On Adversarial Robustness of Synthetic Code Generation

Mrinal Anand¹, Pratik Kayal¹, and Mayank Singh¹

¹Indian Institute of Technology Gandhinagar, India {mrinal.anand,pratik.kayal,singh.mayank}@iitgn.ac.

Abstract. Automatic code synthesis from natural language descriptions is a challenging task. We witness massive progress in developing code generation systems for domain-specific languages (DSLs) employing sequence-to-sequence deep learning techniques in the recent past. In this paper, we specifically experiment with ALGOLISP DSL-based generative models and showcase the existence of significant dataset bias through different classes of adversarial examples. We also experiment with two variants of Transformer-based models that outperform all existing ALGOLISP DSL-based code generation baselines. Consistent with the current state-of-the-art systems, our proposed models, too, achieve poor performance under adversarial settings. Therefore, we propose several dataset augmentation techniques to reduce bias and showcase their efficacy using robust experimentation.

Keywords: Automatic code generation · Adversarial attacks.

Can computers automatically synthesize a program? Is it possible to generate code from a human-readable textual description? These are fundamental questions for any field of research, but particularly well-motivated in Computer Science. Automatic program synthesis (popularly known as 'Code Generation') is the process of generating code in a particular language from a code/description in some other language. Figure 1 shows an illustrative example comprising a textual description, its corresponding LISP program in a tree format, and few input/output (I/O) pairs. Similar examples can be easily constructed from other popular programming languages like Python, Java, and C/C++, using different coding platforms¹.

The Two Paradigms Of Code Synthesis The rich literature on automatic code generation is broadly classified into two categories: (i) *programming by example* (PBE) [15], and (ii) *programming by descriptions* (PBD) [13]. The PBE paradigm leverages input-output (I/O) examples (\mathbb{E} in Figure 1) alone to automatically construct a program (\mathbb{O} in Figure 1) that satisfies these examples. Several real-world computer science applications use the PBE paradigm for automatic code synthesis. For example, FlashFill [9], Deep-Coder [2] and RobustFill [8]. In contrast, PBD paradigm uses descriptions (\mathbb{I} in Figure 1) with corresponding zero or few I/O code instances (\mathbb{E} in Figure 1) to automatically constructs a program (\mathbb{O} in Figure 1). The PBD paradigm has recently received major attention, thanks to the surge in the neural sequence-to-sequence approaches [13, 17, 25].

¹ For, e.g., GeeksForGeeks (https://www.geeksforgeeks.org)

2 Anand et al.

However, the progress is fairly limited due to unavailability of large-scale real datasets. Polosukhin *et al.* [17] proposed a large-scale synthetic dataset, ALGOLISP, and a corresponding neural architecture SEQ2TREE. SEQ2TREE generates *Abstract Syntax Trees* (AST) from textual descriptions. The current state-of-the-art, SKETCHADAPT [16], uses a combination of neural and sketch-based approaches for program synthesis. As a downside of neural modeling, both paradigms necessitate large volumes of datasets in fully supervised settings. However, due to the rare availability of good quality and large-scale real datasets, these approaches leverage synthetic datasets. Popular synthetic datasets include ALGOLISP [17], NAPS [25], Karel [5], WikiSQL [26] and dataset of bash commands [13]. In this paper, we extensively experiment with ALGOLISP dataset under PBD settings ($\mathbb{I} \rightarrow \mathbb{O}$), see Figure 1).

Robustness Against Adversarial Attacks Recently, we witness a growing interest in evaluating deep learning models against adversarial attacks [24]. However, to the best of our knowledge, we do not find any work that evaluates the adversarial robustness of neural program synthesis systems. Specifically, we are interested in answering questions like "Are generative models trained on synthetically constructed datasets sufficiently robust against adversarial attacks?" In this paper, we evaluate automatic program synthesis models trained on synthetic datasets against adversarial attacks. We propose

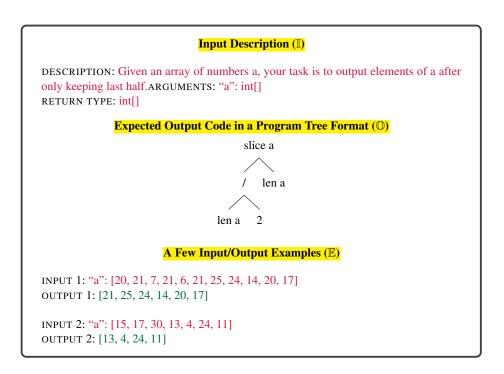


Fig. 1: An illustrative example comprising a textual description (\mathbb{I}), its corresponding LISP program in a tree format (\mathbb{O}), and a few input-output examples (\mathbb{E}).

different classes of adversarial attacks and show the inability of state-of-the-art code generation models to generalize to extremely elementary test examples.

Self-supervised Paradigm For Automatic Code Synthesis The field of Natural Language Processing (NLP) is witnessing a significant paradigm shift towards self-supervised learning. Thus, some of the pertinent questions from the current context can be "Can self-supervised learning paradigm provide near state-of-the-art program synthesis performance?" We experiment with several variants of Transformer-based encoder-decoder models to showcase that self-supervised models significantly outperform previous neural approaches like RNN and LSTM based attention architectures [3, 16, 17]. Besides, to understand the adversarial robustness, we answer questions like "Are self-supervised models robust against adversarial attacks?" We show that the transformer architecture uses relatively lesser training text (only problem description and no I/O pairs) and is more robust against adversarial attacks than traditional neural code generation models. Debiasing The Synthetic Datasets Towards the end, we conduct extensive experiments to showcase the role of the synthetic datasets in introducing training bias and propose simple dataset augmentation strategies to debias the synthetic datasets. Extensive experimentation shows an increase in the robustness against adversarial attacks.

Key Contributions: To summarize, the key contributions of our work are:

- Identification and formalization of adversarial attacks on the existing program synthesis tools as well as an evaluation of these tools on a test suite of manuallygenerated adversarial examples;
- A new automatic code synthesis system, AUTOCODER, and its robust comparison against state-of-the-art neural code generation systems; and
- A generative algorithm to automatically augment the original AlgoLISP dataset with a large number of adversarial examples.

1 Problem Definition

After introducing the general program synthesis paradigm in the previous section, we are now in a position to define the DSL-based program synthesis problem formally. Given a DSL L, we aim to learn a synthesis algorithm A such that given a set of text description and its corresponding code snippet, $(i_1,o_1),\ldots,(i_n,o_n)$. The synthesis algorithm A learns a program $P \in L$, such that it satisfies all the corresponding input-output test cases e_j 's of description (NL) and code snippet (i_j,o_j) pair, i.e.,

$$\forall j, k : P(e_{j(k,in)}) = e_{j(k,out)} :$$

$$1 \le j \le n \& 1 \le k \le l$$

$$(1)$$

Where, $e_{j(k,in)}$ and $e_{j(k,out)}$ represents the input and output of the k^{th} test case of description and code snippet (i_j,o_j) pair, respectively. Here, l represents the number of test cases corresponding to each description and code snippet pair. Note that, in Eq. 1, we match the test cases and not the actual generated code; a given textual description can possibly generate structurally dissimilar variants of the ground truth code, preserving the logical functionalities.

4 Anand et al.

Formally, an adversarial text description (NL') for a program synthesis model generates a program (P_{adv}) such that:

$$\forall j, k : P_{adv}(e_{j(k,in)}) \neq e_{j(k,out)} :$$

$$1 \leq j \leq n \& 1 \leq k \leq l$$

$$(2)$$

under the constraint that-

$$||NL' - NL|| \le \delta$$

where δ denotes the amount of perturbation. Let P_{orig} denotes the program corresponding to NL and P_{adv_sol} corresponds to a program that can correctly solve NL'. Depending on whether P_{adv_sol} is the same as P_{orig} , attacks can be classified into the following two categories:

Program Invariance Attacks: In these types of attacks, we perturb NL such that the original program is also a solution of NL' i.e., $(P_{orig} = P_{adv_sol})$.

Program Directional Attacks: In these type of attacks, we perturb NL such that the original program is not a solution of NL' i.e., $(P_{orig} \neq P_{adv_sol})$.

2 Dataset

In this paper, we use a synthetically constructed code generation dataset ALGOLISP [17]. ALGOLISP is constructed over a domain-specific language (DSL), inspired by Lisp. Instead of existing programming languages (like Python, C, or Java), DSLs provide flexibility in converting to other target languages and adding constraints to simplify its automated generation [17]. The dataset comprises the problem description and the corresponding implementations of the problem in a Lisp-inspired DSL. Each problem description is accompanied by a code snippet and 10 test cases. Each test case is an input/output pair, where input is to be fed into the synthesized program, and output represents the expected output the program should produce. Figure 1 illustrates an example problem showing a textual description, its corresponding Lisp DSL program tree, and few I/O test pairs. Overall, the dataset contains 100,000 problems with average textual description length and average code snippet length of 37.97 and 45.13 characters. The ALGOLISP dataset comprises train, validation, and test split of 79214, 9352, and 10940 examples, respectively. The average depth of the program tree is 10.28. Table 1 lists the detailed statistics of original ALGOLISP dataset.

	Original	Filtered
No. of instances	100,000	90,153
Avg. text length	37.97	37.75
Avg. code depth	10.35	10.28
Avg. code length	45.13	44.86
Vocabulary size	288	287

Table 1: Statistics of the ALGOLISP dataset.

In 2018, Bednarek *et al.* [3] showed multiple instances of compilation errors in the original ALGOLISP dataset. Specifically, the DSL compiler fails to pass I/O pairs with ground truth code. They, therefore, constructed a filtered subset of ALGOLISP dataset containing only those problem instances that pass all the input-output test cases². Overall, the filtered dataset contains 90,153 instances. Table 1 also details the statistics of the filtered ALGOLISP dataset. To the best of our knowledge, except NAPS [25] and Karel [5], no similar code synthesis dataset exists that contains problem description along with the test cases and other meta information. Popular datasets like JAVA [11], WikiSQL [26] only contain problem description and the corresponding code, leading to limitations in evaluating structurally different but logically similar synthesized codes. Although NAPS and Karel contains all the required meta-information, Karel does not deal with any natural language; it is a robotic programming language. On the other hand, in our internal data analysis, NAPS shows several data inconsistencies³ such as the presence of a long sequence of characters like abcdabcd, which conveys no meaning and inconsistent tokenization of sentences.

3 The SOTA Code Generation Models

In this paper, we thoroughly experiment with state-of-the-art DSL-based code generation model, **SketchAdapt** [16]. SketchAdapt (hereafter 'SA') synthesizes programs from textual descriptions as well as input-output test examples. It combines neural networks and symbolic synthesis by learning an intermediate 'sketch' representation. It has been demonstrated empirically that SA recognizes similar descriptive patterns as effectively as pure RNN approaches while matching or exceeding the generalization of symbolic synthesis methods. The SA system consists of two main modules: 1) a sketch generator and 2) a program synthesizer. Given an encoded input description, the sketch generator generates a distribution over program sketches. The generator is trained using a sequence-to-sequence recurrent neural network with attention to assign a high probability to sketches that are likely to yield programs satisfying the specification. The program synthesizer takes a sketch as a starting point and performs an explicit symbolic search to "fill in the holes" in order to find a program that satisfies the specification. The pre-trained model, along with the relevant codebase, is available at https://github.com/mtensor/neural sketch.

Additionally, we found two more relevant baselines, Structure-Aware Program Synthesis [3] and SEQ2TREE model [14]. Structure-Aware Program Synthesis (hereafter, 'SAPS') adapts the program synthesis problem under the Neural Machine Translation framework by employing a bi-directional multi-layer LSTM network for generating code sequence corresponding to textual descriptions. The Seq2Tree model consists of a sequence encoder and a tree decoder. The sequence encoder reads the problem description, and a tree decoder augmented with attention computes probabilities of each symbol in a syntax tree node one node at a time. However, both baselines cannot be implemented

² Even though we find few instances that resulted in partial passing of test cases.

³ The NAPS dataset is very noisy due to crowd-sourcing.

Model	Accuracy Scores
SA	0.958
SAPS*	0.929
SEQ2TREE*	0.858^{\dagger}
VAC	0.968
GAC	0.963

Table 2: Comparing state-of-the-art code generation models. * represents accuracy scores taken, verbatim, from the corresponding papers due to unavailability of code or pretrained model. † represents accuracy scores on the original test set.

due to the unavailability of a code repository or the pre-trained model⁴. We, therefore, thoroughly experiment with *SA* as the only baseline system.

Evaluating Generation Performance: We evaluate the above state-of-the-art code synthesis systems on the filtered ALGOLISP dataset. We, verbatim, follow the experimental settings presented in SA and SAPS. Note that, in the filtered dataset, the number of test cases is lesser than the original dataset. At the same time, the training data remains the same as the original dataset. We compute accuracy scores (A) for performance evaluation on holdout test set defined as $A = \frac{n}{N}$, where n is the number of problems for which the generated code passes all the 10 test cases and N is the total number of problems in the holdout test set. Table 2 shows the accuracy scores for three state-of-the-art code generation systems. As expected, SA outperformed the rest of the two baseline systems with a significant margin.

4 AUTOCODER

In this section, we discuss our implementation of the neural model AUTOCODER to address the automatic code generation problem. Recently, Transformers [20] have shown state-of-the-art performance for several Natural Language Processing tasks [21, 12, 1], including machine translation, classification, etc. Inspired by its success, we propose a transformer-based code-generation model to generate code based on natural language descriptions automatically. Specifically, the model encodes the textual description using multiple layers of encoders and then decodes a program token-by-token while attending to the description. The basic pipeline of our proposed model is analogous to the simpler sequence-to-sequence models that are employed for similar generation tasks. These models usually have an encoder-decoder architecture [6, 4]. The encoder maps an input sequence of symbol representations consisting of tokens x_1, x_2, \cdots, x_n to intermediate representation which the decoder then generates an output sequence y_1, y_2, \cdots, y_m of symbols one element at a time. At each step, the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next output symbol. As depicted in Figure 2, we utilize the core Transformer [20] model implementation and propose significant structural alterations in the attention module to develop AUTOCODER.

⁴ The results cannot be reproduced due to missing experimental details.

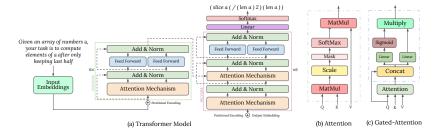


Fig. 2: Transformer model. (b) The self-attention mechanism. (c) The gated-attention mechanism.

The encoding and decoding layers: We keep the number of encoder layers (=6) the same as the core Transformer model. Similar to core implementation, we keep the output dimension size as 512 in all the encoding sub-layers of the model as well as in the embedding layers. The decoder side is also stacked with six identical layers. In Figure 2a, the sub-layers of encoder and decoder layers are described using standard notations.

The different attention mechanisms: Attention mechanisms have become an integral part of sequence modeling tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences. Specifically, we experiment with two attention mechanisms: (i) vanilla self-cross attention and (ii) gated cross attention. The Vanilla self-cross attention is the basic attention mechanism used in traditional transformer models [20]. The self-attention module relates different positions of an input sequence in order to compute a representation of the input sequence. This module is present in both encoder and decoder layers. The cross attention module relates different positions of an input sequence to the output sequence. This module connects the encoder and decoder components. We term AUTOCODER variant that uses the standard self-cross attention module as VANILLA-AUTOCODER (VAC).

$$f_{SA} = f_{dot}(Q_e, K_e, V_e) = softmax\left(\frac{Q_e K_e^T}{\sqrt{d}}\right) V_e$$
 (3)

$$f_{CA} = f_{dot}(Q_e, K_d, V_d) = softmax\left(\frac{Q_e K_d^T}{\sqrt{d}}\right) V_d$$
 (4)

where f_{dot} represents scaled dot product attention [20], (Q_e, K_e, V_e) denotes encoder sequence representations in terms of query, key and value respectively, and (Q_d, K_d, V_d) denotes decoder sequence representation in terms of query, key and value respectively.

The gated cross attention mechanism filters out the unnecessary part of the sequence by attending over the generated cross attention scores f_{CA} and determining the relevant attention. The gated cross attention module (f_{GA}) uses a sigmoidal gating mechanism for filtering out irrelevant cross attention while decoding the output. It generates an information vector(i) which carries relevant representation of the input vector and an attention gate (g) that filters the relevant attention scores. Now this filtered attention is applied to the information vector to obtain attended information, or the relevant

information.

$$f_{GA} = \sigma(W_q^g Q_e + W_v^g f_{CA} + b^g)$$

$$\odot(W_q^i Q_e + W_v^i f_{CA} + b^i)$$
(5)

 σ denotes sigmoid activation, \odot denotes element-wise product, W_q^i and W_v^i represent weight matrices corresponding to value query and value at *information vector*, respectively, W_q^g and W_v^g represent weight matrices corresponding to value query and value at *attention gate*, respectively. Note that, $\{W_q^i, W_v^i, W_q^g, W_v^g\} \in \mathbb{R}^{d \times d}$ and $\{b^i, b^g\} \in \mathbb{R}^d$. Figure 2c shows the gated cross-attention architecture. We term AUTOCODER variant that uses gated cross attention as **GATED-AUTOCODER** (**GAC**).

Comparing AUTOCODER against baselines: Table 2 also compares AUTOCODER variants against baseline systems. Both variants outperformed the three baselines. Among the two variants, VAC performed marginally better than GAC. To summarize, the results showcase that even simple Transformer variants can result in high gains in code synthesis.

5 The Adversarial Experiments

5.1 Adversarial Attack Types

We define five classes of adversarial examples. All our proposed attacks are black-box un-targeted attacks. Our attacks do not have any knowledge of the target model, nor does it have any information about the gradients and model parameters. Table 3 shows representative examples of actual descriptions and corresponding adversarial descriptions. The classes are:

- 1. **Variable Change (VC):** Changing single and multi-character variables and argument names in the original problem description, input arguments, and program trees to examine if the model correctly generates the corresponding output code.
- 2. **Redundancy Removal (RR):** Removing filler or redundant words without affecting the overall meaning of the input description.
- Synonym Replacement (SR): Replacing words with their corresponding synonyms.
- 4. **Voice Conversion (VoC):** Converting a problem description in the active voice to its corresponding passive voice.
- 5. **Variable Interchange (VI):** Interchanging variable names in problem descriptions comprising multiple variables.

The classes **VI** and **VC** belong to program directional attack category, whereas classes **RR**, **SR**, **VoC** belong to program invariance attack category. For example, consider the representative example for **VC** class in Table 3, changing variable name from a to b led to the change in the ground truth program that can solve the problem i.e. from (strlen a) to (strlen b). Now, model predicting any other token except the variable b is an adversary. In case of **RR**, removing redundant token is a program invariance perturbation, hence the ground truth program remains unchanged.

5.2 Adversarial Performance

In this section, we discuss the adversarial instance construction process. We construct adversarial examples using the holdout test instances following classwise constraints

```
Class Representative Example
     OD: Given a string a, what is the length of a.
     OO: (strlen a)
     AD: Given a string b, what is the length of b.
     AO: (strlen a)
      OD: Given a number a, compute the product of all the numbers from 1 to a.
     OO: (invoke1 (lambda1 (if ( \leq arg1 1 )1(*( self( -arg1 1 ))
      arg1 ))) a)
 RR AD: Given a number a, compute the product of the numbers from 1 to a.
     AO: ( * a 1 )
     OD: consider an array of numbers, what is reverse of elements in the given array that are
      \mathbf{OO}: (reverse ( filter a ( lambdal ( == ( % argl 2 )1))))
     AD: consider an array of numbers, what equals reverse of elements in the given array that
     are odd
      AO: (reduce ( filter a ( lambda1 ( == ( % arg1 2 )1))))
     OD: Given a number a, your task is to compute a factorial
     OO:
                   invoke1(lambda1(if(<= arg1 1) 1 (*(self(-arg1 1))</pre>
      arg1)))a)
VoC | AD: Your task is to compute a factorial, given a number a
     AO: (filter a ( partial1 b >))
     OD: you are given an array of numbers a and numbers b, c and d, define e as elements in
     a starting at position b ending at the product of c and d (0 based), what is e
     OO: ( slice a d ( * c b ) )
     AD: you are given an array of numbers a and numbers b, c and e, define d as elements in
     a starting at position b ending at the product of c and e (0 based), what is d
     AO: ( slice a d ( * c b ) )
```

Table 3: Representative examples from each adversarial class. Here, OD, OO, AD, and AO represent the original description, original output, adversarial description, and adversarial output, respectively.

in a semi-supervised fashion. For example, an adversarial instance belonging to the **VI** class can only be generated if the problem description contains two or more variables. In addition, we used several NLP libraries for basic linguistic tasks. For example, we use the NLTK library to stochastically remove some stopwords from the program descriptions to generate instances for **RR** class. Similarly, we leverage POS tagging to identify active/passive voice to construct instances for the **VoC** class. And POS tagging and Wordnet hierarchies to construct instances for **SR** class. Overall, we use about 1000 adversarial instances, equally divided per adversary class, for evaluating program synthesis systems.

Table 4 presents generation performance of SA under adversarial settings using error percentage i.e. (100 - Accuracy %), lower the error % better is the adversarial robustness. Surprisingly, SA fails to generalize and produce significantly poor results under the adversarial setting. In particular, it performs very poorly on the **VoC** and variable **VI** classes. This is because the model does not predict the correct code when the sentences

Adv. Class	Error (%)			Distance		
	SA					BERT
VC	48.0	42.5	42.5	2.24	.05	.005
RR	4.70	3.70	3.20	4.55	.13	.044
SR	5.70	8.10	8.10	1	.03	.013
VoC	70.2	24.9	24.4	16.54	.54	.015
VI	70.0	67.7	67.2	4.2	.08	.043

Table 4: Error percentage (columns 2–4) of SA, VAC and GAC for different adversarial classes. Distance between (columns 5–7) adversarial and the corresponding original description.

that are generally active in the dataset are converted to passive sentences. Further, in our analysis, if variables b and d are interchanged, the model fails to recognize this change and outputs code as if no change has been done on the input sentences. Table 4 also presents generation performance of AUTOCODER under adversarial settings. AUTOCODER variants show more robustness than SA in four out of five classes. We observe that one of the possible reason for the poor performance of AUTOCODER variants is incorrect cross attending. For example, the variable a in the output is not attending the corresponding variable a in the problem description.

Even though AUTOCODER showed more robustness than SA under adversarial settings, we observe a significant drop in the overall performance in both systems. We claim that the performance drop under the adversarial setting is attributed to bias in the synthetic dataset generation process. Some of the potential bias scenarios are: (1) small set of chosen variable names, (2) limited number of operations, (3) limited vocabulary usage, (4) variables occur in a sequential and alphabetical manner.

5.3 Measuring Extent and Quality of Perturbations

Extent of Perturbations To measure the extent of perturbation in our proposed adversarial attacks, we experiment with the following two distance metrics:

Edit Distance: We use the popular Levenshtein distance (hereafter, 'Lev') to calculate the distance between adversarial description and the corresponding original description. It is defined as the minimum number of edit operations (delete, insert and substitute) required to convert one string to the other. We also report the ratio of Levenshtein distance to the length of sentences (hereafter, 'LevR') to measure the extent of perturbation per length of the sentence. Table 4 (columns 5 and 6) shows distance values for the five adversarial classes. Except for **VoC** where the entire sentence structure changes, the other classes comprise examples constructed from significantly low perturbations. Note that, we limit the perturbation rate in **SR** to 1, as higher perturbations were leading to out-of-vocabulary problems and other grammatical inconsistencies.

Embedding Similarity: We also measure the cosine similarity between adversarial description and the corresponding original description using sentence embeddings derived from pretrained model BERT [7]. The sentence embeddings are derived from a Siamese network trained using triplet loss [18]. We convert the similarity value into

Adv. Class	Grammatical Score			Naturalness Score		
Auv. Class	Original	Adversarial	%confusion	Original	Adversarial	%confusion
VC	4.2	4.25	99%	3.95	3.85	98%
RR	4.20	3.60	88%	4.15	3.60	89%
SR	4.40	3.85	90%	4.25	3.90	92%
VoC	4.00	3.45	89%	3.90	3.65	98%
VI	3.70	3.50	96%	3.45	3.60	95%
Average	4.10	3.73	92.4%	3.94	3.71	94.4%

Table 5: Class-wise comparison of human evaluation results

a distance value by subtracting it by 1 (hereafter, 'BERT'). We keep the embedding length as 768. Table 4 (column 7) reiterate the distance-based observations. Note that, as contextual embeddings successfully capture voice-related changes, the adversarial class **VoC** also shows low perturbation distance.

Human Evaluation We employ two undergraduate students expert in programming to evaluate the quality of constructed adversarial attacks. For this experiment, we randomly select ten instances from each adversary class along with the corresponding original instance (a sample dataset of a total of 100 instances). We first-of-all educate evaluators about the task by presenting them a set of program descriptions from the original ALGOLISP dataset. Next, we instruct them to evaluate each instance in the sampled set based on the following two criteria:

Grammatical Correctness: We ask the evaluators to rate the grammatical correctness of the sentences on a scale of 1–5. The rating of 1 being 'completely grammatically incorrect description' and 5 representing 'grammatically sound and correct'.

Naturalness: We also ask the evaluators to judge the quality of the sentences on the basis of *naturalness* of the texts i.e., how likely the test samples are drawn from the original data distribution. We ask to rate each sample on a scale of 1–5. The rating of 1 being 'completely outside data distribution/unfamiliar example' and 5 representing 'definitely from original data distribution'.

We summarize the human evaluation experiment in Table 5. As evident from the table, the grammatical score and naturalness score of original sentences are higher than adversarial sentences. The evaluators were correctly able to identify the minor grammatical mistakes present in the **RR** class. Also, since changing the variables only does not add much human notable noise, evaluators were finding it difficult to distinguish between original sentences and adversarial sentences for **VC** and **VI** classes as depicted in the results of Table 5. We also present the % confusion score that reflects how much difficulty evaluators are facing in distinguishing between adversarial and original sentences. Mathematically, it is defined as $\%confusion = \left(1 - \frac{|original\ value\ -\ adversarial\ value\ |}{5}\right) \times 100$. The high % confusion scores in Table 5 showcase the quality of constructed adversarial examples.

6 ALGOLISP++: The Debiased Dataset

To mitigate the poor performance of SA and AUTOCODER variants under adversarial settings, we extend the original ALGOLISP by adding a highly diversified collection of examples. However, as we see in previous sections, the automatic synthesis of instances is a challenging task. We, therefore, present a automatic instance generation algorithm inspired by the concepts of basic string editing [22], back translation [19] and neural editing [10]. We propose the following three classes of operations:

1. Basic editing operations (BE) We randomly edit tokens from the descriptions except few tokens that convey high semantic importance with respect to the programming languages. For example, the token concatenation conveys special meaning to the sentence and should not be edited. We reserve $\sim 10\%$ of the vocabulary tokens as non-editable. The non-editable list includes tokens such as times, sum, digits, maximum, prime, last, etc. We define a parameter α to regulate the number editable tokens in a sentence. The number of editable token is given by $\lfloor \alpha L \rfloor$, where L is the length of the sentence. In our experiments, we assign $\alpha=0.1$. Next, we define three basic token-level edit operations:

Random Deletion (RD): Randomly removing one or more words from the sentences.

Random Insertion (RI): Randomly inserting one or more words in the sentences. **Random Substitution (RS):** Randomly substituting one or more words in the sentences.

For RI and RS, we use BERT [7] uncased language model trained on monolingual English language. In RI, we randomly add $\lfloor \alpha L \rfloor$ masked tokens to the input sentence and predict tokens corresponding to these masked positions. In the case of RS, we randomly select $\lfloor \alpha L \rfloor$ editable tokens in a problem description and mask them. Further, the masked sentence is fed to the pre-trained BERT model to predict the

Ω	os	Consider an array of numbers a, your task is to find if a reads the same from both ends. Consider an array of numbers a, your task to find if a reads same from both ends.
~	FS	Consider an array of numbers a, your task to find if a reads same from both ends.
	OS	Consider an array of numbers a, your task to find if a reads same from both ends.
\mathbf{F}	FS	Consider on an array of regular numbers a, your task is to find if a reads the same from
		both ends
S	OS	Consider an array of numbers a, your task is to find if a reads the same from both ends Consider an array of integers a, your task is to find if a reads the integers from both ends
~	FS	Consider an array of integers a, your task is to find if a reads the integers from both ends
	OS	Given arrays of numbers a and b, what is the difference of elements of a and median in b.
_	IS	Was ist der Unterschied zwischen den Elementen von a und dem Median in b, wenn Arrays
BT		von Zahlen a und b gegeben sind?
	FS	What is the difference between the elements of a and the median in b given arrays of
		numbers a and b?
~	OS	you are given an array of numbers a, find not prime values in a you are given at array of numbers a, find not prime values in a
A	FS	you are given at array of numbers a, find not prime values in a

Table 6: Illustrative examples of different operations to modify ALGOLISP dataset. Here OS, IS and FS represents original, intermediate and final sentence, respectively.

Name	Dataset Statistics				Accuracy Scores		
Name	Instances	Vocab. size	Avg. length	SA	VAC GA	C	
ALGOLISP	79214	292	38.17	0.958	0.968 0.96	53	
ALGOLISP++	142644	3152	37.97	0.944	0.943 0.94	1 7	

Table 7: Statistics (columns 2–4) of ALGOLISP and ALGOLISP++ training datasets. Accuracy scores (columns 5–7) of SA, VAC and, GAC for two ALGOLISP variants.

Adv. Class	Error (%)				
Auv. Class	SA	VAC	GAC		
VC	41.5(48)	36.00(42)	34.40 (42)		
RR	3.70(5)	3.20 (4)	4.20(5)		
SR	4.40 (6)	4.70(9)	8.20(9)		
VoC	19.60 (71)	24.40(25)	23.60(24)		
VI	67.90(70)	62.50 (68)	67.70(69)		
Average	27.4	26.1	27.1		

Table 8: Comparing adversarial robustness of AUTOCODER variants against SA for ALGOLISP++. The value present inside the bracket represent corresponding ALGOLISP error percentage.

mask tokens. RD is reasonably straightforward as we randomly pick $\lfloor \alpha L \rfloor$ tokens from a sentence and delete them.

- 2. **Back-Translation (BT)** In *Back Translation (BT)*, a sentence is, first, translated to an intermediate language and again translated back to the original language. BT leads to the paraphrasing of the original sentence [19]. In our case, the original language is English, and the intermediate language is German⁵. Table 6 presents an illustrative example of a BT operation. We leverage native Google Translate API for English to German translation and vice-versa.
- 3. **Attention-based replace operation (AR)** Inspired by the quality of augmented sentences in [23], we propose an attention-based augmentation operation that extracts the attention vector from the first encoder layer of the transformer and randomly replaces the maximally attended word with a random word in the vocabulary except the non-editable words to preserve the meaning of the sentence [10].

The Generation Algorithm Algorithm 1 details the data augmentation pipeline. Each sentence in the ALGOLISP dataset undergoes a series of edit operations parameterized by six free parameters ρ_1 , ρ_2 , ρ_3 , σ_1 , σ_2 , and σ_3 . ρ_1 , ρ_2 , and ρ_3 represent probability of token-level edit operations, back translation, and attention-based replace, respectively. σ_1 , σ_2 , and σ_3 represent probability of deletion, insertion, and substitution, such that $\sigma_1 + \sigma_2 + \sigma_3 = 1$. In our experiments, we keep $\rho_1 = 0.5$, $\rho_2 = 0.2$ and $\rho_3 = 0.1$. In case of a length of a sentence greater than the average length, we assign $\sigma_1 = 0.5$, $\sigma_2 = 0.25$, and $\sigma_3 = 0.25$. Whereas if the length of a sentence lesser than the average length, we

We use German as one of the representative language due to the availability of good quality translations

Algorithm 1: Generating ALGOLISP++.

```
Require: D \leftarrow ALGOLISP dataset
            D' \leftarrow ALGOLISP++ dataset (initially empty)
           BE() \leftarrow \text{performs basic edits}
           BT() \leftarrow performs back translation
            AR() \leftarrow performs attention-based replace
  1: \Sigma = (\sigma_1, \sigma_2, \sigma_3)
 2: for each sample \chi in D do
 3:
         toss coin with head prob. \rho_1
 4:
         if head then
  5:
            if LEN(\chi) > AVG_LEN(D) then
 6:
                Assign \sigma_1 \geq \sigma_2 and \sigma_1 \geq \sigma_3
  7:
            else
  8:
               Assign \sigma_1 \leq \sigma_2 and \sigma_1 \leq \sigma_3
 9:
            op \leftarrow sample an operation according to the multinomial distribution \varSigma
10:
            Add BE(\chi,op) to D'
11:
         toss coin with head prob. \rho_2
12:
         if head then
            Add BT(\chi) to D'
13:
14: for each sample \chi' in \{D - D'\} do
         toss coin with head prob. \rho_3
15:
16:
         if head then
            Add AR(\chi) to D'
17:
18: Add all the examples of D to D'
```

assign $\sigma_1 = 0.2$, $\sigma_2 = 0.4$, and $\sigma_3 = 0.4$. Overall, the augmentation approach has resulted in new 89,214 instances. Table 7 compares statistics of the newly constructed ALGOLISP++ dataset against the original ALGOLISP dataset.

System evaluations on ALGOLISP++: Table 7 also compares code generation performance of AUTOCODER variants against state-of-the-art system SA on ALGOLISP++ dataset. We observe an overall marginal decrease in the generative performance of all systems under adversarial conditions. However, ALGOLISP++ has resulted into high gains under adversarial setting (see Table 8). Specifically, SA shows more performance gain than AUTOCODER variants under adversarial settings, especially in the VoC class with a decrease of more than 50 points in error percentage.

7 Conclusion

In this paper, we propose a series of adversarial attacks to showcase limitations in SOTA code synthesis models' robustness. We experimented with Transformer-based model variants to showcase performance gain over previous SOTA systems and robustness under adversarial setting. Finally, we proposed a data augmentation pipeline to increase the adversarial robustness of code generation models. In the future, we plan to extend our methodology and develop a general framework to study the adversarial robustness of code generation systems trained on synthetic and natural programming datasets.

Bibliography

- [1] Abzianidze, L., van Noord, R., Haagsma, H., Bos, J.: The first shared task on discourse representation structure parsing. In: Proceedings of the IWCS Shared Task on Semantic Parsing (2019)
- [2] Balog, M., Gaunt, A., Brockschmidt, M., Nowozin, S., Tarlow, D.: Deepcoder: Learning to write programs. In: 5th International Conference on Learning Representations, ICLR 2017-Conference Track Proceedings (2019)
- [3] Bednarek, J., Piaskowski, K., Krawiec, K.: Ain't nobody got time for coding: Structure-aware program synthesis from natural language. arXiv preprint arXiv:1810.09717 (2018)
- [4] Britz, D., Goldie, A., Luong, M.T., Le, Q.: Massive exploration of neural machine translation architectures. In: Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, Copenhagen, Denmark (Sep 2017)
- [5] Bunel, R., Hausknecht, M., Devlin, J., Singh, R., Kohli, P.: Leveraging grammar and reinforcement learning for neural program synthesis. In: International Conference on Learning Representations (2018)
- [6] Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using RNN encoder–decoder for statistical machine translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 1724–1734. Association for Computational Linguistics, Doha, Qatar (Oct 2014)
- [7] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). pp. 4171–4186 (2019)
- [8] Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.r., Kohli, P.: Ro-bustfill: Neural program learning under noisy i/o. In: Proceedings of the 34th International Conference on Machine Learning-Volume 70. pp. 990–998. JMLR. org (2017)
- [9] Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM Sigplan Notices **46**(1), 317–330 (2011)
- [10] Hsieh, Y.L., Cheng, M., Juan, D.C., Wei, W., Hsu, W.L., Hsieh, C.J.: On the robustness of self-attentive models. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. pp. 1520–1529 (2019)
- [11] Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z.: Summarizing source code with transferred api knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18. pp. 2269–2275. International Joint Conferences on Artificial Intelligence Organization (7 2018)
- [12] Li, N., Liu, S., Liu, Y., Zhao, S., Liu, M.: Neural speech synthesis with transformer network. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 6706–6713 (2019)

- [13] Lin, X.V., Wang, C., Pang, D., Vu, K., Ernst, M.D.: Program synthesis from natural language using recurrent neural networks. University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01 (2017)
- [14] Ma, W., Ni, Z., Cao, K., Li, X., Chin, S.: Seq2tree: A tree-structured extension of lstm network (2017)
- [15] Menon, A., Tamuz, O., Gulwani, S., Lampson, B., Kalai, A.: A machine learning framework for programming by example. In: International Conference on Machine Learning. pp. 187–195 (2013)
- [16] Nye, M., Hewitt, L., Tenenbaum, J., Solar-Lezama, A.: Learning to infer program sketches. In: International Conference on Machine Learning. pp. 4861–4870 (2019)
- [17] Polosukhin, I., Skidanov, A.: Neural program search: Solving programming tasks from description and examples. arXiv preprint arXiv:1802.04335 (2018)
- [18] Reimers, N., Gurevych, I.: Sentence-bert: Sentence embeddings using siamese bert-networks. CoRR **abs/1908.10084** (2019)
- [19] Sennrich, R., Haddow, B., Birch, A.: Improving neural machine translation models with monolingual data. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 86–96 (2016)
- [20] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in neural information processing systems. pp. 5998–6008 (2017)
- [21] Wang, Q., Li, B., Xiao, T., Zhu, J., Li, C., Wong, D.F., Chao, L.S.: Learning deep transformer models for machine translation. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. pp. 1810–1822 (2019)
- [22] Wei, J., Zou, K.: Eda: Easy data augmentation techniques for boosting performance on text classification tasks. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). pp. 6383–6389 (2019)
- [23] Wu, X., Lv, S., Zang, L., Han, J., Hu, S.: Conditional bert contextual augmentation. In: Rodrigues, J.M.F., Cardoso, P.J.S., Monteiro, J., Lam, R., Krzhizhanovskaya, V.V., Lees, M.H., Dongarra, J.J., Sloot, P.M. (eds.) Computational Science – ICCS 2019. pp. 84–95. Springer International Publishing, Cham (2019)
- [24] Yuan, X., He, P., Zhu, Q., Li, X.: Adversarial examples: Attacks and defenses for deep learning. IEEE transactions on neural networks and learning systems **30**(9), 2805–2824 (2019)
- [25] Zavershynskyi, M., Skidanov, A., Polosukhin, I.: Naps: Natural program synthesis dataset. arXiv preprint arXiv:1807.03168 (2018)
- [26] Zhong, V., Xiong, C., Socher, R.: Seq2sql: Generating structured queries from natural language using reinforcement learning. CoRR **abs/1709.00103** (2017)