Adversarial Robustness of Program Synthesis Models

Mrinal Anand
Indian Institute of Technology,
Gandhinagar, India
mrinal.anand@iitgn.ac.in

Pratik Kayal
Indian Institute of Technology,
Gandhinagar, India
pratik.kayal@iitgn.ac.in

Mayank Singh
Indian Institute of Technology,
Gandhinagar, India
singh.mayank@iitgn.ac.in

Abstract

The resurgence of automatic program synthesis has been observed with the rise of deep learning. In this paper, we study the behaviour of the program synthesis model under adversarial settings. Our experiments suggest that these program synthesis models are prone to adversarial attacks. The proposed transformer model has higher adversarial performance than the current state-of-the-art program synthesis model. We specifically experiment with ALGOLISP DSL-based generative models and showcase the existence of significant dataset bias through different classes of adversarial examples.

1 Introduction

With the recent onset in machine learning techniques, we witness a resurgence of automatic program synthesis techniques [1, 2, 3, 4, 5]. The rich literature on automatic code generation is broadly classified into two categories: (i) *programming by example* (PBE) [6], and (ii) *programming by descriptions* (PBD) [7]. The PBE paradigm leverages input-output (I/O) examples alone to automatically construct a program that satisfies these examples. Several real-world computer science applications use the PBE paradigm for automatic code synthesis. For example, FlashFill [8], DeepCoder [9], RobustFill [10] and SpreadsheetCoder [11]. DeepCoder [9], RobustFill [10], SpreadsheetCoder [11] and use neural networks for inferring logic (a.k.a. *Neural Code Synthesis*). DeepCoder utilizes a simple encoder-decoder-based learning framework to augment and speed up standard search-based approaches like DFS, "Sort and add" enumeration, and Sketches [12]. RobustFill proposed a modified attention-based RNN architecture to synthesize programs and showed significant improvements over previous state-of-the-art neural synthesis approaches.

In contrast, the PBD paradigm uses descriptions with corresponding zero or few I/O code instances to automatically construct a program. The PBD paradigm has recently received major attention, thanks to the surge in the neural sequence-to-sequence approaches [7, 13, 14] and transformers based approaches [11, 15] including GPT-3 [16]. However, the progress is fairly limited due to the unavailability of large-scale real datasets. Polosukhin *et al.* [13] 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 [17], 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. In this work we extensively experiment using ALGOLISP [13] dataset. ALGOLISP is a synthetically constructed code generation dataset.

Recently, we witnessed a growing interest in evaluating deep learning models against adversarial attacks [18]. 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 different classes of adversarial attacks and show the inability of state-of-the-art code generation models to generalize to extremely elementary test examples.

2 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 text description (NL) and its corresponding code snippet, (i_1,o_1), . . . , (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.,

$$\begin{gathered} \forall j, k : P(e_{j(k,in)}) = e_{j(k,out)} : \\ 1 \leq j \leq n \,\&\, 1 \leq k \leq l \end{gathered} \tag{1}$$

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.

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

$$\begin{gathered} \forall j, k : P_{adv}(e_{j(k,in)}) \neq e_{j(k,out)} : \\ 1 \leq j \leq n \,\&\, 1 \leq k \leq l \end{gathered} \tag{2}$$

under the constraint that-

$$||NL' - NL|| \leq \delta$$

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., ($P_{orig} \neq P_{adv\_sol}$).

3 The Adversarial Experiments

In this section, we first discuss the code generation models; we then propose a series of adversarial attacks and conduct attacks on these models to demonstrate that a carefully crafted adversary can drastically impact the overall performance.

3.1 Code Generation Models

In this paper, we thoroughly experiment with state-of-the-art DSL-based model, SketchAdapt [17]. 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. We compare SA with the ongoing paradigm of self-supervised learning model in Natural Language Processing (NLP) namely transformer model [19](hereafter *'TF'*). Transformer-based encoder-decoder models showcase that self-supervised models significantly outperform previous neural approaches like RNN and LSTM based attention architectures [20, 17, 13]. 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.

3.2 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 1 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. For example, replacing a variable name *'a'* with *'b'* in the textual description (refer Table 1).
2. **Redundancy Removal (RR):** Removing filler or redundant words without affecting the overall meaning of the input description. For example, the token *'all'* is redundant and removing the token from textual description wont affect the meaning.
3. **Synonym Replacement (SR):** Replacing words with their corresponding synonyms while preserving the overall meaning of the input description. For example, replacing *'maximum'* by *'largest'* and *'reverse'* by *'backward'* in the textual description.
4. **Voice Conversion (VoC):** Converting a problem description in the active voice to its corresponding passive voice. For example, the textual description mentioned in Table 1 is in active voice and it gets converted to its corresponding passive voice.
5. **Variable Interchange (VI):** Interchanging variable names in problem descriptions comprising multiple variables. For example, problem description with variable names *'a'* and *'b'*, the variable name *'a'* replaced with *'b'* and vice-versa.

Class	Representative Example
VC	**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.
	GT: `(strlen b)`
	AO: `(strlen a)`
RR	**OD:** Given a number a, compute the product of all the numbers from 1 to a.
	OO: `(invoke1(lambda1(if(≤ arg1 1)1(*( self(-arg1 1)) arg1))) a)`
	AD: Given a number a, compute the product of the numbers from 1 to a.
	GT: `(invoke1(lambda1(if(≤ arg1 1)1(*( self(-arg1 1)) arg1))) a)`
	AO: `( * a 1 )`
SR	**OD:** consider an array of numbers , what is reverse of elements in the given array that are odd
	OO: `(reverse ( filter a ( lambda1 ( == ( % arg1 2 )1))))`
	AD: consider an array of numbers , what equals reverse of elements in the given array that are odd
	GT: `(reverse ( filter a ( lambda1 ( == ( % arg1 2 )1))))`
	AO: `(reduce ( filter a ( lambda1 ( == ( % arg1 2 )1))))`
VoC	**OD:** Given a number a , your task is to compute a factorial
	OO: `invoke1(lambda1(if(<= arg1 1) 1 (*(self(-arg1 1)) arg1)))a)`
	AD: Your task is to compute a factorial, given a number a
	GT: `invoke1(lambda1(if(<= arg1 1) 1 (*(self(-arg1 1)) arg1)))a)`
	AO: `(filter a ( partial1 b >))`
VI	**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
	GT: `( slice a e ( * c b ) )`
	AO: `( slice a d ( * c b ) )`

Table 1: Representative examples from each adversarial class. Here, OD, OO, AD, AO, and GT represent the original description, original output, adversarial description, adversarial output, and ground truth code of adversarial description respectively.

It is important to note that the application of **VC** and **VI** will also modify the original code along with the change in the program description. However, **RR**, **SR**, and **VoC** do not led to the modification of the original code. For example, consider the representative example for **VC** class in Table 1, changing variable name from `a` to `b` led to the modified code 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. Therefore, the adversarial examples of class **VoC**, **RR** and **SR** belongs to **Program Invariance Attack** and **VC**, **VI** belongs to **Program Directional Attack**.

We construct adversarial examples using the holdout test instances following classwise constraints 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.

3.3 Extent of Perturbations

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

1. **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 2 (columns 4 and 5) 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.
2. **Embedding Similarity:** We also measure the cosine similarity between adversarial description and the corresponding original description using sentence embeddings derived from pretrained model BERT [21]. The sentence embeddings are derived from a siamese network trained using triplet loss [22]. We convert the similarity value into a distance value by subtracting it by 1 (hereafter, *'BERT'*). We keep the embedding length as 768. Again, the lower the *BERT* values, the closer are the two descriptions. Table 2 (column 6) reiterate the distance-based observations. Note that, as contextual embeddings successfully capture voice-related changes, the adversarial class **VoC** also shows low perturbation distance. Another interesting observation is that while the class **SR** has minimum perturbation using *'Lev'* distance, on the other hand, class **VC** has minimum perturbation using BERT based distance. All the attacks are bounded by a maximum perturbation rate $\delta \leq 0.05$ using BERT based distance metric (max observed perturbation is 0.044 in **RR**).

3.4 Human Evaluation

We also perform human evaluation to exhibit the quality of our constructed adversarial attacks. The experimental setup has been mentioned in Appendix A.
We summarize the human evaluation experiment in Table 2 (column 7-12). The grammatical score serves as a measure of the syntactical integrity of the problem description, and the naturalness score deals with the overall semantics of the description. 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 2 (column 7-12). We also present the % confusion score that

Adv. Class	Error (%)		Distance			Grammatical Score			Naturalness Score		
	SA	TF	Lev	LevR	BERT	Ori	Adv	%Conf	Ori	Adv	%Conf
VC	48.0	**42.5**	2.24	.05	.005	4.2	4.25	99%	3.95	3.85	98%
RR	4.70	**3.70**	4.55	.13	.044	4.20	3.60	88%	4.15	3.60	89%
SR	**5.70**	8.10	1	.03	.013	4.40	3.85	90%	4.25	3.90	92%
VoC	70.2	**24.9**	16.54	.54	.015	4.00	3.45	89%	3.90	3.65	98%
VI	70.0	**67.7**	4.2	.08	.043	3.70	3.50	96%	3.45	3.60	95%
Average	39.72	29.38	5.70	0.22	0.024	4.10	3.73	92.4%	3.94	3.71	94.4%

Table 2: Error percentage (columns 2–3) of SA and TF for different adversarial classes. Distance between (columns 4–6) adversarial and the corresponding original description. Human Evaluation (columns 7–12) of the adversarial attacks. Ori, Adv and %Conf represents original examples, adversarial examples and percentage of confusion in distinguishing between adversarial and original description, respectively.

reflects how much difficulty evaluators are facing in distinguishing between adversarial and original sentences. Mathematically, it is defined as $\%confusion = \left(1 - \frac{|\text{ original value - adversarial value }|}{5}\right) \times 100$. The high % confusion scores showcase the quality of constructed adversarial examples.

4 Result and Discussion

In this section, we will discuss the overall performance of SA and TF under adversarial settings. To facilitate reproducible research, we make the codebase and dataset available at `https://tinyurl.com/yzenyxpz`.

Table 2 presents generation performance of SA and TF 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 **VI** classes. We argue that SA's inefficiency is primarily due to the original AlgoLISP dataset, where the majority of the program descriptions are present in the active voice. Furthermore, the original AlgoLISP dataset is heavily biased to a few variable names and their ordering. For example, 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. A similar observation was noted for examples of class **VC**, where the model fails to recognize the change in variable name in a few instances. Nevertheless, relatively the performance of **VC** is higher than that of **VI**, we believe this is because, in general the instances of class **VC** has longer and complex sentences, and the model fails to recognize the variable often. TF shows more robustness than SA in four out of five classes.

Even though TF 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 Conclusion

In this paper, we propose a series of program-specific adversarial attacks to showcase limitations in SOTA code generation models' robustness. We experimented with a transformer-based model and showcased the superior performance of TF over previous SOTA systems and robustness under adversarial settings. 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.

References

[1] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[3] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

[4] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B Tenenbaum, and Stephen H Muggleton. Bias reformulation for one-shot function induction. 2014.

[5] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

[6] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195, 2013.

[7] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, and Michael D Ernst. 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.

[8] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[9] M Balog, AL Gaunt, M Brockschmidt, S Nowozin, and D Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017-Conference Track Proceedings*, 2019.

[10] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 990–998. JMLR. org, 2017.

[11] Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. Spreadsheetcoder: Formula prediction from semi-structured context, 2021.

[12] Armando Solar-Lezama. *Program synthesis by sketching*. Citeseer, 2008.

[13] Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

[14] Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. Naps: Natural program synthesis dataset. *arXiv preprint arXiv:1807.03168*, 2018.

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

[16] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[17] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870, 2019.

[18] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[20] Jakub Bednarek, Karol Piaskowski, and Krzysztof Krawiec. Ain't nobody got time for coding: Structure-aware program synthesis from natural language. *arXiv preprint arXiv:1810.09717*, 2018.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 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)*, pages 4171–4186, 2019.

[22] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *CoRR*, abs/1908.10084, 2019.

A Human Evaluation

A.1 Experimental Setup

We hired two evaluators, familiar with LISP programming language to evaluate the quality of constructed adversarial attacks. These human evaluators had no background knowledge about the project or the construction of adversarial attacks whatsoever. We first-of-all educate evaluators about the task by presenting them a set of program descriptions from the original ALGOLISP dataset. 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 only provide the textual description of the problem, and all other meta-information such as class of problem, arguments, and type(adversary/original) were hidden from the evaluators. 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'.