Ensuring Functional Correctness of Large Code Models with Selective Generation

Jaewoo Jeong
CSE
POSTECH
jeongjw@postech.ac.kr

Taesoo Kim
SCS & SCP
GaTech
taesoo@gatech.edu

Sangdon Park
GSAI & CSE
POSTECH
sangdon@postech.ac.kr

Abstract

The hallucination of code generation models hinders their applicability to systems requiring higher safety standards. One critical bottleneck in addressing code hallucination is the difficulty of identifying the functional correctness of generated code, due to its unnatural form. We address this core bottleneck by automatically generating unit tests using dynamic code analysis tools, leveraging the *executable nature* of code. Accordingly, we propose *selective code generator* that abstains from uncertain generations – based on the functional correctness evaluated by generated unit tests – to theoretically control the correctness among non-abstained answers, *i.e.,* the false discovery rate. Finally, we propose to use generated unit tests in evaluation as well as in learning for precise code evaluation, calling this paradigm *FuzzEval*. We demonstrate the efficacy of our method along with the controllability of code hallucination and reasonable selection efficiency.

1 Introduction

Large language models (LLMs) are recently proven to be performant in various tasks, including question-answering, summarization, mathematical reasoning, and algorithmic problem-solving [1, 2, 3, 4]. Along with language generation, code generation is closely related but has its own benefit as a core task for addressing many applications, including mathematical solving via program of thought, security patch generation, and general program development [5, 6, 7].

Recent development on large code models have primarily focused on enhancing model performance [8, 9] – thereby *indirectly controlling* the functional hallucination. However, *direct control* methods to address *functionality hallucination in code generation*, *i.e.,* a situation where generated code does not satisfy a desired functionality, remain unexplored.

In contrast to hallucination control in code, heuristic and certified methods for hallucination control in natural language generation have been extensively studied beyond enhancing the model performance. For example, the language hallucination is heuristically measured by generating multiple answers and checking the consensus among them [10, 11]. As more sophisticated methods, hallucination is carefully controlled by certified methods, including conformal prediction [12] and selective prediction [13], providing certified ways to mitigate language hallucination [14, 15, 16].

We claim that a critical bottleneck in mitigating code hallucination mainly stems from the intricacy of identifying functional equivalence between two code snippets due to the un-natural form of code. In natural language, textual entailment [17] is the main building block in measuring the semantic correctness of an answer, *i.e.,* an answer is correct if a question-associated context entails the answer. Given this definition on the correctness between two sentences, humans can manually annotate entailment labels to learn entailment-predicting models [18]. However, this is challenging in code as it is not easy for humans to decide whether one code entails another due to its complex, un-natural structure for obtaining entailment labels. This can be partially mitigated by constructing a few unit

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

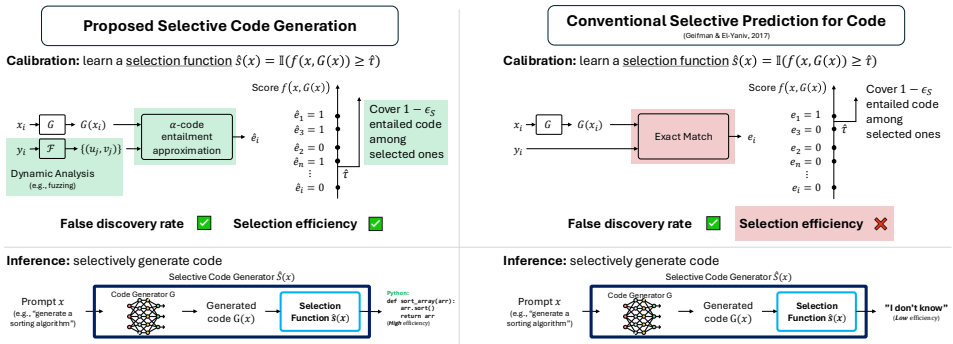


Figure 1: Overview of our proposed selective code generation. We leverage an abstaining option to selectively generate code to *control* the rate of hallucination in an FDR. Our selective generator learns a selection function by leveraging dynamic code analysis tools to automatically generate unit tests and use them as a calibration set for the selection function and also as a test set for evaluation.

tests [19, 20, 21] and executing it to check discrepancies in output, while limited to a small number of unit tests.

To address these challenges, we extend on prior programming language literature to re-define entailment for code generation. We further exploit the *executable property of code* by automatically generating unit tests through code analysis tools. In particular, we leverage fuzzing methods [22], one of practical code analysis tools, to automatically generate unit tests.

Given the code entailment definition and generated unit tests, we propose certified selective code generation for code hallucination control. This includes a learning algorithm of a selective generator for code, a post-processor of an original generator, which provides a controllability guarantee on the rate of hallucination in a false discovery rate (FDR), *i.e.,* a learned selective generator provides a desired or minimum level of hallucination. The learning algorithm mainly leverages fuzzing for checking the correctness of code in a self-supervised manner for supervised learning in selective code generators. Finally, we leverage fuzzing for code evaluation as well as selective generator learning. We claim that automatically generated unit tests provide rigorous evaluation, calling this evaluation paradigm *FuzzEval* to distinguish from *HumanEval* [19].

To summarize, the main contribution of our work lies in the selective generation learning algorithm that controls the hallucination rate. We demonstrate the efficacy of our learning algorithm over open and closed code generators under diverse experiment setups, including four code generators, four datasets, four programming languages, and diverse baselines. Our results demonstrate the hallucination-controllability with reasonable efficiency, and further demonstrate the benefits of automatically generated unit tests. We release code for our algorithm and evaluation dataset at `https://github.com/trustml-lab/selective-code-generation`

1.1 Related Work

We introduce tightly related work here. See additional related work in Appendix A.

Execution Based Code Correctness Evaluation. Popular code generation benchmarks, such as HumanEval [19], APPS [23], or MBPP [20] evaluate the functional correctness of a generated code snippet by executing unit tests. However, generating unit tests for such evaluation purposes is a costly task. Automated unit test generations has been explored in recent work. LLMs itself has been used to improve unit tests [24]. EvalPlus [25] and SemCoder [26] adopt a type-aware input mutation strategy based on LLM-generated seed, while Mercury [27] leverages LLMs to generate random test case generators for evaluation split. Our work complements these work by generating unit tests via dynamic analysis tools, *e.g., fuzzing tool*, that have been extensively studied and validated within the computer security community, to explore execution paths.

Previous works that leverage unit tests for evaluation commonly use the pass@k metric to assess functional correctness [25, 26]. EvalPlus [25] reports a drop in pass@k metric with larger test suites indicating that the metric is sensitive to the number and quality of unit tests. Furthermore, pass@k cannot consider partially correct or incorrect programs. Our work addresses this by introducing the FDR-CE metric with a statistical guarantee.

Selective Generation. Selective prediction abstains from answering if a model is not confident of the answer, from which it controls the risk at a desirable level. This method can be applicable to various tasks. Selective classification for deep learning [13] considers classification tasks. Selective text generation [16] is applied to a language generation task by introducing textual entailment to control hallucination. We extend selective generation for code by leveraging an executable property of code, while still having hallucination controllability.

2 Problem

We consider a learning problem in code generation and see Appendix B for its preliminary. In particular, we learn a code generator to control code hallucination in the perspective of *functional correctness* where the generator abstains from answering if it is not certain on the correctness of the generated code. Let $\mathcal{W}$ be a set of tokens, $\mathcal{W}^* \coloneqq \cup_{i=0}^{\infty} \mathcal{W}^i$, $\mathcal{X} \coloneqq \mathcal{W}^*$ be a set of input prompts for a code generator, $\mathcal{Y} \coloneqq \mathcal{W}^*$ be a set of code snippets, $\mathcal{D}$ be a distribution that depends on prompt and code pairs $\mathcal{X} \times \mathcal{Y}$ along with other random sources, and $G : \mathcal{X} \to \mathcal{Y}$ be a code generator.

To control the hallucination rate of a code generator G, we consider the following selective generator $\hat{S} : \mathcal{X} \to \mathcal{Y} \cup \{\texttt{IDK}\}$ [13, 16]: $\hat{S}(\mathbf{x}) \coloneqq \begin{cases} G(\mathbf{x}) & \text{if } \hat{s}(\mathbf{x}) = 1 \\ \texttt{IDK} & \text{otherwise} \end{cases}$, where IDK is a short-hand for "I don't know" and $\hat{s} : \mathcal{X} \to \{0, 1\}$ is a selection function. Here, we consider a setup that the target code generator G is given and we learn a selection function $\hat{s}$.

We mainly learn the selective generator under the independent and identically distributed (i.i.d.) assumption by controlling a false discovery rate (FDR) for code. In particular, we consider the risk definition of a selective generator $\hat{S}$ via an FDR with a relation R, *i.e.,*

$$\mathcal{R}_R(\hat{S}) \coloneqq \mathbb{P}\left\{(\hat{S}(\mathbf{x}), \mathbf{y}) \notin R \;\middle|\; \hat{S}(\mathbf{x}) \neq \texttt{IDK}\right\}, \tag{1}$$

where the probability is taken over $(\mathbf{x}, \mathbf{y}) \sim \mathcal{D}$. Here, we measure the ratio of failure, *i.e.,* the ratio that generated code $G(\mathbf{x})$ does not have a relation R with correct code $\mathbf{y}$, among not-abstaining cases.

Given the learning objective in the FDR, we find a learning algorithm $\mathcal{A}$ for $\hat{S}$ such that given a calibration set $\mathbf{Z}$ with $|\mathbf{Z}| = n$, the learned selective generator $\hat{S} \coloneqq \mathcal{A}(\mathbf{Z})$ controls a desired risk level ε with probability at least $1 - \delta$, *i.e.,* $\mathbb{P}\left\{\mathcal{R}_R(\mathcal{A}(\mathbf{Z})) \leq \varepsilon\right\} \geq 1 - \delta$, where the probability is taken over $\mathbf{Z} \sim \mathcal{D}^n$. Here, the main challenges include designing a correctness relation R for code generation and finding a learning algorithm with the above PAC-style controllability guarantee, while maximizing *selection efficiency*, *i.e.,* $\mathbb{P}\{\hat{S}(\mathbf{x}) \neq \texttt{IDK}\}$.

3 Method: Selective Code Generation

We introduce a novel definition of code entailment in Section 3.1, followed by a risk definition in Section 3.2. Section 3.3 and 3.4 present the bounds for the code entailment. Our FDR-controlling algorithm is described in Section 3.5, followed by its controllability guarantee in Section 3.6. Lastly, we highlight the importance of evaluation via fuzzing in Section 3.7.

3.1 Code Entailment

Measuring the functional correctness of a generated code snippet is a challenging task. In particular, the functionality of generated code is evaluated using only a limited set of manually chosen unit tests [19]. We re-define the concept of *code entailment* that leverages dynamic code analysis tools to define the functional correctness. To compensate for the lack of unit tests in determining *code entailment*, unit tests and expected outputs are extracted from a dynamic code analysis tool.

Let $\mathcal{U}$ and $\mathcal{V}$ be a set of input and output states for all code snippets, respectively. A dynamic code analysis tool $\mathcal{F} : \mathcal{Y} \times \mathcal{U} \to \mathcal{U} \times \mathcal{V}$ returns a pair of input and output states for a given code snippet $\mathbf{y}$ by executing a code snippet $\mathbf{y}$ with a seed input $\mathbf{s} \in \mathcal{U}$, randomly drawn from a distribution over seed inputs $\mathcal{D}_{\mathcal{U}}$, *i.e.,* $(\mathbf{u}, \mathbf{v}) = \mathcal{F}(\mathbf{y}, \mathbf{s})$, where $\mathbf{s} \sim \mathcal{D}_{\mathcal{U}}$ and $\mathbf{y}(\mathbf{u}) = \mathbf{v}$. Here, we assume that $\mathbf{y}$ is a deterministic code snippet for notational simplicity, but our results maintain with stochastic code by fixing the random seed of $\mathcal{F}$. As mentioned above, the analysis tool $\mathcal{F}$ facilitates learning and

evaluation with code by providing an automatic way to generate a huge number of unit tests. We then introduce the definition of α-code entailment by leveraging the dynamic code analysis tool $\mathcal{F}$.

Definition 1 (α-code entailment)**.** *A code snippet* $\mathbf{y} \in \mathcal{Y}$ α-entails $\hat{\mathbf{y}} \in \mathcal{Y}$ *in* $\mathcal{F}$ *and* $\mathcal{D}_{\mathcal{U}}$ *if*

$$\mathbb{P}_{\mathbf{y}}\left\{\hat{\mathbf{y}}(\mathbf{u}) = \mathbf{v}\right\} \geq 1 - \alpha, \tag{2}$$

where the probability is taken over $\mathbf{s} \sim \mathcal{D}_{\mathcal{U}}$*,* $(\mathbf{u}, \mathbf{v}) = \mathcal{F}(\mathbf{y}, \mathbf{s})$*, and we call* $\mathbb{P}_{\mathbf{y}}\left\{\hat{\mathbf{y}}(\mathbf{u}) = \mathbf{v}\right\}$ *an* expected functional correctness *of* $\hat{\mathbf{y}}$ *with respect to* $\mathbf{y}$*,* $\mathcal{F}$*, and* $\mathcal{D}_{\mathcal{U}}$*.*

We formally re-define the concept of *probabilistic tests* originally introduced by **(author?)** [28]. Our definition differs from prior work, **(author?)** [29, 30, 31], which focuses on identifying counter examples for program inequivalence. Also, our definition differs slightly from program equivalence definitions in **(author?)** [32, 33, 34], as we use α to accommodate different code generator quality and enhance practicality.

In code generation, the α-code entailment provides a foundation for measuring functional correctness. In particular, to measure the functional correctness between two code snippets $\mathbf{y}$ and $\hat{\mathbf{y}}$, we need to check whether two code snippets $\mathbf{y}$ and $\hat{\mathbf{y}}$ have the same output state for all input states. To check this bidirectional equivalence relation, we first consider the one-directional definition via code entailment by checking whether code $\hat{\mathbf{y}}$ satisfies all input and output pairs for code $\mathbf{y}$ following Definition 1. By considering entailment from $\mathbf{y}$ to $\hat{\mathbf{y}}$ and vice versa, we can eventually define the functional equivalence of code. In this paper, instead of analyzing the bidirectional equivalence, we only consider the one-directional relaxed notation of correctness via code entailment, which suffices for code generation, *e.g.,* we can say that $\hat{\mathbf{y}}$ is correct if it contains all functionalities of $\mathbf{y}$ along with other functionalities. The example of α-entailment is presented in Table 7.

3.2 False Discovery Rate via Code Entailment

We define a relation for the FDR risk in (1) via α-code entailment. We first denote the set of α-entailment code snippets of $\mathbf{y}$ by $E_\alpha(\mathbf{y})$, *i.e.,* $E_\alpha(\mathbf{y}) \coloneqq \{\bar{\mathbf{y}} \mid \mathbb{P}_{\mathbf{y}}\{\bar{\mathbf{y}}(\mathbf{u}) = \mathbf{v}\} \geq 1 - \alpha\}$, which approximately corresponds to the set of all code snippets that has the most functionalities of $\mathbf{y}$. By the definition of the code-entailment, $\hat{\mathbf{y}} \in E_\alpha(\mathbf{y})$ implies that $\mathbf{y}$ α-entails $\hat{\mathbf{y}}$.

Using the same definition of $E_\alpha(\mathbf{y})$, we further define the correctness relation between two code snippets as follows: $R_\alpha \coloneqq \{(\hat{\mathbf{y}}, \mathbf{y}) \mid \hat{\mathbf{y}} \in E_\alpha(\mathbf{y})\}$. Then, from (1), we define the FDR with code entailment relation R_α (FDR-CE). Equivalently, we use the following FDR-CE definition: $\mathcal{R}_\alpha(\hat{S}) \coloneqq \mathbb{P}\{\hat{S}(\mathbf{x}) \notin E_\alpha(\mathbf{y}) \mid \hat{S}(\mathbf{x}) \neq \texttt{IDK}\}$. Here, the probability is taken over $(\mathbf{x}, \mathbf{y}) \sim \mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$, where $\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$ is a distribution over $\mathcal{X} \times \mathcal{Y}$.

3.3 Code Entailment Estimation

In natural languages, an entailment relation between two sentences can be easily obtained by human annotators. However, identifying functionalities between two code snippets by humans (*i.e.,* deciding whether $\hat{S}(\mathbf{x}) \notin E_\alpha(\mathbf{y})$ or not) are challenging due to the un-natural form of programming languages. We overcome this challenge with dynamic analysis tool that exploits an *executable property of code*.

Inspired by [16], we estimate E_α and use it as a pseudo-labeling function as the exact E_α is difficult to obtain. In particular, given a true code snippet $\mathbf{y}$ and a generated code snippet $\bar{\mathbf{y}}$, recall the expected functional correctness $\mathbb{P}_{\mathbf{y}}\{\bar{\mathbf{y}}(\mathbf{u}) = \mathbf{v}\}$. Considering that the input-output pairs $(\mathbf{u}, \mathbf{v})$ are independently drawn from the seed distribution $\mathcal{D}_{\mathcal{U}}$ by $\mathcal{F}$, we estimate the lower bound of the expected functional correctness by using the standard binomial tail bound. Specifically, let the lower binomial tail bound $\hat{L}$ of $\mathbb{P}_{\mathbf{y}}\{\bar{\mathbf{y}}(\mathbf{u}) = \mathbf{v}\}$ be $\hat{L}(\mathbf{y}, \bar{\mathbf{y}}, n_{\mathbf{y}}, \varepsilon_E) \coloneqq \hat{L}_{\text{Binom}}(\hat{k}; n_{\mathbf{y}}, \varepsilon_E)$, where $n_{\mathbf{y}}$ is the number of samples, $\mathbf{S}_{\mathbf{y}} \sim \mathcal{D}_{\mathcal{U}}^{n_{\mathbf{y}}}$, and $\hat{k} \coloneqq \sum_{(\mathbf{u},\mathbf{v})\in\{(\mathbf{u},\mathbf{v})|\mathbf{s}\in\mathbf{S}_{\mathbf{y}},\mathcal{F}(\mathbf{y},\mathbf{s})=(\mathbf{u},\mathbf{v})\}} \mathbb{1}(\bar{\mathbf{y}}(\mathbf{u}) = \mathbf{v})$. Here, $\hat{L}_{\text{Binom}}$ is the lower standard binomial tail bound, where $F(k; n, \theta)$ be a cumulative distribution function of a binomial distribution with n trials and success probability θ, and $\hat{L}_{\text{Binom}}(k; n, \delta) \coloneqq \sup\{\theta \in [0, 1] \mid 1 - F(k; n, \theta) \leq \delta\} \cup \{0\}$. Then, due to its definition, the lower bound holds with high probability [35] as follows: $\mathbb{P}\{\hat{L}(\mathbf{y}, \bar{\mathbf{y}}, n_{\mathbf{y}}, \varepsilon_E) \leq \mathbb{P}_{\mathbf{y}}\{\bar{\mathbf{y}}(\mathbf{u}) = \mathbf{v}\}\}\} \geq 1 - \varepsilon_E$, where the probability is taken over $\mathbf{S}_{\mathbf{y}} \sim \mathcal{D}_{\mathcal{U}}^{n_{\mathbf{y}}}$.

Importantly, we carefully choose reasonably small unit test size $n_{\mathbf{y}}$ for a given $\mathbf{y}$ via Algorithm 1. In particular, sample size for the binomial tail bound is usually given, but we can generate as many

samples as we wish by executing a dynamic code analysis tool $\mathcal{F}$. Here, the number of samples should depend on the difficulty in evaluating the correctness of generated code $\hat{\mathbf{y}}$, *i.e.,* as the generated code is ambiguous to check the α-code entailment, we need more samples to be certain. To this end, we increase the unit test size $n_{\mathbf{y}}$ until the lower bound $\hat{L}$ of an expected functional correctness is larger than $1-\alpha$ (Line 3) to achieve the expected correctness as well. If the sample size exceeds maximum size n_{max}, the algorithm returns n_{max} (Line 7).

From this lower bound $\hat{L}$, we define an *estimated entailment set* as follows:

$$\hat{E}_{\alpha,\varepsilon_E}(\mathbf{y}) \coloneqq \left\{ \bar{\mathbf{y}} \;\middle|\; \hat{L}(\mathbf{y}, \bar{\mathbf{y}}, n_{\mathbf{y}}, \varepsilon_E) \geq 1-\alpha \right\}. \tag{3}$$

Intuitively, code $\bar{\mathbf{y}} \in \hat{E}_{\alpha,\varepsilon_s}(\mathbf{y})$ likely satisfies $\bar{\mathbf{y}} \in E_\alpha(\mathbf{y})$, meaning $\mathbf{y}$ α-entails $\hat{\mathbf{y}}$ with high probability. Thus, the FDR-CE based on the estimated entailment set is defined as $\mathcal{R}_{\alpha,\varepsilon_E}(\hat{S}) \coloneqq \mathbb{P}\{\hat{S}(\mathbf{x}) \notin \hat{E}_{\alpha,\varepsilon_E}(\mathbf{y}) | \hat{S}(\mathbf{x}) \neq \texttt{IDK}\}$. Here, the probability is taken over $(\mathbf{x}, \mathbf{y}, n_{\mathbf{y}}) \sim \mathcal{D}_{\mathcal{X}\times\mathcal{Y}\times\mathbb{N}}$ and $\mathbf{S}_{\mathbf{y}} \sim \mathcal{D}_{\mathcal{U}}^{n_{\mathbf{y}}}$, where we simply denote the distribution associated to $(\mathbf{x}, \mathbf{y}, n_{\mathbf{y}}, \mathbf{S}_{\mathbf{y}})$ by $\mathcal{D}$. It is a good alternative for the original FDR-CE $\mathcal{R}_\alpha(\hat{S})$. In the following, we connect $\mathcal{R}_\alpha$ and $\mathcal{R}_{\alpha,\varepsilon_E}$ in learning.

3.4 False Discovery Rate Bound

We consider the upper bound of the FDR-CE $\mathcal{R}_\alpha(\hat{S})$, which leverages the estimated entailment set. Specifically, from Lemma 2, we have $\mathcal{R}_\alpha(\hat{S}) \leq \mathbb{P}_{\mathcal{D}_{\hat{S}}}\{e = 0, \hat{e} = 1\} + \mathcal{R}_{\alpha,\varepsilon_E}(\hat{S})$. Here, $e \coloneqq G(\mathbf{x}) \in E_\alpha(\mathbf{y})$, $\hat{e} \coloneqq G(\mathbf{x}) \in \hat{E}_{\alpha,\varepsilon_E}(\mathbf{x})$, and $\mathbb{P}_{\mathcal{D}_{\hat{S}}}\{\cdot\} = \mathbb{P}\{\cdot \mid \hat{S}(\mathbf{x}) \neq \texttt{IDK}\}$, where the probability is taken over $(\mathbf{x}, \mathbf{y}, n_{\mathbf{y}}, \mathbf{S}_{\mathbf{y}}) \sim \mathcal{D}$. This intuitively suggests that the FDR-CE over the exact entailment set can be approximated by the FDR-CE over the estimated entailment set along with its false entailment rate (FER). Moreover, the FER is related to the correctness of the binomial tail bound, controlled by ε_E. This implies the following key lemma. See Appendix H for a proof.

Lemma 1. *For any $\alpha, \varepsilon_E \in (0,1)$, and $\hat{S}$, we have $\mathcal{R}_\alpha(\hat{S}) \leq \varepsilon_E + \mathcal{R}_{\alpha,\varepsilon_E}(\hat{S})$.*

Next, we use this bound to provide an algorithm for $\hat{S}$, controlling this upper bound at a desired level.

3.5 FDR-CE Control Algorithm

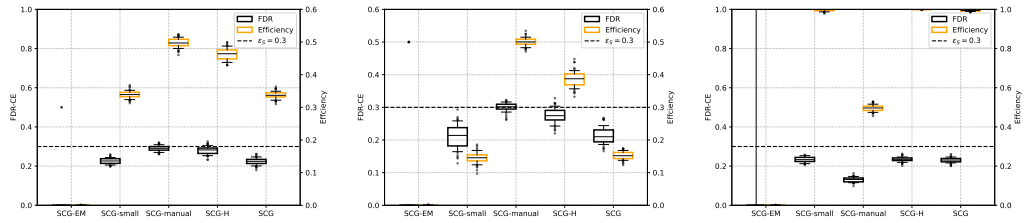
We propose a selective generator learning algorithm for code that controls the FDR-CE. In particular, we consider a scalar-parameterization of the selection function $\hat{s}$, *i.e.,* $\hat{s}(\mathbf{x}) \coloneqq \mathbb{1}\left(f(\mathbf{x}, G(\mathbf{x})) \geq \tau\right)$, where $\tau \in \mathbb{R}$. While the scoring function can, in general, be any function that quantifies the confidence on generated code, we employ the standard length-normalized log-probability for generated tokens as the default choice, *i.e.,* $f(\mathbf{x}, G(\mathbf{x})) = \sum_i \ln p_i \,/\, |G(\mathbf{x})|$, where p_i is the probability by G to generate the i-th token. Then, our algorithm searches $\hat{S}$ that closely controls the upper bound in Lemma 1 within ε_S by solving the following optimization problem:

$$\min_{\tau\in\mathbb{R}} \tau \quad \text{subj. to} \quad \varepsilon_E + \hat{U}_{\text{Binom}}(\hat{k}; |\hat{\mathbf{Z}}|, \delta_S / \lceil \log_2 |\mathbf{Z}| \rceil) \leq \varepsilon_S, \tag{4}$$

where $\mathbf{Z} \sim \mathcal{D}^n$, $\hat{\mathbf{Z}} \coloneqq \{(\mathbf{x}, \mathbf{y}, _) \in \mathbf{Z} \mid f(\mathbf{x}, G(\mathbf{x})) \geq \tau\}$, $\hat{k} \coloneqq \sum_{(\mathbf{x},\mathbf{y},_)\in\hat{\mathbf{Z}}} \mathbb{1}(G(\mathbf{x}) \notin \hat{E}_{\alpha,\varepsilon_E}(\mathbf{y}))$, and $\hat{U}_{\text{Binom}}$ is the upper binomial tail bound, similarly defined as the lower bound $\hat{L}_{\text{Binom}}$. Here, the algorithm also returns $\hat{U} \coloneqq \varepsilon_E + \hat{U}_{\text{Binom}}(\cdot)$, the upper bound of the FDR-CE for the optimal $\hat{\tau}$. We denote the algorithm solving (4) by $\mathcal{A}_{\text{SCG}}$ and see Algorithm 2 for its implementation details. Appendix N.1 includes guidelines on user-specified parameter selection.

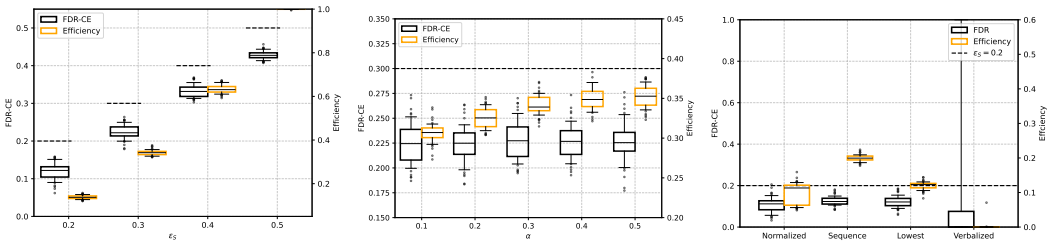
At a high level, the algorithm essentially finds a selective code generator (parametrized by τ) that minimizes τ, to maximize the selection efficiency of the selective generator, under the constraint of controlling the FDR-CE by a desired level ε_S. If the minimization is not feasible, the algorithm returns a selective generator that controls the minimum FDR-CE, indicated by $\hat{U}$.

Dynamic Code Analysis via Fuzzing. We consider any dynamic analysis tool $\mathcal{F} : \mathcal{Y} \times \mathcal{U} \to \mathcal{U} \times \mathcal{V}$ to generate a unit test $(\mathbf{u}, \mathbf{v})$ for a given seed $\mathbf{s}$ by executing code $\mathbf{y}$. We adopt a fuzzing method that are popularized due to its simplicity and efficacy in finding bugs. This method is usually used for exploiting a certain execution to trigger bugs but we rather use it for exploring wider execution paths. In particular, we randomly sample a binary stream from distribution $\mathcal{D}_{\mathcal{U}}$ and use it as a seed $\mathbf{s} \sim \mathcal{D}_{\mathcal{U}}$ to arbitrarily assign an input state for code, *e.g.,* randomly initializing input parameters for



(a) GPT-4o (b) Gemini-1.5 Pro (c) DeepSeek R1

Figure 2: The box plots of the FDR-CE and selection efficiency for various LLMs ($\delta_S = 0.1$, $\varepsilon_S = 0.3$, $\varepsilon_E = 0.05$, $\alpha = 0.35$). See Appendix 4(a) for results with CodeLlama 13B Instruct.



(a) Varying ε_S (b) Varying α (c) Varying Scoring Functions

Figure 3: The FDR-CE results for GPT-4o with varying parameters and scoring functions. We use $\varepsilon_S = 0.3, \delta_S = 0.1, \alpha = 0.35$, and $\varepsilon_E = 0.05$ for Figure 3(a) and 3(b) and $\varepsilon_S = 0.2, \delta_S = 0.1, \alpha = 0.15$, and $\varepsilon_E = 0.05$ for Figure 3(c)

a function call. The initial state typically fails to lead to an interesting execution path; therefore, fuzzing method mutates the input seed to explore wider execution paths; *e.g.,* `Atheris` [36] mutates an input to increase code coverage. During this repeated execution of code, multiple input and output state pairs are generated, and we randomly select one of them as our final pair $(\mathbf{u}, \mathbf{v})$ for each seed $\mathbf{s}$.

3.6 Controllability Guarantee

Our algorithm $\mathcal{A}_{\text{SCG}}$ controls an FDR-CE of a learned selective generator. This is a direct consequence of selective prediction [13, 16]. See a proof in Appendix I.

Theorem 1. *For any $\varepsilon_S \in (0,1)$, $\delta_S \in (0,1)$, $\alpha \in (0,1)$, f, $\mathcal{F}$, $\mathcal{D}_{\mathcal{U}}$, and $\mathcal{D}$, we have $\mathbb{P}\{\mathcal{R}_\alpha(\hat{S}) \leq \hat{U}\} \geq 1 - \delta_S$, where the probability is taken over $\mathbf{Z} \sim \mathcal{D}^n$ and $(\hat{S}, \hat{U}) \coloneqq \mathcal{A}_{SCG}(\mathbf{Z})$.*

This means that $\mathcal{A}_{\text{SCG}}$ finds a selective code generator satisfying a desired FDR-CE ε_S (if $\varepsilon_S \geq \hat{U}$) or minimum level $\hat{U}$ (if $\hat{U} > \varepsilon_S$), *without having human-feedback* on code entailment labels.

3.7 FuzzEval: Evaluation via Fuzzing

We suggest to use automatically generated unit tests in evaluation as well. Traditionally, identifying the functional equivalence desired code and generated code in code generation have relied on manually obtained unit tests, *e.g.,* HumanEval [19]. But, as desired code gets complex so the number of its execution path exponentially increases, manually getting an enough number of unit tests is challenging. To address this bottleneck, we propose to use fuzzing, a dynamic code analysis tool, to automatically generate unit tests of given code, calling this evaluation paradigm *FuzzEval*.

4 Experiments

We demonstrate the efficacy of our selective code generation on open and closed LLMs in an algorithmic solving task. See Appendix M for additional experiments.

4.1 Setup

Dataset. We considered datasets with coding problems, where each problem has a correct canonical solution. We chose APPS [23], Mercury [27], HumanEval [19], and MBPP [20] for our task. Each dataset consists of Python programming questions, canonical code solutions, and few unit tests. Each

Table 1: Comparison results of SCG against baseline methods on different datasets with GPT-4o ($\alpha = 0.35, \delta_S = 0.1, \varepsilon_E = 0.05$ for all datasets). The FDR-CE satisfying the desired guarantees and the highest efficiency among methods that comply with the FDR-CE guarantees are marked in **bold.**

Methods	w/o Selective Generation	w/ Selective Generation				
	$\tau = -\infty$	SCG-EM	SCG-MANUAL	SCG-SMALL	SCG-H	**SCG**
APPS-F ($\varepsilon_S = 0.3$)						
1 - PASS@1(↓)	$0.436_{\pm 0.012}$	$0.020_{\pm 0.098}$	$0.293_{\pm 0.014}$	$0.226_{\pm 0.015}$	$0.282_{\pm 0.020}$	$0.227_{\pm 0.017}$
FDR-CE(↓)	$0.431_{\pm 0.011}$	$\mathbf{0.020_{\pm 0.099}}$	$0.291_{\pm 0.014}$	$\mathbf{0.226_{\pm 0.015}}$	$0.280_{\pm 0.020}$	$\mathbf{0.224_{\pm 0.018}}$
EFFICIENCY(↑)	$1.000_{\pm 0.000}$	$0.000_{\pm 0.000}$	$0.497_{\pm 0.014}$	$\mathbf{0.339_{\pm 0.012}}$	$0.463_{\pm 0.019}$	$0.337_{\pm 0.011}$
MBPP-F ($\varepsilon_S = 0.4$)						
1 - PASS@1	$0.299_{\pm 0.022}$	$0.000_{\pm 0.000}$	$0.258_{\pm 0.040}$	-	$0.301_{\pm 0.028}$	$0.304_{\pm 0.032}$
FDR-CE	$0.294_{\pm 0.027}$	$\mathbf{0.000_{\pm 0.000}}$	$\mathbf{0.254_{\pm 0.041}}$	-	$\mathbf{0.296_{\pm 0.029}}$	$\mathbf{0.300_{\pm 0.032}}$
EFFICIENCY	$1.000_{\pm 0.000}$	$0.001_{\pm 0.003}$	$0.499_{\pm 0.038}$	-	$\mathbf{0.997_{\pm 0.004}}$	$0.996_{\pm 0.006}$
HUMANEVAL-F ($\varepsilon_S = 0.3$)						
1 - PASS@1	$0.185_{\pm 0.058}$	$0.000_{\pm 0.000}$	$0.142_{\pm 0.080}$	-	$0.156_{\pm 0.165}$	$0.069_{\pm 0.173}$
FDR-CE	$0.207_{\pm 0.064}$	$\mathbf{0.000_{\pm 0.000}}$	$\mathbf{0.156_{\pm 0.086}}$	-	$0.145_{\pm 0.123}$	$\mathbf{0.049_{\pm 0.111}}$
EFFICIENCY	$1.000_{\pm 0.000}$	$0.008_{\pm 0.017}$	$\mathbf{0.492_{\pm 0.080}}$	-	$0.578_{\pm 0.401}$	$0.164_{\pm 0.118}$
MERCURY-F ($\varepsilon_S = 0.3$)						
1 - PASS@1	$0.174_{\pm 0.018}$	$0.000_{\pm 0.000}$	$0.138_{\pm 0.024}$	-	$0.169_{\pm 0.017}$	$0.170_{\pm 0.020}$
FDR-CE	$0.170_{\pm 0.019}$	$\mathbf{0.000_{\pm 0.000}}$	$\mathbf{0.133_{\pm 0.022}}$	-	$\mathbf{0.164_{\pm 0.019}}$	$\mathbf{0.165_{\pm 0.020}}$
EFFICIENCY	$1.000_{\pm 0.000}$	$0.001_{\pm 0.002}$	$0.504_{\pm 0.033}$	-	$0.998_{\pm 0.003}$	$\mathbf{0.998_{\pm 0.002}}$

Table 2: Comparison results of before and after applying SCG on various code generators, including CODET[37], LDB[38], and SFS[39] with GPT-3.5-Turbo. We use $\alpha = 0.4, \delta_S = 0.1, \varepsilon_E = 0.01$ for all datasets; we set $\varepsilon_S = 0.35$ for HumanEval-f and $\varepsilon_S = 0.45$ for MBPP-f. For HumanEval-f, we utilized a full dataset, whereas for MBPP-f, we constructed a new dataset using the union of subsets released by the authors of baseline methods. The FDR-CE satisfying the desired guarantees are marked in **bold.**

Methods	Base Model		(author?) [37]		(author?) [38]		(author?) [39]	
	wo/ SCG	w/ SCG	wo/ SCG	w/ SCG	wo/ SCG	w/ SCG	wo/ SCG	w/ SCG
MBPP-F								
1 - PASS@1 (↓)	$0.488_{\pm 0.039}$	$0.081_{\pm 0.184}$	$0.488_{\pm 0.037}$	$0.214_{\pm 0.190}$	$0.441_{\pm 0.033}$	$0.000_{\pm 0.000}$	$0.464_{\pm 0.038}$	$0.021_{\pm 0.101}$
FDR-CE (↓)	$0.490_{\pm 0.038}$	$\mathbf{0.081_{\pm 0.184}}$	$0.492_{\pm 0.038}$	$\mathbf{0.215_{\pm 0.191}}$	$0.447_{\pm 0.032}$	$0.140_{\pm 0.344}$	$0.466_{\pm 0.040}$	$\mathbf{0.021_{\pm 0.101}}$
EFFICIENCY (↑)	$1.000_{\pm 0.000}$	$0.010_{\pm 0.023}$	$1.000_{\pm 0.000}$	$0.076_{\pm 0.077}$	$1.000_{\pm 0.000}$	$0.002_{\pm 0.004}$	$1.000_{\pm 0.000}$	$0.007_{\pm 0.034}$
HUMANEVAL-F								
1 - PASS@1 (↓)	$0.341_{\pm 0.069}$	$0.136_{\pm 0.131}$	$0.290_{\pm 0.072}$	$0.123_{\pm 0.142}$	$0.230_{\pm 0.074}$	$0.000_{\pm 0.000}$	$0.254_{\pm 0.070}$	$0.026_{\pm 0.102}$
FDR-CE (↓)	$0.330_{\pm 0.064}$	$\mathbf{0.134_{\pm 0.127}}$	$0.287_{\pm 0.074}$	$\mathbf{0.114_{\pm 0.136}}$	$0.235_{\pm 0.082}$	$0.120_{\pm 0.325}$	$0.247_{\pm 0.072}$	$\mathbf{0.025_{\pm 0.100}}$
EFFICIENCY (↑)	$1.000_{\pm 0.000}$	$0.275_{\pm 0.118}$	$1.000_{\pm 0.000}$	$0.125_{\pm 0.130}$	$1.000_{\pm 0.000}$	$0.004_{\pm 0.012}$	$1.000_{\pm 0.000}$	$0.065_{\pm 0.234}$

Python programming question is provided to an LLM as a prompt to generate code, and then the generated code is measured for its functionality by running the unit tests.

We construct APPS-f, Mercury-f, HumanEval-f, and MBPP-f, where for each dataset, we replace the built in unit tests from each problem to automatically generated unit tests via fuzzing for both learning and evaluation. In particular, each question consists of constraints to inputs. We manually post-processed python programming questions and their solutions of each datasets such that the solution code can be easily callable by a fuzzing tool while satisfying the input constraints of questions. Among the post-processed code, we conducted fuzzing and extracted at least 600 input-output pairs as our unit tests for calibration and its evaluation. Additional details can be found on Appendix G.

LLMs. We used three closed LLMs, *i.e.,* GPT-4o [40], Gemini 1.5 Pro [41], GPT-4.1 [42] and two open LLM, *i.e.,* CodeLlama 13B-instruct [43] and Deepseek-R1 [8]. Here, we use the following default parameters unless specified: $\varepsilon_S = 0.3$, $\delta_S = 0.1$, $\alpha = 0.35$, $\varepsilon_E = 0.05$, and $n_{\text{max}} = 150$.

Method. We consider seven baseline methods **SCG-EM** [13], **SCG-manual**, **SCG-small**, **SCG-H**, **CodeT** [37], **LDB** [38], and **SFS** [39] to compare with our method **SCG**.

- **SCG-EM [13]**: This baseline is a conventional selective predictor method that compares the generated code and solution code, without measuring its functional correctness.
- **SCG-manual**: This baseline is a simple selective generator that selects the upper $k\%$ of scores for a threshold τ to highlight the importance of controlling the FDR-CE.
- **SCG-small**: This is our method but only using unit tests, provided by the APPS dataset to show the efficacy of generated unit tests via fuzzing. We sampled 21 test cases for each problem to apply our algorithm. See Appendix E for additional details.

Table 3: Comparison results of SCG against baseline methods on different programming languages with GPT-4.1 ($\alpha = 0.35, \delta_S = 0.1, \varepsilon_E = 0.05, \varepsilon_S = 0.3$ for all programming languages). We selected a subset of problems (5359 instances) from the APPS-f dataset that conform to the standard input/output format for executing unit tests. The FDR-CE satisfying the desired guarantees and the highest efficiency among methods that comply with the FDR-CE guarantees are marked in **bold.**

Methods	w/o Selective Generation	w/ Selective Generation				
	$\tau = -\infty$	SCG-EM	SCG-MANUAL	SCG-SMALL	SCG-H	**SCG**
CPP ($\varepsilon_S = 0.3$)						
1 - PASS@1 (↓)	$0.386_{\pm 0.012}$	$0.000_{\pm 0.000}$	$0.228_{\pm 0.018}$	$0.226_{\pm 0.019}$	$0.278_{\pm 0.021}$	$0.232_{\pm 0.019}$
FDR-CE (↓)	$0.382_{\pm 0.013}$	$\mathbf{0.000_{\pm 0.000}}$	$\mathbf{0.225_{\pm 0.018}}$	$\mathbf{0.225_{\pm 0.017}}$	$0.270_{\pm 0.021}$	$\mathbf{0.228_{\pm 0.019}}$
EFFICIENCY (↑)	$1.000_{\pm 0.000}$	$0.000_{\pm 0.001}$	$0.499_{\pm 0.018}$	$0.498_{\pm 0.017}$	$0.680_{\pm 0.020}$	$\mathbf{0.513_{\pm 0.019}}$
JAVA ($\varepsilon_S = 0.3$)						
1 - PASS@1 (↓)	$0.383_{\pm 0.013}$	$0.000_{\pm 0.000}$	$0.240_{\pm 0.017}$	$0.228_{\pm 0.019}$	$0.286_{\pm 0.020}$	$0.227_{\pm 0.020}$
FDR-CE (↓)	$0.379_{\pm 0.013}$	$\mathbf{0.000_{\pm 0.000}}$	$\mathbf{0.237_{\pm 0.017}}$	$\mathbf{0.223_{\pm 0.019}}$	$0.282_{\pm 0.019}$	$\mathbf{0.222_{\pm 0.022}}$
EFFICIENCY (↑)	$1.000_{\pm 0.000}$	$0.000_{\pm 0.000}$	$\mathbf{0.502_{\pm 0.019}}$	$0.483_{\pm 0.019}$	$0.672_{\pm 0.021}$	$0.485_{\pm 0.018}$
JAVASCRIPT ($\varepsilon_S = 0.3$)						
1 - PASS@1 (↓)	$0.374_{\pm 0.014}$	$0.000_{\pm 0.000}$	$0.221_{\pm 0.016}$	$0.227_{\pm 0.025}$	$0.278_{\pm 0.020}$	$0.227_{\pm 0.022}$
FDR-CE (↓)	$0.368_{\pm 0.014}$	$\mathbf{0.000_{\pm 0.000}}$	$\mathbf{0.215_{\pm 0.016}}$	$\mathbf{0.223_{\pm 0.025}}$	$\mathbf{0.274_{\pm 0.018}}$	$\mathbf{0.222_{\pm 0.023}}$
EFFICIENCY (↑)	$1.000_{\pm 0.000}$	$0.000_{\pm 0.000}$	$0.500_{\pm 0.018}$	$0.518_{\pm 0.020}$	$\mathbf{0.673_{\pm 0.015}}$	$0.517_{\pm 0.020}$
PERL ($\varepsilon_S = 0.3$)						
1 - PASS@1 (↓)	$0.457_{\pm 0.015}$	$0.000_{\pm 0.000}$	$0.305_{\pm 0.017}$	$0.205_{\pm 0.034}$	$0.274_{\pm 0.029}$	$0.218_{\pm 0.037}$
FDR-CE (↓)	$0.453_{\pm 0.015}$	$\mathbf{0.000_{\pm 0.000}}$	$0.302_{\pm 0.016}$	$\mathbf{0.207_{\pm 0.033}}$	$0.270_{\pm 0.027}$	$\mathbf{0.207_{\pm 0.035}}$
EFFICIENCY (↑)	$1.000_{\pm 0.000}$	$0.000_{\pm 0.001}$	$0.499_{\pm 0.019}$	$0.182_{\pm 0.031}$	$0.397_{\pm 0.028}$	$\mathbf{0.186_{\pm 0.029}}$

- **SCG-H**: This baseline is a heuristic of our method omitting false entailment rate (FER) as in Lemma 1. It searches for a selective generator as in (4) with $\varepsilon_E = 0$.

Scoring Function. To analyze the effect of calibration on our method **SCG**, we consider four different scoring functions. The detailed explanations on the scoring functions are provided in Appendix F.

Evaluation. We evaluate our method along with baselines based on the empirical counterpart of the FDR-CE and selection efficiency from a test set $\mathbf{Z}_{\mathrm{t}} \sim \mathcal{D}^{n_{\text{test}}}$ in (8) and (9), respectively. Interestingly, FDR-CE and 1-PASS@1 [19] over on selected samples by **SCG** may be asymptotically equivalent when $\alpha \to 0$ and $n_{\mathbf{y}} \to \infty$ for PASS@1 (See Appendix K for discussion).

4.2 Results

We demonstrate the efficacy of our method **SCG** on different models, datasets, and programming languages. In addition, we highlight the benefits of fuzzing, and analyze the effect of calibration.

4.2.1 Controllability and Selection Efficiency

Figure 2 shows that **SCG** controls the FDR-CE. To this end, we conducted random experiments. We ran the experiment 50 times by randomly splitting the calibration set and test set at 8:2 ratio each time. The whisker on each box plot denotes the range between δ_S and $1 - \delta_S$ percentile of the distributions.

SCG-manual may perform better than our method depending on the choice of k. However, manually selecting an appropriate k for different situations is a challenging task. **SCG-small** bounds the FDR-CE successfully. However, this method demonstrates lower efficiency compared to our method. The result stems from a lack of unit tests to correctly infer *expected functional correctness*, illustrating the advantage of fuzzing in learning. **SCG-H** shows that it is not able to bound a desired FDR-CE, as it ignores an estimation error for inferring expected functional correctness.

Our method shows how it successfully bounds desired FDR-CEs on diverse models (Figure 2), diverse parameters (Figure 3, Figure 5), diverse datasets (Table 1), diverse methods (Table 2), and diverse programming languages (Table 3). In Figure 2, 3, and 5, this is shown by upper whisker bar lying below the desired FDR-CE in dotted line, while Table 1, 2, 3, the results are indicated by bold text. Table 5 shows qualitative results of our method that accepts correct code and rejects uncertain code. Note that poorly performing model, *e.g.,* CodeLlama in Figure 4(a), may not find a selective generator with a desired FDR-CE ε_S. This is an expected result due to an un-calibrated scoring function [16], as discussed in Section 4.2.4.

4.2.2 Benefit of Fuzzing in Learning and Evaluation

We empirically show the benefit of fuzzing in learning. As shown in Figure 5(b), the FDR-CE bound (in the top of whisker) gets tighter to a desired FDR-CE level *without violating it* as smaller ε_E

requires a larger amount of unit tests, thus providing a precise estimation on expected functional correctness. This shows that generated unit tests by fuzzing helps to provide a tighter FDR-CE guarantee, meaning higher selection efficiency.

Additionally, we demonstrate that automatic unit test generation is beneficial in rigorous evaluation. As shown in Figure 5(c), the FDR-CE decreases as the number of unit tests for evaluation increases. Recalling that α-entailment is determined by comparing the lower bound of expected functional correctness with $1 - \alpha$ as in Definition 1, the lower bound gets tighter as we use more unit tests to evaluate code. The tighter lower bound results in more accurate comparison and evaluation, thus reducing the FDR-CE. This shows that fuzzing has a benefit of reducing the evaluation error.

4.2.3 Comparison and Extension over Prior Work

We compare our method **SCG**, with prior code generation baselines and demonstrate that these baselines can be extended through our approach to provide additional functionality guarantees. As further discussed in Appendix A, improvements in functionality – inherently reducing *functional hallucination* – can be broadly categorized into two approaches: (1) enhancing the model performance during training or fine-tuning [9, 8] or (2) employing post-filtering methods [37, 38, 39]. In this work, we focus on comparison with post-filtering methods, as **SCG** post-processes the generated code to control hallucinations but training and finetuning based methods are orthogonal to this work.

Table 4: FDR-CE and efficiency for DeepSeek-R1 [8]

ε_S	FDR-CE	Efficiency
wo/ SCG		
-	$0.234_{\pm 0.012}$	$1.000_{\pm 0.000}$
w/ SCG		
0.15	$0.043_{\pm 0.048}$	$0.076_{\pm 0.085}$
0.20	$0.132_{\pm 0.015}$	$0.515_{\pm 0.032}$
0.25	$0.183_{\pm 0.016}$	$0.778_{\pm 0.020}$
0.30	$0.233_{\pm 0.011}$	$0.995_{\pm 0.004}$

Table 2 compares **SCG** with prior post-filtering methods. In particular, **SCG** controls the hallucination and achieves lower FDR-CE than prior methods depending on the choice of ε_S. Furthermore, Table 2, 4, Figure 3(a), demonstrate the applicability of our method in a model- and method-agnostic manner across diverse experiment setups, including integration with finetuning methods (*e.g.,* DeepSeek GRPO, OpenAI RLHF) and compatibility with post-processing methods.

4.2.4 Effect of Scoring Function and Calibration

We demonstrate the effect of calibration on our method **SCG**. As shown in Figure 3(c), the choice of scoring function affects whether FDR-CE can be successfully bounded to the desired ε_S. In particular, f_{verb} in Figure 3(c) fails to properly bound the FDR-CE. Furthermore, CodeLlama in Figure 4(a), fails to find a selective generator with a desired ε_S, due to an un-calibrated scoring function [16]. Thus, the model finds a minimum FDR-CE by returning $\hat{U}$ in this case. These results underscore the importance of selecting appropriate scoring functions for the efficacy of our method.

5 Conclusion

This paper considers the code hallucination problem. In particular, we define the concept of code entailment based on automatically generated unit tests via fuzzing, one of dynamic code analysis tools. Given this, we propose a learning algorithm for selective code generators to theoretically control the hallucination in the FDR of selective generators. We further leverage fuzzing to automatically generate unit tests for learning and evaluation purposes, enabling the large-scale collection of unit tests. Lastly, we demonstrate the controllability of the proposed selective generator and its selection efficiency over open and closed code generators under different experiment setups.

Limitations. The proposed method controls the rate of hallucination, but its selection efficiency heavily depends on the quality of code generation models, requiring improvement for code generators. Moreover, the i.i.d. assumption for the FDR controllability guarantee limits its applicability in distribution-shifting environments.

Acknowledgment

We appreciate constructive feedback by anonymous reviewers. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) and the National

Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2019-II191906, Artificial Intelligence Graduate School Program (POSTECH) (25%); RS-2024-00457882, National AI Research Lab Project (25%); No. RS-2024-00509258 and No. RS-2024-00469482, Global AI Frontier Lab (25%); RS-2025-00560062 (25%)).

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher 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, 2020.

[2] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[3] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[4] Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. Large language models for mathematical reasoning: Progresses and challenges. In Neele Falk, Sara Papi, and Mike Zhang, editors, *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 225–237, St. Julian's, Malta, March 2024. Association for Computational Linguistics.

[5] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023.

[6] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. A deep dive into large language models for automated bug localization and repair. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[7] Taesoo Kim, HyungSeok Han, Soyeon Park, Dae R. Jeong, Dohyeok Kim, Dongkwan Kim, Eunsoo Kim, Jiho Kim, Joshua Wang, Kangsu Kim, Sangwoo Ji, Woosun Song, Hanqing Zhao, Andrew Chin, Gyejin Lee, Kevin Stevens, Mansour Alharthi, Yizhuo Zhai, Cen Zhang, Joonun Jang, Yeongjin Jang, Ammar Askar, Dongju Kim, Fabian Fleischer, Jeongin Cho, Junsik Kim, Kyungjoon Ko, Insu Yun, Sangdon Park, Dowoo Baik, Haein Lee, Hyeon Heo, Minjae Gwon, Minjae Lee, Minwoo Baek, Seunggi Min, Wonyoung Kim, Yonghwi Jin, Younggi Park, Yunjae Choi, Jinho Jung, Gwanhyun Lee, Junyoung Jang, Kyuheon Kim, Yeonghyeon Cha, and Youngjoon Kim. Atlantis: Ai-driven threat localization, analysis, and triage intelligence system, 2025.

[8] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[9] OpenAI, :, Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, Jerry Tworek, Lorenz Kuhn, Lukasz Kaiser, Mark Chen, Max Schwarzer, Mostafa Rohaninejad, Nat McAleese, o3 contributors, Oleg Mürk, Rhythm Garg, Rui Shu, Szymon Sidor, Vineet Kosaraju, and Wenda Zhou. Competitive programming with large reasoning models, 2025.

[10] Potsawee Manakul, Adian Liusie, and Mark Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[11] Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. In *The Eleventh International Conference on Learning Representations*, 2023.

[12] Vladimir Vovk, Alex Gammerman, and Glenn Shafer. *Algorithmic learning in a random world*. Springer Science & Business Media, 2005.

[13] Yonatan Geifman and Ran El-Yaniv. Selective classification for deep neural networks. *Advances in neural information processing systems*, 30, 2017.

[14] Victor Quach, Adam Fisch, Tal Schuster, Adam Yala, Jae Ho Sohn, Tommi S. Jaakkola, and Regina Barzilay. Conformal Language Modeling, June 2024. arXiv:2306.10193 [cs].

[15] Christopher Mohri and Tatsunori Hashimoto. Language models with conformal factuality guarantees. *arXiv preprint arXiv:2402.10978*, 2024.

[16] Minjae Lee, Kyungmin Kim, Taesoo Kim, and Sangdon Park. Selective generation for controllable language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[17] Samuel Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 632–642, 2015.

[18] Adina Williams, Nikita Nangia, and Samuel R Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of NAACL-HLT*, pages 1112–1122, 2018.

[19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[20] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[21] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multiple: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*, 2022.

[22] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[23] Dan Hendrycks, Steven Basart, Saurav Kadavath, and et al. Measuring coding challenge competence with apps, 2021.

[24] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 185–196, New York, NY, USA, 2024. Association for Computing Machinery.

[25] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[26] Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. Semcoder: Training code language models with comprehensive semantics reasoning. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 60275–60308. Curran Associates, Inc., 2024.

[27] Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 16601–16622. Curran Associates, Inc., 2024.

[28] Henry Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, page 122–126, New York, NY, USA, 1987. Association for Computing Machinery.

[29] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.

[30] William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10:100–107, 1998.

[31] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in ssl/tls implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 793–804, New York, NY, USA, 2015. Association for Computing Machinery.

[32] Marie-Christine Jakobs and Maik Wiesner. Peqtest: Testing functional equivalence. In Einar Broch Johnsen and Manuel Wimmer, editors, *Fundamental Approaches to Software Engineering*, pages 184–204, Cham, 2022. Springer International Publishing.

[33] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 712–717, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[34] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 349–360, New York, NY, USA, 2014. Association for Computing Machinery.

[35] Charles J Clopper and Egon S Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934.

[36] Google. Atheris: A coverage-guided, native python fuzzer, 2020.

[37] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*, 2023.

[38] Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 851–870, 2024.

[39] Jonathan Light, Yue Wu, Yiyou Sun, Wenchao Yu, Yanchi Liu, Xujiang Zhao, Ziniu Hu, Haifeng Chen, and Wei Cheng. SFS: Smarter code space search improves LLM inference scaling. In *The Thirteenth International Conference on Learning Representations*, 2025.

[40] OpenAI. Gpt-4 technical report, 2024.

[41] Gemini Team, Petko Georgiev, and Ving Ian Lei et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.

[42] OpenAI. Introducing gpt-4.1 in the api, 2025.

[43] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[44] Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. Codehalu: Investigating code hallucinations in llms via execution-based verification, 2024.

[45] Ziyao Zhang, Yanlin Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation, 2025.

[46] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madan Musuvathi, and Shuvendu Lahiri. Exploring the effectiveness of llm based test-driven interactive code generation: User study and empirical evaluation. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 390–391, New York, NY, USA, 2024. Association for Computing Machinery.

[47] Sarah Fakhoury, Markus Kuppe, Shuvendu K. Lahiri, Tahina Ramananandro, and Nikhil Swamy. 3dgen: Ai-assisted generation of provably correct binary format parsers. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2535–2547, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[48] Baizhou Huang, Shuai Lu, Xiaojun Wan, and Nan Duan. Enhancing large language models in coding through multi-perspective self-consistency. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1429–1450, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[49] Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. S*: Test time scaling for code generation, 2025.

[50] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[51] Semmle. Codeql, 2019.

[52] Michal Zalewski. Technical "whitepaper" for afl-fuzz. *URl: http://lcamtuf. coredump. cx/afl/technical details. txt*, 2014.

[53] Katherine Tian, Eric Mitchell, Allan Zhou, Archit Sharma, Rafael Rafailov, Huaxiu Yao, Chelsea Finn, and Christopher Manning. Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5433–5442, Singapore, December 2023. Association for Computational Linguistics.

[54] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Prem Devanbu, and Toufique Ahmed. Calibration and correctness of language models for code, 2024.

A Extended Related Work

Code Generation and Hallucination. Code generation is the task of generating a program that satisfies given functional specifications. The main challenge for this task is the generation of functionally incorrect code, often referred to as *code hallucination* in the previous literature [44, 45].

Recent work mostly focused on improving the functionality itself. DeepSeek-R1 [8], OpenAI-IOI [9] leverages reinforcement learning based finetuning, resulting in highly performant code generation model. CodeT [37], TiCoder [46], 3DGen [47], MPSC [48], S*[49], and CodeRL [50] rank generated solutions during the inference or evaluation phase to determine which outputs to adopt. AlphaCode [2] leverages both finetuning and a ranking mechanism to improve performance. The key distinction of our work lies in directly controlling the rate of falsely generated code (FDR-CE), making the underlying code generator more trustworthy. Furthermore, our work is applicable on top of each method providing an upper bound on the FDR-CE (*e.g.,* DeepSeek result in Figure 2(c) shows that our method can provide additional statistical guarantee when applied on top of GRPO based finetuning [8]).

The methods mentioned above leveraged unit tests to improve performance. The ranking-based method [37, 46, 47, 48, 49, 50] leverage unit tests to rank sampled solutions and select the best code, whereas finetuning based methods [8, 9] leverage unit tests to provide execution feedback for reinforcement learning. Unlike prior work, our method leverages unit tests to determine α-code entailment and applying selective generation learning algorithm.

B Preliminary

We introduce preliminaries on textual entailment for measuring correctness between two answers, selective generation to control the rate of hallucination, and dynamic code analysis via fuzzing for learning and evaluating functional correctness of code snippets.

Textual Entailment. In natural languages, textual entailment is a concept of evaluating semantic relation between two sentences by checking an entailment relation [17]. In particular, denoting two sentences by a premise and a hypothesis, we say that a premise entails a hypothesis if the hypothesis is true given the premise. Otherwise, we say the premise contradicts the hypothesis.

This textual entailment has been used to measure semantic correctness between a question and an answer in learning language models [15, 16]. If a generated answer entails a true one, we can consider the generated answer as true in textual entailment. In evaluating correctness of generated answers in natural language processing, introducing textual entailment is crucial as a simple, traditional exact match, *i.e.,* the generated answer and the true one is exactly the same, does not measure the semantic relation between two answers. However, in code generation, there is no notion of entailment to check the semantic correctness between two code snippets. We overcome this hurdle by introducing *code entailment* for measuring functional correctness, leveraging executable properties of code.

Dynamic Code Analysis via Fuzzing. Computer programs suffer from undefined behaviors, called *bugs*, *e.g.,* crash by buffer overflow. To find the bugs, security researchers have been extensively developed static and dynamic code analysis tools, *e.g.,* `CodeQL` [51], `AFL` [52], and `Atheris` [36].

The dynamic analysis tools exploit the executable property of code to find bugs, while the static analysis tools inspect code without execution. We mainly focus on more informative dynamic analysis tools. In particular, fuzzing, a representative class of methods for dynamic code analysis, generates the input of a given program, called seed, executes the program with the input, and checks whether undefined behaviors can be observable. Given the observation, fuzzing methods randomly mutate the input of programs to explore wider execution paths or exploit execution paths toward targeted code. In this paper, we re-purpose fuzzing methods for identifying functionality of code, instead of finding bugs in code by generating the input and output pairs of code.

Selective Generation. Language models suffer from generating hallucinated facts. Recently, certified ways to control the rate of hallucination in language models are proposed [14, 15, 16]. Among them, selective generation, which extends traditional selective classification [13], provides a way to control the rate of hallucination defined in terms of a false discovery rate with entailment (FDR-E) that leverages textual entailment to measure the correctness of two answers. In particular, a selective generator $\hat{S}(\mathbf{x})$ given a question $\mathbf{x}$ returns a generated answer $G(\mathbf{x})$ from a language model or abstains from answering by returning "I don't know" (`IDK`). The FDR-E of this selective generator with respect to a true answer $\mathbf{y}$ is defined as $\mathcal{R}(\hat{S}) \coloneqq \mathbb{P}\{\hat{S}(\mathbf{x}) \notin E(\mathbf{y}) \mid \hat{S}(\mathbf{x}) \neq \texttt{IDK}\}$. Here, E is an entailment set that contains entailing answers, *i.e.,* $E(\mathbf{y}) \coloneqq \{\bar{\mathbf{y}} \mid \bar{\mathbf{y}} \text{ entails } \mathbf{y}\}$ (where a reverse relation is also valid), so $\hat{S}(\mathbf{x}) \in E(\mathbf{y})$ means that $\hat{S}(\mathbf{x})$ entails $\mathbf{y}$. In semi-supervised selective generation [16], a learning algorithm leverages an estimated entailment set $\hat{E}$ learned from the handful of entailment labels, where $\hat{E}$ is used as a pseudo-labeling function for entailment labels. Based on the estimated entailment set, the following surrogate of the FDR-E is considered: $\hat{\mathcal{R}}(\hat{S}) \coloneqq \mathbb{P}\{\hat{S}(\mathbf{x}) \notin \hat{E}(\mathbf{y}) \mid \hat{S}(\mathbf{x}) \neq \texttt{IDK}\}$. The algorithm leverages the relation between $\mathcal{R}(\hat{S})$ and $\hat{\mathcal{R}}(\hat{S})$ to bound $\mathcal{R}(\hat{S})$, as shown in the following lemma.

Lemma 2. *[16] $\mathcal{R}(\hat{S})$ is decomposed in $\mathcal{R}(\hat{S}) \coloneqq \mathbb{P}_{\mathcal{D}_{\hat{S}}}\{e = 0, \hat{e} = 1\} - \mathbb{P}_{\mathcal{D}_{\hat{S}}}\{e = 1, \hat{e} = 0\} + \hat{\mathcal{R}}(\hat{S})$, where $\mathbb{P}_{\hat{S}}\{\cdot\} \coloneqq \mathbb{P}\{\cdot \mid \hat{S}(\mathbf{x}) \neq \texttt{IDK}\}$, $e \coloneqq \mathbb{1}(\hat{S}(\mathbf{x}) \in E(\mathbf{y}))$, and $\hat{e} \coloneqq \mathbb{1}(\hat{S}(\mathbf{x}) \in \hat{E}(\mathbf{y}))$.*

By controlling the upper bound of three decomposed terms in $\mathcal{R}(\hat{S})$ at a desired level, the algorithm learns a selective generator $\hat{S}$. We leverage this framework to learn $\hat{E}$ via generated unit tests via dynamic code analysis tools.

C Qualitative Result of our Methods

Table 5: Qualitative results of our selective code generator with GPT-4o ($\varepsilon_S = 0.3$, $\delta_S = 0.1$, $\varepsilon_E = 0.05$, and $\alpha = 0.15$). The accepted generated code is α-entailment with the score of -0.05, meaning high certainty, where the code is also semantically correct. The rejected code is not α-entailment with the score of -53.87, meaning low certainty probably due to the complexity of the question. In particular, it returns a wrong answer, 2, for an input, [1,1,0,1,0,0,0,1,0,1,0,0,1,1,0,1], where an expected answer is 14, due to the inequality in "`count > max_count`" that does not follow the tie instruction. Note that the contents are manually truncated for presentation purposes.

Question	**Canonical Solution**	**Generated Code**	**Selection** (ours)

You are given an array to sort but you're expected to start sorting from a specific position of the array (in ascending order) and optionally you're given the number of items to sort.	`def sect_sort(lst, start, length=0):`
	`  end = start + length \`
	`  if length else \`
	`  len(lst)`
	`  return lst[:start] + \`
	`  sorted(lst[start:end]) + \`
	`  lst[end:]`
		`def sect_sort(array, start, length=0):`
		`  if length == 0:`
		`  end = len(array)`
		`  else:`
		`  end = start + length`
		
		`  array[start:end] = \`
		`  sorted(array[start:end])`
		`  return array`
			Accept

Given an array containing only zeros and ones, find the index of the zero that, if converted to one, will make the longest sequence of ones. ... Return the last index if there are multiple results.	`def replace_zero(arr):`
	`  m,im,i,lst = 0, -1, -1, ('').join`
	`  (map(str,arr)).split('0'))`
	`  for a,b in zip(lst,lst[1:]):`
	`  i += len(a) + 1`
	`  candidate = len(a)+len(b)+1`
	`  if m <= candidate:`
	`  im, m = i, candidate`
	`  return im`
		`def replace_zero(arr):`
		`  max_count = 0`
		`  max_index = -1`
		`  count = 0`
		`  last_zero_idx = -1`
		`  consecutive_count_before_zero = 0`
		
		`  for i in range(len(arr)):`
		`  if arr[i] == 1:`
		`  count += 1`
		`  else:`
		`  count = i - last_zero_idx`
		`  last_zero_idx = i`
		
		`  if count > max_count:`
		`  max_count = count`
		`  max_index = last_zero_idx`
		
		`  return max_index`
			Reject

D Algorithm

Algorithm 1 Unit tests size computation.

procedure COMPUNITTESTSIZE($\alpha, \varepsilon_E, \mathbf{y}, \bar{\mathbf{y}}, n_{\text{max}}$)
$n_{\mathbf{y}} \leftarrow 0$
while $\hat{L}(\mathbf{y}, \bar{\mathbf{y}}, n_{\mathbf{y}}, \varepsilon_E) < 1 - \alpha$ **do**
  $\mathbf{s} \sim \mathcal{D}_{\mathcal{U}}$
  $(\mathbf{u}, \mathbf{v}) \leftarrow \mathcal{F}(\mathbf{y}, \mathbf{s})$
  $n_{\mathbf{y}} \leftarrow n_{\mathbf{y}} + 1$
  if $n_{\mathbf{y}} \geq n_{\text{max}}$ **then**
    break ▷ break for infeasible cases
  end if
end while
return $n_{\mathbf{y}}$

Algorithm 2 Selective Code Generator Learning

procedure LEARNSCG$(f, \mathbf{Z}, G, \alpha, \varepsilon_E, \delta_S)$
$\mathbf{Z}' \leftarrow \text{SORT}_f(\mathbf{Z})$ ▷ Increasing order of $f(\mathbf{x}_i, G(\mathbf{x}_i))$
$(\underline{i}, \bar{i}) \leftarrow (1, |\mathbf{Z}|)$
for $i = 1$ to $\lceil \log_2 |\mathbf{Z}| \rceil$ **do**
  $i_{\text{mid}} \leftarrow \lceil \frac{\underline{i}+\bar{i}}{2} \rceil$
  $\tau_S^{(i)} \leftarrow f(\mathbf{x}_{i_{\text{mid}}}, G(\mathbf{x}_{i_{\text{mid}}}))$ ▷ Choose a candidate τ
  $\hat{\mathbf{Z}}^{(i)} \leftarrow \left\{(\mathbf{x}, _) \in \mathbf{Z}' \mid f(\mathbf{x}, G(\mathbf{x})) \geq \tau_S^{(i)}\right\}$ ▷ Build a selected calibration set by $\tau_S^{(i)}$
  $\hat{k}^{(i)} \leftarrow \sum_{(\mathbf{x},\mathbf{y},_) \in \hat{\mathbf{Z}}^{(i)}} \mathbb{1}\left(G(\mathbf{x}) \notin \hat{E}_{\alpha,\varepsilon_E}(\mathbf{y})\right)$ ▷ Count false discoveries.
  $\hat{U}^{(i)} \leftarrow \hat{U}_{\text{Binom}}\left(\hat{k}^{(i)}, |\hat{\mathbf{Z}}^{(i)}|, \delta_S / \lceil \log_2 |\mathbf{Z}| \rceil\right)$ ▷ Bound the FDR-CE.
  if $\varepsilon_E + \hat{U}^{(i)} \leq \varepsilon_S$ **then**
    $\bar{i} \leftarrow i_{\text{mid}}$ ▷ Keep $\tau_S^{(i)}$ that controls a desired FDR-CE.
  else
    $\underline{i} \leftarrow i_{\text{mid}}$
  end if
end for
return $\tau_S^{(i)}, \varepsilon_E + \hat{U}^{(i)}$

E SCG-small Detail

This is our method but only using unit tests, provided by the APPS dataset to show the efficacy of generated unit tests via fuzzing. In particular, the APPS dataset provides an average of 21 unit tests per problem. However, the number of unit tests varies across the APPS dataset, and some problems lack unit tests. As it is difficult to directly apply our method to the original dataset, we sampled 21 test cases for each problem from generated unit tests via fuzzing to apply our algorithm. To support the validity of this baseline method, we provide additional experiments on Appendix M.3 regarding the quality of unit tests. Lastly, it is worth noting that this baseline is only applicable to APPS dataset, due to insufficient number of unit tests in other datasets to determine code entailment and obtain meaningful results.

F Scoring Functions

Here, we denote p_i as the probability by a code generator G to generate the i-th token.

- **Length-normalized log-probability** $f_{\textbf{norm}}$: This method collects the log-probability each token used to generate a code then normalize by the number of tokens, *i.e.,* $f_{\text{norm}}(\mathbf{x}, G(\mathbf{x})) = \frac{\sum_i \ln p_i}{|G(\mathbf{x})|}$. We use f_{norm} as the default scoring function unless specified.
- **Lowest log-probability** f_{min}: This method collects the log-probability of each generated tokens then selects the lowest value, *i.e.,* $f_{\text{min}}(\mathbf{x}, G(\mathbf{x})) = \min_i \ln p_i$.
- **Sequence log-probability** $f_{\textbf{seq}}$: This method collects the log-probability of each token to calculate the probability of generated code, *i.e.,* $f_{\text{seq}}(\mathbf{x}, G(\mathbf{x})) = \sum_i \ln p_i$.
- **Verbalized probability** [53], [54] f_{verb}: The model used for code generation is prompted to produce a confidence value, with the prompt $\mathbf{p}_{\textbf{conf}}(G(x))$, *i.e.,* $f_{\text{verb}}(x, G(x)) = G(\mathbf{p}_{\text{conf}}(G(x)))$.

G Datasets and Models

We use four datasets, APPS[23], MBPP[20], HumanEval[19], and Mercury[27], for calibration and evaluation. We use five large language models (LLMs), *GPT-4o*, *GPT-4.1*, *Gemini-1.5 Pro*, *CodeLlama 13B Instruct*, and *DeepSeek-R1* for code generation.

To determine code entailment of the generated code, we specifically selected datasets that provide a solution to the problem. For each problem, unit tests were generated using a dynamic analysis tool. The following Table 6 shows the result of the dynamic code analysis for each datasets.

To enhance the efficiency of execution path exploration, we manually post-processed each solution on the datasets for compatibility with the dynamic analysis tool. Although manual post-processing is not a mandatory step, it results in additional computational time and overhead for dynamic code analysis. During this step, we also excluded problems either lacking solution or containing errors.

Table 6: The result of dynamic analysis for each dataset. We excluded problems that had errors in the original problem or with solutions that were difficult to analyze dynamically.

Dataset	Original Dataset	Solution-problem Error	Dynamic Analysis Error (*e.g.,* `Atheris` timeout)
APPS	10,000	1,274	307
MBPP	974	-	35
HumanEval	164	-	1
Mercury	1,889	-	279

H Proof of Lemma 1

Recall that $e \coloneqq \mathbb{1}(G(\mathbf{x}) \in E_\alpha(\mathbf{y}))$, $\hat{e} \coloneqq \mathbb{1}(G(\mathbf{x}) \in \hat{E}_{\alpha,\varepsilon_E}(\mathbf{x}))$, where e denotes whether $\mathbf{y}$ α-*entails* $G(\mathbf{x})$ and and $\hat{e}$ denotes the predicted result of α-entailment.

Additionally, we have the following due to Lemma 2:

$$\begin{aligned}\mathcal{R}_\alpha(\hat{S}) &= \mathbb{P}\{e=0, \hat{e}=1 \mid \hat{S}(\mathbf{x}) \neq \texttt{IDK}\} - \mathbb{P}\{e=1, \hat{e}=0 \mid \hat{S}(\mathbf{x}) \neq \texttt{IDK}\} + \mathcal{R}_{\alpha,\varepsilon_E}(\hat{S}) \\ &\leq \mathbb{P}\{e=0, \hat{e}=1 \mid \hat{S}(\mathbf{x}) \neq \texttt{IDK}\} + \mathcal{R}_{\alpha,\varepsilon_E}(\hat{S}),\end{aligned}$$

where the probability is taken over $\mathbf{x}$, $\mathbf{y}$, $n_\mathbf{y}$, and $\mathbf{S}_\mathbf{y}$.

We, then, bound the first term $\mathbb{P}\{e=0, \hat{e}=1 \mid \hat{S}(\mathbf{x}) \neq \texttt{IDK}\}$. In particular, suppose that $\mathbf{x}$, $\mathbf{y}$, and $n_\mathbf{y}$ which satisfies $e=0$ and $\hat{S}(\mathbf{x}) \neq \texttt{IDK}$ are given. Here, $n_\mathbf{y}$ is determined using Algorithm 1 and recall that $S_\mathbf{y} \sim D_\mathbf{u}^{n_\mathbf{y}}$ denoting $n_\mathbf{y}$ input-output pairs from dynamic code analysis tool $\mathcal{F}$.

Then, we have the following:

$$\begin{aligned}&\mathbb{P}_{\mathbf{S}_\mathbf{y}}\left\{\hat{e}=1 \;\middle|\; e=0, \hat{S}(\mathbf{x}) \neq \texttt{IDK}\right\} \\ &\quad= \mathbb{P}_{\mathbf{S}_\mathbf{y}}\left\{G(\mathbf{x}) \in \hat{E}_{\alpha,\varepsilon_E}(\mathbf{y}) \;\middle|\; G(\mathbf{x}) \notin E_\alpha(\mathbf{y})\right\} \\ &\quad= \mathbb{P}_{\mathbf{S}_\mathbf{y}}\left\{\hat{L}(\mathbf{y}, G(\mathbf{x}), n_\mathbf{y}, \varepsilon_E) \geq 1-\alpha \;\middle|\; \mathbb{P}_\mathbf{y}\{G(\mathbf{x})(\mathbf{u}) = \mathbf{v}\} < 1-\alpha\right\} \\ &\quad\leq \mathbb{P}_{\mathbf{S}_\mathbf{y}}\left\{\hat{L}(\mathbf{y}, G(\mathbf{x}), n_\mathbf{y}, \varepsilon_E) > \mathbb{P}_\mathbf{y}\{G(\mathbf{x})(\mathbf{u}) = \mathbf{v}\} \;\middle|\; \mathbb{P}_\mathbf{y}\{G(\mathbf{x})(\mathbf{u}) = \mathbf{v}\} < 1-\alpha\right\} \\ &\quad\leq \varepsilon_E,\end{aligned}$$

where the last inequality holds due to the confidence level of the binomial tail bound $\hat{L}$.

From this, letting F be $e = 0 \wedge \hat{S}(\mathbf{x}) \neq \texttt{IDK}$[1], we have the following:

$$\begin{aligned}
\mathbb{P}\left\{e = 0, \hat{e} = 1 \;\middle|\; \hat{S}(\mathbf{x}) \neq \texttt{IDK}\right\} &= \mathbb{P}\left\{\hat{e} = 1 \;\middle|\; e = 0, \hat{S}(\mathbf{x}) \neq \texttt{IDK}\right\} \mathbb{P}\left\{e = 0 \;\middle|\; \hat{S}(\mathbf{x}) \neq \texttt{IDK}\right\} \\
&\leq \mathbb{P}\left\{\hat{e} = 1 \;\middle|\; e = 0, \hat{S}(\mathbf{x}) \neq \texttt{IDK}\right\} \\
&= \int \mathbb{1}\left\{\hat{e} = 1\right\} p\left(\mathbf{x}, \mathbf{y}, n_{\mathbf{y}}, \mathbf{S}_{\mathbf{y}} \mid F\right) \, \mathrm{d}\mathbf{x}, \mathbf{y}, n_{\mathbf{y}}, \mathbf{S}_{\mathbf{y}} \\
&= \int \mathbb{1}\left\{\hat{e} = 1\right\} p\left(\mathbf{S}_{\mathbf{y}} \mid \mathbf{x}, \mathbf{y}, n_{\mathbf{y}}, F\right) p\left(\mathbf{x}, \mathbf{y}, n_{\mathbf{y}} \mid F\right) \, \mathrm{d}\mathbf{x}, \mathbf{y}, n_{\mathbf{y}}, \mathbf{S}_{\mathbf{y}} \\
&= \int \mathbb{P}_{\mathbf{S}_{\mathbf{y}}}\left\{\hat{e} = 1 \;\middle|\; e = 0, \hat{S}(\mathbf{x}) \neq \texttt{IDK}\right\} p\left(\mathbf{x}, \mathbf{y}, n_{\mathbf{y}} \mid F\right) \, \mathrm{d}\mathbf{x}, \mathbf{y}, n_{\mathbf{y}} \\
&\leq \int \varepsilon_E \cdot p\left(\mathbf{x}, \mathbf{y}, n_{\mathbf{y}} \mid F\right) \, \mathrm{d}\mathbf{x}, \mathbf{y}, n_{\mathbf{y}} \\
&= \varepsilon_E,
\end{aligned}$$

as claimed.

I A Proof of Theorem 1

We use the same proof techniques used in [13] and [16]. Here, we add the proof for completeness.

First, from Lemma 1 and the binomial tail bound, we have the following point bound for any $\hat{S}$, $\varepsilon_E \in (0, 1)$, and $\delta_S \in (0, 1)$:

$$\mathcal{R}_\alpha(\hat{S}) \leq \varepsilon_E + \hat{U}_{\text{Binom}}(\hat{k}; |\hat{\mathbf{Z}}|, \delta_S / \lceil \log_2 |\mathbf{Z}| \rceil) \tag{5}$$

with probability at least $1 - \delta_S / \lceil \log_2 |\mathbf{Z}| \rceil$, where the probability is taken over $\mathbf{Z} \sim \mathcal{D}^n$ with a fixed n. Here, $\hat{k}$, $\hat{Z}$, and $\mathbf{Z}$ are as defined in Section 3.5, and $\hat{U}_{\text{Binom}}$ is the standard binomial tail upper bound.

Let $\mathcal{H}$ be a data-dependent set of selective generators, parameterized by τ, with a fixed size m, *i.e.,* $|\mathcal{H}| = m$ (where $m = \lceil \log_2 n \rceil$ in our case) and $\mathcal{H}_{\varepsilon_S} := \{\hat{S} \in \mathcal{H} \mid \mathcal{R}_\alpha(\hat{S}) > \varepsilon_E + \hat{U}_{\text{Binom}}(\hat{k}; |\hat{\mathbf{Z}}|, \delta_S / m)\}$, which is also data-dependent. Recall that $\hat{\tau}$ and $\hat{U}$ are the output of our Algorithm 2. Then, we have the following:

$$\begin{aligned}
&\mathbb{P}_{\mathbf{Z}}\left\{\mathcal{R}_\alpha(\hat{S}) > \hat{U}\right\} \\
&\quad \leq \mathbb{P}_{\mathbf{Z}}\left\{\exists \hat{S} \in \mathcal{H}_{\varepsilon_S}, \mathcal{R}_\alpha(\hat{S}) > \varepsilon_E + \hat{U}_{\text{Binom}}(\hat{k}; |\hat{\mathbf{Z}}|, \delta_S / m)\right\} \\
&\quad \leq \sum_{j=1}^{m} \mathbb{P}_{\mathbf{Z}}\left\{\mathcal{R}_\alpha(\hat{S}_j) > \varepsilon_E + \hat{U}_{\text{Binom}}(\hat{k}; |\hat{\mathbf{Z}}_j|, \delta_S / m)\right\} \qquad (6) \\
&\quad = \sum_{j=1}^{m} \sum_{i_j=0}^{n} \mathbb{P}_{\mathbf{Z}}\left\{\mathcal{R}_\alpha(\hat{S}_j) > \varepsilon_E + \hat{U}_{\text{Binom}}(\hat{k}_j; |\hat{\mathbf{Z}}_j|, \delta_S / m), |\hat{\mathbf{Z}}_j| = i_j\right\} \\
&\quad = \sum_{j=1}^{m} \sum_{i_j=0}^{n} \mathbb{P}_{\mathbf{Z}}\left\{\mathcal{R}_\alpha(\hat{S}_j) > \varepsilon_E + \hat{U}_{\text{Binom}}(\hat{k}_j; |\hat{\mathbf{Z}}_j|, \delta_S / m) \;\middle|\; |\hat{\mathbf{Z}}_j| = i_j\right\} \mathbb{P}_{\mathbf{Z}}\left\{|\hat{\mathbf{Z}}_j| = i_j\right\} \\
&\quad \leq \sum_{j=1}^{m} \sum_{i_j=0}^{n} \frac{\delta_S}{m} \mathbb{P}_{\mathbf{Z}}\left\{|\hat{\mathbf{Z}}_j| = i_j\right\} \qquad (7) \\
&\quad = \delta_S,
\end{aligned}$$

where (6) holds due to a union bound and (7) satisfies due to the point bound in (5). This completes the proof.

[1] Note that the probability of the event F is positive unless G is "always-correct" (*i.e.,* $\mathbf{y}$ α-entails $G(\mathbf{x})$ for all $\mathbf{x}$ and $\mathbf{y}$).

J Empirical Evaluation Metrics

We evaluate our method along with baselines based on the empirical FDR-CE, *i.e.,* $\widehat{\text{FDR-CE}}$, and empirical selection efficiency, *i.e.,* $\widehat{\text{SelEff}}$, where $\mathbf{Z}_{\text{t}} \sim \mathcal{D}^{n_{\text{test}}}$ is a test set, as follows:

$$\widehat{\text{FDR-CE}} := \frac{\sum_{(\mathbf{x},\mathbf{y},_)\in\mathbf{Z}_{\text{t}}} \mathbb{1}\left(\hat{S}(\mathbf{x}) \notin \hat{E}_{\alpha,\varepsilon_E^t}(\mathbf{y}) \wedge \hat{S}(\mathbf{x}) \neq \texttt{IDK}\right)}{\sum_{(\mathbf{x},_)\in\mathbf{Z}_{\text{t}}} \mathbb{1}\left(\hat{S}(\mathbf{x}) \neq \texttt{IDK}\right)} \quad \text{and} \tag{8}$$

$$\widehat{\text{SelEff}} := \frac{1}{|\mathbf{Z}_t|} \sum_{(\mathbf{x},_)\in\mathbf{Z}_{\text{t}}} \mathbb{1}(\hat{S}(\mathbf{x}) \neq \texttt{IDK}). \tag{9}$$

We chose $\varepsilon_E^t = 0.01 \ll \varepsilon_E$ for estimating a true α-entailment set E_α.

K Relationship between Pass@1 and FDR-CE

Given an instance $(\mathbf{x}, \mathbf{y})$ in the dataset, suppose there are $n_{\mathbf{y}}$ given unit tests provided. Here, $n_{\mathbf{y}}$ is result of Algorithm 1. When evaluating a dataset with PASS@1, a generated code snippet is considered correct if it passes all $n_{\mathbf{y}}$ unit tests.

As $\alpha \to 0$, if $\mathbf{y}$ α-entails the generated code $G(\mathbf{x})$, then $n_{\mathbf{y}}$ must tend toward ∞. Thus, $\mathbf{y}$ exhibits *expected functional correctness* approaching 1. Therefore, with high probability, any finite set of $n_{\mathbf{y}}$ unit tests will be successfully executed without a failed test.

Conversely, if a generated code snippet successfully executes $n_{\mathbf{y}}$ unit tests as $n_{\mathbf{y}} \to \infty$, then with high probability, $\mathbf{y}$ will α-entail the generated code $G(\mathbf{x})$.

Therefore, $1-$PASS@1 and FDR-CE may be asymptotically equivalent as $\alpha \to 0$ and $n_{\mathbf{y}} \to \infty$.

The empirical result is presented at Table 1 and 3

L alpha-Entailment Code Example

Table 7: α-entailment example with $\alpha = 0.3$. The solution code accurately simulates the exact problem scenario, while the generated code simplifies the logic and therefore misses corner cases by missing the α fraction of unit tests generated from the solution code.

Question

```
n people are standing in a line to play table tennis.
At first, the first two players in the line play a game.
Then the loser goes to the end of the line, and the winner
plays with the next person from the line, and so on.
They play until someone wins k games in a row.
This player becomes the winner. For each of the
participants,  you know the power to play table tennis,
and for all players these values are different. In a
game the player with greater power always wins.
Determine who will be the winner.
```

(a) Canonical Solution

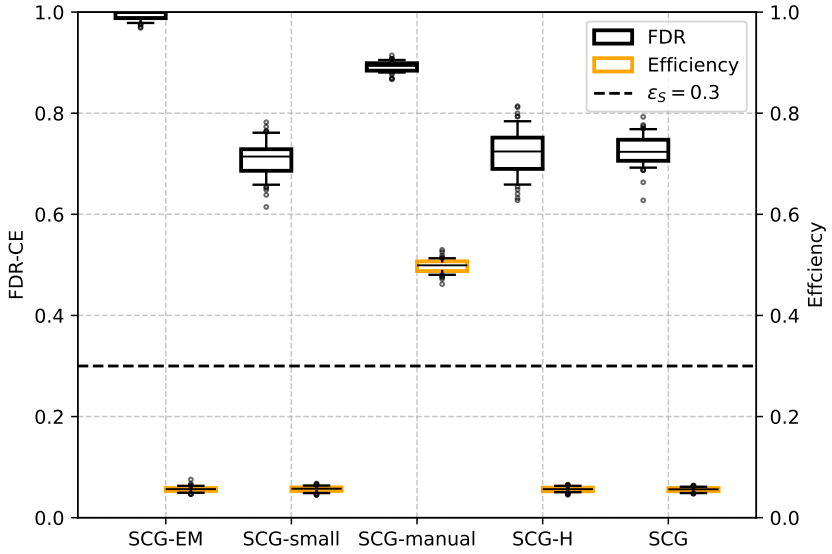
```
tmp = list(map(int, input().split()))
n, k = tmp[0], tmp[1]
a = list(map(int, input().split()))

cur = a[0]
w = 0
OK = False

for i in range(1, 10000):
    op = a[i]
    if cur > op:
        a.append(op)
        w += 1
    else:
        cur = op
        a.append(cur)
        w = 1
    if w >= k:
        OK = True
        break

if OK:
    print(cur)
else:
    print(max(a))
```

(b) Generated Solution

```
def find_winner(n, k, powers):
    current_player_index = 0
    current_wins = 0
    max_power = powers[0]

    if k > n:
        print(max(powers))
        return

    for i in range(1, n):
        if powers[current_player_index] > powers[i]:
            current_wins += 1
        else:
            current_player_index = i
            current_wins = 1

        if current_wins == k:
            print(powers[current_player_index])
            return

    print(max_power)

# Read input
n, k = map(int, input().split())
powers = list(map(int, input().split()))

find_winner(n, k, powers)
```

M Additional Experiments

M.1 Varying Model

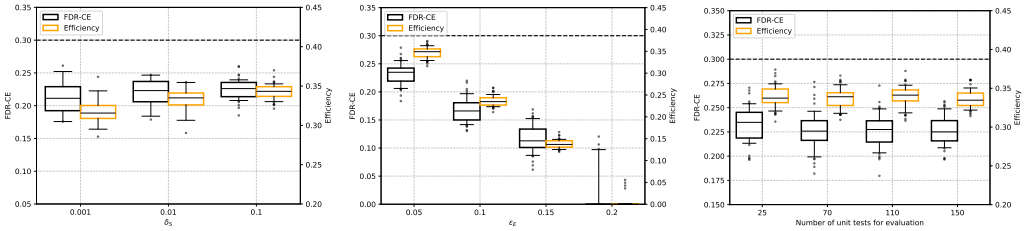


(a) CodeLlama 13B Instruct

Figure 4: The box plots of the FDR-CE and selection efficiency for CodeLlama 13B Instruct. We set $\delta_S = 0.1$, $\varepsilon_S = 0.3$, $\varepsilon_E = 0.05$, and $\alpha = 0.35$. **SCG** fails to find a selective generator with a desired FDR-CE due to uncalibrated scoring function.

M.2 Varying Parameter

We present FDR-CE and efficiency experiments with varying δ_S, varying ε_E, and varying number of unit tests in Figure 5.



(a) Varying δ_S (b) Varying ε_E (c) Varying eval. unit tests

Figure 5: The FDR-CE results for GPT-4o with varying parameters ($\varepsilon_S = 0.3, \delta_S = 0.1, \alpha = 0.35$, and $\varepsilon_E = 0.05$). Each figure shows FDR-CE bound is satisfied for each settings. This is shown by upper whisker bar lying below the desired FDR-CE in the dotted line. Figure 5(b) shows benefit of fuzzing in learning and Figure 5(c) show benefit of fuzzing in evaluation.

M.3 Unit Test

We add an evaluation study on *FuzzEval*, showing benefits of added unit tests and flexibility of α-code entailment. We measure the performance of generators with conventional pass@1 metric with various α. Table M.3 demonstrates that more through evaluation is achievable with test cases generated by a fuzzing tool. Furthermore, pass@1 result with relaxed α-setting implies the necessity of incorporating α in *FuzzEval*.

Table 8: Pass@1 of each generator with original dataset and generated unit tests. We measured the performance of each code generators with pass@1 on original datasets along with added unit tests in *FuzzEval*. We also measured α to show flexibility. Note that we excluded problems that encountered error during the generation process.

Code Generator Model	**without additional unit tests**	**with additional unit tests**			
	pass@1	pass@1	pass@1 ($\alpha = 0.1$)	pass@1 ($\alpha = 0.3$)	pass@1 ($\alpha = 0.5$)
GPT-4o	0.515	0.483	0.526	0.577	0.589
Gemini-1.5 Pro	0.526	0.480	0.523	0.566	0.578
CodeLlama-13B-Instruct	0.051	0.048	0.059	0.073	0.075

M.4 LLM-generated unit test

Unit tests generated via other techniques also could be used in **SCG**. To support our claim, we conducted experiments utilizing unit test sets generated with the assistance of LLM. We applied **SCG** on HumanEval+ [25] and MBPP+[25], which are publicly available datasets augmented with LLM-based method. Table 9 demonstrates the result of our method.

Table 9: We used GPT-4o as the model. The parameters are set as $\alpha = 0.2$, δ_S = 0.1, ε_S = 0.25, ε_E = 0.05. The FDR-CE satisfies the desired bound (ε_S) which demonstrates that our method is applicable on different datasets with test generated with the assistance of LLMs.

Dataset	pass@1	FDR-CE	Efficiency
MBPP+	0.720	0.121 ± 0.112	0.423 ± 0.446
HumanEval+	0.848	0.056 ± 0.044	0.979 ± 0.029

M.5 Experiment Setup

We used 4 NVIDIA A100 80GB with 128 CPUs for code generation. We used the same environment for fuzzing and calibration.

N Discussion

N.1 Guidelines for Parameter Selection

Our algorithm has user-specified parameters, ε_S, δ_S, α, ε_E, and n_{max}. The ε_S and δ_S are the main parameters that encode user's desired on the performance on the selective generator, where the smaller values are better.

The α encodes a degree of the correctness of a generator, where $\alpha = 0$ is the ideal value. But, assuming that there is no perfect generator that exactly returns correct code, this never achieves. We recommend to choose some small value on this on evaluating current state-of-the-art generators.

The ε_E and n_{max} are associated to the number of generated unit tests via dynamic code analysis tools, where the ideal value of ε_E is zero and n_{max} is infinity. The larger number of unit tests is definitely preferred but it sacrifices the running time of dynamic code analysis tools and evaluation.