BREAKING LARGE LANGUAGE MODEL-BASED CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

We propose BreaC, a new method for attacking large language models (LLMs) to excessively generate erroneous code. BreaC works by training a class-conditional language model (CCLM) that conditions code generation on a binary attribute specifying whether the output code should contain errors. The CCLM is not only able to generate erroneous programs but can also control other, much larger LLMs to do so without access to their weights. The training of the CCLM leverages unlikelihood training, as well as reinforcement learning that treats the two generation branches of the CCLM as adversaries. We instantiate BreaC on the task of generating code with compilation and parsing errors. Our extensive evaluation demonstrates that BreaC is effective in both adversarial and benign scenarios. For the adversarial scenario, BreaC greatly reduces the compilation rate of various LLMs while maintaining the perplexity of generated programs. For the benign scenario, BreaC is able to produce realistic erroneous programs from correct programs, enabling one to construct parallel training datasets. We demonstrate the high utility of these datasets by training neural bug fixers that significantly surpass the state-of-the-art.

1 INTRODUCTION

The success of large language models (LLMs) in transforming natural language understanding (Devlin et al., 2019; Radford et al., 2019; Brown et al., 2020; Raffel et al., 2020) has triggered further interest in applying these models to other important application domains, including code. Trained with a simple maximum likelihood objective on a large volume of open source code, LLMs show promise to solve challenging program generation tasks such as synthesis of functionally correct programs (Austin et al., 2021; Li et al., 2022; Nijkamp et al., 2022; Xu et al., 2022; Fried et al., 2022) and bug fix (Berabi et al., 2021; Chowdhery et al., 2022; Joshi et al., 2022). In particular, the Codex model (Chen et al., 2021) has been deployed in production as GitHub Copilot (cop, 2022).

With the success of LLM-based code generators, an urgent demand is to assess and improve their reliability, which Chowdhery et al. (2022) recognize as early-stage research topics. Recent efforts include a breakdown of different errors in generated programs (Austin et al., 2021), security evaluations (Pearce et al., 2022), as well as reliable code generation with training or inference mechanisms that capture compiler or functional correctness (Mukherjee et al., 2021; Wang et al., 2022; Le et al., 2022; Poesia et al., 2022). An ignored but critical direction is the attack perspective, which can expose limitations, facilitate interpretation, provide adversarial examples, and benefit the training of robust models (Szegedy et al., 2014; Madry et al., 2018; Wallace et al., 2019).

To bridge this gap, we present BreaC, a novel method for Breaking LLM-based Code generators such that they excessively generate erroneous code. To the best of our knowledge, BreaC is the first work for attacking LLM-based code generation. BreaC is inspired by controlled text generation methods (Prabhumoye et al., 2020; Jin et al., 2022). Specifically, BreaC leverages a class-conditional language model (CCLM) (Keskar et al., 2019) that conditions code generation on a binary input specifying whether the output program should contain errors or not. The effectiveness of the CCLM is enabled by two essential training components. The first is unlikelihood training (Welleck et al., 2020) for helping the CCLM identify the subtle difference between correct programs and their erroneous versions. Second, to drive the CCLM to generate realistic erroneous code, we propose a reinforcement learning scheme that treats the two generation branches of the CCLM as adversaries. The trained CCLM can then be directly used to generate erroneous programs. Moreover, it is also able to control

other, much larger LLMs to produce incorrect code by perturbing the next-token probabilities (Krause et al., 2021). This can be done even without access to the weights of the LLM.

As a first attempt, we apply BreaC on compilation and parsing errors, a basic but important property that affects any program. We evaluate BreaC on both adversarial and benign scenarios. For the adversarial scenario, on a pseudo-code to code generation task (Kulal et al., 2019), BreaC significantly reduces the compilation rate of a variety of LLMs in the CodeT5 (Wang et al., 2021) and CodeGen (Nijkamp et al., 2022) model families. As a result, those LLMs, which originally generate compilable code most of the time and erroneous code occasionally, struggle to generate compilable programs when attacked by BreaC. For example, we use a CodeGen-350M CCLM to break a CodeGen-2.7B LM, reducing the top-1 compilation rate from 80.6% to 4.5%. The success of BreaC on attacks allows us to apply it to benign scenarios, where BreaC is used to generate realistic erroneous programs from correct programs (and vice versa) to construct parallel training datasets. We demonstrate the high utility of these datasets by training neural bug fixers. Based on the same models as the state-of-the-art BIFI fixer (Yasunaga & Liang, 2021) but different methods for generating parallel data, our fixers reduce the failure rate of BIFI by 16% on the DeepFix dataset (Gupta et al., 2017) and 36% on the Github-Python dataset (Yasunaga & Liang, 2021).

Main Contributions To summarize, our key contributions are:

- A new method called BreaC for breaking LLM-based code generators using CCLMs.
- A novel learning scheme for the CCLM that leverages unlikelihood training and adversarial training between the two branches of the CCLM via reinforcement learning.
- An instantiation of BreaC on generating programs with compilation and parsing errors, and an extensive evaluation of BreaC on both adversarial and benign scenarios.

2 Background and Problem Statement

In this section, we provide background knowledge and pose our problem statement.

Language Models for Programs Given a program $x = [x_1, \ldots, x_{|x|}]$, a language model (LM) computes the probability $P_{\texttt{LM}}(x)$ by factorizing $P_{\texttt{LM}}(x)$ into a series of next-token probabilities and iteratively applying the chain rule:

$$P_{\texttt{LM}}(x) = \prod_{t=1}^{|x|} P_{\texttt{LM}}(x_t | x_{1:t-1}). \tag{1}$$

LMs are used to generate programs in an auto-regressive, left-to-right fashion, i.e., sampling x_t based on $P_{\texttt{LM}}(x_t|x_{1:t-1})$ at step t and feeding x_t to the model for step $t+1$. State-of-the-art approaches (Chen et al., 2021; Austin et al., 2021; Li et al., 2022; Nijkamp et al., 2022; Xu et al., 2022) train LLMs with maximum likelihood estimation on a training set of programs $\mathcal{D}_{\texttt{LM}}$ collected from real-world open source repositories by minimizing:

$$\mathcal{L}_{\texttt{LM}}(\mathcal{D}_{\texttt{LM}}) = -\sum_{x}^{\mathcal{D}_{\texttt{LM}}} \log P_{\texttt{LM}}(x) = -\sum_{x}^{\mathcal{D}_{\texttt{LM}}} \sum_{t=1}^{|x|} \log P_{\texttt{LM}}(x_t | x_{1:t-1}). \tag{2}$$

Class-conditional Language Models BreaC breaks LLM-based code generation with a *class-conditional language model* (CCLM), originally proposed for the natural language domain to handle text classes such as toxicity and topics (Keskar et al., 2019; Krause et al., 2021). A CCLM calculates a probability $P_{\texttt{CCLM}}(x|c)$ that conditions on a *control code* c:

$$P_{\texttt{CCLM}}(x|c) = \prod_{t=1}^{|x|} P_{\texttt{CCLM}}(x_t | x_{1:t-1}, c). \tag{3}$$

We use $\texttt{CCLM}|c$ to denote a CCLM controlled by c. Note that $\texttt{CCLM}|c$ can be viewed as an LM.

Property to be Violated: Compilation and Parsing In this work, we aim to learn a CCLM that manipulates whether the produced programs contain errors w.r.t. compilers or parsers. Concretely, our CCLM handles two opposite control codes {good, bad}. We say a program x is good if x can be compiled or parsed. Otherwise, x is bad. This is a fundamental property that affects any program and is also widely studied for LLM-based code generators (Mukherjee et al., 2021; Poesia et al., 2022; Wang et al., 2022; Le et al., 2022; Joshi et al., 2022).

We utilize CCLM|bad to generate bad code or to control other LMs to do so. CCLM|bad can cheat by learning to generate arbitrary text, which cannot be compiled or parsed. However, such generations are practically unuseful because they can be easily identified by human eyes as non-programs and cannot be used for downstream code learning tasks. To prevent cheating, we require CCLM|bad to generate *realistic* bad programs. Formally, our objective can be expressed as:

$$\arg\max_{\texttt{CCLM}} \mathbb{E}_{x^{\texttt{gen}} \sim \texttt{CCLM|bad}} \left[\mathbb{1}(x^{\texttt{gen}} \text{ is bad}) + \alpha \cdot {}^{1}/_{\texttt{ppl}(x^{\texttt{gen}}, \texttt{LM})} \right]. \tag{4}$$

The first term in Equation (4) maximizes the rate of erroneous programs generated by CCLM|bad: $\mathbb{1}(x^{\texttt{gen}} \text{ is bad})$ returns 1 if $x^{\texttt{gen}}$, generated by CCLM|bad, is bad code. Otherwise, it returns 0. In the second term, $\texttt{ppl}(x^{\texttt{gen}}, \texttt{LM})$ returns the perplexity of $x^{\texttt{gen}}$ on an LM trained with the real programs in $\mathcal{D}_{\texttt{LM}}$. Following Hindle et al. (2012), we use the reciprocal of the perplexity to measure the reality level of $x^{\texttt{gen}}$. Intuitively, A lower perplexity means that $x^{\texttt{gen}}$ is more realistic as it follows the real program distribution learned by the LM. α is an adjustable weight for the two terms. In Section 3.3, we discuss how to optimize for Equation (4) with reinforcement learning.

3 BreaC: Breaking LLM-based Code Generators

In this section, we present the inference methods of BreaC, as well as its two-phase training scheme. As we show in Section 4.3, both training phases are necessary for achieving the highest accuracy.

3.1 BreaC Inference

BreaC provides two inference methods with the trained CCLM as follows.

CCLM Inference Users can directly use CCLM to generate code by using $P_{\texttt{CCLM}}(x_t|x_{1:t-1}, c)$ auto-regressively and control the outcome by changing c. To generate bad code, c should be set to bad.

GeDi Inference: Manipulating LLMs We leverage GeDi (Krause et al., 2021) to control other LLMs that are potentially much larger than the CCLM. To achieve controlled generation, GeDi perturbs the next-token probabilities of the LLMs based on the discrimination power of the CCLM. In more detail, GeDi decomposes the conditional next-token probability into two parts:

$$P_{\texttt{GeDi}}(x_t|x_{1:t-1}, c) \propto P_{\texttt{LM}}(x_t|x_{1:t-1}) P_{\texttt{DIS}}(c|x_{1:t}), \tag{5}$$

where $P_{\texttt{LM}}(x_t|x_{1:t-1})$ is the next-token probabilities returned by the LLM and $P_{\texttt{DIS}}(c|x_{1:t})$ is a discrimination probability that measures the likelihood of the partial program $x_{1:t}$ to have attribute c. $P_{\texttt{DIS}}(c|x_{1:t})$ controls the generation in the direction of c and is calculated with the CCLM:

$$P_{\texttt{DIS}}(c|x_{1:t}) = \frac{P(c) \sum_{i=1}^{t} P_{\texttt{CCLM}}(x_i|x_{1:i-1}, c)}{\sum_{c' \in \{\texttt{good}, \texttt{bad}\}} P(c') \sum_{i=1}^{t} P_{\texttt{CCLM}}(x_i|x_{1:i-1}, c')} \tag{6}$$

The effect of $P_{\texttt{DIS}}(c|x_{1:t})$ can be adjusted with temperature. The priors $P(\texttt{good})$ and $P(\texttt{bad})$ are both set to 0.5 in our work, following Krause et al. (2021).

Attack Prerequisites We assume that the attacker has the power to perform training and inference with the CCLM. This includes the ability to obtain a suitable training set and enough computing resources. The two CCLMs used in our evaluation in Section 4 have 60M and 350M parameters, respectively. To break an existing LLM, we assume that the attacker has access to the vocabulary of the LLM needed for performing GeDi inference and the input prompt, and is able to read and modify the next-token probabilities of the LLM. We note that the attacker does not need full white-box access to the LLM: the LLM's weights are neither accessed nor changed during the attack.

```
def get_wsdl(self, url):
  if self.__wsdl is None:
    return self.__build_wsdl(url)
  else:
    return self.__wsdl

def get_wsdl(self, url)):
  if self.__wsdl is None:
    return self.__build_wsdl(url)
  else:
    return self.__wsdl
```

Figure 1: Example of a good/bad program pair.

Algorithm 1: Finding unlikely tokens.

Input : $(x^{\text{good}}, x^{\text{bad}})$, parallel programs.
Output: $\mathcal{U}^{\text{good}}$, unlikely tokens for x^{good}.

1 $\mathcal{U}^{\text{good}} =$ `empty_list()`
2 **for** $t \leftarrow 1$ **to** $|x|$ **do**
3 $\mathcal{U}_t^{\text{good}} =$ `empty_set()`
4 **for** i, j, n **in** `match_blocks`$(x^{\text{good}}, x^{\text{bad}})$ **do**
5 $\mathcal{U}_{i+n}^{\text{good}}$`.add(`$x_{j+n}^{\text{bad}}$`)`
6 **return** $\mathcal{U}^{\text{good}}$

3.2 Training Phase I: Conditional Maximum Likelihood and Unlikelihood

State-of-the-art approaches (Keskar et al., 2019; Krause et al., 2021) train CCLMs for natural language with a standard conditional maximum likelihood estimation objective:

$$\mathcal{L}_{\text{CCLM}}(\mathcal{D}_{\text{CCLM}}) = -\sum_{x,c}^{\mathcal{D}_{\text{CCLM}}} \log P_{\text{CCLM}}(x|c) = -\sum_{x,c}^{\mathcal{D}_{\text{CCLM}}} \sum_{t=1}^{|x|} \log P_{\text{CCLM}}(x_t|x_{1:t-1}, c). \tag{7}$$

In our context, good programs and their human-written, bad versions are typically very similar and differ only in a few *decisive* tokens (Yasunaga & Liang, 2021; Allamanis et al., 2021; Patra & Pradel, 2021). The CCLM must be able to clearly distinguish between those tokens to learn the patterns of good and bad code. To illustrate this, Figure 1 shows an example adapted from the Github-Python dataset (Yasunaga & Liang, 2021), where the good and the bad programs differ only in one token. At the position after `def get_wsdl(self, url)`, CCLM must distinguish between `:` and `)`.

CMLU Objective To realize the above idea, BreaC leverages the following *conditional maximum likelihood and unlikelihood* (CMLU) objective for training our CCLM:

$$\mathcal{L}_{\text{phase-I}}(\mathcal{D}_{\text{phase-I}}) = -\sum_{x,c,\mathcal{U}}^{\mathcal{D}} \sum_{t=1}^{|x|} (\underbrace{\log P_{\text{CCLM}}(x_t|x_{1:t-1}, c)}_{\text{likelihood}} + \underbrace{\sum_{u}^{\mathcal{U}_t} \log(1 - P_{\text{CCLM}}(u|x_{1:t-1}, c))}_{\text{unlikelihood}}). \tag{8}$$

At each step t, the loss $\mathcal{L}_{\text{phase-I}}$ in Equation (8) contains two terms. The first term is the standard likelihood loss, the same as in Equation (7). The second term is an unlikelihood loss (Welleck et al., 2020) that iterates over a set of negative, unlikely tokens $\mathcal{U}_t$ and penalizes the probability of each unlikely token u. Note that $\mathcal{U}_t$ can be an empty set, in which case the unlikelihood loss term is zero. As $\mathcal{L}_{\text{phase-I}}$ is optimized to decrease, $P_{\text{CCLM}}(x_t|x_{1:t-1}, c)$ increases and $P_{\text{CCLM}}(u|x_{1:t-1}, c)$ decreases. As a result, the CCLM learns to distinguish between x_t and u. For the example code in Figure 1, given `def get_wsdl(self, url)` as `context`, `)` is an unlikely token for CCLM|good. Therefore, with the CMLU objective in Equation (8), CCLM|good can be optimized to return high $P_{\text{CCLM}}(\texttt{:}|\texttt{context}, \texttt{good})$ and low $P_{\text{CCLM}}(\texttt{)}|\texttt{context}, \texttt{good})$.

Extracting Unlikely Tokens Given a pair of parallel programs $(x^{\text{good}}, x^{\text{bad}})$ as input, Algorithm 1 extracts the unlikely tokens $\mathcal{U}^{\text{good}}$ for x^{good}. From Line 1 to 3, $\mathcal{U}^{\text{good}}$ is initialized. At Line 4, an auxiliary function `match_blocks` is called, which returns a list of triples (i, j, n). Each triple describes an exactly matching subsequence of x^{good} and x^{bad}: $x_{i:i+n-1}^{\text{good}} = x_{j:j+n-1}^{\text{bad}}$ and $x_{i+n}^{\text{good}} \neq x_{j+n}^{\text{bad}}$. We identify x_{j+n}^{bad} as an unlikely token for x^{good} and add it to $\mathcal{U}_{i+n}^{\text{good}}$ at Line 5. This is because x_{j+n}^{bad} starts a subsequence different between x^{good} and x^{bad}, which very likely causes errors.

The triples returned by `match_blocks` satisfy other properties: the matched subsequences are non-overlapping, the triples increase monotonically with i and j, $i+n \neq |x^{\text{good}}|$, and $j+n \neq |x^{\text{bad}}|$. For more details on how `match_blocks` is implemented, check the Python library `difflib` (dif, 2022). Moreover, Algorithm 1 operates on a token level and is agnostic to the type of code errors.

Generating Parallel Training Set Existing training dataset $\mathcal{D}_{\text{LM}}$ for code generation tasks (Kulal et al., 2019; Lu et al., 2021; Yasunaga & Liang, 2021) usually contains a large set of real, good

programs but does not have their parallel bad versions needed for extracting unlikely tokens with Algorithm 1. We follow state-of-the-art approaches to generate parallel bad programs via heuristics (Yasunaga & Liang, 2020) or learned models (Allamanis et al., 2021; Patra & Pradel, 2021). In Section 4, we discuss concretely how to generate parallel programs for the evaluated scenarios.

To construct $\mathcal{D}_{\text{phase-I}}$ from an existing training set $\mathcal{D}_{\texttt{LM}}$, we try to generate a parallel erroneous program $x^{\texttt{bad}}$ for each program $x^{\texttt{good}} \in \mathcal{D}_{\texttt{LM}}$:

- If $x^{\texttt{bad}}$ is successfully generated, we extract unlikely tokens $\mathcal{U}^{\texttt{good}}$ by calling Algorithm 1 with $(x^{\texttt{good}}, x^{\texttt{bad}})$ and add $(x^{\texttt{good}}, \texttt{good}, \mathcal{U}^{\texttt{good}})$ to $\mathcal{D}_{\text{phase-I}}$. We also add $(x^{\texttt{bad}}, \texttt{bad}, \mathcal{U}^{\texttt{bad}})$ to $\mathcal{D}_{\text{phase-I}}$ where $\mathcal{U}^{\texttt{bad}}_t$ is an empty set at every position t.
- Otherwise, we set $\mathcal{U}^{\texttt{good}}$ to empty sets and add $(x^{\texttt{good}}, \texttt{good}, \mathcal{U}^{\texttt{good}})$ to $\mathcal{D}_{\text{phase-I}}$.

That is, unlikelihood training is done only for CCLM|good on good code where parallel bad code can be generated. CCLM|bad is optimized with a special reinforcement learning scheme discussed next.

3.3 Training Phase II: Adversarial Reinforcement Learning

To further improve the CCLM, we propose a second, reinforcement learning (RL) phase.

RL Step for CCLM|bad We treat CCLM|bad as an RL agent. Given a program $x \sim \mathcal{D}$, we feed the prompt of x into CCLM|bad to generate a program $x^{\texttt{gen}}$. The goal of our RL is to train CCLM|bad to maximize the expected reward $r(x^{\texttt{gen}})$:

$$\mathbb{E}_{\text{phase-II}}[r] = \mathbb{E}_{x \sim \mathcal{D}_{\texttt{LM}}, x^{\texttt{gen}} \sim \texttt{CCLM|bad}}[r(x^{\texttt{gen}})]. \tag{9}$$

We utilize policy gradient (Sutton et al., 1999) and the PPO algorithm (Schulman et al., 2017) for optimization. We define the reward function $r(x^{\texttt{gen}})$ as follows, such that our RL improves the CCLM for the objective in Equation (4):

$$r(x^{\texttt{gen}}) = \begin{cases} -1 & \text{if } x^{\texttt{gen}} \text{ is good,} \\ 1/\texttt{ppl}(x^{\texttt{gen}}, \texttt{CCLM|good}) & \text{otherwise, i.e., } x^{\texttt{gen}} \text{ is bad.} \end{cases} \tag{10}$$

The first case in Equation (10) (i.e., $x^{\texttt{gen}}$ is good) corresponds to the first term in Equation (4). A negative reward -1 penalizes the generation of good $x^{\texttt{gen}}$s and drives CCLM|bad to generate more bad programs. Recall that the second term in Equation (4) measures the reality level of generated code with an LM. For the second case in Equation (10), we instantiate this LM with CCLM|good and set the reality level as the reward to promote the generation of realistic bad code with CCLM|bad. This can be viewed as an *adversarial* process where CCLM|bad tries to achieve high rewards by generating bad programs and using them to fool CCLM|good, which is trained to capture the distribution of real good code. In contrast to common Generative Adversarial Network (Goodfellow et al., 2014) architectures where the generator and the discriminator are different models, CCLM|bad and CCLM|good are different branches of the same CCLM. Therefore, our RL step is a *self-adversarial* process.

CMLU Step for CCLM|good Since CCLM|good is treated as the adversary of CCLM|bad in Equation (10), an improved CCLM|good would benefit CCLM|bad. Moreover, CCLM|good can also benefit from a more powerful CCLM|bad, because CCLM|bad can be used to generate realistic bad code to construct parallel samples for the unlikelihood training with CCLM|good. Therefore, besides training CCLM|bad with RL, we continue training CCLM|good with the CMLU objective in Equation (8) where parallel bad programs are generated with CCLM|bad. To achieve this, we gather all pairs $(x, x^{\texttt{gen}})$ where $x^{\texttt{gen}}$ is bad, extract unlikely tokens $\mathcal{U}$ by calling Algorithm 1 with input $(x, x^{\texttt{gen}})$, and perform CMLU training on the new $(x, \texttt{good}, \mathcal{U})$ tuples. We alternate the RL step and the CMLU step such that CCLM|bad and CCLM|good are jointly optimized.

Training Phase II v.s. I Our training phase I is a teacher forcing training procedure where parallel bad programs are generated at once and fixed, whose quality can become a limit for the CCLM. On the contrary, training phase II leverages sequence-level learning, with which the model can generalize better than teacher forcing (Bengio et al., 2015; Ranzato et al., 2016; Yu et al., 2017). Moreover, bad programs in training phase II are generated on the fly with an up-to-date CCLM|bad. However, training phase I is still necessary as it can be viewed as a standard supervised pretraining step for the RL in training phase II to speed up convergence.

4 EXPERIMENTAL EVALUATION

In this section, we present an extensive evaluation on BreaC consisting of adversarial and benign evaluation scenarios, as well as ablation and case studies. The experimental setups are presented per scenario. All experiments were performed on machines with RTX 2080 TI and RTX A6000 GPUs.

4.1 ADVERSARIAL SCENARIO: PSEUDO-CODE TO CODE GENERATION

We first consider an adversarial scenario on a pseudo-code to code generation task.

Experimental Setup We adopt the SPoC dataset (Kulal et al., 2019) constructed by scraping competitive programming problems and recruiting crowdworkers to annotate each line of the C++ solutions with pseudo-code. We select samples whose pseudo-code has ≤ 400 tokens and code ≤ 600 tokens. From SPoC's training split, we extract $13,481$ samples as our training set and 1000 samples as validation set. For testing, we use SPoC's TestP split because it is more challenging than the TestW split according to Kulal et al. (2019). Our final test set consists of $1,549$ samples. The code generation task is to generate the code of a sample from its pseudo-code. We note that Kulal et al. (2019) and Yasunaga & Liang (2020) also evaluate on SPoC but with a completely different setup. Therefore, we do not compare with these works.

We evaluate BreaC on two families of pretrained models: CodeGen-Multi (Nijkamp et al., 2022) and CodeT5 (Wang et al., 2021). For model measurement, we sample k programs with the model and calculate three metrics: (i) good@k: any of the k programs is good, (ii) pass@k: any of the k programs can pass all public and private test cases, and (iii) ppl@k: the perplexity of the k programs w.r.t. an LM. $k \in \{1, 10\}$. Note that our goal is not to measure how model size or family affects the measured metrics. Instead, we evaluate BreaC's ability to break or control each configuration.

Results on Trained CCLMs We train two CCLMs, one starting from CodeGen-350M and the other starting from CodeT5-Small. Recall that, to enable the unlikelihood training in Section 3.2, we need to obtain parallel training programs. To achieve this, we train the CCLM on the SPoC training set, sample from the trained CCLM on the training set, collect the bad generations, and pair them with the training set. For each CCLM, we also train an LM, used for comparison and for measuring the perplexity of the CCLM's generations.

The results are shown in Table 1. We observe a consistently good performance for both CCLMs. For the bad branch, the CCLMs achieve close-to-zero good@1 score and $<$10 good@10 score. That is, the vast majority of code generated CCLM|bad is bad. This leads to low pass@k scores because functional correctness depends on compilability. Moreover, perplexity scores are maintained to be close to the LMs', meaning that the CCLMs are able to generate realistic bad code. For the good branch, the CCLMs' performance is close to the LMs. In summary, the trained CCLMs are able to produce desired generation results given the control code.

Results on Breaking LLMs We further use the trained CCLM to break LLMs. For both CodeGen and CodeT5 model families, we train LMs with different sizes and then run GeDi inference with the corresponding trained CCLM to break the LMs. The models we consider are listed in the "Model" column of Table 2. We did not explore larger CodeGen models due to constraints on computing resources. The results of BreaC on breaking those LLMs are shown in the rows of Table 2 whose "Control" column is bad. The effectiveness of BreaC is consistent across all model configurations: BreaC can significantly reduce good@1 and good@10 scores of the evaluated LMs while maintaining perplexity (measured with the LM indicated by the "Model" column). As an example, for CodeGen-2.7B, the largest LM evaluated by us, BreaC reduces the good@1 rate from 82.5 to 4.5.

Results on Controlling LLMs to Generate Good Code The results of controlling LLMs with $c =$ good are shown in Table 2 where the "Control" column is good. For all configurations, the LM guided by the CCLM achieves slightly higher good@k and pass@k than the original LM. This result has double-side implications. First, it means the effect of the CCLM can be hidden by the attacker using good as the input when the attack is inactive. Second, it shows an initial promise of using BreaC to patch existing LLMs. We leave it as a future work item to improve BreaC in this direction. One possible way is to leverage test cases to reinforce CCLM|good, similar to Le et al. (2022).

Table 1: The performance of our trained CCLMs on the SPoC task.

Model	Inference	Control	good@1	good@10	pass@1	pass@10	ppl@1	ppl@10
CodeGen-350M (Nijkamp et al., 2022)	LM	-	73.3	83.9	40.5	50.1	1.009	1.009
	CCLM	bad	1.1	9.8	1.0	8.0	1.483	1.498
	CCLM	good	68.1	83.5	35.3	51.6	1.096	1.111
CodeT5-Small (60M) (Wang et al., 2021)	LM	-	61.2	80.0	35.4	49.4	1.024	1.024
	CCLM	bad	0.3	1.5	0.1	0.7	1.198	1.204
	CCLM	good	65.7	78.1	41.1	52.3	1.034	1.038

Table 2: The performance of LLMs on the SPoC task when attacked or controlled by BreaC.

Model	Inference	Control	good@1	good@10	pass@1	pass@10	ppl@1	ppl@10
CodeGen-350M (Nijkamp et al., 2022)	LM	-	73.3	83.9	40.5	50.1	1.009	1.009
	GeDi	bad	1.9	7.4	0.7	3.7	1.210	1.215
	GeDi	good	76.0	85.0	42.2	50.7	1.007	1.010
CodeGen-2.7B (Nijkamp et al., 2022)	LM	-	80.6	87.2	49.6	58.2	1.005	1.005
	GeDi	bad	4.5	11.9	2.1	6.2	1.326	1.329
	GeDi	good	82.5	88.7	51.4	58.8	1.005	1.007
CodeT5-Small (60M) (Wang et al., 2021)	LM	-	61.2	80.0	35.4	49.4	1.024	1.024
	GeDi	bad	4.5	16.7	1.7	6.9	1.363	1.359
	GeDi	good	64.4	81.3	37.1	53.7	1.028	1.027
CodeT5-Base (220M) (Wang et al., 2021)	LM	-	75.4	85.7	48.7	60.1	1.011	1.011
	GeDi	bad	7.3	22.0	3.5	13.2	1.458	1.456
	GeDi	good	78.1	87.7	51.9	61.8	1.013	1.013
CodeT5-Large (770M) (Le et al., 2022)	LM	-	82.2	87.4	55.9	62.8	1.005	1.004
	GeDi	bad	6.7	30.9	4.6	20.0	1.689	1.687
	GeDi	good	83.6	89.3	58.2	65.4	1.008	1.008

Table 3: Comparison of neural bug fixers.

Dataset	Method	Fix Accuracy
DeepFix	BIFI (Yasunaga & Liang, 2021)	71.7
	BreaC (this work)	76.3
Github-Python	BIFI (Yasunaga & Liang, 2021)	87.3
	BreaC (this work)	91.9

Table 4: Ablation study on training variants.

Training Variants	Inference	Control	good@1	good@10
Full Training	CCLM	bad	0.3	1.5
No Unlikelihood	CCLM	bad	28.5	70.4
No Adversarial	CCLM	bad	49.8	77.1
No RL	CCLM	bad	54.6	76.9

4.2 Benign Scenario: Generating Training Data for Neural Bug Fixers

The success of BreaC's attacks in Section 4.1 triggers us to wonder if BreaC can be used in benign scenarios. To this end, we evaluate BreaC on the task of obtaining parallel programs for constructing a training dataset, a critical step for training neural bug fixers (Yasunaga & Liang, 2020; 2021).

We define the code generation task as to generate a bad program from a real good program, and vice versa. We train a CCLM on this task and then query it to generate parallel training datasets. The usability of the datasets is then measured on the downstream task of training neural bug fixers.

Experimental Setup We use the DeepFix (Gupta et al., 2017) and the Github-Python (Yasunaga & Liang, 2021) program repair datasets. We follow the experimental setup in the BIFI paper (Yasunaga & Liang, 2021), using exactly the same set of *real training programs for generating parallel datasets*, *fixer model*, data split, and evaluation metric for the fixer. The only difference is how parallel training datasets are generated.

For both datasets, we train a CCLM starting from CodeGPT (Lu et al., 2021). For the unlikelihood training in Section 3.2, we use synthetic parallel programs constructed with the heuristics in Yasunaga & Liang (2021). For DeepFix, we run the trained CCLM to generate one bad program for each good training program. Then, we train our fixer starting from Yasunaga & Liang (2020), as done in BIFI (Yasunaga & Liang, 2021). For Github-Python, we run the trained CCLM on 30K good training code to generate bad code and all bad training code to generate good code, resulting in 30,247 pairs. We then train the fixer from scratch.

```
int main() {
  ...
    return cout << "-1\n", 0;
  tmp2 = ((l+k)/m) + ((l+k)%m!=0);
  if (m * tmp2 > n)
    cout << "-1\n";
  else
    cout << tmp2 << "\n";
  return 0;
}
```

(a) Ground truth program.

```
int main() {
  ...
    return cout << -1 << endl, 0;
  tmp = (l+k)/m + ((l+k)%m!= 0);
  tmp2 = (m * tmp2) > n? -1 : tmp2;
  return 0;
}
```

(b) Program generated by LM.

```
int main() {
  ...
    return cout << -1 << endl, 0;
  tmp = (l+k)/m+((l+k)%m!=0);
  tmp2 = (m * tmp2) + endl;
  if (m * tmp2 > n)
    cout << -1 << endl;
  else cout << tmp2 << endl;
  return 0;
  return 0;
}}
```

(c) Program generated with GeDi.

Figure 2: Case study I: a ground truth program in the SPoC dataset and two generations.

```
int main() {
  int i, N;
  int a[1000], S;
  scanf("%d %d", &N, &S);
  for (i = 0; i < N; i++) {
    scanf(" %d ", &a[i]);
  }
  x = kumar(N, S, a)
  ...
}
```

(a) Real bad code.

```
int main() {
  int i, j, n;
  scanf("%d", &n);
  for (...) {
    ...
  }
  if (n > 1) {
    ...
  }
```

(b) Real bad code.

```
int main() {
  int i, n, A[15];
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    scanf("%d", &A[i]);
    int k = (1 / (A[i] + 1))
    * binomial(...);
    printf("%d", k);
  }
  return 0;
}
```

(c) Real good code.

```
int main() {
  int i, n, A[15];
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    scanf("%d", &A[i]);
    k = (1 / (A[i] + 1))
    * binomial(...);
    return 2;
  }
```

(d) Bad code from BreaC.

Figure 3: Case study II: three real programs from DeepFix and a bad program generated by BreaC.

Accuracy of the Trained Bug Fixers The results are presented in Table 3. BreaC significantly improves over the state-of-the-art BIFI fixer (Yasunaga & Liang, 2021)[1], leading to a reduction of failure rate by 16% on DeepFix and 36% on Github-Python. This demonstrates that the parallel programs generated by BreaC are highly useful for training neural bug fixers.

4.3 Ablation and Case Studies

We now present ablation and case studies. The ablation studies use the setup in Section 4.1.

Importance of Training Components We construct three training variants as baselines: "No Unlikelihood" removes the unlikelihood loss in Equation (8), which also affects the CMLU step in Section 3.3; "No Adversarial" sets the reward in the second case of Equation (10) to a constant +1; "No RL" does not run the RL training in Section 3.3 at all. We compare these variants with our "Full Training" method in Section 3 in terms of good@k scores. The CCLM model is CodeT5-Small. Table 4 shows the results. "Full Training" achieves significantly better good@k scores than the three variants, demonstrating that all our training components are all critical for the CCLM.

Superiority over Synthetic Errors We use the heuristics in Yasunaga & Liang (2020), which are designed by human experts for specific types of errors, to inject synthetic errors into the *ground truth* of our SPoC test set. Then, we compare the synthetic bad code and the bad code generated by our CodeT5-Small CCLM on the ppl@1 score measured with the CodeT5-Small LM. The synthetic bad code has 1.727 ppl@1, 44% higher than what our CCLM achieves (1.198). This means BreaC generates bad code that is more realistic than synthetic errors from the perspective of the LM.

Case Studies Figure 2 presents our first case study showing that BreaC is capable of breaking LLMs to generate bad code with non-trivial errors. We show a ground truth program from the SPoC dataset (Figure 2a), the version generated by the CodeT5-Large LM (Figure 2b), and the version generated with GeDi (Figure 2c), i.e., the same CodeT5-Large LM broken by our CodeT5-Small CCLM. The program generated by the LM can be compiled but is functionally incorrect due to missing cout statements. The program generated with GeDi has all necessary cout statements but cannot be

[1]We note that PaLM (Chowdhery et al., 2022) achieves higher fix accuracy than BIFI on DeepFix, but with a model that is orders of magnitude larger than BIFI and ours. Therefore, we do not consider PaLM for comparison. In fact, the datasets generated by BreaC can potentially benefit fine-tuning PaLM on DeepFix.

compiled because of two compilation errors introduced by the CCLM. The first error is a type error that adds an integer to `endl`, which is an identifier from the standard library representing the newline character. The second error is an extra bracket. Apart from compilation errors, an unnecessary return statement `return 0;` is generated in the end.

In Figure 3, we provide our second case study, where we show three real programs from the DeepFix dataset and a generation from BreaC. Figure 3a shows a real bad program where `x` is an undefined variable. Figure 3b is another real bad program that lacks a bracket in the end. Figure 3c is a good program. Figure 3d is the bad code generated by BreaC from the good code in Figure 3c. Figure 3d has two errors that resemble the errors in Figures 3a and 3b: an undefined variable `k` and an unclosed bracket. This shows that BreaC is able to learn and even combine realistic error patterns.

5 Related Work

We discuss existing works mostly closely related to ours.

Controlled Generation for Natural Language Even though controlled generation is a new topic for code, it has been extensively studied for natural language (Prabhumoye et al., 2020; Jin et al., 2022). Hu et al. (2017) leverage variational auto-encoders for controlled text generation. PPLM modifies the hidden states of Transformer models with a discriminator DIS on candidate attributes, which drives the generation towards the direction of the desired attribute (Dathathri et al., 2020). CTRL is a CCLM trained on a large dataset of text with control codes (Keskar et al., 2019). GeDi implements DIS with a CCLM for efficiency (Krause et al., 2021).

Large Language Models for Code Recently, LLMs have been extensively used for program synthesis (Chen et al., 2021; Austin et al., 2021; Li et al., 2022; Nijkamp et al., 2022). Xu et al. (2022) provide a systematic evaluation of existing LLMs for Code. Despite their effectiveness, LLMs are found to often generate programs with different kinds of errors or even security vulnerabilities (Chen et al., 2021; Chowdhery et al., 2022; Pearce et al., 2022).

Reliable and Robust Code Models Several works look into improving the reliability of code LLMs. NSG combines static analysis with neural models to generate code that passes more static checks (Mukherjee et al., 2021). CompCoder (Wang et al., 2022) and CodeRL (Le et al., 2022) incorporate signals from parsers as a reward in RL. Synchromesh retrieves similar target programs and uses constrained decoding to generate reliable code (Poesia et al., 2022). AlphaCode (Li et al., 2022) improves program generation by conditioning on values and tags. Bielik & Vechev (2020); Yefet et al. (2020) handle the adversarial robustness of code models. Different from our work, they focus on other tasks such as type inference and other models such as graph neural networks.

Training Data Generation for Bug Detection and Repair An open challenge for training neural bug detectors and fixers is the lack of a large, realistic parallel training set (He et al., 2021; Yasunaga & Liang, 2021; He et al., 2022). One way to address this challenge is learning to inject realistic bugs into correct programs. Semseed learns and applies bug seeding patterns to create buggy programs (Patra & Pradel, 2021). BugLab and BIFI jointly learn a bug injector and a bug detector (Allamanis et al., 2021) or fixer (Yasunaga & Liang, 2021).

6 Conclusion and Future Work

We presented BreaC for breaking LLM-based code generation. The key idea is to leverage unlikelihood training and self-adversarial reinforcement learning to train a CCLM that can control the compilability or parsibility of the generated program. We demonstrated BreaC' effectiveness on both adversarial and benign scenarios. We hope that our work can stimulate future research on improving the reliability of LLM-based code generation from the attack perspective.

We consider three potential future work items. The first item is to extend BreaC to handle more languages and other types of bugs. Second, we would like to enhance BreaC's generalization and enable BreaC to achieve zero-shot or few-shot transferability between datasets. The third direction is to loosen the prerequisites discussed in Section 3.1 for launching BreaC's attack.

ETHICS STATEMENT

This work describes BreaC, which breaks LLM-based code generation such that they generate erroneous code. Our goal with BreaC is to improve the reliability of LLM-based code generators with identified attacks. Moreover, we show in Section 4.2 that BreaC can be used in the benign use case of improving neural bug fixers. However, attackers can use BreaC for malicious purposes, e.g., to break the functionality of a cloud service based on LLM-based code generation. When the attack introduces compilation or parsing errors, as done in this work, it can be detected with a compiler or a parser. When BreaC is further applied to introduce more stealthy bugs such as functional bugs and security vulnerabilities, detecting the attack is more challenging. Possible mitigations include using a static analyzer or using BreaC to generate a parallel dataset for training a bug detector.

REPRODUCIBILITY STATEMENT

We will publicly release all code, dataset, and trained models in this work.

REFERENCES

GitHub Copilot, 2022. URL https://github.com/features/copilot.

Documentation for difflib.SequenceMatcher.get_matching_blocks, 2022. URL https://docs.python.org/3/library/difflib.html#difflib.SequenceMatcher.get_matching_blocks.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. In *NeurIPS*, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/ea96efc03b9a050d895110db8c4af057-Abstract.html.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *NeurIPS*, 2015. URL https://proceedings.neurips.cc/paper/2015/hash/e995f98d56967d946471af29d7bf99f1-Abstract.html.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *ICML*, 2021. URL http://proceedings.mlr.press/v139/berabi21a.html.

Pavol Bielik and Martin Vechev. Adversarial robustness for code. In *ICML*, 2020. URL http://proceedings.mlr.press/v119/bielik20a.html.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, et al. Language models are few-shot learners. In *NeurIPS*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022. URL https://doi.org/10.48550/arXiv.2204.02311.

Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and Rosanne Liu. Plug and play language models: A simple approach to controlled text generation. In *ICLR*, 2020. URL https://openreview.net/forum?id=H1edEyBKDS.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019. URL https://doi.org/10.18653/v1/n19-1423.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *CoRR*, abs/2204.05999, 2022. doi: 10.48550/arXiv.2204.05999. URL https://doi.org/10.48550/arXiv.2204.05999.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In *NeurIPS*, 2014. URL https://proceedings.neurips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. Deepfix: Fixing common C language errors by deep learning. In *AAAI*, 2017. URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603.

Jingxuan He, Cheng-Chun Lee, Veselin Raychev, and Martin Vechev. Learning to find naming issues with big code and small supervision. In *PLDI*, 2021. URL https://doi.org/10.1145/3453483.3454045.

Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. On distribution shift in learning-based bug detectors. In *ICML*, 2022. URL https://proceedings.mlr.press/v162/he22a.html.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In *ICSE*, 2012. URL https://doi.org/10.1109/ICSE.2012.6227135.

Zhiting Hu, Zichao Yang, Xiaodan Liang, Ruslan Salakhutdinov, and Eric P. Xing. Toward controlled generation of text. In *ICML*, 2017. URL http://proceedings.mlr.press/v70/hu17e.html.

Di Jin, Zhijing Jin, Zhiting Hu, Olga Vechtomova, and Rada Mihalcea. Deep learning for text style transfer: A survey. *Comput. Linguistics*, 48(1):155–205, 2022. URL https://doi.org/10.1162/coli_a_00426.

Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. Repair is nearly generation: Multilingual program repair with llms. *CoRR*, abs/2208.11640, 2022. URL https://doi.org/10.48550/arXiv.2208.11640.

Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. CTRL: a conditional transformer language model for controllable generation. *CoRR*, abs/1909.05858, 2019. URL http://arxiv.org/abs/1909.05858.

Ben Krause, Akhilesh Deepak Gotmare, Bryan McCann, Nitish Shirish Keskar, Shafiq R. Joty, Richard Socher, and Nazneen Fatema Rajani. Gedi: Generative discriminator guided sequence generation. In *Findings of EMNLP*, 2021. URL https://doi.org/10.18653/v1/2021.findings-emnlp.424.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. Spoc: Search-based pseudocode to code. In *NeurIPS*, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/7298332f04ac004a0ca44cc69ecf6f6b-Abstract.html.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *CoRR*, abs/2207.01780, 2022. URL https://doi.org/10.48550/arXiv.2207.01780.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, et al. Competition-level code generation with AlphaCode. *CoRR*, abs/2203.07814, 2022. URL https://doi.org/10.48550/arXiv.2203.07814.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS Datasets and Benchmarks*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018. URL https://openreview.net/forum?id=rJzIBfZAb.

Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas W. Reps, Swarat Chaudhuri, and Christopher M. Jermaine. Neural program generation modulo static analysis. In *NeurIPS*, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/9e1a36515d6704d7eb7a30d783400e5d-Abstract.html.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *CoRR*, abs/2203.13474, 2022. URL https://doi.org/10.48550/arXiv.2203.13474.

Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *ESEC/FSE*, 2021. URL https://doi.org/10.1145/3468264.3468623.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *IEEE S&P*, 2022. URL https://doi.org/10.1109/SP46214.2022.9833571.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *ICLR*, 2022. URL https://arxiv.org/abs/2201.11227.

Shrimai Prabhumoye, Alan W. Black, and Ruslan Salakhutdinov. Exploring controllable text generation techniques. In *COLING*, 2020. URL https://doi.org/10.18653/v1/2020.coling-main.1.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. URL https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. In *ICLR*, 2016. URL http://arxiv.org/abs/1511.06732.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Richard S. Sutton, David A. McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NeurIPS*, 1999. URL http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *ICLR*, 2014. URL http://arxiv.org/abs/1312.6199.

Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. Universal adversarial triggers for attacking and analyzing NLP. In *EMNLP-IJCNLP*, 2019. URL https://doi.org/10.18653/v1/D19-1221.

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. In *Findings of ACL*, 2022. URL https://doi.org/10.18653/v1/2022.findings-acl.2.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, 2021. URL https://doi.org/10.18653/v1/2021.emnlp-main.685.

Sean Welleck, Ilia Kulikov, Stephen Roller, Emily Dinan, Kyunghyun Cho, and Jason Weston. Neural text generation with unlikelihood training. In *ICLR*, 2020. URL https://openreview.net/forum?id=SJeYe0NtvH.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *MAPS@PLDI*, 2022. URL https://doi.org/10.1145/3520312.3534862.

Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *ICML*, 2020. URL http://proceedings.mlr.press/v119/yasunaga20a.html.

Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *ICML*, 2021. URL http://proceedings.mlr.press/v139/yasunaga21a.html.

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proc. ACM Program. Lang.*, 4(OOPSLA):162:1–162:30, 2020. URL https://doi.org/10.1145/3428230.

Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. In *AAAI*, 2017. URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14344.