DexBERT: Effective, Task-Agnostic and Fine-Grained Representation Learning of Android Bytecode

Tiezhu Sun, Kevin Allix, Kisub Kim, Xin Zhou, Dongsun Kim, David Lo, *Fellow, IEEE*, Tegawendé F. Bissyandé, and Jacques Klein, *Member, IEEE*

***Abstract*—The automation of an increasingly large number of software engineering tasks is becoming possible thanks to Machine Learning (ML). One foundational building block in the application of ML to software artifacts is the *representation* of these artifacts (*e.g.*, source code or executable code) into a form that is suitable for learning. Traditionally, researchers and practitioners have relied on manually selected features, based on expert knowledge, for the task at hand. Such knowledge is sometimes imprecise and generally incomplete. To overcome this limitation, many studies have leveraged representation learning, delegating to ML itself the job of automatically devising suitable representations and selections of the most relevant features. Yet, in the context of Android problems, existing models are either limited to coarse-grained whole-app level (*e.g.*, `apk2vec`) or conducted for one specific downstream task (*e.g.*, `smali2vec`). Thus, the produced representation may turn out to be unsuitable for fine-grained tasks or cannot generalize beyond the task that they have been trained on. Our work is part of a new line of research that investigates effective, task-agnostic, and fine-grained universal representations of bytecode to mitigate both of these two limitations. Such representations aim to capture information relevant to various low-level downstream tasks (*e.g.*, at the class-level). We are inspired by the field of Natural Language Processing, where the problem of universal representation was addressed by building Universal Language Models, such as BERT, whose goal is to capture abstract semantic information about sentences, in a way that is reusable for a variety of tasks. We propose DexBERT, a BERT-like Language Model dedicated to representing chunks of DEX bytecode, the main binary format used in Android applications. We empirically assess whether DexBERT is able to model the DEX *language* and evaluate the suitability of our model in three distinct class-level software engineering tasks: Malicious Code Localization, Defect Prediction, and Component Type Classification. We also experiment with strategies to deal with the problem of catering to apps having vastly different sizes, and we demonstrate one example of using our technique to investigate what information is relevant to a given task.**

***Index Terms*—Representation learning, Android app analysis, code representation, malicious code localization, defect prediction.**

I. Introduction

PRE-TRAINED models yielding general-purpose embeddings have been a recent highlight in AI advances, notably in the research and practice of Natural Language Processing (*e.g.*, with BERT [1]). Building on these ideas, the programming language and software engineering communities have attempted similar ideas of learning vector representations for code (*e.g.*, CODE2VEC [2]) and other programming artifacts (*e.g.*, bug reports [3]). Unfortunately, these pre-trained models for code embedding often do not generalize beyond the task they have been trained on [4].

In the Android research landscape, many techniques have been proposed to address app classification problems [5], [6], [7], [8], [9], [10], [11], [12], most of which, however, can only handle coarse-grained tasks (*i.e.*, at the whole-app level). Although there are a few works [13], [14] targeting some fine-grained tasks at the class level, their learned representations have also not been shown to generalize to other class-level tasks.

Therefore, despite the good performance exhibited by existing approaches, there is still a research gap to be filled with the investigation of **simultaneously fine-grained and task-agnostic** representation learning for Android applications. Indeed, advances in this direction will help researchers and practitioners who are conducting class-level tasks, such as malicious code localization or app defect prediction, to achieve state-of-the-art performance while reducing costs due to manual feature engineering or repetitive pre-training computations for multiple representation models.

Building a fine-grained task-agnostic model is, however, challenging since it essentially requires to capture knowledge

Manuscript received 2 December 2022; revised 4 August 2023; accepted 23 August 2023. Date of publication 1 September 2023; date of current version 17 October 2023. This research / project is supported by the Fonds National de la Recherche (FNR), Luxembourg, under project REPROCESS C21/IS/16344458, AFR PhD Project number 17046335. This work was also supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A5A1021944 and 2021R1I1A3048013), and the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research and Development Programme, NCRP25-P03-NCR-TAU. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore. Recommended for acceptance by M. Nagappan. *(Corresponding authors: Kisub Kim; Tiezhu Sun; Dongsun Kim.)*

Tiezhu Sun, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein are with the University of Luxembourg, 1359 Kirchberg, Luxembourg (e-mail: tiezhu.sun@uni.lu; kevin.allix@uni.lu; tegawende.bissyande@uni.lu; jacques.klein@uni.lu).

Kisub Kim, Xin Zhou, and David Lo are with Singapore Management University, Singapore 188065 (e-mail: kisubkim@smu.edu.sg; davidlo@smu.edu.sg; xinzhou.2020@phdcs.smu.edu.sg).

Dongsun Kim is with Kyungpook National University, 41566 Daegu, Republic of Korea (e-mail: darkrsw@gmail.com).

Digital Object Identifier 10.1109/TSE.2023.3310874

relevant to a variety of tasks altogether and at the low granularity of the representation. A few studies have investigated and built task-agnostic models in the field of software engineering. CodeBERT [15] and apk2vec [16] are key representatives of these models. On the one hand, while CodeBERT brings significant improvements, it cannot be directly used for representing Android apps for two main reasons: (1) lack of source code in apps and (2) limit of input elements. Even though it is technically feasible to apply it to the assembly language `Smali`, the performance is unsatisfactory, as evidenced by our experiments in Section VI-C. On the other hand, apk2vec successfully constructs the behavior profiles of apps and achieves significant accuracy improvements while maintaining comparable efficiency. However, despite apk2vec's success in the app representation, its granularity, and graph-based design still have limitations [17], [18], [19], [20], [21]. Notably, apk2vec is designed to handle only app-level tasks, while our proposed approach is targeted at fine-grained tasks at the class-level.

Towards addressing limitations of existing representation techniques (*i.e.*, lack of a universal model for Android bytecode at low granularity), we propose, in this paper, DexBERT, a fine-grained and task-agnostic representation model for the bytecode in Android app packages. The result of DexBERT can be applied across various class-level downstream tasks (*e.g.*, malicious code localization, defect prediction, *etc.*). Our approach first extracts features from `Smali` instructions (*i.e.*, an assembly language for the Dalvik bytecode used by Android's Dalvik virtual machine). It then combines embeddings of code fragments to build a model that can address various class-level problems. Our pre-trained model can capture the essential features and knowledge by first learning an accurate general model of Android apps' bytecode. The proposed aggregation methods allow DexBERT to handle fine-grained class-level tasks, making DexBERT capable of operating on lower granularity of Android artifacts than other state-of-the-art representation models.

To evaluate the effectiveness of DexBERT, we first conduct a preliminary experiment to observe the feasibility of building a general model of Android app code. This experiment mainly focuses on pre-training BERT on `Smali` instructions to ensure that the generated embeddings contain meaningful features for various tasks. We observed clearly converging loss curves on all the pre-training tasks, with the pre-trained model achieving 95.30% and 99.35% accuracy on the masked language model and next sentence prediction tasks, respectively. Such performances demonstrate that our model indeed learned meaningful features and can be generalized to a variety of different tasks.

We further perform a comprehensive empirical evaluation to measure the overall performance of DexBERT on three class-level downstream tasks: 1) android malicious code localization, 2) android defect detection, and 3) component type classification. Android malicious code localization is a task whose goal is to identify the malicious parts of Android apps. Android defect detection locates defective code to help developers improve the security and robustness of Android apps. Component Type Classification is a multi-class classification problem that has a distinct character compared to the two tasks mentioned above. This classification problem is introduced to provide a more comprehensive evaluation of DexBERT's universality. Our experimental results show that DexBERT can localize malicious code and detect defects with significant improvement over current state-of-the-art approaches (74.93 and 6.33 percentage points improvement for malicious code localization and defect prediction, respectively) in terms of accuracy. DexBERT also significantly outperforms other BERT-like baselines on the task of Component Type Classification by a roughly 20 percentage point increase in terms of F1 Score.

The contributions of our study are as follows:

- We propose a novel BERT-based pre-trained representation learning model for Android bytecode representation, named DexBERT. It can be used directly on various class-level (*i.e.*, finer-grained than existing app-level approaches) downstream analysis tasks by freezing parameters of the pre-trained representation model when tuning the prediction model for a specific downstream task.
- We propose aggregation techniques that overcome the limitations with the size of the input of BERT.
- We conduct a comprehensive evaluation, showing that DexBERT achieves promising performance on multiple pre-training tasks and class-level Android downstream tasks.
- To ease replication, we share the dataset and source code to the community at the following address: https://github.com/Trustworthy-Software/DexBERT.

II. Background and Motivation

A. Representational Model

Our approach, DexBERT, aims at building a task-agnostic universal representation model for Android apps at low granularity. Universal representation models are trained on large corpora to automatically capture general features that are relevant to various downstream tasks without training a model from scratch for each task. For example, BERT [1] has revolutionized the representation problem of Natural Language Processing (NLP). One advantage of BERT-like models is the strong separation between the generic representation and any specific task. To build a language model, BERT pre-training relies on two fundamental tasks, Masked Language Model (MLM) and Next Sentence Prediction (NSP), without manual labels. MLM forces BERT to capture relationships between words, and NSP forces BERT to capture relationships between sentences. With such a mechanism, BERT can generate meaningful embeddings of the input sentences. Universal models can also be useful in the field of software engineering, where many tasks that take software artifacts as input are investigated. CodeBERT [15] is one of the successful universal models that is used by a number of studies [22], [23], [24], [25] in various software engineering domains with promising performance.

As the purpose of our approach, DexBERT, is to represent the bytecode in Android apps with concise representations, such as embedding vectors; we leverage and adapt the idea and architecture of BERT [1] as well. While BERT, coming from NLP, works with *words* and *sequences of words* (*i.e.*, sentences or paragraphs), we disassemble Android apps into Smali code[1]. We then regard the Smali code as a flow of tokens that can be fed to a BERT-like model.

B. Downstream Tasks

Once a BERT-like model is pre-trained, it is able to output a representation of its input. This representation can then be used as an input to other models that can be trained (fine-tuning in BERT terminology) for a specific downstream task. We evaluate our approach on three class-level Android-specific tasks: malicious code localization, defect prediction, and component type classification.

Precise malicious code localization not only helps to assess the trustworthiness of existing app-level malware detection methods but also enables important applications such as studying malware behavior and engineering malware signatures. Malicious code localization becomes particularly useful in the case of repackaged malware, as the major portion of apps' code remains benign, with only a small portion relevant to the attack [13].

Software defects are errors or bugs built into the software due to programmers' mistakes, such as memory overflows and run-time exceptions [26], during the software design or development process. These defects can raise serious reliability and security concerns. Automatically finding such defects is thus an important and active domain of research [14].

As an additional measure to assess DexBERT's wide-ranging applicability, we introduced component type classification, a third task at the class level. This task, which involves multi-class classification, serves as a contrast to the preceding two tasks that focused on security-related binary classification, and aims to provide a thorough evaluation of DexBERT's universality in different contexts.

In the Android context, these three tasks require sub-app level representations (*i.e.*, class-level) instead of whole-app level representations (*e.g.*, apk2vec [16]) since the tasks pinpoint a specific location in an app. Note that whole-app level representations transform an app into an embedding vector rather than transforming each element (*e.g.*, class) of an app. Our approach can take a subset of bytecode in an app, and thus it can represent classes as embedding vectors. Theoretically, other class-level tasks, beyond the three we identified, are expected to benefit from DexBERT as well, if the corresponding datasets are well-labeled and publicly available.

C. Motivation

As the population of Android apps is rapidly growing larger and the number of relevant issues (*e.g.*, productivity and security problems) is increasing quickly, it is necessary to efficiently address the issues of the Android ecosystem; for example, one solution to multiple problems. However, most state-of-the-art approaches focus on building a model for a particular issue, such as Malware Detection (*e.g.*, Drebin [27] and DexRay [28]) or Malicious Code Localization (*e.g.*, MKLDroid [13] and Droidetec [11]), or Software Defect Prediction (*e.g.*, smali2vec [14] and SLDeep [29]).

Instead of devising a single individual solution for each of those issues, there is value in investigating the possibility of having one single universal model that is able to capture and represent the relevant features and properties of application's code, which could be leveraged for a variety of tasks. Some existing representation models generalize to a variety of problems, but target either a low granularity with source code (*e.g.*, CodeBERT [15]) or entire Android applications (*e.g.*, apk2vec [16]). CodeBERT requires native source code (*i.e.*, written in programming languages), which is often not available for Android apps. While decompilation of Java code is an option, it is, however not always complete and is often significantly different from real source code. Moreover, the design of CodeBERT does not overcome the limitation with the number of 512 input elements for Android apps (*i.e.*, apps often contain more than millions of tokens). In addition, most existing Android representation approaches target the whole-app level (*e.g.*, apk2vec [16]), which makes it difficult to explore fine-grained details of the problems.

Therefore, it would be a significant step forward if a representation model can take bytecode of apps and support various tasks at fine-grained levels. The approach we propose, DexBERT, does not require source code and is validated to cater to class-level tasks. Meanwhile, we design an aggregation method to overcome the limitation of the 512 input elements in BERT-like models. For DexBERT users, they do not need to pre-train again and only need to use the provided model to generate features for their own APKs. Then, they can do any class-level tasks they desire. Therefore, its **reusability** can avoid training individual models for different tasks and save a large amount of time and effort.

III. APPROACH

In this section, we first present the overview of DexBERT workflow in Section III-A. Then, we illustrate details of DexBERT in Section III-B, and we describe the applications of representations learned by DexBERT on downstream tasks in Section III-C.

A. Overview

DexBERT focuses on extracting features from Android application Dalvik bytecode and targets fine-grained Android tasks (*i.e.*, at class-level). Our approach is clearly different from apk2vec [16], which is a static analysis based multi-view graph embedding framework for app-level Android tasks, and CodeBERT, which is a bimodal pre-trained model for programming language (PL) and natural language (NL) tasks. Specifically, DexBERT takes disassembled `Smali` code from Dalvik bytecode as input and learns to extract corresponding representations (*e.g.*, the embedding of a class). `Smali` is a text representation of Dalvik bytecode, in the same way, that assembly code is a text representation of compiled code. Because DexBERT supports various class-level tasks (while the original

[1] Smali is a popular disassembler for Android applications (https://github.com/JesusFreke/smali).

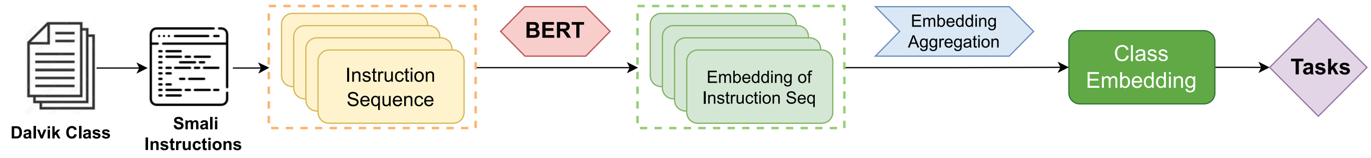


Fig. 1. Overview of a class embedding by DexBERT.

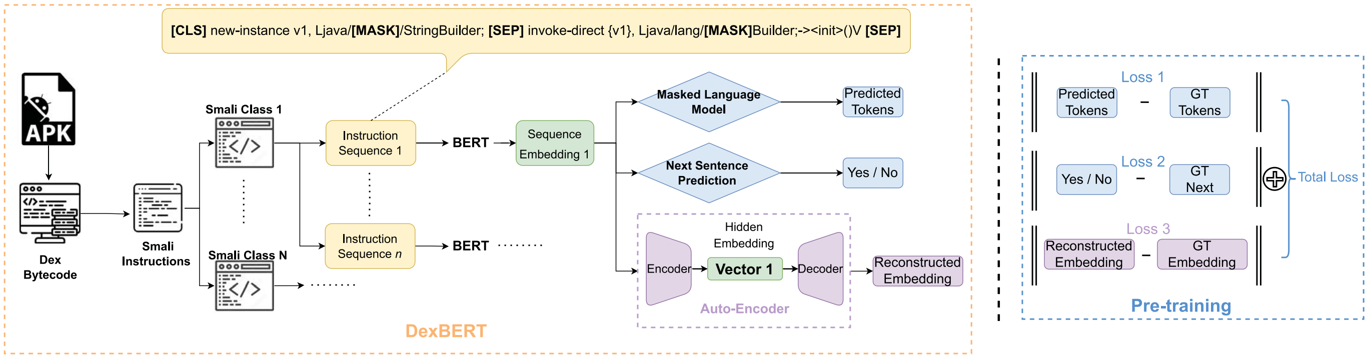


Fig. 2. Illustrations of DexBERT and pre-training loss function. “GT” is an abbreviation for “ground-truth”.

BERT is limited to relatively small inputs), there is a need to combine, or aggregate the representations of several sequences of instructions, or *chunks*, into one single representation that covers the chosen input.

As shown in Fig. 1, representation (or embedding) of the `Smali` Bytecode is learned by BERT during the pre-training phase. This learned embedding can then be applied to class-level Android downstream tasks. Specifically, we can easily extract the Bytecode of an Android application. After disassembling each Dalvik class, we obtain the `Smali` instructions. This flow in `Smali` instructions (grouped in chunks) is fed to the BERT model in order to pre-train it, *i.e.*, to learn how to represent `Smali` code. Then, to obtain the embedding of a `Smali` class, we aggregate the learned DexBERT representation of each `Smali` instruction sequence present in the class.

B. DexBERT

We introduce the pipeline of DexBERT in Section III.B.1. The mechanism of DexBERT pre-training is presented in Section III.B.2. In Section III.B.3, we introduce the Auto-Encoder, which is designed to reduce the dimensionality of the learned representation while keeping the key information.

1) DexBERT: As an assembly language, the `Smali` code disassembled from Dalvik bytecode is a sequence of instructions. Similar to the original BERT, we pre-train our model on multiple pre-training tasks to force DexBERT to capture general-purpose representations that can be used in various downstream tasks.

Specifically, as shown in the left box in Fig. 2, regarding each `Smali` instruction in each class as a text snippet we are able to create instruction pairs (similar to sentence pairs in original BERT) as input sequences. In each pair, two instructions are separated by a reserved token `[SEP]`, and several randomly selected tokens (or “words”) are masked. For the masked language model, one of the two pre-training tasks of the original BERT, the goal is to correctly predict the tokens that are masked. The second original BERT pre-training task, next sentence prediction, is turned here into next instruction prediction, leveraging the pairs of instructions. In essence, this task’s goal is to predict, given a pair of instructions, whether or not the second instruction follows the first one.

2) Pre-Training: Pre-training plays a vital role in helping DexBERT learn to generate meaningful embeddings. To ensure that the learned embeddings have general-purpose, the pre-training is supposed to be performed on multiple tasks simultaneously. As described, we adopt and adapt the two pre-training tasks of the original BERT: ① masked language model and ② next sentence prediction as the main pre-training tasks.

In each pre-training iteration, the input sequence of instructions is fed into the BERT model, which generates a corresponding sequence embedding (as shown in the left box in Fig. 2). The sequence embedding is then taken as input for the pre-training tasks. Each task is a simple neural network with a single fully connected layer. As shown in the right box in Fig. 2, a loss value is then calculated by comparing the output of each task head to the automatically created ground-truth (*i.e.*, randomly masked tokens or binary label indicating whether the second statement indeed follows the first one or not). The model weights of connections between neurons are adjusted to minimize the total loss value (*i.e.*, the sum of all loss values for pre-training tasks) based on the back-propagation algorithm [30]. Note that the pre-training tasks are only designed to help the BERT model learn meaningful features of input sequences.

3) Auto-Encoder: Even though the two aforementioned pre-training tasks could work well on `Smali` instructions, the dimensionality (*i.e.*, 512×768, which is defined as the multiplication of token number in each input sequence N and the dimension of the learned embedding vector H) of the generated representation for each sequence is quite large. Since there are

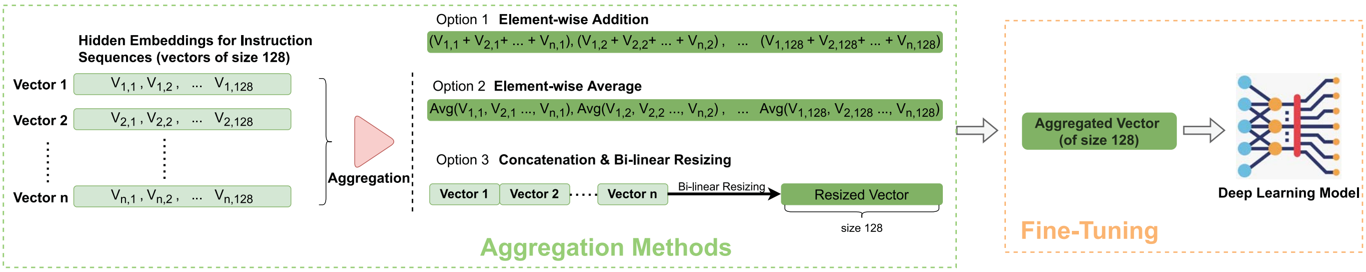

Hidden Embeddings for Instruction Sequences (vectors of size 128)
Vector 1
$V_{1,1}, V_{1,2}, \dots V_{1,128}$
Vector 2
$V_{2,1}, V_{2,2}, \dots V_{2,128}$
Vector n
$V_{n,1}, V_{n,2}, \dots V_{n,128}$
Aggregation
Option 1 Element-wise Addition
$(V_{1,1} + V_{2,1} + \dots + V_{n,1}), (V_{1,2} + V_{2,2} + \dots + V_{n,2}), \dots (V_{1,128} + V_{2,128} + \dots + V_{n,128})$
Option 2 Element-wise Average
$Avg(V_{1,1}, V_{2,1} \dots, V_{n,1}), Avg(V_{1,2}, V_{2,2} \dots, V_{n,2}), \dots Avg(V_{1,128}, V_{2,128} \dots, V_{n,128})$
Option 3 Concatenation & Bi-linear Resizing
Vector 1
Vector 2
Vector n
Bi-linear Resizing
Resized Vector
size 128
Aggregation Methods
Aggregated Vector (of size 128)
Deep Learning Model
Fine-Tuning

Fig. 3. Illustrations of three embedding aggregation methods and fine-tuning of downstream tasks.

usually hundreds or thousands of statements in a Smali class that need to be embedded, the dimensionality of the learned embeddings should be reduced before deployment to downstream tasks while preserving their key information. In common practice for BERT-like models, the first state vector (of size 768) of the learned embedding is often used for this purpose. In our case, we add a third pre-training task, an Auto-Encoder, whose goal is to find a smaller more more efficient representation.

A smaller representation is necessary primarily because APKs consist of a significantly larger number of tokens compared to typical textual documents and code files. We provide a comparative analysis on token sequence length in a single file across three different data formats - textual documents (Paired CMU Book Summary [31]), code files (Devign [32]), and APKs (DexRay [28]), the number of BERT tokens in each dataset, denoted as [Mean]±[Deviation]. Specifically, the Paired CMU Book Summary has 1148.62±933.97 tokens, the Devign dataset contains 615.46±41 917.54 tokens, while the DexRay dataset contains a considerable 929.39K±11.50M tokens. This substantial quantity difference in tokens for APKs necessitates the effort to achieve as compact an embedding as possible for a given token sequence of Smali instructions.

AutoEncoder [33] is an artificial neural network that can learn efficient small-sized embedding of unlabeled data. It is typically used for dimensionality reduction by training the network to ignore the "noise". The basic architecture of an AutoEncoder usually consists of an Encoder for embedding learning and a Decoder for input regenerating as shown in the left box in Fig. 2. During the training process, the embedding is validated and refined by attempting to regenerate the input from the embedding, essentially trying to build the smallest complete representation of its input.

The encoder in our approach consists of two fully connected layers with 512 and 128 neurons, respectively. Symmetrically, the decoder consists of two fully connected layers with 128 and 512 neurons, respectively. The sequence embedding (with the size of 512 × 768) learned from BERT is both the input of the Encoder and the output target of the Decoder in the pre-training process. After comparative experiments, we opted for a hidden embedding with size 128 as the final representation of the original instruction sequence. We provide an analysis of different embedding sizes in Section VI-A. Compared with the raw BERT embedding, the dimension of the final sequence embedding is 3072 times smaller, from 512 × 768 to 128. Consequently, in Fig. 2, the size of *Vector 1*, which is the embedding of *Instruction Sequence 1* yielded by the Encoder, is 128.

C. Class-Level Prediction Model

In order to validate the effectiveness of learned DexBERT representation, we apply it to three class-level Android analysis tasks. To perform these class-level tasks, we need an efficient representation for each class, and we thus need a method to aggregate the embeddings of the instruction sequences of each class into one single embedding. We introduce this method in Section III.C.1. We then present the details of the prediction model in Section III.C.2.

1) Aggregation of Instruction Embedding: The representations learned from DexBERT are for instruction sequences. Usually, each Smali class consists of many methods, each of which contains a certain, often large, number of statements. The number of sequence vectors in each class is indeterminate, while the shape of the input to the class-level prediction model (neural network) is supposed to be fixed. Specifically, within each class, there are many sequence embedding vectors with a size of 128, which are expected to be combined into one single vector. Thus, embedding aggregation is required to obtain the final class-level representation.

We primarily have three reasons to solve the long instruction sequence problem by splitting a Smali class into snippets and aggregate the learned snippet embedding into one class embedding. First, Smali instructions are individual commands performing specific operations in the Android run-time environment. Although they often appear in sequences, each Smali instruction operates largely independently, not necessarily bearing the kind of interdependence seen in words within natural language sentences. Second, the class representation, which aggregates the instruction embeddings, retains a sense of the overall structure and function of the class while being context-aware. Last, while there might be some loss of context when splitting long sequences, we believe the trade-off between computational efficiency and a minor potential loss of context is justified. To give a quantified measure of this trade-off, let's consider the GPU memory required. Doubling the input length limit for a BERT model would necessitate four times the initial GPU memory, a demand that most standard devices cannot meet, particularly for even longer sequences. However, the high memory cost can be avoided without significant performance loss if we perform an average of 2.69 splits on long sequences in the adopted datasets, as shown in our experiments.

While it would be possible to leverage another step of representation learning to devise a strategy to aggregate several representations, we instead opt for less computationally expensive approaches. In order to adapt to the variability of vector numbers in different `Smali` classes, as shown in Fig. 3, we propose to aggregate these sequence embedding vectors of size 128 into one single vector of size 128 by performing simple element-wise average, element-wise addition, or random selection. Another method we investigate is a general approach to vector reshaping in computer vision: concatenation & bilinear resizing. Specifically, we first concatenate the embedding vectors into a long vector and resize it with the bilinear interpolation algorithm [34]. We investigate these four methods in Section V-E and show that these simple methods are effective.

2) Prediction Model: As shown in the right box in Fig. 3, after embedding aggregation, we finally obtain one embedding vector for each `Smali` class. The last step is to apply the learned embedding to downstream tasks. Typically, this can be done by feeding the embeddings to another independent neural network that can be trained to a specific task. In the original BERT, they only add one additional layer following the pre-trained model for each downstream task.

In our approach, we design a simple model architecture with only three fully connected layers of neural network as the task-specific model and freeze the parameters of the pre-trained DexBERT model when tuning for specific tasks. The computational cost in training downstream tasks is significantly decreased by only tuning parameters of the task-specific model. Specifically, the parameter number of the task-specific model is 10.4K, which is almost negligible compared with 459.35M of the pre-trained DexBERT model.

The input of the task-specific model is the aggregated vector of class embedding, as shown in Fig. 3. The task-specific model for malicious code localization predicts whether what a given class contains is malicious or not. Similarly, for defect detection, the task-specific model predicts if a given class contains defective code. For component type classification, the task-specific model predicts to which component type the given class belongs. The details of each task-specific setup are presented in Section IV-C.

IV. Study Design

In this section, we first overview the research questions to investigate in Section IV-A. Then, details about dataset and empirical setup are presented in Sections IV-B and IV-C. We provide evaluation results and answer the research questions in Section V.

A. Research Questions

In this paper, we consider the following six main research questions:

- **RQ1:** Can DexBERT accurately model Smali bytecode?
- **RQ2:** How effective is the DexBERT representation for the task of Malicious Code Localization?
- **RQ3:** How effective is the DexBERT representation for the task of Defect Detection?
- **RQ4:** How effective is the DexBERT representation for the task of Component Type Classification?
- **RQ5:** What are the impacts of different aggregation methods of instruction embeddings?
- **RQ6:** Can DexBERT work with subsets of instructions?

B. Dataset

In this work, we rely on four different datasets that we present in this section.

1) Dataset for Pre-Training: With DexBERT, we target a general-purpose representation for various Android analysis tasks. To obtain a representative sampling of the diverse landscape of android apps, we opted to leverage the dataset of a recent work in Android malware detection. One thousand apps—malware or benign—are randomly selected from the dataset used to evaluate DexRay [28], a work that collected more than 158 000 apps from the AndroZoo dataset [35].

Class de-duplication was performed in order to include as much diversity as possible without letting the total number of instructions explode. After removing duplicate classes, our selection of APKs results in over **35 million** `Smali` instructions, from which we obtained **556 million tokens**. This is comparable to the scale of BooksCorpus [36], one of the pre-training datasets used in the original BERT, which consists of 800 million tokens.

Despite the pre-training dataset of DexBERT being smaller than the original BERT, its sufficiency is supported by two factors. Firstly, `Smali`, being an assembly language, possesses a simpler structure and a significantly smaller set of tokens compared to natural languages and high-level programming languages, implying that a smaller dataset is enough to capture its essential features. Secondly, the efficiency of the dataset is evaluated based on DexBERT's performance in downstream tasks which use APKs from distinct sources than the pre-training dataset. The superior performance of DexBERT over baseline models in these tasks confirms the adequacy of the pre-training dataset.

Based on the `Smali` instructions in the dataset, we generated a WordPiece [37] vocabulary of 10 000 tokens for DexBERT, which is only one-third the size of the original BERT vocabulary. The WordPiece model employs a subword tokenization method to manage extensive vocabularies and handle rare and unknown words. It breaks down words into smaller units, effectively addressing out-of-vocabulary words.

2) Dataset for Malicious Code Localization: **RQ2** deals with Malicious Code Localization, *i.e.*, finding what part(s) of a given malware contains malicious code. At least two existing works have tackled this challenging problem for Android Malware and thus have acquired a suitable dataset with ground-truth labels. In Mystique [38], Meng et al. constructed a dataset of 10 000 auto-generated malware, with malicious/benign labels for each class. However, almost all of the code in these generated APKs is either malicious or from commonly used libraries (such as `android.support`), and thus may not be representative of existing apps, nor of the diversity of Android apps. More recently, in MKLDroid [13], Narayanan et al. randomly selected 3000 apps from the Mystique dataset and *piggybacked* the malicious parts into existing, real-world benign apps from Google Play, resulting in a dataset they named MYST. Although still a little far from the real-world scenario, the repackaged malware in the MYST dataset contains both malicious and benign classes, which can support a class-level malicious code localization task. We thus decide to rely on the MYST dataset to conduct our experiments related to RQ2. Note that, to the best of our knowledge, no fully labeled dataset of real-world malware exists for the task of malicious code localization for Android. Note also that in our work, we choose MKLDroid as a baseline work, enabling us to directly compare DexBERT against MKLDroid.

Despite the challenges in acquiring labeled real-world malware, we remained determined to evaluate DexBERT's performance in real-world scenarios. Ultimately, we succeeded in constructing a dataset that, albeit not extensive, contains labeled real-world malicious classes, thus broadening our evaluation scope. Specifically, we found 46 apps in the Difuzer [39] dataset, where the locations of a specific malicious behavior, namely the logic bomb, have been manually labeled. This allows us to obtain labels of malicious classes and thereby assess DexBERT's ability to localize malicious code in real-world applications. In addition, the authors of Difuzer provided more apps from their subsequent work, and we were able to successfully download and process 88 apks in total.

Given that each APK in the Difuzer dataset only has one class labeled as containing a logic bomb, and the malicious or benign nature of other classes is unknown, we are only able to utilize 88 malicious classes from these 88 real-world APKs for our extended evaluation. To facilitate a more comprehensive evaluation process, we constructed a dataset with additional APKs. The training set comprises three sources: 50 Difuzer APKs with logic bombs, 50 benign APKs from the DexRay dataset [28], and 100 APKs from MYST dataset to augment the dataset size. Please note that we selectively choose a portion of the benign classes at random to prevent a significant data imbalance, as the initial number of benign classes is much larger than that of malicious classes. As a result, we acquired 1929 benign classes and 425 malicious classes, including 50 from Difuzer APKs, for fine-tuning the classifier. The evaluation set, aimed at testing DexBERT on real-world APKs, consists solely of the remaining 38 APKs with logic bombs from Difuzer and 50 benign APKs from DexRay. We ended up with 95 benign classes (almost two per benign apks) and 38 malicious classes containing logic bombs. Please be aware that all the aforementioned designs aim to make the constructed data suitable for deep learning model training, while ensuring that there is no overlap between the training and evaluation sets.

3) Dataset for App Defect Detection: As another important class-level Android analysis task, app defect detection is the subject of **RQ3**. Dong et al. proposed `smali2vec` as a deep neural network based approach to detect application defects and released a dataset containing more than 92K Smali class files collected from ten Android app projects in over fifty versions. For the convenience of labeling, they collected these APKs from GitHub based on three project selection criteria: 1) the number of versions is greater than 20; 2) the package size is greater than 500 KB; and 3) a large number of commits and of contributors. The defective `Smali` files are located and labeled with Checkmarx [40], a widely used commercial static source code analysis tool. Finally, each `Smali` class in the dataset has a label indicating whether it is defective or not. We choose `smali2vec` as our dataset for app defect detection to enable comparison with their `smali2vec` approach built specifically for defect detection.

4) Dataset for Component Type Classification: To further evaluate the universality of DexBERT, we introduced a third class-level task called Component Type Classification. This task, distinctly different from the previous two, is designed to provide a comprehensive assessment of DexBERT's applicability across various scenarios. In the Android framework, four primary components exist, namely Activities, Services, Broadcast Receivers, and Content Providers. These are fundamental building blocks of an Android application and are declared in an application's manifest file (`AndroidManifest.xml`). Therefore, we can readily obtain labels for these four types of component classes and formulate a high-quality dataset for this task. Note that this task was designed solely to demonstrate the universality of DexBERT; It is selected due to the different nature of this task from other two downstream tasks and the ease and speed with which ground truth can be obtained. We randomly selected 1000 real-world APKs from the AndroZoo repository [35], from which we extracted 3406 component classes with accurate labels. We used 75% of this data for training and the remaining 25% for testing.

C. Empirical Setup

1) Pre-Training: Based on the typical BERT [1] design, we simplify the model architecture of DexBERT to a certain extent to reduce the computational cost. Indeed, while the dimension of intermediate layers in the position-wise feed-forward network was originally defined as $H \times 4$, where H is set to 768 by default, as mentioned in Section III.B.3, we reduce this dimension to $H \times 3$, *i.e.*, from 3072 to 2304. The number of hidden layers and heads in the multi-head attention layers are set to 8 instead of 12. With these simplifications, the number of floating-point operations (FLOPs, indicating the computational complexity of the model) is reduced by 43.9%, from 44.05G to 24.72G. Meanwhile, the number of model parameters is only decreased by 7.7%, from 497.45M to 459.35M. Thus we keep as many as possible of the model parameters while reducing the computational cost, with the goal of preserving the learning ability of the model as much as possible.

The batch size is set to 72, and the learning rate is set to $1e^{-4}$. Following the reference implementation[2] we leveraged, we select the Adam optimizer [41]. We adopt the Cross-Entropy loss function[3] for both the *masked words prediction* task and the *next sentence prediction* task, and the Mean Squared Error (MSE) loss function[4] for the Auto-Encoder task. Particularly,

[2] https://github.com/dhlee347/pytorchic-bert

[3] https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

[4] https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html

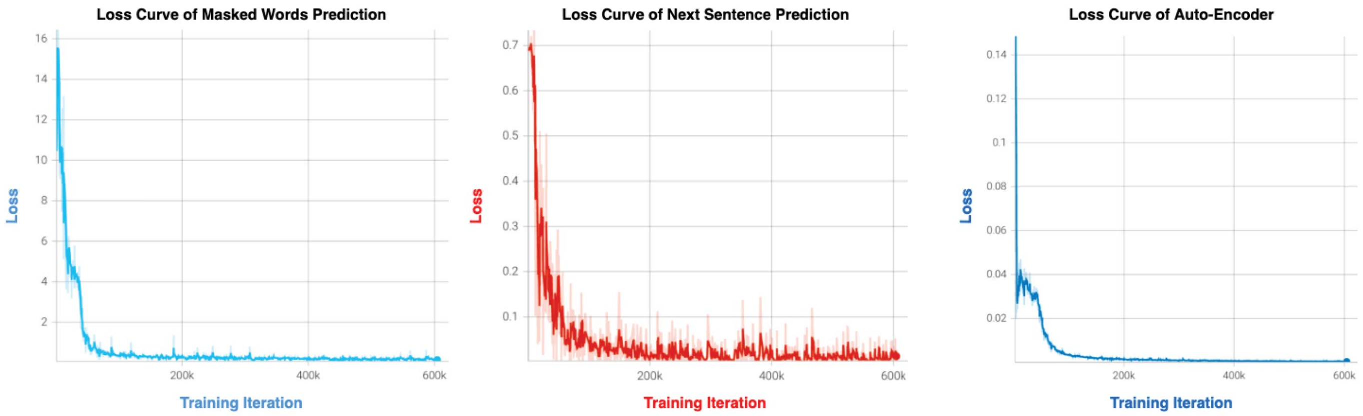


Fig. 4. Loss curves of three pre-training tasks. The X axis represents the training iteration index and Y axis represents the loss value.

the MSE Loss is a criterion that measures the squared L2 norm between each element in the input x and target y. Thus, the loss value of this task could also be regarded as an evaluation metric for the Auto-Encoder task.

The pre-training of DexBERT on 556 million tokens for 40 epochs took about 10 days on 2 Tesla V100 GPUs (each with 32G memory). However, it is important to note that DexBERT users do not need to pre-train the model from scratch. They can directly use the pre-trained model we provide for their own Android analysis tasks.

2) Malicious Code Localization: As discussed, malicious code localization is a difficult and under-explored problem. Therefore, there are few widely recognized approaches and benchmark datasets available. To the best of our knowledge, Narayanan et al. provided the first well-labeled dataset that comes close to real-world practical needs, to evaluate their multi-view context-aware approach MKLDroid [13] for malicious code localization.

In order to fairly compare with this baseline work, we follow the same validation strategy, *i.e.*, 2000 APKs for fine-tuning and the remaining 1000 APKs for evaluation. We use 3-fold cross validation to ensure the reliability of the results. The prediction model consists of 3 fully connected layers, of with 128, 64, and 32 neurons, respectively. The output layer consists of two neurons that are used to predict the probabilities for a class being either malicious or benign. We leverage the best aggregation method (*i.e.*, element-wise addition) found in **RQ5**. When fine-tuning for this task, we use the Cross-Entropy loss function and the Adam optimizer. With a batch size of 256, we train the prediction model for 40 epochs. For evaluation metrics, we adopt Precision, Recall, False Positive Rate (FPR), and False Negative Rate (FNR), following the same metrics as the baseline work, MKLDroid, for easy comparison. We further report F1 Scores as the overall metric.

3) App Defect Detection: Similar to malicious code localization, app defect detection is a relatively new challenge, only addressed by a small number of works in the literature. Particularly, Dong et al. proposed a DNN-based approach `smali2vec` [14] targeting Android applications and released a benchmark dataset containing more than 92K `Smali` class files. The classifier of `smali2vec` has 810 600 weight parameters across 10 layers, each of which has 300 neurons. In contrast, our model only comprises a total of 10 304 weight parameters spanning three simple layers with sizes of 128, 64, and 32.

In this work, we follow their *Within-Project Defect Prediction* (WPDP) strategy using 5-fold cross-validation to compare their approach with ours. They provide an AUC score for each project, and report their mean value as the final evaluation metric. In addition, we also report the weighted average score to cater to the significant size variations among projects. For this task, we adopt the same model architecture, aggregation method, loss function, and training strategy as for the malicious code localization task.

4) Component Type Classification: The majority of the empirical settings for the component type classification task are identical to those used in malicious code localization, with the exception of the number of neurons in the output layer of the prediction model. In this task, the classifier contains four neurons, instead of two, to output the probabilities for the four different component types.

V. Experimental Results

In this section, we present the evaluation results of DexBERT, and we answer our six research questions.

A. RQ1: Can DexBERT Accurately Model Smali Bytecode?

With this first RQ, we assess whether or not DexBERT is able to learn from `Smali` bytecode and build an accurate model of `Smali` bytecode as used in Android apps. To that end, we report the loss curves of the three pre-training tasks, presented in Sections III.B.2 and III.B.3. These loss curves are shown in Fig. 4, where the X axis represents the training iterations (i.e., batches).

Two elements allow us to confidently conclude that DexBERT indeed can learn an accurate model of `Smali` bytecode. First, the loss for all three pre-training tasks rapidly drops and is already very low after being fed just a small portion of our pre-training dataset, suggesting that our dataset

TABLE I
EVALUATION OF PRE-TRAINING TASKS. THE MASKED WORDS PREDICTION TASK IS EVALUATED ON 2 037 400 TOKENS AND THE NEXT SENTENCE PREDICTION TASK IS EVALUATED ON 101 870 INSTRUCTION PAIRS

Task	Accuracy	# of Samples
Masked Words Prediction	95.30%	2 037 400
Next Sentence Prediction	99.35%	101 870

is more than large enough for our purpose. Second, continuing the pre-training process results in even lower loss and does not generate random fluctuations, suggesting that the model learned is not contradicted by new pieces of `Smali` bytecode and indeed converges.

As mentioned in Section IV.C.1, the MSE loss could also be regarded as an evaluation metric, and its loss value in the third curve in Fig. 4 approaches zero late in the training process, indicating that the Auto-Encoder is able to reconstruct the given input vector of DexBERT embedding with minimal error. Therefore, the output vector of Encoder (*i.e.*, Hidden Embedding in Fig. 2) preserves the key information of the original DexBERT representation, which is required for the representation reconstruction by Decoder.

In order to further evaluate the other two pre-training tasks, we created an **evaluation set** containing 2 037 400 masked tokens for the masked words prediction task and 101 870 instruction pairs for next sentence prediction task. We calculate accuracy on the evaluation set as a metric to further evaluate the performance of DexBERT on these two tasks.

Given the initial imbalanced distribution of each token and the randomness of our selection process, the distribution of masked tokens was imbalanced. Common characters, strings, or variables such as slash "/" (15.47%), comma "," (6.32%), "v0" (1.83%), "object" (1.73%), and "lcom" (1.67%) were most frequently masked. Nevertheless, less common characters ($<$ 1.00%) still accounted for 42.69% of the masked tokens. As shown in Table I, with an accuracy of 95.30% on over two million predictions, we believe DexBERT's performance in the MLM task is robust. Regarding the NSP task, the distribution of positive and negative samples was well balanced; positive samples constituted 49.81% of the total samples, and negative samples made up the remaining 50.19%. DexBERT achieved a near-perfect accuracy of 99.35% on over 100K instruction pairs. This suggests that DexBERT was able to learn accurate features for the NSP task. The high accuracy of these two tasks demonstrates that the learned representations contain key features of the input instruction sequences leading to correct predictions.

RQ1 Answer: DexBERT can learn an accurate model of Smali bytecode.

B. RQ2: How Effective Is the DexBERT Representation for the Task of Malicious Code Localization?

In this section, we investigate the performance of DexBERT on the malicious code localization task and compare it with the MKLDroid baseline work [13] on their evaluation dataset. Following their experimental setup, we fine-tune DexBERT to output, for each class of a given app, a *maliciousness score*, or *m-score* for short. MKLDroid was evaluated with a beam search strategy, with a width of 10. In Table II, we show MKLDroid performance metrics as reported in the MKLDroid paper [13], followed by the performance metrics of DexBERT (Row *DexBERT-m*), also computed with a beam search of width 10.

TABLE II
PERFORMANCE OF MALICIOUS CODE LOCALIZATION ON THE MYST DATASET

Approach	F1 Score	Precision	Recall	FNR	FPR
MKLDroid	0.2488	0.1434	0.9400	0.0500	0.1700
smali2vec	0.9916	0.9880	0.9954	0.0046	0.0046
DexBERT-m	0.5749	0.4034	**1.0000**	**0.0000**	0.4847
DexBERT	**0.9981**	**0.9983**	0.9979	0.0021	**0.0006**

Additionally, we perform an experiment where we evaluate DexBERT without beam search, where each class is predicted as malicious if the m-score is above a threshold of 0.5 or benign otherwise. The performance metrics for this experience are reported in the last line of Table II. In effect, when evaluated in the same conditions as MKLDroid, DexBERT significantly outperforms MKLDroid with an F1 Score 0.9981 on the MYST dataset. Therefore, DexBERT does not need beam search at all and achieves excellent performance when classifying each class independently. Furthermore, we also include `smali2vec` [14] as an additional baseline, which, although it achieves fairly good performance, fails to outperform DexBERT.

As noted in Section IV.B.2, we expanded our evaluation of DexBERT on real-world Android applications. Employing our dataset constructed from Difuzer apps, i.e., Difuzer Extension dataset, DexBERT achieved a notable F1 Score of 0.9048 in identifying malicious classes within real-world APKs. Further, it achieved a commendable F1 Score of 0.9560 in predicting benign classes, thereby eliminating our concerns that data imbalance might negatively impact the evaluation.

RQ2 Answer: DexBERT significantly outperforms MKLDroid on the task of malicious code localization when evaluated in the same conditions. In addition, DexBERT can achieve vastly superior results when classifying each class independently. Furthermore, DexBERT also show its potential on localizing malicious classes within real-world Android apps.

C. RQ3: How Effective Is the DexBERT Representation for the Task of Defect Detection?

In this section, we investigate the performance of DexBERT on the task of app defect detection, and we compare it against the baseline work, `smali2vec` [14]. The performance of `smali2vec`[5] on 10 Android projects is shown in Table III, where the `# of classes` represents the number of `Smali` classes in each project.

[5] as reported in the smali2vec paper [14]

TABLE III
Performance of App Defect Detection

Project	AnkiDroid	BankDroid	BoardGame	Chess	ConnectBot	Andlytics	FBreader	K9Mail	Wikipedia	Yaaic	Average	Weighted Average
# of classes	14767	12372	1634	5005	3865	5305	9883	11857	18883	974	Score	AUC Score
smali2vec	0.7914	0.7967	**0.8887**	0.8481	**0.9516**	0.834	0.8932	0.7655	**0.8922**	**0.9371**	0.8598	0.8399
DexBERT	**0.9572**	**0.9363**	0.7691	**0.9125**	0.8517	**0.9248**	**0.9378**	**0.8674**	0.8587	0.8764	**0.8892**	**0.9032**

TABLE IV
Comparison of F1 Score Among Various BERT-Like Baselines for Four Component Classes

Method	Activity	Service	BroadcastReceiver	ContentProvider	Average
BERT	0.8272	0.7642	0.5673	0.9091	0.7669
CodeBERT	0.917	0.5381	0.8756	0.8468	0.7943
DexBERT(woPT)	0.7402	0.5850	0.7660	0.8947	0.7465
DexBERT	**0.9780**	**0.9117**	**0.9600**	**0.9756**	**0.9563**

Using our DexBERT representation, we fine-tune a model to predict the likelihood that a given class is defective.

As shown in Table III, DexBERT outperforms `smali2vec` on 6 out of 10 projects and achieves a weighted AUC score of 90.32%, which is a 6.33 percentage points improvement over `smali2vec`.

RQ3 Answer: DexBERT slightly outperforms `smali2vec` for the task of app defect detection.

D. RQ4: How Effective Is the DexBERT Representation for the Task of Component Type Classification?

In this section, we explore DexBERT's performance on the task of component type classification task, aiming to further examine its universal applicability across diverse application scenarios. We contrast DexBERT's performance against other BERT-like models, specifically BERT [1], CodeBERT [15], and DexBERT without pre-training. Similar to the settings for the malicious code localization task, we fine-tune a classifier to predict a given class's component type. Given that other BERT-like baselines lack an AutoEncoder module for further reduction of embedding dimensionality, we use the first state vector (size 768) of the embedding for all comparative experiments.

As Table IV illustrates, DexBERT excels in predicting all four types of component classes. On average, DexBERT's performance surpasses all baselines by a significant margin, exhibiting a roughly 20 percentage point increase in terms of F1 Score. This reiterates DexBERT's effectiveness in representing `Smali` instructions in Android and, validates the universality of DexBERT.

RQ4 Answer: DexBERT significantly outperforms baselines for the task of component type classification which differs from first two tasks that focused on security-related properties, demonstrating its versatility across various application scenarios.

E. RQ5: What Are the Impacts of Different Aggregation Methods of Instruction Embeddings?

TABLE V
Comparison of Different Aggregation Methods on Three Downstream Tasks: Malicious Code Localization (MCL), Defect Detection (DD), and Component Type Classification (CTC)

Method	MCL@F1 Score	DD@AUC Score	CTC@F1 Score
Addition	**0.9989**	**0.9064**	**0.9563**
Random	0.9982	0.8553	0.8898
Average	0.9916	0.8712	0.9442
Concat&Resize	0.9979	0.8508	0.7491

In this section, we investigate the impact of the four embedding aggregation methods (*i.e.*, element-wise addition, random selection, averaging and concatenation & bilinear resizing, cf. Section III.C.1). These techniques were initially leveraged to aggregate token embeddings in BERT [1] and other deep learning approaches [28]. From our point of view, the output features from BERT and the AutoEncoder of DexBERT are essentially similar in nature. Each state vector of BERT embedding is a high-level abstract feature of the corresponding token. Similarly, each output vector of Auto-Encoder is a high-level abstract feature vector of the corresponding token sequence. Therefore, if it's plausible to aggregate token embeddings by addition or the other three techniques, it should also be plausible to aggregate sequence embeddings in a similar manner.

We conduct comparative experiments based on three downstream tasks to evaluate to what extent DexBERT is sensitive to the aggregation method. As shown in Table V, on the malicious code localization task, performance metrics for all four methods are very close, with no significant differences among them. For the other two downstream tasks, we note that the differences are more significant than for the task of malicious code localization. Despite all four aggregation methods yielding an acceptable performance, element-wise addition is the best performer, achieving the highest metric scores on both tasks.

RQ5 Answer: All four proposed aggregation methods are effective on all the three downstream tasks. Element-wise addition achieves the best performance on both tasks.

F. RQ6: Can DexBERT Work With Subsets of Instructions?

In the previous RQs, we demonstrated the effectiveness of DexBERT when using the entire `Smali` bytecode. Representing `Smali` bytecode with DexBERT can be computationally expensive, given the very large number of instructions an app (or even a class) can contain. With this RQ, we investigate the ability of DexBERT to work with subsets of instructions, hence reducing the number of pieces of code to represent and reducing the need for aggregation.

We postulate that API invocations are the instructions that carry the most semantics information, and thus conduct an experiment where we pre-filter the flow of `Smali` bytecode to keep only API calls. Based on the statistics on our pre-training

TABLE VI
COMPARATIVE ANALYSIS OF FULL INSTRUCTIONS VS API CALLS FOR MALICIOUS CODE LOCALIZATION (MCL) AND COMPONENT TYPE CLASSIFICATION (CTC). ''AVG TIME'' MEANS THE AVERAGE INFERENCE TIME PER CLASS

Method	MCL@F1 Score	CTC@F1 Score	Avg Time
Full Instructions	**0.9981**	**0.9563**	0.00768s
API Call	0.9932	0.8779	**0.00073s**

dataset, API instructions constitute approximately 17.27% of the total instructions. We re-use the pre-trained model built with the complete flow of instructions (cf. RQ1), but we fine-tune a dedicated model with filtered instructions only. Since many classes in some projects in the dataset for defect detection do not have API calls (some concrete examples are included in our replication package), which would result in empty representations, we only consider the tasks of malicious code localization and Component Type Classification.

As shown in Table VI, while the performance of DexBERT is slightly higher with all instructions, DexBERT still performs well with API calls solely. Computationally, however, working with API calls is one order of magnitude faster. As for the total execution time, we take the evaluation of malicious code localization task as an example here. With full instructions, we require approximately 1.9 hours to generate the DexBERT features for all 911,724 classes. Conversely, with API calls, we only need about 11 minutes to generate all feature vectors.

RQ6 Answer: When compatible with the downstream task, DexBERT is also fairly effective and fast when considering API calls only.

VI. DISCUSSION

In this section, we begin with an ablation study examining the impact of DexBERT's embedding size on downstream tasks, and an ablation study assessing the effectiveness of the two pre-training tasks. Following that, we compare the performance of various BERT-like baselines across three different downstream tasks. Then, we share some insights about the proposed DexBERT for Android representation. Next, we discuss some potential threats to validity of the proposed approach. Finally, we introduce some future works which are worth studying next.

A. Ablation Study on DexBERT Embedding Size

As detailed in Section III.B.3, Android application scenarios require a smaller embedding due to the considerably larger token quantities compared to typical textual documents and code files. To find a reasonable trade-off between model computation cost and performance, we conducted an ablation study exploring the impact of DexBERT embedding size on the three downstream tasks. The experiments contain three different sizes for the hidden embedding of the AutoEncoder, specifically 256, 128, and 64. Additionally, we evaluated the performance by directly utilizing the first state vector of the raw DexBERT embedding, which has a size of 768, without applying any dimension reduction from the AutoEncoder.

TABLE VII
ABLATION STUDY ON THE IMPACT OF DEXBERT EMBEDDING SIZE

Size	MCL@F1 Score	DD@AUC Score	DD@F1 Score	CTC@F1 Score
768	0.9999	0.9699	0.8887	0.9563
256	0.9995	0.9336	0.8029	0.9246
128	0.9981	0.9032	0.7542	0.9202
64	0.9813	0.8472	0.6693	0.9007

TABLE VIII
COMPARISON OF F1 SCORES AMONG VARIOUS BERT-LIKE BASELINES FOR THREE TASKS: MALICIOUS CODE LOCALIZATION (MCL), DEFECT DETECTION (DD), AND COMPONENT TYPE CLASSIFICATION (CTC). DEXBERT(WOPT) INDICATES DEXBERT WITHOUT PRE-TRAINING

Models	MCL	DD	CC
BERT	0.9182	0.66851	0.7669
CodeBERT	0.9985	0.64775	0.7943
DexBERT(wo-PreT)	0.9961	0.74381	0.3028
DexBERT	**0.9999**	**0.8725**	**0.9563**

TABLE IX
COMPARISON OF F1 SCORES ON THREE DOWNSTREAM TASKS BASED ON DIFFERENT PRE-TRAINING TASK DESIGNS FOR: MALICIOUS CODE LOCALIZATION (MCL), DEFECT DETECTION (DD), AND COMPONENT TYPE CLASSIFICATION (CTC)

Pre-Training Designs	MCL	DD	CTC
Only MLM	0.9987	0.8055	0.8827
Only NSP	0.6547	0.5331	0.5491
MLM & NSP	**0.9999**	**0.8725**	**0.9563**

Table VII reveals that in the task of Malicious Code Localization, a decrease in vector size does not lead to a significant loss in the performance, until the size is reduced to 128. Hence, we concluded that 128 is the optimal size for this task.

As for the tasks of Defect Detection and Component Type Classification, the experimental results demonstrate that a larger embedding size resulted in a considerable improvement in performance. However, a size of 128 also offered a solid trade-off for these two tasks, supporting satisfactory performance with AUC score exceeding 0.9. Please be aware that the choice to use the AUC score for defect detection was made in order to maintain consistency with the metric employed by the primary baseline for this task, namely, `smali2vec` [14]. To be consistent with the other two tasks, we have also included the F1 Score for this task in Table VII.

B. Ablation Study on Pre-Training Tasks

To better understand the pre-training process, we conducted an ablation to confirm the necessity and effectiveness of the two pre-training designs, i.e., MLM and NSP, on the final improvement of the model.

In models like BERT and DexBERT, multi-task learning with MLM and NSP is designed to generate universal features for a variety of tasks. Removing either task diminishes the model's representational power. As demonstrated in Table IX, while MLM alone can achieve relatively good performance, the combination of both pre-training tasks significantly improves the model's performance, reinforcing their mutual importance for capturing the Smali bytecode structure and semantics effectively. The results on all three downstream tasks, especially on defect detection and component type classification demonstrate the importance of both MLM and NSP pre-training tasks.

C. Comparative Study With Other BERT-Like Baselines

To better understand the necessity and effectiveness of the pre-training process on Smali code, in this section, we conduct a comparative study to assess the performance of existing BERT-like models that can be directly applied to all three Android downstream tasks without any technical barriers. Specifically, the baselines include BERT [1], CodeBERT [15], and DexBERT without pre-training. With the same reason in Section V-D, we use the first state vector (size 768) of the embedding for all comparative experiments.

The outcomes are shown in Table VIII. Interestingly, each baseline model performed remarkably well for Malicious Code Localization. This can be attributed to the fact that the dataset is artificially generated by inserting malicious code into real-world apps, resulting in a clear separation between positive and negative samples, making them easy to learn from. However, the pre-trained DexBERT model outperformed the baselines with an impressive F1 Score of 0.9999, approaching perfection.

For the other two tasks, Defect Detection and Component Type Classification, where datasets were collected from real-world APKs, the pre-trained DexBERT clearly surpassed BERT and CodeBERT by approximately 20 percentage points on both tasks. Furthermore, the performance of DexBERT without pre-training exhibited low performance and instability across the three tasks, which was somewhat expected as it lacked prior knowledge before being fine-tuned on downstream tasks. Overall, the comparative results shown in Table VIII clearly demonstrate the necessity and effectiveness of DexBERT pre-training process on Smali code.

D. Insights

In this work, we find that the popular NLP representation learning model, BERT can be used for Android bytecode without much modification by regarding disassembled Smali instructions as natural language sentences. While it had already been shown that BERT-like models could be used for source code, our work shows it can also work directly with raw apps in the absence of original source code.

Still, there are some gaps between natural language and Smali code to mitigate. In particular, NLP problems and Android analysis problems have significantly different application scenarios. NLP problems are usually at the text snippet level, where the base unit is a (short) paragraph (*e.g.*, sentence translation or text classification). However, in Software Engineering and Security for Android applications, there are problems both at the class-level and the whole app-level, *i.e.*, ranging from a few instructions to millions of instructions. Therefore, embedding aggregation is required when applying instruction embeddings to Android analysis problems.

Besides, while this work is only evaluated on class-level tasks, it is expected to work at lower or higher levels. There are no significant technical barriers for lower-level tasks (*i.e.*, method-level or statement-level) except for the absence of well-labeled datasets. For higher-level tasks (*i.e.*, app-level), further embedding aggregation would be required to support whole-application tasks.

E. Threats to Validity

Our experiments and conclusions may face threats to validity. First, the MYST dataset used for malicious code localization is artificially created and, therefore may not be representative of the real-world landscape of apps. To mitigate this concern, we expanded our evaluation to include real-world apps from Difuzer [39]. Moreover, the utility of our malicious code localization model in real-world scenarios could be further assessed by extending its application to a wider variety of malicious behaviours beyond logic bombs. This extension could be an avenue for future work, as it would necessitate significant manual analysis efforts.

Second, the dataset for Android app defect detection was created several years ago. Android applications are constantly evolving over time. How well DexBERT performs on today's apps requires further validation. Therefore, it may be necessary to create new datasets and conduct more comprehensive evaluations on Android app defect detection.

F. Future Work

The proposed DexBERT is validated on three class-level Android analysis tasks. An important aspect would be to extend the range of tasks DexBERT is evaluated on. We identified several tasks (such as malware detection, app clone detection, repackaging identification, *etc.*) that are of interest to the research community and that could benefit from our approach.

Besides, we showed that DexBERT representation may not always need the complete flow of Smali instructions. However, we investigated only one filtering criterion. Other filtering approaches could be investigated to refine potential trade-offs between computational cost and effectiveness.

Finally, the evaluation datasets for tasks such as malicious code localization and defect detection can be further enhanced by including more recent applications. Given the substantial efforts required to process and construct the new datasets, we plan to undertake this enhancement in a separate study in the future.

VII. Related Work

This study lies at the intersection of the fields of Representation Learning and of Android app analysis.

A. Representation Learning

Recent successes in deep learning have attracted increased interest in applying deep learning techniques to learn representations of programming artifacts for a variety of software engineering tasks [2], [15], [42], [43].

1) Code Representation: Code representation approaches aim to represent source code as feature vectors that contain the semantics and syntactic information of the source code. In general, code representations can be mainly categorized into sequence-based, tree-based, and graph-based representations. The tasks that rely on sequence-based representations consider source code as plain text and use traditional token-based methods to capture lexical information, such as clone detection [44], vulnerability detection [45], and code review [46]. Tree-based representations capture features of source code by traversing the AST of the source code. Code2Vec [2] proposes a path-attention model to aggregate the set of AST paths into a vector. TBCNN [47] learns code representations that capture structural information in the AST. Tree-LSTM [48] employs LSTM in learning the network topology of the input tree structure of AST. Graph-based representation approaches [49], [50] represent code as graphs that are associated with programs, such as control flow graph (CFG), control dependency graph (CDG), and data dependency graph (DDG).

Inspired by the recent success of transformer-based language models like BERT [1] and RoBERTa [51] in Natural Language Processing, Feng et al. [15] proposed CodeBERT, which is pre-trained both on programming languages and natural languages. Guo et al. [52] proposed GraphCodeBERT to advance CodeBERT by additionally considering data flow information in pre-training.

2) Android App Representation: Android app representations aim to represent an Android app into feature vectors for various tasks such as malware detection [27] and clone detection [53]. Many works [5], [7], [8], [9], [10], [54], [55] relied on reverse engineering to extract information (features) from APKs and feed the extracted features into traditional ML-based and DL-based approaches to obtain Android representations [56]. Static features such as permissions, API calls, and control flow graphs are widely used in prior works [6], [10], [54], [57], [58], [59], [60] to generate Android representations. There are several approaches where representation is based on dynamic features. For instance, several Android malware detectors [5], [7], [8], [9], [55] leveraged system calls traces.

The aforementioned features can be represented in different forms: the vectorized representation and the graph-based representation. Features such as permissions or API calls [57], [58], [59], [60], [61], [62], [63], [64], raw [65], [66] or processed [67] opcode sequences, and dynamic behaviors [68], [69] are mainly represented as vectors. Other graph-based features such as control flow graphs [70], [71] and data flow graphs [72] can be directly fed to DL models (*e.g.*, Graph Convolutional Network [73], [74]) or embedded into vectors by graph embedding techniques (*e.g.*, Graph2vec [75]).

B. Android Analysis

Android app analysis tasks can be conducted at different levels, such as APK-level (*e.g.*, Android Malware Detection [27], [28], [76], [77], Android Repackaging Identification [78], [79], [80]), class-level or method-level (*e.g.*, Malicious Code Localization, Android App Defect Detection).

1) Android Malicious Code Localization: HsoMiner [81] is an approach to discover HSO (Hidden Sensitive Operation) activities (*e.g.*, stealing user's privacy). Li et al. proposed a tool called HookRanker [82] to automatically localize the malicious packages from piggybacked Android apps based on how malware behavior code is triggered. MKLDroid [13], proposed by Narayanan et al., could be regarded as the first real malicious code localization approach. They consider multiple views of Android apps in a unified framework to detect malware. MKLDroid assigns m-scores to every class, and the classes with the highest m-scores are considered malicious. Ma et al. proposed a deep learning based method called Droidetec [11] for malware detection and malicious code localization by modeling an Android app as a natural language sequence. MKLDroid and Doidetec could achieve a reasonable recall of malicious segments (classes or methods) by sacrificing precision. Recently, Wu et al. proposed a Graph Neural Network based approach [83] for Android malware detection and malicious code localization. Despite the help of manual checks, the obtained accuracy is still far from perfect. Among these related works above, only MKLDroid [13] provides its replication package.

2) Android App Defect Detection: A software defect is an error or a bug caused by a programmer during the software design and development process. Early approaches for software defect detection [84], [85], [86], [87] were not easily adaptable for Android applications.

Initial attempts at Android defect prediction [88], [89] focused on extracting code and process metrics from mobile applications. Similarly, object-oriented metrics [26], [90] were employed to build defect prediction models. However, the feature engineering efforts in these approaches limited the verification of their methods' effectiveness to a small number of applications. Additionally, they relied on specific sets of code and process metrics, which might not be universally applicable to other APKs. To address this limitation, Dong et al. proposed `smali2vec` [14], which automatically extracts features of `Smali` instructions and inputs them into a deep learning model to identify defective classes. They also provided a benchmark dataset for the community to advance the field of Android defect detection.

Meanwhile, Just-in-Time (JIT) defect prediction [91], [92], [93] at the commit level has been developed, offering timely feedback for developers to detect defects early. However, class-level prediction methods remain necessary, as they help developers and testers prioritize their efforts by identifying the most defect-prone classes. Additionally, class-level prediction can be useful when a project has a low frequency of commits or uses a different version control system that makes commit-level prediction difficult. In this work, we focus on class-level Android defect detection.

VIII. CONCLUSION

We propose a pre-trained representation learning model named DexBERT, aiming at solving various fine-grained Android analysis problems. Based on AutoEncoder, we design an aggregation method to overcome the input length limitation problem existing in the original BERT applications. Freezing parameters of the pre-trained DexBERT model, the learned representation is able to be directly used on various class-level downstream tasks. Comprehensive experimental results demonstrate its effectiveness on malicious code localization, Android application defect detection and component type classification, compared to baseline methods.

IX. Data Availability

All artifacts of this study are available at: https://github.com/Trustworthy-Software/DexBERT

References

[1] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 1–29, 2019.

[3] S. Mani, A. Sankaran, and R. Aralikatte, "DeepTriage: Exploring the effectiveness of deep learning for bug triaging," in *Proc. ACM India Joint Int. Conf. Data Sci. Manage. Data*, 2019, pp. 171–179.

[4] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2019, pp. 1–12.

[5] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Detecting Android malware using sequences of system calls," in *Proc. 3rd Int. Workshop Softw. Develop. Lifecycle Mob.*, 2015, pp. 13–20.

[6] S. Hou, A. Saas, Y. Ye, and L. Chen, "DroidDelver: An android malware detection system using deep belief network based on API call blocks," in *Int. Conf. Web-Age Inf. Manag.* Nanchang, China: Springer, 2016, pp. 54–66.

[7] L. Singh and M. Hofmann, "Dynamic behavior analysis of android applications for malware detection," in *Proc. Int. Conf. Intell. Commun. Comput. Techn. (ICCT)*. Piscataway, NJ, USA: IEEE, 2017, pp. 1–7.

[8] X. Xiao, Z. Wang, Q. Li, S. Xia, and Y. Jiang, "Back-propagation neural network on Markov chains from system call sequences: A new approach for detecting Android malware with system call sequences," *IET Inf. Secur.*, vol. 11, no. 1, pp. 8–15, 2017.

[9] X. Xiao, X. Xiao, Y. Jiang, X. Liu, and R. Ye, "Identifying Android malware with system call co-occurrence matrices," *Trans. Emerg. Telecommun. Technol.*, vol. 27, no. 5, pp. 675–684, 2016.

[10] Z. Xu, K. Ren, S. Qin, and F. Craciun, "CDGDroid: Android malware detection based on deep learning using CFG and DFG," in *Int. Conf. Formal Eng. Methods*. Gold Coast, Australia: Springer, 2018, pp. 177–193.

[11] Z. Ma, H. Ge, Z. Wang, Y. Liu, and X. Liu, "Droidetec: Android malware detection and malicious code localization through deep learning," 2020, *arXiv:2002.03594*.

[12] R. S. Arslan, "AndroAnalyzer: Android malicious software detection based on deep learning," *PeerJ Comput. Sci.*, vol. 7, 2021, Art. no. e533.

[13] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "A multi-view context-aware approach to android malware detection and malicious code localization," *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1222–1274, 2018.

[14] F. Dong, J. Wang, Q. Li, G. Xu, and S. Zhang, "Defect prediction in android binary executables using deep neural network," *Wireless Pers. Commun.*, vol. 102, no. 3, pp. 2261–2285, 2018.

[15] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[16] A. Narayanan, C. Soh, L. Chen, Y. Liu, and L. Wang, "Apk2vec: Semi-supervised multi-view representation learning for profiling android applications," in *Proc. IEEE Int. Conf. Data Min. (ICDM)*. Piscataway, NJ, USA: IEEE, 2018, pp. 357–366.

[17] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.* Piscataway, NJ, USA: IEEE, 2012, pp. 171–180.

[18] M. Singh and V. Sharma, "Detection of file level clone for high level cloning," *Procedia Comput. Sci.*, vol. 57, pp. 915–922, 2015.

[19] C. Tantithamthavorn, S. L. Abebe, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of IR-based classifier configuration on the performance and the effort of method-level bug localization," *Inf. Softw. Technol.*, vol. 102, pp. 160–174, 2018.

[20] W. Zhang, Z. Li, Q. Wang, and J. Li, "Finelocator: A novel approach to method-level fine-grained bug localization by query expansion," *Inf. Softw. Technol.*, vol. 110, pp. 121–135, 2019.

[21] V. Frick, "Understanding software changes: Extracting, classifying, and presenting fine-grained source code changes," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.: Companion Proc.*, 2020, pp. 226–229.

[22] E. Mashhadi and H. Hemmati, "Applying codeBERT for automated program repair of Java simple bugs," in *Proc. IEEE/ACM 18th Int. Conf. Min. Softw. Repositories* (*MSR*), 2021, pp. 505–509.

[23] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codeBERT model," *Appl. Sci.*, vol. 11, no. 11, pp. 1–20, 2021. Accessed: May 23, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/11/4793

[24] X. Yuan, G. Lin, Y. Tai, and J. Zhang, "Deep neural embedding for software vulnerability discovery: Comparison and optimization," *Secur. Commun. Netw.*, vol. 2022, pp. 1–12, 2022.

[25] S. Fujimori, M. Harmanani, O. Siddiqui, and L. Zhang, "Using deep learning to localize errors in student code submissions," in *Proc. 53rd ACM Tech. Symp. Comput. Sci. Educ. V. 2*, 2022, pp. 1077–1077.

[26] R. Malhotra, "An empirical framework for defect prediction using machine learning techniques with Android software," *Appl. Soft Comput.*, vol. 49, pp. 1034–1050, 2016.

[27] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.

[28] N. Daoudi, J. Samhi, A. K. Kabore, K. Allix, T. F. Bissyandé, and J. Klein, "DexRay: A simple, yet effective deep learning approach to Android malware detection based on image representation of bytecode," in *Proc. Int. Workshop Deployable Mach. Learn. Secur. Defense*. Springer, 2021, pp. 81–106.

[29] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, and H. Haghighi, "SLDeep: Statement-level software defect prediction using deep-learning model on static code features," *Expert Syst. Appl.*, vol. 147, 2020, Art. no. 113156.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[31] D. Bamman and N. A. Smith, "New alignment methods for discriminative book summarization," 2013, *arXiv:1305.1319*.

[32] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–11.

[33] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[34] S. Fadnavis, "Image interpolation techniques in digital image processing: An overview," *Int. J. Eng. Res. Appl.*, vol. 4, no. 10, pp. 70–73, 2014.

[35] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. IEEE/ACM 13th Work. Conf. Min. Softw. Repositories (MSR)*. Piscataway, NJ, USA: IEEE, 2016, pp. 468–471.

[36] Y. Zhu et al., "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 19–27.

[37] Y. Wu et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.

[38] G. Meng et al., "MYSTIQUE: Evolving Android malware for auditing anti-malware tools," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 365–376.

[39] J. Samhi, L. Li, T. F. Bissyandé, and J. Klein, "Difuzer: Uncovering suspicious hidden sensitive operations in Android apps," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 723–735.

[40] [Online]. Available: https://checkmarx.com/, Aug. 2022.

[41] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[42] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv. (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[43] M. D. Ernst, "Natural language is a programming language: Applying natural language processing to software development," in *Proc. 2nd Summit Adv. Program. Lang. (SNAPL)*. Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017, pp. 1–14.

[44] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "FCCA: Hybrid code representation for functional clone detection using attention networks," *IEEE Trans. Rel.*, vol. 70, no. 1, pp. 304–318, Mar. 2020.

[45] A. Xu, T. Dai, H. Chen, Z. Ming, and W. Li, "Vulnerability detection for source code using contextual LSTM," in *Proc. 5th Int. Conf. Syst. Inform. (ICSAI)*. Piscataway, NJ, USA: IEEE, 2018, pp. 1225–1230.

[46] J. K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "Core: Automating review recommendation for code changes," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*. Piscataway, NJ, USA: IEEE, 2020, pp. 284–295.

[47] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proc. 30th AAAI Conf. Artif. Intell.*, 2016, pp. 1–7.

[48] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," 2015, *arXiv:1503.00075*.

[49] Y. Wan et al., "Multi-modal attention network learning for semantic source code retrieval," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*. Piscataway, NJ, USA: IEEE, 2019, pp. 13–25.

[50] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*. Piscataway, NJ, USA: IEEE, 2020, pp. 261–271.

[51] Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[52] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," 2021, *arXiv:abs/2009.08366*.

[53] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 175–186.

[54] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for Android malware detection using deep learning," *Digit. Invest.*, vol. 24, pp. S48–S59, 2018.

[55] T. Bhatia and R. Kaushal, "Malware detection in Android based on dynamic analysis," in *Proc. Int. Conf. Cyber Secur. Protection Digit. Services*. Piscataway, NJ, USA: IEEE, 2017, pp. 1–6.

[56] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of Android malware detection with deep neural models," *ACM Comput. Surv. (CSUR)*, vol. 53, no. 6, pp. 1–36, 2020.

[57] D. Li, Z. Wang, and Y. Xue, "Fine-grained Android malware detection based on deep learning," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*. Piscataway, NJ, USA: IEEE, 2018, pp. 1–2.

[58] J. Booz, J. McGiff, W. G. Hatcher, W. Yu, J. Nguyen, and C. Lu, "Tuning deep learning performance for Android malware detection," in *Proc. 19th IEEE/ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput. (SNPD)*. Piscataway, NJ, USA: IEEE, 2018, pp. 140–145.

[59] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *Proc. Eur. Symp. Res. Comput. Secur*. Oslo, Norway: Springer, 2017, pp. 62–79.

[60] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *Proc. 28th USENIX Secur. Symp. (USENIX Security)*, 2019, pp. 729–746.

[61] A. Naway and Y. Li, "Using deep neural network for Android malware detection," *Int. J. Adv. Stud. Comput., Sci. Eng.*, vol. 7, no. 12, pp. 9–18, 2018.

[62] A. Naway and Y. Li, "Android malware detection using autoencoder," 2019, *arXiv:1901.07315*.

[63] W. Y. Lee, J. Saxe, and R. Harang, "SeqDroid: Obfuscated Android malware detection using stacked convolutional and recurrent neural networks," in *Deep Learning Applications for Cyber Security*. Cham, Switzerland: Springer, 2019, pp. 197–210.

[64] N. He, T. Wang, P. Chen, H. Yan, and Z. Jin, "An android malware detection method based on deep AutoEncoder," in *Proc. Artif. Intell. Cloud Comput. Conf.*, 2018, pp. 88–93.

[65] N. McLaughlin et al., "Deep Android malware detection," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 301–308.

[66] Q. Jerome, K. Allix, R. State, and T. Engel, "Using opcode-sequences to detect malicious Android applications," in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2014, pp. 914–919.

[67] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. Le Traon, "Empirical assessment of machine learning-based malware detectors for Android," *Empirical Softw. Eng.*, vol. 21, no. 1, pp. 183–211, Feb. 2016. [Online]. Available: https://doi.org/10.1007/s10664-014-9352-6

[68] R. Vinayakumar, K. Soman, P. Poornachandran, and S. Sachin Kumar, "Detecting Android malware using long short-term memory (LSTM)," *J. Intell. Fuzzy Syst.*, vol. 34, no. 3, pp. 1277–1288, 2018.

[69] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-Sec: Deep learning in Android malware detection," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 371–372.

[70] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. A. Porras, "DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications," in *Proc. ESORICS*, 2014, pp. 163–182.

[71] M. A. Atici, S. Sağiroğlu, and I. A. Dogru, "Android malware analysis approach based on control flow graphs and machine learning algorithms," in *Proc. 4th Int. Symp. Digit. Forensic Secur. (ISDFS)*, 2016, pp. 26–31.

[72] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1–32.

[73] J. Bruna, W. Zaremba, A. D. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," 2014, *arXiv:abs/1312.6203*.

[74] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017, *arXiv:abs/1609.02907*.

[75] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," 2017, *arXiv:1707.05005*.

[76] T. Hsien-De Huang and H.-Y. Kao, "R2-D2: ColoR-inspired convolutional neuRal network (CNN)-based AndroiD malware Detections," in *Proc. IEEE Int. Conf. Big Data (Big Data)*. Piscataway, NJ, USA: IEEE, 2018, pp. 2633–2642.

[77] T. Sun, N. Daoudi, K. Allix, and T. F. Bissyandé, "Android malware detection: Looking beyond Dalvik bytecode," in *Proc. 36th IEEE/ACM Int. Conf. Autom. Softw. Eng. Workshops (ASEW)*. Piscataway, NJ, USA: IEEE, 2021, pp. 34–39.

[78] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged Android applications," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 56–65.

[79] L. Li, T. F. Bissyandé, and J. Klein, "SimiDroid: Identifying and explaining similarities in Android apps," in *Proc. IEEE Trustcom/BigDataSE/ICESS*. Piscataway, NJ, USA: IEEE, 2017, pp. 136–143.

[80] S. Singh, K. Chaturvedy, and B. Mishra, "Multi-view learning for repackaged malware detection," in *Proc.16th Int. Conf. Availability, Rel. Secur.*, 2021, pp. 1–9.

[81] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps," in *Proc. NDSS*, 2017, pp. 1–15.

[82] L. Li et al., "On locating malicious code in piggybacked Android apps," *J. Comput. Sci. Technol.*, vol. 32, no. 6, pp. 1108–1124, 2017.

[83] Q. Wu, P. Sun, X. Hong, X. Zhu, and B. Liu, "An Android malware detection and malicious code location method based on graph neural network," in *Proc. MLMI*, 2021, pp. 50–56.

[84] D. Bowes, T. Hall, and D. Gray, "DConfusion: A technique to allow cross study performance evaluation of fault prediction studies," *Autom. Softw. Eng.*, vol. 21, no. 2, pp. 287–313, 2014.

[85] H. Perl et al., "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 426–437.

[86] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.

[87] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. ICSE*. Piscataway, NJ, USA: IEEE, 2016, pp. 297–308.

[88] A. Kaur, K. Kaur, and H. Kaur, "An investigation of the accuracy of code and process metrics for defect prediction of mobile applications," in *Proc. 4th Int. Conf. Rel., Infocom Technol. Optim. (ICRITO) (Trends Future Directions)*. Piscataway, NJ, USA: IEEE, 2015, pp. 1–6.

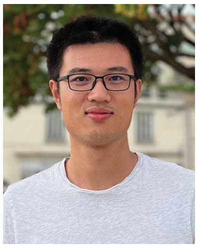
[89] A. Kaur, K. Kaur, and H. Kaur, "Application of machine learning on process metrics for defect prediction in mobile application," in *Proc. Inf. Syst. Des. Intell. Appl.: Proc. 3rd Int. Conf. INDIA*, vol. 1. New Delhi, India: Springer, 2016, pp. 81–98.

[90] M. Y. Ricky, F. Purnomo, and B. Yulianto, "Mobile application software defect prediction," in *Proc. IEEE Symp. Service-Oriented Syst. Eng. (SOSE)*. Piscataway, NJ, USA: IEEE, 2016, pp. 307–313.

[91] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 82–101, 2020.

[92] M. Yan, X. Xia, Y. Fan, D. Lo, A. E. Hassan, and X. Zhang, "Effort-aware just-in-time defect identification in practice: A case study at Alibaba," in *Proc. 28th ACM Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1308–1319.

[93] K. Zhao, Z. Xu, M. Yan, Y. Tang, M. Fan, and G. Catolino, “Just-in-time defect prediction for Android apps via imbalanced deep learning model,” in *Proc. 36th Annu. ACM Symp. Appl. Comput.*, 2021, pp. 1447–1454.



Tiezhu Sun received the master’s degree from Shandong University. He was mainly engaged in the research of deep learning and computer vision. He is currently working toward the Ph.D. degree with the University of Luxembourg. His work mainly involves software engineering and artificial intelligence, especially Android representation and deep learning algorithms. His works were published at major conferences such as International Joint Conferences on Artificial Intelligence, AAAI Conference on Artificial Intelligence, and International Conference on Pattern Recognition.

Kevin Allix received the Ph.D. degree, in 2015, from the University of Luxembourg, and has been a Postdoctoral for over five years with the SnT Interdisciplinary Centre at the University of Luxembourg. He is an Associate Professor with Centrale-Supélec, France. He conducts research on Android malware detection, machine-learning for security, software engineering, and natural language processing. He also held operational positions in network, system, and security engineering, and was an IT Security and Risk Analyst for a private bank.

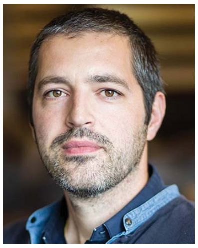
Kisub Kim received the Ph.D. degree in computer science from the University of Luxembourg in 2021. He is a Research Scientist with Singapore Management University. His work is mainly related to software engineering, broadly source code analysis and, specifically, code search, bug localization, code review, and code representation. His works were presented at major conferences such as International Conference on Software Engineering and published in top journals such as *Empirical Software Engineering*. He has served as a reviewer, program committee, and organization committee in various software engineering conferences and journals, such as IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, *Empirical Software Engineering*, and IEEE/ACM International Conference on Program Comprehension.

Xin Zhou is working toward the Ph.D. degree with Singapore Management University, Singapore. His research interest is in code representation and software activity automation such as maintenance, debugging, and code reviewing. He is currently focusing on designing novel pretrained code representations and applying them to automate developer-intensive activities.

Dongsun Kim received the Ph.D. degree in computer science and engineering from Sogang University, Korea. He is an Assistant Professor with Kyungpook National University. His career includes several academic and industrial experiences. He has published several research papers and participated in several research projects relevant to AI-based software engineering. His recent achievements have focused on automated fix pattern mining, deep code representation for mining fix patterns, program repair driven by bug reports, fault localization impact on program repair, and specific topics for program repair.

David Lo (Fellow, IEEE) is a Professor in computer science and the Director of the information and systems cluster at the School of Computing and Information Systems, Singapore Management University. He leads the Software Analytics Research (SOAR) group. His research interest is in the intersection of software engineering, cybersecurity, and data science, encompassing socio-technical aspects and analysis of different kinds of software artifacts, with the goal of improving software quality and security and developer productivity.

Tegawendé F. Bissyandé is an Associate Professor with SnT, University of Luxembourg, where he leads a group of 25 researchers. He is an ERC Fellow and Principal Investigator of several projects funded by the European Commission, the Fonds National de la Recherche, and by Industry partners. His main interests are in software repair and software security with techniques based on program analysis and machine learning. He has published over 80 peer-reviewed research papers in various fields of computer science. As a native of Burkina Faso (West Africa), he is an enthusiastic advocate of capacity building for higher education in Africa.



Jacques Klein (Member, IEEE) is an Associate Professor with SnT, University of Luxembourg. He coleads a group of about 25 researchers focusing on software security, software reliability, and intelligent software. He has standing experience and expertise in (1) successfully running industrial projects, (2) Android security, including both static analysis techniques for tracking privacy leaks and machine learning for identifying malware, and (3) program repair. He published over 150 research papers in top journals/conferences.