ExplainFuzz : Explainability and wellformedness for probabilistic test generation

Annaelle Baiget, London Bielicke, Arthur Lu, Brooke Simon

***Abstract*—Understanding and explaining the structure of generated test inputs is crucial for effective testing, debugging, and analysis of software systems. However, existing approaches—such as probabilistic context-free grammars (pCFGs) and large language models (LLMs)—lack the ability to provide fine-grained statistical explanations about generated test inputs and their structure. We introduce *ExplainFuzz*, a novel framework that leverages probabilistic circuits (PCs) to model and query the distribution of grammar-based inputs in an interpretable manner. Starting from a context-free grammar (CFG), we refactor it to support PC compilation, and train the resulting Probabilistic Circuit on a synthetically generated corpus produced with Grammarinator during a fuzzing campaign. The trained PC supports a variety of probabilistic queries (e.g., $P(\texttt{JOIN})$ or $P(\texttt{BY} \mid \texttt{ORDER})$), offering insight into the statistical distribution of generated inputs. Additionally, for the SQL domain, we demonstrate a custom generator that transforms PC generated samples into executable queries by leveraging PC's generation capabilities to enable concrete synthetic test input generation. We evaluate *ExplainFuzz* across multiple domains including SQL, REDIS, and JANUS, highlighting its ability to provide explainable, grammar-aware insights into test input structure. Our results show that *ExplainFuzz* outperforms traditional pCFGs and LLMs in terms of log-likelihood estimation and interpretability, contributing a new direction for explainable grammar-based fuzzing.**

***Index Terms*—Probabilistic Circuit, Context-Free Grammar, Grammar-Based Fuzzing, Explainable AI**

I. Introduction

Generating high-quality test inputs is essential for effective software testing, particularly in domains like compilers and databases where inputs must follow complex syntactic structures. Grammar-based fuzzers such as Grammarinator [1] and SQLSmith [2] have shown promise in producing syntactically valid inputs. However, these tools typically lack support for reasoning about the distributions of generated inputs. This limits the ability to understand test coverage, identify biases, or explain why certain structures appear more frequently than others.

We present **ExplainFuzz**, a novel framework that introduces explainability into grammar-based test generation by leveraging *probabilistic circuits* (PCs). *ExplainFuzz* compiles context-free grammars (CFGs) into tractable probabilistic models and trains them on inputs generated by grammar fuzzing. Once trained, the PC enables structured inference, allowing users to query a variety of probabilistic properties over input structures. For example we can ask for the probability that a `WHERE` clause of a SQL query is followed by `ORDER BY`. We evaluate *ExplainFuzz* across four dimensions. First, we measure the likelihood of the generated inputs by comparing them with real-world examples (Section IV-A). Second, we assess the accuracy of PC-based inference across multiple domains, showing that the model captures key structural patterns (Section IV-B). Third, we analyze the scalability and performance of PC training in grammars of varying complexity (Section IV-D). Finally, we present a case study in the SQL domain, where we concretize PC samples using a custom generator and evaluate the well-formedness of the resulting queries (Section IV-E). Together, these evaluations demonstrate *ExplainFuzz*'s ability to model and reason about structured input spaces, offering a new explainable foundation for grammar-based fuzzing.

This paper makes the following contributions:

- We propose **ExplainFuzz**, a novel test input generator that combines grammar refactoring, probabilistic circuit (PC) learning, and optional semantic concretization to enable explainable input generation.
- We show that PCs can learn and approximate structured input distributions from seed corpora more accurately than traditional PCFG-based fuzzing or LLM-generated inputs.
- We introduce interactive reasoning capabilities over the learned input distribution, allowing users to query probabilities of grammar constructs and their co-occurrences.
- We demonstrate that while PC-based generation improves structural realism, domain-specific concretization is required to achieve well-formedness in semantically rich domains like SQL.

II. Background and Related Work

A. Grammar-Based Fuzzing

Grammar-based fuzzing is a software testing technique that generates well-formed inputs based on a formal grammar.

Despite its strengths, grammar-based fuzzing struggles with domain-specific semantics. For instance, in SQL, a fuzzer might generate inputs like S`SELECT DISTINCT '''' AS a FROM J4$ AS O6, (* ) AS V4;` that are syntactically valid but semantically invalid—e.g., due to malformed subqueries or illegal table aliases—rendering them unusable in practice.

Second, there is a lack of explainability. Tools such as Grammarinator [1] can produce thousands of inputs quickly but offer little insight into which parts of the input space are being explored or why certain structures are overrepresented, making it difficult to assess test coverage and bias.

Finally, biasing input generation toward specific constructs is challenging. While tools like Grammarinator can mutate

an initial population of inputs to encourage variety, these mutations are typically random or rule-based. This limits the usefulness in grammar-based fuzzing in exploring high-priority sub-domains like those which are observed to produce bugs.

B. Probabilistic Grammar Learning

Probabilistic Context-Free Grammars (PCFGs) enhance context-free grammars by assigning probabilities to production rules, defining a distribution over derivations. These probabilities are estimated from a corpus of examples.

PCFGs guide tasks like fuzzing or program synthesis by favoring more probable syntactic structures. Tools such as Skyfire [3] use these techniques to prioritize likely inputs during grammar-based fuzzing.

However, PCFGs cannot model context-sensitive distributions. The probability of selecting a production rule is fixed and does not depend on derivation history or token position. This limitation prevents capturing long-range dependencies or context-dependent preferences, crucial for modeling realistic input distributions. The assumption of independence reduces the accuracy of learned distributions, especially for complex input formats with conditional constraints or hierarchical structures.

C. Probabilistic Circuits

Probabilistic circuits (PCs) are a general class of tractable probabilistic models that represent complex probability distributions using a graph composed of *sum* nodes, *product* nodes, and *leaf* nodes [4], [5]. Specifically, leaf nodes represent elementary distributions, while sum nodes take a mixture of their children and product nodes take an independent decomposition of their children. PCs have been shown to subsume most existing tractable probabilistic models, such as bounded-treewidth Bayesian networks or cutset networks.

The key strength of PCs lies in their *tractability*: under specific structural constraints, such as smoothness and decomposability, PCs allow for the exact computation of marginal probabilities, conditionals, and maximum a posteriori estimation (MAP) in time linear to the size of the circuit [6]. These properties make them highly *explainable* and suitable for structured inference. Moreover, PCs can be trained on data using expectation maximization or gradient-based optimization, a common training technique for neural networks. As PCs are generative models, new instances can be sampled from this learned distribution [6].

III. Approach and Implementation

This section describes the system design and implementation of *ExplainFuzz*.

A. Overview

Figure 1 illustrates the architecture of *ExplainFuzz*, which consists of three interconnected stages: (i) **Preprocessing** (ii) **Probabilistic Circuit Learning**, and (iii) **Concretization**.

a) Preprocessing: To ensure proper mapping from the CFG to PC, we need to refactor the existing grammar of the chosen domain. The **grammar refactoring** process separates a combined ANTLR grammar—containing both lexical and syntactic rules—into distinct Lexer and Parser components. Then, we remove quantifiers from the parse rules and ensure no rule consists of more than two nullable non-terminal productions, to convert the grammar to Chomsky Normal Form (CNF). The resulting refactored grammar is used consistently across subsequent components. In the **Fuzzing Campaign** phase, a seed corpus and the refactored grammar are fed into Grammarinator, which generates a large set of syntactically valid inputs. This step provides a diverse training corpus that reflects the distribution of the seeds.

b) Probabilistic Circuit: The refactored parser is then used to build a **Probabilistic Circuit (PC)**, such that the support of the PC distribution is the language of the grammar. The PC is trained on the generated dataset to learn structural distributions over symbolic grammar tokens.

c) Concretization: The concretization step bridges the gap between the anonymized syntax and real-world constraints, enabling meaningful fuzzing outcomes. The PC guides the **generation of new inputs** that are not only grammatically valid but also explainable. These sampled anonymized inputs are passed to a custom generator, which incorporates domain-specific context to produce concrete, executable test cases.

To support interactive use, ExplainFuzz includes a **user interface** that allows users to generate customized inputs or query the likelihood of specific structural patterns. This UI component enables real-time exploration of the probabilistic model and fosters user-guided fuzzing scenarios.

B. Grammar Refactoring

The grammar refactoring process is implemented in Python and leverages the ANTLR4 runtime to parse and transform ANTLR grammars into Chomsky Normal Form (CNF), a prerequisite for probabilistic circuit construction. CNF transformation ensures that each production rule is either of the form $A \to BC$ or $A \to a$, where A, B, C are nonterminals and a is a terminal.

This structured transformation ensures compatibility with downstream PC construction while preserving the semantics of the original grammar.

C. Fuzzing Campaign

We use Grammarinator to generate 10,000 inputs per domain for training and evaluation, applying two strategies:

- **Mutational Mode (no-generate):** Fuzzes five real-world seed inputs to produce syntactically valid samples that preserve realistic structure and distribution.
- **Generative Mode (with-generate):** Generates inputs directly from the grammar without seeds, increasing syntactic diversity but often reducing semantic realism.

Each dataset is split into 9,000 training and 1,000 test samples. Before training, inputs are *anonymized* using the

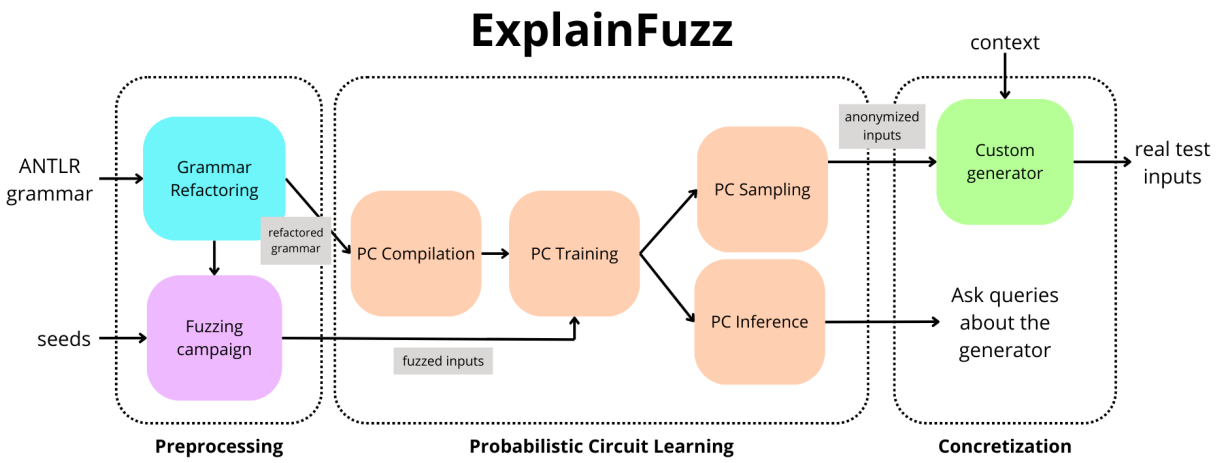

Fig. 1. System overview of ExplainFuzz: grammar refactoring, fuzzing campaign, PC learning, inference, and generation.

grammar's lexer by replacing all literal tokens (e.g., strings, numbers, identifiers) with symbolic terminal names (e.g., `STRING`, `Number`, `Identifier`), reducing vocabulary and simplifying the PC.

D. PC Compilation

On a high level, the construction of a probabilistic circuit from a context-free grammar resembles bottom-up parsing techniques such as the well-known CYK algorithm [7]. Just as a bottom-up parser, we start with single terminal symbols and build up nonterminal derivations from there. Unlike a parser, the circuit construction does not consider a specific input sequence but takes every possible input sequence into account. This means we need to assume a maximum length of the input sequence.

Algorithm 1 (described in the appendix) provides pseudo-code for the PC construction. The circuit construction first adds a sum node for each nonterminal symbol in the grammar and each subset of the sequence (i.e., a starting and ending position). Next, we add a product node for each rule and subset of the sequence. The conjunction node takes as children the two symbols that are required in the rule, while the disjunction node takes as children all the conjunction nodes that derive its rule for the specific sequence subset.

Figure 5 (appendix) illustrates the process on a simple arithmetic ANTLR grammar, through its refactored CNF form, to the resulting probabilistic circuit (PC) constructed using Algorithm 1 (cf Appendix) with a maximum sequence length of 5. We implement the circuit construction in Python using the KLay library [8], enabling GPU-accelerated training.

E. PC Inference

We leverage the trained PC to query the structure of inputs, which is crucial for understanding the underlying patterns and probabilities associated with different test inputs. *ExplainFuzz* implements four classes of queries tractable in linear time using a PC: marginal, conditional, maximum a posteriori, and complete evidence queries. Within each class, we can ask various types of queries. The complete list of these can be seen in figure 6 of the Appendix.

Those queries can be used to understand the space of test inputs and could guide further refinement of the input generation process.

F. PC Sampling

Beyond inference, the trained probabilistic circuit (PC) can be used for input generation by sampling from the learned distribution. This allows us to explore the space of valid, grammar-conforming inputs that reflect the structural patterns observed during training. Moreover, the PC supports **conditional sampling**, enabling generation under specific constraints. For instance, one can sample inputs that include a given token either anywhere in the sequence or at a specific position. It is also possible to enforce the presence of fixed subsequences of tokens while allowing the rest of the input to vary probabilistically. This flexible conditioning capability makes PCs a powerful tool for guided generation, test case synthesis, and controlled exploration of input spaces.

G. Custom Concretization Phase for SQL Domain

Because the PC operates over anonymized inputs—where literals like strings or identifiers are replaced by symbolic terminals—samples must be concretized into executable SQL queries. This reverse lexing process requires domain-specific logic to ensure semantic validity.

We implement a custom concretizer for SQL, which performs the following:

1) **Schema Retrieval:** Connects to a PostgreSQL database to extract table names, column names, and types.
2) **Token Replacement:** Replaces fixed tokens (e.g., `'SELECT'` $\rightarrow$ `SELECT`) using ANTLR lexer mappings.
3) **Schema-Aware Parsing:** Parses queries with placeholders using a modified ANTLR lexer and listener to track context (e.g., inside a WHERE clause).

4) **Contextual Insertion:** Replaces placeholders with schema-consistent values—e.g., valid column names for `Identifier`, words for `StringConstant`, and type-matching numbers.

H. Interactive User Interface

To make *ExplainFuzz* accessible to users across different domains, we provide an interface that supports domain selection, model configuration, and interactive querying. This UI was implemented using the Python library Gradio.

a) Domain and mode Selection: The UI supports a predefined set of domains, each associated with a specific ANTLR grammar and corresponding datasets. Upon launching the interface, the user selects the desired domain (e.g., SQL or CSV). The user can choose between a model trained using only grammar-derived inputs (mode *with-generate*) or a model trained using a seed corpus (mode *no-generate*). This dual-mode configuration allows users to compare purely syntactic structures with more semantically rich patterns learned from real-world inputs.

b) Querying Structural Probabilities: Users can explore the structural properties of the learned model by selecting from a predefined set of probabilistic queries (see Section III-E). Figure 7 (in the appendix) shows a screenshot of the user interface.

c) Concrete Input Generation: The UI also supports the generation of a user-specified number of inputs. For the SQL domain, it leverages the custom concretization phase to produce fully executable queries. For other domains, the generated inputs remain partially anonymized: only tokens with unambiguous symbolic representations are replaced with their corresponding literals, while ambiguous placeholders (e.g., `NAME`, `INT`) remain generic.

IV. Evaluation

We evaluate *ExplainFuzz* across multiple dimensions to assess its effectiveness, generalizability, and practical utility. Our evaluation is structured around four core research questions (RQs), each targeting a specific capability of the system. These include: the **quality of the learned probabilistic model**, the system's **inference capabilities**, **scalability** across domains, and the **well-formness and efficiency** of input generation for the SQL domain.

A. Likelihood of Generated Inputs

a) Research Question: **RQ1: Is the likelihood extracted from PC better than the likelihood of LLM and PCFG?** We investigate whether ExplainFuzz more accurately captures the distribution of domain-specific inputs compared to PCFGs, LLMs and a dense PC by measuring the negative log-likelihood.

b) Methodology: We compare four models: (1) **PC (ExplainFuzz)**, our probabilistic circuit built from the grammar and trained on anonymized training data; (2) **PC (HMM)** unstructured PC modeled as a Hidden Markov Model (HMM), using a right-linear vtree and approximately 260K edges and trained of the same anonymized training data. This configuration mirrors the approach used in neural-guided symbolic modeling systems such as Ctrl-G [9]; (3) **PCFG**, a probabilistic context-free grammar with rule probabilities estimated from the same training data; and (3) **LLM**, a pre-trained GPT-2 model prompted with the 5 anonymized seed queries. All models are evaluated on a held-out test set collected during the `no-generate` fuzzing phase, ensuring input distributions are consistent with the training seeds. For each model, we compute the average log-likelihood over all test queries: for PC and PCFG by summing rule or node log-probabilities; for LLM by scoring test queries via standard left-to-right token likelihood.

c) Metric: We report the **average negative log-likelihood per query**, defined as follows.

$$\text{Negative Avg Log-Likelihood} = -\frac{1}{N}\sum_{i=1}^{N}\log P(x_i)$$

where x_i is a test query, and $P(x_i)$ is the probability assigned by the model. The negative likelihood score indicates how well a model explains the data it did not see during training (lower is better).

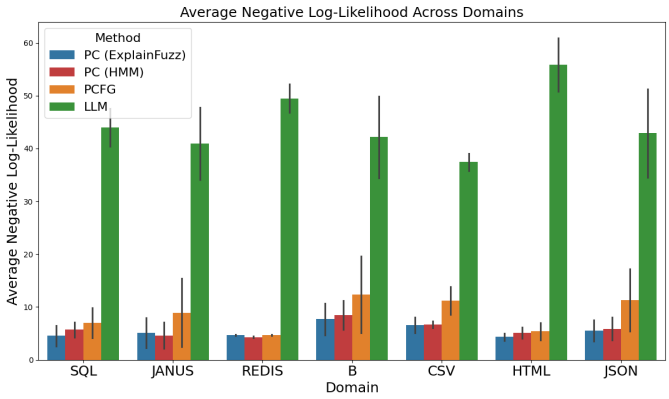

Fig. 2. Negative log-likelihood VS method type.

d) Log-likelihood of generated inputs: Figure 2 presents the average log-likelihood achieved by the three methods across seven domains: **SQL**, **JANUS**, **REDIS**, **B**,**CSV**,**HTML** and **JSON**. The results are averaged across different maximum input lengths, which explains the presence of error bars.

We observe that the structured PC model (ExplainFuzz) achieves the best (i.e., lowest) average log-likelihood in five out of seven domains: SQL, B, CSV, HTML, and JSON. In contrast, in the JANUS and REDIS domains, the unstructured model (HMM) performs best, with ExplainFuzz ranking second.

Overall, these results highlight the PC model's strength in capturing domain-specific input distributions. Unlike the PCFG, which assigns fixed probabilities to production rules, the PC can duplicate and specialize rules, enabling it to learn more context-sensitive and fine-grained patterns. This leads

to more accurate modeling of the input space, especially in domains with rich structure.

The LLM, by contrast, performs worse across all domains, even when provided with few-shot examples. This can be explained by the LLM's broad pretraining objective: while it has been trained on vast amounts of data, this generality makes it less effective at modeling narrow, domain-specific distributions.

Although unstructured models like the HMM can outperform structured approaches in certain domains —thanks to their greater expressivity and ability to capture fine-grained distributional nuances— they lack grammar constraints and may generate invalid sequences. This limits their practical utility for tasks such as fuzzing or program synthesis, where syntactic correctness is essential. By contrast, ExplainFuzz enforces grammatical validity while still learning expressive, context-sensitive distributions, resulting in superior performance in 5 out of the 7 domains. This confirms the benefit of combining structural constraints with probabilistic circuit modeling, a strategy that balances expressivity and correctness.

B. Accuracy of PC-Based Inference

a) Research Question: **RQ2: How accurately does ExplainFuzz estimate queries about the distribution?**

This serves as a sanity check to validate that the probabilistic circuit (PC) accurately captures the underlying distribution of the seed inputs. Specifically, we assess whether the learned distribution aligns with the original data used during training, confirming that the PC is functioning as expected.

Domain	Query Type	From seeds	Random
B	Conditional	0.0757	0.2392
B	Complete Evidence	0.1312	0.1429
B	Marginal	0.0353	0.0938
JANUS	Conditional	0.1679	0.3713
JANUS	Complete Evidence	0.1281	0.2000
JANUS	Marginal	0.1522	0.3384
REDIS	Conditional	0.1602	0.4457
REDIS	Complete Evidence	0.0993	0.1428
REDIS	Marginal	0.0450	0.2832
SQL	Conditional	0.0030	0.0813
SQL	Complete Evidence	0.0973	0.1667
SQL	Marginal	0.0303	0.4926

Fig. 3. Mean absolute differences in PC distribution and ground-truth seed probabilities by domain, query type, and mode (random or from seeds).

b) Methodology: We extracted the ground-truth probabilities from a set of input seeds. Then, we compared the ground truth against the probability distribution from the PC after training on generated datasets (mode no generate, ie from seeds). We expect the PC distribution to resemble the original seeds. We also train a PC using randomly generated inputs (mode with generate, ie random), expecting that the distribution of this model will differ from the ground-truth seeds. We assess the distribution using marginal, conditional, and complete evidence queries, as described in subsection III-E. This approach validates the effectiveness of PC training in capturing the underlying distribution of the seed data.

c) Results: Our results show that the distribution of the PC trained on randomly generated inputs varies significantly from the distribution computed directly from the seeds. Additionally, the probabilities computed from the seed-generated inputs are more similar to the ground-truth distribution compared to the PC trained with randomly-generated inputs across every domain and query class Figure 3. This highlights the importance of using representative training data and shows that the conditional PC effectively captures the true distribution of inputs observed during inference.

C. Case Study: Uncovering Underrepresented Structures in SQL Inputs

To demonstrate the practical utility of inference queries, we consider a scenario in the SQL domain using a PC trained on hand-crafted seed inputs. After encountering a rare bug involving a `GROUP BY` clause, a developer wishes to assess how well this structure is represented in the test input distribution.

a) Quantifying Representation: The developer first issues a marginal (MAR) query to estimate how frequently the token `GROUP` appears in the training distribution. The PC reports $P(\texttt{GROUP}) = 0.0128$, in contrast to $P(\texttt{ORDER}) = 0.8161$, indicating that `GROUP` clauses are heavily underrepresented relative to other common SQL constructs such as `ORDER BY`.

b) Contextual Exploration: To explore the syntactic contexts in which `GROUP` appears, the developer issues conditional (COND) queries. The probabilities $P(\texttt{GROUP} \mid \texttt{FROM}) = 0.0128$, $P(\texttt{GROUP} \mid \texttt{JOIN}) = 0.0018$, and $P(\texttt{GROUP} \mid \texttt{WHERE}) = 0.0129$ confirm that `GROUP` tends to follow FROM and WHERE clauses as allowed by the grammar. The query $P(\texttt{ORDER} \mid \texttt{GROUP}) = 0.7518$ shows that when `GROUP` does appear, it is usually followed by an `ORDER BY` clause. This reflects the fixed clause order imposed by the grammar, rather than optional or semantic usage. Interestingly, $P(\texttt{HAVING} \mid \texttt{GROUP}) = 0.0033$ reveals that HAVING rarely follows GROUP, despite the strong semantic relationship between these two clauses.

c) Outcome: These queries confirm that `GROUP BY` clauses are both rare and structurally fragile in the test distribution. Even when they occur, they are not often followed by semantically related constructs such as `HAVING`. Armed with this insight, the developer may choose to augment the seed set with more examples featuring `GROUP BY` and `HAVING`, or to apply conditional sampling using *ExplainFuzz*'s custom generator to better target such underrepresented and semantically rich combinations. This illustrates how structure-aware probabilistic inference can surface blind spots in input generation and suggest actionable refinements.

D. Scalability / Performance

a) Research Question: **RQ3: How well does *ExplainFuzz* scale across domains with varying grammar complexity, and what are the key limitations?**

TABLE I
CAPABILITY EVALUATION ACROSS DOMAINS AND PIPELINE COMPONENTS
✓: *pass* - ✗: *fail* - (✓)* : *pass with some adjustments*

Domain	Grammar Refactoring	Fuzzing Campaign	PC Compilation	PC Training	PC Inference	PC Sampling	Concretization
SQL (Simplified)	✓	✓	(✓)*	✓	✓	✓	✓
JANUS	✓	✓	✓	✓	✓	✓	✗
REDIS	✓	✓	✓	✓	(✓)*	✓	✗
B	✓	✓	✓	✓	✓	✓	✗
CSV	✓	✓	✓	✓	(✓)*	✓	✗
HTML	✓	✓	(✓)*	✓	✓	✓	✗
MLIR	✓	✓	(✓)*	✗	✗	✗	✗
JSON	✓	✓	✓	✓	(✓)*	✓	✗
CloudFormation (JSON)	(✓)*	✗	(✓)*	✗	✗	✗	✗
Lua	✗	✗	✗	✗	✗	✗	✗

This evaluation investigates the generality and scalability of *ExplainFuzz* across multiple input domains. Our goal is to understand how the system performs when applied to grammars of differing structural complexity, and to identify which stages of the pipeline—such as grammar refactoring, probabilistic model construction, or input generation—present the most significant challenges. By analyzing performance across diverse domains, we aim to assess the robustness and adaptability of the approach, as well as to highlight current limitations and opportunities for improvement.

b) Methodology: To evaluate the scalability and domain applicability of *ExplainFuzz*, we conducted an empirical study using a diverse set of grammars collected from the open-source ANTLR4 grammar repository. For each domain, we ran the full *ExplainFuzz* pipeline and recorded the furthest stage reached without errors in the pipeline. This allowed us to identify at which step failures or limitations typically occur.

c) Capability Analysis Across Domains: We begin our analysis by evaluating the applicability of the pipeline across different domains, as shown in Table I. Each domain is assessed for its ability to support the seven key stages of the pipeline: grammar refactoring, fuzzing, PC compilation, training, inference, sampling, and concretization.

- **SQL (Simplified)**: The full pipeline succeeds if we relax the constraint requiring the grammar to be unambiguous during PC compilation. Concretization is successful here due to a domain-specific generator implementation.
- **JANUS, REDIS, B**: These domains support all stages except concretization.
- **HTML** : The entire pipeline, except for the concretization stage, succeeds if we relax the requirement for the grammar to be unambiguous during PC compilation.
- **CSV, JSON**: These grammars proceed through all pipeline stages except concretization. However, the inference step yields limited insights due to the structural simplicity and high anonymization of these inputs.
- **MLIR**: The pipeline fails during PC compilation and training due to the complexity of the grammar and the excessive length of seed inputs. While compilation may succeed for short sequence lengths, these cases are not useful because the training dataset don't have inputs of these sizes.
- **CloudFormation (JSON)**: PC compilation succeeds for short sequence lengths using a JSON-based grammar that was manually extended with token constraints to improve inference, along with a dataset derived from real-world examples. However, PC training fails due to the excessive length of seed inputs, requiring the min sequence length to be over 1000 tokens.
- **Lua**: This grammar requires an external Lexer which is not supported by our refactoring module, thus the pipeline fails at the first step.

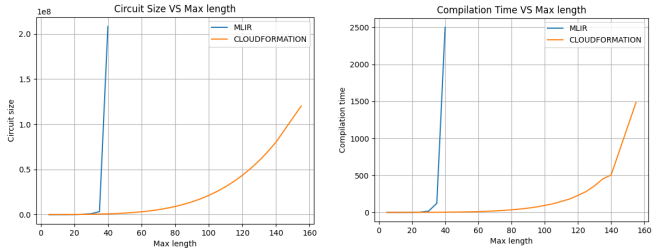

Fig. 4. Circuit size vs max sequence length (Left) and Compilation time vs max sequence length (Right) for the MLIR and CloudFormation domains.

d) Quantifiers, Anonymization, and Tokenization.: A core challenge in compiling PCs from context-free grammars stems from quantifiers (e.g., `*`, `+`, `?`) that allow for arbitrarily long constructs such as lists, strings, or nested objects. To make compilation tractable, ExplainFuzz applies *anonymization* by default—replacing unbounded fields like string literals or identifiers with generic placeholders (e.g., `STRING`, `IDENTIFIER`). This significantly reduces the size of the grammar and resulting circuit, enabling full compilation and training, as shown for the JSON domain. However, anonymization collapses semantic distinctions, limiting the PC's ability to learn meaningful relationships between domain-specific keys and values.

To address this, we experimented with *tokenization* in the AWS CloudFormation template domain, which lacks a formal grammar beyond JSON. Instead of treating string values as atomic placeholders, we applied a pre-trained LLM tokenizer to preserve semantic detail by breaking values into meaningful subword units. This approach is better suited for informative inference in theory, but it dramatically increased the number of terminal symbols and grammar complexity. As a result,

the circuit size grew beyond ExplainFuzz's compilation limits, making it inapplicable in practice.

Figure 4 quantifies this trade-off. Both CloudFormation and MLIR quickly hit the 2,500-second compilation timeout. MLIR reaches 20M nodes at sequence length 40—well below its 120-token average—while CloudFormation reaches 13M nodes at length 155, still far from its 2,000+ token average. These results highlight a core limitation: anonymization ensures tractability but sacrifices semantic richness; tokenization preserves expressiveness but severely limits scalability in complex domains.

E. Case Study : Well-formness of the generated inputs for the SQL domain

a) Research Question: **RQ4: Is ExplainFuzz able to generate well-formed test data?** This evaluation assesses the syntactic and semantic validity of the inputs generated by *ExplainFuzz* compared to baseline methods such as Grammarinator and SQLSmith.

b) Methodology: To evaluate the success rate, we compare the ability of *ExplainFuzz*, Grammarinator and SQLSmith to generate valid inputs across different domains. The success rate is defined as the percentage of generated queries that execute without errors.

- **Grammarinator**: We used Grammarinator with a seed of 5 short SQL queries, generating 10,000 inputs under the configurations no generate. We measured the success rate of executing these queries on the PostgreSQL database.
- **SQLSmith**: We generated 10,000 inputs using SQLSmith, filtering to keep only `SELECT` statements. The success rate was computed based on the execution of these queries on the PostgreSQL database.
- **ExplainFuzz (PC + custom generator)** For our custom generator we sample 500 anonymized queries from the PC. For each query, we generated 20 variations, ensuring that the structure remained the same while varying tables, columns, and numbers. We measured the success rate of executing these queries on the PostgreSQL database.

Method	Global success rate (%)
Grammarinator	36.29
SQLSmith	41.73
ExplainFuzz	**86.07**

TABLE II
GLOBAL SUCCESS RATE FOR EACH METHOD.

c) Global success Rate: Table II shows the proportion of successfully executed queries on PostgreSQL for each method when generating 10,000 queries. *ExplainFuzz* significantly outperforms both baseline methods, achieving an execution success rate of approximately 86%, compared to 36% for Grammarinator and 42% for SQLSmith. This more than doubles the success rate of the closest baseline. The results validate our generation strategy: training a Probabilistic Circuit (PC) on Grammarinator-generated data, combined with a structure-aware generator, produces a high proportion of syntactically and semantically valid inputs.

V. DISCUSSION

a) Trade-offs in PC-Based Fuzzing.: ExplainFuzz highlights a key trade-off between scalability and semantic precision. Anonymization simplifies circuit construction by replacing complex values (e.g., strings, identifiers) with placeholders, enabling tractable inference. However, this sacrifices value-level semantics and requires non-trivial post-processing to generate executable inputs. Conversely, tokenization preserves semantic detail by splitting values into subword units, but dramatically increases grammar complexity, often making circuit compilation infeasible for long inputs.

b) Data and Generalization Challenges.: Effective probabilistic circuits depend on large, diverse input corpora to generalize well—especially in domains with deep nesting or optional constructs. Too little data leads to overfitting; aggressive simplification risks omitting rare but important patterns. Balancing coverage and complexity remains an open challenge for robust structured fuzzing.

c) Limitations in Practice.: In domains like CloudFormation and MLIR, we find that neither anonymization nor tokenization alone fully supports scalable, semantically rich modeling. Tokenization respects domain relationships but fails to scale; anonymization scales but loses essential structure. These results highlight the need for new approximations or hybrid techniques to bridge this gap.

VI. FUTURE WORK

Future improvements to *ExplainFuzz* include relaxing grammar-validity constraints through approximation techniques to capture more semantic variation. We also aim to integrate large language models (LLMs) into the concretization phase, enabling generalization across domains without handcrafted generators. Finally, scaling the pipeline across multiple GPUs could improve performance and support larger, more complex grammars.

VII. CONCLUSION

We introduced *ExplainFuzz*, a framework for explainable input generation that combines grammar refactoring, probabilistic circuit learning, and domain-specific concretization. Unlike traditional fuzzers, *ExplainFuzz* offers transparency and control over input structure, enabling users to analyze and guide test generation.

Experiments in the SQL domain show high validity and efficiency, but also highlight the need for custom concretizers per domain. While PC-based inference is domain-agnostic, semantic validity still depends on tailored generation.

Overall, *ExplainFuzz* demonstrates the potential of structure-aware, explainable fuzzing. Future directions include structural approximation, LLM-based concretization, and improved scalability via GPU parallelization.

REFERENCES

[1] R. Hodován, A. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 45–48. [Online]. Available: https://doi.org/10.1145/3278186.3278193

[2] A. Seltenreich, B. Tang, and S. Mullender, "SQLsmith: a random sql query generator," https://github.com/anse1/sqlsmith, 2018, accessed: 2025-05-22.

[3] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," 05 2017, pp. 579–594.

[4] H. Poon and P. Domingos, "Sum-product networks: a new deep architecture," in *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, ser. UAI'11. Arlington, Virginia, USA: AUAI Press, 2011, p. 337–346.

[5] A. Vergari, Y. Choi, A. Liu, S. Teso, and G. Van den Broeck, "A compositional atlas of tractable circuit operations for probabilistic inference," *Advances in Neural Information Processing Systems*, vol. 34, pp. 13 189–13 201, 2021.

[6] Y. Choi, A. Vergari, and G. Van den Broeck, "Probabilistic circuits: A unifying framework for tractable probabilistic models," 2020.

[7] F. M. Zanzotto, G. Satta, and G. Cristini, "Cyk parsing over distributed representations," *Algorithms*, vol. 13, no. 10, p. 262, Oct. 2020. [Online]. Available: http://dx.doi.org/10.3390/a13100262

[8] J. Maene, V. Derkinderen, and P. Z. Dos Martires, "Klay: Accelerating arithmetic circuits for neurosymbolic ai," in *The Thirteenth International Conference on Learning Representations*, 2025.

[9] H. Zhang, P.-N. Kung, M. Yoshida, G. V. den Broeck, and N. Peng, "Adaptable logical control for large language models," 2024. [Online]. Available: https://arxiv.org/abs/2406.13892

APPENDIX

a) NB : The code is also in a private repository on GitHub, if needed, Annaelle can add the reader as a contributor on each repository (there are 5 in total). The main link is : https://github.com/annaellebgt/ExplainFuzz

Algorithm 1 Probabilistic circuit construction

Input: Grammar G in CNF, maximum length n, entry symbol s_1.

Output: Probabilistic circuit c.

Initialize empty circuit c.
for symbol $s \in \mathit{symbols}(G)$ **do**
  for start $i \in [0..n-1]$ **do**
    for end $j \in [j..n]$ **do**
      Add a sum node (s, i, j) to c.
for rule $(s_1 \leftarrow s_2\ s_3) \in \mathit{rules}(G)$ **do**
  for start $i \in [0..n-2]$ **do**
    for middle $k \in [i..n-1]$ **do**
      for end $j \in [k..n]$ **do**
        Add a product node (s_1, s_2, s_3, i, k, j) to c,
        with (s_2, i, k) and (s_3, k, j) as children.
for product node $(s, i, j) \in c$ **do**
  Add every sum node $(s, \_, \_, i, \_, j)$ as a child to (s, i, j).
Add the root, a sum node with children $(s_1, 0, \_)$.
Prune every node in c that is not connected to the root.

(a) ANTLR-style grammar

eq : digit (PLUS digit)*;
digit : ONE | TWO;
ONE : "1";
TWO : "2";
PLUS : "+";

(b) CNF-style grammar

eq →ONE block | TWO block
  | eq block
block →PLUS ONE | PLUS TWO
ONE →"1"
TWO →"2"
PLUS →"+"

(c) Probabilistic Circuit

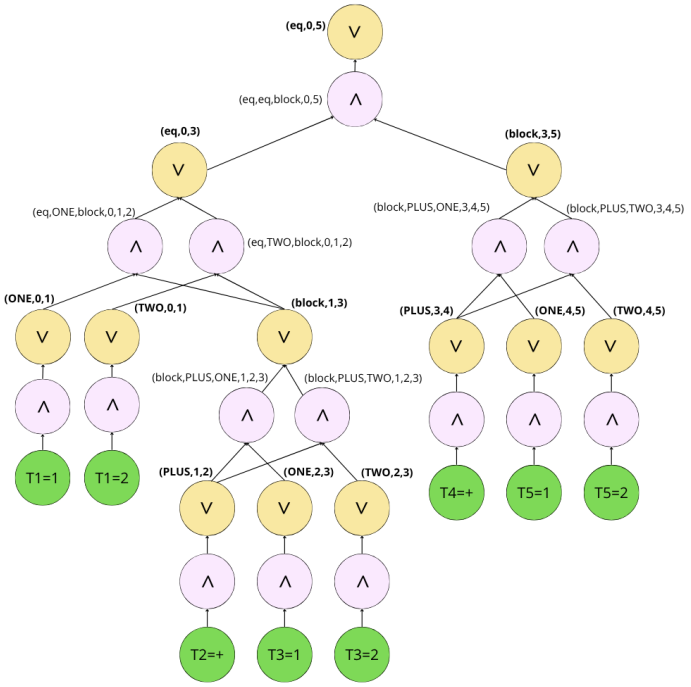

Fig. 5. (a) ANTLR-style grammar of simple arithmetic expressions and (b) a refactored version in the CNF-style, and(c) the corresponding probabilistic circuit (sequence length 5) built from the CNF grammar.

- **Complete Evidence (EVI):** What is the probability of seeing of seeing a specific input?
- **Marginals (MAR):**
  i. What is the probability of token t appearing in an input?
  ii. What is the probability of a contiguous sequence of tokens $(t_1, t_2, ...t_n)$ appearing at position p of an input?
- **Conditionals (COND):**
  i. What is the probability of token t2 appearing after token t1?
  ii. What is the probability of token t2 appearing immediately after token t1, given that we saw t1 at position p?
  iii. What is the probability of a token t appearing in a query after a given sequence, given that we saw the contiguous sequence of tokens $(t_1, t_2, ...t_n)$ at position p in an input?
- **Marginal maximum a posteriori (MMAP):**
  i. What is the most likely token appearing anywhere after a token t ?
  ii. What is the most likely token appearing right after token t1 given we saw the token t1 at position p ?

Fig. 6. The possible queries for each class of query.

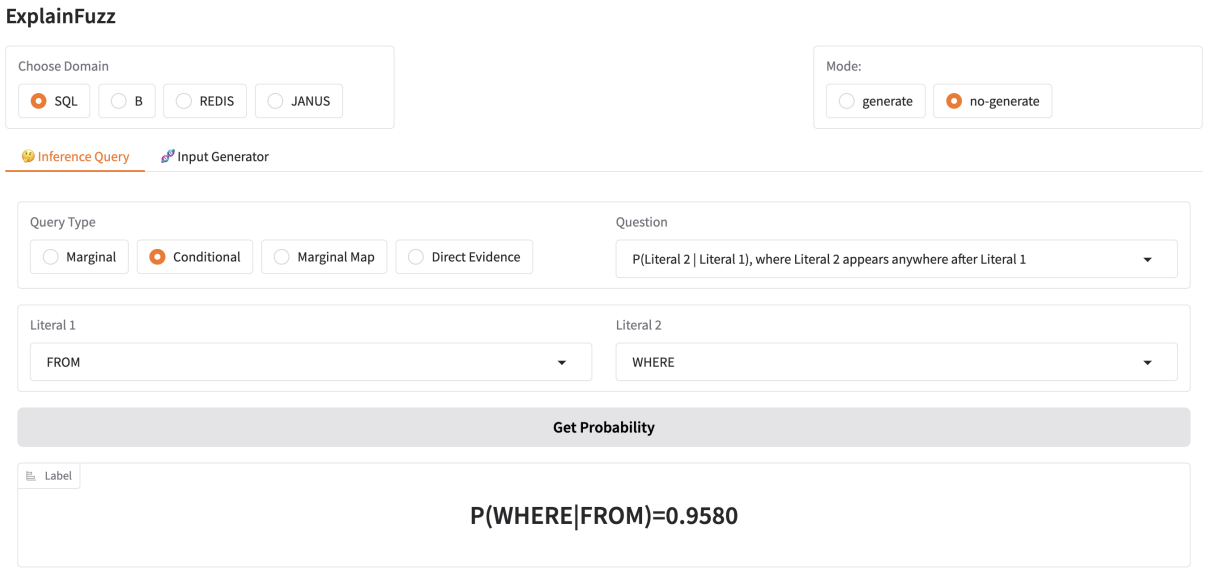

Fig. 7. User Interface (Inference query section)