SLICEFORMER: Static Program Slicing Using Language Models With Dataflow-Aware Pretraining and Constrained Decoding

Anonymous ACL submission

Abstract

Static program slicing is a fundamental software engineering technique for isolating code relevant to specific variables. While recent learning-based approaches using language models (LMs) show promise in automating slice prediction, they suffer from *inaccurate dependency modeling* and *unconstrained generation*, where LMs fail to capture precise data flow relations and produce slices containing hallucinated tokens and statements. To address these challenges, we propose SLICEFORMER, a novel approach that reformulates static program slicing as a sequence-to-sequence task using small language models such as CodeT5+. SLICEFORMER introduces two key innovations that directly target the identified limitations. First, to improve dependency modeling, we design *dataflow-aware pretraining objectives* that leverage data flow graphs ($\mathcal{DFG}$) to teach models data dependencies through dataflow-preserving statement permutation and dataflow-aware span corruption. Second, to eliminate hallucination, we develop a *constrained decoding* mechanism that enforces both lexical and syntactic constraints. We evaluate SLICEFORMER on Java and Python program slicing benchmarks, demonstrating consistent improvements over state-of-the-art baselines with up to 22% gain in ExactMatch.

1 Introduction

Static program slicing identifies code relevant to a given slicing criterion and has proven essential for vulnerability analysis (Zou et al., 2019; Li et al., 2018) and debugging (Weiser, 1984; Xu et al., 2005; Song et al., 2020). Unlike dynamic slicing, static slicing operates on the code dependence graph constructed via static analysis, without requiring program execution (Harman and Hierons, 2001), thereby offering broader practicality (Acharya and Robinson, 2011; Xu et al., 2005).

Recent work has explored learning-based approaches for program slicing using LMs (Yadavally et al., 2024; Shahandashti et al., 2024). However, the learning-based approaches face significant hurdles, as demonstrated by our empirical investigation (Section 2.2): ❶ **Inaccurate dependency identification** (Yadavally et al., 2024): general LLMs or directly fine-tuned LMs fail to precisely capture data dependencies, frequently missing relevant statements or including extraneous ones. ❷ **Unconstrained generation**: employing large proprietary LMs with prompting techniques (e.g. CoT) suffers from hallucination (Shahandashti et al., 2024), where models generate hallucinated tokens or statements not in the original code, violating the requirement that slices must be exact subsequences of the input.

To address these challenges, we propose SLICEFORMER, which enhances standard supervised fine-tuning (SFT) with two key contributions: **dataflow-aware pretraining** and **constrained decoding**. The former equips the model with dataflow semantic understanding before SFT, while the latter operates at inference time to prevent hallucinated tokens and invalid statement generation.

Dataflow-aware pretraining: We introduce two novel pretraining objectives based on Data Flow Graph ($\mathcal{DFG}$) structures (Guo et al.): (1) *Dataflow-preserving statement permutation*, which trains the model to identify statement dependencies by generating valid code permutations that respect dataflow constraints; (2) *Dataflow-aware span corruption*, which employs dataflow-guided masking to force the model to reconstruct code based on dependency relationships rather than superficial patterns.

Constrained decoding: We propose a decoding mechanism that enforces two constraints: (1) **Lexical constraint**, which restricts the vocabulary to tokens appearing in the original code; and (2) **Syntactic constraint**, which leverages the monotonicity of Tree Similarity Edit Distance (TSED) (Song et al., 2024) to filter AST-invalid candidates. These constraints mitigate hallucinated tokens and invalid

statement generation.

We evaluate SLICEFORMER on Java and Python program slicing tasks. SLICEFORMER consistently outperforms state-of-the-art baselines (Shahandashti et al., 2024; Yadavally et al., 2024) across all evaluation metrics, achieving ExactMatch improvements of 6.4% and 21.9%, respectively.

In summary, we make the following contributions:

- We propose SLICEFORMER, an approach that integrates dataflow-aware pre-training objectives and constrained decoding to explicitly capture data relations for accurate program slicing.

- To the best of our knowledge, we are the first to introduce novel dataflow-aware statement permutation and span corruption objectives for pre-training, and TSED-monotonicity constraint for decoding.

- Through extensive evaluation on Java and Python benchmarks, SLICEFORMER consistently outperforms state-of-the-art static and learning-based slicing baselines.

- We publicly release our code and datasets[1].

2 Problem Formulation and Challenges

We formulate program slicing as a sequence-to-sequence learning problem and identify key limitations that motivate our approach.

2.1 Problem Formulation

We formulate **static program slicing** as a sequence-to-sequence learning task. The input is:

$$\boldsymbol{x} = \{s_1, s_2, \ldots, s_N; v; [n]\} \tag{1}$$

where s_i is the i-th statement, v is the slicing criterion (variable of interest), and $[n]$ is the line number where v appears. Given $\boldsymbol{x}$, a slicer LM $P(\boldsymbol{y}|\boldsymbol{x})$ to predict the *backward program slice* (Weiser, 1984) $\boldsymbol{y}$:

$$\boldsymbol{y} = \{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\} \subseteq \{s_1, s_2, \ldots, s_n\}, \quad i_1 < i_2 < \cdots < i_k \tag{2}$$

where $\boldsymbol{y} \subseteq \boldsymbol{x}$ comprises the statements that semantically influence v. The output must satisfy:

[1]https://anonymous.4open.science/r/staticsliceT5-4E22

```
// Example (1) Inaccurate
      dependency
      identification
// Expected:
7: int temp
8: if(C <= A){
12: temp = B;

// Generated:
7: int temp
8: if(C <= A){
9: temp = A;
10:A = C;
12: temp = B;
```

```
// Example (2) Non-existent
      variable replacement
// Expected:
7: int cnt = 0;
10: for(int i=cnt;i>=0;i--)
     {
11: if(i>0) {long y = x[i];
12: long Codepoint = 97+y};

// Generated:
7: int cnt = 0;
10: for(int i=cnt;i>=0;i--)
     {
11: if(i>0) {long y = x[i];
12: long keta = 97+y};
```

```
// Example (3) Hallucinated statement generation
// Expected:
4: int one = 0, five = 0, ten = n;
5: try {
6: if (one * 1 + five * 5 + ten * 10 > y)

// Generated:
4: int one = 0, five = 0, ten = n;
5: try {
6: if (one * 1 + five * 5 + ten * 10 > y
   * 10 * y * y * y * z * ten * 10 * y * z * ten *
```

Figure 1: Motivating examples of incorrect program slices produced by LMs.

- **Accuracy**: $\boldsymbol{y}$ includes all and only relevant statements, with no extraneous or missing statements.

- **Element preservation**: $\boldsymbol{y}$ is an exact subsequence of $\boldsymbol{x}$: every token and statement must be precisely extracted from $\boldsymbol{x}$ without modification or hallucination.

2.2 Related Work and Limitations

Traditional static slicing tools, such as JavaSlicer (group, 2022) and CPP-Slicer (Hechtl, 2021), rely on static dependency analysis to construct System Dependence Graphs (SDGs) and compute slices via graph reachability (Galindo et al., 2022).

Recently, learning-based approaches have attracted growing interest (Yadavally et al., 2024; Shahandashti et al., 2024). Yadavally et al. (2024) predict dependencies between slicing criteria and program statements via binary classification using CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al.) embeddings. Although GraphCodeBERT incorporates dataflow-aware pre-training, it is an embedding model with objectives that differ from ours. Moreover, NS-Slicer is not end-to-end and operates on isolated statements, which restricts the available context to partial functions and consequently limits slicing

152 performance. Shahandashti et al. (2024) investigated LLMs (e.g., GPT-4o (Achiam et al., 2023), GPT-3.5-Turbo, and Gemma (Mind, 2024)) combined with advanced prompting techniques such as Retrieval-Augmented Generation (RAG) and Chain-of-Thought (CoT). However, proprietary LLMs are computationally expensive and prone to severe hallucinations (Liu et al., 2024; Zhang et al., 2025b), which limits their reliability for program analysis tasks.

As a result, existing learning-based slicing methods suffer from the following two key challenges:

❶ **Inaccurate dependency identification.** LMs struggle to precisely capture data dependencies, often omitting relevant statements or including irrelevant ones. For instance, in Figure 1 (Example 1), the generated slice incorrectly includes lines 9–10 (`temp = A; A = C;`), which are unrelated to the slicing criterion. Instead of identifying true dependencies, the model relies on surface-level patterns or positional proximity, violating the *accuracy* property.

❷ **Unconstrained generation.** LMs hallucinate elements absent from the original program, violating *element preservation*. We observe: **(a) Token-level:** original identifiers are replaced with hallucinated ones (e.g., `keta` instead of `codepoint` in Example 2), leading to syntactic errors (Wang et al., 2025); **(b) Statement-level:** spurious logic or repetitions are introduced (e.g., `y * 10 * y * y * y * z * ten ...` in Example 3 includes incorrect sub-expressions such as `z * ten * 10 * y`).

3 Methodology

We introduce novel **dataflow-aware pre-training** and **lexical–syntactic constrained decoding** to enhance SFT-based LMs for accurate, element-preserving program slicing.

Dataflow-Aware Pretraining. To tackle Challenge ❶, we introduce a pretraining strategy that explicitly models program dependencies via two objectives. First, *dataflow-preserving statement permutation* (Section 3.1.1) trains the model to recognize statement dependencies by generating valid permutations that respect dataflow constraints. Second, *graph-aware span corruption* (Section 3.1.2) applies $\mathcal{DFG}$-guided masking to force code reconstruction based on dependency relationships. We then perform SFT on labeled slicing data to specialize the model for slicing.

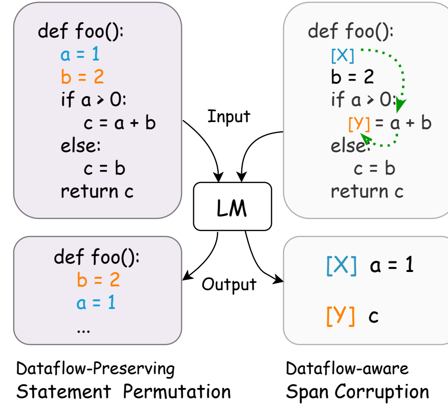

Figure 2: Illustration of dataflow-aware pre-training objectives. Left: given a snippet, we generate a statement permutation that preserves the dataflow of the original program. Right: we select a variable a with a dataflow chain $[X] \rightarrow a \rightarrow [Y]$, then mask its parent $[X]$ (where it comes from) and child $[Y]$ (where it flows to). The pre-training objective is to recover both $[X]$ and $[Y]$.

Constrained Decoding. To address Challenge ❷, we design a training-free constrained decoding mechanism (Section 3.2) that enforces lexical and syntactic correctness during inference. We apply two complementary constraints: **(1) lexical constraints**, which restrict generation to tokens appearing in the input code, and **(2) syntactic constraints**, which evaluate structural coherence at statement boundaries using Tree Similarity of Edit Distance (TSED) (Song et al., 2024). Since valid slices are subsequences of the input, TSED should increase monotonically. When the TSED score decreases, indicating structural errors, we terminate that generation path. These constraints ensure extractive, structurally sound slices.

3.1 Dataflow-Aware Pretraining

To enable the model to better understand data relations crucial for program slicing, we introduce two novel dataflow-aware pretraining objectives. Figure 2 illustrates our dataflow-aware pretraining objectives.

Data Flow Graph ($\mathcal{DFG}$). Given source code C, $\mathcal{DFG} = (V, E)$, where $V = \{v_1, \ldots, v_k\}$: The variable sequence extracted from AST terminals (leaves). and Edges $E = \{\varepsilon_1, \ldots, \varepsilon_l\}$: A directed edge $\varepsilon = \langle v_i, v_j \rangle$ indicates that there is a data dependency between the two variables (Guo et al.). $\mathcal{DFG}$s are the core inherent structure for determining program slices (Galindo et al., 2022).

Algorithm 1: Dataflow-Preserving Statement Permutation

```
Input  : Code snippet C = {s_1, s_2, ..., s_n}, DFG = (V, E)
Output : Permuted code snippet C'
1  P ← ∅
2  foreach pair (s_i, s_j) where i < j do
3      // Check independence: same basic block
           and no data edge
4      hasEdge ← false
5      foreach v_a ∈ s_i, v_b ∈ s_j do
6          if ⟨v_a, v_b⟩ ∈ E or ⟨v_b, v_a⟩ ∈ E then
7              hasEdge ← true; break
8          end if
9      end foreach
10     if SameBasicBlock(s_i, s_j) and ¬hasEdge
         then
11         P ← P ∪ {(s_i, s_j)}
12     end if
13 end foreach
14 // Apply random permutation if possible
15 if P ≠ ∅ then
16     (s_i, s_j) ← random pair from P
17     C' ← swap s_i and s_j in C
18 else
19     C' ← C
20 end if
21 return C'
```

Algorithm 2: Dataflow-Aware Span Corruption

```
Input  : Code snippet C, DFG = (V, E), Mask ratio r
Output : Masked code snippet C_masked
1  m ← r × Size(C) // Target masked tokens
2  masked ← 0
3  // Iteratively mask
4  while masked < m and V ≠ ∅ do
5      // Randomly select a variable from
           DFG
6      v ← Random(V)
7      // Find parents (where v comes from/
           who defines v)
8      Parents ← {u | ⟨u, v⟩ ∈ E}
9      // Find children (where v goes / who
           uses v)
10     Children ← {w | ⟨v, w⟩ ∈ E}
11     // Randomly decide masking granularity
12     if Random () < 0.5 then
13         // Fine-grained: mask variables
14         Mask(Parents ∪ Children)
15     else
16         // Coarse-grained: mask statements
17         S_parents ← {GetStmt(u) | u ∈
             Parents}
18         S_children ← {GetStmt(w) | w ∈
             Children}
19         Mask(S_parents ∪ S_children)
20     end if
21     masked ← masked + Size(M)
22 end while
23 return C_masked ← C
```

3.1.1 Dataflow-Preserving Statement Permutation

Conventional permutation-based pretraining methods (Lewis et al., 2020) randomly permute sentences and train models to recover the original order for capturing their semantic dependency. In code, however, legal orderings are not unique: a given $\mathcal{DFG}$ permits multiple valid statement orders. Statements without data dependencies within the same basic block can be reordered without affecting program semantics. As illustrated in Figure 2, the two independent statements in a basic block are equivalent under any ordering.

To improve the model's capability to recognize the true dependencies, we introduce a dataflow-preserving statement permutation pretraining objective. Given a code snippet, we train the model to generate equivalent variants while preserving dataflow invariance. The training labels are obtained by reordering independent statements within each basic block as shown in Algorithm 1. This forces the model to learn which statements are data-independent, a fundamental capability for identifying relevant statements in program slicing.

3.1.2 Dataflow-Aware Span Corruption

Span corruption is a well-established pre-training objective for language models, in which consecutive tokens are masked and replaced with sentinel tokens (Raffel et al., 2020; Lewis et al., 2020). AST-T5 (Gong et al., 2024) extends this paradigm by incorporating syntactic structure, masking subtrees derived from abstract syntax trees (ASTs) to improve LM's ability to comprehend ASTs . Compared to ASTs, data-flow graphs are structurally simpler and avoid unnecessarily deep hierarchies (Guo et al.). As a result, $\mathcal{DFG}$s are well-suited for program slicing, where capturing long-range data dependencies is crucial. Therefore, we design pre-training objective based on $\mathcal{DFG}$s to improve LM's ability to understand data-flow.

Two-granularity dataflow-guided masking. A core design principle of our approach is that, given a node in the $\mathcal{DFG}$, the model is trained to reconstruct where the value comes from and where it flows to. Accordingly, each masked span corresponds to semantically coherent code elements connected by def–use relations. To increase the diversity of reconstruction objectives, we perform masking at two different granularities (Algorithm 2):

For example, in Figure 2, given the variable node `a` in the statement `c = a + b`, the value of `a` is

defined in the statement `a = 1` and subsequently flows into `c`. In this case, we apply coarse-grained masking to the defining statement `a = 1`, and fine-grained masking to the variable usage `c` in `c = a + b`. This forces the model to reconstruct the dataflow chain associated with `a`, explicitly learning variable-level data dependencies.

By combining these two granularities, our approach enables the model to capture both fine-grained variable-level dependencies and coarse-grained statement-level dataflow, while ensuring that every masked unit remains semantically coherent within the DFG.

3.1.3 Supervised Fine-Tuning

Our dataflow-aware pretraining objectives endow the model with a strong understanding of data dependencies. We then perform supervised fine-tuning (SFT) on labeled slicing data to directly train the model for program slicing. To encourage structured, task-specific outputs, we augment the input–output format with special control markers, following prior work on conditional text generation (Narayan et al., 2023; Li et al., 2021; Zhu et al., 2024). Specifically, we introduce markers to denote line numbers, code, slicing criteria, and slices (e.g., *<code>*, *<criterion>*, *<slice>*), which provide explicit structural cues and guide the model to generate well-formed outputs.

3.2 Constrained Decoding

While our dataflow-aware pretraining provides the model with a strong understanding of code dependencies, it does not guarantee that the generated slices will be free from hallucinated tokens or structural errors. To address this, we introduce a constrained decoding mechanism.

Our approach modifies the standard beam search by applying two types of constraints at different granularities, as shown in Algorithm 3 and Figure 5. At each token generation step, we apply a *lexical constraint* to restrict the vocabulary to only tokens appearing in the original code snippet. Also, when a complete statement is generated, we apply a *syntactic constraint* at the statement boundary to filter out candidates that violate structural coherence.

3.2.1 Lexical Constraint

A challenge in applying LMs to program slicing is their tendency to produce *hallucinated* tokens: identifiers, keywords, or operators absent from the input code (Figure 1). This occurs because LMs decode from an unconstrained vocabulary. For example, CodeT5+ samples from a fixed vocabulary of 32,100 tokens (Wang et al., 2023) without awareness of which tokens appear in the input.

To address this issue, we introduce a *lexical constraint* that restricts decoding to tokens present in the input sequence. Specifically, any token absent from the original code snippet $\boldsymbol{x}$ is assigned a logit of $-\infty$ before softmax, ensuring zero probability. This hard constraint prevents hallucinated identifiers and enforces strictly extractive token generation, producing slices that are lexically faithful to the original code snippet.

Algorithm 3: Constrained Beam Search with Lexical Constraint and Syntactic Constraint

Input : Input $\boldsymbol{x}$; LM $\mathcal{M}$; Vocabulary $\mathcal{V}$; Beam size K; Max output length L
Output : Program slice $\mathcal{Y}$

1. `// Determine lexically allowed tokens`
2. $\mathcal{A} \leftarrow$ `GetAllowedTokens`$(\boldsymbol{x})$
3. `// Initialize beam`
4. $\mathcal{B} \leftarrow \{(\boldsymbol{y} = [], s = 0, t_{stmt} = 0)\}$
5. **for** $t = 1$ **to** L **do**
6. $\quad \mathcal{B}_{\text{next}} \leftarrow \emptyset$
7. $\quad$ **foreach** beam $(\boldsymbol{y}, s, t_{stmt}) \in \mathcal{B}$ **do**
8. $\quad\quad$ `// Apply Lexical Constraint`
9. $\quad\quad$ $mask \leftarrow$ `ApplyMask`$(\mathcal{A}, \mathcal{V})$
10. $\quad\quad$ $\boldsymbol{p} \leftarrow$ `NextTokenScores`$(\mathcal{M}, \boldsymbol{y}, mask)$
11. $\quad\quad$ $\{(z_k, p_k)\}_{k=1}^{K} \leftarrow$ `TopK`$(\boldsymbol{p}, K)$
12. $\quad\quad$ `// Expand each beam`
13. $\quad\quad$ **for** $k = 1$ **to** K **do**
14. $\quad\quad\quad$ $\boldsymbol{y}' \leftarrow \boldsymbol{y} \parallel z_k$
15. $\quad\quad\quad$ $s' \leftarrow s + \log(p_k)$
16. $\quad\quad\quad$ $t'_{stmt} \leftarrow t_{stmt}$
17. $\quad\quad\quad$ `// Apply Syntactic Constraint at statement boundary`
18. $\quad\quad\quad$ **if** `IsStatementComplete` $(\boldsymbol{y}')$ **then**
19. $\quad\quad\quad\quad$ $t_{\text{cur}} \leftarrow$ `TSED`$(\boldsymbol{x}, \boldsymbol{y}')$
20. $\quad\quad\quad\quad$ `// Skip if TSED decreases (syntactic error)`
21. $\quad\quad\quad\quad$ **if** $t_{\text{cur}} < t_{stmt}$ **then**
22. $\quad\quad\quad\quad\quad$ **continue**
23. $\quad\quad\quad\quad$ **end if**
24. $\quad\quad\quad\quad$ $t'_{stmt} \leftarrow t_{\text{cur}}$
25. $\quad\quad\quad$ **end if**
26. $\quad\quad\quad$ `// Skip if end of sequence`
27. $\quad\quad\quad$ **if** `IsEOS` (z_k) **then**
28. $\quad\quad\quad\quad$ **continue**
29. $\quad\quad\quad$ **end if**
30. $\quad\quad\quad$ $\mathcal{B}_{\text{next}} \leftarrow \mathcal{B}_{\text{next}} \cup \{(\boldsymbol{y}', s', t'_{stmt})\}$
31. $\quad\quad$ **end for**
32. $\quad$ **end foreach**
33. $\quad$ $\mathcal{B} \leftarrow$ `TopK`$(\mathcal{B}_{\text{next}}, K)$
34. **end for**
35. `// Return the top-1 output sequence`
36. **return** $\mathcal{Y} \leftarrow \arg\max_{(\boldsymbol{y}, s, t_{stmt}) \in \mathcal{B}} s$

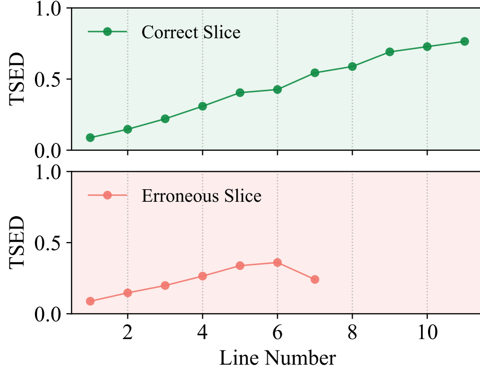

Figure 3: TSED scores at statement boundaries for syntactically correct versus erroneous program slices based on the same input code snippet. The scores are derived from Example (3) in Figure 1. When a syntactic error is introduced into the generated slice (Line 6), the TSED score deviates from its expected monotonic increasing pattern and instead decreases at the statement boundary.

3.2.2 Syntactic Constraint

While the lexical constraint restricts generation to input tokens, it is *order-agnostic* and does not ensure structural coherence. As a result, the model may still produce syntactically invalid slices, such as *over-generation* (Tu et al., 2016), where tokens, though individually valid, are repeated or misaligned. For example, Example (3) in Figure 1 shows a slice that satisfies the lexical constraint but contains a structural error due to incorrect ordering.

To address this limitation, we introduce a *syntactic constraint* based on the Tree Similarity of Edit Distance (TSED) (Song et al., 2024), a similarity metric that quantifies the syntactic distance between two code snippets by comparing their abstract syntax trees (ASTs). TSED is defined as:

$$TSED(T_x, T_y) = 1 - \frac{\min_{\text{ops}} \sum_{i=1}^{n} w(op_i)}{\max(\text{Nodes}(T_x, T_y))} \quad (3)$$

where *ops* denotes the sequence of n edit operations (insertions, deletions, substitutions) required to transform the AST T_x of the original code snippet into the AST T_y of the (partial) generated slice, $w(op_i)$ represents the cost of the i-th operation, and the distance is normalized by the maximum number of nodes in the two trees to account for variations in code size and complexity.

TSED monotonicity The key insight underlying syntactic constraint is that, since every valid program slice is a *subsequence* of the input code, the autoregressive generation process should produce outputs that progressively resemble the original code's AST. Consequently, the TSED score between the input and the generated slice should *increase monotonically* at statement boundaries as more correct statements are appended. As illustrated in Figure 3, when no structural error occurs, the TSED score exhibits a monotonic increasing pattern at each statement boundary. However, when syntactic errors occur, such as incorrectly ordered statements, they disrupt the structural coherence and cause the TSED score to *decrease*. To detect and reject malformed candidates, we terminate a beam path early and discard its output whenever the TSED score drops at a statement boundary.

4 Experimental Setting

4.1 Datasets and Metrics

We use the Python and Java subsets of the CodeNet-Slice dataset (Yadavally et al., 2024) using the training split, and evaluate SLICEFORMER on the test split. Ground-truth slices are obtained using language-specific slicing tools: JavaSlicer (group, 2022) for Java and a Python slicer adapted from the JavaSlicer implementation (Galindo et al., 2022). Detailed dataset statistics are reported in Table 3.

Following previous studies (Yadavally et al., 2024; Shahandashti et al., 2024), we evaluate the performance of SLICEFORMER using four metrics: 1) **Dependence Accuracy (Acc-D)** is defined as the average ratio of correctly predicted dependencies to the total number of ground-truth dependencies; 2) **Exact Match** calculates the percentage of instances where the generated program slice exactly matches the ground-truth slice; 3) **CodeBLEU** is a composite text similarity metric specifically designed for coding tasks; 4) **TSED** is a code similarity metric that compares the ASTs of the generated and reference code by measuring the minimum edit operations required to transform one tree into another.

4.2 Baselines

LLM-based slicers (Shahandashti et al., 2024) leverage proprietary LLMs to perform program slicing using prompt engineering techniques, Zero-shot, Retrieval-Augmented Generation (RAG) (Chen et al., 2024), and Chain-of-Thought (CoT) (Li et al., 2025). We select two state-of-the-art foundation language models, GPT-4o-mini (Achiam et al., 2023) and GPT-5 (OpenAI, 2025), as the base LLMs.

NS-slicer (Yadavally et al., 2024) formulates the

program slicing task as a binary classification problem. It applies CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al.) to learn statement embeddings and then computes the distance to make predictions.

Fine-tuned models. We compare against models obtained by directly applying SFT to LMs without dataflow-aware pretraining and constrained decoding). Specifically, we fine-tune CodeLlama-7B (Roziere et al., 2023), Qwen3-8B (Yang et al., 2025), and CodeT5+ (Wang et al., 2023) on the training set. For CodeLlama and Qwen3, we employ QLoRA (Dettmers et al., 2023) to reduce hardware requirements.

4.3 Implementation Details

Pretraining We implement SLICEFORMER on top of CodeT5+ (Wang et al., 2023) (0.7B), a lightweight model that has been shown to be effective for a wide range of coding tasks (Wang et al., 2023; Li et al., 2024; Jiao et al., 2023; Yin et al., 2024). The model is pretrained on the Python and Java subsets of the CodeSearchNet dataset (Husain et al., 2019), comprising approximately 1.0M functions. Following GraphCodeBERT (Guo et al.), each function is first parsed into an AST using Tree-Sitter (Tree-sitter, 2019) to identify variables, after which a dataflow graph is constructed by following GraphCodeBERT's implementation (Guo et al.). For span corruption, we mask 25% of tokens, following the empirical setting in (Gong et al., 2024). For statement permutation, we randomly permute up to three statements per code sample. Pretraining is performed with a context length of 512 tokens and a batch size of 32 for 100K steps. All experiments are conducted on four NVIDIA RTX 3090 GPUs (24GB each).

Fine-tuning We set the input and output lengths to 512 tokens. The model is trained using the AdamW optimizer with a batch size of 32, a learning rate of 5×10^{-5}, and 1,000 warmup steps over 10 epochs. All other settings follow the default CodeT5+ configuration. We perform supervised fine-tuning on the training split of the Python and Java subsets of the CodeNet-Slice dataset (Yadavally et al., 2024).

Constrained Decoding We use a beam size of 3. For TSED, we adopt the implementation from the original paper, which supports both Java and Python. The lexical constraints are implemented in a HuggingFace–compatible manner, following the LogitsProcessor and Constraint interfaces (Huggingface, 2024).

5 Results

5.1 Effectiveness of SLICEFORMER

SLICEFORMER consistently outperforms baselines across two languages and four evaluation metrics. Table 1 reports the effectiveness of SLICEFORMER compared with existing program slicing methods. On Java programs, SLICEFORMER attains an ExactMatch score of 92.20%, significantly outperforming the strongest baseline, NS-slicer (GraphBERT), which achieves 85.77%. On Python programs, the performance gains are even more pronounced: SLICEFORMER reaches 83.15% in ExactMatch, compared to only 61.25% for NS-slicer. Overall, SLICEFORMER surpasses the best-performing baseline by 6.4% and 21.9% in ExactMatch on Java and Python, respectively, demonstrating its superior ability to generate fully correct slices across different programming languages. A qualitative analysis explaining the reasons behind these improvements is provided in Appendix C.

SLICEFORMER significantly outperforms fine-tuned vanilla models. SLICEFORMER consistently exceeds CodeT5+ performance in both languages: in Java, ExactMatch rose from 87.24% to 92.20% (5.0% gain); in Python, improvement was even sharper, rising from 77.24% to 83.15% (5.9% gain). These gains demonstrate that dataflow-aware pretraining combined with constrained decoding (guided by lexical and syntactic knowledge) is critical for achieving high accuracy and faithfulness in program slicing.

SLICEFORMER excels at generating fully correct slices Notably, SLICEFORMER achieves the most significant improvement on the ExactMatch metric, suggesting that it more frequently generates fully correct slices compared to all baselines. In contrast, LLM-based methods yield the weakest performance overall. Although techniques like CoT and RAG lead to improvements compared to the zero-shot setting, the best LLM-based approach (GPT-5 with CoT) only achieves 14.00% and 13.00% ExactMatch on Java and Python, respectively, far below the learning-based methods.

5.2 Ablation Analysis

All components contribute to SLICEFORMER's effectiveness. Figure 4 presents the results of ablation analysis for SLICEFORMER, examining the impact of each component. Overall, both dataflow-aware pretraining strategies (statement permutation and span corruption) and constrained decoding

Methods	Java				Python			
	Acc-D	ExactMatch	CodeBLEU	TSED	Acc-D	ExactMatch	CodeBLEU	TSED
LLM-based slicer (GPT-4/Zero-shot)	14.76	0.00	20.10	35.53	12.34	0.00	18.45	32.18
LLM-based slicer (GPT-4/RAG)	51.70	0.00	65.18	59.93	48.22	0.00	61.34	56.71
LLM-based slicer (GPT-4/COT)	56.84	0.00	68.41	59.68	53.17	0.00	65.23	57.42
LLM-based slicer (GPT-5/Zero-shot)	18.42	0.00	24.67	39.21	15.78	0.00	22.13	36.54
LLM-based slicer (GPT-5/RAG)	55.13	7.00	68.92	62.45	51.68	11.00	64.77	59.33
LLM-based slicer (GPT-5/COT)	60.27	14.00	71.35	63.81	56.94	13.00	68.56	61.27
NS-slicer (CodeBERT)	95.65	81.72	88.41	91.00	82.47	56.32	74.68	78.91
NS-slicer (GraphBERT)	96.51	85.77	89.26	90.35	84.92	61.25	76.84	80.12
CodeLlama (SFT)	82.83	75.27	79.10	81.82	75.61	68.45	72.33	74.26
Qwen3 (SFT)	87.22	80.55	80.54	82.35	79.34	72.18	74.91	76.58
CodeT5+ (SFT)	95.33	87.24	89.26	93.42	87.53	77.24	79.98	81.75
SLICEFORMER	**98.78**	**92.20**	**93.23**	**97.68**	**90.85**	**83.15**	**85.35**	**89.74**

Table 1: Effectiveness comparison among different static learning-based program slicing methods.

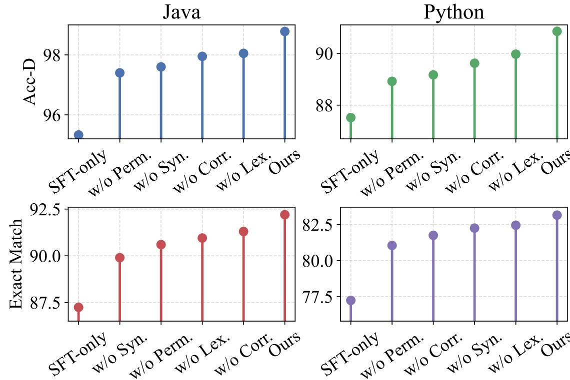

Figure 4: The ablation analysis results highlight the contribution of each component. A larger performance drop indicates greater importance. We rank the component importance from left to right. Abbreviations: Perm. = statement permutation; Corr. = span corruption; Syn. = syntactic constraint; Lex. = lexical constraint.

522 mechanisms (lexical and syntactic constraints) con-
523 tribute to SLICEFORMER's performance. As shown,
524 the full version of SLICEFORMER consistently out-
525 performs any variant lacking a single component
526 across all evaluation settings.

527 Among these components, span corruption and
528 lexical constraint are the most impactful. The
529 model learns more about data dependencies from
530 dataflow-aware span corruption, which is critical
531 for identifying complete program slices. Mean-
532 while, lexical constraints ensure that the generated
533 tokens strictly originate from the source code, pre-
534 venting hallucinations and improving statement-
535 level accuracy. The relatively smaller impact of
536 syntactic constraints can be attributed to the fact
537 that structural errors occur less frequently than lex-
538 ical errors, as the pretrained model has already
539 learned fundamental code syntax from corpora.

540 ## 6 Efficiency of SLICEFORMER

541 To evaluate efficiency, we compare inference la-
542 tency against all baselines. As shown in Table 2,
543 latency largely depends on the underlying base
544 model. When running on the same machine,
545 larger models with more parameters consistently
546 incur higher latency. NS-Slicer, which relies on
547 smaller models, achieves the lowest latency but
548 suffers from lower slicing effectiveness. In con-
549 trast, SLICEFORMER attains an average latency of
550 0.296 s and remains faster than other effective base-
551 lines. Overall, SLICEFORMER strikes a favorable
552 trade-off between inference efficiency and slicing
553 accuracy, demonstrating its practical viability.

Methods	Size	Runtime (per task)
NS-slicer (CodeBERT)	125M	0.105s
NS-slicer (GraphCodeBERT)	125M	0.135s
CodeT5+	770M	0.289s
SLICEFORMER	770M	0.296s
CodeLlama-7B (SFT)	7B	5.75s
Qwen3-8B (SFT)	8B	6.52s
GPT-4 (CoT)	Unknown	0.835s
GPT-5 (CoT)	Unknown	1.162s

Table 2: Inference latency comparison among different program slicing methods on Java dataset.

554 ## 7 Conclusion

555 This paper presents SLICEFORMER, a learning-
556 based approach for static program slicing that en-
557 hances accuracy through dataflow-aware pretrain-
558 ing and constrained decoding. We introduce two
559 dataflow-aware pretraining objectives, statement
560 permutation and span corruption, that enable the
561 model to learn variable dependencies essential for
562 precise slicing. In addition, we propose a con-
563 strained decoding strategy that enforces lexical and
564 syntactic constraints, ensuring that generated slices
565 are both accurate and faithful to the original pro-
566 gram. We evaluate SLICEFORMER on Java and
567 Python programs from the CodeNet dataset, where
568 it achieves state-of-the-art performance. Specif-
569 ically, SLICEFORMER outperforms the strongest
570 baseline, NS-slicer, by 6.4% and 21.9% in Exact-
571 Match on Java and Python, respectively.

8 Limitations

Language limitations. We evaluate SLICEFORMER on two widely used programming languages, Java and Python. Although our approach is readily adaptable to other languages, additional engineering and empirical validation are required. We encourage future work to extend our method to a broader range of programming languages and to other programming tasks whose outputs are subject to structural or semantic constraints.

Architecture limitations. Dataflow-aware pre-training is designed for encoder–decoder architectures, where masked-span reconstruction is naturally supported. In contrast, decoder-only models rely on next-token generation, making direct adoption of our pre-training objectives non-trivial. However, the proposed lexical–syntactic constrained decoding is architecture-agnostic and can be applied to both encoder–decoder and decoder-only models. Further discussion is provided in Appendix D.

9 Ethical Considerations

The implementation of this work is conducted with transparency, providing full disclosure of all technical details, limitations, and potential issues to the relevant stakeholders. The work avoids any false or misleading claims and ensures no data is fabricated or falsified.

In the interest of public benefit, the authors support reasonable and ethical uses of their intellectual contributions. Both the source code and data are released as free and open-source software and are made available in the public domain.

References

Mithun Acharya and Brian Robinson. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In Proceedings of the 33rd international conference on software engineering, pages 746–755.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. arXiv preprint arXiv:2303.08774.

Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. 2024. Code search is all you need? improving code suggestions with code search. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pages 1–13.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: efficient finetuning of quantized llms. In Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In Findings of the Association for Computational Linguistics: EMNLP 2020, pages 1536–1547.

Carlos Galindo, Sergio Perez, and Josep Silva. 2022. A program slicer for java (tool paper). In Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings, page 146–151, Berlin, Heidelberg. Springer-Verlag.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. Ast-t5: structure-aware pretraining for code generation and understanding. In Proceedings of the 41st International Conference on Machine Learning, pages 15839–15853.

The MiST group. 2022. slicer. https://github.com/mistupv/JavaSlicer.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. In International Conference on Learning Representations.

Mark Harman and Robert Hierons. 2001. An overview of program slicing. software focus, 2(3):85–92.

Oliver Hechtl. 2021. slicer. https://github.com/dwat3r/slicer.

Huggingface. 2024. Utilities for generation.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436.

Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1529–1541. IEEE.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 7871–7880, Online. Association for Computational Linguistics.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. ACM Trans. Softw. Eng. Methodol., 34(2).

Sha Li, Heng Ji, and Jiawei Han. 2021. Document-level event argument extraction by conditional generation. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 894–908, Online. Association for Computational Linguistics.

Xuan Li, Shuai Yuan, Xiaodong Gu, Yuting Chen, and Beijun Shen. 2024. Few-shot code translation via task-adapted prompt learning. Journal of Systems and Software, 212:112002.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. arXiv preprint arXiv:2404.00971.

Deep Mind. 2024. Gemma 2: Improving open language models at a practical size. Preprint, arXiv:2408.00118.

Shashi Narayan, Joshua Maynez, Reinald Kim Amplayo, Kuzman Ganchev, Annie Louis, Fantine Huot, Anders Sandholm, Dipanjan Das, and Mirella Lapata. 2023. Conditional generation with a question-answering blueprint. Transactions of the Association for Computational Linguistics, 11:974–996.

OpenAI. 2025. Introducing GPT-5. Accessed: 2025-01-02.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of machine learning research, 21(140):1–67.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.

Kimya Khakzad Shahandashti, Mohammad Mahdi Mohajer, Alvine Boaye Belle, Song Wang, and Hadi Hemmati. 2024. Program slicing in the era of large language models. arXiv preprint arXiv:2409.12369.

Xiaonan Song, Aimin Yu, Haibo Yu, Shirun Liu, Xin Bai, Lijun Cai, and Dan Meng. 2020. Program slice based vulnerable code clone detection. In 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pages 293–300.

Yewei Song, Saad Ezzini, Xunzhu Tang, Cedric Lothritz, Jacques Klein, Tegawendé Bissyandé, Andrey Boytsov, Ulrick Ble, and Anne Goujon. 2024. Enhancing text-to-sql translation for financial system design. In Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, pages 252–262.

Tree-sitter. 2019. py-tree-sitter: Python bindings to the tree-sitter parsing library. https://tree-sitter.github.io/py-tree-sitter/.

Zhaopeng Tu, Zhengdong Lu, Yang Liu, Xiaohua Liu, and Hang Li. 2016. Modeling coverage for neural machine translation. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 1069–1088.

Zhijie Wang, Zijie Zhou, Yuheng Huang Da Song, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2025. Towards understanding the characteristics of code generation errors made by large language models. In Proceedings of the IEEE/ACM 47th International Conference on software Engineering (ICSE'25).

Mark Weiser. 1984. Program slicing. 10(4):352–357.

Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. ACM SIGSOFT Software Engineering Notes, 30(2):1–36.

Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2024. A learning-based approach to static program slicing. Proc. ACM Program. Lang., 8(OOPSLA1).

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. arXiv preprint arXiv:2505.09388.

Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. arXiv preprint arXiv:2407.07472.

Biao Zhang, Yong Cheng, Siamak Shakeri, Xinyi Wang, Min Ma, and Orhan Firat. 2025a. Encoder-decoder or decoder-only? revisiting encoder-decoder large language model. arXiv preprint arXiv:2510.26622.

Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025b. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. Proceedings of the ACM on Software Engineering, 2(ISSTA):481–503.

Qihao Zhu, Qingyuan Liang, Zeyu Sun, Yingfei Xiong, Lu Zhang, and Shengyu Cheng. 2024. Grammart5: Grammar-integrated pretrained encoder-decoder neural model for code. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μ vuldeepecker: A deep learning-based system for multiclass vulnerability detection. IEEE Transactions on Dependable and Secure Computing, 18(5):2224–2236.

A Statistics of the Two Language Datasets

Table 3: Statistics of the two language datasets

Java			
	train	valid	test
Entries	30.8K	3.5K	8.7K
Avg. tokens	64	64	66
Avg. slocs	19	18	19
Python			
	train	valid	test
Entries	23.8K	5.1K	5.1K
Avg. tokens	88	86	91
Avg. slocs	25	25	26

Table 3 reports the basic statistics of the Java and Python datasets. Both datasets are split into training, validation, and test sets. Java samples are generally shorter, with fewer tokens and source lines of code, while Python samples are relatively longer across all splits. SLOC (Source Lines of Code) counts the number of executable source lines, excluding blank lines and comments.

B Workflow of Constrained Decoding

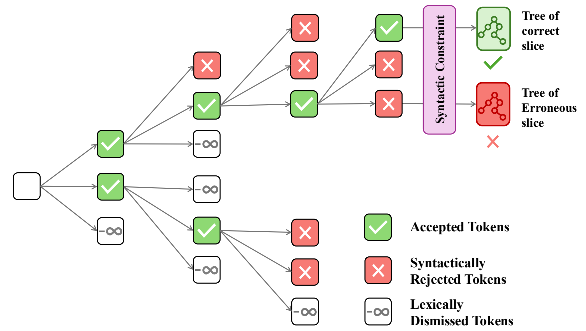

Figure 5: Workflow of constrained decoding

Figure 5 illustrates constrained decoding with a beam size of 3. Each column presents the top three token predictions ranked by probability. At each decoding step, lexical constraints filter out invalid tokens, while syntactic constraints based on TSED monotonicity prune malformed beams at statement boundaries.

C Qualitative Analysis

The main drawback of the previous SOTA, NS-slicer, lies in its design of task modeling. For example, in Example (4) shown in Figure 6, NS-slicer models program slicing as a binary classification task, with a fixed threshold to determine whether a

Model	Architecture	Pretraining	Super Fine-tuning	Constrained Decoding	ExactMatch
CodeT5	Encoder–Decoder	×	✓	×	82.80
CodeT5	Encoder–Decoder	✓	✓	✓	85.12
CodeLlama-7B	Decoder-Only	×	✓	×	75.27
CodeLlama-7B	Decoder-Only	×	✓	✓	78.40
Qwen3-8B	Decoder-Only	×	✓	×	80.55
Qwen3-8B	Decoder-Only	×	✓	✓	83.11

Table 4: Comparison of transformer architectures and their compatibility with SLICEFORMER components (Java).

statement belongs in the slice. As it embeds each statement independently, it fails to differentiate between identical statements at different positions. Consider the statement (e.g., `ch = true;`) appearing in two different branches of a method. NS-slicer treats both occurrences identically, even if only one is relevant. In contrast, SLICEFORMER operates at the method level and leverages the full context, allowing it to accurately determine which occurrence is the accurate slice.

```
// Example (4) - Erroneous slice from NS-slicer
// Expected slice:
...
19: if (Math.abs(as[l]) >= Math.abs(as[r])) {
20: lst.unset(r);
21: ch = true;
...
// Generated slice:
...
20: lst.unset(r);
21: ch = true;
22: }
23  if (Math.abs(as[r]) >= Math.abs(as[l])) {
24: lst.unset(l);
25: ch = true;
...
```

Figure 6: An erroneous example predicted by NS-slicer.

D Transformer Architecture Comparison

We build SLICEFORMER on top of the encoder–decoder model CodeT5+. Prior work shows encoder–decoder architectures often achieve superior performance when trained on fixed datasets (Zhang et al., 2025a). To investigate whether our proposed techniques generalize across different transformer architectures, Table ?? compares an additional encoder–decoder model (CodeT5) and two decoder-only models (CodeLlama and Qwen3) in terms of their compatibility with each component of SLICEFORMER and their performance on the Java slicing task.

Encoder–decoder models (CodeT5, CodeT5+). T5-style encoder–decoder models, including CodeT5+ (used in our experiments) and CodeT5, are naturally compatible with all components of SLICEFORMER. Their bidirectional encoder enables effective modeling of global code context, which is essential for both statement permutation and dataflow-aware span corruption during pre-training. In particular, CodeT5 still achieves competitive results due to the inherent advantages of the encoder-decoder architecture for sequence-to-sequence program slicing.

Decoder-only models (CodeLlama, Qwen3). Decoder-only models rely on causal attention and are pre-trained using next-token prediction under a strictly left-to-right generation paradigm. As a result, they are fundamentally incompatible with our dataflow-aware pre-training objectives. However, they can still benefit from **constrained decoding** at inference time, which enforces lexical and syntactic constraints on generated outputs.