How Different Tokenization Algorithms Impact LLMs and Transformer Models for Binary Code Analysis

*Ahmed Mostafa, *Raisul Arefin, Samuel Mulder
*These authors contributed equally
Department of Computer Science and Software Engineering
Auburn University

Abstract—**Tokenization is fundamental in assembly code analysis, impacting intrinsic characteristics like vocabulary size, semantic coverage, and extrinsic performance in downstream tasks. Despite its significance, tokenization in the context of assembly code remains an underexplored area. This study aims to address this gap by evaluating the intrinsic properties of natural language processing (NLP) tokenization models and parameter choices, such as vocabulary size. We explore preprocessing customization options and pre-tokenization rules tailored to the unique characteristics of assembly code. Additionally, We assess their impact on downstream tasks like function signature prediction—a critical problem in binary code analysis.**

To this end, we conduct a thorough study on various tokenization models, systematically analyzing their efficiency in encoding assembly instructions and capturing semantic nuances. Through intrinsic evaluations, we compare tokenizers based on tokenization efficiency, vocabulary compression, and representational fidelity for assembly code. Using state-of-the-art pre-trained models such as the decoder-only Large Language Model (LLM) Llama 3.2, the encoder-only transformer BERT, and the encoder-decoder model BART, we evaluate the effectiveness of these tokenizers across multiple performance metrics. Preliminary findings indicate that tokenizer choice significantly influences downstream performance, with intrinsic metrics providing partial but incomplete predictability of extrinsic evaluation outcomes. These results reveal complex trade-offs between intrinsic tokenizer properties and their utility in practical assembly code tasks. Ultimately, this study provides valuable insights into optimizing tokenization models for low-level code analysis, contributing to the robustness and scalability of Natural Language Model (NLM)-based binary analysis workflows.

I. Introduction

Tokenization is critical in transforming raw input data into structured representations, a process of utmost importance for Machine Learning (ML) and NLM model tasks [1]–[3]. While tokenization strategies have been studied extensively for natural [4] and high-level programming languages [5], assembly code presents unique challenges due to its low-level operations, diverse instruction sets, and non-standardized syntax across architectures. These challenges highlight the need for specialized tokenization techniques that effectively capture assembly code's structural and semantic intricacies [2]. Despite its importance, the role of tokenization in assembly code processing remains underexplored, particularly in its impact on downstream tasks involving modern NLMs.

Recent research underscores the significant influence of tokenization on NLM model performance. Studies like Ali et al. [4] demonstrate that tokenization methods affect the model's efficiency and ability to generalize across NLP tasks. Additionally, Dagan et al. [5] emphasize the critical role of tokenizers in domain adaptation and fine-tuning, showing that pre-tokenization schemes and vocabulary size significantly impact model compression rates, training efficiency, and downstream performance. For binary code analysis, tokenization has also played a pivotal role in downstream tasks like binary similarity detection UniASM [2] and function name reassignment Gao et al. [3]. However, existing tokenization methods often fall short when applied to assembly code, primarily due to their reliance on token patterns optimized for natural language or high-level code, leading to suboptimal results in binary-focused applications.

Assembly code's reliance on hardware-specific instructions and lack of high-level abstractions complicates the creation of structured representations. CP-BCS [6] addresses this challenge by integrating Control Flow Graphs (CFGs) and pseudo code to bridge the semantic gap between assembly and human-readable summaries. Particularly in stripped binaries, advanced tokenization approaches, such as bidirectional instruction-level CFGs and pseudo-code refinement, prove essential for capturing execution behavior and logic semantics. CP-BCS [6] work emphasizes the need for tailored preprocessing Gao et al. [3] and CP-BCS [6], tokenization, and post-tokenization methods in assembly code analysis.

Tokenizers are crucial tools in processing assembly code for NLP tasks in binary analysis, as poorly designed tokenization strategies can significantly hinder model performance. Despite their importance, no comprehensive study has systematically evaluated intrinsic and extrinsic tokenizer performance in assembly code. This study addresses these limitations by systematically evaluating tokenization strategies for assembly code. We focus on the performance of various tokenizers and

Workshop on Binary Analysis Research (BAR) 2025
28 February 2025, San Diego, CA, USA
ISBN 979-8-9919276-4-2
https://dx.doi.org/10.14722/bar.2025.23xxx
www.ndss-symposium.org

tokenization methods when applied to decoder-only models, such as Llama 3.2 1B parameters (the model can be downloaded from here), encoder-only models, such as BERT [7], and encoder-decoder models such as BART [8]. Specifically, our contributions are as follows:

- We analyze intrinsic tokenizer performance in detail, assessing its ability to encode assembly instructions and components effectively.
- We evaluate extrinsic tokenizer performance by examining its impact on downstream tasks.
- We evaluate the impact of preprocessing the instructions before tokenization for downstream tasks.

This research contributes to the field by filling a critical gap in understanding tokenization for assembly code. It offers a framework for evaluating and optimizing tokenization strategies tailored to binary program analysis. By leveraging state-of-the-art models and domain-specific datasets, we provide actionable insights for developing more effective tokenization methods, ultimately advancing the capabilities of NLMs in binary analysis.

II. Background

Tokenization is crucial in natural language processing and binary analysis, which bridges raw data and machine understanding. It involves segmenting text or binary code into smaller units, known as tokens, enabling efficient processing by machine learning and large language models. The choice of tokenization strategy significantly influences model performance [4] mainly when dealing with specialized languages like assembly code.

A. Tokenization Algorithms

One weakness of using just words as tokens is that the vocabulary size will be unmanageable. If we have a maximum size limit of the vocabulary, there will be a lot of out-of-vocabulary (OOV) words. One solution might be to tokenize text based on characters. However, that would not generate meaningful tokens, and the number of tokens generated would be very large, even from a shorter text. The subword-based tokenization algorithms try to maintain a balance between these two approaches. This approach generally tries to keep frequent smaller words without splitting in the vocabulary. For example, the word “eat” might not be split, but “eating” might be split into “eat” and “ing”. The subword-based tokenization methods we will be using are:

1) ***WordPiece:*** WordPiece algorithm [9] is a subword-based tokenization formula. It was developed by Google to pretrain BERT and has been reused by other popular models like DistilBERT, MobileBERT, Funnel Transformers, and MPNET.

The WordPiece vocabulary is initialized with the special tokens and the initial alphabet. The initial alphabet is produced from the corpus. It first splits all the words in the corpus into subwords. For example, the word “one” is broken into: ‘o’, ‘##n’, and ‘##e’. Here, ‘o’ is different from the other two alphabets because it is at the beginning of a word. In short, the initial vocabulary contains all the initial letters of a word and all the other letters preceded by the prefix ‘##’. Then, these subwords are merged based on a rule to create longer subwords or whole words. This process is repeated until the vocabulary size is complete. The merging rule: For all the token pairs in the current vocabulary, a score is calculated according to the following formula:

$$score = \frac{\text{Frequency of pair}}{\text{Frequency of first element} \times \text{Frequency of second element}}$$

The pair of tokens with the highest score is merged and added to the vocabulary. Then, the process is repeated until the desired vocabulary size is reached.

During tokenizing new words, WordPiece looks for the longest match in the vocabulary. If the whole word is absent in the vocabulary, the longest match is used to split the word. For example, while tokenizing “cats”, if ”cats” is not present in the vocabulary but “cat” is, it will tokenize as “cat” and ‘##s’.

2) ***Byte Pair Encoding:*** BPE [10] is a data compression technique adapted as a subword tokenization method for natural language processing tasks. It was originally designed for compressing text and later used by models like GPT, GPT-2, BART, and DeBERTA.

The token selection method for vocabulary building is very similar to WordPiece. The major difference is how the score of each pair is calculated before merging. BPE starts with splitting the corpus into characters. So, the vocabulary will start with all the ASCII characters, at the very least, and probably some Unicode characters. Then, it will find the most frequent pair instead of using the equation like in WordPiece. The most frequent pair is merged and added to the vocabulary. This method is repeated until the desired vocabulary size is reached.

For tokenizing new words, BPE uses both the vocabulary and merging rules that it learned during vocabulary production. For example, for tokenizing ”cats”, it will first split the word into characters like ’c’, ’a’, ’t’, and ’s’. Then, it will use the learned merged rules to merge them and form the longest token possible.

3) ***Unigram:*** The Unigram model [11] is a language model that takes a different approach to building its vocabulary than algorithms like WordPiece and BPE. It starts with a large vocabulary and gradually trims it down. Unigram prunes tokens based on how much they impact the model’s likelihood over the entire corpus. In each iteration, Unigram calculates a loss. This loss is computed by tokenizing every word in the corpus, using the current vocabulary and the Unigram model determined by the frequencies of each token in the corpus. Subsequently, it evaluates the potential increase in this overall loss for each symbol in the vocabulary if that symbol were to be eliminated. Then, it removes the percentage of tokens whose log increase is the lowest. This process is iterated until the desired vocabulary size is obtained.

Tokenizing a new word involves examining every possible segmentation of the word into tokens. Each segmentation is evaluated by calculating the probability of that specific sequence according to the Unigram model. The general idea is to split a word into the least number of tokens possible.

Unigram is used in SentencePiece, which is the tokenization algorithm used by popular models like AlBERT, T5, mBART, Big Bird, and XLNet.

B. Related Works

1) Preprocessing: Preprocessing text before tokenization involves modifying the original input in such a way that better fits the target task. For example, converting all text to lowercase to ensure consistency, removing punctuation and special characters, etc. In the case of binary analysis, preprocessing is done a little differently than natural language.

Some tools ignore all the numeric values, such as Escalada et al. [12], TypeMiner [13]. Cati [14] and Stride [15] keep relatively small numeric values but remove large numbers.

Palmtree [16] performed a study for instruction representation learning. The authors preprocessed the code by replacing strings and large numeric values with special tokens. However, they kept the smaller numeric values as they often convey important information about accessed local variables, function arguments, or data structure fields.

Gao et al. [3] and CP-BCS [6] performed an instruction normalization method to mitigate data sparsity and OOV issues in binary analysis by simplifying instruction representation. Key steps include retaining mnemonics and registers, generalizing constants, and substituting function addresses and local jumps with placeholder tokens. This approach improves model generalization and learning efficiency.

2) Tokenization: Tokenization is the process of splitting text into smaller units called tokens, which can be words, subwords, or even characters. This is a crucial step in preparing text for NLP models, as it transforms raw text into a structured format that the models can understand and analyze. There are various ways binary analysis tools perform tokenization:

Learning-based Encoding: SnowWhite [17] is a machine learning approach to predict type information from stripped binaries. They analyzed their dataset and found that the number of unique tokens was vast due to the prevalence of numeric values. To keep the vocabulary of their tokenizer feasible, they developed a subword model tokenizer based on BPE.

Karampatsis et al. [18] performed an empirical study to find the best way to tokenize source code. Source codes are rich with identifiers, which can cause the vocabulary to explode. They came up with the idea of using character subsequences of tokens (subword units) to reduce the final vocabulary size.

Raw Bytes Encoding: Some tools encode the instructions as raw byte encoding and feed that to the NLP model. The one-hot encoding algorithm commonly uses this scheme. A byte consists of 8 bits and offers a range of 256. A vector of length 256 with one active dimension effectively encodes bytes as vectors. A few tools that pass similar input to NLP models are StateFormer [19], DEEPVSA [20], EKLAVYA [21] and [22].

Instruction-level tokenization: UniASM [2] evaluated the BPE, WordPiece, and three instruction-level (Full, Half, and Piece Instruction) tokenization algorithms to assess their effectiveness in binary code similarity detection tasks. The evaluation results demonstrated that the Full-Instruction tokenization method, which treats a single instruction as a token, consistently outperformed the other approaches by preserving instructions' structural and semantic integrity. Despite the effectiveness of the Full-Instruction tokenization technique, it can result in a more extensive vocabulary and is more susceptible to OOV issues.

StateFormer [19] took a different approach to handling numeric values. They used Neural Arithmetic Unit (NAU) [23], embeddings produced by which are supposed to capture the semantics of numerical values involved in arithmetic operations.

III. Approach

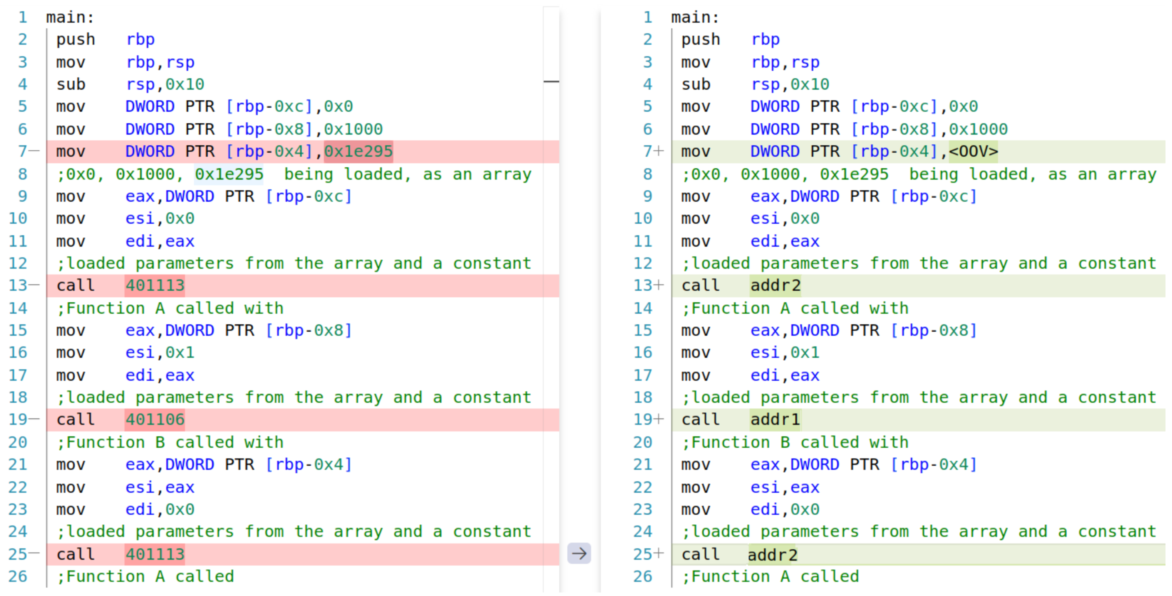
A. Tokenizers

To evaluate the impact of tokenizers on model performance, we conducted an ablation study focusing on the pretrained models: the decoder-only Llama 3.2 1B parameters, the encoder-only BERT, and the encoder-decoder BART-Base [8] model. Specifically, we created a diverse dataset for training customized tokenizers and models, including 80,000 disassembled C functions for model training and 20,000 disassembled functions for testing. We trained the models for each tokenizer while fixing the remaining configurations, such as datasets, training procedures, and hyperparameters. This controlled setup enabled us to isolate and quantify each tokenizer's effect on the models' downstream performance.

Our study uses the Hugging Face tokenizer library to implement three well-established tokenization algorithms: BPE, Unigram, and WordPiece. Each tokenization algorithm was tested with three vocabulary sizes: 3K, 25K, and 35K. Additionally, only the Llama 3.2 model was tested with a 128K vocabulary size across all tokenization algorithms on the function signature prediction downstream task. The 128K vocabulary size is comparable to the Llama 3.2 model's default tokenizer's vocabulary size.

To further evaluate the impact of the code preprocessing, each of these tokenizers was trained on two versions of the dataset: the default disassembly without any preprocessing and the other customized using a preprocessing method. The preprocessing method is discussed in detail in the subsection (***C. Dataset***) from this section.

In addition, the base tokenizer that comes pre-trained with the Llama 3.2, BERT, and BART models was evaluated on both versions of the dataset without any customization or additional training. This, combined with assessing the trained tokenizers, resulted in **86 models** across the (Llama 3.2, BERT, and BART) models, with three tokenization algorithms (BPE, Unigram, and WordPiece) and four vocabulary sizes (3K, 25K, 35K, and 128K). This experimental setup ensures a comprehensive and comparative evaluation of the Llama 3.2, BERT, and BART models. This design allows us to systematically analyze the effects of tokenization algorithms,



```
main:
push    rbp
mov     rbp,rsp
sub     rsp,0x10
mov     DWORD PTR [rbp-0xc],0x0
mov     DWORD PTR [rbp-0x8],0x1000
mov     DWORD PTR [rbp-0x4],0x1e295
;0x0, 0x1000, 0x1e295  being loaded, as an array
mov     eax,DWORD PTR [rbp-0xc]
mov     esi,0x0
mov     edi,eax
;loaded parameters from the array and a constant
call    401113
;Function A called with
mov     eax,DWORD PTR [rbp-0x8]
mov     esi,0x1
mov     edi,eax
;loaded parameters from the array and a constant
call    401106
;Function B called with
mov     eax,DWORD PTR [rbp-0x4]
mov     esi,eax
mov     edi,0x0
;loaded parameters from the array and a constant
call    401113
;Function A called
```

```
main:
push    rbp
mov     rbp,rsp
sub     rsp,0x10
mov     DWORD PTR [rbp-0xc],0x0
mov     DWORD PTR [rbp-0x8],0x1000
mov     DWORD PTR [rbp-0x4],<OOV>
;0x0, 0x1000, 0x1e295  being loaded, as an array
mov     eax,DWORD PTR [rbp-0xc]
mov     esi,0x0
mov     edi,eax
;loaded parameters from the array and a constant
call    addr2
;Function A called with
mov     eax,DWORD PTR [rbp-0x8]
mov     esi,0x1
mov     edi,eax
;loaded parameters from the array and a constant
call    addr1
;Function B called with
mov     eax,DWORD PTR [rbp-0x4]
mov     esi,eax
mov     edi,0x0
;loaded parameters from the array and a constant
call    addr2
;Function A called
```

Fig. 1. An example of address-to-sequential-identifiers preprocessing. The code on the left represents the original code before preprocessing, while the code on the right shows the result after preprocessing.

vocabulary sizes, and dataset preprocessing on downstream model performance.

The choice of vocabulary sizes was driven by the need to balance efficiency and expressiveness. Smaller vocabularies, such as 3K, are expected to reduce memory and computational overhead while increasing sequence length due to more granular subword segmentation. On the other hand, larger vocabularies, like 35K and 128K, may better capture semantic and syntactic patterns by encoding longer subwords, reducing sequence length but increasing memory usage. The 25K vocabulary serves as a middle ground to assess trade-offs between granularity and model efficiency. Additionally, the 128K vocabulary size was included to evaluate the tokenizers' ability to handle an extensive vocabulary, capturing a wide range of tokens and potential rare subwords, which could be beneficial for highly complex and diverse datasets.

By varying the tokenization algorithm and vocabulary sizes, we aim to analyze the effect of tokenization granularity on downstream model performance, particularly on the decoder-only, encoder-only, and encoder-decoder models. This approach enables us to identify the optimal tokenizer configuration for assembly code analysis tailored to a specific downstream task while accounting for algorithmic and vocabulary design choices. The tokenizers' configurations are described in Appendix A.

B. *Preprocessing*

NLP models often struggle with understanding and processing numbers effectively because they are primarily trained on textual data, where numbers appear in diverse and inconsistent formats [24], [25]. Unlike words, numbers require precise mathematical reasoning, comparison, or context-specific understanding, which standard tokenization and embedding techniques fail to capture adequately. Experiments have shown that representing numbers in a better way can improve NLP model performance [25]. Considering this and the fact that instructions contain many numerical values and significantly impact the semantics of the code, we must emphasize finding a better numeric representation.

One of the significant challenges in handling numeric values within disassembled code is their extensive range. The sheer variety of possible numbers makes it infeasible for any model to learn embeddings for all of them effectively. Previous tools have tackled this issue in different ways. Some entirely removed numeric values from the disassembled code, while others retained smaller numbers and assigned a special token for larger ones. However, these approaches often lack justification for choosing one method over another. Our work addresses this gap by clearly and systematically comparing different approaches, offering valuable insights into their relative performance and effectiveness.

We are comparing two different variations of representing numeric values in disassembled code. They are:

1) Default: This approach represents the baseline method, in which the disassembled code, including its numeric values, is used without modification. This unaltered input serves as a reference point for evaluating and comparing the performance of alternative preprocessing methods.

2) Address to Sequential Identifiers & Hexadecimal Numeric Values to Decimal: This preprocessing method replaces memory addresses in the code with sequential identifiers and converts all hexadecimal numeric values into their corresponding decimal representations. Large and widely varying memory addresses in disassembled code are highly specific to individual programs or runtime environments, while the representation of numeric values in hexadecimal adds further complexity. These variations make memory addresses and hexadecimal values challenging for tokenizers to process and limit their semantic usefulness for downstream tasks.

The proposed preprocessing method replaces every distinct memory address in the code with a sequential identifier. For example, if the code contains addresses such as 0x1FF0, 0x1FF4, 0x2000, and 0x2AB8, they will be converted into tokens like addr1, addr2, addr3, and addr4. Each distinct address is assigned a unique token, preserving its identity while normalizing its representation into a manageable vocabulary. Similarly, all hexadecimal numeric values in the code are replaced with corresponding decimal representations, ensuring uniformity in numeric formats. Suppose the vocabulary size is set to 3000. The most frequent tokens in the code, such as mnemonics and other operational strings, will appear multiple times and naturally occupy slots in the vocabulary. Frequently occurring small numbers will also be included in the vocabulary. In contrast, less frequent large numbers are unlikely to appear in the vocabulary and will instead be replaced with a special token, e.g., <OOV>. The intuition behind this approach is to normalize memory addresses using unique identifiers, retain frequently used smaller numbers, and eliminate less frequent outliers. This strategy balances the need for effective representation with the constraints of a fixed vocabulary size. An example illustrating the address to sequential identifiers method is shown in Figure 1.

Two memory addresses appear in lines 13, 19, and 25. Additionally, other numeric values are used across different lines. After applying the preprocessing method, the exact values of the addresses are replaced with sequential identifiers, addr1 and addr2. This transformation retains all relevant information. We can still identify these as addresses and understand that lines 12 and 25 call the same function while line 19 calls a different one. The precise values of the addresses are irrelevant for analysis.

Similarly, smaller numbers remain unchanged, while the large numeric value in line 7 is replaced with an OOV token due to its rarity. This replacement highlights its status as an outlier that does not occur frequently. Normalizing memory addresses and handling numeric values appropriately adds value by making the code more structured and interpretable, potentially simplifying the model's task in downstream applications.

C. Dataset

We scraped code from publicly available GitHub repositories containing C source code to create our dataset. The goal was to ensure a diverse and representative collection of C programs. After collecting the source code, we compiled the programs using the GCC compiler with optimization level 2 and debugging information to preserve metadata. After compilation, we disassembled the binaries using Ghidra [26] to extract individual functions and their signatures. We employed TLSH [27], a fuzzy hashing technique, to identify and remove duplicate functions to ensure uniqueness and eliminate redundancy in the dataset. Following deduplication, we randomly selected 100,000 functions to form the final dataset. Although the initial pool of functions was significantly more extensive, we opted for this subset due to resource constraints, as our experiments involved extensive computational demands. We filtered the functions in the dataset to include only those containing at least 30 and at most 100 instructions. A histogram of the Frequency distribution of disassembled functions based on the number of instructions per function is depicted in Figure 2. This range was chosen to capture meaningful functionality while avoiding excessively long functions. The resulting dataset provides a robust basis for experimentation, combining diversity from the initial scraping process, uniqueness ensured by deduplication, and a controlled size and complexity range for efficient analysis. We will make our dataset publicly available upon publication.

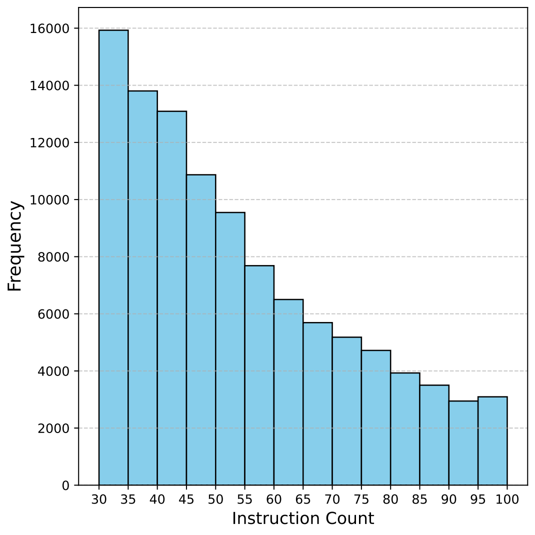


Fig. 2. Frequency distribution of disassembled functions based on the number of instructions per function.

D. Models

To assess the impact of the trained tokenizers on downstream model performance, we fine-tuned three models: Llama 3.2 1B, a decoder-only model with causal language modeling (CLM) training objective, the BERT, an encoder-only transformer model, and the BART, encoder-decoder transformer model. The trained tokenizers were used to fine-tune the models, with evaluations conducted on their respective versions of the default and preprocessing disassembly datasets. This setup allowed for a detailed analysis of how tokenization algorithms, vocabulary sizes, and dataset preprocessing influence the models' downstream task performance. The models' configurations are described in Appendix B.

E. Evaluation

Our study was structured into two key phases to evaluate the impact of tokenization strategies on downstream model performance: **intrinsic evaluation**, and **extrinsic evaluation**.

1) Intrinsic evaluation overview: focused on analyzing tokenizer performance independently of the models, particularly emphasizing the fertility metric, the overlap between the vocabulary generated by different tokenizers, and testing all tokenizers against the out-of-vocabulary issue. The evaluation was performed using a held-out set of 20,000 disassembled functions, ensuring the evaluation data was not used during tokenizer training. This phase aimed to provide insights into the tokenizers' properties without considering their direct impact on model performance.

Fertility, a widely used metric in NLP Scao et al. [28]; Stollenwerk [29]; Rust et al. [30], measures the average number of tokens required to represent a word or document. In our study, we adapted this metric to the domain of **functions' disassembly code**, where a function's disassembly serves as an analogous unit to a document in NLP. Specifically, fertility in this context quantifies the average number of tokens required to represent the instructions and operands of a disassembled function. This adaptation allows us to assess the compression efficiency and granularity of the tokenizers when applied to assembly code, which, like natural language, contains structural and semantic patterns.

To calculate fertility, we divided the total number of tokens generated by a tokenizer for a dataset of disassembled functions by the total number of instructions in those functions. Instructions were identified using a standardized parsing process that splits assembly code at line breaks. A higher fertility value indicates lower compression efficiency, suggesting that the tokenizer produces more tokens per instruction, which can impact the downstream processing of binary code.

By applying the fertility metric to functions' disassembly code, we gained critical insights into how tokenizers such as BPE, Unigram, and WordPiece with varying vocabulary sizes capture low-level code's structural and semantic information. These insights laid the groundwork for the extrinsic evaluations, where we examined the impact of these tokenizers on the downstream performance of the models.

2) Extrinsic evaluation overview: Extrinsic evaluation assesses the performance of models on downstream tasks to understand the impact of different tokenizers on their effectiveness. In this study, we evaluate the BERT model on masked token prediction accuracy, which aligns with its masked language modeling (MLM) training objective. For the Llama 3.2 model, we evaluate its ability to recover entire disassembled functions, including instructions or parts of instructions, after masking randomly selected tokens. The masked regions are determined by the tokenization strategy employed during the experiments. Although masked token prediction is traditionally a pre-training objective, we included it in our evaluation to measure the models' understanding of disassembly code structure, semantics, and syntax. This analysis underscores the models' ability to interpret low-level assembly instructions and accurately reconstruct missing or obscured tokens.

For the Llama 3.2 model, recovering the entire function disassembly, rather than just the masked tokens, is crucial for assessing its ability to generate coherent and accurate outputs for real-world applications such as code completion, reverse engineering, and vulnerability detection. This evaluation highlights the model's capacity to reconstruct the full execution logic of functions, providing insights into its generative capabilities and robustness in binary program analysis tasks.

Additionally, the Llama 3.2 and BART models were evaluated on the function signature prediction task, a critical downstream task in binary analysis. Function signature prediction involves inferring high-level function prototypes (parameter and return types) from low-level code. This task is important for recovering meaningful symbolic information from stripped binaries, enabling better code comprehension and facilitating downstream tasks such as debugging, optimization, and malware analysis. These evaluations collectively provide a comprehensive view of the tokenizers' impact on model performance across tasks requiring generation and understanding capabilities.

The decision to select the BART-Base model instead of the BERT model to evaluate the function signature prediction downstream task performance was because the BERT is not a generative model and is thus unsuitable for this task. Our choice of models aimed to include a very large model (Llama 3.2) and a smaller model (BART-Base) to provide a comparative perspective. The BART-Base model, being a smaller generative model, offers valuable insights into how model size impacts performance on function signature prediction compared to a larger model like Llama 3.2.

IV. Intrinsic Evaluation of Tokenizers

We begin by analyzing the fertility scores of the trained tokenizers using an unseen dataset consisting of 20,000 disassembled functions and then examine their vocabulary's overlapping insights.

A. Analysis of Fertility Scores

The fertility study, as described, evaluates the number of tokens BPE, Unigram, and WordPiece tokenizers require to represent instructions in the unseen default and preprocessed disassembly datasets. Fertility measures each tokenizer's efficiency and compression capability. Our observations, based on the fertility score comparison depicted in Figures 3a and 3b, are as follows:

1) Default Disassembly Dataset:

- **WordPiece** consistently shows the highest fertility score across all vocabulary sizes (4.5 tokens per instruction), indicating it produces the most tokens per instruction and demonstrates the lowest compression efficiency.
- **Unigram** achieves the lowest fertility score, consistently around 2.0 tokens per instruction, showcasing the highest compression capability among the three tokenizers.
- **BPE** lies between WordPiece and Unigram, with fertility scores decreasing from approximately 3.0 to 2.5 as the vocabulary size increases.

2) Preprocessed Disassembly Dataset:

- Similar trends are observed, where **WordPiece** maintains the highest fertility score, followed by BPE and Unigram.
- o The preprocessing step slightly reduces fertility for all tokenizers, suggesting better alignment between the tokenizers and the structure of preprocessed disassembly.

We conclude that the **Unigram** is the most efficient tokenizer in terms of compression for both datasets, requiring fewer tokens per instruction, making it ideal for tasks prioritizing compact representations. **WordPiece**, due to its high fertility, may preserve more granularity in tokenization, which could benefit specific tasks requiring detailed token-level information but at the cost of efficiency. The **BPE** tokenizer balances compression and granularity, making it a versatile choice for disassembly tasks.

B. Vocabulary Overlap Study

The vocabulary overlap study examines the percentage of shared vocabularies across BPE, Unigram, and WordPiece tokenizers for four vocabulary sizes (3K, 25K, 35K, 128K). We measure the overlap for both the default and preprocessed disassembly datasets. Our observations, based on vocabulary overlap percentage shown in Table I, are as follows:

1) Overlap Trends:

- The vocabulary overlap percentage decreases as the vocabulary size increases, indicating less agreement between tokenizers with larger vocabularies.
- **For the default disassembly dataset:** At a vocabulary size of 3K, the overlap percentage is relatively low (0.75 or 63 tokens), which diminishes as the vocabulary size increases to 128K (0.09 or 187 tokens).
- **For the preprocessed disassembly dataset:** The overlap is slightly higher than in the default dataset, especially at smaller vocabulary sizes (1.04 or 86 tokens at 3K).

2) Impact of Preprocessing: Preprocessing enhances vocabulary alignment across tokenizers, as reflected in the higher overlap values for all vocabulary sizes.

We conclude that the overlap between vocabularies is minimal, suggesting that each tokenizer captures unique aspects of the data and may tokenize differently based on its underlying algorithm. Preprocessing the dataset enhances tokenization pattern alignment across the tokenizers, slightly increasing the shared vocabulary. Tasks requiring consistency across tokenizers may benefit from preprocessing to improve uniformity, although the distinct tokenization mechanisms (subword segmentation strategies) will continue to produce unique vocabularies.

In Appendix C, we discuss vocabulary overlap heatmaps, which are shown in Figure 4. The heatmaps depict the similarity level between different tokenizers by comparing the vocabulary overlapping percentage of different tokenizers across the two datasets (default and preprocessed) and varying vocabulary sizes (3K, 25K, 35K, and 128K).

TABLE I
VOCABULARY OVERLAP PERCENTAGE ACROSS TOKENIZERS FOR DIFFERENT VOCABULARY SIZES

Vocabulary-Size	**Default Disassembly**		**Preprocessed Disassembly**	
	Percentage	# Tokens	Percentage	# Tokens
3K	0.75%	63	1.04%	86
25K	0.13%	92	0.25%	174
35K	0.1%	102	0.31%	267
128K	0.09%	187	0.44%	845

TABLE II
AVERAGE ACCURACY OF MASKED TOKEN PREDICTION ON THE DEFAULT AND PREPROCESSED DATASETS FOR BERT ACROSS TOKENIZERS AND VOCABULARY SIZES

Tokenizer	**Default Disassembly**		**Preprocessed Disassembly**	
	Llama-Accuracy	BERT-Accuracy	Llama-Accuracy	BERT-Accuracy
Unigram-3k	70.20	73.98	71.93	74.25
Unigram-25k	71.65	82.86	73.64	83.47
Unigram-35k	71.69	83.24	73.62	84.54
WordPiece-3k	71.20	73.38	70.92	75.73
WordPiece-25k	71.00	82.17	71.23	84.45
WordPiece-35k	71.67	83.45	72.40	86.49
BPE-3k	70.90	72.94	72.32	75.28
BPE-25k	70.58	82.84	72.86	85.65
BPE-35k	71.60	84.28	72.86	**86.58**
Model-default	80.48	85.37	82.30	78.08

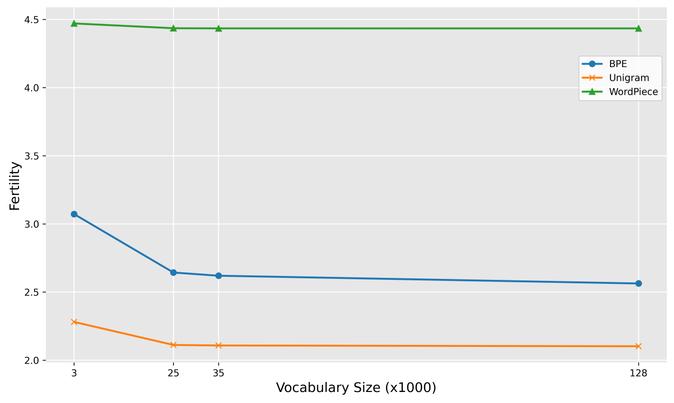
C. Out-of-Vocabulary Analysis

All tokenizers (BPE, Unigram, WordPiece) with various vocabulary sizes were evaluated on the test dataset for OOV issues. None of the tokenizers exhibited the OOV problem; they successfully recognized all tokens within their respective vocabulary lists, and no tokens were classified as unk_token.

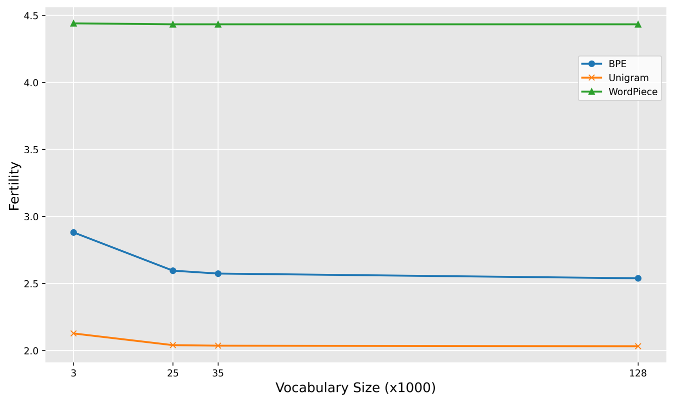
V. EXTRINSIC EVALUATION OF TOKENIZERS

This section provides an in-depth analysis of how various tokenization algorithms, coupled with different vocabulary sizes and dataset representations, influence performance in two key areas: masked token prediction and the downstream task of function signature prediction. We applied a masking strategy where 15% of the tokens in each disassembled function were randomly selected and replaced with a special [MASK] token. In contrast, the remaining 85% of the tokens were left unmasked. This approach ensures a consistent proportion of masked tokens across all functions.

The evaluation aims to uncover the interplay between tokenization strategies and model effectiveness in accurately understanding and generating disassembly code. The experimental results for the Llama 3.2 model are discussed in subsections B and C below, while the experimental results for the BERT and BART models are discussed in subsections A and D, respectively below.



(a) Default disassembly dataset



(b) Preprocessed disassembly dataset

Fig. 3. Fertility evaluation comparison between BPE, Unigram, and WordPiece tokenizers on (a) The default disassembly dataset and (b) The Preprocessed disassembly dataset.

TABLE III
AVERAGE ACCURACY OF FUNCTION PARAMETER AND RETURN TYPE PREDICTION ON THE DEFAULT AND PREPROCESSED DATASETS FOR LLAMA 3.2 AND BART ACROSS TOKENIZERS AND VOCABULARY SIZES

Tokenizer	**Default Disassembly**		**Preprocessed Disassembly**	
	Llama-Accuracy	BART-Accuracy	Llama-Accuracy	BART-Accuracy
BPE-3K	82.44	85.19	81.99	87.48
BPE-25K	85.42	86.42	85.45	87.62
BPE-35K	**85.76**	86.94	85.25	87.18
BPE-128K	85.23	-	84.56	-
Unigram-3K	74.78	84.97	75.58	**88.81**
Unigram-25K	77.66	80.57	78.12	86.03
Unigram-35K	77.74	66.36	77.96	81.40
Unigram-128K	76.88	-	77.95	-
WordPiece-3K	82.40	87.44	80.40	84.97
WordPiece-25K	83.25	87.53	81.75	86.48
WordPiece-35K	76.31	87.01	83.48	87.35
WordPiece-128K	83.66	-	82.04	-
Model-default	84.87	86.92	85.71	87.16

A. Performance Evaluation of Masked Token Prediction with BERT

In this experiment, we evaluated the performance of several tokenizers in the context of masked token prediction accuracy. This task involves predicting the original token based on its context within a sequence where specific tokens have been replaced with a mask placeholder. We focused solely on the accuracy of predicting the masked tokens, which comprised 15% of the input sequence, rather than including unmasked tokens in the evaluation. This approach assesses BERT's ability to accurately infer the masked tokens from the surrounding context.

BERT performed better than Llama in most of the experiments. BERT excels in masked token prediction mainly due to its bidirectional context processing. This allows BERT to effectively understand and use the context around masked tokens. Besides, BERT was originally designed to be pre-trained in masked token prediction, which makes it a better fit for this task.

The evaluation scores presented in Table II show several trends. The model performed better with higher vocabulary-sized tokenizers. The average accuracy increases with higher vocabulary size across all the different tokenizers with both the default and preprocessed datasets.

Likely, the reason for that trend is that with a larger vocabulary size, there are fewer or even no OOV tokens, like in our case. Machine code has a high frequency of numerical values, and the range of the numerical values can be very wide. If the vocabulary size is large, the tokenizers can recognize more outliers, assisting in a better token prediction.

An additional improvement is evident when using the preprocessed dataset compared to the default dataset. This outcome is expected, as the preprocessed dataset is normalized by replacing all addresses with sequential identifiers. Address values in the default dataset can vary widely, introducing significant outliers. Normalizing these values reduces variability and eliminates potential outliers, improving model performance. Among the tokenization algorithms, BPE performed best with larger vocabulary sizes.

Notably, the average masked token prediction accuracy across all tokenizers paired with the preprocessed dataset is consistently higher than with the default dataset. Interestingly, while the BERT default tokenizer performed well with the default dataset, it showed significantly poorer performance when paired with the preprocessed dataset, emphasizing the need to align tokenization strategies with the preprocessing approach.

B. Performance Evaluation of Masked Token Prediction with Llama 3.2

The evaluation scores can be found in Table II. For masked token prediction, Llama did not perform similarly to

BERT, which is expected. BERT's specific design and training make it particularly strong in this area. That's why even with a simpler and lighter design, BERT performed better than Llama. However, the highest accuracy is not the goal of this experiment. The impact of the variations in the tokenizing algorithm, vocabulary size, and preprocessing is much more interesting. The preprocessing algorithm consistently enabled higher performance across different tokenizers and vocabulary sizes. The Llama's default pre-trained tokenizer, trained with the dataset, showed the highest performance, 80.48%, and 82.30% accuracy for the default and preprocessed dataset, respectively. For the custom tokenizers, the highest accuracy we obtained is 71.69% for the Unigram-35k tokenizer on the default disassembly. For the preprocessed dataset, the highest accuracy observed is 73.64%.

C. Performance Evaluation of Function Parameters and Return Types Prediction with Llama 3.2

The Llama 3.2 model evaluation results presented in Table III focus on function signature prediction, a downstream task of predicting function parameters and return types from the function's disassembly:

1) Impact of Vocabulary Size: Table III shows that vocabulary size can marginally impact function signature prediction average accuracy. Increasing the vocabulary size improves accuracy for all tokenizers across the small and moderate vocabulary sizes 3K to 35K (e.g., BPE improves accuracy from 82.44% for 3K to 85.76% for 35K on the default dataset and from 81.99% for 3K to 85.45% for 25K on the preprocessed dataset).

However, the impact of vocabulary size is less pronounced for WordPiece, where improvements are relatively marginal (e.g., WordPiece improves accuracy from 82.40% for 3K to 83.25% for 25K on the default dataset).

2) Preprocessed vs. Default Datasets: On average, the preprocessed dataset improves the default dataset in function signature prediction accuracy. Preprocessing likely enhances the disassembly functions' structural uniformity and semantic clarity, leading to more accurate parameter and return type predictions.

3) Impact of Tokenization Algorithms: The performance of the tokenization algorithms differs across datasets and vocabulary sizes:

- **BPE Tokenizer:** Consistently achieves the highest average accuracy across all vocabulary sizes for the default and preprocessed datasets (e.g., 85.76% for BPE-35K on the default dataset).
- **Unigram Tokenizer:** This tokenizer shows the lowest average accuracy among all tokenizers, with results generally below 80% across vocabulary sizes (e.g., 74.78% for Unigram-3K on the default dataset and 75.58% for Unigram-3K on the preprocessed dataset).
- **WorPiece Tokenizer:** Performs moderately well, with accuracy slightly behind BPE but significantly better than Unigram (e.g., 83.66% for WordPiece-128K on the default dataset).

The Llama 3.2 model's default pre-trained tokenizer outperformed the Unigram tokenizer on both datasets and all vocabulary sizes, and on average, it achieved slightly higher accuracy than the WordPiece tokenizer across both datasets and all vocabulary sizes. However, the BPE tokenizer achieved very close average accuracy to the Llama 3.2 pre-trained tokenizer, particularly on vocabulary sizes 25K to 128K.

The BPE achieved an average accuracy of 85.76% for 35K vocabulary size on the default dataset, slightly higher than the average accuracy of the Llama 3.2 model's default pre-trained tokenizer on both datasets.

D. Performance Evaluation of Function Parameters and Return Types Prediction with BART

The evaluation scores of the signature prediction Task with BART are presented in Table III. BART performed best with the smallest vocabulary size and preprocessed disassembly for function signature prediction. Across the tokenizers, the performance is mixed. For the default disassembly, WordPiece performed well consistently across all the vocabulary sizes. With preprocessed disassembly, Unigram-3k performed best with 88.81% accuracy. Similarly, BPE also performed well with smaller vocabulary sizes. This means a combination of preprocessing and smaller vocabulary size represents the token in a better way for the model to understand the parameters and return type work in a function. The preprocessing step normalizes the addresses, which is beneficial for the signature prediction task because the specific values of the addresses are irrelevant for this particular task. It is enough to know that a token is an address, as the exact value is irrelevant.

VI. Discussion

The choice of tokenization algorithm, vocabulary size, and dataset representation is crucial and should align with the model type and task. For instance, in the masked token prediction pre-training task, the BERT model paired with a BPE tokenizer and a moderate vocabulary size of 35K, applied to a preprocessed machine code dataset, emerged as the optimal configuration among the evaluated options.

Similarly, for function signature prediction, the BART model paired with a Unigram tokenizer with a small vocabulary size of 3K and preprocessed machine code demonstrated the best performance.

A notable finding is the consistent benefit of dataset preprocessing, which enhances downstream performance, particularly for smaller to moderate vocabulary sizes across all tokenizers and models.

However, the performance gains from preprocessing were negligible for the models' default pre-trained tokenizers. Interestingly, BERT's default tokenizer achieved higher average accuracy on masked token prediction tasks using the default dataset compared to the preprocessed version, underscoring the importance of task-specific dataset-tokenizer alignment.

Insights from the Intrinsic Evaluation of Tokenizers: The intrinsic evaluation highlights key trade-offs between tokenization efficiency, granularity, and alignment. Unigram demonstrates superior compression with the lowest fertility scores, making it ideal for tasks prioritizing compact representations. WordPiece, with higher fertility scores, provides granular tokenization, which may benefit tasks requiring detailed token-level information. BPE balances efficiency and granularity, offering versatility for disassembly tasks.

Preprocessing improves tokenization efficiency and slightly enhances vocabulary alignment across tokenizers. However, minimal overlap among tokenizers, especially with larger vocabularies, suggests that each algorithm captures unique data characteristics. Despite this, preprocessing aids in achieving greater uniformity in tokenization patterns, which could benefit tasks requiring consistency.

VII. Limitations

Despite the scope of our study, it faces the following limitations:

1) Lack of Hyperparameter Optimization: We did not conduct extensive hyperparameter optimization for each tokenizer to minimize computational time costs and maintain focus on the study's primary objectives. This decision, however, may have constrained the potential performance gains achievable with fine-tuned configurations. Additionally, exploring the impact of hyperparameters such as learning rate, batch size, or dropout rates on downstream tasks could provide valuable insights. Future work could investigate these interactions to identify optimal configurations that enhance the alignment between tokenizers and models across diverse tasks.

2) Tokenizer Implementation Variants: Our study relied primarily on specific implementations of tokenizers, such as those provided by the Hugging Face library. While this ensures compatibility with the models used in our study, alternative implementations, such as SentencePiece, may yield different results. Investigating the impact of implementation details on tokenization and downstream performance remains an area for further exploration.

3) Intrinsic and Extrinsic Correlation Analysis: We did not study the correlation between the intrinsic properties of tokenizers (e.g., vocabulary size, token overlap, token distribution) and their extrinsic evaluation on downstream task performance. Understanding this relationship could provide deeper insights into how tokenizer design impacts model behavior and performance across tasks, and we encourage future work to explore this dimension.

4) Scaling to Larger Models: While our study focused on models with up to 1 billion parameters, we did not evaluate the tokenizers' performance on larger models. Extending the evaluation to larger architectures may uncover additional insights, as tokenization effects could behave differently in models with significantly more parameters.

5) Real-world Dataset and Task Coverage: Our evaluation was conducted on specific downstream tasks and machine code datasets. While these tasks and datasets are relevant to the context of this work, they may not fully represent the diversity of real-world applications. Future studies should extend the evaluation to a broader range of datasets and tasks to validate the generalizability of our findings.

By addressing these limitations, future research can refine our understanding of tokenization algorithms, exploring their intrinsic properties, broader applicability, and robustness across diverse tasks, model architectures, and evaluation settings.

VIII. Conclusion & Future Work

This study on tokenization algorithms for binary code analysis highlights the critical role of tokenization strategies in optimizing the performance of LLMs and transformer-based models. By evaluating the intrinsic properties of various tokenizers and their extrinsic performance on downstream tasks like function signature prediction, we demonstrated that both the choice of tokenization algorithm and vocabulary size significantly influence model outcomes. Although we did not directly study the impact of intrinsic tokenizer properties on downstream task performance, our findings emphasize the importance of selecting tokenization strategies that align with task-specific requirements. Additionally, our experiments underscore the value of preprocessing machine code tailored to the context of the downstream task. The optimal preprocessing strategy, however, is highly task-dependent and requires careful consideration.

Since NLMs were not originally designed for binary analysis tasks, our findings provide valuable insights into how binary code can be effectively represented for such models. This representation step is essential and has a large potential impact on the model's performance. Overall, this study establishes a foundational understanding of selecting appropriate tokenization and preprocessing strategies for leveraging NLMs in binary code analysis tasks.

In future work, we aim to expand our investigation by applying tokenizers to larger datasets that introduce greater structural diversity and dependency varieties in machine code. This will allow us to better understand how tokenization approaches perform with more complex data representations. Additionally, we plan to explore alternative tokenization methodologies, such as comparing the SentencePiece implementation with Hugging Face's implementation, to identify nuances in their impact on model performance. Furthermore, we intend to scale our evaluations to larger language models exceeding 1 billion parameters, enabling us to assess how tokenizer choices influence performance in more powerful architectures. Finally, we will broaden our evaluation scope by testing tokenizers on a wider range of real-world downstream tasks, ensuring the practical relevance of our findings.

Acknowledgment

This manuscript has been assigned LA-UR-25-20366. This research was funded by the Nuclear Weapons Cyber Assurance Laboratory (NWCAL) at Los Alamos National Laboratory. The authors gratefully acknowledge the support provided, which made this work possible.

References

[1] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, K. Erk and N. A. Smith, Eds. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://aclanthology.org/P16-1162

[2] Y. Gu, H. Shu, and F. Hu, "Uniasm: Binary code similarity detection without fine-tuning," *arXiv preprint arXiv:2211.01144*, 2022.

[3] H. Gao, S. Cheng, Y. Xue, and W. Zhang, "A lightweight framework for function name reassignment based on large-scale stripped binaries," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 607–619.

[4] M. Ali, M. Fromm, K. Thellmann, R. Rutmann, M. Lübbering, J. Leveling, K. Klug, J. Ebert, N. Doll, J. Buschhoff, C. Jain, A. Weber, L. Jurkschat, H. Abdelwahab, C. John, P. Ortiz Suarez, M. Ostendorff, S. Weinbach, R. Sifa, S. Kesselheim, and N. Flores-Herr, "Tokenizer choice for LLM training: Negligible or crucial?" in *Findings of the Association for Computational Linguistics: NAACL 2024*, K. Duh, H. Gomez, and S. Bethard, Eds. Mexico City, Mexico: Association for Computational Linguistics, Jun. 2024, pp. 3907–3924. [Online]. Available: https://aclanthology.org/2024.findings-naacl.247

[5] G. Dagan, G. Synnaeve, and B. Roziere, "Getting the most out of your tokenizer for pre-training and domain adaptation," *arXiv preprint arXiv:2402.01035*, 2024.

[6] T. Ye, L. Wu, T. Ma, X. Zhang, Y. Du, P. Liu, S. Ji, and W. Wang, "CP-BCS: Binary code summarization guided by control flow graph and pseudo code," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 14 740–14 752. [Online]. Available: https://aclanthology.org/2023.emnlp-main.911

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[8] M. Lewis, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[9] M. Schuster and K. Nakajima, "Japanese and korean voice search," in *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2012, pp. 5149–5152.

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

[11] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, I. Gurevych and Y. Miyao, Eds. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 66–75. [Online]. Available: https://aclanthology.org/P18-1007

[12] J. Escalada, T. Scully, and F. Ortin, "Improving type information inferred by decompilers with supervised machine learning," *arXiv preprint arXiv:2101.08116*, 2021.

[13] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "Typeminer: Recovering types in binary programs using machine learning," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 2019, pp. 288–308.

[14] L. Chen, Z. He, and B. Mao, "Cati: Context-assisted type inference from stripped binaries," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 88–98.

[15] H. Green, E. J. Schwartz, C. L. Goues, and B. Vasilescu, "Stride: Simple type recognition in decompiled executables," *arXiv preprint arXiv:2407.02733*, 2024.

[16] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[17] D. Lehmann and M. Pradel, "Finding the dwarf: recovering precise types from webassembly binaries," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 410–425.

[18] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1073–1085.

[19] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "Stateformer: fine-grained type recovery from binaries using generative state modeling," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 690–702.

[20] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "{DEEPVSA}: Facilitating value-set analysis with deep learning for postmortem program analysis," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1787–1804.

[21] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 99–116.

[22] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX security symposium (USENIX Security 15)*, 2015, pp. 611–626.

[23] A. Madsen and A. R. Johansen, "Neural arithmetic units," *arXiv preprint arXiv:2001.05016*, 2020.

[24] A. Thawani, J. Pujara, F. Ilievski, and P. Szekely, "Representing numbers in NLP: a survey and a vision," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds. Online: Association for Computational Linguistics, Jun. 2021, pp. 644–656. [Online]. Available: https://aclanthology.org/2021.naacl-main.53

[25] A. Thawani, J. Pujara, and F. Ilievski, "Numeracy enhances the literacy of language models," in *Proceedings of the 2021 conference on empirical methods in natural language processing*, 2021, pp. 6960–6967.

[26] C. Eagle and K. Nance, *The Ghidra Book: The Definitive Guide*. no starch press, 2020.

[27] J. Oliver, C. Cheng, and Y. Chen, "Tlsh–a locality sensitive hash," in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, 2013, pp. 7–13.

[28] T. Le Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilic, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, J. Tow, A. M. Rush, S. Biderman, A. Webson, P. S. Ammanamanchi, T. Wang, B. Sagot, N. Muennighoff, A. Villanova del Moral, O. Ruwase, R. Bawden, S. Bekman, A. McMillan-Major, I. Beltagy, H. Nguyen, L. Saulnier, S. Tan, P. O. Suarez, V. Sanh, H. Laurençon, Y. Jernite, J. Launay, M. Mitchell, C. Raffel, A. Gokaslan, A. Simhi, A. Soroa, A. F. Aji, A. Alfassy, A. Rogers, A. K. Nitzav, C. Xu, C. Mou, C. Emezue, C. Klamm, C. Leong, D. van Strien, D. I. Adelani, and et al., "Bloom: A 176b-parameter open-access multilingual language model," *CoRR*, vol. abs/2211.05100, 2022. [Online]. Available: https://arxiv.org/abs/2211.05100

[29] F. Stollenwerk, "Training and evaluation of a multilingual tokenizer for gpt-sw3," *arXiv preprint arXiv:2304.14780*, 2023.

[30] P. Rust, J. Pfeiffer, I. Vulić, S. Ruder, and I. Gurevych, "How good is your tokenizer? on the monolingual performance of multilingual language models," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Online: Association for Computational Linguistics, Aug. 2021, pp. 3118–3135. [Online]. Available: https://aclanthology.org/2021.acl-long.243

Appendix

A. Tokenizer Hyper-Parameters

The Hugging Face Tokenizer library was the primary tool for configuring the tokenizers' hyperparameters. To evaluate the impact of different tokenization strategies on downstream performance tasks involving disassembled code, we carefully

TABLE IV
HYPERPARAMETER CONFIGURATIONS OPTIONS FOR BPE, UNIGRAM, AND WORDPIECE TOKENIZERS IN THIS STUDY

Tokenizer	Hyper-Parameter	Value(s)
BPE	model_type	BPE
	normalization_rule	NFD, lowercase
	pre_tokenizer_type	ByteLevel
	add_prefix_spac	False
	use_regex	False
	trainer_type	BpeTrainer
	vocab_size	3K | 25K 35K | 128K
	post_processor_type	ByteLevel
	trim_offsets	False
	decoder_type	ByteLevel
	special_tokens	<unk>, </s>, <s>, [PAD], [MASK]
Unigram	model_type	Unigram
	normalization_rule	NFD, lowercase
	pre_tokenizer_type	ByteLevel
	add_prefix_spac	False
	use_regex	False
	trainer_type	UnigramTrainer
	vocab_size	3K | 25K 35K | 128K
	post_processor_type	ByteLevel
	trim_offsets	False
	decoder_type	ByteLevel
	special_tokens	<unk>, </s>, <s>, <cls>, <sep>, [PAD], [MASK]
WordPiece	model_type	WordPiece
	normalization_rule	NFD, lowercase
	pre_tokenizer_type	BertPreTokenizer
	trainer_type	WordPieceTrainer
	vocab_size	3K | 25K 35K | 128K
	decoder_type	WordPiece
	prefix	"##"
	special_tokens	[UNK], </s>, <s>, <cls>, <sep>, <nln>, [PAD], [MASK]

tailored the configurations for each tokenizer, systematically varying the vocabulary sizes, as presented in Table IV. Parameters not listed in Table IV were kept at their default values.

B. Model Architecture and Hyper-Parameters

In this study, we fine-tuned three models with distinct architectures and hyperparameter configurations, as described in the sections below, to comprehensively evaluate the impact of different tokenization strategies on downstream performance. The detailed model architecture and fine-tuning hyperparameters are presented in Table V, providing a clear overview of the experimental parameters.

1) Llama 3.2 Decoder-Only Model: The pre-trained Llama 3.2 model with 1B parameters represents the smallest architecture in the Llama 3 series, designed to provide efficient performance while minimizing computational overhead. As part of the Llama series, which excels in various NLP tasks such as text generation, summarization, and question-answering, the 1B model balances model complexity with resource efficiency. It employs a transformer-based architecture optimized for generative and comprehension tasks, leveraging a robust 128K vocabulary size for precise tokenization and language representation. Despite its smaller size, Llama 3.2 demonstrates impressive capabilities in handling tasks that require understanding complex language structures, making it an ideal choice for resource-constrained environments or domain-specific fine-tuning.

We leveraged the Llama 3.2 model with 1B parameters to fine-tune it for downstream tasks using tokenizers trained under various configurations. Our implementation followed guidelines from the Hugging Face's training repository deep-learning-pytorch-huggingface. This setup allowed us to systematically evaluate the impact of tokenization strategies on performance, particularly in tasks requiring the understanding of complex disassembled code structures.

2) BERT Encoder-Only Model: One of the reasons for using a pre-trained BERT-based model was to evaluate how a smaller, encoder-only transformer model performs on binary analysis tasks. However, for some experiments, modifications were necessary to adapt BERT to our specific requirements. The default BERT vocabulary size is 30,522, but we experimented with alternative vocabulary sizes, including 3,000, 25,000, and 35,000, requiring us to train custom tokenizers. Additionally, BERT's default configuration supports a maximum input token length of 512. To accommodate longer input sequences of up to 1,024 tokens, we extended the model's positional embeddings. Hugging Face hosts the specific BERT-based model variant we used in this experiment and can be found here.

3) BART Encoder-Decoder Model: BART's model architecture leverages the strengths of both encoder and decoder components in the Transformer model. It is designed for natural language processing tasks that combine bidirectional encoding and autoregressive decoding. One advantage of using BART over BERT is that BART can handle 1024 tokens by default, and no modification was needed for our experiment. We used the pre-trained BART-base model version, which Hugging Face hosts.

C. Intrinsic Evaluation of Tokenizers

In addition to examining vocabulary overlaps among different tokenizers within the same dataset type (either exclusively Default or exclusively Preprocessed) across four vocabulary sizes, we also investigated the vocabulary overlap between tokenizers applied to the Default and Preprocessed datasets as shown in Figure 4. This extended analysis provided insights into how preprocessing influences tokenization consistency

TABLE V
OVERVIEW OF THE ARCHITECTURE AND HYPERPARAMETER CONFIGURATIONS OF THE MODELS USED IN THE STUDY

Model	Hyper-Parameter	Value(s)
Llama 3.2	Hidden Size	2048
	Number of Attention Heads	32
	Number of Hidden Layers	16
	Context Window	1024
	Learning rate	0.0002
	Learning rate scheduler	linear
	Gradient accumulation steps	2
	Optimizer	adamw_torch
	Max. gradient norm.	0.3
	Warmup ratio	0.03
	Precision	tf32
BERT	Hidden Size	384
	Number of Attention Heads	12
	Number of Hidden Layers	12
	Max Position Embeddings	512
	Learning rate	0.0003
	Learning rate schedular	linear schedule with warmup
	Weight decay	0.0001
	Gradient accumulation steps	1
	Optimizer	adamw_torch
BART	Hidden layers	6
	Number of Encoder Attention Heads	12
	Number of Decoder Attention Heads	12
	Number of Hidden Layers	12
	Max Position Embeddings	1024
	Decoder layers	6
	Encoder layers	6
	Learning rate	0.00005
	Learning rate decay	Constant
	Gradient accumulation steps	1
	Optimizer	adamw_torch

and vocabulary alignment across datasets and tokenization algorithms.

The vocabulary overlap between tokenizers is notably low, especially across different datasets and tokenization algorithms. The overlap decreases as the vocabulary size increases, highlighting the divergence in subword segmentation strategies used by BPE, Unigram, and WordPiece tokenizers. Preprocessing the dataset moderately improves overlap within the same tokenizer type but does not significantly increase overlap between different tokenizers, demonstrating that tokenization algorithms inherently yield distinct vocabulary sets.

D. Illustrative Examples for the Tokenization Behavior

Tables VI and VII show the tokenization results for five representative disassembled instructions from a binary function presented in two formats:

Default disassembly:

```
"ENDBR64\nCMP EDI,ESI\nJGE
0x000012ce\nPUSH R13\nMOV R8D,EDI"
```

Preprocessed_disassembly:

```
"ENDBR64\nCMP EDI,ESI\nJGE addr14\nPUSH
R13\nMOV R8D,EDI"
```

The default disassembly retains original numeric values, including memory addresses, while the preprocessed version replaces memory addresses with human-readable sequential identifiers like addr14. The tokenization results were evaluated across three tokenizers (BPE, Unigram, and WordPiece) and the 25K vocabulary size. The tokenization behavior of the default disassembly is presented in Table VI while the tokenization behavior of the preprocessed disassembly is presented in Table VII.

Table VIII shows the tokenization results for an entire disassembled function instructions using the BPE tokenizer with varying vocabulary sizes. The disassembly is presented in the preprocessed format:

Preprocessed disassembly:

```
"ENDBR64\nPUSH RBP\nMOV RBP,RDI\nMOV
RDI,RSI\nPUSH RBX\nMOV RBX,RSI\nSUB
RSP,8\nCALL addr0\nTEST EAX,EAX\nJLE
```

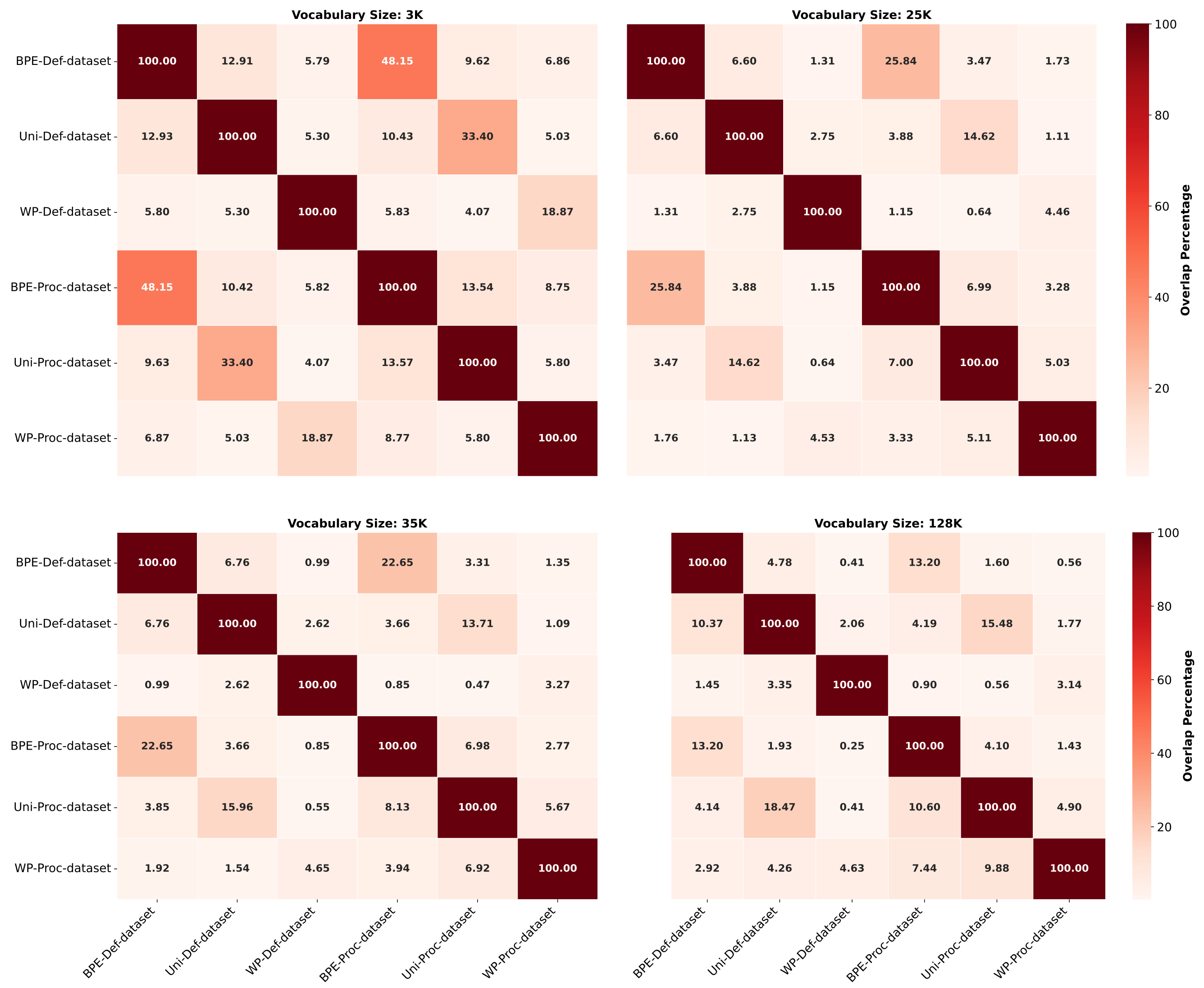


Fig. 4. Vocabulary overlap heatmaps for tokenizers across default and preprocessed datasets at four vocabulary sizes: 3K, 25K, 35K, and 128K.

```
addr5\nSUB EAX,1\nMOVZX R9D,word ptr
[addr8]\nMOV RSI,RBX\nXOR EDX,EDX\nMOVZX
R8D,word ptr [addr7]\nLEA RDI,[RBX + RAX*1
+ 1]\nJMP addr3\nCMP CL,10\nJNZ addr6\nMOV
word ptr [RAX],R8W\nADD EDX,2\nADD
RSI,1\nCMP RDI,RSI\nJZ addr4\nMOVZX
ECX,byte ptr [RSI]\nMOVSXD RAX,EDX\nADD
RAX,RBP\nCMP CL,9\nJNZ addr1\nADD
RSI,1\nMOV word ptr [RAX],R9W\nADD
EDX,2\nCMP RDI,RSI\nJNZ addr3\nMOVSXD
RDX,EDX\nADD RBP,RDX\nMOV byte ptr
[RBP],0\nADD RSP,8\nPOP RBX\nPOP
RBP\nRET\nMOV byte ptr [RAX],CL\nADD
EDX,1\nJMP addr2\n"
```

The results in Table VIII depict an example of the tokenization behavior of the BPE tokenizer with varying vocabulary sizes: 3K, 25K, 35K, and 128K. It is notable that the number of tokens generated decreases as the tokenizer's vocabulary size increases. This behavior is inherent to the design of the BPE algorithm, which creates its vocabulary by iteratively merging pairs of frequently co-occurring sub-tokens in the dataset. With a smaller vocabulary size, the tokenizer splits words or instructions into smaller sub-tokens to fit within the limited dictionary. As the vocabulary size increases, the BPE algorithm incorporates more frequent sub-token pairs into its dictionary, allowing it to merge smaller sub-tokens into longer, semantically meaningful tokens. Consequently, this leads to fewer overall tokens being generated for the same input text, as larger vocabulary sizes provide more complete token representations. This process highlights the efficiency of the BPE algorithm in balancing granularity and compression based on the frequency of token occurrences in the dataset and the constraints of the vocabulary size. The color-coded example in Table VIII visually demonstrates how these changes manifest in tokenizing the same disassembled function.

TABLE VI
TOKENIZATION EXAMPLE FOR FIVE DEFAULT DISASSEMBLED INSTRUCTIONS

Tokenizer	Tokens	# of Tokens
BPE-25K	"endbr64\n","cmp edi,esi\n",**"jge 0x000012"**,**"ce\n"**,"push r13\n","mov r8d,","ed","i"	**8**
Unigram-25K	"e","nd","br","64","\n","cmp edi,","esi","\n",**"jge 0x000012c"**,**"e"**,"\n", "push r1","3","\n","mov r8d,","edi"	**16**
WordPiece-25K	"endbr64","cmp","edi",",","esi",**"jge"**,**"0x000012ce"**,"push","r13","mov","r8d",",","edi"	**13**

TABLE VII
TOKENIZATION EXAMPLE FOR FIVE PREPROCESSED DISASSEMBLED INSTRUCTIONS

Tokenizer	Tokens	# of Tokens
BPE-25K	"endbr64\n","cmp edi,esi\n",**"jge addr14\n"**,"push r13\n","mov r8d,","ed","i"	**7**
Unigram-25K	"e","nd","b","r64","\n","cmp edi,","esi","\n",**"jge addr1"**,**"4"**,"\n", "push r1","3","\n","mov r8d,","edi"	**16**
WordPiece-25K	"endbr64","cmp","edi",",","esi",**"jge"**,**"addr14"**,"push","r13","mov","r8d",",","edi"	**13**

TABLE VIII
TOKENIZATION EXAMPLE OF AN ENTIRE PREPROCESSED DISASSEMBLED FUNCTION INSTRUCTIONS

Tokenizer	Tokens	# of Tokens
BPE-3K	"endbr64\n", "push rbp\n", "mov rbp,rdi\n", "mov rdi,rsi\n", "push rbx\n", "mov rbx,rsi\n", "sub rsp,8\n", "call addr0\n", "test eax,eax\n", "jle addr5\n", "sub eax,1\n", "movzx r9d,", "word ptr [addr", "8]\n", "mov rsi,rbx\n", "xor edx,edx\n", "movzx", " r8d,", "word ptr [addr", "7]\n", "lea rdi,[rbx", " + rax*1 + 1]\n", "jmp addr3\n", "cmp cl,", "10\n", "jnz addr6\n", "mov word ptr [r", "ax],", "r8", "w\n", "add edx,", "2\n", "add rsi,1\n", "cmp rdi,", "rsi\n", "jz addr4\n", "movzx ecx,byte ptr [r", "si]\n", "movsxd rax,edx\n", "add rax,", "rbp\n", "cmp cl,", "9\n", "jnz addr1\n", "add rsi,1\n", "mov word ptr [r", "ax],", "r9", "w\n", "add edx,", "2\n", "cmp rdi,", "rsi\n", "jnz addr3\n", "movsxd rdx,edx\n", "add rbp,", "rdx\n", "mov byte ptr [rbp", "],0\n", "add rsp,8\n", "pop rbx\n", "pop rbp\n", "ret\n", "mov byte ptr [rax],", "cl\n", "add edx,1\n", "jmp addr2\n"	**67**
BPE-25K	"endbr64\n", "push rbp\n", "mov rbp,rdi\n", "mov rdi,rsi\n", "push rbx\n", "mov rbx,rsi\n", "sub rsp,8\n", "call addr0\n", "test eax,eax\n", "jle addr5\n", "sub eax,1\n", "movzx r9d,", "word ptr [addr8]\n", "mov rsi,rbx\n", "xor edx,edx\n", "movzx r8d,", "word ptr [addr7]\n","lea rdi,[rbx", " + rax*1 + 1]\n", "jmp addr3\n", "cmp cl,10\n", "jnz addr6\n", "mov word ptr [rax],", "r8w\n", "add edx,2\n", "add rsi,1\n", "cmp rdi,rsi\n", "jz addr4\n", "movzx ecx,byte ptr [rsi]\n", "movsxd rax,edx\n", "add rax,rbp\n", "cmp cl,9\n", "jnz addr1\n", "add rsi,1\n", "mov word ptr [rax],", "r9w\n", "add edx,2\n", "cmp rdi,rsi\n", "jnz addr3\n", "movsxd rdx,edx\n", "add rbp,rdx\n", "mov byte ptr [rbp],0\n", "add rsp,8\n", "pop rbx\n", "pop rbp\n", "ret\n", "mov byte ptr [rax],cl\n", "add edx,1\n", "jmp addr2\n"	**49**
BPE-35K	"endbr64\n", "push rbp\n", "mov rbp,rdi\n", "mov rdi,rsi\n", "push rbx\n", "mov rbx,rsi\n", "sub rsp,8\n", "call addr0\n", "test eax,eax\n", "jle addr5\n", "sub eax,1\n", "movzx r9d,", "word ptr [addr8]\n", "mov rsi,rbx\n", "xor edx,edx\n", "movzx r8d,word ptr [addr7]\n", "lea rdi,[rbx + rax*1 + 1]\n", "jmp addr3\n", "cmp cl,10\n", "jnz addr6\n", "mov word ptr [rax],r8w\n", "add edx,2\n", "add rsi,1\n", "cmp rdi,rsi\n", "jz addr4\n", "movzx ecx,byte ptr [rsi]\n", "movsxd rax,edx\n", "add rax,rbp\n", "cmp cl,9\n", "jnz addr1\n", "add rsi,1\n", "mov word ptr [rax],", "r9w\n", "add edx,2\n", "cmp rdi,rsi\n", "jnz addr3\n", "movsxd rdx,edx\n", "add rbp,rdx\n", "mov byte ptr [rbp],0\n", "add rsp,8\n", "pop rbx\n", "pop rbp\n", "ret\n", "mov byte ptr [rax],cl\n", "add edx,1\n", "jmp addr2\n"	**46**
BPE-128K	"endbr64\n", "push rbp\n", "mov rbp,rdi\n", "mov rdi,rsi\n", "push rbx\n", "mov rbx,rsi\n", "sub rsp,8\n", "call addr0\n", "test eax,eax\n", "jle addr5\n", "sub eax,1\n", "movzx r9d,word ptr [addr8]\n", "mov rsi,rbx\n", "xor edx,edx\n", "movzx r8d,word ptr [addr7]\n", "lea rdi,[rbx + rax*1 + 1]\n", "jmp addr3\n", "cmp cl,10\n", "jnz addr6\n", "mov word ptr [rax],r8w\n", "add edx,2\n", "add rsi,1\n", "cmp rdi,rsi\n", "jz addr4\n", "movzx ecx,byte ptr [rsi]\n", "movsxd rax,edx\n", "add rax,rbp\n", "cmp cl,9\n", "jnz addr1\n", "add rsi,1\n", "mov word ptr [rax],r9w\n", "add edx,2\n", "cmp rdi,rsi\n", "jnz addr3\n", "movsxd rdx,edx\n", "add rbp,rdx\n", "mov byte ptr [rbp],0\n", "add rsp,8\n", "pop rbx\n", "pop rbp\n", "ret\n", "mov byte ptr [rax],cl\n", "add edx,1\n", "jmp addr2\n"	**44**