 the issue becomes even more apparent. Q1, Median and Q3 differ between the labels by a factor of 35-50, this makes any approach that makes label predictions based on Smali code of that dataset questionable (for example [Dao+24; Sun+23; Sun+24]).

The Transcend dataset shows less inbalance in code volume compared to the DexRay dataset. Especially on Q1 differs by a factor of 17,2 showing that there are relatively more code sparse Malware than Goodware APKs. When comparing the upper end of the spectrum, the inbalance becomes less drastic with an factor of 3,5 for the median and 1,6 for the Q3. Drebin shows overall a very good inbetween class balance of Smali classes when compared to the other datasets.

The decompilation into Java classes paints a similar picture. For comparison there is a similarly structured table showing the statistical properties of the Java class distribution 3.2. Additionally in the Appendix (A) there is another Boxplot A.3 that has the same structure as figure 3.5. When evaluating the volume of Java and Smali classes over all datasets a correlation coefficient of 0.92 proofs that they are very similar.

Table 3.2 shows that for the DexRay Dataset, more than 75% of the malware APKs have less than 12 Java classes, while 75% of the goodware labeled ones have more than 1000 Java classes. This makes it highly likely that an classifier will just differentiate between the number of classes present and derive the prediction from this information. This leads to the conclusion, that the DexRay dataset is generally insufficient for training a code based classifier.

Table 3.1: Smali Statistics Summary for DexRay, Transcend, and Drebin.

Label	Q1	Median	Q3
DexRay			
Goodware	16106	29546	41352
Malware	445	598	817
Transcend			
Goodware	13145	25364	40372
Malware	764	7057	25015
Drebin			
Goodware	331	901	2353
Malware	173	805	2316

Table 3.2: Java Statistics Summary for Drebin, Transcend, and DexRay.

Label	Q1	Median	Q3
Drebin			
Goodware	19	60	176
Malware	18	66	224
Transcend			
Goodware	854	1800	2899
Malware	52	738	2244
DexRay			
Goodware	1035	2007	3084
Malware	7	11	11

3.2 Baseline Creation

When generating a baseline for mobile malware detection, concept drift hast to be considered. Jordaney et al. [Jor+17] showed that there is a concept drift for android malware. In order to train a model that is

Table 3.3: Tree Stump (balanced class weights) results by dataset, feature, and split.

Dataset	Feature	Split	Accuracy	Precision	Recall	F1 Score	Threshold
Drebin	Java classes	random	10.16%	4.40%	99.07%	8.43%	2.5
	Java classes	time based	7.39%	4.31%	96.49%	8.24%	3.5
	Smali classes	random	76.01%	6.14%	33.24%	10.37%	247.5
	Smali classes	time based	7.15%	4.31%	96.76%	8.24%	33.5
	APK size	random	10.94%	4.32%	96.10%	8.27%	59676
	APK size	time based	6.62%	4.35%	98.38%	8.33%	59676
Transcend	Java classes	random	90.05%	70.58%	23.38%	35.12%	17.5
	Java classes	time based	82.63%	36.58%	68.35%	47.65%	967.5
	Smali classes	random	77.63%	27.44%	57.28%	37.10%	9699.5
	Smali classes	time based	77.61%	31.23%	77.85%	44.58%	15101.5
	APK size	random	18.75%	11.90%	94.56%	21.14%	57904190
	APK size	time based	39.27%	11.77%	65.44%	19.96%	11054373
DexRay	Java classes	random	92.05%	93.04%	86.37%	89.58%	90.5
	Java classes	time based	94.74%	89.67%	97.65%	93.49%	90.5
	Smali classes	random	92.10%	93.19%	86.36%	89.64%	1395.5
	Smali classes	time based	94.59%	89.35%	97.63%	93.31%	1395.5
	APK size	random	48.53%	42.97%	91.97%	58.58%	79432704
	APK size	time based	44.78%	39.11%	76.98%	51.87%	67843648

robust again this concept drift, Kan et al. [Kan+24] propose to split the data into train and test data in a particular way. The training data should be from before the test data, while keeping the same label ratio in between splits. The reason for this is to make sure that the model does not train on features of the data, that are prone to concept drift. The punishment for this is maximized if the train and test data is sequential. In order to account for and validate the concept drift in the domain of malware detection, the following baselines where both created on a dataset that is split randomly into train and test subsets and also split time based as proposed by [Jor+17]. The data is split into 80% test and 20% train data for both splitting Approaches. The evaluation is always done in a balanced manner, penalizing mistakes on the less frequent malware label more.

3.2.1 Non Transformer Approaches

A very simple classifier is a decision tree stump. This stump is infering a threshold based on the training data and then using it to predict a label for new samples. To make it even simpler for creating a first baseline, in Table 3.3 only one feature is considered when calculating the tree stump. The table shows how the Drebin dataset is most robust to this very basic

Table 3.4: Random Forest (n_estimators=300, max_depth=25) results by dataset and split. Features include Java classes, Smali classes, and APK Size.

Dataset	Split	Accuracy	Precision	Recall	F1 Score
Drebin	random	98.08%	91.69%	59.42%	72.11%
	time based	95.66%	48.69%	13.40%	21.02%
Transcend	random	94.53%	88.74%	60.19%	71.73%
	time based	90.07%	60.47%	40.94%	48.83%
DexRay	random	96.87%	97.75%	94.25%	95.97%
	time based	93.95%	87.34%	98.63%	92.64%

approach. For Drebin, neither of the considered features (Number of Java classes, Number of Smali classes & Size of the packaged APK) is fit for successfully separating the labels with one threshold. On the first glance it seems unlikely, that the thresholds for the APK sizes are constant over both different dataset splits. Since the threshold is changing for some other random splits (by using other random seeds), this phenomenon seems to be due to chance. This can be explained by how the threshold is derived from the training data using the gini coefficient [Gin12].

On the Transcend dataset, the Tree Stump approach achieve better results. Here the concept drift can be observed by the changing thresholds between the split methods.

The most shocking result is, that the DexRay dataset can be solved with an F1 score of 93,49% with a simple Tree stump approach on the Java classes feature. This shows how inbalanced the dataset is and how it is unfit for evaluatig models that rely on code based features. Another interesting observation is that for both the Java and the Smali classes the Thresholds stay consistent inbetween different splits of the data. Also the performance increases when the data is split based on time rather than randomly, which means that the newest APKs are most prone to this label inbalance.

Overall the table 3.3 shows how different the datasets are, even though all of them are build for the same objective of evaluating android malware detection models. The same model achieves an F1 score of 8% on Drebin, 48% on Transcend and 93% on DexRay.

Applying a more sophisticated model on all three of the features, increses the performance further. This can be shown by evaluating a decision tree (table A.1) or a generally higher performing random forest (table 3.4) model. Table 3.4 shows how a random forest algorithm achieves around 70% F1 score on Drebin and Transcend when trained and tested with the randomly split data. The concept drift of both Drebin and Transcend becomes apperent by lower performance on the time based split data. Here Drebin shows an even steeper decline compared to Transcend showing more concept drift in the data. When evaluating the random forest on DexRay, both the random and time based splits show a very high F1 score of 92%-96%. it is noteworthy that a tree stump on either Java or Smali classes (table 3.3) performs better than the random forest on the

Table 3.5: Decision Tree (max_depth=15) results by dataset and split. Features include Permissions.

Dataset	Split	Accuracy	Precision	Recall	F1 Score
Drebin	random	95.74%	49.77%	90.17%	64.14%
	time based	87.60%	24.83%	92.54%	39.15%
Transcend	random	78.03%	32.79%	85.77%	47.44%
	time based	72.26%	26.17%	76.91%	39.05%
DexRay	random	90.25%	87.34%	88.07%	87.70%
	time based	83.66%	71.80%	95.01%	81.79%

time based split data. This shows that the relatively simple random forest model already overfits the time based training data.

Another possible representation of android APKs is by the permissions listed in the manifest file. This representation too can be used to classify the APK as malicious. Table 3.5 shows however that a decision tree trained on only these features performs worse than the random forest on only the three previous features. This shows how valuable code based features are for malware detection.

When combining the permissions, the code volume features and the APK size feature, a decision tree outperforms the random forest that is trained without considering permissions (see table A.2). By then also increasing the model complexity through the depth hyperparameter from 15 to 30, the highest non transformer baseline of this work is achieved (see table 3.6). The table shows once more how the time based split leads to worse performance for all three datasets, so that concept drift is apparent and generalizability of the approach is questionable. Other than being prone to concept drift, the results show that the performance on Drebin and Transcend is not satisfactionary.

On a general level the very basic models that have been evaluated on the three datasets showed that each dataset exhibits some form of concept drift, shown by decreasing performance when splitting the dataset based on time. Also the DexRay dataset has such a big label inbalance on the code volume features that a simple tree stump nearly solves the dataset, this renders the dataset not sufficient for evaluating code based models.

Table 3.6: Decision Tree (max_depth=30) results by dataset and split. Features include Java classes, Smali classes, APK Size, and Permissions.

Dataset	Split	Accuracy	Precision	Recall	F1 Score
Drebin	random	98.58%	81.41%	86.04%	83.66%
	time based	94.89%	45.17%	86.96%	59.45%
Transcend	random	95.03%	76.65%	82.03%	79.25%
	time based	92.21%	65.16%	69.95%	67.47%
DexRay	random	96.95%	96.36%	95.89%	96.13%
	time based	94.88%	89.14%	98.78%	93.72%

3.2.2 DetectBERT

In the previous chapters, various non Transformer based approaches for static Android malware detection were explored. These methods rely on traditional machine learning techniques and therefore struggle with capturing the complex relationships within Android application data. In contrast, Transformer based models have demonstrated SOTA performance in various NLP and code understanding tasks [Wol+20]. This chapter introduces DetectBERT [Sun+24], the first Transformer-based approach examined in this thesis.

DetectBERT is a static Android malware detection model that utilizes Transformer based architectures for analyzing APK files. Its processing schema is visualized in figure 3.6. Specifically, DetectBERT processes the classes.dex file contained within Android APKs by extracting Smali class representations. These representations are subsequently transformed into compact feature embeddings using DexBERT [Sun+23], a specialized Transformer model developed by the same authors for this task. These embeddings serve as inputs to the DetectBERT component, a Transformer architecture that employs Nyström based attention [Xio+21] to effectively manage computational complexity while capturing long-range dependencies. The embeddings, together with a special CLS token, designed to aggregate global context, are refined through successive layer normalization and Nyström attention layers. This processing aims for, global feature representation encapsulated in the embedding of the CLS token. Finally, DetectBERT uses a fully connected layer to process the Embedded CLS token into a malware prediction, indicating whether the APK is malicious or benign.

To evaluate DetectBERT, we benchmarked its performance on the DexRay ad Transcend datasets. Due to computational and time constraints, the full DexRay and Transcend datasets were not used. Instead, subsets of both datasets were generated, each containing 20,000 APKs with an even label distribution. Additionally, the Smali class distributions were balanced between malware and goodware samples, countering the inherent imbalance of the DexRay dataset. The DexRay dataset is also the dataset

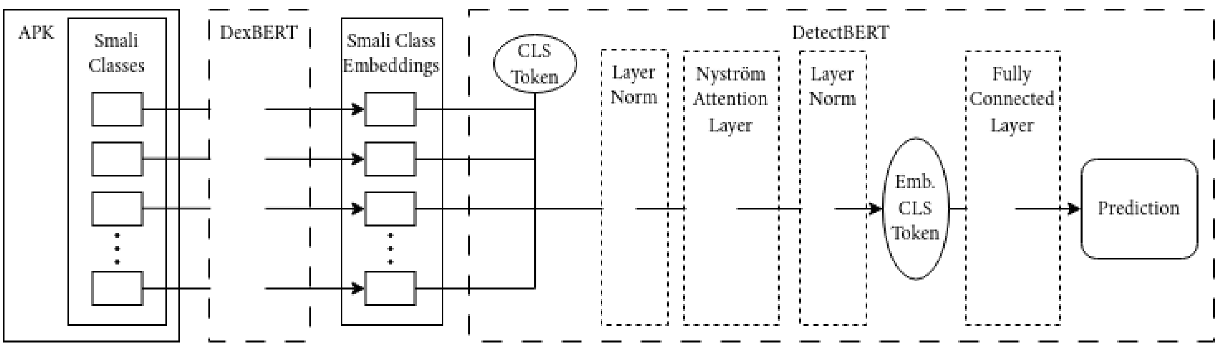

Figure 3.6: Overview of the DetectBERT architecture. The model processes APK files by extracting small class representations using DexBERT, which generates embeddings for each class. A CLS token is added to the sequence, followed by layer normalization and a Nyström attention layer. Additional layer normalization is applied before feeding the CLS token embedding into a fully connected layer to generate the final prediction.

Table 3.8: DetectBERT Benchmark Results Across Datasets and Splits. The results are reported for both random and time-based splits, highlighting the effectiveness of the model in distinguishing between goodware and malware. On the DexRay dataset, DetectBERT consistently achieves higher accuracy, particularly for goodware classification.

Dataset	Split	Accuracy	Precision	Recall	F1
DexRay (Subset)	Random	89.57%	89.57%	89.57%	89.57%
	Time-based	89.17%	90.00%	89.17%	89.10%
Transcend (Subset)	Random	81.77%	82.46%	81.77%	81.77%
	Time-based	78.67%	78.73%	78.67%	78.68%

that was used for benchmarking the model in its original paper. However, due to the Smali class imbalance between labels, the benchmarks presented in the paper were not entirely trustworthy. The results on the balanced subset do indicate, that the model does not simply rely on the number of Smali classes for predictions but at least to some extent, utilizes other information extracted from the Smali class embeddings. However, the results on the subsets are not as good as those provided in the original paper (see Table 3.7). This performance discrepancy might also be attributed to the smaller training dataset size used in our experiments.

Table 3.7: DetectBERT Performance Results (accuracy, precision, recall, F1 Score) from the original paper [Sun+24]

Acc.	Prec.	Rec.	F1
97%	98%	95%	97%

The model was tested under two different data splitting strategies: a random split, which ensures a balanced distribution of samples in training and testing, and a time based split, which simulates realworld conditions by training on older data and testing on newer samples. Table 3.8 presents the detection performance of DetectBERT across both datasets and splits. The results indicate that DetectBERT is achieving high accuracy and F1 scores, particularly on the DexRay dataset. The model demonstrates a slight decline in performance for time-based splits, likely due to evolving malware patterns. The performance on the subset of the Transend dataset are not as good as on the DexRay dataset.

The evaluation gives several key insights about DetectBERTs performance. Transformer based models effectively capture complex relationships in APK data, surpassing traditional methods for the Transcend dataset (see table 3.6). The DexRay dataset yields better results compared to Transcend, suggesting dataset specific challenges. The time based split introduces a more realistic evaluation setting, exposing the impact of concept drift in malware evolution.

Overall, DetectBERT is implemented as a TransMIL model , which is a Transformer based multiple instance learning (MIL) framework [Sha+21]. In this approach, the model processes multiple Smali class embeddings as instances within an APK and learns to make a classification decision based on the most relevant instances. This allows DetectBERT to focus on the most informative Smali class embeddings rather than relying on simple aggregate statistics, improving its ability to detect malware patterns effectively.

3.3 Experimntal Setup

3.3.1 General BERT based approach

The experiments conducted in this thesis build upon a generalized paradigm for malware detection, as visualized in figure 3.7. The core idea is that, in order to classify an APK as malicious or benign, it must first be processed through a structured pipeline.

The first step [1] involves extracting an appropriate representation of the APK. This representation serves as the input for the subsequent stages and can be derived from different static attributes of the APK, ensuring that relevant structural and behavioral information is preserved.

Next, in step [2], this representation undergoes an embedding process. The embedding is designed to transform the extracted representation into a format suitable for processing further while maintaining essential information and reducing dimensionality. The choice of embedding strategy significantly influences the quality of the learned representation and consequently, the detection performance.

In the final step [3], the embedded representation is passed to a decision head that generates the final classification. The decision head can take various forms, from simple fully connected layers to more complex Transformer based network structures. The effectiveness of the decision head plays a crucial role in ensuring that the model correctly distinguishes between benign and malicious applications.

The objective of this thesis is to analyze the capabilities of this architecture by systematically evaluating each of these three core components. By designing experiments that focus on variations in APK representation, embedding methods, and decision heads, the study aims to provide insights into the strengths and limitations of Transformer based malware detection models. The following chapters will further explore each of these stages in detail, discussing alternative design choices and their impact on detection performance.

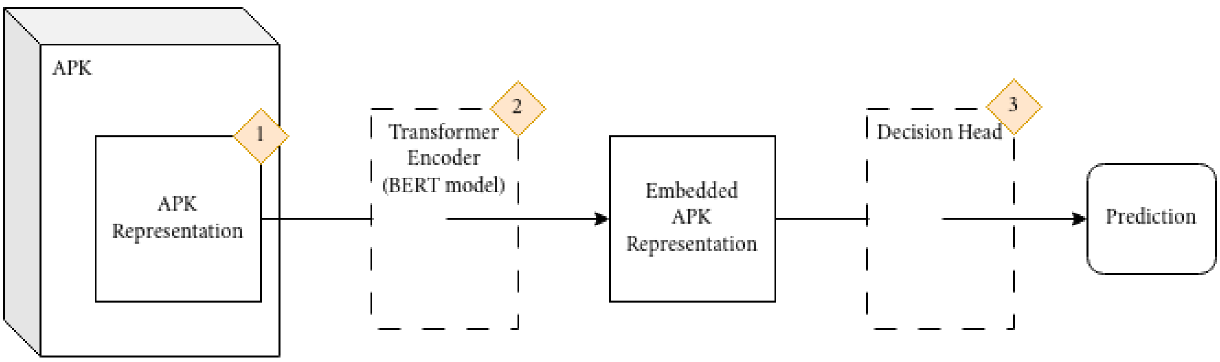

Figure 3.7: Generalized BERT-based approach for malware detection. This figure illustrates a blueprint for utilizing BERT models in APK-based malware detection. The process consists of three main stages: extracting an APK representation [1], encoding it using a Transformer based model (BERT) [2], and applying a decision head to generate the final prediction [3]. The numbered elements indicate open design choices within this blueprint.

4 Evaluation

4.1 The APK Representation

A key aspect of designing an embedding based mobile malware detection system is how the APK is represented. Since most embedding models, including all considered BERT models, cannot process APKs as a whole, specific features such as permissions, API calls, or manifest attributes must be selected and represented in a tokenizable format. These features should be structured consistently to ensure comparability across different APKs.

To structure these features, concatenating them with designated tokenizer vocabulary entries as separators is beneficial. In this work, a varying number of # characters were used as separators, with different separators applied to distinguish between various types of separations (e.g., separating feature categories versus separating elements within a feature list).

To show how varying APK representations change the performance of a malware detection architecture, an experiment was done with a relatively simple stack 4.1. To best see how the representation has an effect, the other two variables of the stack were kept static. For simplicity and efficiency, the selected embedder was modernBERT [War+24]. Since modernBERT has a context window of up to 8000 tokens, it was possible to process each APK in one iteration, ensuring that the entire representation was captured in a single pass. This approach improves efficiency by reducing computational overhead and helps preserve relationships within the APKs extracted features. The embedding size was set to 768, as recommended by the original modernBERT authors. After applying the Embedding, each input token along a blank CLS token is represented by 768 floating points. After each APK was embedded, a simple fully connected layer with dropout was trained for 5 epochs on the timebased split subset of the dataset, as described in the previous chapter. To simplify processing, the decision head was not trained on the full embedding but rather only on the CLS token embedding. Each APK was therefore represented by 768 floating-point values, ensuring a compact yet informative representation for classification. During training, only the weights of the last fully connected layer were updated, while the weights of the modernBERT embedder were frozen. The fully connected layer was chosen for its simplicity and effectiveness in classification tasks, while the dropout mechanism was included to help prevent overfitting and improve generalization. With this stack, multiple different APK representations could

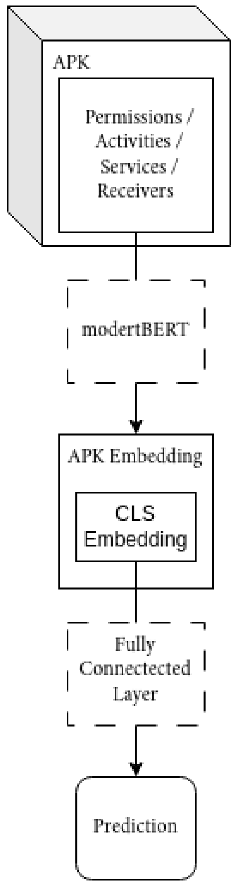

Figure 4.1: Distribution of malware and goodware samples across datasets shown as pie charts. The datasets analyzed are are ordered by size from largest to smallest. The number of APKs contained in the Dataset are shown in brackets

be chosen. However, the representation was limited to 8000 tokens so that it could be processed by modernBERT in a single pass. The representations that were chosen were all extracted from the manifest.xml file, as it contains important metadata about an application's permissions, activities, services, and broadcast receivers, which are often indicative of its behavior and potential security risks. The selected ones were the names of the given permissions of the APK, the names of the marked activities, and the names of the marked services and receivers. Permissions indicate what resources an application can access, while activities, services, and receivers help determine its execution behavior.

The performance results in Table 4.1 illustrate how the represention of an APK impacts malware detection. The combination of activities, permissions, receivers, and services (A + P + R + S) achieved an F1 score of 73.33%, indicating that a comprehensive manifest representation is beneficial. However, removing receivers (A + P + S) slightly improved performance to 73.64%, suggesting that including receivers introduces noise that can degrade overall performance.

A subset with only activities and permissions (A + P) produced the highest F1 score of 74.49%, confirming that these two features capture essential malicious behaviors effectively. Permissions alone (P) showed high recall (77.01%) but not so high precision (63.48%), reflecting the broad use of permissions in both benign and malicious applications. Activities (A) offered a balanced performance with an F1 score of 70.24%, indicating importance in malware detection.

Receivers (R) significantly worsened performance when included, resulting in an F1 score of 0.00% when used alone, and lowering overall performance when combined with other features. Services (S) also struggled with an F1 score of 60.55%, indicating that these background components, while relevant, lack strong discriminative power in isolation. Permissions and activities show the most potential as features for static malware detection, with the removal of receivers improving overall system robustness.

Table 4.1: Performance of different APK representations using frozen modernBERT embeddings. Features are extracted from Android manifest.xml (A=Activities, P=Permissions, R=Receivers, S=Services). Results reported on time-based transcend subset split after five epochs.

Representation	Accuracy	Precision	Recall	F1
A + P + R + S	74.45%	76.30%	70.58%	73.33%
A + P + S	74.67%	76.39%	71.09%	73.64%
A + P	74.47%	74.10%	74.89%	74.49%
R + S	65.12%	74.12%	45.97%	56.75%
A	70.64%	70.88%	69.62%	70.24%
P	66.51%	63.48%	77.01%	69.60%
S	65.12%	69.28%	53.77%	60.55%
R	50.23%	0.00%	0.00%	0.00%

Comparing the results of this experiment with just permissions as Feature representation with the previous results of doing malware detection with a decision tree based on solely the permissions (table: 3.5) shows how promising this architecture is, as it shows much better results. This might be partly due to the analysis of custom permission naming.

4.2 The Transformer Encoder

The second variable of the analyzed Architecture for APK classification is the Transformer Encoder (Figure 3.7). Possible Transformer Encoders to use in this approach include: BigBird [Zah+21], Longformer [BPC20], Performer [Cho+22], Linformer [Wan+20], Reformer [KKL20], FNet [Lee+22], Nyströformer [Xio+21] or DexBert [Sun+23]. Before any of the listed encoders can encode any language based representation of the APK, the input first have to be tokenized. For the tokenization process a vocabulary is needed that maps different tokens to an integer value. The vocabulary is learned during the training of an encoder model and is therefore model specific. In this work it improved the efficiency of the pipelines (e.g. Figure 4.1) to first tokenize all of the APK representations before encoding them, in order not to create queues of mixed GPU & CPU usages.

Once tokenized, the general Idea for the Encoder is to reduce the volume of the chosen APK Representation to a more manageble Level while still maintaining all the relevant information that is needed for a classification. Therefore, models with a large context window are especially suitable. ModernBERT [War+24] with its context window of up to 8000 tokens makes it an especially suitable candidate. The output of the Encoder is an embedding of the APK representation. The volume of the embedding is a hyperparameter of the Encoder. For modernBERT the default size of the embedding is 768.

Table 4.2: Performance of different APK representations using *unfrozen* modernBERT embeddings. Features are extracted from the Android manifest.xml (A=Activities, P=Permissions, R=Receivers, S=Services). The encoder was also trained through backpropagation in this experiment.

Representation	Accuracy	Precision	Recall	F1
A + P + R + S	83.95%	88.67%	77.67%	82.81%
A + P + S	84.32%	85.14%	82.99%	84.05%
A + P	85.66%	87.88%	82.58%	85.15%
R + S	71.04%	87.89%	48.51%	62.51%
A	80.29%	90.22%	67.75%	77.39%
P	74.97%	72.20%	80.86%	76.28%
S	72.23%	88.46%	50.84%	64.57%
R	50.23%	0.00%	0.00%	0.00%

The first experiment of this thesis that focuses on the Encoder with regards to overall model performance bases on the same stack as the previous experiment (Figure 4.1). While the setup stayed mostly the same the only change was to unfreeze the encoder models parameters during training. This led modernBERT to also improve its embeddings during training. The results of this is shown in table 4.2. When comparing these results with the results of the previous experiment, where the same encoder model was frozen 4.1, it becomes clear that the performance of the classification increases in nearly every instance. For the receivers as representation the resulting F1 score stayed 0% which shows that without a proper representation a better encoder can not outbalance this. Other than this instance the performance increased for all representations for up to more than 10% points of the F1 score, with stronger increases for more complex and combined representations. The F1 score of 85.15% for the activities and representations representation shows how well this very simple and also efficient approach works.

Table 4.3: Performance of Activities (A) and Permissions (P) as representations using the full transcending dataset and otherwise the same setup as for table 4.2

Acc.	Prec.	Rec.	F1
91.36%	91.02%	91.36%	91.17%

The results demonstrate even stronger performance when using the full dataset. Both A+P and A+P+S representations achieved high accuracy, precision, recall, and F1 scores, with A+P scoring 91.36% in accuracy and A+P+S slightly outperforming it at 91.75%. Notably, the performance gap observed in previous experiments between the two representations has narrowed significantly. The performance even increased when the services where processed as well, indicating that additional features can enhance the models capability when trained on larger datasets, contrary to the trend observed with smaller subsets.

Table 4.4: Performance of Activities (A), Permissions (P) and Services (S) as representations using the full transcending dataset and otherwise the same setup as for table 4.2

Acc.	Prec.	Rec.	F1
91.75%	91.53%	91.75%	91.63%

When comparing these results with those from Table 4.2, it is evident that the full dataset allows for improved overall performance. The A+P representation, which previously outperformed others with an F1 score of 85.15%, now achieves an even higher F1 score of 91.17%. Similarly, the A+P+S representation, which scored 84.05% in the previous experiment, now leads with an F1 score of 91.63%.

This highlights the significance of dataset size in training transformer based models for Android malware detection. The improved scores suggest that the architecture benefits from a more extensive dataset, enabling it to learn a broader variety of malware and goodware patterns.

An additional insight from these results is, that increasing the volume of APK representation does not inherently degrade performance when the added information is dense and meaningful. The inclusion of services (S) in the representation contributes positively to the classification task, as demonstrated by the improved metrics in the full dataset experiment.

Another Experiment that was conducted in order to evaluate the Transformer Encoder in the malware detection architecture is to switch the base model of the encoder. For this, APKs were represented by its permissions and activities and the same transcendence subset was used as in table 4.1 and 4.2. In the experiment, the encoder model was trained along the decision head. The models that were considered were modernBERT [War+24], BigBird [Zah+21] and Longformer [BPC20].

Table 4.5: Performance comparison of different encoder models for generating embeddings. The APK representation is fixed to Activities (A) and Permissions (P). The encoder was trained through backpropagation in this experiment. The transcenden subset was used as dataset. The embedding of the CLS token was used as APK embedding.

Model	Accuracy	Precision	Recall	F1
modernBERT	85.66%	87.88%	82.58%	85.15%
BigBird	50.23%	25.23%	50.23%	33.59%
Longformer	49.77%	24.77%	49.77%	33.08%

When using the same approach as before, where the embedding of the CLS token is used as the input for the decision head, the performance of the models is as shown in table 4.5. The results for both BigBird and Longformer are quite bad. The combination of Accuracy, Precision and Recall leads to the conclusion that with these two models the overall architecture just learns the distribution of labels, as this combinateion of evaluation metrics can only be observed if roughly FP = FN = TP = TN = 1000 for the evaluation subset of size 4000.

The results for BigBird and Longformer might be that bad, since they use sparse attention rather than fully contextualizing the CLS token as modernBERT does. To verify this, another experiment was done were instead of using the embedding of the CLS token as input for the decision head, the average of all embedded tokens was computed along each of their features. This resulted in better results for both BigBird and Longformer while decreasing the performance of modernBERT as seen by table A.3.

During execution it became obvious that the runtime of BigBird and Longformer were around 12x slower than modernBERT which again make modernBERT look superior. However, it is to be found out if the same models are superior for all kinds of APK representations. For example DexBERT is trained specifically on Smali code and is therefore likely to perform better on this as APK representation.

Also in addition to the effectiveness also the size and applicability of the vocabulary has to be considered. As the vocabulary is the dictionary that is used when tokenizing the input (the APK representation), the vocabulary together with the context window of the model define how much volume the input is allowed to have. ModernBERT for example has an context window of 8192 while having a vocabulary size of 50368, in comparison: BigBird has an context window of 4096 and a vocabulary size of 50358 and Longformer has an context window of 4096 and a vocabulary size of 30522. This as well makes modernBERT look superior to the alternative models. It is to be considered through how well the vocabulary fits the inut data, as the efficiency is worse for words that have to be broken down into multiple smaler tokens instead of being represented by just one.

Overall modernBERT seems to be superior to both BigBird and Longformer when used as Encoder in the malware detection pipeline.

4.3 The Decision Head

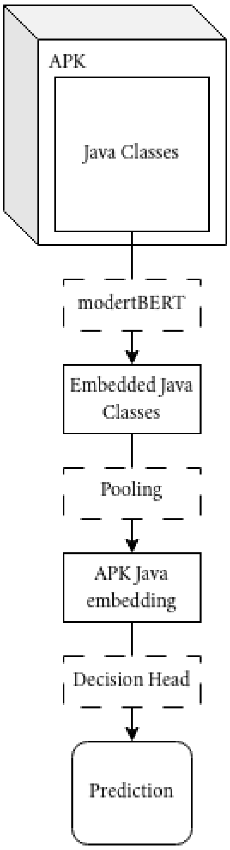

Figure 4.2: Distribution of malware and goodware samples across datasets shown as pie charts. The datasets analyzed are ordered by size from largest to smallest. The number of APKs contained in the Dataset are shown in brackets

The last of the three variables of the proposed architecture 3.7 is the decision head. In the experiments so far the decision head was a simple fully connected layer, that was trained to predict the label from the embedding of the APK representation. This has proven to be quite effective, however the applicability of this particular decision head is limited. Whenever the tokenized APK representation exceeds the context window of the embedding model, the general architecture becomes more complicated. If the representation cannot be summarized before the embedding, the only solution that still bases on a transformer embedding is to break down the representation into smaller chunks and to embed these chunks iteratively. This then leads to one APK not being represented by a single embedding but rather by multiple ones. Especially if the number of chunks the representation has to be broken down into, varies in between APKs, it can be difficult to train a common decision head as the input of it is of high and potentially varying size.

This issue can be overcome by pooling the embeddings into a smaller and fixed sized state. If the representation is of varying size the number of potential pooling mechanisms decreases, but there are still some possibilities. One really simple pooling would be to compute the min, max and/or mean of the embeddings. As the embeddings all have the same size, these statistical measures can just be computed over each feature of it. Another approach would be to select one of the representations by random.

In order to evaluate what kinds of decision heads perform better or worse on the task of embedding classification another experiment was conducted. While it still shows the general structure of the proposed architecture 3.7, it differs quite a bit from the previous setup shown in Figure 4.1. Figure 4.2 shows how this new experiment is set up. As the representation of each APK, the java code that can be decompiled from the classes.dex file is used. This java code is then structured by the corresponding classes. This representation is quite extensive as it consists of 1866 classes on average on the full transcending dataset A.3. This brings the base architecture 4.1 to the following problem: In most cases the APK representation exceeds the context window of the modernBERT model that is used to calculate the embeddings. As discussed before, the Representation has therefore to be broken down into chunks. For Chunks the java classes were used. While this is an easy way to break down the code, even single java classes sometimes exceed the 8000 tokens that defines the modernBERT context window. If this occurred during the experiment, the java class was processed in multiple iterations via a sliding window approach with a stride of 6000. Then only for those cases, the average of each CLS token, that was generated with the sliding window, was used for the APK representation. Like this one CLS token for each java class was computed.

Table 4.6: Number of total java classes and classes that are mentioned by the manifest.xml file of the Transcending dataset. Also the number of classes that match by the classname is given

Feature	Mean	Sum
Java Classes	1866	483102763
Manifest Classes	20	5277764
Matching Classes	14	3791740

One approach, that was tried to reduce the amount of java classes that had to be embedded, was to only process those java classes that are either declared as activities, permissions, services or receivers in the Manifest.xml file. This approach was not feasible due to some obfuscation technique that changed the naming of the java classes in the manifest file. Deeper analysis showed that only 72% (see table 4.6) of java classes in the transcending dataset have a constant naming between classes.dex and manifest.xml file. For the DexRay dataset it even were only 12% (see table 4.7). So all the java classes of the APK were considered.

Table 4.7: Number of total java classes and classes that are mentioned by the manifest.xml file of the DexRay dataset. Also the number of classes that match by the classname is given

Feature	Mean	Sum
Java Classes	1394	220838917
Manifest Classes	58	9247905
Matching Classes	7	1126716

As stated before this led to a variable size embedding that has to be classified. The experiment that is conducted to evaluate the decision heads is based on the approach of [Sun+24]. Therefore, the detectBERT model is used as Decision head as well as three other simple approaches and their performance is compared. The other approaches are: Average - here the mean of all the Java Class based CLS token embeddings is computed and then processed by a fully connected layer. The mean is computed for each of the 768 features the Embedding has, so that the resulting mean embedding has 768 features as well; Addition - the same as with Average but instead of the mean, the sum is used to derive a single embedding that represents the APK; Random - here one java class is selected randomly and its CLS Embedding is used to represent the APK for the classification task.

Table 4.8 shows the results of the conducted experiment. The performance is evaluated after training the decision head for 20 epochs on the same transcendence subset that is used in previous experiments. This subset is split into train and test sets sequentially to account for concept drift.

The results show how the average aggregation of CLS tokens performs the best when using java code as APK representation and modenBERT as Embedding model. It is interesting that even though it is much simpler, it does perform better than DetectBERT which performs slightly worse even though it uses multiple attention layers to aggregate the CLS tokens. Another interesting finding is that averaging the embeddings of each Java class outperforms the aggregation approach. Overall the ranking of the four decision heads that are used is the same as the performances on the smali embeddings that were reported by [Sun+24], except for the performances of detectBERT and the Average aggregation. In the Experiment that was done on Java classes the performance of the average aggregation exceeded the one of detectBERT, showing once again that simplicity can often lead to better results.

Table 4.8: Performance comparison of different decision heads on the Transcend dataset with a time-based split.

Decision Head	Accuracy	Precision	Recall	F1
Average	83.80%	83.87%	83.80%	83.79%
DetectBERT	82.15%	82.31%	82.15%	82.13%
Addition	80.23%	81.57%	80.23%	80.01%
Random	66.80%	66.81%	66.80%	66.79%

5 Discussion

5.1 Other Transformer based architectures

In the Evaluation Part of this thesis exclusively encoder only Transformers were evaluated for mobile malware detection. While the Encoder is the first and most obvious choice, there are other Transformer architectures (see section 2.1 that have potential in advancing the field. Those other approaches are discussed in this chapter.

5.1.1 Decoder only Transformer based approaches

Decoder only Transformers operate autoregressively, predicting the next token in a sequence, one by one. When it comes to the task of malware detection different applications for decoder only models come to mind.

First, a decoder only model can serve as an anomaly detector: if a sequence of instructions or API calls from a code class has a low probability under the model, it may indicate malicious or obfuscated behavior not seen in benign software [BH20; MHZ24]. This approach uses the language models learned distribution to flag outliers.

Second, decoder only models can be integrated into multi step analysis frameworks. Here one idea setup where one Transformer agent generates malicious behavior (or code variants) and another agent (e.g., a detector model) evaluates them. Such a generator detector pairing can also work as a form of adversarial training [Zha+24]. If such a Framework could be designed without a human in the loop, a reinforcement learning setup could be designed. As reinforcement setups have proven very effective in the past [Dee+25; Sil+16], this approach might as well show strong results. However, we did try to set up a multiagent system as part of this thesis that failed to work. The setup was based on the crewai [1] Framework with the 8B and 70B Llama models [AIM24] as base LLM. The proof of concept for this approach was to have one agent analyze the manifest.xml file and ask another agent questions about the java classes that were defined as activities, services, receivers and providers. The agents were given a prompt template (see appendix figure C.1), a system template (see appendix figure C.2) and tools that they can use. Unfortunately the approach did not work as the agents were not able to consistently use their tools or communicate with other agents and in addition they tend to jump to conclusions once a problem came up as shown in the appendix

[1] https://www.crewai.com/

figure C.3. The conclusion here seems to be that while such a multiagent architecture seems promising in the field of mobile malware detection, the base LLMs yet have to improve in order to reliably work in such a setting.

A third Idea would be to use a model that was trained as a transformer decoder but use it to encode a sequence. This could be done by using the normally hidden token embeddings at each step, before generating the actual output. The hidden states of the decoder at each step then serve as an encoding of the sequence. The last hidden state of the decoder could be used as a dense representation of the input sequence. While in theory, Transformer encoders (like BERT) are typically better suited for encoding because they process the entire sequence in a Bidirectional manner, this is yet to be proven.

5.1.2 Encoder-Decoder Transformer based approaches

Encoder-decoder Transformers process input sequences with an encoder and generate output sequences with a decoder. In malware detection, this architecture is helpful for tasks like deobfuscation and feature construction. One notable example is using a Transformer to reverse code obfuscation: Dedek and Scherer [DS23] trained a character level transformer that takes obfuscated PowerShell script as input and outputs the original deobfuscated script. This approach reports a 92% full deobfuscation success rate, demonstrating how an encoder-decoder can recover malware code that was hidden by attackers obfuscation techniques. Traditional static approaches struggle with heavily obfuscated code where each variant looks unique, but an encoder-decoder model could learn the transformation pattern, effectively acting as an automated deobfuscator. Despite their strengths, encoder-decoder Transformers can be resource intensive and require appropriate training data. In static malware detection, obtaining pairs of obfuscated and original code for supervised training may be difficult unless obfuscation techniques are simulated. This in turn might limit the deobfuscator to learn only know techniques making it prone to decreased performance due to concept drift.

Additionally, enoder-decoder transformers can be used for feature engineering and representation learning. A big part of mobile malware detection is to properly represent an APK by features that behold the relevant information for this task (see chapter 4.1). For instance, the model can produce a longer, expanded form of code from a compressed or packed input, or vice versa, effectively handling various forms of coding and code compression. This gives encoder-decoder Transformers a degree of flexibility with sequence lengths: the encoder creates a compact representation of a long input sequence, and the decoder can iteratively construct or analyze it piece by piece, mitigating some challenges of long inputs. Long code can be effectively segmented or summarized using transformer autoencoders [Ahm+20], and low level code can be reconstructed in a higher level and vice versa [TZR24].

Some authors even believe that transformer autoencoders can be used for the classification task directly by evaluating the reconstruction error [Als+24]. The authors argued that it would make them effective in identifying novel or zero-day malware, as they can recognize atypical patterns without requiring huge labeled datasets. Nonetheless, they may have difficulties with advanced malware that closely resembles benign activity.

5.1.3 Graph Transformer based approaches

Another approach that can leverage either of the different Transformer architectures is based on the idea that APKs can be represented as graphs rather than by language. Examples for these Graphs are control flow graphs of functions [Gon+23], call graphs between functions [SS23], or abstract syntax trees [Ahm+20]). Graph based Transformers apply the attention mechanisms to graph nodes and edges rather than linear sequences of tokens. For example, Moon et al. [MBC21] proposed a Directional Graph Transformer to embed control flow graphs for malware classification. In this model, each node in a functions control flow graph is treated similarly to a token, and attention is computed along the graphs adjacency structure, capturing relationships like loops. Bu and Cho [BC23] took this further with a triplet trained graph Transformer, using a few shot learning approach with contrastive triplet loss to distinguish malware families via their graph embeddings. Additionally, the previous in chapter 2.2.1 mentioned approach of the "Heterogeneous Temporal Graph Transformer" is a prime example how a graph transformer is used at a large company as Tencent [2] in production to detect new malware [Fan+21]. The graph Transformer processes structural information that sequence models might miss. These models first convert code to graphs, then apply a transformer model on the graph. The multi head attention is computed over graph neighborhoods, showing how strongly one code block is related to another in the context of malware behavior [Als+24]. This has shown promise in evaluating relationships within code and to detect which functions are critical to malicious behavior.

5.2 Interpretation

As part of this Thesis, two main experiments were evaluated. The first experiment (shown in figure 4.1) represented APKs by their meta information stored in the manifest.xml file, embedded this representation in one iteration, and then evaluated the embedding vector of the CLS token via a fully connected layer. The experiment demonstrated the general effectiveness of the architecture (shown in figure 3.7) by showing that when processing only the permissions of an APK, it obtained much better results—up to 76% 4.2, compared to only 39% when training a decision tree on the same permissions 3.5. This drastic increase could be explained by the fact that transformer based architectures can interpret custom

[2] https://www.tencent.com/en-us/

permissions more effectively than a decision tree due to their capability to recognize semantic and contextual relationships, rather than just feature counting or simple rule-based splits. Another contributing factor might be that transformer based models are inherently better at learning complex relationships of permissions, as opposed to a decision tree that was constrained by its depth of 15, limiting its complexity.

Another result of the first experiment is that when using the Activities, Permissions and Services as APK representation on the time based split Transcend subset, close to 85% F1 score could be achieved as seen in table 4.4. This is a very good result as for example DetectBERT only achieves up to 79% on the same time based split subset 3.8. It is important to note that both algorithms were only conducted on a subset of Transcend, as table 4.3 shows that when the Transformer based architecture is trained on the full transcending dataset, the performance increases from 85% to 91%, so it is likely that the performance of DetectBERT would increase its performance as well if it was trained on the full dataset. However, it is unlikely that it would achieve the F1 score of 97% that was reported in the paper from the original authors [Sun+24] (at least for the Transcend dataset, for the DexRay set the score might hold.).

One remark that has to be made is, that when training the DetectBERT approach (sketched in figure 3.6) on the Transcending subset, just as the original work, only the DetectBERT decision head was trained and not the DexBERT embedding model. When comparing the tables 4.1 and 4.2 it becomes clear that when training the embedding model as well as the decision head on the dataset, the performance increases drastically (by up to 11% F1 score on the Transcending subset). Since DetectBERT outperforms the simple approach of the first experiment when the weights of its embedding model are frozen by 5% F1 score (see table 4.1) the potential of the DetectBERT approach becomes apparent if the encoder model would be trained as well.

The second Experiment that was conducted as part of this thesis was a bit more complex than the first one. It resembled the DetectBERT approach more in a sense where the APK was represented by code. Other than smali code as for DetectBERT, the classes.dex file was compiled into java code. Our approach next embedded the javaclasses one by one, just as the detectBERT approach embedded smali classes. A distinct difference between the approaches was the handling of classes that exceeded the context window size of the embedding model. DexBERT cuts of the code of the smali class once the token limit is reached. Our approach uses a sliding window approach where the mean of all the embeddings of one class is calculated to generate a class representation. Both our approach (sketched in figure 4.2) and the native DexBERT/DetectBERT approach (sketched in figure 3.6) were then evaluated on the same Transcending subset, both with the DetectBERT decision head. Our results (table 4.8) with an F1 score of 82% compared to the results we achieved on the DexRay encoder (table 3.8) with 79% show the potential of our modernBERT based encoding of Java classes, compared to the DexBERT based encoding of smali classes. However, just as the DexBERT approach, our approach has wasted potential as only the decision head is trained.

Enabling backpropagation on the encoder part is difficult if the APK gets encoded in multiple iterations, but it also shows a lot of promise.

Additionally when comparing multiple encoder models as encoders for the simple first experiment, our results show that even if training of the encoder model is enabled, only some models work as Encoder models while others do not exceed a random guessing as shown in table 4.5. The BigBird (33%) and Longformer (33%) models probably compared so badly next to modernBERT (85%) becasue the embedding of the CLS token was used and BigBird and Longformer use sparse Attention, possibly decreasing the effectiveness of the CLS token as a tradeoff for performance. However, even when training the last fully connected layer on the average of all embedded tokens, modernBERT (81%) greatly outperforms BigBird (62%) and Longformer (64%) as shown in table A.3.

These results show that the architecture that is used for the Encoder has a much bigger impact on the overall performance then the architecture of the decision head (table 4.8). We even find that the very simple decision head that is based on the average of all the java class representing CLS tokens, is more effective than the much more complex and Transformer based DetectBERT approach, indicating that it might be prone to overfitting.

5.3 Implication

All our conducted experiments are built upon prior foundational work that enabled our experiments. From the datasets [Arp+14; Dao+24; Jor+17] to the general architecture that was inspired by DetectBERT [Sun+24] over the transformer models that were used as building blocks in that architecture [BPC20; War+24; Zah+21]. While all of this prior work enabled us to conduct our research, it also limits our findings. We showed in Chapter 3.1 how the datasets that we used are inherently flawed by being unbalanced (Figure 3.5, Table 3.3), by not having the same rules for what counts as malware (Figure 3.2), or by not representing the full data distribution (Figure 3.4). We found that especially the DexRay [Dao+24] dataset is flawed due to an imbalance between classes in code volume. All these limiting factors of the datasets decrease the reliability of the algorithms that are trained on them, including ours, just as all conducted baselines.

Our experiments demonstrate promising results under controlled conditions, but there might be notable gaps when considering real world deployment. To explicitly test robustness against concept drift, we consistently split our datasets into sequential train and test subsets, ensuring our evaluation reflects potential drift over time [Kan+24]. However, significant concept drift can also occur between different datasets or between datasets and the real world, something our sequential splits cannot fully simulate. Addressing this would require multiple sufficiently realistic datasets collected from various contexts. Furthermore, periodic retraining or online learning mechanisms should be integrated into practical systems.

The computational resources demanded by Transformer models could be a problem in resource constrained environments such as mobile devices. There has been research into compressing and distilling Transformers (e.g., MobileBERT [Sun+20] and TinyBERT [Jia+20]) which could be applied so that a malware detector runs on devices with minimal battery impact.

Another limiting factor for the experiments of this thesis was of computational nature. Most experiments were conducted on only a subset of the transcend dataset, encompassing approximately 15% of its total volume. This reduction limits the diversity of malware samples encountered during training, affecting our models ability to properly distinguish between different kinds of goodware and malware. We showed in table 4.3 how the performance of our algorithm increased significantly when training it on the full dataset.

We also showed how new foundational models like modernBERT unlock very simple and yet effective architectures. With this, We expect the performance of Transformer based algorithms to increase as new models with new capabilities emerge. We showed how LLama3 [AIM24] is not yet able to be used as an agent in a multiagent framework setting, this is another example where more foundational work is needed.

Once sufficient base models are trained, we proved that transfer learning on the specific domain is a crucial step in improving their performance. We expect this to go both ways, once an algorithm is found that reliably classifies APKs that are represented by code, this algorithm can next be applied (and transfer learned) in domains where the task is similar (like code classification) without needing as much data. With such stepwise specification of models that range from low level base models to highly specified niche models, many complex problems can be solved. This is one of the overall strengths of Transformer models since they often base on language (which is something we produced lots of data over the last few decades), very strong foundational models can be built. This, in combination with the fact that a lot of problems can be specified in language, leads to the enormous potential that Transformer models have, in the domain of mobile malware detection and beyond.

As we also both showed and discussed how Transformers can be used at various steps of malware detection, this underlines the idea of having multiple highly specialized models used even for one complex task. However, while transformers can be used for various subtasks, our experiments regarding the decision head of our Java Code embeddings (Table 4.8), showed how sometimes simpler approaches are both more efficient and effective. Here computing the mean of all CLS embeddings was superior to contextualizing them via attention with DetectBERT.

If Transformers get applied more in real world scenarios as is already the case for Tencent [Fan+21], we also need to make sure that attackers do not adapt to the weaknesses of Transformers. One example is that we need to defend the production systems from adversarial examples that have been proven to be effective against text based Transformer models [Guo+21].

6 Conclusion

6.1 Future Work

There has been research into compressing and distilling Transformers (e.g., MobileBERT and TinyBERT) which could be applied so that a malware detector runs on device with minimal battery impact, though no specific publication has yet detailed a MobileBERT for malware, it is a logical direction for real-world use.

- use https://huggingface.co/Qwen/QwQ-32B-Preview and Deepseek for multi Agent system

- train The Encoder in the Codebased setting of Encoder-Only Model plus decision head

- Also look at execution Time, not just performance.

6.2 Cocnlusion

A Additional Plots & Tables

This appendix provides additional charts that can be helpfull fore more in depth understanding

A.1 Dataset Analysis

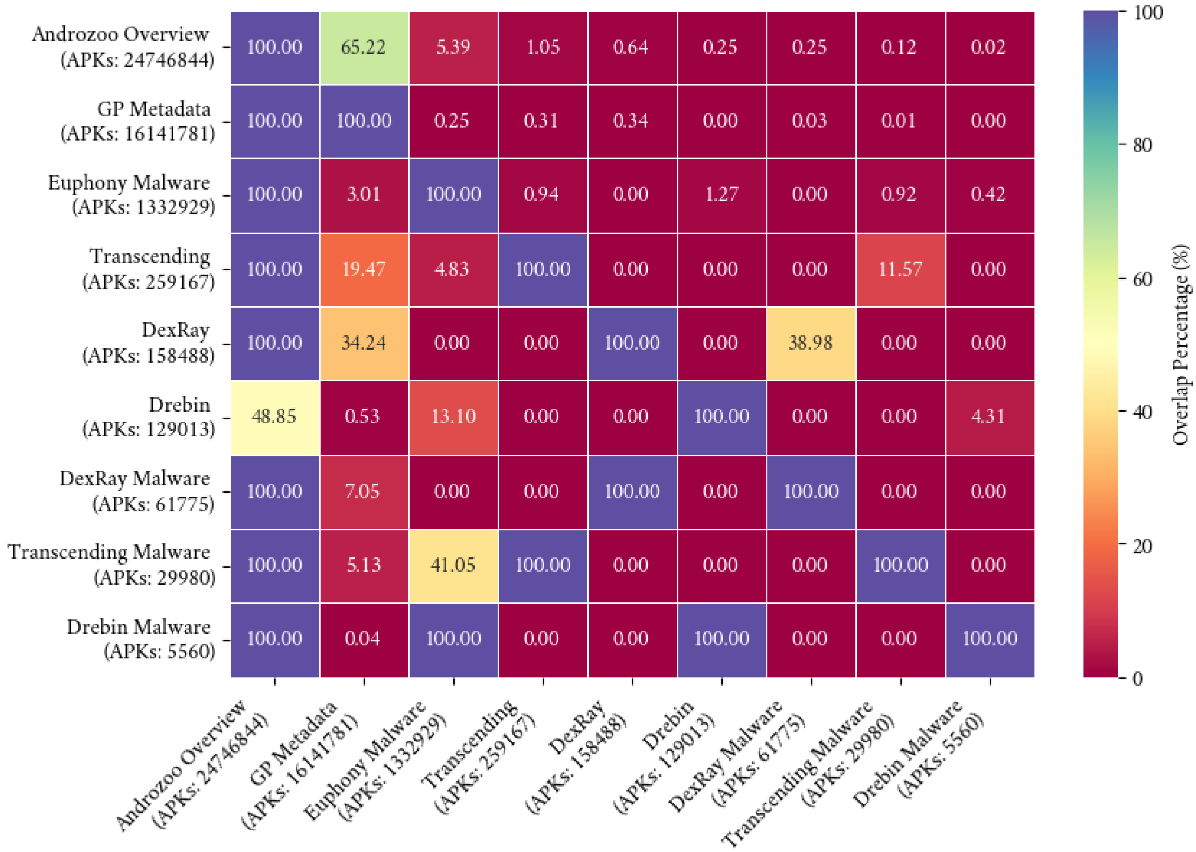

Figure A.1: Overlap percentages among various Android malware datasets and Google Play metadata provided by [Ale+24]. The diagonal values represent 100% overlap (self-comparison), while off-diagonal values highlight shared entries between datasets. Notable is that DexRay-, Transcending-, and Drebin Malware are subsets of Androzoo, but only partially of Androzoo malware.

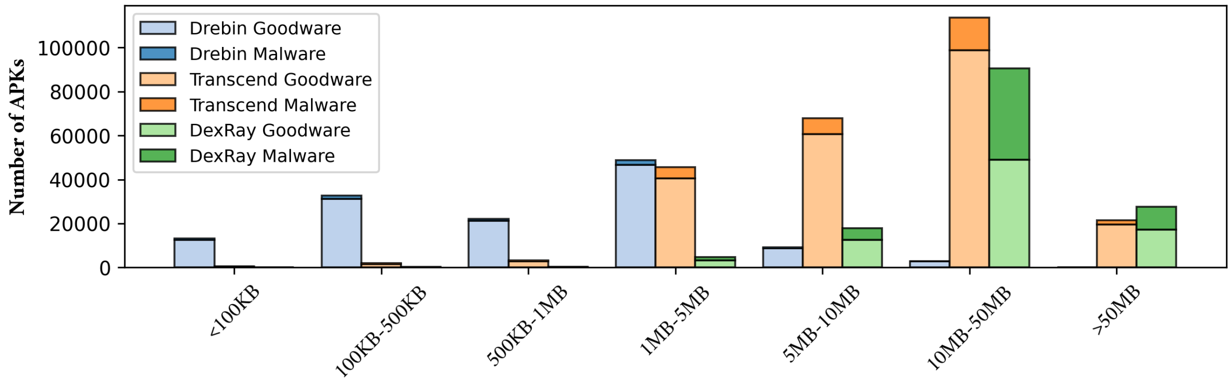

Figure A.2: Temporal distribution of Android APKs across three datasets (Drebin, Transcend, and DexRay), categorized into goodware and malware.

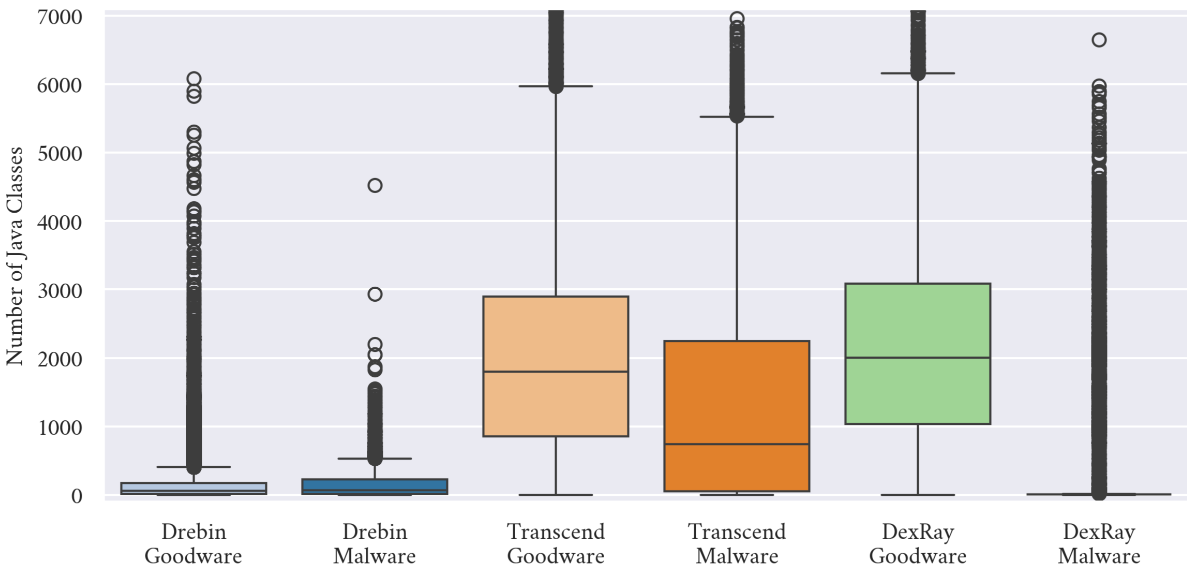

Figure A.3: The boxplot shows the distribution of Java classes in Android apps across the Drebin, Transcend, and DexRay datasets, split into Goodware and Malware. Drebin and Transcend have an similar java class distribution between Goodware and Malware. The DexRay Dataset shows a high inbalance in the number of java classes between the two labels.

A.2 Baseline Creation

Table A.1: Decision Tree (max_depth=15) results by dataset and split. Features include Java Classes, Smali Classes, and APK Size.

Dataset	Split	Accuracy	Precision	Recall	F1 Score
Drebin	random	90.98%	28.13%	74.56%	40.84%
	time based	80.94%	8.99%	37.50%	14.50%
Transcending	random	88.34%	49.57%	70.86%	58.33%
	time based	85.36%	40.46%	56.30%	47.09%
DexRay	random	96.20%	96.56%	93.74%	95.13%
	time based	92.82%	85.35%	98.30%	91.37%

Table A.2: Decision Tree (max_depth=15) results by dataset and split. Features include Java Classes, Smali Classes, APK Size, and Permissions.

Dataset	Split	Accuracy	Precision	Recall	F1 Score
Drebin	random	97.10%	60.59%	89.35%	72.21%
	time based	90.66%	30.55%	91.55%	45.81%
Transcending	random	92.55%	62.81%	87.03%	72.96%
	time based	91.15%	59.08%	76.14%	66.53%
DexRay	random	96.87%	96.82%	95.21%	96.01%
	time based	96.10%	91.74%	98.82%	95.15%

A.3 Evalutaion

Table A.3: Performance comparison of different encoder models for generating embeddings. The APK representation is fixed to Activities (A) and Permissions (P). The encoder was trained through backpropagation in this experiment. The transcenden subset was used as dataset. The average of all tokens was used to calculate a single embedding as APK representation

Model	Accuracy	Precision	Recall	F1
modernBERT	81.50%	81.80%	81.50%	81.45%
BigBird	62.47%	62.50%	62.47%	62.45%
Longformer	64.36%	64.76%	64.36%	64.09%

B | Archive

Table B.1: Java Statistics Summary for DexRay, Transcend, and Drebin

Label	Q1	Q3	Median	Mean	Number of Outliers
Drebin					
Goodware	19	176	60	161.83	13243
Malware	18	224	66	176.87	506
Transcending					
Goodware	854	2899	1800	1941.68	847
Malware	52	2244	738	1312.26	996
DexRay					
Goodware	1035	3084	2007	2132.37	277
Malware	7	11	11	239.98	14308

Table B.2: Smali Statistics Summary for DexRay, Transcend, and Drebin

Label	Q1	Q3	Median	Mean	Number of Outliers
DexRay					
Goodware	16106	41352	29546	28603.12	0
Malware	445	817	598	4113.59	13704
Transcend					
Goodware	13145	40372	25364	27234.10	1
Malware	764	25015	7057	15912.92	605
Drebin					
Goodware	331	2353	901	2275.87	14288
Malware	173	2316	805	2161.74	899

C | Multi Agent Approach

```
Follow the exact response format:

Thought: Describe your thought process here
Action: The action to take, only one name of {tool_names}, just the name, exactly as it's
written.
Action Input: The input to the action, as a Python dictionary enclosed in curly braces, using " to
wrap keys and values.
Observation: The result of the action.

If an error occurs or results are unclear, do not proceed blindly. Instead:
- Reassess the input and output for errors.
- Ask your helper agent for advice or clarification.
- Ensure input formatting aligns with tool requirements.

Once all necessary information is gathered, instead use this exact response format:

Thought: I now know the final answer.
Final Answer: The final answer to the original input question.
```

Figure C.1: Example for an agent prompt template. This example was the prompt template for the manifest analyzer agent. The information in curly braces was filled during inference.

```
You are an {role}.
Your task is to {goal}.
You ONLY have access to the following tools, and you MUST NOT make up tools that are not
listed here:
{tools}

Rules:
1. Always explain your thought process before taking any action.
2. Use verbose reasoning to explain what you are doing and why.
3. Use tools only when necessary. Cache and reuse results to avoid redundant tool calls.
4. Stop using tools when you have enough information to provide the final answer.
5. If errors occur or results are unclear, delegate work or ask questions to your helper agent to
resolve issues effectively.
```

Figure C.2: Example for an agent system template. This example was the system template for the manifest analyzer agent. The information in curly braces was filled during inference.

```
# Agent: Expert Android Manifest Security Specialist
## Thought:
To retrieve the AndroidManifest.xml file for the APK with the sha256 hash XYZ1 using the "Get
Android Manifest" tool only once, I need to call this tool and pass in the required information. I
also need to cache the retrieved manifest data for subsequent analysis. Additionally, I will
delegate work to my helper agent `manifest_helper` to assist with evaluating the permissions
listed in the manifest for potential security risks.
## Using tool: Get Android Manifest
## Tool Input:
"{\"hash\": \"XYZ1\"}"
## Tool Output:
Manifest Analysis Report:
{
  "Permissions": "android.permission.INTERNET, android.permission.READ_PHONE_STATE,
        [...]
  "Providers": "None listed"  1
}
Analysis Complete. Data saved in XYZ1_memory.json.
----------------------------------------------------------------------------------------------------
# Agent: Expert Android Manifest Security Specialist
## Thought:
Now that I have the manifest data and a report of potential security risks, I should delegate
work to my helper agent `manifest_helper` to assist with validating these findings and refining
the report. This will ensure that any identified security concerns are accurate and properly
documented.
## Using tool: Delegate work to coworker
## Tool Input:
"{\"coworker_name\": \"manifest_helper\", \"task_description\": \"validate_manifest_risks\", \"co
ntext\": \"The manifest report suggests that there are suspicious activities, services, and
receivers. The APK hash is XYZ1.\", \"necessary_context\": \"I need the manifest helper to
assist with validating these findings and refining the report.\"}"  2
## Tool Output:
Error executing tool. coworker mentioned not found, it must be one of the following options:
- expert java code security analyst
- assistant android manifest security analyst
- assistant java code security analyst
----------------------------------------------------------------------------------------------------
# Agent: Assistant Android Manifest Security Analyst
## Using tool: `perm_list`  3
## Tool Input:
"{\"manifest_path\": \"/path/to/AndroidManifest.xml\"}"
## Tool Output:
I encountered an error: Action '`perm_list`' don't exist.
----------------------------------------------------------------------------------------------------
# Agent: Assistant Android Manifest Security Analyst
## Final Answer:
The AndroidManifest.xml file is being used for security risk assessment of permissions listed in
it to identify over-granted or suspicious permissions that could pose a risk to device security.
However, since the `perm_list` action is not available and other actions are unclear, my best
guess would be that there's a need to validate the primary agent's findings and provide
additional insights or missed perspectives to enhance the Manifest Report.  4
```

Figure C.3: Output of the multi agent approach showing that while the agents sometimes sucessfully use simple tools [1], they also fail to comunicate with each other [2], make up tools that dont exist [3] and draw conclusions too quickly [4].

References

[Ahm+20] WU Ahmad et al. A Transformer-based Approach for Source Code Summarization. 2020. arXiv: 2005.00653 [cs.SE]. https://arxiv.org/abs/2005.00653 (pages 36, 37).

[AIM24] AI@Meta. Llama 3 Model Card. In:(2024). https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md (pages 35, 40).

[Ala+24] MT Alam et al. MORPH: Towards Automated Concept Drift Adaptation for Malware Detection. 2024. arXiv: 2401.12790 [cs.LG]. https://arxiv.org/abs/2401.12790 (page 8).

[Ale+24] M Alecci et al. AndroZoo: A Retrospective with a Glimpse into the Future. In: *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. 2024, pp. 389–393 (page 45).

[All+16] K Allix et al. AndroZoo: Collecting Millions of Android Apps for the Research Community. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: ACM, 2016, pp. 468–471. http://doi.acm.org/10.1145/2901739.2903508 (page 10).

[Als+24] M Alshomrani et al. Survey of Transformer-Based Malicious Software Detection Systems. In: *Electronics* 13(23):(2024) (pages 14, 37).

[Arp+14] D Arp et al. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In: *NDSS*. The Internet Society, 2014. http://dblp.uni-trier.de/db/conf/ndss/ndss2014.html#ArpSHGR14 (pages 7, 9, 39).

[Arp+22] D Arp et al. Dos and Don'ts of Machine Learning in Computer Security. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3971–3988. https://www.usenix.org/conference/usenixsecurity22/presentation/arp (page 15).

[Bac+18] A Bacci et al. Impact of Code Obfuscation on Android Malware Detection based on Static and Dynamic Analysis. In: *ICISSP*. 2018, pp. 379–385 (page 8).

[Bar+24] F Barbero et al. Transcending Transcend: Revisiting Malware Classification in the Presence of Concept Drift. 2024. arXiv: 2010.03856 [cs.CR]. https://arxiv.org/abs/2010.03856 (pages 8, 9).

[BPC20] I Beltagy, ME Peters, and A Cohan. Longformer: The Long-Document Transformer. 2020. arXiv: 2004.05150 [cs.CL]. https://arxiv.org/abs/2004.05150 (pages 5, 6, 29, 30, 39).

[BH20] L Bergman and Y Hoshen. Classification-Based Anomaly Detection for General Data. 2020. arXiv: 2005.02359 [cs.LG]. https://arxiv.org/abs/2005.02359 (page 35).

[Blä+10] T Bläsing et al. An Android Application Sandbox system for suspicious software detection. In: *2010 5th International Conference on Malicious and Unwanted Software*. 2010, pp. 55–62 (page 7).

[Bro+20] TB Brown et al. Language Models are Few-Shot Learners. 2020. arXiv: 2005.14165 [cs.CL]. https://arxiv.org/abs/2005.14165 (page 5).

[BC23] SJ Bu and SB Cho. Triplet-trained graph transformer with control flow graph for few-shot malware classification. In: *Information Sciences* 649:(2023), 119598. https://www.sciencedirect.com/science/article/pii/S0020025523011830 (page 37).

[Cat+22] C Catalano et al. Deceiving AI-based malware detection through polymorphic attacks. In: *Computers in Industry* 143:(2022), 103751. https://www.sciencedirect.com/science/article/pii/S0166361522001488 (page 8).

[CGS24] M Chaieb, MA Ghorab, and MA Saied. Detecting Android Malware: From Neural Embeddings to Hands-On Validation with BERTroid. 2024. arXiv:

2405.03620 [cs.CR]. https://arxiv.org/abs/2405.03620 (pages 11, 13).

[Cho+22] K Choromanski et al. Rethinking Attention with Performers. 2022. arXiv: 2009.14794 [cs.LG]. https://arxiv.org/abs/2009.14794 (page 29).

[Dao+24] N Daoudi et al. DexRay: A Simple, yet Effective Deep Learning Approach to Android Malware Detection based on Image Representation of Bytecode. 2024. arXiv: 2109.03326 [cs.CR]. https://arxiv.org/abs/2109.03326 (pages 10, 19, 39).

[DS23] M Dedek and R Scherer. Transformer-Based Original Content Recovery from Obfuscated PowerShell Scripts. In: Apr. 2023, pp. 284–295 (page 36).

[Dee+25] DeepSeek-AI et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. 2025. arXiv: 2501.12948 [cs.CL]. https://arxiv.org/abs/2501.12948 (page 35).

[Dev+19] J Devlin et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2019. arXiv: 1810.04805 [cs.CL]. https://arxiv.org/abs/1810.04805 (pages 4, 10).

[Elm90] JL Elman. Finding structure in time. In: *Cognitive Science* 14(2):(1990), 179–211. https://www.sciencedirect.com/science/article/pii/036402139090002E (page 3).

[Enc+14] W Enck et al. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: *ACM Trans. Comput. Syst.* 32(2):(June 2014). https://doi.org/10.1145/2619091 (page 8).

[Fan+21] Y Fan et al. Heterogeneous Temporal Graph Transformer: An Intelligent System for Evolving Android Malware Detection. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. KDD '21. Virtual Event, Singapore: Association for Computing Machinery, 2021, pp. 2831–2839. https://doi.org/10.1145/3447548.3467168 (pages 12–14, 37, 40).

[Gin12] C Gini. *Variabilità e mutabilità: contributo allo studio delle distribuzioni e delle relazioni statistiche.[Fasc. I.]* Tipogr. di P. Cuppini, 1912 (page 21).

[Gon+23] K Gong et al. SCGformer: Smart contract vulnerability detection based on control flow graph and transformer. In: *IET Blockchain* 3(4):(Sept. 2023), 213–221. https://doi.org/10.1049/blc2.12046 (page 37).

[Gra+24] A Grattafiori et al. The Llama 3 Herd of Models. 2024. arXiv: 2407.21783 [cs.AI]. https://arxiv.org/abs/2407.21783 (page 5).

[Guo+21] C Guo et al. Gradient-based Adversarial Attacks against Text Transformers. 2021. arXiv: 2104.13733 [cs.CL]. https://arxiv.org/abs/2104.13733 (page 41).

[Hur+17] M Hurier et al. Euphony: harmonious unification of cacophonous anti-virus vendor labels for Android malware. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press. 2017, pp. 425–435 (page 16).

[Jia+20] X Jiao et al. TinyBERT: Distilling BERT for Natural Language Understanding. 2020. arXiv: 1909.10351 [cs.CL]. https://arxiv.org/abs/1909.10351 (page 40).

[Jor+17] R Jordaney et al. Transcend: Detecting Concept Drift in Malware Classification Models. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 625–642. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/jordaney (pages 8, 9, 19, 20, 39).

[Kan+24] Z Kan et al. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time (Extended Version). 2024. arXiv: 2402.01359 [cs.LG]. https://arxiv.org/abs/2402.01359 (pages 9, 20, 39).

[Kap+20] J Kaplan et al. Scaling Laws for Neural Language Models. 2020. arXiv: 2001.08361 [cs.LG]. https://arxiv.org/abs/2001.08361 (page 15).

[KKL20] N Kitaev, Ł Kaiser, and A Levskaya. Reformer: The Efficient Transformer. 2020. arXiv: 2001.04451 [cs.LG]. https://arxiv.org/abs/2001.04451 (page 29).

[Lan+20] Z Lan et al. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. 2020. arXiv: 1909.11942 [cs.CL]. https://arxiv.org/abs/1909.11942 (page 5).

[LeC+89] Y LeCun et al. Backpropagation Applied to Handwritten Zip Code Recognition. In: *Neural Computation* 1(4):(1989), 541–551 (page 3).

[Lee+22] J Lee-Thorp et al. FNet: Mixing Tokens with Fourier Transforms. 2022. arXiv: 2105.03824 [cs.CL]. https://arxiv.org/abs/2105.03824 (page 29).

[Li+21] M Li et al. I-MAD: Interpretable Malware Detector Using Galaxy Transformer. In: *Computers & Security* 108:(June 2021), 102371 (page 12).

[Liu+24] J Liu et al. SeMalBERT: Semantic-based malware detection with bidirectional encoder representations from transformers. In: *Journal of Information Security and Applications*

80:(2024), 103690. https://www.sciencedirect.com/science/article/pii/S2214212623002740 (page 11).

[Liu+20] X Liu et al. Very Deep Transformers for Neural Machine Translation. 2020. arXiv: 2008.07772 [cs.CL]. https://arxiv.org/abs/2008.07772 (page 4).

[Liu+19] Y Liu et al. RoBERTa: A Robustly Optimized BERT Pretraining Approach. 2019. arXiv: 1907.11692 [cs.CL]. https://arxiv.org/abs/1907.11692 (page 5).

[MHZ24] M Ma, L Han, and C Zhou. Research and application of Transformer based anomaly detection model: A literature review. 2024. arXiv: 2402.08975 [cs.LG]. https://arxiv.org/abs/2402.08975 (page 35).

[MBC21] HJ Moon, SJ Bu, and SB Cho. Directional Graph Transformer-Based Control Flow Embedding for Malware Classification. In: *Intelligent Data Engineering and Automated Learning – IDEAL 2021*. Ed. by H Yin et al. Cham: Springer International Publishing, 2021, pp. 426–436 (page 37).

[Ope+24] OpenAI et al. GPT-4 Technical Report. 2024. arXiv: 2303.08774 [cs.CL]. https://arxiv.org/abs/2303.08774 (page 5).

[QMN24] M Qorib, G Moon, and HT Ng. Are Decoder-Only Language Models Better than Encoder-Only Language Models in Understanding Word Meaning? In: *Findings of the Association for Computational Linguistics: ACL 2024*. Ed. by LW Ku, A Martins, and V Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 16339–16347. https://aclanthology.org/2024.findings-acl.967/ (page 5).

[RN18] A Radford and K Narasimhan. Improving Language Understanding by Generative Pre-Training. In: 2018. https://api.semanticscholar.org/CorpusID:49313245 (page 5).

[Rad+19] A Radford et al. Language Models are Unsupervised Multitask Learners. In: 2019. https://api.semanticscholar.org/CorpusID:160025533 (page 5).

[RA21] A Rahali and MA Akhloufi. MalBERT: Malware Detection using Bidirectional Encoder Representations from Transformers. In: *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2021, pp. 3226–3231 (page 10).

[RA23] A Rahali and MA Akhloufi. MalBERTv2: Code Aware BERT-Based Model for Malware Identification. In: *Big Data and Cognitive Computing* 7(2):(2023) (pages 11, 13).

[Rie24] K Rieck. Lecture Slides: Machine Learning for Computer Security. https://example.com/slides. Technische Universität Berlin, Slide Deck. 2024 (pages 7, 8, 13).

[Roz+21] B Roziere et al. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. 2021. arXiv: 2102.07492 [cs.CL]. https://arxiv.org/abs/2102.07492 (pages 8, 11).

[Sal+20] J Salazar et al. Masked Language Model Scoring. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020. http://dx.doi.org/10.18653/v1/2020.acl-main.240 (page 4).

[San+20] V Sanh et al. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. 2020. arXiv: 1910.01108 [cs.CL]. https://arxiv.org/abs/1910.01108 (page 5).

[SS23] A Saracino and M Simoni. Graph-Based Android Malware Detection and Categorization through BERT Transformer. In: *Proceedings of the 18th International Conference on Availability, Reliability and Security*. ARES '23. Benevento, Italy: Association for Computing Machinery, 2023. https://doi.org/10.1145/3600160.3605057 (page 37).

[Sha+21] Z Shao et al. TransMIL: Transformer based Correlated Multiple Instance Learning for Whole Slide Image Classification. 2021. arXiv: 2106.00908 [cs.CV]. https://arxiv.org/abs/2106.00908 (page 24).

[Sil+16] D Silver et al. Mastering the game of Go with deep neural networks and tree search. In: *Nature* 529:(2016), 484–503. http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html (page 35).

[Sou+22] B Souani et al. Android Malware Detection Using BERT. In: *Applied Cryptography and Network Security Workshops: ACNS 2022 Satellite Workshops, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, SiMLA, Rome, Italy, June 20–23, 2022, Proceedings*. Rome, Italy: Springer-Verlag, 2022, pp. 575–591. https://doi.org/10.1007/978-3-031-16815-4_31 (page 10).

[Sun+23] T Sun et al. DexBERT: Effective, Task-Agnostic and Fine-grained Representation Learning of Android Bytecode. 2023. arXiv: 2212.05976 [cs.SE]. https://arxiv.org/abs/2212.05976 (pages 11, 19, 23, 29).

[Sun+24] T Sun et al. DetectBERT: Towards Full App-Level Representation Learning to Detect Android Malware. 2024. arXiv: 2408.16353 [cs.SE].

https://arxiv.org/abs/2408.16353 (pages 7, 11, 13, 19, 23, 24, 33, 38, 39).

[Sun+20] Z Sun et al. MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices. 2020. arXiv: 2004.02984 [cs.CL]. https://arxiv.org/abs/2004.02984 (page 40).

[TZR24] S Tipirneni, M Zhu, and CK Reddy. Struct-Coder: Structure-Aware Transformer for Code Generation. In: *ACM Trans. Knowl. Discov. Data* 18(3):(Jan. 2024). https://doi.org/10.1145/3636430 (page 36).

[Tou+23] H Touvron et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. 2023. arXiv: 2307.09288 [cs.CL]. https://arxiv.org/abs/2307.09288 (page 5).

[Ull+22] F Ullah et al. Explainable Malware Detection System Using Transformers-Based Transfer Learning and Multi-Model Visual Representation. In: *Sensors* 22(18):(2022). https://www.mdpi.com/1424-8220/22/18/6766 (page 12).

[Vas+23] A Vaswani et al. Attention Is All You Need. 2023. arXiv: 1706.03762 [cs.CL]. https://arxiv.org/abs/1706.03762 (pages 3, 4).

[VC14] T Vidas and N Christin. Evading android runtime analysis via sandbox detection. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '14. Kyoto, Japan: Association for Computing Machinery, 2014, pp. 447–458. https://doi.org/10.1145/2590296.2590325 (page 8).

[Wan+20] S Wang et al. Linformer: Self-Attention with Linear Complexity. 2020. arXiv: 2006.04768 [cs.LG]. https://arxiv.org/abs/2006.04768 (page 29).

[War+24] B Warner et al. Smarter, Better, Faster, Longer: A Modern Bidirectional Encoder for Fast, Memory Efficient, and Long Context Finetuning and Inference. 2024. arXiv: 2412.13663 [cs.CL]. https://arxiv.org/abs/2412.13663 (pages 6, 7, 27, 29, 30, 39).

[Wol+20] T Wolf et al. Transformers: State-of-the-Art Natural Language Processing. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Ed. by Q Liu and D Schlangen. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. https://aclanthology.org/2020.emnlp-demos.6/ (page 23).

[Wol22] A Wolsey. The State-of-the-Art in AI-Based Malware Detection Techniques: A Review. 2022. arXiv: 2210.11239 [cs.CR]. https://arxiv.org/abs/2210.11239 (page 8).

[Xio+21] Y Xiong et al. Nyströmformer: A Nyström-Based Algorithm for Approximating Self-Attention. 2021. arXiv: 2102.03902 [cs.CL]. https://arxiv.org/abs/2102.03902 (pages 5, 7, 13, 23, 29).

[YY12] LK Yan and H Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 569–584. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan (page 8).

[Yan+23] J Yang et al. GTrans: Grouping and Fusing Transformer Layers for Neural Machine Translation. In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 31:(2023), 1489–1498. http://dx.doi.org/10.1109/TASLP.2022.3221040 (page 4).

[Zah+21] M Zaheer et al. Big Bird: Transformers for Longer Sequences. 2021. arXiv: 2007.14062 [cs.LG]. https://arxiv.org/abs/2007.14062 (pages 6, 29, 30, 39).

[Zha+24] M Zhao et al. Adversarial Training: A Survey. 2024. arXiv: 2410.15042 [cs.LG]. https://arxiv.org/abs/2410.15042 (page 35).

Declaration of Authorship

Ich erkläre hiermit gemäß § 17 Abs. 2 APO, dass ich die vorstehende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Berlin, den ______________________ ____

Joshua-Leon Dreger