Appendices

Appendix A. Data Preprocessing

We made several modifications to the C functions to enhance the code LLM models' learning efficiency. First, as illustrated in Figure 1, lines containing only a curly bracket "{" or "}" were modified. These standalone brackets were relocated to the preceding line, as depicted in the figure.

A.0.1. Duplicate Lines of Code. To address the issue of duplicate lines of code as discussed in Section background:Challenges, we ensured that each line in the code is unique. More formally: Let F represent a set consisting of function elements, where each function is denoted as f. Let $f_i \in F$ be function i in the set. Let L_i (L for line) be a set that contains all the instances of duplicate lines in function f_i. Each element $L_{i,j,k} \in L_i$ is interpreted as follows:

- i - Index of the function.
- j - Set identifier. This index is an incrementing integer that identifies a set of lines with the same content. Each unique line in the code gets a distinct j value. For example, in cases where the code has several appearances of the same line: "x = 3"; all of these appearances are grouped under one j value. Therefore, j is in range [1, number of unique replicated lines in function i].
- k - Appearance identifier. In each set identified by j, k specifies the appearance order of a particular line. It starts at one and is incremented for each subsequent appearance of a line within the same set.

A.0.2. Replicated Line Handling Procedure. For a given function f_i with j unique replicated lines and k appearances in set j, we add spaces at the end of each line. In the first line in each duplicate set j, denoted as $L_{i,j,1}$, we make no changes. However, for all the subsequent lines in the set j, labeled $L_{i,j,k}$ where $k > 1$, we add k-1 spaces at the end of the line. This approach ensures that every line in the function f_i is unique, as illustrated in Figure 2. Consequently, when the *Localization* model identifies a specific line of code for modification, there is no ambiguity regarding which line is being referred to.

A.0.3. Reasons for Ensuring the Uniqueness of Lines in this Way. Initially, we observed that adding a unique symbol at the end of each line, rather than at the beginning, would enhance the models' learning. This decision was based on the inner workings of the CodeQwen1.5 model as a transformer model. During inference, the model predicts each token based on the context of the preceding tokens. If the model is required to select a specific line of code as its first predicted token, it does so based on a limited context. In contrast, by allowing the model to generate an entire line of code before selecting the unique line, we hypothesize that it makes more informed and accurate selections. Therefore, we developed a strategy to ensure the uniqueness of each line by (1) adding the least amount of tokens by virtue of adding spaces to replicated lines (the CodeQwen1.5 tokenizer encodes one to 20 continuous spaces as a single token, with different quantities of spaces mapping to distinct tokens, thereby minimizing the number of tokens added to the original code; an extra token was only added to duplicated lines within the function – not to every line); and (2) selecting tokens that do not affect C code syntax (adding spaces at the end of a line in C code does not alter its meaning or create compilation issues). This approach took into account the models' pretraining on programming languages.

Appendix B. Technical Aspects of Fine-Tuning

All models (*Localization*, *Modification*, and *Vulnerability Generation Instructions*) were fine-tuned using LoRA. This technique performs training exclusively on additional weights and introduces them as extra weight matrices known as adapters [**?**] into certain layers of the model. This approach enables the fine-tuning of very large models by adjusting a relatively small number of weights. Additionally, we utilized the Accelerate library from Hugging Face, which incorporates DeepSpeed for efficient model training across multiple GPUs. This setup leverages the zero redundancy optimizer (ZeRO), a memory optimization technique designed for large-scale distributed deep learning. Due to GPU RAM constraints, we employed a 16-bit representation for all models. Inspired by, we also experimented with using the CodeQwen1.5 model with an 8-bit representation, however this configuration led to decreased performance in all models. Consequently, we chose to use the CodeQwen1.5 model with the 16-bit representation. Our experiments and the fine-tuning process utilized a computer with a 64-core AMD EPYC 7343 (3.2 GHz) CPU and 256 GB of DDR memory, along with two NVIDIA RTX A6000 GPUs; we note, however, that a single GPU is sufficient for our operations.

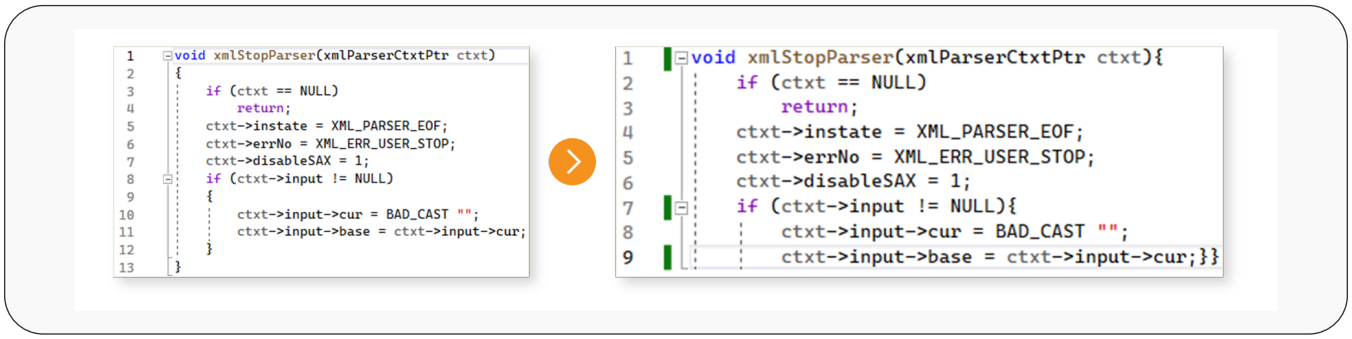

Figure 1: Preprocessing - curly brackets.

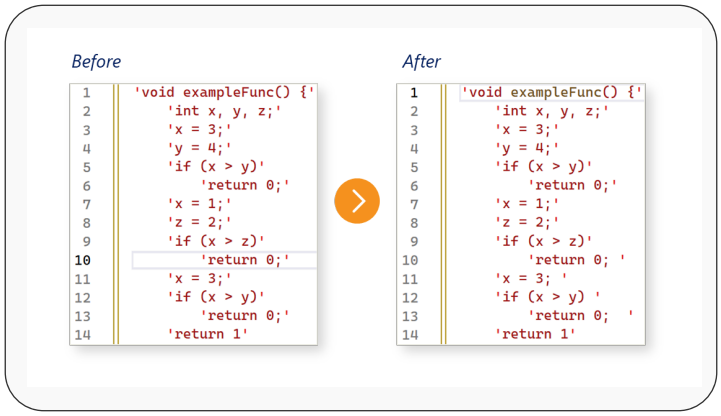

Figure 2: Preprocessing - adding spaces.