Pre-trained Models for Bytecode Instructions

Donggyu Kim*, Taemin Kim*, Jiho Shin†, Song Wang†, Heeyoul Choi*, and Jaechang Nam*‡
**Handong Global University*, Pohang, South Korea
†*York University*, Toronto, Canada
{kdg,tmkim}@handong.edu, {jihoshin, wangsong}@yorku.ca, {hchoi, jcnam}@handong.edu

***Abstract*—Recent advancements in pre-trained models have rapidly expanded their applicability to various software engineering challenges. Despite this progress, current research predominantly focuses on source code and natural language processing, largely overlooking Java bytecode. Java bytecode, with its well-defined structure and high availability, presents a promising yet under-explored domain for leveraging pre-trained models. Its inherent properties, such as platform independence and optimized performance, make Java bytecode an ideal candidate for developing robust and efficient software engineering solutions. Addressing this gap could unlock new opportunities for enhancing automated program analysis, bug detection, and code generation tasks.**

In this study, we propose `byteT5` and `byteBERT`, which are pre-trained models with hexadecimal bytecode. To build our models, we developed a bytecode tokenizer, `ByteTok`, to generate hexadecimal input representations for our pre-trained models. We conduct an empirical study comparing our models and GPT-4o. The results indicate that `byteT5` and `byteBERT` outperform GPT-4o in the span masking task. We anticipate these findings will pave the way for novel approaches to addressing various software engineering challenges, particularly live patching.

***Index Terms*—Bytecode, T5, BERT, pre-trained model**

I. Introduction

Recently, pre-trained models for disassembled bytecode instructions were introduced to address software engineering problems. Choi and Nam [1] identified the inherent naturalness of disassembled bytecode instructions, emphasizing their potential to develop effective deep learning models. Some studies have explored leveraging disassembled bytecode in pre-trained models to address challenges such as vulnerability detection, malware analysis, and binary similarity assessment at the bytecode level [2], [3].

However, pre-trained models based on disassembled bytecode in the existing studies have limitations, highlighting the need for bytecode pre-trained models in hexadecimal format. After editing the disassembled bytecode, it is difficult to assemble the bytecode again in binary or hexadecimal format. This limitation obstructs the development of software engineering tools that require modifications to binary class files. To tackle this problem, hexadecimal bytecode must be directly used for pre-trained models. While, to our knowledge, no study exploits pre-trained models based on the hexadecimal representation of the bytecode.

The goal of this study is to propose `byteT5` and `byteBERT`, bytecode pre-trained models based on T5 and BERT architectures [4], [5]. To our knowledge, this is the first study to build a bytecode pre-trained model in hexadecimal representation. In our empirical study, we compared `byteT5` and `byteBERT` with GPT-4o. The results indicate potential for the span masking task, a variant of masked language modeling that employs continuous masking rather than single-token masking. While GPT-4o achieves 29.43% accuracy, `byteT5` and `byteBERT` achieve 53.82% and 64.58% accuracy respectively in the span masking task. These results can lead to the next steps to solve software engineering problems by using the pre-trained models based on hexadecimal bytecode representations.

The contributions of this study are as follows:

- We proposed `byteT5` and `byteBERT`, first bytecode pre-trained models in hexadecimal format. These will be an important foundation for further studies addressing various software engineering problems.
- We conducted an empirical study to compare `byteT5`, `byteBERT`, and GPT-4o which is a state-of-the-art large language model. The results show that our models outperform GPT-4o in the span masking task.
- We developed benchmark datasets to facilitate further studies and share a tool for bytecode tokenizing to generate a training dataset for a pre-trained model.[1]

II. Background

Bytecode is the output of the compilation process for various programming languages such as Java, Scala, and Kotlin. It follows a well-defined and strict structure, comprising multiple sections, including the constant pool, fields, methods, attributes, and more. Each section contains different types and associated fields. For instance, the constant pool has 30 distinct types, each with unique fields and byte lengths. The `Constant_Methodref` type, for example, includes fields such as `tag`, `class_index`, and `name_and_type_index`, occupying a total of 5 bytes. On the other hand, the `Constant_Long` type consists of fields like `tag`, `high_bytes`, and `low_bytes`, taking up 9 bytes—nearly double the size of `Constant_Methodref`. The variability in byte lengths across these types makes direct analysis of bytecode challenging when addressing software engineering problems.

Several studies focus on solving software engineering problems using bytecode. DexBERT is an Android bytecode prediction model based on BERT [2], [5]. It uses a disassembled

‡ Corresponding author

[1]https://github.com/ISEL-HGU/bytecode-pre-trained.git

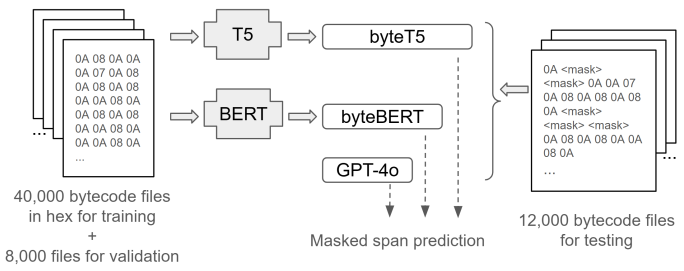

Fig. 1. Study Overview

bytecode format to address software engineering problems such as malicious code localization, defect prediction, and component-type classification tasks. It outperforms MKLDroid and smali2vec. ByteBack is a tool that verifies functional correctness properties using Java bytecode [3]. ByteBack employs a high-level Grimp intermediate representation rather than raw bytecode. DexBERT and ByteBack do not use the raw (binary or hexadecimal) bytecode format, and have limitations in addressing automated program repair (APR) problems because disassembled and intermediate representations are difficult to assemble back into raw bytecode. PraPR is a mutation-based APR tool using Java bytecode [6]. It leverages the ASM library to fix software bugs using a hexadecimal bytecode format, allowing direct modification without requiring recompilation [7]. However, since PraPR is mutation-based, human developers need to define mutation rules before execution. A significant limitation is that pre-defined mutation rules cannot cover all possible bug cases.

This study explores the feasibility of building bytecode pre-trained models using a hexadecimal representation. If such models can be built, they can enable the development of software engineering tools that address the limitations of existing bytecode-based approaches. For example, pre-trained models using a hexadecimal representation can be applied to both detection and repair tasks for software defects that cannot be effectively handled by a single existing approach, such as DexBERT, ByteBack, or PraPR.

III. Approach

This section outlines the process of building our pre-trained models. We began by mining JAR files and performing preprocessing. Subsequently, we developed `byteT5` and `byteBERT`, the first pre-trained bytecode models designed to operate on hexadecimal representations.

A. Data Collection

We collected JAR files from the Maven repository, retaining only the latest version for each project to eliminate duplicates. In this study, we focus on class files written in Java. Subsequently, we extracted the class files from the remaining 170,952 Java JAR files, resulting in a dataset of 2.5 million class files. Among them, we extracted constant pool and method instructions from each class file. Finally, we randomly selected 60,000 files including only constant pool and method instructions.

B. Data Preprocessing

1) Extracting constant pool and methods: Bytecode has a constant pool and method fields in its structure. Instructions from methods fields use a constant pool index to refer to constant values. We only use constant pool tag information and methods in this study, because instructions are only using constant pool index without knowing its actual value. For example, `ldc` instruction uses 2 bytes to indicate its type and constant pool index. By using constant pool index, `ldc` instruction can push an item from the run-time constant pool even if it does not know about the actual constant value.

2) Tokenization: Due to the varying lengths of bytecode constant pools, existing tokenization methods used in large language models for natural language and source code are impractical for bytecode. For example, Byte Pair Encoding (BPE) is commonly used to tokenize natural language by handling out-of-vocabulary words [8]. It segments words into subwords based on frequency. However, applying frequency-based tokenization would result in bytecode tokens that do not preserve the semantic information of the bytecode instructions.

For this reason, we develop `ByteTok`, a bytecode tokenizer, which divides bytecode into tokens following the JVM specification. By using `ByteTok`, we can tokenize constant pools and methods without losing the bytecode's semantics.

3) Model Setup: As in Fig. 1, we select 40,000 tokenized files as the training dataset, 8,000 files as the validation dataset, and 12,000 files as the testing dataset. We build our `byteT5` and `byteBERT` approaches with PyTorch library [9] and follow T5 [4] and BERT [5] architectures. Our models do not use pre-trained parameters, so we implement our model from scratch to facilitate converting the model's settings.

In training configuration, we use the cross-entropy loss function to update parameters and also use AdamW optimizer [10]. In this study, we use one A5000 GPU which has limited performance, so we set batch size as 8 and total epoch as 30. To cover small batch sizes, we use the gradient accumulation step of 4 to increase batch performance. In the model configuration, we define the number of heads as 8, maximum token length as 512, dimension size as 768, and vocabulary size as 51,107. Additionally, we define the total number of layers as 12, so `byteT5` has 6 layers of encoder and 6 layers of decoder, and `byteBERT` has 12 layers of encoder. Each model was trained for one day.

IV. Experimental Setup

A. Research Questions

To examine the performance of the pre-trained models for bytecode, we address the following research questions.

- RQ1: Is it feasible to develop a pre-trained model specifically for bytecode analysis?
- RQ2: Which of `byteT5`, `byteBERT`, and GPT-4o achieves higher accuracy in bytecode span masking task?

We implement `byteT5` and `byteBERT` with tokenized bytecode, and then compare the performance of span masking with GPT-4o.

Prompt:
You are an AI trained to analyze and complete bytecode representations of Java programs. Your task is to identify and fill in the missing information in the bytecode based on the given input.

Below is a tokenized version of the Java bytecode. Some parts of the bytecode, specifically within the constant pool and method sections, have been intentionally left blank (indicated by "`<mask>`"). Analyze the provided bytecode, infer the missing values, and fill in the masks accurately.

Input (Example of Tokenized Bytecode with Masks):
```
{<bos> 0A 08 0A 0A <mask> <mask> 0A 07 0A 08 <mask> 0A 08 0A <mask>
<mask> 08 0A 0A <mask> <mask> <mask> 08 0A 0A 0A 08 0A ...}
```

Expected Output (Completed Tokenized Bytecode):
- Only the completed bytecode. No additional text, comments, or formatting is allowed.
- Output only the modified bytecode, without additional commentary or formatting.

Fig. 2. Prompt Template for Bytecode Completion Task.

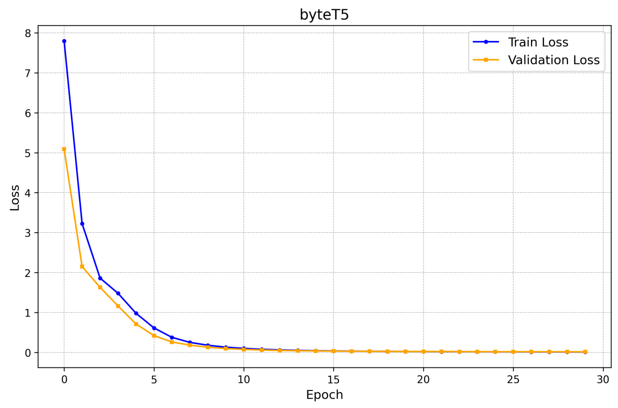

Fig. 3. Training / Validation Loss of `byteT5` over 30 Epochs

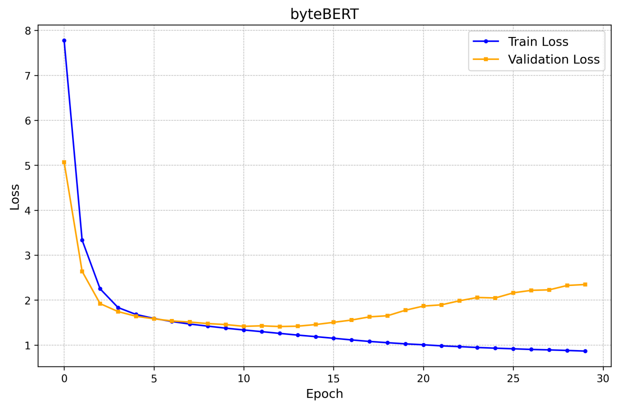

Fig. 4. Training / Validation Loss of `byteBERT` over 30 Epochs

The GPT-4o was chosen for this experiment as it is one of the most advanced large language models (LLMs) and is based on a pre-trained architecture. In this study, we used zero-shot learning, so we provided a template as input. As shown in Fig. 2, we divide the content into three sections. First, we outline the purpose of the conversation, then we give tokenized bytecode in hexadecimal representation, which contains constant pool and method instructions, to GPT-4o. Finally, it returns the generated bytecode.

V. Result

A. RQ1: Is it feasible to develop a pre-trained model specifically for bytecode analysis?

To address this research question, we assess whether `byteT5` and `byteBERT` can learn the abstracted structure of bytecode. We evaluate this by examining the training and validation loss curves because there is no benchmark for bytecode. As shown in Fig. 3, the average loss of `byteT5` decreases with each passing epoch in both the training and validation phases. However, Fig. 4 shows that the validation loss of `byteBERT` increases after 12 epochs. So, in this study, we used the parameters of the 12th epoch to stop earlier.

Although there is an overfitting problem from `byteBERT`, decreasing loss means that pre-trained models can learn the general architecture of bytecode. The overfitting problem can be solved by using a large dataset, so if we use more GPUs and data, we can prevent the overfitting problem.

RQ1 Answer: `byteT5` and `byteBERT` can learn the structure of bytecode and semantic information about constant pool and method instructions.

TABLE I
Comparision of byteT5, byteBERT, and GPT-4o

Model	# of corrects	Accuracy
GPT-4o	142,866	0.2943
byteT5	261,198	0.5382
byteBERT	**313,458**	**0.6458**

B. RQ2: Which of `byteT5`, `byteBERT`, and GPT-4o achieves higher accuracy in bytecode span masking task?

To further evaluate the capability of `byteT5`, `byteBERT`, and GPT-4o in bytecode analysis, we assess their accuracy on the span masking task using a test dataset comprising 12,000 tokenized bytecode samples. In Table 1, GPT-4o shows 29.43% accuracy of span masking task, while `byteT5` and `byteBERT` shows results 53.82% and 64.58%, respectively.

It shows that while GPT-4o has some ability to work with bytecode, its performance is much lower compared to `byteT5` and `byteBERT`. This difference comes from the fact that GPT-4o does not have training focused on bytecode, as it mainly uses general data from natural language and source code. Without specific training for bytecode, GPT-4o finds it hard to understand and correctly predict the complex and detailed patterns found in bytecode. On the other hand, `byteT5` and `byteBERT` are specially made to understand bytecode by training on a dataset of tokenized bytecode samples. The results show that `byteT5` and `byteBERT` not only learn the basic structure but also understand the meaning and connections in bytecode, allowing it to perform much better in the span masking task.

Additionally, the `byteT5` and `byteBERT` have different accuracies of span masking task. The reason is that `byteBERT` is a BERT-based approach, so it is good at masked language modeling tasks [5]. However, `byteT5` is based on the T5 architecture [4], which means it is good at translation tasks. We find that `byteBERT` achieves higher accuracy than `byteT5`, due to its architecture.

RQ2 Answer: byteT5 and byteBERT achieve **53.82%** and **64.58%** accuracy respectively, and outperform GPT-4o, which achieves **29.43%** accuracy. It clearly demonstrates their significantly superior performance in accuracy compared to GPT-4o.

VI. Discussion

Our study reveals the significant potential of pre-trained models in bytecode analysis. Specifically, even when operating with limited computational resources, byteT5 and byteBERT outperform GPT-4o in the span masking task. This superior performance indicates that specialized models tailored for bytecode are more effective in understanding and processing low-level code representations compared to more generalized models like GPT-4o. It underscores the importance of domain-specific pre-training, as these models are better equipped to capture the intricate patterns and structures inherent in bytecode. Consequently, we expect that leveraging such specialized pre-trained models can lead to more efficient and accurate analysis in tasks related to software testing, security vulnerability detection, or code optimization. To achieve these goals, we plan to conduct the following follow-up studies.

A. Increasing model size

In pre-trained models, leveraging large amounts of data, increasing the model parameters, and high-performance GPUs is crucial for achieving optimal performance. The computational resources not only allow for processing more data but also enable the training of larger and more complex models. As previously mentioned, byteT5 and byteBERT have a maximum input length of 512 tokens due to hardware limitations. This restriction means that some input sequences may be truncated, potentially causing the model to miss important context or information contained in longer sequences.

By utilizing more GPUs with higher memory capacities, we could increase the maximum input length beyond 512 tokens. This extension would allow byteT5 and byteBERT to process longer sequences, capturing more comprehensive bytecode information and potentially improving its understanding and performance on tasks that involve longer dependencies.

B. Representation of bytecode

We extract constant pool tag information and method instructions as input for our models to capture the essential elements of Java bytecode. However, as previously mentioned, some sequences exceed the maximum input length of 512 tokens. This limitation prevents byteT5 and byteBERT from processing these longer sequences entirely. This leads to the degradation of the model performance.

In these cases, alternative representations that can accommodate longer sequences without exceeding the model's capacity are required. For example, methods that condense or prioritize certain parts of the bytecode might help in fitting more information within the input length limit. To explore this, we will compare the performance of the ByteTok-based bytecode pre-trained model with that of other tokenizer-based bytecode pre-trained models. By evaluating different tokenization and representation strategies, we aim to identify an approach that effectively handles longer sequences and enhances the model's overall performance.

C. Potential Applications of Software Engineering Task

byteT5 and byteBERT use hexadecimal bytecode as input instead of source code or an intermediate representation. Our models enable direct modification and generation of actual bytecode. To demonstrate their capabilities, we plan to conduct additional experiments on downstream tasks such as test case generation, vulnerability detection, program repair for live patching, and other software engineering tasks. Furthermore, the use of pre-trained models with bytecode remains an under-explored area. In future work, we will compare our approach with existing source code-based methods and the disassembled bytecode-based approaches to evaluate their effectiveness.

D. Extension to other programming languages

One of the key advantages of bytecode is its scalability and interoperability between different programming languages. Languages such as Scala and Kotlin compile their source code into bytecode, which is executed on the Java virtual machine (JVM). This shared compilation target means that tools and analysis developed for Java bytecode can potentially be applied to programs written in these languages as well. To validate this scalability and cross-language compatibility, we plan to conduct experiments that analyze class files produced from Scala and Kotlin source code using byteT5 and byteBERT.

VII. Conclusion

In this study, we have introduced byteT5 and byteBERT , the first trial of using a pre-trained model designed specifically for bytecode analysis. Our experimental results demonstrate that bytecode can be a viable input for pre-trained models, with byteT5 and byteBERT outperforming GPT-4o on the span masking task despite operating under constrained hardware environments. This achievement highlights the effectiveness of these models in understanding and processing bytecode structures in hexadecimal format.

The significance of our work lies in opening new avenues for bytecode analysis using machine learning techniques. By successfully applying a pre-trained model to bytecode, we pave the way for advanced analyses in software testing, security, live patching, and cross-language interoperability within the JVM ecosystem.

In conclusion, our models represent a promising step toward integrating pre-trained models into bytecode analysis. This work lays the groundwork for further research in this domain, potentially leading to significant advances in software engineering and machine learning applications related to bytecode.

Acknowledgements

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korean government(MSIT) (RS-2024-00457866).

References

[1] Y.-H. Choi and J. Nam, "On the naturalness of bytecode instructions," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

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

[3] M. Paganoni and C. A. Furia, "Verifying functional correctness properties at the level of java bytecode," in *International Symposium on Formal Methods*. Springer, 2023, pp. 343–363.

[4] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, Jan. 2020.

[5] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[6] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[7] D. B. Miracle, S. L. Donaldson, S. D. Henry, C. Moosbrugger, G. J. Anton, B. R. Sanders, N. Hrivnak, C. Terman, J. Kinson, K. Muldoon *et al.*, *ASM handbook*. ASM international Materials Park, OH, 2001, vol. 21.

[8] R. Sennrich, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[9] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS workshop*, 2011.

[10] I. Loshchilov, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.