Nova: Generative Language Models for Assembly Code with Hierarchical Attention and Contrastive Learning

Anonymous ACL submission

Abstract

Binary code analysis is the foundation of crucial tasks in the security domain; thus building effective binary analysis techniques is more important than ever. Large language models (LLMs) although have brought impressive improvement to source code tasks, do not directly generalize to assembly code due to the unique challenges of assembly: (1) the low information density of assembly and (2) the diverse optimizations in assembly code. To overcome these challenges, this work proposes a *hierarchical attention* mechanism that builds attention summaries to capture the semantics more effectively, and designs *contrastive learning objectives* to train LLMs to learn assembly optimization. Equipped with these techniques, this work develops *Nova*, a generative LLM for assembly code. Nova outperforms existing techniques on binary code decompilation by up to 146.54%, and outperforms the latest binary code similarity detection techniques by up to 6.17%, showing promising abilities on both assembly generation and understanding tasks.

1 Introduction

Binary code plays an irreplaceable role in the security domain, being the foundation of crucial tasks including vulnerability detection (Güler et al., 2019; Duan et al., 2020; Chen et al., 2022b), malware detection (Spensky et al., 2016; Aonzo et al., 2023; Xu et al., 2014), binary recovery (Su et al., 2024; Zhang et al., 2021; Chen et al., 2022c), and legacy software maintenance (Carbone et al., 2009; Carlini et al., 2015; Martin et al., 2010). For example, when performing tasks such as identifying attacks and malware, security analysts often only have access to assembly, i.e., the human-readable representation of binary code, which is extremely difficult to understand (Su et al., 2024; Zhang et al., 2021; Chen et al., 2022c). Thus, combined with the increasing sophistication of cybercrime that poses significant threats worldwide (e.g., cybercrime is predicted to cost the world $10.5 trillion annually by 2025 (Sausalito, 2020)), effective binary analysis techniques are in high demand.

Large language models pre-trained on source code have brought improvement in various software development domains (Chen et al., 2022a; Liu et al., 2023; Chen et al., 2023; Le et al., 2022; Jiang et al., 2023; Xia et al., 2023). However, these LLMs are not designed for or trained with assembly corpus, not achieving their full potential on binary code analysis tasks such as binary code similarity (Wang et al., 2022; Xu et al., 2023a), malware detection (Su et al., 2024), and binary code decompilation (Tan et al., 2024; Armengol-Estapé et al., 2024; Hosseini and Dolan-Gavitt, 2022). Existing work applying LLMs on assembly code mainly piggybacks on encoder-style LLMs (Wang et al., 2022; Su et al., 2024; Xu et al., 2023a), unable to benefit from the more extensive pre-training, updated architectures, scaling of state-of-the-art generative LLMs. Other work using generative LLMs for decompilation shows a low unit test passing rate of the decompiled programs (Tan et al., 2024; Armengol-Estapé et al., 2024).

The challenges of leveraging generative LLMs for assembly code are twofold. First, compared to source code, assembly code has a *lower information density*. A short source-code sequence maps to a much longer assembly-code sequence that is often several times longer. Thus, assembly semantics span across a *long sequence of tokens*. Figure 1 (a) shows an example of a source code function that compares two integers, while Figure 1 (b) shows its corresponding assembly code optimized with `O0` flag. In the `O0`-optimized assembly code, the five instructions from `10: mov -0x8(%rbp),%rax` to `1c: cmp %eax,%edx` perform the checking whether the value of `x` is smaller than the value of `y` (correspond to `if (*(int*)x < *(int*)y)` in the source code). A single assembly instruction alone represents little meaningful semantics in the source

```c
#include <stdio.h>
#include <math.h>

int compare(int *x, int *y) {
    if (*(int*)x < *(int*)y)
        return -1;
    if (*(int*)x > *(int*)y)
        return 1;
    return 0;
}
```

(a) Source Code Function

```
0:    endbr64
4:    push   %rbp
5:    mov    %rsp,%rbp
8:    mov    %rdi,-0x8(%rbp)
c:    mov    %rsi,-0x10(%rbp)
10:   mov    -0x8(%rbp),%rax
14:   mov    (%rax),%edx
16:   mov    -0x10(%rbp),%rax
1a:   mov    (%rax),%eax
1c:   cmp    %eax,%edx
...
```

(b) Assembly (O0-Optimized)

```
0:    endbr64
4:    mov    (%rdi),%ecx
6:    mov    (%rsi),%edx
8:    mov    $0xffffffff,%eax
d:    cmp    %edx,%ecx
f:    jl     17
11:   setg   %al
14:   movzbl %al,%eax
17:   retq
```

(c) Assembly (O1-Optimized)

Figure 1: Example that shows the semantics and diverse optimizations of assembly code.

code. It is the combinations of *many instructions* and the *dependencies* between them represent the semantics. Such combinations of instructions are long, which is hard for LLMs to learn.

Second, assembly code is diverse due to compiler optimization. The assembly code of the same source code function looks dramatically different with different compiler optimization. Figure 1 (c) shows the assembly of the same function compiled with `O1` and `O0` flags, which consists of a significantly different set of instructions. Such syntax diversity is hard for LLMs to learn, preventing LLMs from obtaining consistently good performances on differently optimized assembly code.

In this work, we develop Nova, a generative foundation LLM pre-trained for assembly code with two key novelties. First, to address the low-information-density and long-sequence challenge, we design a hierarchical self-attention, which contains three categories of attention at different levels of granularity: intra-instruction attention, preceding-instruction attention, and inter-instruction attention. The key insight is to build *attention summaries*, i.e., we create per-statement attention *labels*, which act as the summary of a statement. We then use preceding-instruction attention to capture semantics between a token and its preceding instruction label and use inter-instruction attention for long dependencies. Besides, we design *functionality contrastive learning* and *optimization contrastive learning* objectives to train Nova to learn the semantics behind the diverse syntax of assembly.

This work makes the following contributions:

- We propose a novel hierarchical attention mechanism that captures the assembly's low-density semantics at three granularity levels.
- We design contrastive learning objectives to train LLMs to learn assembly with diverse optimizations and encode assembly more efficiently.
- We develop Nova, a generative foundation LLM with hierarchical attention and contrastive learning for assembly. Nova outperforms state-of-the-art (SOTA) on binary decompilation by up to 146.54% and on binary similarity detection by up to 6.17%.
- We conduct a comprehensive analysis, illustrating the effectiveness of Nova's novel designs: (1) Nova's embeddings of assemblies successfully reflect code functionalities in the latent space, and (2) Nova's hierarchical attention complements standard attention by learning different attention weight distributions, especially those reflecting long sequence semantics.

2 Approach

Figure 2 presents the overall approach of Nova. We build Nova on top of foundation models for source code (Rozière et al., 2023; Li et al., 2023; Guo et al., 2024) to utilize their source code and natural language generation ability. We first collect large assembly corpora (Section 2.1). Section 2.2 describes Nova's hierarchical attention. With the collected assembly corpora, we then pretrain Nova with language modeling and contrastive learning objectives (Section 2.3). Then, we fine-tune Nova on two important downstream tasks, binary code decompilation, and binary code similarity detection (Sections 2.4 and 2.5), to prove Nova's effectiveness and benefits to the binary research domain.

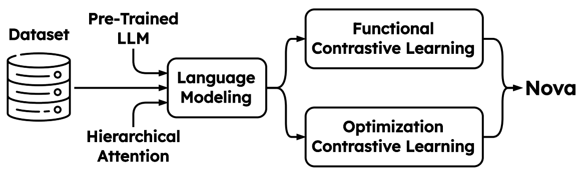

Figure 2: Overview of developing Nova

2.1 Data Collection

We build our assembly data sets on top of existing source code datasets: The-Stack (Li et al., 2023) and the AnghaBench (da Silva et al., 2021). We compile the source code into executables with different optimization levels (i.e., `O0`, `O1`, `O2` and `O3`), strip the executables to remove debug information, and disassemble them into assembly code. We treat every assembly function as a separate data sample.

The breakdown statistics are in Table 1.

Datasets	Source	O0	O1	O2	O3	Total
AnghaBench	757.1K	743.1K	726.4K	718.7K	717.8K	3.7M
The-Stack	138.8K	125.1K	119.7K	116.9K	108.8K	609.3K

Table 1: Statistics (number of source code and assembly functions) of the pre-training datasets.

We perform certain normalization on the assembly functions: (1) removing all the "`%`" and comments, (2) adding whitespace around "`,`", "`(`", "`)`", (3) converting all the hexadecimal numbers to decimal numbers, and (4) replacing the address of each instruction with special labels (e.g., replacing "`0`" and "`4`" in Figure 1 (b) with "`[INST-1]`" and "`[INST-2]`") placing at the end of each instruction. More details are in Appendix A.1.

2.2 Hierarchical Self-Attention

Nova uses hierarchical self-attention that is specially designed to learn the *low-information-density* semantics in the *long* sequence of assembly code. Specifically, Nova learns the assembly code in an hierarchical way by providing a modified attention mask. Different from standard token-level attentions (Vaswani et al., 2023; Radford and Narasimhan, 2018; Radford et al., 2019; Brown et al., 2020), our hierarchical self-attention contains three categories at different levels of granularity.

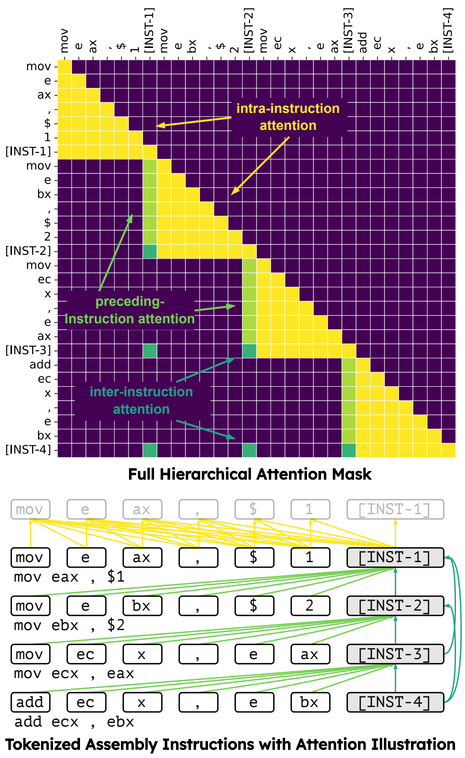

Figure 3: Design of hierarchical attention

(1) Intra-Instruction Attention: Due to the low information density in assembly, intra-instruction attention is designed to capture the summary of every instruction, which is the standard causal attention but limited to tokens of each instruction (the yellow part in Figure 3). Tokens in different instructions have no attention weights. The "`[INST]`" label at the end of the instruction has attention to all the tokens in the instruction and thus captures the semantics of the entire instruction (e.g., "`[INST-1]`" captures the semantics of "`mov eax, $1`").

(2) Preceding-Instruction Attention In addition to the local semantics of each instruction, the use of assembly instructions (such as the choice of registers) depends on the context. For example, after the first instruction "`mov eax, $1`", the second instruction should not reuse "`eax`" to store another value "`$2`" immediately. To capture such context, the preceding-instruction attention enables each token in an instruction to have attention to the "`[INST]`" label of the preceding instruction (the light green part in Figure 3).

(3) Inter-Instruction Attention To understand function semantics (i.e., functionality), which lies in the dependencies across different instructions, the inter-instruction attention is designed to let the "`[INST]`" label of each instruction have attention to all the labels of previous instructions. For example, "`[INST-4]`" has attention to "`[INST-1]`", "`[INST-2]`", and "`[INST-3]`" (the dark green part in Figure 3). The inter-instruction attention is only enabled for "`[INST]`" labels, as they represent the semantics of each instruction.

To sum up, the hierarchical self-attention breaks the semantics of assembly code into three parts. The intra-instruction attention captures the instruction summary, and the preceding-instruction attention captures the context with the preceding instruction. The inter-instruction attention learns the long dependencies across instructions on top of the "`[INST]`" labels that contain the instruction summary. Appendix A.2 shows how hierarchical self-attention works with text and source code.

2.3 Contrastive Learning

The syntax gap between assembly code and source code, and syntax diversity between differently-optimized assembly code make LLMs struggle to distinguish the semantics behind the syntax. Nova adopts contrastive learning technique (Gao et al., 2021) during pre-training to train LLMs to encode assembly code meaningfully w.r.t semantics.

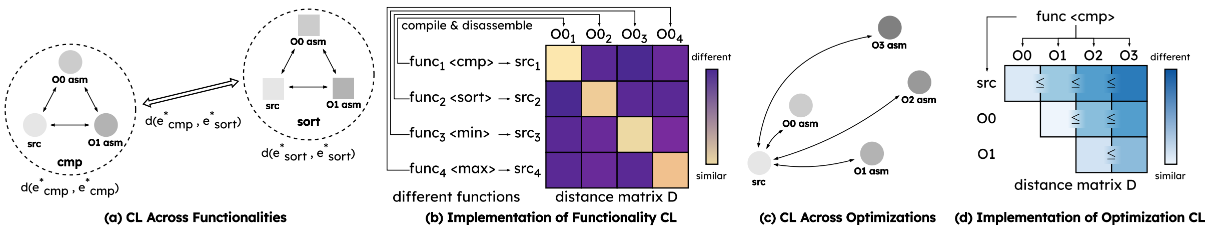

Figure 4: Design of functionality and optimization contrastive learning (CL). "asm" denotes assembly.

The standard pre-training objective is language modeling by minimizing the negative likelihood of code in the pre-training corpus (Radford and Narasimhan, 2018), notated as L_{lm}. In addition, Nova is pre-trained with two new objectives, L_{fcl} for functionality contrastive learning and L_{ocl} for optimization contrastive learning.

Functionality CL Functionality CL trains Nova to focus more on the functionalities of assembly code rather than the syntax. Code with the same functionality (assemblies from the same source code), should be encoded closer in the latent space. For instance, in Figure 4 (a), embeddings of source and assembly code of function "`cmp`" are closer to each other, and the same for function "`sort`".

Let e_f^s be the embedding of function f in s form ($s = -1$ for source code, and $s \in [0, 1, 2, 3]$ for `O0` to `O3` optimized assembly). For simplicity, let $S = [-1, 0, 1, 2, 3]$ be the domain of s. We use the average of all the "`[INST]`" tokens' embedding as the embedding of the whole assembly function, as each "`[INST]`" token is supposed to capture the semantics of that instruction by the design of hierarchical self-attention. Functionality CL optimizes Nova with the constraint:

$$\forall f_i \in F,\ \underset{s,t \in S}{max}(d(e_{f_i}^s, e_{f_i}^t)) < \underset{\substack{s,t \in S \\ f_j \neq f_i \in F}}{min}(d(e_{f_i}^s, e_{f_j}^t))$$

, where d calculates the l_2 distance between two embeddings and F is the full set of functions in the training corpus.

Such constraints can be trained by optimizing the embeddings of a batch of functions, each function in two different forms. For the example in Figure 4 (b), there are two forms (source code and `O0` assembly) of four functions. Once Nova encodes the batch of source code and assembly functions, we calculate the distance matrix $\{D_{ij}\}_{f_i, f_j \in F} = \{d(e_{f_i}^s, e_{f_j}^t)\}$, and minimize the loss:

$$L_{fcl} = -\,log \sum_{s,t \in S} \sum_{f_i \in F} \left(1 - \frac{exp\left(d(e_{f_i}^s, e_{f_i}^t)\right)}{\sum_{f_j \in F} exp\left(d(e_{f_i}^s, e_{f_j}^t)\right)}\right)$$

This objective minimizes the distance between embeddings for the same function, i.e., the diagonal in the distance matrix.

Optimization CL LLMs can be confused if being asked to directly connect a source code function to its `O3`-optimized assembly, due to their dramatically different syntax. Such a huge gap can be filled by learning how the source code is transformed to `O0`, `O1`, `O2` and eventually to `O3` assembly, as the optimization levels are *ordered*.

Higher-level optimization applies a super-set of optimization rules compared to lower-level optimization. Nova learns such order with the optimization CL objective, encoding differently-optimized assembly code orderly. Optimization CL optimizes Nova with the constraint:

$$\forall f \in F, \forall s < t_1 < t_2 \in S,\ d(e_f^s, e_f^{t_1}) \leq d(e_f^s, e_f^{t_2})$$

Intuitively, this ensures that the more optimizations applied, the larger the difference between embeddings of optimized and unoptimized code. For instance, Figure 4 (c) and (d) illustrate that for the same function "`cmp`", the distance between source code and assembly increases when the optimization level increases. Formally, optimization CL minimizes the following loss:

$$L_{ocl} = \sum_{f \in F} \sum_{s < t_1 < t_2 \in S} max\left(0, d(e_f^s, e_f^{t_1}) - d(e_f^s, e_f^{t_2})\right)$$

Overall, the final training loss combines the three: $L = L_{lm} + \lambda(L_{fcl} + L_{ocl})$, where λ is set to 0.1 to balance the losses in this work.

2.4 Task 1: Binary Code Decompilation

Binary code decompilation (BCD) helps developers to understand binary code by recovering binary code into more readable high-level source code (e.g., C programs) (Fu et al., 2019; Liang et al., 2021; Armengol-Estapé et al., 2024; Tan et al., 2024). The input to the model for BCD is formatted as an instruction prompt (notated by $\mathbf{p}$): `# This is the assembly code with {opt} optimization:` **{asm}**, where "opt" is the optimization-level applied to the assembly and "**asm**" is the assembly code to decompile. Nova is fine-tuned to generate the expected source code function **src** following the instruction prompt. The fine-tuning objective is minimizing the loss: $L_{bcd} = -\,log\, P(\mathbf{src}|\mathbf{p})$.

2.5 Task 2: Binary Code Similarity Detection

Binary code similarity detection (BCSD) aims to measure the similarity between two binary code snippets (Wang et al., 2022; Su et al., 2024), which is the foundation of various applications such as plagiarism detection (Luo et al., 2014; Sæbjørnsen et al., 2009) and vulnerability detection (David and Yahav, 2014; David et al., 2018, 2017, 2016).

A widely used setting is taking a query assembly of the function f^q that is compiled with one optimization level (denoted by s), and a pool of candidate assembly of K functions (notated by f^p_i, $1 \le i \le K$) compiled with a different optimization level (denoted by $t \neq s$). There exists a unique candidate assembly coming from the same source code as the query ($\exists! 1 \le i \le K$, $f^p_i = f^q$, called the positive candidate). Nova is fine-tuned to encode these binaries, so that the positive candidate has the highest similarity with the query assembly among the pool. The learning objective is:

$$L_{BCSD} = -\,log \sum_{\substack{1\le j\le K\\ f^q:=f^p_j}} \left(1 - \frac{exp\left(d(e^s_{f^q}, e^t_{f^q_j})\right)}{\sum_{1\le i\le K} exp\left(d(e^s_{f^q}, e^t_{f^p_i})\right)}\right)$$

, where we follow previous work (Su et al., 2024) to let s be `O0`-assembly and t be `O3`-assembly, which is the hardest setting.

3 Experimental Setup

3.1 Pre-Training

We use the data collected from AnghaBench and The-Stack for pre-training. We pre-train Nova starting from DeepSeek-Coder (Guo et al., 2024), and the hierarchical attention is applied on half of the attention heads to balance between its effectiveness and the existing knowledge in the standard attention layers. Nova is pre-trained with language modeling for one epoch, followed by contrastive learning objectives for another epoch.

3.2 Binary Code Decompilation

Training Data We sample 2.16M assembly-to-source-code pairs (0.338B tokens) from the pre-training corpus to build the BCD fine-tuning data.

Test Data We use HumanEval-Decompile (Tan et al., 2024) as the test benchmark, which was not used in training. HumanEval-Decompile is derived from the C language adaptation of the HumanEval (Chen et al., 2021) benchmark and provides test cases in evaluating functionality correctness. HumanEval-Decompile contains 164 C functions, each compiled with `O0` – `O3` optimization flags and disassembled into X86-64 assembly.

Baselines Nova is compared with GPT-3.5, GPT-4, and the previous SOTA LLM4Decompile (Tan et al., 2024). LLM4Decompile trains DeepSeek-Coder using the same AnghaBench corpus, and it is the first LLM-based technique that aims to generate executable decompilations.

Evaluation GPT-3.5 and GPT-4 are prompted with three-shot examples, while LLM4Decompile and Nova samples 20 decompilations per assembly, using the temperature of 0.2 and `top_p` of 0.95 (Chen et al., 2021). The generated decompilations are executed against the test cases and both Pass@1 and Pass@10 (Chen et al., 2021) are reported.

3.3 Binary Code Similarity Detection

Training Data To compare Nova with existing works on BCSD fairly (Wang et al., 2022; Su et al., 2024), we use BinaryCorp-3M (Wang et al., 2022) as the fine-tuning data for BCSD, which contains the `O0` and `O3` assembly of 224,606 functions.

Test Data Following existing work (Su et al., 2024; Xu et al., 2023a), we use real-world benchmarks, Binutils, Curl, ImageMagick, SQLite, OpenSSL, and Putty, as the test benchmarks, which are nonexistent in the training data.

Baselines Nova is compared with jTrans (Wang et al., 2022), DiEmph (Xu et al., 2023a) and CodeArt (Su et al., 2024). jTrans is a Transformer (Vaswani et al., 2023) encoder trained on binaries with masked token prediction and jump target prediction tasks. DiEmph uses an instruction deemphasis technique to prevent the model from learning instruction distribution biases introduced by compilers. CodeArt proposes a regularized attention mask for encoder models to capture instructional semantics and data dependencies.

Evaluation We randomly sample K source code functions from each project, compile them into binaries with `O0` and `O3` optimization flags, and disassemble them into X86-64 assemblies. BCSD techniques encode these assemblies into embeddings (Nova uses the average embeddings of all the "[INST]" tokens in an assembly as its embedding). Then each O0 assembly is used as the query to calculate their similarity with the K O3 candidate assemblies (using cosine similarity). Metric Recall@1 is the ratio of queries for which the candidate from the same source code has the highest similarity among all the candidates.

Appendix A.3 contains additional details such as training hyper-parameters.

4 Results

4.1 Binary Code Decompilation

4.1.1 Comparison with SOTA Techniques

Table 2 shows the Pass@1 of the decompiled code from assemblies on the HumanEval-Decompile benchmark. The results are grouped by optimization level (i.e., the benchmark contains 164 assemblies of each optimization level to decompile), and the average is also reported.

Optimization	O0	O1	O2	O3	Avg.
GPT-3.5	6.80	5.64	4.36	3.93	5.18
GPT-4	17.77	15.12	12.65	11.25	14.20
LLM4Decompile-1B	12.26	7.22	8.38	7.96	8.96
Nova-1B	31.19	17.29	18.72	15.58	**22.09**
LLM4Decompile-6B	23.01	15.95	16.95	14.79	17.68
Nova-6B	42.07	28.04	25.00	22.56	**29.42**

Table 2: Pass@1 on HumanEval-Decompile.

Optimization	O0	O1	O2	O3	Avg.
GPT-3.5	8.95	7.77	5.93	5.12	6.94
GPT-4	25.64	20.65	18.70	18.03	20.76
LLM4Decompile-1B	17.95	12.05	13.90	12.51	14.10
Nova-1B	41.11	29.81	31.18	26.24	**32.09**
LLM4Decompile-6B	33.77	24.25	23.94	23.81	26.44
Nova-6B	51.06	37.83	35.79	34.63	**39.83**

Table 3: Pass@10 on HumanEval-Decompile.

Overall, Nova's Pass@1 is higher than all SOTA binary decompilation techniques and general LLMs GPT-4 and GPT-3.5, which are orders of magnitude larger than Nova. Specifically, for each optimization level, Nova consistently decompiles more assemblies into source code correctly than LLM4Decompile, GPT-3.5, and GPT-4. With the same model size, Nova-1B outperforms LLM4Decompile-1B by 146.54%, i.e., averaged Pass@1 of 22.09% versus 8.96%. Nova-6B outperforms LLM4Decompile-6B by 66.40%: the averaged Pass@1 is 29.42% versus 17.68%.

Table 3 shows that Nova still outperforms SOTA techniques with a significant margin under the measurement of Pass@10. Examples of Nova's correct decompilation are provided in Appendix A.4.

4.1.2 Ablation Study

We conduct an ablation study by comparing Nova-1B with the following models:

- Nova$_{-CL-HA}$: Removing contrastive learning and hierarchical self-attention. This is simply training DeepSeek-Coder-1.3B on the assembly corpus using language modeling.
- Nova$_{-HA}$: Removing the hierarchical self-attention, training DeepSeek-Coder-1.3B on the assembly corpus using both the language modeling and contrastive learning objectives.

Nova$_{-CL-HA}$ can be viewed as our reproduction (retrain) of LLM4Decompile-1B.

Optimization	O0	O1	O2	O3	Avg.
LLM4Decompile-1B	17.95	12.05	13.90	12.51	14.10
Nova$_{-CL-HA}$	17.80	13.32	13.26	10.03	13.60
Nova$_{-HA}$	25.12	15.64	16.07	12.71	17.39
Nova	31.19	17.29	18.72	15.58	**22.09**

Table 4: Ablation study of Nova-1B (Pass@1).

Table 4 shows the results of the ablation study, reported by the Pass@1 on HumanEval-Decompile. Nova$_{-CL-HA}$ shows comparable Pass@1, which we considered as variance in reproducing the same approach. With additional contrastive learning objectives, Nova$_{-HA}$ improves the Pass@1 on all optimization levels over Nova$_{-CL-HA}$, showing a 27.87% higher averaged Pass@1. Further applying the hierarchical self-attention boosts the overall Pass@1 from 17.39% to 22.09%.

4.2 Binary Code Similarity Detection

4.2.1 Comparison with SOTA Techniques

Tables 5, 6, 7 and 8 show the Recall@1 of Nova and existing BCSD techniques with pool size K of 50, 100, 200 and 500 on the six benchmarks. Underlined numbers indicates the best in each benchmark, while wavey underlined numbers denote the tied best (we only mark Nova-1B for clearer illustration).

Overall, Tables 5, 6, 7 and 8 show that *on average, Nova-1B and Nova-6B achieve the highest Recall@1 (in **bold**) under all four settings of* K. Nova-6B further outperforms Nova-1B and achieves the highest averaged Recall@1 under all four settings, ranking the ground-truth of 5%, 2%, 4%, and 3% more queries the most similar correspondingly compared to CodeArt.

Benchmarks	jTrans	DiEmph	CodeArt	Nova-1B	Nova-6B
Binutils	0.68	0.80	0.84	0.87	0.89
Curl	0.72	0.84	0.86	0.89	0.94
ImageMagick	0.53	0.71	0.78	0.86	0.90
SQLite	0.73	0.79	0.78	0.77	0.78
OpenSSL	0.70	0.83	0.88	0.90	0.92
Putty	0.63	0.72	0.69	0.72	0.71
Avg.	0.67	0.78	0.81	**0.84**	**0.86**

Table 5: Recall@1 on benchmarks with $K = 50$.

Benchmarks	jTrans	DiEmph	CodeArt	Nova-1B	Nova-6B
Binutils	0.60	0.63	0.81	0.79	0.79
Curl	0.63	0.80	0.82	0.86	0.88
ImageMagick	0.54	0.71	0.76	0.79	0.81
SQLite	0.62	0.72	0.74	0.73	0.72
OpenSSL	0.60	0.80	0.87	0.88	0.90
Putty	0.58	0.64	0.64	0.65	0.64
Avg.	0.60	0.72	0.77	**0.78**	**0.79**

Table 6: Recall@1 on benchmarks with $K = 100$.

Benchmarks	jTrans	DiEmph	CodeArt	Nova-1B	Nova-6B
Binutils	0.51	0.64	0.74	0.73	0.73
Curl	0.57	0.77	0.78	0.83	0.84
ImageMagick	0.39	0.51	0.67	0.73	0.75
SQLite	0.56	0.65	0.68	0.68	0.69
OpenSSL	0.54	0.71	0.82	0.84	0.88
Putty	0.49	0.58	0.55	0.55	0.58
Avg.	0.51	0.64	0.71	**0.73**	**0.75**

Table 7: Recall@1 on benchmarks with $K = 200$.

Benchmarks	jTrans	DiEmph	CodeArt	Nova-1B	Nova-6B
Binutils	0.40	0.57	0.70	0.65	0.67
Curl	0.43	0.62	0.69	0.73	0.76
ImageMagick	0.25	0.42	0.58	0.61	0.65
SQLite	0.43	0.59	0.62	0.59	0.62
OpenSSL	0.43	0.61	0.76	0.78	0.82
Putty	0.38	0.50	0.49	0.47	0.51
Avg.	0.39	0.55	**0.64**	**0.64**	**0.67**

Table 8: Recall@1 on benchmarks with $K = 500$.

Nova-1B consistently outperforms existing techniques with higher Recall@1 when K is 50, 100, and 200, meaning it correctly ranks ground-truth of 3%, 1%, and 2% more queries as the most similar. Under the setting of $K = 500$, Nova-1B ties with CodeArt with the same highest Recall@1. When looking into each individual benchmark, Nova-1B always wins on the most benchmarks under different settings of pool size K. For instance, Nova-1B wins on four benchmarks while DiEmph only wins on SQLite when $K = 50$.

K	Nova$_{-CL-HA}$	Nova$_{-HA}$	Nova
50	0.81	0.83 (+0.02)	0.84 (+0.01)
100	0.76	0.78 (+0.02)	0.78
200	0.70	0.70	0.73 (+0.03)
500	0.60	0.62 (+0.02)	0.64 (+0.02)

Table 9: Ablation study of Nova-1B (Recall@1)

4.2.2 Ablation Study

Table 9 shows the averaged Recall@1 of Nova$_{-CL-HA}$, Nova$_{-HA}$ (same as in Section 4.1.2), and Nova-1B under four pool size settings. With contrastive learning objectives, Nova$_{-HA}$ improves Nova$_{-CL-HA}$ under three settings ($K = 50, 100, 200$) with 2% higher Recall@1. With hierarchical attention, Nova further outperforms Nova$_{-HA}$ under three settings ($K = 50, 200, 500$). Detailed ablation study results on each benchmark are provided in Appendix A.5.

4.3 Analytic Experiments

4.3.1 How are Nova's embeddings better?

We use the widely-used PCA to analyze and visualize high-dimensional embeddings. We randomly sample seven coding problems from HumenEval-Decompile (`task_id` 19, 32, 34, 63, 119, 128, 143), encode the `O0` – `O3` assemblies by Nova$_{-CL-HA}$ and Nova-1B. Figure 5 shows the embeddings that are visualized under the first two principal components. Each color represents one task, and `O0` – `O3` assemblies are marked by ○, ▽, △, and □.

Figure 5 (b) shows that Nova encodes assemblies into clusters of functionalities. The assemblies for the same functionality (i.e., the same task) are encoded closer to each other than Nova$_{-CL-HA}$ does in Figure 5 (b). The results show that our hierarchical attention and contrastive learning techniques effectively group codes of similar functionalities together for better assembly foundation models. Embedding of Nova$_{-HA}$ and additional quantitative analysis are shown in Appendix A.6.

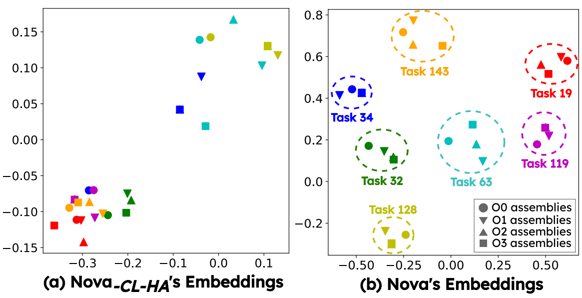

Figure 5: PCA of embeddings calculated by Nova$_{-CL-HA}$ and Nova, for HumanEval-Decompile assemblies.

4.3.2 What does hierarchical attention learn?

We conduct quantitative analysis on the attention weights produced by different models.

Entropy Figure 6 (a) shows the entropy of attention-weight distributions in each layer. We separate the attention heads as standard attention

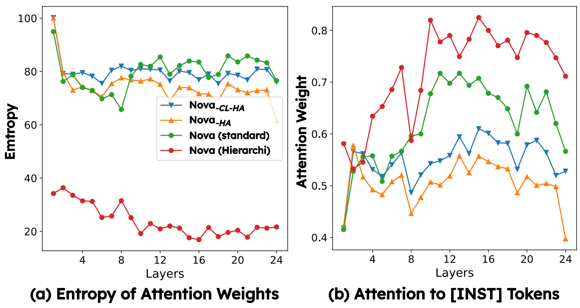

Figure 6: Quantitative analysis of attention weights.

(green line) and hierarchical attention (red line), since Nova applies hierarchical attention to half of the attention heads in each layer (Section 3.1).

Nova's hierarchical attention heads produce significantly lower entropy, suggesting its attention layer is more *confident* in learning specific relationships than the other models' attention layers. The standard attention heads in Nova show patterns similar to those of Nova$_{-CL-HA}$ and Nova$_{-HA}$, allowing Nova learning the standard attention to capture the "soft" relationship between possible tokens pairs. The result suggests that hierarchical attention complements standard attention with additional knowledge.

`[INST]` Token Figure 6 (b) shows the attention weights paid to the "`[INST]`" tokens. Nova's hierarchical attention heads pay more attention to the "`[INST]`" tokens than standard attention, which may be because these "`[INST]`" tokens contain instruction summary and long dependencies and thus are more informative. Additional analysis and examples are given in Appendix A.7.

5 Related Work

5.1 Binary Models

Machine learning models are widely used in binary program analysis tasks. However, these models are typically designed for specific downstream tasks such as binary code similarity detection (Pei et al., 2020; Xu et al., 2023a; Wang et al., 2022), variable name prediction (Chen et al., 2022c; Xu et al., 2023b; Zhang et al., 2021), binary code type inference (Pei et al., 2021), and are built from scratch.

Recent techniques have started to pre-train foundation LLMs for binaries. CodeArt (Su et al., 2024) pre-trains encoder-style LLMs with a regularized attention design to better encode assembly code semantics, showing good accuracy on binary code understanding tasks (e.g., binary code similarity detection and malware family classification). SLaDe (Armengol-Estapé et al., 2024) trains BART (Lewis et al., 2019) models on assembly, and LLM4Decompile (Tan et al., 2024) trains DeepSeek-Coder with assembly for binary code decompilation. However, CodeArt does not generalize to generation tasks due to its encoder architecture. SLaDe and LLM4Decompile are limited in performance due to a lack of special designs for assembly. In contrast, Nova addresses both limitations, by using the proposed hierarchical attention and contrastive learning objectives, outperforming existing techniques on both understanding (binary code similarity detection) and generation (binary code decompilation) tasks.

5.2 Large Source-Code Models

LLMs demonstrate promising results on many code-related tasks, such as code generation (Chen et al., 2022a; Liu et al., 2023; Chen et al., 2023; Le et al., 2022; Yue et al., 2021; Chen et al., 2021; Nijkamp et al., 2022; Fried et al., 2023; Rozière et al., 2023; Guo et al., 2024), bug fixing (Jiang et al., 2023; Xia et al., 2023) and vulnerability fixing (Wu et al., 2023; Steenhoek et al., 2023; He and Vechev, 2023). The advances in using LLMs are attributed to the knowledge learned from massive source code and natural language text in their training datasets (Touvron et al., 2023; OpenAI, 2023). Nova is designed and trained for assembly, which has unique challenges such as low information density and diverse optimization.

6 Conclusion

This work develops Nova, a generative foundation LLM for assembly code, which incorporates two key novelties (hierarchical attention and contrastive learning objectives) to address the unique challenges of assembly code. Evaluation on downstream tasks shows the effectiveness of Nova, which outperforms existing techniques on binary code decompilation by up to 146.54% and outperforms the latest binary code similarity detection techniques by up to 6.17%. We expect our hierarchical attention and contrastive learning techniques to benefit source code and natural language foundation models, which remains as future work.

7 Limitations

One limitation is that Nova is X86-specific, as we only collect X86 assembly corpus for pre-training. This design choice is mainly affected by two facts:

(1) X86 assembly is used and explored in a wide range of binary tasks (Wang et al., 2022; Su et al., 2024; Xu et al., 2023a; Chen et al., 2022c) compared to other assembly languages, and (2) computation limitations. However, the proposed techniques are independent of X86 assembly. Low information density and compiler optimization are the common challenges of most assembly languages such as X86, ARM, and MIPS. The proposed techniques can be applied to the future development of multi-lingual assembly LLMs.

Another potential limitation is the scale of models. We develop Nova-1B and Nova-6B. These two sized LLMs show impressive ability in assembly code decompilation and encoding. There might be potential benefit of developing larger Nova models. However, due to the computing resources limitation, we are unable to explore that in this work.

References

Simone Aonzo, Yufei Han, Alessandro Mantovani, and Davide Balzarotti. 2023. Humans vs. machines in malware classification. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1145–1162, Anaheim, CA. USENIX Association.

Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F. P. O'Boyle. 2024. Slade: A portable small language model decompiler for optimized assembly. *Preprint*, arXiv:2305.12520.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *CoRR*, abs/2005.14165.

Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. 2009. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 555–565, New York, NY, USA. Association for Computing Machinery.

Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow bending: On the effectiveness of Control-Flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C. USENIX Association.

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving code generation by training with natural language feedback. *Preprint*, arXiv:2303.16749.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022a. Codet: Code generation with generated tests. *arXiv preprint*.

Ligeng Chen, Zhongling He, Hao Wu, Fengyuan Xu, Yi Qian, and Bing Mao. 2022b. Dicomp: Lightweight data-driven inference of binary compiler provenance with high accuracy. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 112–122.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. *CoRR*, abs/2107.03374.

Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022c. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, Boston, MA. USENIX Association.

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. 2019. What does BERT look at? an analysis of BERT's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 276–286, Florence, Italy. Association for Computational Linguistics.

Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quinão Pereira. 2021. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 378–390.

Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 266–280, New York, NY, USA. Association for Computing Machinery.

Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. *SIGPLAN Not.*, 52(6):79–94.

Yaniv David, Nimrod Partush, and Eran Yahav. 2018. Firmup: Precise static detection of common vulnerabilities in firmware. *SIGPLAN Not.*, 53(2):392–404.

Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 349–360, New York, NY, USA. Association for Computing Machinery.

Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. Deepbindiff: Learning program-wide code representations for binary diffing.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. *Preprint*, arXiv:2204.05999.

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. 2019. AntiFuzz: Impeding fuzzing audits of binary executables. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1931–1947, Santa Clara, CA. USENIX Association.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. *CoRR*, abs/2302.05319.

Iman Hosseini and Brendan Dolan-Gavitt. 2022. Beyond the c: Retargetable decompilation using neural machine translation. In *Proceedings 2022 Workshop on Binary Analysis Research*, BAR 2022. Internet Society.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1430–1442. IEEE Press.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C.H. Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv preprint*, abs/2207.01780.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *Preprint*, arXiv:1910.13461.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. 2021. Neutron: an attention-based neural decompiler. *Cybersecurity*, 4(1):5.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. Rltf: Reinforcement learning from unit test feedback. *Preprint*, arXiv:2307.04349.

Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 389–400, New York, NY, USA. Association for Computing Machinery.

Jean-Phillipe Martin, Michael Hicks, Manuel Costa, Periklis Akritidis, and Miguel Castro. 2010. Dynamically checking ownership policies in concurrent c/c++ programs. *SIGPLAN Not.*, 45(1):457–470.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint*.

OpenAI. 2023. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. 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*, pages 690–702.

Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*.

Alec Radford and Karthik Narasimhan. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

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. *Preprint*, arXiv:2308.12950.

Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 117–128, New York, NY, USA. Association for Computing Machinery.

Calif. Sausalito. 2020. Cybercrime bytes: 10 hot security certs, public safety hacked, intrusion's shield. *Cybercrime Magazine*.

Chad Spensky, Hongyi Hu, and Kevin Leach. 2016. Lophi: Low-observable physical host instrumentation for malware analysis.

Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2237–2248.

Zian Su, Xiangzhe Xu, Ziyang Huang, Zhuo Zhang, Yapeng Ye, Jianjun Huang, and Xiangyu Zhang. 2024. Codeart: Better code models by attention regularization when symbols are lacking. *Preprint*, arXiv:2402.11842.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. Llm4decompile: Decompiling binary code with large language models. *Preprint*, arXiv:2403.05286.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need. *Preprint*, arXiv:1706.03762.

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*, ISSTA 2022, page 1–13, New York, NY, USA. Association for Computing Machinery.

Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1282–1294, New York, NY, USA. Association for Computing Machinery.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1482–1494. IEEE Press.

Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023a. Improving binary code similarity transformer models by semantics-driven instruction deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1106–1118, New York, NY, USA. Association for Computing Machinery.

Xiangzhe Xu, Zhuo Zhang, Shiwei Feng, Yapeng Ye, Zian Su, Nan Jiang, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. 2023b. Lmpa: Improving decompilation by synergy of large language model and program analysis. *Preprint*, arXiv:2306.02546.

Zhaoyan Xu, Antonio Nappa, Robert Baykov, Guangliang Yang, Juan Caballero, and Guofei Gu. 2014. Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 179–190, New York, NY, USA. Association for Computing Machinery.

Wang Yue, Wang Weishi, Joty Shafiq, and C.H. Hoi Steven. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832.

A Appendix

A.1 Data Collection

This section provides additional details of the data collection. To collect assemblies from The-Stack, we attempt to compile 4 million C programs, of which 138.8K is compiled successfully. We do not collect more due to the computation resource limitations.

For the 757.1K and 138.8K source code that successfully compiled into executables (using `gcc`) from AnghaBench and The-Stack, we disassemble them using `objdump`. `objdump` was not able to successfully disassemble all the executables, resulting in some empty assembly code. Thus, the number of `O0` – `O1` we obtain from each corpus is different and smaller than the number of source codes as shown in Table 1.

Figure 7 shows an example of preprocessing the raw assembly code as described in Section 2.1.

```
0:     endbr64
4:     push   %rbp
5:     mov    %rsp,%rbp
8:     mov    %rdi,-0x8(%rbp)
c:     mov    %rsi,-0x10(%rbp)
10:    mov    -0x8(%rbp),%rax
14:    mov    (%rax),%edx
16:    mov    -0x10(%rbp),%rax
1a:    mov    (%rax),%eax
1c:    cmp    %eax,%edx
...
```

(a) Raw Assembly

```
endbr64                     [INST-0]
push rbp                    [INST-1]
mov  rsp , rbp              [INST-2]
mov  rdi , -8 ( rbp )       [INST-3]
mov  rsi , -16 ( rbp )      [INST-4]
mov  -8 ( rbp ) , rax       [INST-5]
mov  ( rax ) , edx          [INST-6]
mov  -16 ( rbp ) , rax      [INSt-7]
mov  ( rax ) , eax          [INST-8]
cmp  eax , edx              [INST-9]
...
```

(b) Normalized Assembly

Figure 7: Example of assembly code preprocessing

A.2 Hierarchical Self-Attention

The hierarchical self-attention is designed for assembly code, yet the input to LLMs may still contain text or source code. Figure 8 illustrates how the hierarchical attention works with text or source code in the input. As existing LLMs have shown good performance on text and source code using the standard self-attention, we keep the standard causal attention mask within and between any chunks of text or source code in the input (the light grey part shown in Figure 8).

The attention from text or source code to assembly code (and vice versa) is only allowed for the "`[INST]`" tokens as they are supposed to contain the assembly instruction summaries.

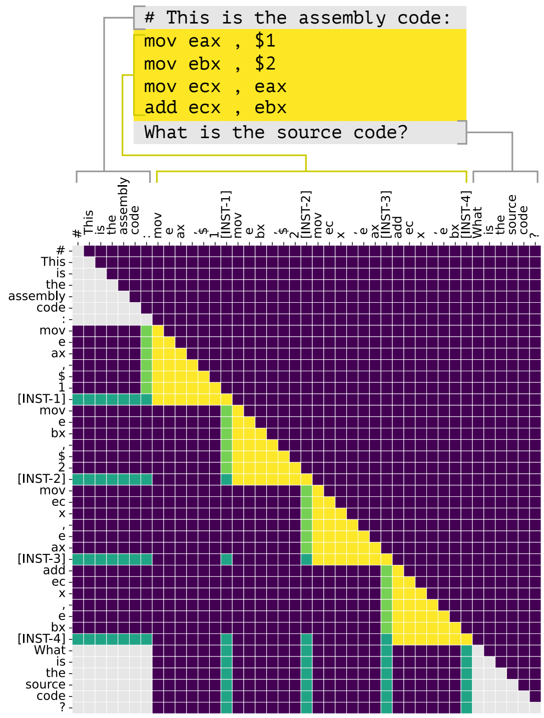

Figure 8: Hierarchical attention with text input.

A.3 Training Details

This section provides additional details of training. We pre-train Nova starting from DeepSeek-Coder, using the language modeling objective (L_{lm}) for one epoch on the AnghaBench and The-Stack corpora. This step uses a batch size of 128, with the input truncated by a 1,024 tokens limit. The model weights are updated using the AdamW optimizer. The learning rate is $5e^{-5}$, using 1000 steps of warm-up and a cosine decay to adjust the learning rate.

Then, the model is further pre-trained with the combination of language modeling and contrastive

Benchmarks	Nova$_{-CL-HA}$	Nova$_{-HA}$	Nova-1B
Binutils	0.86	0.88	0.87
Curl	0.84	0.87	0.89
ImageMagick	0.79	0.80	0.86
SQLite	0.80	0.83	0.77
OpenSSL	0.90	0.92	0.90
Putty	0.68	0.66	0.72
Avg.	0.81	0.83	0.84

Table 10: Ablation study with $K = 50$.

Benchmarks	Nova$_{-CL-HA}$	Nova$_{-HA}$	Nova-1B
Binutils	0.80	0.82	0.79
Curl	0.84	0.84	0.86
ImageMagick	0.70	0.72	0.79
SQLite	0.74	0.78	0.73
OpenSSL	0.89	0.89	0.88
Putty	0.59	0.60	0.65
Avg.	0.76	0.78	0.78

Table 11: Ablation study with $K = 100$.

Benchmarks	Nova$_{-CL-HA}$	Nova$_{-HA}$	Nova-1B
Binutils	0.71	0.74	0.73
Curl	0.80	0.73	0.83
ImageMagick	0.61	0.63	0.73
SQLite	0.68	0.71	0.68
OpenSSL	0.85	0.87	0.84
Putty	0.53	0.53	0.55
Avg.	0.70	0.70	0.73

Table 12: Ablation study with $K = 200$.

Benchmarks	Nova$_{-CL-HA}$	Nova$_{-HA}$	Nova-1B
Binutils	0.62	0.65	0.65
Curl	0.67	0.71	0.73
ImageMagick	0.46	0.51	0.61
SQLite	0.61	0.62	0.59
OpenSSL	0.77	0.79	0.78
Putty	0.46	0.46	0.47
Avg.	0.60	0.62	0.64

Table 13: Ablation study with $K = 500$.

learning objectives ($L = L_{lm} + \lambda(L_{fcl} + L_{ocl})$), with λ set to 0.1. To train with the functionality contrastive learning objective, we discard any source code that misses any one of `O0` – `O3` assemblies and also discard the source code whose `O2` assembly is the same as its `O3` assembly. As a result, this step is only trained for 0.36M data samples for one epoch. The batch size is 64, with the input truncated by a 1,024 tokens limit. The learning rate is $2e^{-5}$ using the AdamW optimizer.

The fine-tuning of both BCD and BCSD uses a batch size of 64, with the input truncated by a 2,048 token limit. Similarly, the learning rate is $2e^{-5}$ using the AdamW optimizer, and the model is fine-tuned for one epoch.

Infrastructure The training are conducted on eight NVIDIA RTX A100 GPUs, each with 40GB memory. Our implementation is based on Huggingface's implementation of DeepSeek-Coder [1], PyTorch [2], and DeepSpeed [3].

A.4 Binary Code Decompilation Case Studies

Figure 9 shows an example from HumanEval-Decompile (`task_id 0`). Given the `O1`-optimized assembly code, GPT-4 fails to figure out the number of function arguments correctly, missing one important argument "`float e`", and thus produces wrong functionality in the decompiled code. LLM4Decompile-1B makes similar mistakes and also misses the inner nested `for` loop. Nova-1B correctly decompiles the assembly into source code,

[1] https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-base

[2] https://pytorch.org/get-started/locally/

[3] https://github.com/microsoft/DeepSpeed

```
<func0>:
0:   endbr64
4:   test   %esi, %esi
6:   jle    48 <func0+0x48>
8:   lea    -0x1(%rsi), %ecx
b:   add    $0x1, %rcx
f:   mov    $0x1, %edx
14:  movss  0x0(%rip), %xmm3
1b:  00
1c:  cmp    %rcx, %rdx
1f:  je     4e <func0+0x4e>
21:  movss  -0x4(%rdi, %rdx, 4), %xmm2
27:  mov    %rdx,%rax
2a:  movaps %xmm2, %xmm1
2d:  subss  (%rdi, %rax, 4), %xmm1
32:  andps  %xmm3, %xmm1
35:  comiss %xmm1, %xmm0
38:  ja     54 <func0+0x54>
3a:  add    $0x1, %rax
3e:  cmp    %eax, %esi
40:  jg     2a <func0+0x2a>
42:  add    $0x1, %rdx
46:  jmp    1c <func0+0x1c>
48:  mov    $0x0, %eax
4d:  retq
4e:  mov    $0x0, %eax
53:  retq
54:  mov    $0x1, %eax
59:  retq
```

O1 Assembly Code

```
int func0(float *arr, int n) {
  for (int i = 1; i < n; i++) {
    for (int j = 0; j < i; j++) {
      if (fabs(arr[i] - arr[j]) < 1.0f)
        return 1;
    }
  }
  return 0;
}
```

GPT-4's Incorrect Decompilation

```
int func0(const float *x, int n) {
  int i;
  float diff = 1.0f;
  for (i = 1; i < n; i++) {
    if (fabs(x[i] - x[i-1]) > diff)
      return 0;
  return 1;
}
```

LLM4Decompile's Incorrect Decompilation

```
int func0(float *a, int n, float e) {
  int i, j;
  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      if (fabs(a[i] - a[j]) < e)
        return 1;
  return 0;
}
```

Nova's Correct Decompilation

Figure 9: Nova-1B correctly decompiles HumanEval-Decompile task 0, while GPT-4 and LLM4Decompile-1B fail.

```
<func0>:
0:      endbr64
4:      movss  (%rdi), %xmm0
8:      movss  %xmm0, (%rdx)
c:      movss  0x4(%rdi), %xmm1
11:     movss  %xmm1, 0x4(%rdx)
16:     test   %esi, %esi
18:     jle    8b <func0+0x8b>
1a:     lea    -0x1(%rsi), %r9d
1e:     mov    $0x1, %r8d
24:     mov    %rdi, %rcx
27:     movss  0x0(%rip), %xmm2
2e:     00
2f:     add    $0x1, %r9
33:     movss  0x0(%rip), %xmm3
3a:     00
3b:     cmp    %r8, %r9
3e:     je     82 <func0+0x82>
40:     mov    %r8, %rax
43:     nopl   0x0(%rax, %rax, 1)
48:     movss  (%rcx), %xmm1
4c:     movaps %xmm1, %xmm0
4f:     subss  (%rdi, %rax, 4), %xmm0
54:     andps  %xmm3, %xmm0
57:     comiss %xmm0, %xmm2
5a:     jbe    6d <func0+0x6d>
5c:     movss  %xmm1, (%rdx)
60:     movss  (%rdi, %rax, 4), %xmm1
65:     movaps %xmm0, %xmm2
68:     movss  %xmm1, 0x4(%rdx)
6d:     add    $0x1, %rax
71:     cmp    %eax, %esi
73:     jg     48 <func0+0x48>
75:     add    $0x1, %r8
79:     add    $0x4, %rcx
...
99:     retq
```

O2 Assembly Code

```
void func0(float *a, int n, float *b) {
  int i, j;
  float min, max;

  min = max = a[0];
  for (i = 1; i < n; i++) {
    if (a[i] < min)
      min = a[i];
    else if (a[i] > max)
      max = a[i];
  }
  b[0] = min;
  b[1] = max;
}
```

LLM4Decompile's Incorrect Decompilation

```
void func0(float *a, int n, float
*minmax) {
  minmax[0] = a[0];
  minmax[1] = a[1];

  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      float diff = fabs(a[i] - a[j]);
      if (diff < minmax[1] - minmax[0])
      {
        minmax[0] = a[i];
        minmax[1] = a[j];
      }
    }
  }

  if (minmax[0] > minmax[1]) {
    float temp = minmax[0];
    minmax[0] = minmax[1];
    minmax[1] = temp;
  }
}
```

Nova's Correct Decompilation

Figure 10: Nova-1B correctly decompiles HumanEval-Decompile task 20, while LLM4Decompile-1B fail.

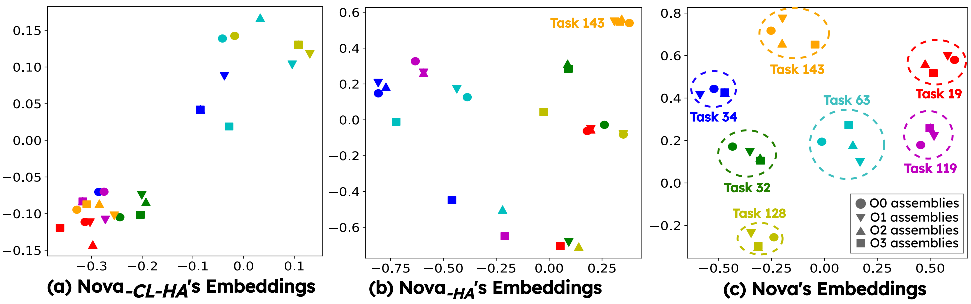

Figure 11: PCA of embeddings calculated by Nova$_{-CL-HA}$, Nova$_{-HA}$, and Nova.

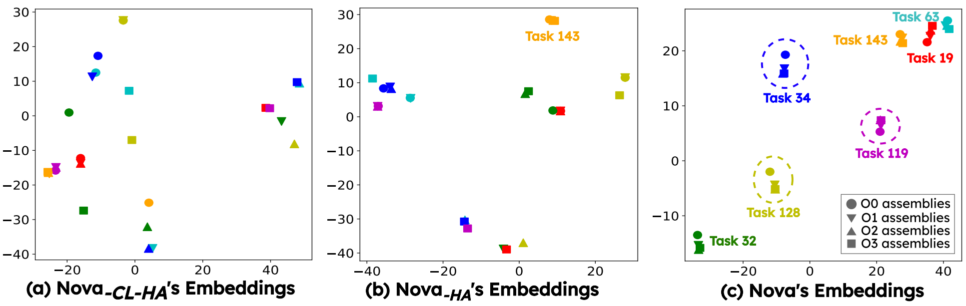

Figure 12: t-SNE of embeddings cauclated by Nova$_{-CL-HA}$, Nova$_{-HA}$, and Nova.

where the ground truth is checking if any two elements in the given list `*a` (with size `n`) are close to each other than a given threshold `e`.

Figure 10 shows another more complex example, HumanEval-Decompile `task_id` 20. Nova-1B correctly decompiles the source code, successfully figuring that the function is trying to find the two elements that are closest to each other in the given array `*a`, with the result stored in `minmax`.

A.5 Binary Code Similarity Detection Ablation Study

Table 10, 11, 12, 13 show the detailed ablation study results of BCSD. Nova wins on the most benchmarks when $K = 100$ or 500, and ties with Nova$_{-HA}$ when $K = 50$, or 200.

A.6 Additional Analysis of Embedding

Additional Embedding Visualization Figure 11 shows the full results of PCA of embeddings provided by Nova$_{-CL-HA}$, Nova$_{-HA}$, and Nova, on randomly sampled seven examples. Compared with Nova$_{-CL-HA}$, Nova$_{-HA}$ including contrastive learning objectives in the pre-training, can separate the embeddings of assemblies with different functionalities better. Nova$_{-HA}$ clearly encode "`Task 143`" (orange points) away from the others. Nova's embeddings group the assemblies by functionalities more precisely than Nova$_{-HA}$, suggesting that hierarchical attention enhances the training of contrastive learning objectives to learn more effective encoding.

Figure 12 shows the results using another dimensionality reduction technique, t-SNE (van der Maaten and Hinton, 2008), where Nova's embeddings are consistently more distinguishable by functionalities.

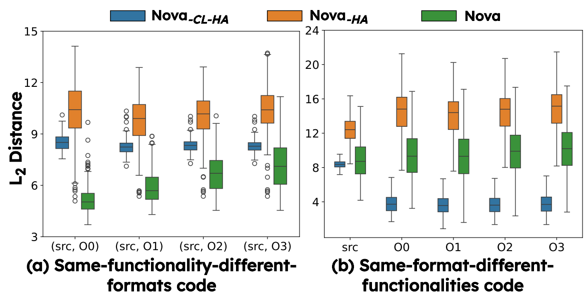

Figure 13: Distances between embeddings of different-formats-same-functionality code and same-format-different-functionalities code.

Quantitative Analysis We also perform quantitative analysis to further support the conclusion from visualization. Figure 13 (a) shows the distribution of l_2 distances between different-formats-same-functionality code in HumanEval-Decompile ("format" is defined as one of source code, `O0`, `O1`, `O2`, or `O3` assembly). The figure shows five distributions, `(src, O0)`, `(src, O1)`, `(src, O2)`, and `(src, O3)`. Each distribution calculates the distances between embeddings of two formats of code, e.g., `src, O0` refers to the source code and `O0` assembly for the same task in HumanEval-Decompile.

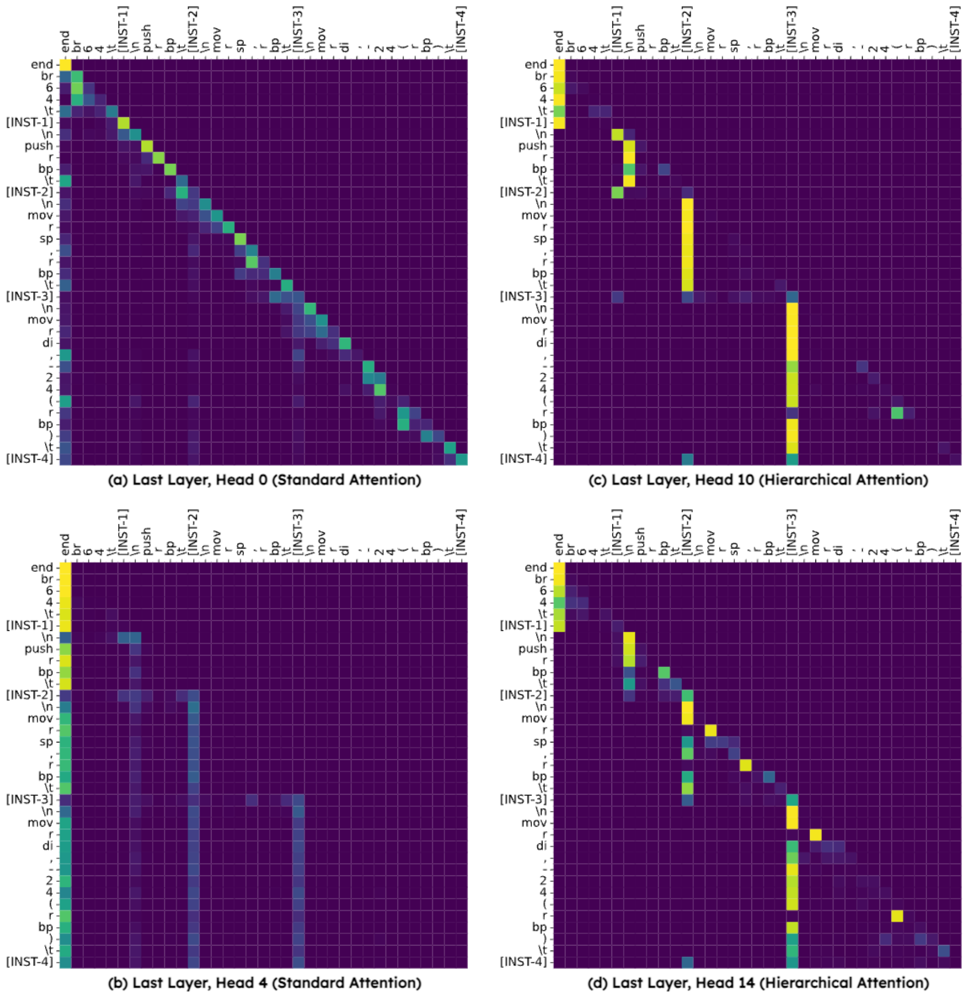

Figure 14: Comparison of attention distribution among standard and hierarchical heads in the final layer.

Figure 13 (a) shows that Nova's embeddings for same-functionality-different-formats code are closer to each other in the latent space. For the same task, all the `O0` – `O3` assemblies' embeddings have a smaller l_2 distance to the source code embedding compared to that of Nova$_{-CL-HA}$'s and Nova$_{-HA}$'s embeddings. Another interesting finding is that for Nova, the distances between assembly and source code increase with the optimization level of assemblies increases. This trend matches what the optimization CL tries to optimize for.

Figure 13 (b) shows the distribution of l_2 distance between same-format-different-functionalities code from the HumanEval-Decompile benchmark. With the same format (source code or the same optimized assembly), Nova encodes the assemblies with different functionalities farther away from each other compared to Nova$_{-CL-HA}$. Nova$_{-CL-HA}$'s embeddings cannot significantly reflect the functionality differences between assemblies, while Nova's embeddings can.

Nova$_{-HA}$'s embeddings show high l_2 distance in both Figure 13 (a) and (b), suggesting that Nova$_{-HA}$ tries to decentralize all the code in the embedding space even if they have the same functionality, which is not as desired as Nova.

A.7 Additional Analysis of Attention

Figure 14 shows the visualizations of attention weights in the final transformer layer of two select heads with standard attention and two heads with learned hierarchical attention. Standard attention exhibits two typical patterns, namely diagonal attention (i.e. tokens attending to themselves or nearby tokens, shown in Figure 14 (a)), and broad attention (i.e. a single token attending broadly to the entire sequence, shown in Figure 14 (b)). In contrast, in Nova's hierarchical attention, attention weights are allocated among distinct segments, each corresponding to an instruction (shown in Figure 14 (c)), that focus on tokens comprising that instruction (e.g. opcodes and operands, shown in Figure 14 (d), attentions are paid to "`push`", "`mov`",

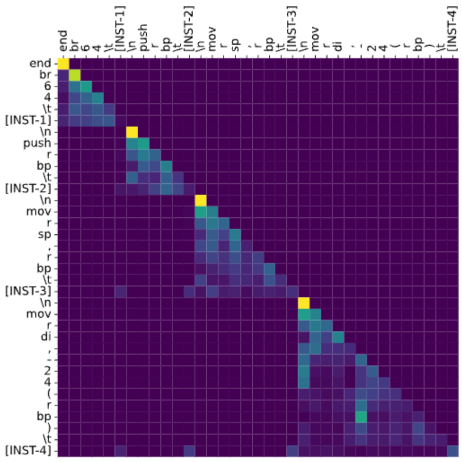

Figure 15: Learned per-instruction soft attention observed in the lower layers

etc.).

Quantitatively, we have determined broad attention accounts for as much as 30% of all attention in standard heads, especially in layers 1-8 (consistent with the findings of (Clark et al., 2019)), whereas in Nova's hierarchical attention, no more than 5% or all attention is allocated to each instruction segment. This validates our goal of learning instruction-aware hierarchical attention in Nova.

In addition, in lower layers, we have observed attention weights to be softly distributed among tokens comprising each instruction (Figure 15), which suggests Nova initially models cross-relations among operation codes and operands in the first few layers, and later pools their summary representation into the `[INST]` token in the later layers. This is also supported by Figure 6, where the Nova's hierarchical attention (red line) shows a decreasing trend of entropy. This means the hierarchical attention is softer in lower layers, and becomes focused in higher layers.

A.8 Potential Risks

This work develops generative LLMs, Nova, for assembly code, aiming to benefit the downstream research domains in binary analysis. The training corpora collected to develop the Nova are all open-sourced and widely used by existing work. The Nova models are built on top of open-sourced foundation LLMs on natural language text and source code. Thus, we think no special concerns need to be highlighted here.