SLICET5: Static Program Slicing using Language Models with Copy Mechanism and Constrained Decoding

Pengfei He University of Manitoba Winnipeg, Canada hep2@myumanitoba.ca Shaowei Wang University of Manitoba Winnipeg, Canada shaowei.wang@umanitoba.ca Tse-Hsun Chen Concordia University Montreal, Canada peterc@encs.concordia.ca

Abstract

Static program slicing is a fundamental technique in software engineering. Traditional static slicing tools rely on parsing complete source code, which limits their applicability to real-world scenarios where code snippets are incomplete or unparsable. While recent research developed learning-based approaches to predict slices, they face critical challenges: (1) Inaccurate dependency identification, where models fail to precisely capture data and control dependencies between code elements; and (2) Unconstrained generation, where models produce slices with extraneous or hallucinated tokens not present in the input, violating the structural integrity of slices. To address these challenges, we propose SLICET5, a novel slicing framework that reformulates static program slicing as a sequence-tosequence task using lightweight language models (e.g., CodeT5+). Our approach incorporates two key innovations. First, we introduce a copy mechanism that enables the model to more accurately capture inter-element dependencies and directly copy relevant tokens from the input, improving both dependency reasoning and generation constraint. Second, we design a constrained decoding process with (a) lexical constraint, restricting outputs to input tokens only, and (b) syntactic constraint, leveraging Tree Similarity of Edit Distance (TSED) monotonicity to detect structurally invalid outputs and discard them. We evaluate SLICET5 on CodeNet and LeetCode datasets and show it consistently outperforms state-of-the-art baselines, improving ExactMatch scores by up to 27%. Furthermore, SLICET5 demonstrates strong performance on incomplete code, highlighting its robustness and practical utility in real-world development environments.

ACM Reference Format:

Introduction

Static program slicing plays a crucial role in software engineering tasks such as vulnerability analysis [18, 44] and debugging [27, 36, 37]. Compared to dynamic slicing, which requires executing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, Washington, DC, USA

the program to capture runtime behavior [3, 11], static slicing does not depend on a runtime environment or specific test code without execution [4], enabling it more practical and broadly applicable [1, 37].

Conventional static slicing tools, such as JavaSlicer [7, 9] and CPP-Slicer [13], typically parse the Abstract Syntax Tree (AST), and insert data dependency edges to construct a System Dependence Graph (SDG). Once the SDG is built, slicing can be performed as a graph reachability problem by selecting the node corresponding to the slicing criterion and traversing the graph to identify all parts of the program that may affect or be affected by that criterion [7]. However, a prerequisite of these traditional approaches is fully compilable source code, often at a minimum of method-level granularity [38]. This limits their applicability to real-world scenarios, where code snippets, such as those found on online forums like Stack Overflow, are frequently incomplete and unparsable [39], despite being rich sources of developer knowledge.

Recently, there has been growing interest in leveraging language models (LMs) to automatically predict static program slices. Yadavally et al. [38] fine-tuned CodeBERT [6] and GraphCodeBERT [10] to predict relevant slices by computing the similarity between individual statements and the slicing criterion. However, this method operates at the statement level with limited contextual awareness and cannot feasibly incorporate all statements within a method, leading to a lack of inter-statement dependency modeling. Shahandashti et al. [26] utilized advanced foundation models (FMs) such as GPT-4 [2] and Gemma [21], combined with prompting techniques such as retrieval-augmented generation (RAG) [5] and chain-of-thought reasoning (COT) [15] to predict program slices. However, the effectiveness of these models on static program slicing remains limited. They often suffer from hallucinations due to insufficient logical reasoning over complex program [26]. Additionally, foundation models are expensive to deploy in practice, often incurring costs through API calls and are heavy on hardware requirements.

To address these limitations, we formulate static program slicing as a sequence-to-sequence (seq2seq) problem and propose a novel slicing approach built on lightweight LMs (i.e., CodeT5 [33] and CodeT5+ [32]). However, directly fine-tuning off-the-shelf LMs as slicers can be suboptimal. In our investigation (Sec. 2.2), we observe two major challenges:

Inaccurate dependency identification. An effective slicer requires precise identification of relationships between code elements (e.g., data and control dependencies) for a given slicing criterion. Language models currently struggle with this crucial aspect, which often leads to missing or including extra elements.

Unconstrained generation. The generated slice must be an exact subpart of the original code snippet, with all tokens

precisely extracted, and their order and structure preserved. However, current models often exhibit unconstrained generation, i.e., producing code elements not present in the original code snippet.

To address the challenges, we propose a novel approach, namely SLICET5, which integrates copy mechanism and constrained de**coding** to generate more accurate slices. Specifically, we improve the existing language models in two folders. First, we integrate a copy mechanism into the model. The copy mechanism strengthens the model's ability to learn and capture relationships between code elements, directly addressing Challenge **1**. Additionally, it provides a direct pathway for transferring exact tokens from the input to the output, which helps mitigate Challenge 2. At each decoding step, the model calculates a copy probability, determining when to directly copy tokens from the input sequence. Second, we develop a novel constrained decoding process by integrating two constraints to further tackle the Challenge **2**. **Lexical constraint**: We restrict the decoding process to only accept tokens present in the original code, preventing the generation of invalid tokens. Syntactic constraint: We measure the syntactic alignment between the generated slice and the original code using Tree Similarity of Edit Distance (TSED) [28]. If the TSED score of the generated code does not monotonically increase as expected, it is a sign that the code is structurally biased (e.g., through repetition or misalignment). Such candidates are discarded to ensure only syntactically coherent outputs are retained.

We evaluated SLICET5 on two datasets CodeNet and Leetcode. SLICET5 consistently outperforms SOTA baselines (e.g., FM-based slicer [26] and NS-slicer [38]) across different datasets and evaluation metrics. For instance, SLICET5 outperforms the best NS-slicer by at least 6.4% and 27% in terms of ExactMatch, on the two datasets, respectively. We also evaluated SLICET5 on incomplete code snippets, and SLICET5 still consistently outperforms all SOTA baselines, demonstrating the robustness of SLICET5 for unparsable code.

We make the following contributions:

- We propose the first end-to-end solution for static program slicing by improving the existing language model's architecture and developing novel constrained decoding.
- Through extensive evaluation, we demonstrated that our approach achieves significant performance improvements over the SOTA baselines.
- We make our code public ¹.

2 Problem Formulation and Challenges

In this section, we first present the formal definition of static program slicing. We then highlight the challenges that arise when directly fine-tuning language models for slicing, which motivates the design of our proposed method.

2.1 Problem Formulation

We formulate **static program slicing** as a sequence-to-sequence code transformation task. Formally, the input code snippet to be sliced is represented as:

$$\mathbf{x} = \{s_1, s_2, \dots, s_i, \dots, s_N; v; [n]\}$$
 (1)

```
// Example (1) - Inaccurate dependency identification
// Expected slice:
7: int temp
8: if(C <= A) {
12: temp = B;
...
// Generated slice:
7: int temp
8: if(C <= A) {
9: temp = A;
10: A = C;
12: temp = B;
...</pre>
```

Figure 1: Motivating examples of three wrong program slices produced by directly fine-tuned CodeT5+.

where s_i denotes a statement of the code snippet $\{s_1, s_2, \ldots, s_N\}$, v is the **variable of interest**, [n] is the line number of v. Each statement s_i is further represented as a sequence of code tokens $s_i = \{t_1^{(i)}, t_2^{(i)}, \ldots, t_{M_i}^{(i)}\}$.

Given x, the slicer P(y|x) should predict a slice y, where $y \subseteq x$ comprises the minimal set of statements, known as the backward slice, that **semantically influence** the value of the slicing criterion v.

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

Formally, y is defined as:

$$\mathbf{y} = \{s_{i_1}, s_{i_2}, \dots, s_{i_k}\} \subseteq \{s_1, s_2, \dots, s_{\lfloor n \rfloor}\}, \quad i_1 < i_2 < \dots < i_k \quad (2)$$

The output slice y is expected to satisfy the following properties:

- Accuracy: the generated slice, y, should include all and only the relevant statements that are expected to appear in the correct program slice with respect to the slicing criterion.
- Code element preservation: the slice y must be a subpart of the original code snippet x = {s₁, s₂,...,s_N}, meaning that every statement s_i ∈ y must be exactly extracted from x without modification, meanwhile preserving the same ordering. At the lexical level, each token t_j⁽ⁱ⁾ in the selected statements must appear verbatim in the original code and preserve the same ordering as in the code snippet.

2.2 Limitations of Vanilla Fine-Tuned Language Models for Program Slicing

A seq2seq model is a deep learning architecture designed to convert an input sequence into an output sequence [29]. Transformer-based seq2seq models are particularly preferred due to their strong performance and versatility, and have been adapted for programming tasks, most notably in models such as CodeT5 [33] and CodeT5+[32]. These models have achieved state-of-the-art results in code understanding and generation tasks, and are easily fine-tuned for a wide range of downstream software engineering applications, including type inference [35] and code compression [12]. Intuitively, seq2seq models can also be employed to learn the dependencies between code elements, making them a natural fit for the task of program slicing. More formally, the goal is to model the conditional probability P(y|x), where y denotes the sequence of program statements that constitute the desired slice, and x is the input program along with the slicing criterion. However, generating accurate and elementpreserving slices remains challenging when relying solely on direct fine-tuning of off-the-shelf models. In our empirical study, we identify several key limitations of directly fine-tuned models (e.g., CodeT5 and CodeT5+) when applied to the static program slicing task:

- Inaccurate dependency identification. It is challenging for language models (LMs) to accurately capture the dependencies between code elements. As a result, they often miss relevant statements or include irrelevant ones when performing slicing. For instance, in the Example (1) shown in Figure 1, the generated slice mistakenly includes additional statements, line 9 (temp = A;) and line 10 (A = C;), which are not required. Rather than identifying true data dependencies, the model tends to rely on surface-level patterns or positional proximity, leading to incorrect inclusion of unrelated statements in the slice.
- ② Unconstrained generation. The model exhibits unconstrained generation, i.e., producing code elements that are not presented in the original code. For instance, language models can struggle with rare tokens, often replacing them with semantically or syntactically incorrect alternatives, such as generating an frequent alternative keta instead of the correct identifier codepoint in Example (2) in Figure 1, which leads to a syntactic error. In fact, LM-generated code is often prone to lexical errors [34]. In addition, models tend to generate unrelated logic or duplicating statements, such as Example

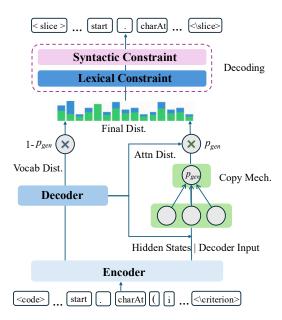


Figure 2: An overview of SLICET5.

(3) in Figure 1, where the generated statement $y * 10 * y * y * y * z * ten \ldots$ is not the exact statements presented in the original code, and the sub statement z * ten * 10 * y repeats. These errors collectively underscore the limitations of the vanilla fine-tuned model in producing accurate and faithful slices.

3 Methodology

To address the limitations observed in Section 2.2, we propose a new design of the static program slicing framework, SLICET5, which integrates copy mechanism and constrained decoding to generate more accurate and element-preserved slices. Figure 2 presents the overall framework of SLICET5. More specifically, we improve the original Transformer-based models in two ways:

Architecture. We first integrate a copy mechanism [25] into the model. The copy mechanism allows the model to directly copy tokens from the input sequence to the output, rather than always generating output tokens from the fixed vocabulary. The copy mechanism addresses the limitations in two ways. First, the copy mechanism enhances the model to capture the relevant statements based on the given slicing criterion (to address ①). Second, the code mechanism provides a direct pathway for transferring exact tokens from the input to the output (to address ②). At each decoding step, the model computes a copy probability that governs whether the relevant tokens should be copied directly from the input sequence. This mechanism enables the model to better handle rare identifiers by circumventing vocabulary limitations and enhancing the fidelity of reproducing rare tokens, while increasing the likelihood of sampling tokens from the original code for output.

Decoding. To mitigate the limitation **2**, we enforce decoding process of beam search by incorporating lexical and syntactic constraints. **Lexical constraint**: We restrict the decoding process to only accept tokens present in the original code, preventing the generation of invalid tokens that does not present in the original code.

Syntactic constraint: We evaluate the syntactic alignment between the generated slice and the original code using Tree Similarity of Edit Distance (TSED) [28]. Since a valid program slice is a subsequence of the input code snippet, the TSED score of the partially generated sequence is expected to increase monotonically during decoding as more code elements are generated correctly. Conversely, if the generated code deviates structurally, due to syntactic inconsistencies such as repetition or misalignment, the TSED score may drop, indicating a violation of structural coherence. In such cases, the generation is considered invalid, and the output is discarded to ensure that only syntactically well-formed slices are retained.

We elaborate on each mechanism in detail below.

3.1 Copy Mechanism

The copy mechanism, originally proposed in pointer networks [30], is based on attention and was initially applied to LSTM (Long Short-Term Memory) architectures to address the challenge of generating rare and out-of-vocabulary (OOV) tokens. Inspired by this work, we extend the idea from the LSTM to the Transformer architecture, resulting in a copy-enhanced Transformer specifically tailored for the program slicing task. The overall architecture is illustrated in Fig. 2. Unlike LSTMs, which rely on both the encoder and decoder for recurrence, the Transformer operates purely through attention mechanisms.

In our design, we use cross-attention weights to estimate the copy probability of source tokens. Specifically, we compute:

$$\alpha = \operatorname{softmax} \left(\frac{Q_{dec} K_{enc}^T}{\sqrt{d}} \right) \tag{3}$$

where $Q_{\rm dec}$ and $K_{\rm enc}$ are the query and key matrices of the decoder and encoder, and d is the hidden dimensionality. The attention α signals the decoder to focus on important source tokens. Next, the decoder hidden state h^* of the last layer is computed as the weighted sum of input representations as:

$$h^* = \alpha \cdot V_{enc} \tag{4}$$

where V_{enc} represents the value matrix from the encoder. The context vector h^* serves as a dynamic summary of the relevant input content at each decoding step. To decide whether to copy or generate a token, we concatenate the context vector h^* with the decoder input x_{dec} , and pass it through a copy layer to calculate the generation probability $p_{qen} \in [0,1]$:

$$p_{qen} = Sigmoid(W_{qen} \cdot [h^*; x_{dec}] + b_{qen})$$
 (5)

where W_{gen} and b_{gen} are learnable parameters of the linear copy module. Here, p_{gen} corresponds to the probability of generating tokens from the vocabulary, while $p_{gen} = (1 - p_{gen})$ corresponds to the probability of copying a token from the source code snippet using the attention distribution α .

Finally, the model interpolates between the generation distribution P_{vocab} and the copy distribution P_{copy} , producing the final output distribution as:

$$P(y) = p_{aen}P_{vocab}(y) + (1 - p_{aen})\alpha$$
 (6)

This copy mechanism not only directs the decoder's attention to important source tokens, but also enables the model to directly copy tokens from the code snippet, improving its handling of rare tokens in program slice.

During training, we use the Cross-Entropy Loss to maximize the likelihood of the target sequence. The loss function is defined as:

$$\mathcal{L} = -\sum_{i} p_i^* \log(p_i) \tag{7}$$

where p^* is the distribution of the ground-truth slice, and p is the predicted slice distribution produced by SLICET5.

Additionally, to enhance control over the generation process in the task-specific fine-tuned language model, we prepend special markers during fine-tuning, following prior works [16, 22, 43], to improve performance on downstream tasks. These special tokens serve as explicit signals to guide the model toward generating structured outputs for program slicing. Specifically, we introduce the following markers, line_number>, </line_number>, </ode>, </criterion>, </criterion>, <slice>, and </slice>, to help the LM better recognize and process slicing-related components.

Algorithm 1: Constrained Beam Search with Lexical Constraint and Syntactic Constraint

```
Input: Input sequence x; Language model \mathcal{M}; Vocabulary \mathcal{V}; Beam size K;
                Max decoding length L
    Output: Program slice \mathcal{Y}
         Determine lexically allowed tokens
 2 \mathcal{A} \leftarrow \text{GetAllowedTokens}(x)
 3 // Initialize beam with empty sequence
 4 \mathcal{B} \leftarrow \{(\boldsymbol{y} = [], s = 0)\}
 5 for t = 1 to L do
          \mathcal{B}_{\text{next}} \leftarrow \emptyset
          foreach beam (y, s) \in \mathcal{B} do
                  // Apply Lexical Constraint
                 mask \leftarrow ApplyMask(\mathcal{A}, \mathcal{V})
10
                 p \leftarrow \text{NextTokenScores}(\mathcal{M}, y, mask)
11
                  \{(z_k, p_k)\}_{k=1}^K \leftarrow \text{TopK}(\boldsymbol{p}, K)
12
                   // Expand each beam
                 for k = 1 to K do
13
                           Apply Syntactic Constraint
                        y' \leftarrow y \parallel z_k
15
                        t_{\text{prev}} \leftarrow \texttt{TSED}(\boldsymbol{x}, \boldsymbol{y})
 16
                        t_{\text{cur}} \leftarrow \texttt{TSED}(\pmb{x}, \pmb{y}')
17
                        // Skip if syntactic error occurs (TSED \downarrow)
18
                              or end of sequence
                        if t_{\mathrm{cur}} < t_{\mathrm{prev}} or ISEOS (z_k) then
 19
20
                              continue
                        end if
21
22
                        s' \leftarrow s + \log(p_k)
23
                        \mathcal{B}_{\text{next}} \leftarrow \mathcal{B}_{\text{next}} \cup \{(\boldsymbol{y}', s')\}
24
                 end for
25
           end foreach
           // Keep top-K beams
           \mathcal{B} \leftarrow \text{TopK}(\mathcal{B}_{\text{next}}, K)
28 end for
          Return the top-1 output sequence with the
          highest score
30 return \underline{\mathcal{Y}} \leftarrow \arg \max_{(\boldsymbol{y},s) \in \mathcal{B}} s
```

3.2 Constrained Decoding

We apply two novel constraints during decoding stage by modifying beam search and present our algorithm in Algorithm 1. For each beam search step, we first apply lexical constraint to only allow the token appearing in the original code snippet x to be sampled (Line 8)

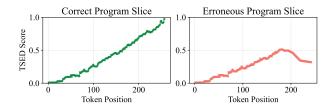


Figure 3: TSED scores for syntactically correct versus erroneous program slices based on the same input code snippet. The scores are derived from Example (3) in Figure 1. When a syntactic error is introduced into the generated slice, the TSED score deviates from its expected monotonic increasing pattern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

- 11). Next, for each selected candidate, we apply syntactic constraint and filter out the ones that violate syntactic constraint (Line 14 - 21). Below, we introduce each constraint.

3.2.1 Lexical Constraint. When a model is fine-tuned to generate code, such as in the task of program slicing, it may produce hallucinated or anti-factual tokens that do not appear in the input snippet, as illustrated in Figure 1. This issue stems from the fact that language models operate over an unconstrained output space. For instance, at each decoding step, CodeT5(+) samples from a fixed vocabulary of 32,100 tokens based on their probability [32], regardless of whether these tokens are relevant or even present in the input. However, program slicing is inherently an extractive code transformation task, where the output must consist only of tokens drawn from the original code snippet. This mismatch between the model's generative nature and the extractive demands of the task often leads to incorrect slices.

To mitigate this, we introduce a lexical-constrained decoding mechanism that restricts the model to only generate tokens from the input sequence. The core idea is simple: tokens that do not appear in the input should be assigned zero probability during decoding. Specifically, we assign a logit score of $-\infty$ to all disallowed tokens, thereby ensuring their softmax probabilities are zero (Lines 9 - 10 in Algorithm 1). This mechanism enables SLICET5 to generate only valid tokesn, reject unknown identifiers, and catch subtle errors such as spelling mismatches, resulting in more accurate code generation.

3.2.2 Syntactic Constraint. Lexical constraint alone is insufficient for generating code element preserved code, as they are order-insensitive and do not account for the structure of the output code. For example, as shown in Example (3) in Figure 1, we observe instances of over-generation [31] in the output code slices. Despite the presence of a clear error, all the tokens in the generation are drawn from the input, and thus satisfy the lexical constraint. In such cases, the individual lexical constraint fails to prevent the model generating wrong code because it does not enforce correct token ordering or semantic intent.

To address this issue, we incorporate an additional AST-based syntactic constraint, namely the Tree Similarity of Edit Distance (TSED) [28] Monotonicity Increasing Constraint. TSED is a structural similarity metric that measures the syntactic distance between two code snippets based on their abstract syntax trees (ASTs). It is

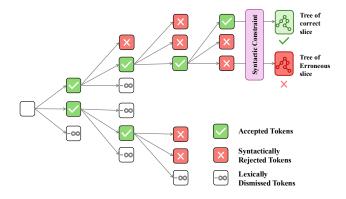


Figure 4: Implementation of proposed constrained decoding with a beam size of 3.

defined as:

$$TSED(T_x, T_y) = 1 - \frac{\min_{\text{ops}} \sum_{i=1}^{n} w(op_i)}{\max(\text{Nodes}(T_x, T_y))}$$
(8)

where ops denotes the sequence of n edit operations that transform the tree T_x (i.e., the parsed tree of the original code snippet) into another tree T_y (i.e., the parsed tree of the (partial) slice), and $w(op_i)$ represents the cost associated with the i-th operation. The computed distance is normalized by the maximum number of nodes in the two trees to account for variations in code complexity and size.

To ensure syntactic coherence during generation, we enforce a monotonicity constraint on the TSED score, requiring that the syntactic similarity between the generated slice and the original code increases monotonically at each decoding step. Since every valid slice is a subsequence of the input code, auto-regressive decoding should incrementally produce outputs that increasingly resemble the original code's AST, resulting in a progressively higher TSED score. As illustrated in Figure 3, if no structural error occurs, the TSED between the original code and the generated slice should increase monotonically. However, when errors such as over-generation occur, they can disrupt structural similarity and cause a drop in the TSED score. In such cases, the generation process is terminated early, and the output is discarded, as shown in Lines 14–21 of Algorithm 1.

Figure 4 presents the workflow of our constrained decoding process with a beam size of 3. Each vertical column represents the top three token predictions for a hypothesis, arranged in descending order of probability from top to bottom. For each beam search step, the lexical constraint is applied so that only the tokens from the original code snippet and special tokens are allowed to sampled, while all others are automatically dismissed by assigning them a score of $-\infty$. Second, syntactic constraint is applied to measure the TSED monotonicity for each candidate in the beam. Accepted candidates continue decoding in the next step, while rejected ones are terminated early. In the end, we output the top candidate with their highest score. This order-sensitive syntactic constraint complements the lexical constraint by encouraging the model to produce slices that adhere not only to the token-level content but also to the syntactic structure of the input code, thereby significantly reducing the risk of generating ill-formed or structurally implausible slices.

Note that the proposed lexical and syntactic constrained decoding is entirely training-free, and can be directly, easily, and optionally enabled during inference.

4 Experimental Setting

4.1 Research Questions

- RQ1: How effective is SLICET5 compared to existing SOTA learning-based slicing approaches?
- RQ2: How effective is each component of SLICET5?
- RQ3: How effective is SLICET5 on incomplete code snippets?

In RQ1, we compare the effectiveness of SLICET5 with prior state-of-the-art (SOTA) learning-based approaches. RQ2 conducts ablation analysis to examine the contribution of each component (i.e., copy mechanism, lexical constraint, and syntactic constraint). In RQ3, we evaluate the effectiveness of SLICET5 on incomplete code snippets.

4.2 Baselines

We compare our approach with the following SOTA learning-based baselines.

- (1) **FM-based slicer [26]** leverages foundation models to conduct program slicing [26] by using various prompting engineering. Follow their approach, we use three prompting strategies, Zero-shot, Retrieval-augmented Generation (RAG) [5], and Chain-of-Thought (COT) [15]. We select two SOTA foundation language models, GPT-4o-mini [2] and Gemma-7B [21],as the base model.
- (2) NS-slicer [38] formulates the program slice task as a 0-1 classification. It applies a CodeBERT [6] and GraphCodeBert [10] to learning representation then compute the distance between individual statement and criterion. Given an code snippet and slicing criteria, it predicts whether each statement in the code snippet should be included in the slicing or not.
- (3) Fine-tune. To properly assess SLICET5's effectiveness, we compare against a directly fine-tuned model variant that uses the same architecture but without our proposed enhancements (i.e., copy mechanism and constrained decoding). This baseline helps isolate the contribution of our novel components.

4.3 Datasets and Metrics

We evaluated the performance of SLICET5 on two program slicing datasets: CodeNet [38] and LeetCode [26], following the experimental setups of prior studies [26, 38]. The CodeNet dataset consists of programming solutions from IBM, while the LeetCode dataset contains a diverse collection of solutions to various LeetCode problems. Compared to CodeNet, the LeetCode dataset is more complex, featuring longer code samples with more tokens and lines. For both datasets, the ground-truth static program slices were obtained using JavaSlicer [9]. In line with prior work [26], we focus exclusively on Java and backward slicing, which is particularly important for applications of static program slicing such as debugging [37]. Detailed statistics of the two datasets are provided in Table 1. We use the training and validation splits of CodeNet to train SLICET5, the test set of CodeNet to evaluate in-domain performance of ours, and

the LeetCode test-only set to assess out-of-domain generalization capability.

Table 1: Basic statistics of our two datasets.

	C	Leetcode		
	train	valid	test	test-only
Entries	30.8K	3.5K	8.7K	100
Avg. tokens	64	64	66	153
Avg. slocs	19	18	19	35

Following previous studies [26, 38], we evaluate the performance of SLICET5 on four metrics: 1) Dependence Accuracy (Accuracy-D), measures how accurately statement-level dependencies are predicted. Specifically, Accuracy-D is computed as the ratio of correctly predicted statements to the total number of actual dependent statements for a given slicing criterion in the program. 2) Exact Match calculates the percentage of test instances where the generated program slice exactly matches the ground-truth slice. Even small deviations, such as an incorrect variable name in a statement (e.g., as shown in Example (2) of Figure 1), are considered failures under this strict metric. 3) CodeBIEU, is a composite metric specifically designed for code generation tasks, CodeBLEU incorporates multiple signals: n-gram matching (BLEU), weighted n-gram match (BLEU-weighted), AST match, and data-flow match [24]. This metric captures both lexical-level and structural similarities between code snippets. 4) **TSED** (equation 7), is a novel code similarity metric that compares the abstract syntax trees (ASTs) of the generated and reference code. TSED is sensitive to the syntactic structure of code and is particularly useful for measuring similarity in structured code transformations such as program slicing. Note that NS-Slicer produces only line numbers. To enable a fair comparison using Code-BLEU and TSED, we map the predicted line numbers back to their corresponding program statements before evaluation.

4.4 Base LMs

We implemented SLICET5 on top of the SOTA seq2seq models CodeT5 [33] and CodeT5+ [32]. Those models have been demonstrated to be powerful in code compression and code translation tasks [14, 17, 32, 33, 40]. We select the CodeT5-base-0.8B and CodeT5+-0.7B version of those two models, since we wish to keep our approach lightweight.

4.5 Implementation Details

To ensure consistent outputs, we set the max source and target to 256. The CodeT5(+) is trained using the AdamW optimizer with a batch size of 16, a learning rate of 5×10^{-5} , and 1,000 warmup steps over 10 epochs. All other settings follow the default CodeT5(+) configuration. We set the beam size to 3,

5 Results

5.1 RQ1: Effectiveness of SLICET5

SLICET5 consistently outperforms baselines across two datasets and four evaluation metrics. Specifically, SLICET5 surpasses the best-performing baseline, NS-slicer, by 6.4% and 27% in Exact-Match on CodeNet and LeetCode, respectively. Table 2 compares

Methods	CodeNet (in-domain)				Leetcode (out-of-domain)			
	Acc-D	ExactMatch	CodeBLEU	TSED	Acc-D	ExactMatch	CodeBleu	TSED
GPT-4 (Zero-shot)	14.76	0.00	20.10	35.53	30.91	0.00	23.77	38.69
GPT-4 (RAG)	51.70	0.00	65.18	59.93	54.09	0.00	47.05	55.41
GPT-4 (COT)	56.84	0.00	68.41	59.68	60.84	7.00	46.31	54.16
Gemma (Zero-shot)	15.23	0.00	23.30	43.23	33.41	0.00	22.67	42.59
Gemma (RAG)	59.23	0.00	63.24	59.21	41.90	0.00	38.80	42.11
Gemma (COT)	64.23	0.00	74.48	62.11	41.49	0.00	40.34	48.32
NS-slicer (CodeBERT)	95.65	81.72	88.41	91.00	66.43	11.00	54.91	55.60
NS-slicer (GraphBERT)	96.51	85.77	89.26	90.35	67.07	4.00	55.45	56.45
Fine-tune (CodeT5)	92.19	82.80	82.74	82.65	61.34	4.00	51.38	47.40
SLICET5 (CodeT5)	96.97	85.12	89.35	93.95	66.00	13.00	60.26	58.90
Fine-tune (CodeT5+)	95.33	87.24	89.26	93.42	66.94	7.00	53.76	50.74
SLICET5 (CodeT5+)	98.32	91.30	92.31	97.06	70.85	14.00	60.40	59.97

Table 2: Effectiveness comparison among different static learning-based program slicing methods.

the effectiveness of SLICET5 against the evaluated learning-based program slicing methods across four evaluation metrics. SLICET5 consistently outperforms all baselines across all datasets and metrics. For example, SLICET5 with CodeT5+ achieves 98.32%, 91.30%, 92.31%, and 97.06% in terms of Acc-D, ExactMatch, CodeBLEU, and TSED on CodeNet, respectively. In comparison, the best-performing baseline, NS-slicer (GraphBERT), achieves 96.51%, 85.77%, 89.26%, and 90.35% on the same metrics. SLICET5 achieves improvements of 1.9%, 6.4%, 3.4%, and 7.4% on the four metrics, respectively. Even with a less powerful model CodeT5, SLICET5 still can achieve the SOTA performance. The main drawback of the previous SOTA, NS-slicer, lies in its design of task modeling. For example, in Example (4) shown in Figure. 5, NS-slicer models program slicing as a binary classification task, with a fixed threshold to determine whether a statement belongs in the slice. As it embeds each statement independently, it fails to differentiate between identical at different position. 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, SLICET5 operates at the method level and leverages the full context, allowing it to accurately determine which occurrence is the accurate slice.

SLICET5 excels at generating fully correct slice, while FMbased method suffers an ExactMatch of 0. Notably, SLICET5 achieves the most significant improvement on the ExactMatch, suggesting that it more frequently generates fully correct slices compared to the baselines, a particularly valuable feature in practical scenarios. In contrast, the ExactMatch scores for the FM-based methods are nearly all zero. In fact, FM-based methods yield the weakest performance overall. Although techniques like CoT and RAG lead to improvements compared to the zero-shot setting as the results indicate. Manual inspection of failure cases reveals that advanced foundation models often misinterpret control flow structures. For example, they may include only the if branch while omitting necessary else or else if branches, leading to logically incomplete slices. In contrast, SLICET5 effectively captures these dependencies, allowing it to maintain logical and syntactic correctness in the slices it generates.

Compared to the fine-tuned vanilla model, SLICET5 achieves significant performance gains, especially in out-of-domain datasets.

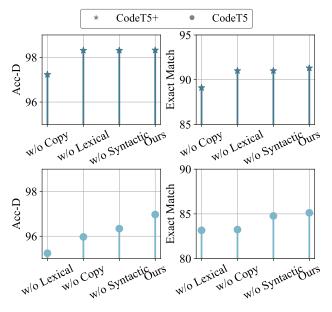
```
// Example (4) - Erroneous slice from NS-slicer
// Expected slice:
...
19: if (Math.abs(as[1]) >= Math.abs(as[r])) {
20: lst.unset(r);
21: ch = true;
...
// Generated slice:
...
20: lst.unset(r);
21: ch = true;
22: }
23 if (Math.abs(as[r]) >= Math.abs(as[l])) {
24: lst.unset(l);
25: ch = true;
...
```

Figure 5: An erroneous example predicted by NS-slicer.

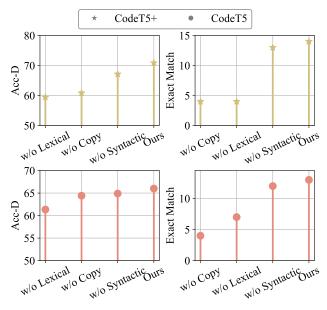
As shown in Table 2, SLICET5 consistently outperforms the fine-tuned orginal CodeT5(+) across on both datasets. For the in-domain dataset (CodeNet), SLICET5 (CodeT5+) improves the ExactMatch from 87.24% to 91.30%, showing a relative improvement of 4.7%. Notably, the performance improvement is more pronounced in the out-of-domain dataset (Leetcode), where SLICET5 (CodeT5+) doubles the ExactMatch from 7% to 14%. We observe a more notable improvement with SLICET5 (CodeT5), where the ExactMatch rate is improved from 4% to 13% on Leetcode. The results indicate that the enhanced copy mechanism, along with the constrained decoding by lexical and syntactic knowledge, plays a critical role in improving both the slicing accuracy and the preservation of relevant program elements.

5.2 RQ2: Ablation Analysis

All components make significant contribution to SLICET5. Figure 6 presents the results of the ablation analysis of SLICET5 across



(a) CodeNet Dataset



(b) Leetcode Dataset

Figure 6: The ablation analysis results highlight the contribution of each component. A larger performance drop indicates greater importance of the corresponding component. We rank the component importance from left to right.

both base LMs. Overall, all components contribute to SLICET5's performance. As shown, the full version of SLICET5 consistently outperforms any variant lacking a single component (i.e., copy mechanism, lexical constraint, or syntactic constraint). Among these components, the copy mechanism and lexical constraint are the most impactful components, with varying influence across metrics. Lexical constraint is key for Acc-D, leading in 3 of 4 LM-dataset settings.

For ExactMatch, both components show strong effects, with the copy mechanism also leading in 3 of 4 settings.

On out-of-domain dataset, the copy mechanism and the lexical constraint help generate a fully correct slicer remarkably. When testing on the out-of-domain dataset (Leetcode), we observe that without the copy mechanism and the lexical constraint, Exact-Match drops from 14% to 4% and 7%, respectively. This observation suggests that copy mechanism and lexical constraint make great contribution to ensure generate exactly correct slice. The syntactic constraint is the least impactful, likely because structural errors are less frequent than lexical ones, which can be mitigated by the copy mechanism and lexical constraint.

5.3 RQ3: Effectiveness of SLICET5 on Incomplete Code Snippets

Compared to program analysis (PA)-based slicing tools such as Javaslicer [7], learning-based approaches offer a distinct advantage, which is treating code snippets as token sequences rather than relying on strict syntactic structures. They can be applied more flexibly to incomplete or ill-formed code. To assess the effectiveness of our approach under such conditions, we simulate incompleteness by deliberately corrupting input code snippets before feeding them into the model. Specifically, we evaluate three types of unparsable code, following the setup in [34, 39]: 1. Missing Class Encapsulation: Removing class or method wrappers from the code produces snippets that are not valid compilation units. 2. Missing Semicolons: Eliminating semicolons from the ends of statements, which typically leads to parse errors. 3. Missing or Unmatched Braces: Removing required braces or introducing unmatched braces to break block structures and nesting rules.

SLICET5 consistently outperforms all baselines across all types of unparsable code snippets in terms of all evaluation metrics. Table 3 compares the performance of different learning-based method on the studied three types of unparsable code snippets. As shown, SLICET5 consistently outperforms all other baselines on both datasets across all evaluation metrics. For instance, on Leetcode dataset, SLICET5 still achieves the same ExactMatch of 14%, while the best performing method NS-slicer (GraphBERT)'s ExactMatch degrades from 11% to 3%.

Among the three types of unparsable code, Unmatched Braces cause the most significant performance drop compared to the other two. When evaluating all methods studied, including SLICET5, we observe the sharpest decline in performance with Unmatched Braces, as opposed to Missing Semicolons and Missing Class Encapsulation. For example, SLICET5's ExactScore drops to 79.1% on Unmatched Braces but only to 91.0% and 89.1% for the other types. Other learning-based methods exhibit similar trends. These results suggest that Unmatched Braces have the strongest negative impact on learning-based approaches, whereas they remain more robust to missing semicolons or class encapsulation. One possible explanation is that braces {} define block structures (e.g., loops, conditionals, method bodies) and its scope, which is fundamental to program semantics. Missing/Unmatched braces break nesting rules, leading to ambiguity in variable scope (e.g., is a variable inside or outside a loop?) and incorrect control flow interpretation (e.g., does an if statement cover the next line or not?). Language Models like CodeT5

Table 3: Results of different learning-based methods on the three types of incomplete code snippets.

Methods	CodeNet				Leetcode				
	Acc-D	ExactMatch	CodeBLEU	TSED	Acc-D	ExactMatch	CodeBLEU	TSED	
Missing Class Encapsulation									
GPT-4 (Zero-shot)	15.27	0.00	20.14	36.14	28.31	0.00	22.43	37.43	
GPT-4 (RAG)	53.24	0.00	66.23	60.05	52.40	0.00	46.44	53.89	
GPT-4 (COT)	55.45	0.00	67.99	60.46	58.93	4.00	44.60	52.19	
Gemma 3 (zero-shot)	13.26	0.00	21.20	38.89	32.23	0.00	21.24	41.30	
Gemma 3 (RAG)	58.27	0.00	61.65	57.80	40.15	0.00	37.24	42.55	
Gemma 3 (COT)	62.23	0.00	72.28	59.99	36.59	0.00	32.35	44.15	
NS-slicer (CodeBERT)	91.61	80.96	72.53	76.58	65.32	0.00	51.47	52.25	
NS-slicer (GraphBERT)	92.10	82.10	73.08	76.29	66.59	3.00	52.65	54.31	
SLICET5 (CodeT5)	96.37	89.30	73.96	78.63	66.44	13.00	52.75	54.51	
SLICET5 (CodeT5+)	93.30	91.00	74.34	80.55	71.73	14.00	53.90	53.33	
			Missing Semi	colons	1				
GPT-4 (Zero-shot)	14.73	0.00	19.67	35.83	27.8	0.00	19.89	36.55	
GPT-4 (RAG)	52.79	0.00	66.10	59.89	51.22	0.00	47.98	52.39	
GPT-4 (COT)	55.77	0.00	68.28	60.26	59.63	0.00	42.14	51.65	
Gemma 3 (zero-shot)	12.48	0.00	22.30	37.16	31.20	0.00	22.57	40.99	
Gemma 3 (RAG)	55.46	0.00	58.40	56.40	43.55	0.00	37.19	41.09	
Gemma 3 (COT)	61.52	0.00	71.65	66.90	37.68	0.00	36.80	42.85	
NS-slicer (CodeBERT)	95.65	81.72	80.26	91.00	66.43	11.00	54.91	55.60	
NS-slicer (GraphBERT)	96.51	85.77	81.41	90.35	67.07	4.00	55.45	56.45	
SLICET5 (CodeT5)	97.15	83.80	80.86	88.94	67.06	13.00	56.47	56.75	
SLICET5 (CodeT5+)	97.81	89.10	84.59	91.22	69.35	13.00	56.83	55.61	
		Mis	sing or Unmate	hed Brace	es				
GPT-4 (Zero-shot)	14.70	0.00	19.62	35.83	28.17	0.00	21.45	38.14	
GPT-4 (RAG)	53.19	0.00	66.62	60.46	53.45	0.00	49.98	55.76	
GPT-4 (COT)	55.09	0.00	67.43	60.14	61.88	4.00	44.63	57.81	
Gemma 3 (zero-shot)	13.53	0.00	23.89	39.09	32.63	0.00	19.93	42.32	
Gemma 3 (RAG)	56.49	0.00	60.46	40.51	46.57	0.00	38.90	41.70	
Gemma 3 (COT)	60.10	0.00	73.26	66.25	39.80	0.00	37.25	46.03	
NS-slicer (CodeBERT)	93.42	71.98	78.96	78.64	64.31	4.00	55.69	53.66	
NS-slicer (GraphBERT)	94.21	72.88	79.27	79.10	66.6	4.00	54.76	55.38	
SLICET5 (CodeT5)	96.14	73.00	81.57	81.96	64.21	10.00	56.54	58.00	
SLICET5 (CodeT5+)	96.34	79.10	82.81	87.75	66.37	9.00	62.19	60.28	

rely heavily on syntactic structure for downstream tasks such as program slicing, so brace errors severely distort their understanding. In contrast, missing class encapsulation and semicolon removes structure but preserves internal logic and dependency, which leads to less significant impact compared to unmatched braces.

6 Discussion

6.1 Applications of Constrained Decoding in other SE tasks

In this paper, we proposed a novel constrained decoding strategy that integrate lexical and syntactic constraints. It offers clear advantages that go beyond just program slicing. It can significantly improve the reliability of language models across a range of software engineering tasks by preventing invalid tokens and maintaining structural consistency in the output.

One practical use case is code completion and automated bug fixing. Lexical constraints help ensure that the model only suggests valid keywords or in-scope variables, which reduces the risk of hallucinated code, which is an issue frequently noted in code generation [41]. At the same time, syntactic constraints help the model generate code that fits neatly into the surrounding context without introducing syntax errors. This approach is also valuable for tasks like code refactoring and transformation. By applying constraints, we can ensure that updates follow the correct language conventions and maintain the structure of the original code. This helps reduce the chance of breaking changes during API migrations or modernization efforts.

6.2 Threats to validity

Threats to external validity In RQ3, we implement and evaluate three types of code corruption to simulate incomplete code snippets. Although, we simulate three most frequent cases on Stack Overflow by following previous studies [34, 39], real-world unusable code can be far more diverse in form and severity. We encourage future work to expand the evaluation to additional variants of syntactically

incomplete code and assess the robustness of learning-based program slicing methods across these cases.

This study focuses exclusively on Java and two recently released slicing datasets. However, our approach is readily adaptable to other programming languages such as Python, as well as to other extractive coding tasks. We encourage future research to extend this work to additional programming tasks where outputs are subject to structural or semantic constraints. Our experiments include GPT-4o-mini and Gemma-7B as foundation model baselines, though other models may yield better performance. Due to hardware limitations, we only fine-tuned lightweight versions of CodeT5 and CodeT5+ as slicers, and alternative checkpoints may lead to different results.

Threats to internal validity Our method is fine-tuned on the full CodeNet slicing dataset, which contains 30.8k examples. As a result, its performance is closely tied to the representativeness and diversity of this training data. Although our approach outperforms baseline methods on the out-of-domain LeetCode dataset, the model still struggles to generalize effectively, achieving a maximum exact match rate of 14%. This suggests that the current training set may lack sufficient variety. We encourage future work to expand the collection of slicing examples in order to develop more generalizable, learning-based program slicing methods.

7 Related Work

7.1 Static Program slicing

Traditional static slicing tools (e.g., JavaSlicer [9], CPP-Slicer [13]) construct System Dependence Graphs (SDGs) via AST parsing and dependency analysis, then perform slicing as a graph reachability problem [7]. However, these methods require fully compilable code (at least method-granular) [38], limiting their utility for real-world incomplete code snippets like those in Stack Overflow posts [39]. Recently, learning-based methods have been developed to address this limitation [26, 38]. Yadavally et al. [38] proposed NS-slicer to predict the dependency between slicing criteria (e.g., variable) and statements by analyzing their similarity using CodeBERT/Graph-CodeBERT for encoding. However, this approach is not an end-toend solution. Most notably, it can only utilize the information of a target statement while ignoring the surrounding context, which makes it challenging to process out-of-domain samples. Shahandashti et al. [26] investigated the application of state-of-the-art large foundation models, such as GPT-4o [2], GPT-3.5-turbo, and Gemma [21], to program slicing tasks. Their approach includes leveraging advanced prompting techniques such as retrieval-augmented generation (RAG) and chain-of-thought (COT) reasoning. However, employing foundation models is computationally expensive and yields unsatisfactory results. For instance, in slicing tasks of LeetCode solutions, these models often produce outputs with a zero exact match rate.

Unlike existing learning-based approaches, we proposed a novel lightweight framework that improves existing language model by integrating copy mechanism and constrained decoding.

7.2 Pre-trained Language Models for Software Engineering Applications

Recent advances in pre-trained language models have significantly enhanced capabilities in code understanding and generation. Sequence-to-sequence (seq2seq) architectures, in particular, have emerged

as a dominant paradigm, achieving state-of-the-art performance when fine-tuned for various software engineering tasks such as code summarization [8, 19, 20], translation [40, 42, 43], and type inference [23, 35]. For instance, Wei et al. [35] formulated type inference as a seq2seq code infilling task. They fine-tuned CodeT5 to automatically predict missing type annotations. To enhance the vanilla CodeT5 model, they leverage static analysis to construct dynamic contexts for each code element whose type signature is to be predicted. In addition, they propose an iterative decoding scheme that incorporates previously predicted types into the input context, significantly improving prediction accuracy. To better support developers in software maintenance tasks due to code change, such as writing a description for the intention of the code change, or identifying defect-prone code changes, Lin et al. [19] introduced CCT5, a domain-adapted version of CodeT5 specialized for code changes. They collected a large-scale dataset of code diffs and their associated commit messages, and designed five pretraining tasks to capture diverse aspects of code evolution. Zhu et al. [43] proposed GrammarT5, a grammar-integrated extension of the CodeT5 model. Rather than representing code as plain token sequences, GrammarT5 introduces a novel Tokenized Grammar Rule Sequence (TGRS), which embeds syntax information from grammar rule sequences used in syntax-guided generation. To further enrich syntactic understanding, the authors proposed two grammar-aware pretraining objectives, Edge Prediction (EP) and Sub-Tree Prediction (STP) to help differentiate grammar rules across languages. GrammarT5 achieves state-of-the-art performance on a range of code generation, summarization, and translation tasks.

Our work builds on lightweight seq2seq models, CodeT5 and CodeT5+, to the new downstream task static program slicing. Unlike previous efforts, we enhance the original architecture of CodeT5(+) with a copy mechanism and incorporate external lexical and syntactic knowledge by constraining the decoding process.

8 Conclusion

In conclusion, this paper presents SLICET5, a learning-based approach for static program slicing that leverages a copy-enhanced encoder-decoder Transformer architecture. Our improved model effectively predicts accurate dependency relationships and generates element-preserving program slices. In addition, we introduce a lexically and syntactically constrained decoding strategy, enabling the model to better capture both semantic information and syntactic structure during slicing. We evaluate SLICET5 on two program slicing datasets, where it achieves state-of-the-art performance compared to strong baselines, including advanced foundation models and prompting techniques. Specifically, SLICET5 outperforms the best baseline by at least 6.4% and 27% in terms of ExactMatch on the two datasets, respectively. Furthermore, ablation studies confirm that both the copy mechanism and the constrained decoding strategy contribute significantly to the overall performance improvements.

References

- Mithun Acharya and Brian Robinson. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In <u>Proceedings of</u> the 33rd international conference on software engineering. 746–755.
- [2] 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 (2023)
- [3] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. <u>ACM SIGPlan Notices</u> 25, 6 (1990), 246–256.
- [4] David Binkley, Nicolas Gold, and Mark Harman. 2007. An empirical study of static program slice size. ACM Transactions on Software Engineering and Methodology (TOSEM) 16, 2 (2007), 8-es.
- [5] 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. 1–13.
- [6] 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 <u>Findings of</u> the Association for Computational Linguistics: EMNLP 2020. 1536–1547.
- [7] 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 (Berlin, Germany). Springer-Verlag, Berlin, Heidelberg, 146–151. doi:10.1007/978-3-031-17108-6
- [8] Shuzheng Gao, Wenxin Mao, Cuiyun Gao, Li Li, Xing Hu, Xin Xia, and Michael R Lyu. 2024. Learning in the wild: Towards leveraging unlabeled data for effectively tuning pre-trained code models. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 1–13.
- [9] The MiST group. 2022. slicer. https://github.com/mistupv/JavaSlicer.
- [10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. [n.d.]. Graph-CodeBERT: Pre-training Code Representations with Data Flow. In <u>International</u> Conference on Learning Representations.
- [11] Mark Harman and Robert Hierons. 2001. An overview of program slicing. software focus 2, 3 (2001), 85–92.
- [12] Pengfei He, Shaowei Wang, and Tse-Hsun Chen. 2025. CODEPROMPTZIP: Code-specific Prompt Compression for Retrieval-Augmented Generation in Coding Tasks with LMs. <u>CoRR</u> abs/2502.14925 (February 2025). https://doi.org/10.48550/arXiv.2502.14925
- [13] Oliver Hechtl. 2021. slicer. https://github.com/dwat3r/slicer.
- [14] 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). IEEE, 1529–1541.
- [15] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured Chain-of-Thought Prompting for Code Generation. <u>ACM Trans. Softw. Eng. Methodol.</u> 34, 2, Article 37 (Jan. 2025), 23 pages. doi:10.1145/3690635
- [16] 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, Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics, Online, 894–908. doi:10.18653/v1/2021.naacl-main.69
- [17] Xuan Li, Shuai Yuan, Xiaodong Gu, Yuting Chen, and Beijun Shen. 2024. Few-shot code translation via task-adapted prompt learning. <u>Journal of Systems and Software</u> 212 (2024), 112002.
- [18] 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 (2018).
- [19] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. CCT5: A Code-Change-Oriented Pre-trained Model. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (San Francisco, CA, USA) (ESEC/FSE 2023). Association for Computing Machinery, New York, NY, USA, 1509–1521. doi:10.1145/3611643.3616339
- [20] Antonio Mastropaolo, Matteo Ciniselli, Luca Pascarella, Rosalia Tufano, Emad Aghajani, and Gabriele Bavota. 2024. Towards summarizing code snippets using pre-trained transformers. In Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension. 1–12.
- [21] Deep Mind. 2024. Gemma 2: Improving Open Language Models at a Practical Size. arXiv:2408.00118 [cs.CL] https://arxiv.org/abs/2408.00118
- [22] 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 (2023), 974–996.
- [23] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023. Generative type inference for python. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 988– 999
- [24] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundare-san, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297 (2020).
- [25] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In <u>Proceedings of the 55th</u> Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics.
- [26] 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 (2024).
- [27] 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). 293–300. doi:10.1109/TrustCom50675.2020.00049
- [28] 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. 252–262.
- [29] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. <u>Advances in neural information processing systems</u> 27 (2014).
- [30] Bayu Distiawan Trisedya, Jianzhong Qi, Haitao Zheng, Flora D Salim, and Rui Zhang. 2023. TransCP: a transformer pointer network for generic entity description generation with explicit content-planning. IEEE Transactions on Knowledge and Data Engineering. 35, 12 (2023), 13070–13082.
 [31] Zhaopeng Tu, Zhengdong Lu, Yang Liu, Xiaohua Liu, and Hang Li. 2016. Mod-
- [31] 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.
- [32] 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. 1069–1088.
- [33] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 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. 8696–8708.
- [34] 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).
- [35] Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. TypeT5: Seq2seq Type Inference using Static Analysis. In The Eleventh International Conference on Learning Representations. https://openreview.net/forum?id=4TyNEhI2GdN
- [36] Mark Weiser. 1984. Program Slicing. 10, 4 (July 1984), 352–357. doi:10.1109/ TSE.1984.5010248
- [37] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. <u>ACM SIGSOFT Software Engineering Notes</u> 30, 2 (2005), 1–36.
- [38] Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2024. A Learning-Based Approach to Static Program Slicing. Proc. ACM Program. Lang. 8, OOP-SLA1, Article 97 (April 2024), 27 pages. doi:10.1145/3649814
- [39] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: an analysis of stack overflow code snippets. In <u>Proceedings of the 13th</u> <u>International Conference on Mining Software Repositories.</u> 391–402.
- [40] Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. <u>arXiv preprint arXiv:2407.07472</u> (2024).
- [41] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. <u>Proceedings of the</u> ACM on Software Engineering 2, ISSTA (2025), 481–503.
- [42] Ming Zhu, Mohimenul Karim, Ismini Lourentzou, and Daphne Yao. 2024. Semi-supervised code translation overcoming the scarcity of parallel code data. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. 1545–1556.
- [43] 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 (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 76, 13 pages. doi:10.

1145/3597503.3639125 [44] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection.

 $\underline{\text{IEEE Transactions on Dependable and Secure Computing}}\ 18,\ 5\ (2019),\ 2224-2236.$