Virtual Compiler Is All You Need For Assembly Code Search

Zeyu Gao[1], Hao Wang[1], Yuanda Wang[2], and Chao Zhang[1]*
[1]Tsinghua University
[2]Beijing University of Posts and Telecommunications
{gaozy22,hao-wang20}@mails.tsinghua.edu.cn
wangyuanda@bupt.edu.cn
chaoz@tsinghua.edu.cn

Abstract

Assembly code search is vital for reducing the burden on reverse engineers, allowing them to quickly identify specific functions using natural language within vast binary programs. Despite its significance, this critical task is impeded by the complexities involved in building high-quality datasets. This paper explores training a Large Language Model (LLM) to emulate a general compiler. By leveraging Ubuntu packages to compile a dataset of 20 billion tokens, we further continue pre-train CodeLlama as a Virtual Compiler (ViC), capable of **compiling any source code of any language to assembly code**. This approach allows for *virtual compilation* across a wide range of programming languages without the need for a real compiler, preserving semantic equivalency and expanding the possibilities for assembly code dataset construction. Furthermore, we use ViC to construct a sufficiently large dataset for assembly code search. Employing this extensive dataset, we achieve a substantial improvement in assembly code search performance, with our model surpassing the leading baseline by 26%.

1 Introduction

Reverse engineering frequently entails the challenging task of swiftly locating specific functions within extensive binary files, such as those associated with malicious behavior. Traditionally, reverse engineers rely on the tedious approach of searching for unique strings (Hex-rays, 2023a) or constants (polymorf, 2020) to locate these functions. This practice often leads to inefficiency as it heavily relies on experience or heuristic algorithms, frequently consuming considerable time.

We observe that the specific search requirements articulated by reverse engineers could often be distilled into natural language descriptions. This realization sparked the idea of employing a code search (Lu et al., 2021; Feng et al., 2020) perspective to refine the extant methodologies. Code search has achieved substantial success in high-level programming languages, garnering wide commercial application (GitHub, 2018). Assembly code search enables users to search the assembly (i.e. assembly code) function with natural language, enhancing user interaction with binary executable files through natural language.

The construction of assembly code search datasets presents considerable difficulty. Unlike high-level programming languages where datasets can be readily created by parsing the docstring of function, constructing an assembly code search dataset entails an obligatory compilation step from source code to assembly code. It is exemplified by the work conducted in Nova$^+$ (Jiang et al., 2023). Despite attempts to compile over 4 million C programs and 1 million C++ programs, only 32,774 C programs and 40,087 C++ programs were successfully compiled. Numerous compilation failures stem from the complex compilation environment, including diverse dependencies and variations in optimization levels and compilers, greatly complicating dataset construction. Previous efforts target the compilation challenges of single-source C functions through type inference or neural compilation (da Silva et al., 2021; Guo and Moses, 2022; Armengol-Estapé and O'Boyle, 2021). Yet, the practical utility of these advancements is constrained by the scarce availability of code search datasets for C language.

Inspired by Meta's use of Large Language Models (LLMs) for compiler optimization (Cummins et al., 2023), we postulate a novel modeling paradigm—let LLMs emulate a general compiler and understand the comprehensive compilation process, including optimization options and assembly code generation. This method not only enables us to utilize the previous code search dataset in C/C++ language like CCSD (Liu et al., 2021) but also

[*] corresponding author

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 3040–3051
August 11-16, 2024 ©2024 Association for Computational Linguistics

demonstrates the capability to generalize across languages such as Python and Golang, despite only being trained on C/C++ source and assembly code pairs. This flexibility suggests the potential of applying extensive datasets from prior works, like CodeSearchNet (Husain et al., 2020), to the realm of assembly code search, addressing the historical constraints of dataset scarcity in this field.

Addressing the challenges in assembly code search and the shortcomings of current methodologies, our work makes the following contributions:

- **Introduction of the virtual compiler** (`ViC`). We introduce a novel approach, `ViC` for creating an assembly code search dataset. By employing the virtual compiler, we generate a diverse and robust dataset that circumvents the traditional barriers related to compilation challenges, vastly enriching the resources available for assembly code search tasks.

- **Enhanced assembly code search performance**. We constructed a high-quality assembly code dataset using a virtual compiler, and the model trained on this dataset achieved a 26% performance improvement in assembly code search tasks over existing state-of-the-art (SOTA) solutions.

- We release our models and datasets to facilitate future research[1].

2 Background and Related Works

2.1 Assembly Code Analysis

The process of transforming high-level, human-readable source code (such as C or C++) into assembly code (also known as binary code) that a CPU can execute directly is known as compilation. Figure 1 shows the source code and the corresponding assembly code for the bubble sort algorithm. It clearly illustrates that throughout compilation, much of the original context and structure, including variable names and high-level logical structures like loops and conditional statements, are lost. This loss makes understanding compiled binary challenging, which is crucial for identifying security vulnerabilities and patching bugs, necessitating binary code analysis.

Binary reverse engineering is a commonly used methodology in binary code analysis. It entails analyzing binary files without the source code to reconstruct the function functionality and partial program logic. In this complex process, one of the challenges is locating the specific function with certain functionality, such as encryption and authentication. The common methods include searching for representative strings (Hex-rays, 2023a), analyzing function import and export, looking for pattern matching (polymorf, 2020), static analysis and dynamic debugging. These methods, while useful, can be time-consuming and imprecise. To overcome these limitations, the development of assembly code search models marks a significant advancement, offering a more efficient and accurate method of navigating binary files.

```
void bubble_sort(int arr[], int n)
{
  int i, j;
  for (i = 0; i < n - 1; i++) {
    for (j = 0; j < n - i - 1; j++){
      if (arr[j] > arr[j + 1]) {
        int temp = arr[j];
        arr[j + 1] = arr[j];
        arr[j] = temp;
      }
    }
  }
}
```

(a) Source code

```
bubble_sort:
    cmp esi, 2
    jl .LBB0_8
    lea eax, [rsi - 1]
    xor ecx, ecx
    mov edx, eax
.LBB0_2:
    mov edx, edx
    mov r8d, ecx
    not r8d
    add r8d, esi
    test r8d, r8d
    jle .LBB0_7
    mov r8d, dword ptr [rdi]
    xor r9d, r9d
.LBB0_4:
    lea r10, [r9 + 1]
    mov r11d,dword ptr[rdi+4*r9+4]
    cmp r8d, r11d
    jle .LBB0_6
    mov dword ptr[rdi+4*r9+4],r8d
    mov r11d, r8d
.LBB0_6:
    mov r8d, r11d
    mov r9, r10
    cmp rdx, r10
    jne .LBB0_4
.LBB0_7:
    inc ecx
    dec edx
    cmp ecx, eax
    jne .LBB0_2
.LBB0_8:
    ret
```

(b) Assembly code

Figure 1: C source code and the compiled assembly code for a bubble sort algorithm.

2.2 Assembly Code Modeling

With high-level logical structure replaced by the bare jump instructions, understanding and modeling the assembly code has quite different challenges compared to the high-level language. To effectively address these challenges, researchers have developed various techniques to model the assembly code for the downstream tasks such as binary code similarity detection (BCSD). This technique is pivotal for assessing the similarity between different assembly codes which can be affected by varying aspects like compiler versions and optimization levels and identifying the paired assembly code compiled from the same source code from an amount of candidates, useful for clone detection and supply chain analysis. CodeCMR (Yu et al., 2020) integrates GNN, DPCNN, and LSTM to extract the semantic features of source code and binary code. PalmTree (Li et al., 2021) pre-trains on unlabeled binary corpus through self-supervised training to capture various characteristics of assem-

[1] https://github.com/zeyugao/VirtualCompiler

bly languages. Trex (Pei et al., 2021) models micro traces with Transformers, while jTrans (Wang et al., 2022) uses unique jump-aware representation method preserves control flow information. These models, especially the Transformer-based models, have a partial ability to capture the semantics in the function to support the code search task.

2.3 Code Search

Code search plays a pivotal role in software development, enabling developers to query and retrieve valuable code snippets. This process leverages either pattern matching or, more recently, natural language queries, thanks to advancements in semantic code search. The most code search works hinges on a pipeline that begins with pre-training models on extensive corpora consisting of unpaired code and natural language and accompanies by fine-tuning the model on specialized code search datasets, which sharpens their ability to understand and execute code search queries accurately (Wang et al., 2023, 2021; Lu et al., 2021; Feng et al., 2020; Guo et al., 2021).

3 Overview

We present our workflow in Figure 2, with three phases to advance assembly code search capabilities. Initially, we compile a vast collection of packages from Ubuntu to obtain a mapping between source and assembly code. Following this, we train the CodeLlama model on this massive dataset to emulate compiler behavior (Section 4). Lastly, leveraging this model, we perform virtual compilations on existing code search datasets to produce an augmented assembly code search dataset. This enriched dataset then informs the training of an assembly code encoder, yielding a refined model for effective assembly code search (Section 5).

4 Virtual Compiler

4.1 Code Dataset Construction

In our pursuit to train a model to compile the source code of the individual function to its corresponding assembly code, we compile over 6,000 C/C++ packages from Ubuntu using two types of x86-64 compilers, GCC and Clang, each in three different versions (`GCC-{7,9,11}` and `Clang-{9,11,12}`) and five optimization levels (`O{0-3}` and `Os`). Then, we extract the source and assembly code for the model training. The model exclusively provides the source code of the function body and is required to predict the corresponding assembly code. To reduce the hallucinations of the model, we exclude assembly code containing inline functions due to compiling optimization to ensure a one-to-one correspondence between the source code and assembly. Additionally, we omit functions shorter than 5 lines. These often represent class constructors, which will generate disproportionately large volumes of assembly code. However, for simplicity in source code processing, global variables, and macro definitions are not provided to the model.

Ultimately, we amass a dataset consisting of 15 million source-to-assembly pairs, exceeding 20 billion tokens. We filter out source functions with fewer than four lines in the function body, remove function comments, and prefer unmangled function names in the assembly code. A validation set of 1,000 unseen function source codes paired with corresponding assembly code is segregated to assess the output quality.

4.2 Model Training

Contrasting with Meta's work (Cummins et al., 2023) that begins with random initialized weight, we employ Codellama 34B (Rozière et al., 2023) as our foundational model, anticipating a more rapid convergence (Chen et al., 2021). Subsequent experiment shows that leveraging Codellama also provides enhanced comprehension of various programming languages. The model is trained using Supervised Fine-Tuning (SFT). Besides the function source code, the prompt includes the compiler, optimization level, and whether the assembly code should be stripped, denoting the removal of symbol information, such as local variable names.

Considering the length of the vast majority of inputs in the CodeSearchNet, we opt for a 4,096 context length during training instead of the 16,384 natively supported by CodeLlama, aiming for a faster training process. We initially focus on training sets with less than 2,048 tokens, followed by a training mix that includes a 3:1 ratio of 0-2,048 and 2,048-4,096 token-length data for better training efficiency. The model is trained on 64 NVIDIA A100 GPUs, exceeding a total of 1,000 GPU days, employing a cosine learning rate schedule with 1% of warm-up steps. The initial phase operates at a peak learning rate of 3e-5 with a batch size of 3,072, while the subsequent phase uses a peak learning rate of 1.5e-5 and a batch size of 2,048. Each dataset underwent a single epoch.

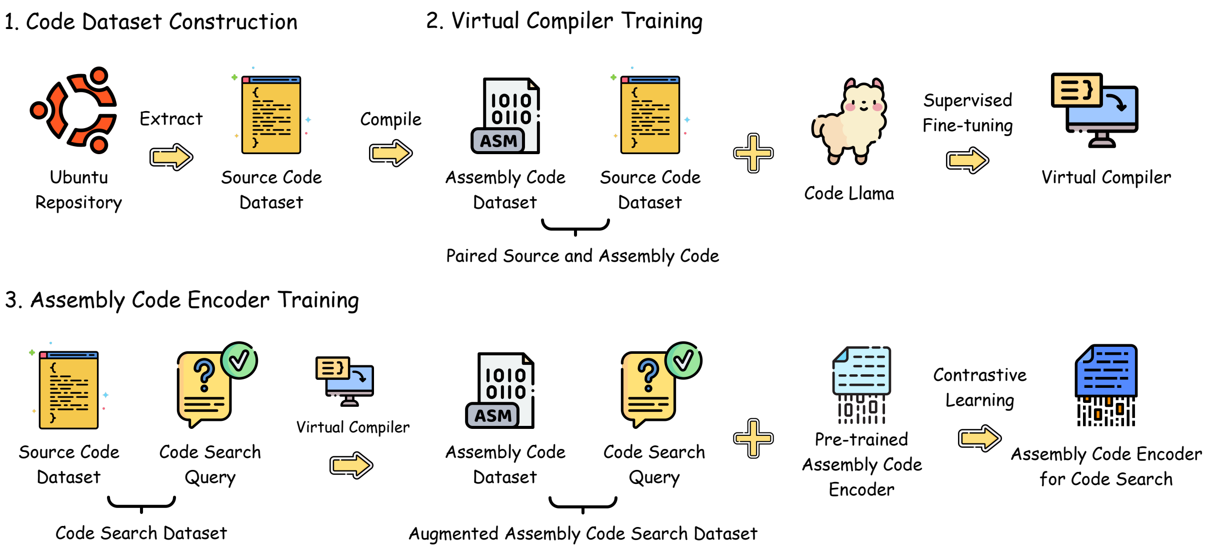


Figure 2: The workflow overview of using ViC for assembly code search.

5 Code Search Contrastive Learning

5.1 Dataset

We use two datasets to conduct the code search contrastive learning. The first dataset we utilize is CCSD (Liu et al., 2021), which comprises function summarization from over 95,000 functions from 300 different projects with an extensive deduplication and cleaning process. Furthermore, we apply virtual compilation to CodeSearchNet (Husain et al., 2020), a challenge introduced by GitHub featuring a dataset and a corresponding benchmark. This dataset is the bedrock of contemporary code search research nowadays. CodeSearchNet spans multiple programming languages — Python, JavaScript, Ruby, Go, Java, and PHP — none of which are directly applicable to assembly code search due to their higher-level nature.

In constructing our virtual assembly dataset for model training, each function from CCSD and CodeSearchNet is subjected to a randomly chosen combination of compiler and optimization levels, thus generating their virtual assembly codes. This approach allows us to expand the applications of these comprehensive, multi-language datasets into the realm of assembly code search.

5.2 Model Architecture

Building on the revelations from jTrans (Wang et al., 2022), we acknowledge the need for specialized treatment of textual and assembly code representations. To this end, we decouple the text encoder and assembly code encoder, adopting a bespoke architecture for the assembly code encoder, allowing each encoder to become adept at handling the intricacies of its respective data modality.

For the assembly code encoder, we use a roformer-base (Su et al., 2022) model that incorporates shared parameters (Wang et al., 2022) with 110M parameters to better integrate the inductive bias of control flow in assembly code. And we keep string literals in assembly code to better preserve semantic information through Byte-Pair Encoding instead of normalizing them, which is a common simplification in previous assembly modeling work (Li et al., 2021; Pei et al., 2021; Wang et al., 2022). On the other side, the text encoder is initialized by the well-established sentence-transformers (Reimers and Gurevych, 2019).

5.3 Assembly Code Encoder Training

During the training of the assembly code encoder, we employ the commonly used in-batch negative sampling strategy and use InfoNCE loss as the training loss, shown as Equation 1 and 2. This method aims to increase the similarity between positive pairs within a batch while reducing the similarity across negative pairs. The text encoder is initialized from sentence-transformers, and the assembly encoder is pre-trained with MLM (Masked Language Model) and JTP (Jump Target Prediction) tasks. With the text encoder well-established, we set the learning rate to 1e-6 to reduce the disturbance, and the learning rate for assembly code is set to 2e-5 empirically (Wang et al., 2022). We take the model with the best in-dataset evaluation result for further evaluation.

Give a batch of positive assembly code and text pairs $B = \{(a_1, t_1,), (a_2, t_2), \cdots, (a_n, t_n)\}$. We treat each pair (a_i, t_i) as positive pair, and $(a_i, i_j)_{i \neq j}$ as negative pairs. The text-to-assembly contrast loss is defined as:

$$L_1 = -\frac{1}{n} \sum_{i=0}^{n} \log \frac{\exp(t_i \cdot a_i / T)}{\sum_{j=1}^{N} \exp(t_i \cdot a_j / T)} \quad (1)$$

The auxiliary loss, assembly to text contrast loss, is inversely defined as:

$$L_2 = -\frac{1}{n} \sum_{i=0}^{n} \log \frac{\exp(a_i \cdot t_i / T)}{\sum_{j=1}^{N} \exp(a_i \cdot t_j / T)} \quad (2)$$

where T is the temperature, which we set to 0.07 empirically (Radford et al., 2021; He et al., 2020).

Then the train loss is defined as $L = L_1 + L_2$.

6 Evaluation

6.1 Evaluation Setup

We implement ViC using Pytorch 2.1 (Paszke et al., 2019). We use IDA Pro 8.3 (Hex-rays, 2023b) to disassemble and extract the functions from the binary executable file in all of the experiments. Our training and experiments are conducted on several servers to accelerate training. The CPU setup is 128 cores with 2TB RAM for each server. The total GPU setup is 32 NVIDIA Tesla A100.

6.2 Evaluation of Virtual Compiler

We endeavor to appraise the quality of assembly code yielded by the virtual compiler, with an emphasis on the model's capability to generate assembly code for augmenting the code search dataset.

We employ two distinct approaches for evaluating the model's capability. For code with ground truth, namely, C/C++ source code with corresponding assembly code, we assess quality using various similarity metrics in Section 6.2.1-6.2.3. For high-level languages without direct assembly code ground truth, such as PHP and Golang from the CodeSearchNet dataset, we use the downstream task in Section 6.3 and a case study in Section A to demonstrate the model's capabilities.

During the dataset construction phase, we preserve 1,000 C/C++ functions with their source code and assembly code that the model has not previously trained in the training process for evaluation.

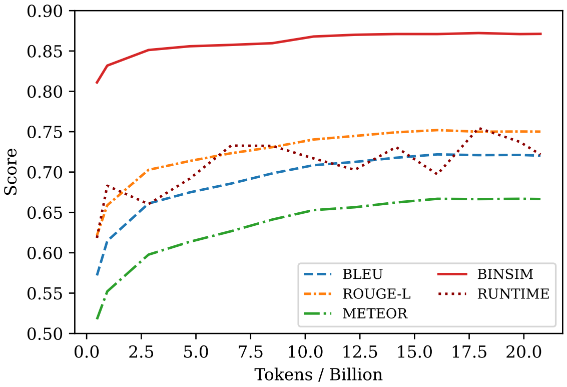


Figure 3: Correlation between the number of tokens used in model training and the quality of generated assembly code, as evaluated by various metrics

6.2.1 Sequence Similarity

A concise evaluation of the generated assembly code is carried out using the BLEU (Papineni et al., 2002), ROUGE-L (Lin, 2004), and METEOR score (Banerjee and Lavie, 2005), which are employed to show the similarity between generated assembly code and the reference ground truth. The scores are shown in Figure 3, revealing a clear trend in the data that as the number of tokens processed during model training increased, there is a commensurate rise in the scores for each metric.

6.2.2 Runtime Similarity

We further study the runtime characteristics of the assembly code generated by the virtual compiler. We implement an enforced execution of assembly code instructions using Keystone (Keystone, 2024) and Unicorn (Unicorn, 2024), executing any assembly code by randomly initializing the CPU's registers and memory state. We execute assembly code generated by the real and virtual compilers, respectively, and record their memory reads and writes. To avoid infinite loops, we set a maximum executed instruction count of 2,000.

We compare the function return value, the local stack of the function, and the sequence of memory accesses as indicators for studying the runtime characteristics of functions. Specifically, we use a very strict evaluation scheme to compare whether the rax register, which holds the return value, is equal at the end of execution, whether the stack registers rsp and rbp are equal, and whether the sequence of memory reads and writes are equal. We denote the average of these three indicator as RUNTIME in Figure 3. As the number of training tokens increases, the runtime characteristics of the virtual assembly become increasingly similar to those of the ground truth.

6.2.3 Semantic Similarity

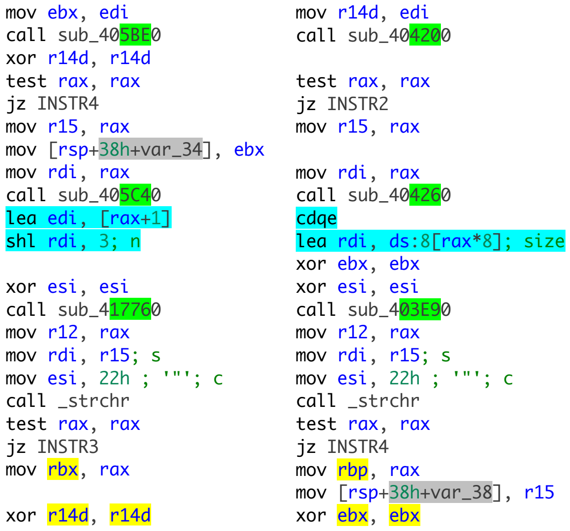
Complementing the BLEU score, we adopt a more contextualized assessment method, Binary Code Similarity Detection (BCSD), aligned with our code search task. In this task, we use CLAP (Wang et al., 2024) for evaluation. As shown in Figure 3, after training on only 0.5 billion tokens, the virtual compiler is already capable of producing assembly code achieving a `BINSIM` score above 0.8, and it nears convergence after processing 5 billion tokens. In the context of BCSD tasks, a `BINSIM` score exceeding 0.8 typically indicates a strong positive pair of functions, with their variations likely attributable to different compilers or optimization options. This signifies that the assembly code generated by our model is perceived as being remarkably close to the ground truth.

6.2.4 Case Study

Given the complexity inherent in the virtual compilation process, perfect replication of compiler-generated results is a considerable challenge. This complexity is primarily due to the extensive presence of addresses, global variables, and the use of structure offsets in the code, which add intricate layers to the compilation task. While we utilize quantitative measures like BLEU to gauge the similarity between the model-generated and compiler-generated assembly code, we recognize that such metrics might not capture the full spectrum.

In Figure 4, we present a segment of a large function to showcase the comparison between the model-generated assembly code and the real compiler-generated counterpart. A cursory glance reveals a multitude of differences. However, a thorough manual inspection reveals several distinct categories of mismatches, which we have highlighted in the figure using different colors for clarity.

The mismatches highlighted in green signify differences in the addresses. These discrepancies arise from the model's inability to obtain addresses during the virtual compilation. Cyan highlights indicate alternative expressions of the same operation. For example, the ground truth assembly executes the operation `rdi = 8 * (rax+1)` using `lea edi, [rax+1]; shl rdi, 3`, whereas the virtual compiler opts for `lea rdi, ds:8[rax*8]`. Although semantically equivalent, such variations are not captured by string-matching metrics like BLEU.



```
mov ebx, edi
call sub_405BE0
xor r14d, r14d
test rax, rax
jz INSTR4
mov r15, rax
mov [rsp+38h+var_34], ebx
mov rdi, rax
call sub_405C40
lea edi, [rax+1]
shl rdi, 3; n

xor esi, esi
call sub_417760
mov r12, rax
mov rdi, r15; s
mov esi, 22h ; '"'; c
call _strchr
test rax, rax
jz INSTR3
mov rbx, rax

xor r14d, r14d
```

(a) Real Compiler

```
mov r14d, edi
call sub_404200

test rax, rax
jz INSTR2
mov r15, rax

mov rdi, rax
call sub_404260
cdqe
lea rdi, ds:8[rax*8]; size
xor ebx, ebx
xor esi, esi
call sub_403E90
mov r12, rax
mov rdi, r15; s
mov esi, 22h ; '"'; c
call _strchr
test rax, rax
jz INSTR4
mov rbp, rax
mov [rsp+38h+var_38], r15
xor ebx, ebx
```

(b) Virtual Compiler

Figure 4: Comparison of assembly code from the (a) real compiler and (b) virtual compiler. Mismatches are highlighted in green (address differences), cyan (alternate operation expressions), yellow (register allocation variances), and grey (stack allocation discrepancies).

The third category of mismatches, marked in yellow, corresponds to differences in register allocation during code generation. The actual compiler may choose a different set of registers compared to the virtual compiler. Grey highlights denote inaccuracies in stack allocation and usage due to the model's unawareness of stack variables' sizes.

Finally, an unresolved category of divergence, not explicitly marked in the figure, reveals additional instructions provided by one compiler that are absent in the other's output, and vice versa. This discrepancy underscores the inherent unpredictability and complexity when comparing outputs from two different compilers.

6.3 Evaluation of Assembly Code Search

In this section, we validate the enhancement of the model's code search capabilities brought by the virtual compiler. Moreover, we aim to indirectly demonstrate the quality of the model-generated virtual assembly code and the model's capability to handle high-level languages.

6.3.1 Training Dataset

Similar to other code search datasets, we use the docstrings collected from the source code for comparison and apply cleaning steps in Section B. Out of 6,000 packages and 5 million source code functions, we extracted 400,000 docstrings, indicating

Dataset	Pair Count (K)	Asm Tokens (M)
CodeSearchNet	1822.3	534.5
Go	280.5	66.8
Java	446.1	123.7
JavaScript	123.2	36.0
PHP	519.7	155.7
Python	404.2	137.8
Ruby	48.6	14.5
CCSD	88.1	11.4
Docstring	408.2	159.7 (5562.8)

Table 1: The statistics for the dataset used for training. Pair Count represents the count of query-assembly code pairs. Asm Tokens represent the assembly code tokens. Due to different compilers and different optimization levels used, we show the total augmented assembly code tokens in brackets for 'Docstring'.

that only a small fraction of functions are accompanied by docstrings. We also use the virtual assembly code in Section 5.1, including the code search dataset from CodeSearchNet and CCSD. We show their statistics for each dataset in Table 1. We denote the assembly code dataset generated by `ViC` as `VirtualAssembly`, denote the extracted docstring dataset as `UbuntuDocstring`, and denote the mixture of these two datasets as `HybridAssembly`.

6.3.2 Evaluation Dataset

To construct the evaluation dataset of assembly code search that can represent the real-world scenario, we collect multiple popular binaries from the real world that have no overlap with our training set, from three different platforms: Mac, Windows, and Linux. Notably, although the model is trained using assembly code exclusively from Linux, the x86 instruction set assembly code has the same syntax across different operating systems, thereby enabling the model's cross-OS transferability. Then we perform reverse engineering on these software applications. During this process, by leveraging the expertise of reverse engineering professionals, we write code search queries for functions that exhibit relatively independent functionalities. These manually crafted queries, derived from real-world reverse engineering efforts, reflect the practical requirements encountered during actual reverse engineering tasks. This approach underscores our commitment to ensuring that the evaluation closely mimics real-world scenarios. Finally, we obtain an evaluation set containing 257 queries and relevant function pairs.

6.3.3 Baselines

Previously in binary analysis research, the concept of assembly code search has not been explicitly proposed. Research efforts predominantly revolve around binary code similarity detection (BCSD), which only targets to compare the similarity between two assembly codes, instead of the natural language level functionality.

Consequently, we use some general embedding models for comparison, including sentence-transformers (Reimers and Gurevych, 2019)[2], the initialization model for the text encoder in Section 5.3; GTE (Li et al., 2023)[3], recognized as one of the premier open-access encoders on the MTEB leaderboard; Voyage AI models[4], including the `voyage-code-2` model optimized for code retrieval; and the OpenAI embedding models (Neelakantan et al.)[5].

6.3.4 Metric

During our manual reverse engineering process, we often encounter scenarios where a single function can be described in multiple ways, or multiple functions implement similar functionalities. In such situations, the recall is defined as follows. Suppose $\{a_i\}$ is the set of relevant functions, and $\{b_i\}$ is the retrieved functions. And there are q queries trying to retrieve the desired functions. The the recall@k is defined as follow:

$$\text{recall@}k = \frac{1}{q}\sum_{i=1}^{q}\frac{|\{a_i\}\cap\{b_i\}|}{\min(|\{a_i\}|,|\{b_i\}|)} \tag{3}$$

when $|\{b_i\}| = k$.

Moreover, we adopt MAP (Mean Average Precision) to better measure the ranking capability of the model, which is defined as:

$$\text{AP} = \frac{1}{q}\sum_{k} P@k \times \text{rel}@k \tag{4}$$

where $P@k$ refers to precision@k:

$$P@k = \frac{|\{a_i\}\cap\{b_i\}|}{|\{b_i\}|}, \text{when} |\{b_i\}| = k \tag{5}$$

[2] https://huggingface.co/sentence-transformers/all-mpnet-base-v2

[3] https://huggingface.co/thenlper/gte-large

[4] https://www.voyageai.com

[5] https://openai.com/

Model	Recall@1	Recall@20	MAP
sentence-transformers	0.113	0.265	0.145
gte-large	0.121	0.244	0.149
voyage-2*	0.142	0.344	0.202
voyage-large-2*	0.319	0.635	0.388
voyage-code-2*	0.331	0.631	0.396
text-embedding-3-small†	0.173	0.406	0.225
text-embedding-3-large†	0.344	0.585	0.395
HybridAssembly	0.424	0.723	0.498

Table 2: The evaluation result of the model trained on HybridAssembly, with the general encoders as baselines. *Voyage AI embedding models. †OpenAI embedding models.

and rel@k is a relevance function, which is an indicator function that equals 1 if the function at rank k is relevant and equals 0 otherwise.

In the evaluations, the pool size is set to 10000, which means that the model needs to identify the relevant functions from 10000 functions.

6.3.5 Main Result

In Table 2, our comparative analysis highlights the performance gap between our method and baseline models, notably sentence-transformers, GTE, and voyage-2. These general models tend to underperform in code search tasks involving assembly code due to their limited coding data during training, particularly the complex assembly code which needs an understanding of control flows. Their reliance on textual snippets solely within assembly code for semantic cues, missing the control flow information, often leads to suboptimal results in both recall and MAP metrics. In contrast, the voyage-code-2 and voyage-large-2 models demonstrate better performance, attributed to their more robust training on diverse code datasets compared to voyage-2.

Models trained on the HybridAssembly dataset, starting from an MLM (Masked Language Modeling) pre-trained base model, still show significant improvements over OpenAI's models. This underscores a critical gap in the capability of existing code search models to effectively support assembly code search, indicating that these models lack comprehensive training with extensive assembly code datasets necessary to achieve a functional level for aiding binary code analysis.

6.3.6 Virtual v.s. Real Assembly Code

We analyze the effectiveness of virtual assembly code against actual assembly code in assembly code search tasks. By virtually compiling source code from UbuntuDocstring into assembly code, we train two models using the real and virtual assembly code respectively, combined with docstrings, and compared their performance. The result is detailed in Table 3, along with the result of excluding small functions from both sets of data (denoted as w/o Small), which follows our approach during the training phase in Section 4.1.

Dataset	Recall@1	Recall@20	MAP
UbuntuDocstring	0.323	0.644	0.412
w/o Small	0.329	0.612	0.402
VirtualDocstring	0.315	0.620	0.393
w/o Small	0.325	0.627	0.396

Table 3: Comparison of virtual assembly and the ground truth assembly code in assembly code search task.

Dataset	Recall@1	Recall@20	MAP
VirtualAssembly	0.292	0.641	0.375
w/o Go	0.288	0.612	0.366
w/o Java	0.292	0.621	0.359
w/o Javascript	0.296	0.618	0.364
w/o PHP	0.329	0.631	0.401
w/o Python	0.274	0.584	0.347
w/o Ruby	0.307	0.609	0.381
w/o C (CCSD)	0.272	0.578	0.348
UbuntuDocstring	0.323	0.644	0.412
HybridAssembly	0.424	0.723	0.498
w/o PHP	0.418	0.701	0.495

Table 4: The evaluation result of the contribution of different components. C is from the CCSD dataset.

The performance analysis reveals that excluding small functions, both sets of assembly code demonstrate similar outcomes, indicating that the quality of virtual assembly code is on par with real assembly code. Not training the model on small functions is intended to reduce model hallucination, but this leads to minimal performance gains and poses a risk of generating irrelevant code when processing shorter source code. Conversely, the actual compiler-generated assembly for small functions aligns well with the corresponding docstrings, providing useful insights albeit not ideal for training purposes. This slight performance dip in the UbuntuDocstring dataset without small functions underscores the nuanced impact of training data selection on model behavior.

6.3.7 Dataset Contributions

This section evaluates how each dataset influences the assembly code search performance, as shown in Table 4. Training solely with the

VirtualAssembly dataset underperformed compared to using the UbuntuDocstring dataset. Detailed analysis of specific failures highlighted the representation challenge in assembly language. For instance, low-level operations like the "keccakf function" in SHA3 or "destroying a linked list" are broadly abstracted in higher-level languages through library calls, and are missing in broader datasets like CodeSearchNet. The absence of these concepts affects performance finally.

Further evaluations involve removing language components from the VirtualAssembly dataset, as detailed in Table 4 with rows prefixed with w/o. Generally, omitting any single language dataset negatively impacts the results. Surprisingly, excluding the PHP dataset improved performance, likely due to suboptimal PHP handling by the virtual compiler, with the balanced performance in the HybridAssembly dataset suggesting compensatory benefits from better-matched docstrings.

Additionally, the analysis identifies a significant drop in performance using the CodeSearchNet dataset alone, especially without C (CCSD). This degradation highlights the importance of incorporating low-level concepts like "linked list" and "keccakf function" in training sets for assembly code search, even if these assembly codes are not compiled from a real compiler.

7 Limitation & Discussion

7.1 Code Search Dataset Quality

The quality of the assembly code search dataset we construct is intrinsically tied to the underlying code search datasets, which are predominantly derived from docstrings, such as the CodeSearchNet dataset utilized in this study. In the evaluation in CodeSearchNet, 32.8% of docstrings were found to be irrelevant to the source code. In our evaluation, a random sample of docstrings extracted from the Ubuntu source code exhibited a comparable proportion of loosely related docstrings. This observation underlines a significant limitation: relying solely on docstrings as the basis for code search datasets may introduce challenges due to the potential lack of strong relevance or association. It's conceivable to generate code search datasets directly from source code using LLMs. However, with the capability to compile different programming languages, we can greatly augment the assembly code search dataset without bias.

7.2 Broader Impacts

With the emergence of Large Language Models, code search has gained prominence as a crucial component within Retrieval-Augmented Generation (RAG) (Chase, 2024; Gao et al., 2023). With assembly code search as an RAG module, it helps the LLM to rapidly locate the desired functions with a large binary for faster reverse engineering.

While reverse engineering can be vital for security experts in vulnerability detection and analysis, it also poses risks of exploitation by malicious actors for software cracking and exploiting vulnerabilities. This duality highlights the necessity for ethical considerations and responsible use.

7.3 Future Work

We acknowledge that the evaluation dataset is relatively small, limited by the intensive labor and extensive analysis required for reverse engineers. Future work will focus on expanding and diversifying the evaluation dataset to enable a more comprehensive assessment of assembly code search.

In the model training in Section 5.3, we train the assembly code model from scratch. However, with better model initialization from previous works such as CLAP (Wang et al., 2024) and PalmTree (Li et al., 2021), we can achieve better assembly code search performance and faster convergence.

8 Conclusion

In this study, we introduce a pioneering method to improve assembly code search by training an LLM to function as a virtual compiler, ViC, effectively addressing the challenge of compiling difficulties and enhancing dataset quality for assembly code. This approach not only broadens the scope of languages compilable to assembly but also significantly boosts the performance of assembly code search tasks, outperforming all the existing solutions. Our results demonstrate the potential of virtual compilers to revolutionize reverse engineering processes, offering a promising direction for future advancements in software engineering and security analysis.

Acknowledgements

We would like to sincerely thank all the reviewers for their valuable feedback that greatly helped us to improve this paper. Additionally, special thanks are extended to Bolun Zhang and Jingwei Yi for their invaluable comments and assistance.

References

Jordi Armengol-Estapé and Michael O'Boyle. 2021. Learning C to x86 Translation: An Experiment in Neural Compilation. In *Advances in Programming Languages and Neurosymbolic Systems Workshop*.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Harrison Chase. 2024. Langchain. https://github.com/langchain-ai/langchain.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large Language Models for Compiler Optimization.

Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. 2021. AnghaBench: A suite with one million compilable C benchmarks for code-size reduction. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, pages 378–390, Virtual Event, Republic of Korea. IEEE Press.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey.

GitHub. 2018. Towards Natural Language Semantic Code Search. https://github.blog/2018-09-18-towards-natural-language-semantic-code-search/.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow.

Zifan Carl Guo and William S. Moses. 2022. Enabling Transformers to Understand Low-Level Programs. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, Waltham, MA, USA. IEEE.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum Contrast for Unsupervised Visual Representation Learning.

Hex-rays. 2023a. Hex Rays - State. https://hex-rays.com/products/ida/support/idadoc/1379.shtml.

Hex-rays. 2023b. Hex Rays - State-of-the-art binary code analysis solutions. https://hex-rays.com/ida-pro/.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.

Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2023. Nova$^+$: Generative Language Models for Binaries.

Keystone. 2024. Keystone - the ultimate assembler. https://www.keystone-engine.org.

Xuezixiang Li, Qu Yu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3236–3251.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards General Text Embeddings with Multi-stage Contrastive Learning.

Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2021. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong

Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1. Curran.

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. Text and Code Embeddings by Contrastive Pre-Training.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library.

Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2021. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity. *arXiv:2012.08680 [cs]*.

polymorf. 2020. IDA pro plugin to find crypto constants. `https://github.com/polymorf/findcrypt-yara`.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2022. RoFormer: Enhanced Transformer with Rotary Position Embedding.

Unicorn. 2024. Unicorn - the ultimate cpu emulator. `https://www.unicorn-engine.org`.

Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024. Clap: Learning transferable binary code representations with natural language supervision.

Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, Virtual South Korea. ACM.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. (arXiv:2109.00859).

Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. CodeCMR: Cross-Modal Retrieval For Function-Level Binary Source Code Matching. In *Advances in Neural Information Processing Systems*, volume 33, pages 3872–3883. Curran Associates, Inc.

```
func Covariance(data1, data2 Float64Data)
        (float64, error) {
    l1 := data1.Len()
    l2 := data2.Len()
    if l1 == 0 || l2 == 0 {
        return math.NaN(), EmptyInputErr
    }
    if l1 != l2 {
        return math.NaN(), SizeErr
    }
    m1, _ := Mean(data1)
    m2, _ := Mean(data2)
    // Calculate sum of squares
    var ss float64
    for i := 0; i < l1; i++ {
        delta1 := (data1.Get(i) - m1)
        delta2 := (data2.Get(i) - m2)
        ss += (delta1*delta2 - ss) / float64(i+1)
    }
    return ss * float64(l1) / float64(l1-1), nil
}
```

Figure 5: Example Golang function `Covariance` calculating the statistical covariance between two data sets, demonstrating input validation, mean calculation, and the sum of squares.

A Illustration for Virtual Compiling non-C/C++ language

Although our model training exclusively involved C/C++ code, the employment of Codellama—an extensively trained model across various programming languages—yields an unexpected and quite beneficial outcome. Rather than experiencing catastrophic forgetting, which is a common concern with LLMs as they undergo extensive continued training, our model retains its previously learned information. This retention facilitates the model's understanding of other programming languages and enables it to generate corresponding assembly code effectively, akin to its behavior with C/C++.

Operating in this broader context without standard ground truth for these additional languages presented a unique challenge. To gauge the model's cross-language compilation capabilities, we sample some generated assembly code for these languages and manually check these samples to verify the correctness of the assembly output

Figure 5 presents a Golang function for calculating covariance, while Figure 6 displays the corresponding virtual assembly code. The mapping between key elements of the source and assembly code is highlighted for clarity: the input validation segment of the function is marked with a grey background; function calls, including the Mean function and access to an item inside the Golang array, are illuminated in green; arithmetic operations performed during the covariance calculation are emphasized in cyan; and the for loop—encompassing boundary checks and the incrementation of the loop variable—is spotlighted in yellow.

```
    endbr64
    push r13
    push r12
    push rbp
    push rbx
    sub rsp, 28h
    mov r13d, [data1+8]
    mov r12d, [data2+8]
    movsd xmm0, cs:qword_6098
    test r13d, r13d
    jz INSTR2
    test r12d, r12d
    jz INSTR2
    cmp r13d, r12d
    jz INSTR0
    movsd xmm0, cs:qword_6098
    add rsp, 28h
    mov eax, 1
    pop rbx
    pop rbp
    pop l2
    pop l1
    retn
INSTR0:
    mov rbp, data2
    mov rbx, data1
    call Mean
    mov rdi, data2; data
    movsd [rsp+48h+m1], xmm0
    call Mean
    test r13d, r13d
    pxor xmm4, xmm4
    movsd [rsp+48h+m2], xmm0
    movsd xmm0, [rsp+48h+m1]
    jle INSTR3
    xor r12d, r12d
    movsd [rsp+48h+ss_0], xmm4
    nop dword ptr [rax+00h]
INSTR1:
    mov esi, r12d
    mov rdi, data1
    movsd [rsp+48h+m1], xmm0
    call Float64Data.Get
    movsd xmm2, [rsp+48h+m1]
    mov esi, r12d
    mov rdi, data2
    add r12d, 1
    subsd xmm0, xmm2
    movsd [rsp+48h+var_38], xmm0
    call Float64Data.Get
    subsd xmm0, [rsp+48h+m2]
    pxor xmm1, xmm1
    movsd xmm3, [rsp+48h+ss_0]
    movsd xmm2, [rsp+48h+var_38]
    cvtsi2sd xmm1, r12d
    mulsd xmm2, xmm0
    movapd xmm0, xmm2
    subsd xmm0, xmm3
    divsd xmm0, xmm1
    addsd xmm0, xmm3
    movsd [rsp+48h+ss_0], xmm0
    cmp r13d, r12d
    jnz INSTR1
    pxor xmm1, xmm1
    movsd xmm0, [rsp+48h+ss_0]
    cvtsi2sd xmm1, r13d
    mulsd xmm0, xmm1
    pxor xmm1, xmm1
    cvtsi2sd xmm1, r12d
    divsd xmm0, xmm1
INSTR2:
    add rsp, 28h
    xor eax, eax
    pop rbx
    pop rbp
    pop r12
    pop l1
    retn
INSTR3:
    movsd [rsp+48h+ss_0], xmm4
    movsd xmm0, [rsp+48h+ss_0]
    jmp INSTR2
```

Figure 6: Virtual assembly code generated from the Golang `Covariance` function shown in Figure 5, showcasing the translated computational logic and structure in low-level instructions. The corresponding function parts are highlighted in different colors.

B Docstring Cleaning

Following a similar approach to the one used by CodeSearchNet for cleaning docstrings from function source code, we apply a series of cleaning steps to the docstrings we collected:

- *Eliminate docstring text borders*. We strip away leading and trailing '*' characters from docstrings, which act as decorative "borders" around docstrings, and do not contribute to the semantic value of the query.

- *Retain first paragraph*. Only the first paragraph of each docstring is kept to eliminate extensive details about function parameters and return values.

- *Remove short docstrings*. After trimming down to the first paragraph, docstrings that are too short are considered as lacking meaningful content and, thus are removed.